Università
degli Studi
di Padova

DEPARTMENT OF
INFORMATION
ENGINEERING
UNIVERSITY OF PADOVA


**University of Padova**
**Department of Information Engineering**


Ph.D. School on Information Engineering – XXV Cycle
*Information and Communication Science and Technologies*

# Performance Optimization Of GPU ELF-Codes

**School Director**
Prof. Matteo Bertocco

**Section Coordinator**
Prof. Carlo Ferrari

**Supervisor**
Prof. Gianfranco Bilardi


**PhD Candidate**
Fausto Artico

# Abstract


**Abstract in English**

GPUs (Graphic Processing Units) are of interest for their favorable ratio $\frac{GF/s}{price}$. Compared to the beginning - early 1980's - nowadays GPU architectures are more similar to general purpose architectures but with (much) larger numbers of cores - the GF100 architecture released by NVIDIA in 2009-2010, for example, has a true hardware cache hierarchy, a unified memory address space, double precision performance and has a maximum of 512 cores.

Exploiting the computational power of GPUs for non-graphics applications - past or present - has, however, always been hard. Initially, in the early 2000's, the way to program GPUs was by using graphic libraries API's (exclusively), which made writing non-graphics codes non-trivial and tedious at best, and virtually impossible in the worst case. In 2003, the Brook compiler and runtime system was introduced, giving users the ability to generate GPU code from a high level programming language. In 2006 NVIDIA introduced CUDA (Compute Unified Device Architecture). CUDA, a parallel computing platform and programming model specifically developed by NVIDIA for its GPUs, attempts to further facilitate general purpose programming of GPUs. Code edited using CUDA is portable between different NVIDIA GPU architectures and this is one of the reasons because NVIDIA claims that the user's productivity is much higher than previous solutions, however optimizing GPU code for utmost performance remains very hard, especially for NVIDIA GPUs using the GF100 architecture - e.g., Fermi GPUs and some Tesla GPUs - because a) the real instruction set architecture (ISA) is not publicly available, b) the code of the NVIDIA compiler - nvcc - is not open and c) users can not edit code using the real assembly - ELF in NVIDIA parlance.

Compilers, while enabling immense increases in programmer productivity, by eliminating the need to code at the (tedious) assembly level, are incapable of achieving, to date, performance similar to that of an expert assembly programmer with good knowledge of the underlying architecture. In fact, it is widely accepted that high-level language programming and compiling even with a state-of-the-art compilers loose, on average, a factor of 3 in performance - and sometimes much more - over what a good assembly programmer could achieve, and that even on a conventional, simple, single-core machine. Compilers for more complex machines, such as NVIDIA GPUs, are likely to do much worse because among other things, they face (even more) complex trade-offs between often undecidable and NP-hard problems. However, because NVIDIA a) makes it virtually impossible to gain access to the actual assembly language used by its GF100 architecture, b) does not publicly explain many of the internal mechanisms implemented in its compiler - nvcc - and c) makes it virtually impossible to learn the details of its very complex GF100 architecture in sufficient detail to be able to exploit them, obtaining an estimate of the performance difference between CUDA programming and machine-level programming for NVIDIA GPUs using the GF100 architecture - let alone achieving some *a priori* performance guarantees of shortest execution time - has been,

prior to this current work, impossible.

To optimize GPU code, users have to use CUDA or PTX (Parallel Thread Execution) - a virtual instruction set architecture. The CUDA or PTX files are given in input to nvcc that produces as output fatbin files. The fatbin files are produced considering the target GPU architecture selected by the user - this is done setting a flag used by nvcc. In a fatbin file, zero or more parts of the fatbin file will be executed by the CPU - think of these parts as the C/C++ parts - while the remaining parts of the fatbin file - think of these parts as the ELF parts - will be executed by the specific model of the GPU for which the CUDA or PTX file has been compiled. The fatbin files are usually very different from the corresponding CUDA or PTX files and this lack of control can completely ruin any effort made at CUDA or PTX level to optimize the ELF part/parts of the fatbin file that will be executed by the target GPU for which the fatbin file has been compiled.

We therefore reverse engineer the real ISA used by the GF100 architecture and generate a set of editing guidelines to force nvcc to generate fatbin files with at least the minimum number of resources later necessary to modify them to get the wanted ELF algorithmic implementations - this gives control on the ELF code that is executed by any GPU using the GF100 architecture. During the process of reverse engineering we also discover all the correspondences between PTX instructions and ELF instructions - a single PTX instruction can be transformed in one or more ELF instructions - and the correspondences between PTX registers and ELF registers. Our procedure is completely repeatable for any NVIDIA Kepler GPU - we do not need to rewrite our code.

Being able to get the wanted ELF algorithmic implementations is not enough to optimize the ELF code of a fatbin file, we need in fact also to discover, understand, and quantify some not disclosed GPU behaviors that could slow down the execution of ELF code. This is necessary to understand how to execute the optimization process and while we can not report here all the results we have got, we can however say that we will explain to the reader a) how to force even distributions of the GPU thread blocks to the streaming multiprocessors, b) how we have discovered and quantified several warp scheduling phenomenons, c) how to avoid phenomenons of warp scheduling load unbalancing, that it is not possible to control, in the streaming multiprocessors, d) how we have determined, for each ELF instruction, the minimum quantity of time that it is necessary to wait before a warp scheduler can schedule again a warp - yes, the quantity of time can be different for different ELF instructions - e) how we have determined the time that it is necessary to wait before to be able to read again the data in a register previously read or written - this too can be different for different ELF instructions and different whether the data has been previously read or written - and f) how we have discovered the presence of an overhead time for the management of the warps that does not grow linearly to a liner increase of the number of residents warps in a streaming multiprocessor.

Next we explain a) the procedures of transformation that it is necessary to apply to the ELF code of a fatbin file to optimize the ELF code and so making its execution time as short as possible, b) why we need to classify the fatbin files generated from the original fatbin file during the process of optimization and how we do this using several criteria that as final result allow us to determine the positions, occupied by each one of the fatbin files generated, in a taxonomy that we have created, c) how using the position of a fatbin file in the taxonomy we determine whether the fatbin file is eligible for an empirical analysis - that we explain - a theoretical analysis or both, and d) how - if the fatbin file is eligible for a theoretical analysis - we execute the theoretical analysis that we have devised and give an *a priori* - without any previous execution of the fatbin file - shortest ELF code execution time guarantee - this if the fatbin file satisfies all the requirements of the theoretical

analysis - for the ELF code of the fatbin file that will be executed by the target GPU for which the fatbin file has been compiled.

**Abstract in Italian**

GPUs (Graphic Processing Units) sono di interesse per il loro favorevole rapporto $\frac{GF/s}{price}$. Rispetto all'inizio - primi anni 70 - oggigiorno le architectture GPU sono più simili ad architectture general purpose ma hanno un numero (molto) più grande di cores - la architecttura GF100 rilasciata da NVIDIA durante il 2009-2010, per esempio, ha una vera gerarchia di memoria cache, uno spazio unificato per l'indirizzamento in memoria, è in grado di eseguire calcoli in doppia precisione ed ha un massimo 512 core.

Sfruttare la potenza computazionale delle GPU per applicazioni non grafiche - passate o presenti - è, comunque, sempre stato difficile. Inizialmente, nei primi anni 2000, la programmazione su GPU avveniva (esclusivamente) attraverso l'uso librerie grafiche, le quali rendevano la scrittura di codici non grafici non triviale e tediosa al meglio, e virtualmente impossibile al peggio. Nel 2003, furono introdotti il compilatore e il sistema runtime Brook che diedero agli utenti l'abilità di generare codice GPU da un linguaggio di programmazione ad alto livello. Nel 2006 NVIDIA introdusse CUDA (Compute Unified Device Architecture). CUDA, un modello di programmazione e computazione parallela specificamente sviluppato da NVIDIA per le sue GPUs, tenta di facilitare ulteriormente la programmazione general purpose di GPU. Codice scritto in CUDA è portabile tra differenti architectture GPU della NVIDIA e questa è una delle ragioni perché NVIDIA afferma che la produttività degli utenti è molto più alta di precedenti soluzioni, tuttavia ottimizare codice GPU con l'obbiettivo di ottenere le massime prestazioni rimane molto difficile, specialmente per NVIDIA GPUs che usano l'architecttura GF100 - per esempio, Fermi GPUs e delle Tesla GPUs - perché a) il vero instruction set architecture (ISA) è non pubblicamente disponibile, b) il codice del compilatore NVIDIA - nvcc - è non aperto e c) gli utenti non possono scrivere codice usando il vero assembly - ELF nel gergo della NVIDIA.

I compilatori, mentre permettono un immenso incremento della produttività di un programmatore, eliminando la necessità di codificare al (tedioso) livello assembly, sono incapaci di ottenere, a questa data, prestazioni simili a quelle di un programmatore che è esperto in assembly ed ha una buona conoscenza dell'architettura sottostante. Infatti, è largamente accettato che programmazione ad alto livello e compilazione perfino con compilatori che sono considerati allo stato dell'arte perdono, in media, un fattore 3 in prestazione - e a volte molto di più - nei confronti di cosa un buon programmatore assembly potrebbe ottenere, e questo perfino su una macchina convenzionale, semplice, a singolo core. Compilatori per macchine più complesse, come le GPU NVIDIA, sono propensi a fare molto peggio perché tra le altre cose, essi devono determinare (persino più) complessi trade-offs durante la ricerca di soluzioni a problemi spesso indecidibili e NP-hard. Peraltro, perché NVIDIA a) rende virtualmente impossibile guadagnare accesso all'attuale linguaggio assembly usato dalla architettura GF100, b) non spiega pubblicamente molti dei meccanismi interni implementati nel suo compilatore - nvcc - e c) rende virtualmente impossible imparare i dettagli della molto complessa architecttura GF100 ad un sufficiente livello di dettaglio che permetta di sfruttarli, ottenere una stima delle differenze prestazionali tra programmazione in CUDA e programmazione a livello macchina per GPU NVIDIA che usano la architecttura GF100 - per non parlare dell'ottenimento *a priori* di garanzie di tempo di esecuzione più breve - è stato, prima di questo corrente lavoro, impossbile.

Per ottimizare codice GPU, gli utenti devono usare CUDA or PTX (Parallel Thread Execution) - un instruction set architecture virtuale. I file CUDA or PTX sono dati in input a nvcc che produce come output fatbin file. I fatbin file sono prodotti considerando l'architecttura GPU selezionata dall'utente - questo è fatto settando un flag usato da nvcc. In un fatbin file, zero o più parti del fatbin file saranno eseguite dalla CPU - pensa a queste parti come le parti C/C++ - mentre le rimanenti parti del fatbin file - pensa a queste parti come le parti ELF - saranno eseguite dallo specifico modello GPU per il quale i file CUDA or PTX sono stati compilati. I fatbin file sono normalmente molto differenti dai corrispodenti file CUDA o PTX e questa assenza di controllo può completamente rovinare qualsiasi sforzo fatto a livello CUDA o PTX per otimizzare la parte o le parti ELF del fatbin file che sarà eseguita / saranno eseguite dalla GPU per la quale il fatbin file è stato compilato.

Noi quindi scopriamo quale è il vero ISA usato dalla architettura GF100 e generiamo un insieme di linea guida per scrivere codice in modo tale da forzare nvcc a generare fatbin file con almeno il minimo numero di risorse successivamente necessario per modificare i fatbin file per ottenere le volute implementazioni algoritmiche in ELF - questo da controllo sul codice ELF che è eseguito da qualsiasi GPU che usa l'architettura GF100. Durante il processo di scoperata del vero ISA scopriamo anche le corrispondenze tra istruzioni PTX e istruzioni ELF - una singola istructione PTX può essere transformata in one o più istruzioni ELF - e le corrispondenze tra registri PTX e registri ELF. La nostra procedura è completamente ripetibile per ogni NVIDIA Kepler GPU - non occorre che riscrivamo il nostro codice.

Essere in grado di ottenere le volute implementazioni algoritmiche in ELF non è abbastanza per ottimizzare il codice ELF di un fatbin file, ci occorre infatti anche scoprire, comprendere e quantificare dei comportamenti GPU che non sono divulgati e che potrebbero rallentare l'esecuzione di codice ELF. Questo è necessario per comprendere come eseguire il processo di ottimizzazione e mentre noi non possiamo riportare qui tutti i risultati che abbiamo ottenuto, noi possiamo comunque dire che spiegheremo al lettore a) come forzare una distribuzione uniforme dei GPU thread blocks agli streaming multiprocessors, b) come abbiamo scoperto e quantificato diversi fenomeni riguardanti il warp scheduling, c) come evitare fenomeni di warp scheduling load unblanacing, che è non possible controllare, negli streaming multiprocessors, d) come abbiamo determinato, per ogni istruzione ELF, la minima quantità di tempo che è necessario attendere prima che un warp scheduler possa schedulare ancora un warp - si, la quantità di tempo può essere differente per differenti istruzioni ELF - e) come abbiamo determinato il tempo che è necessario attendere prima di essere in grado di leggere ancora un dato in un registro precedentemente letto o scritto - questo pure può essere differente per differnti istruzioni ELF e differente se il dato è stato precedentemente letto o scritto - e f) come abbiamo scoperto la presenza di un tempo di overhead per la gestione dei warp che non cresce linearmente ad un incremento lineare del numero di warp residenti in uno streaming multiprocessor.

Successivamente, noi spiegamo a) le procedure di trasformazione che è necessario applicare al codice ELF di un fatbin file per ottimizzare il codice ELF e così rendere il suo tempo di esecuzione il più corto possibile, b) perché occorre classificare i fatbin file generati dal fatbin file originale durante il processo di ottimizzazione e come noi facciamo questo usando diversi criteri che come risultato finale permettono a noi di determinare le posizioni, occupate da ogni fatbin file generato, in una tassonomia che noi abbiamo creato, c) come usando la posizione di un fatbin file nella tassonomia noi determiniamo se il fatbin file è qualificato per una analisi empirica - che noi spieghiamo - una analisi teorica o entrambe and d) come - supponendo il fatbin file sia qualificato per una

analisi teorica - noi eseguiamo l'analisi teorica che abbiamo ideato e diamo *a priori* - senza alcuna precedente esecuzione del fatbin file - la garanzia - questo supponendo il fatbin file soddisfi tutti i requisiti dell'analisi teorica - che l'esecuzione del codice ELF del fatbin file, quando il fatbin file sarà eseguito sulla architettura GPU per cui è stato generato, sarà la più breve possibile.

# Contents

# Chapter 1

# Structure of the Thesis

To get a very synthetic summary of the main contributions of this thesis the reader can read 15, for a more detailed description of the main contributions the reader can read 13, while to get an idea of the problems a) that afflict the papers in literature and b) that we have instead addressed and solved, the read can read 14. After this, to get a further level of detail, this time about all the contributions of the thesis, the reader can read the summary section of each chapter - to facilitate his/her research job in the thesis we describe in this chapter the structure of the thesis. Finally, for the greatest level of detail and to understand the procedures used to get each one of the results, the reader can read in detail each single chapter. The structure of the thesis is the following:

- In chapter 2 we introduce the reader to the GPU world. In 2.1 we describe the structure of 2 while in 2.2 we talk of PTX, the parallel thread execution virtual machine and instruction set architecture of the GF100 architecture - the GPU architecture used. In 2.3 we describe what is disclosed of the GPU compiler, nvcc, and its behaviors when it takes in input CUDA or PTX codes and produces as output fatbin files containing the ELF codes that the GF100 architecture has to execute. In section 2.4 we explain what happens each time a fatbin file is launched and the role of each GPU thread used to execute the fatbin file while in section 2.5 we define what is a launch configuration. In section 2.6 we instead define what are the instruction configurations, explain because they are important, explain how we will time their executions, explain the type of dependences we can consider in each instruction configuration, explain which types of dependences are important and which not and give some examples of instruction configurations;

- In chapter 3 we describe the GF100 architecture. In 3.1 we describe the structure of 3 while in 3.2 we describe the main components of the GF100 architecture. In 3.3 we analyze what is disclosed about the many parts composing some of the main components - the streaming multiprocessors - of the GF100 architecture. In section 3.4 we calculate the theoretical peak performances per second of the Tesla C2070 - the GPU that we use. In section 5 we summarize, from the quantitative point of view, the disclosed architectural features of the Tesla C2070;

- In chapter 4 we introduce several type of performances. In 4.1 we describe the structure of 4 while in each one of the remaining sections of the chapter we consider a different type of performance, give its definition, explain because it is important or not and put in evidence which of them we consider when we want to optimize the ELF code in a fatbin file;

- In chapter 5 we explain that when we calculate the efficiency of an ELF code in a fatbin file in reality we are calculating a lower bound on the real ELF code efficiency. In 5.1 we describe the structure of 5 while in 5.2 we explain more in detail why we need to calculate a lower bound on the real ELF code efficiency and underline the challenge about the quantification of the tightness of this lower bound. In 5.3 we describe the problem of the warp scheduling on the not disclosed shared hardware resources and in 5.4 we stress its influence on the ELF execution time. In 5.5 we explain that also if we eliminate the problem given by the variability, due to the warp scheduling, of the ELF code execution time, it is not yet possible to quantify the tightness of the lower bound on the real ELF code efficiency. In 5.6 we describe the warp scheduling mechanism. In 5.7 we explain that, also whether it is evident that it is not possible to quantify how much tight is the lower bound, the lower bound is always the more tight that it is possible. In 5.8 we explain that the results we get in 5.7 are valid in all the possible cases;

- In chapter 6 we reverse engineer the real instruction set architecture - and so not the PTX but the ELF - to be able to get the wanted ELF algorithmic implementations when we write code - PTX is the lowest way to write code so we need to write code in PTX, give it in input to nvcc, get as output a fatbin file and modify in the fatbin file the ELF code corresponding to the PTX code. In 6.1 we describe the structure of 6 while in 6.2 we explain the procedure necessary to localize in a fatbin file the ELF instructions corresponding to the PTX instructions of a PTX file given in input to nvcc for the generation of the fatbin file. In 6.3 we explain that to be able to modify ELF code is necessary to understand the correspondences between single PTX instructions and ELF instructions used in the fatbin files to execute the single PTX instructions and we explain how we accomplish this - using specific editing rules to edit the PTX files, checking the structures of the PTX files and their corresponding ELF codes in the fatbin files, understanding the number, type and matches between the PTX registers used in each single PTX instruction and their counterparts in the ELF code. In 6.4 we therefore build a database storing all the correspondences a) between single PTX instructions and ELF instructions necessary to execute each single PTX instruction and b) between PTX registers used in the single PTX instructions and the ELF registers used in the ELF instructions necessary to execute each single PTX instruction. In 6.5 we reverse engineer the binary codes of each single ELF instruction of interest. In 6.6 we explain how we produce fatbin files satisfying the resource constraints we need - this to be able later to modify ELF codes in fatbin files. In 6.7 we describe all the steps of the procedure necessary to generate the wanted ELF algorithmic implementations;

- In chapter 7 we discover, understand and quantify some not disclosed GPU behaviors In 7.1 we describe the structure of 7 while in 7.2 we subdivide the not disclosed GPU behavior we want to discover, understand and quantify in two categories - global and local. In 7.3 we describe the global GPU assignment and scheduling architectural features necessary to discover, understand and quantify the global GPU behaviors and the local streaming multiprocessor PTX and ELF architectural features necessary to discover, understand and quantify the local GPU behaviors. In 7.4 we explain the importance of having a priori guarantees that the PTX and the ELF codes, that we use to quantify the GPU architectural features, can not be slowed down in their executions by the bandwidths and the latencies of the GPU memories and we explain a) how we get these a priori guarantees giving to the PTX and to the ELF codes specific structures and b) how we construct such PTX and ELF codes. In 7.5 we specify the

launch configurations that we use for the quantification of the GPU architectural features distinguishing between launch configurations used for the quantification of the global GPU assignment and scheduling architectural features and the local streaming multiprocessor PTX and ELF architectural features. In 7.6 we quantify the global GPU assignment and scheduling architectural features and the local streaming multiprocessor PTX and ELF architectural features;

- In chapter 8 we explain how to transform a fatbin file to increase the probability to get a greater lower bound on its real ELF code efficiency. In 8.1 we describe the structure of 8 while in 8.2 a description of the procedures we use to modify a fatbin file. In 8.3 we describe the procedure to generate the set of launch configurations that is used when we analyze a fatbin file. In 8.4 we explain the procedure that takes in input the fatbin file that is necessary to optimize and that produces as output a) a set of fatbin files that is used to analyze the original fatbin file in input and b) a set of launch configurations for each one of the fatbin files generated;

- In chapter 9 we talk about the possible warp scheduling policy that could be executed by the warp schedulers in the streaming multiprocessors of the GF100 architecture. In 9.1 we describe the structure of 9 while in 9.2 with explain what is reasonable to assume being true about the warp scheduling policy - this considering the results of 7. In 9.3 we talk however of the impossibility of knowing whether what said in 9.2 is the truth in the real world. In 9.4 we therefore introduce the warp scheduling policy that we believe is the warp scheduling policy that the warp schedulers in the streaming multiprocessors of the GF100 architecture execute, this at least in the case when the bandwidths and the latencies of the GPU memories can not slow down the execution of a fatbin file. In 9.5 we talk instead about the possibility that other warp scheduling policies are executed by the warp schedulers in the streaming multiprocessors of the GF100 architecture and explain why, in our opinion, it is unlikely that the warp schedulers in the streaming multiprocessors of the GF100 architecture execute warp scheduling policies different from the warp scheduling cycling policy. In 9.6 we conclude the chapter describing the advantages of the warp scheduling policy and its only disadvantage;

- In chapter 10 we introduce a taxonomy for fatbin files. In 10.1 we describe the structure of 10 while in 10.2 a summary of the consequences of what said in the previous chapter about the first of the five factors necessary to classify a fatbin file, the warp scheduling policy executed by the warp schedulers in the streaming multiprocessors. In 10.3 we talk of the second of the five factors, the presence or not of branches in the ELF code of the fatbin file and of its consequences. In 10.4 we talk of the third of the five factors, the eviction policies used for the l2 and l1 caches of the GF100 architecture. In 10.5 we talk of the fourth of the five factors, the reads and writes of the GPU threads used to execute the fatbin file. In 9.4.5 we talk of the last of the five factors, the presence of ELF instructions of synchronization in the fatbin file. Finally, in 10.7, we talk of the consequences of the possible combinations, generated by these 5 factors, on the fatbin files generated, using the procedures described in 8, for the optimizations;

- In chapter 11 we list the possible combinations given by a) the position, of the fatbin files, generated using the procedures described in 8, in the taxonomy for fatbin files introduced in the 10, and b) the reader's goals, and we explain the process necessary to select the analysis

or the analyses that can be executed on the fatbin files and we describe one of the two possible analyses - the empirical one;

- In chapter 12 we explain how we guarantee a priori ELF code shortest execution times using the other of the two possible analyses - the theoretical one. In 12.1 we describe the structure of 12 while in 12.2 we describe the theoretical proof that it is necessary to prove that the execution of the ELF code of a fatbin file can not be slowed down by the bandwidths and the latencies of the GPU memories. In 12.3, supposing the execution of an ELF code of a fatbin file can not be slowed down by the bandwidths and the latencies of the GPU memories, we instead describe how to determine the minimum number of resident warps that it is necessary in each streaming multiprocessor to avoid pipeline stalls.

- In chapter 13 we summarize our contributes. At the introduction in 13.1 follows the four sections of the chapter. In section 13.2 we summarize our contributions about the reverse engineering of the real ISA and the modification of ELF code. In 13.3 we summarize our contributions about the discovery, understanding and quantification of not disclosed GPU behaviors. In 13.4 we summarize our contributions about the transformation of the ELF code of the original fatbin file to optimize and the selection of the launch configurations to use during the analysis/analyses. In 13.5 we summarize our contributions about the procedures of analysis developed to analyze the fatbin files generated for the optimization of the original fatbin file.

- In chapter 14 we describe in 14.2 the previous work and highlight in 14.3 the problems a) that afflict all the results of all the papers that we were able to find but b) do not afflict our work and the results got in this thesis.

- In chapter 15 we write the conclusions and explain the future research directions that could be followed to continue to develop the four main topics of this thesis and so a) the reverse engineering of the real ISA and the modification of ELF code to be able to have complete control on the ELF codes executed by GPUs, b) the discovery, understanding and quantification of not disclosed GPU behaviors to get data to use to understand how to optimize ELF code, c) the processes of transformation that can be applied to a fatbin file for its optimization and d) the analysis of ELF codes 1) with the development of methods of classification for ELF codes to understand the analysis/analyses that it is possible to execute on ELF codes, 2) with the development of empirical analyses and 3) with the development of theoretical analyses able to give a priori guarantees on the execution times of ELF codes.

# Chapter 2

# Introduction to GPUs

## 2.1 Introduction

In this chapter we introduce the reader to GPUs. The discussions are valid for all the GPUs using the GF100 or later architecture - see 3 for a description of the GF100 architecture and its main hardware components.

We start talking of PTX, one of the possible "tools" to edit GPU code. We use PTX because it "facilitates" the reverse engineering of the real instruction set architecture - 6.5. Understanding the real instruction set architecture is necessary to be able to modify - if necessary - the parts of the codes the GPU is going to execute. Next we talk of the NVIDIA compiler - nvcc. We describe what nvcc takes in input, compiles and what produces as output - fatbin files. We explain the many parts composing a fatbin file and later - 6.2 - we explain how we localize the several parts in each fatbin file, this to be able - if necessary - to modify the parts, containing the ELF code - 2.3 - that correspond to the PTX code that we edit.

Follow a discussion on what the GPU threads execute when we launch a fatbin file - a subset of instructions of the real instruction set architecture. The subset is in ELF and is one of the parts of the fatbin file executed by the GPU - 6.2.

We therefore explain how we need logically configure the GPU threads, that we want to execute a fatbin file, before each execution of the fatbin file. This is important because different launch configurations imply different parameters to use in the analysis process of each fatbin file and so it could be that when some launch configurations are used to execute a fatbin file, the couple ( fatbin file , launch configuration ) has a greater probability to satisfy all the requirements of the analysis process if some launch configurations are used instead of others.

Finally we define what is an instruction configuration and describe the instruction configuration features that we use in 7 to discover, understand and quantify the GPU behaviours - the GPU behaviors are used for the analysis or the analyses and the modification of the ELF codes in the fatbin files.

## 2.2 Parallel Thread Execution

GPU codes can be written in several ways, one of the possible ways is using PTX. As reported in the NVIDIA PTX manual - [52] - PTX is however much more than only one of the possible

ways to edit GPU code because PTX at its core is a parallel thread execution virtual machine and instruction set architecture (isa).

As reported in [52] the main aims of PTX are the following: 1) provide a stable isa that spans multiple GPU generations, 2) achieve performance in compiled applications comparable to native GPU performance, 3) provide a machine-independent isa for C/C++ and other compilers to target, 4) provide a code distribution isa for application and middleware developers, 5) provide a common source-level isa for optimizing code generators and translators, which map PTX to specific target machines, 6) facilitate hand-coding of libraries, performance kernels, and architecture tests, 7) provide a scalable programming model that spans GPU sizes from a single unit to many parallel units.

PTX is the lowest of the "high" level "programming languages" that we can use to edit GPU code. We use PTX to reverse engineer the real instruction set architecture of the Tesla C2070 - the GPU we use in this thesis - because using PTX to edit GPU code we can skip several phases of the compiling chain used by nvcc. Skipping several phases we get a compiled code gone under a minor number of transformation phases of all the other possible cases where any of the other available programming languages is used to edit GPU code.

We can not have the guarantee that the compiled codes achieved given in input PTX code to nvcc are more near to mirror the original PTX codes of all the other possible cases where the GPU code is written using any of the other available programming languages but PTX "facilitates" the job of understanding a) as each single PTX instruction is transformed by nvcc and b) which and how many ELF instructions in the fatbin file produced as output by nvcc are used to execute each single PTX instruction.

We always need to give in input to the NVIDIA CUDA compiler - nvcc - each one of the PTX codes we edit. In the next section we therefore describe the nvcc job when it get in input PTX code and later the nvcc job when it get in input GPU codes written not using PTX.

## 2.3 NVIDIA CUDA Compiler

We now know that one of the possible ways to edit GPU code is using PTX. PTX code can not however to be executed, in its original form, by the GPU. Before the GPU is able to execute PTX code or any other code that can be written using any NVIDIA tool or programming language it is necessary to compile the code using nvcc, the NVIDIA CUDA compiler.

The nvcc source code is not open so the things we know of nvcc are written in the NVIDIA nvcc manual. Nvcc can take in input two types of different files. Both the types of files contain code we want to be executed by the GPU but the code we want to be executed by the GPU has to be completely written using a) only PTX or b) using only one or more of the others programming languages allowed by NVIDIA. The two different type of files nvcc can take in input are the following:

- The .PTX or parallel thread execution files. The PTX files contain only GPU code and the GPU code in them can be only PTX code. When the nvcc compiler takes in input a PTX file it produces as output a fatbin - fat binary - file. The fatbin file contain the PTX code transformed in GPU assembly - let us call the GPU assembly ELF, this considering that when we use cuobjdump, 6.2, it returns as output an interpretation text file of what it defines being a fatbin ELF code.

  Our analysis of the ELF code - 6 - shows that is possible that one or more ELF instructions

corresponds to a single PTX instruction but that there are also ELF instructions that do not correspond to any PTX instruction. More, the dimension of an fatbin file is bigger of the dimension of each ELF instruction - 8 bytes - times the number of ELF instructions used to execute the PTX code - 6.2.

- The .cu or CUDA files. The CUDA files contain CPU and GPU code. The GPU code in the CUDA files can not be PTX and is written using one of the several programming languages made available by NVIDIA.

  When a CUDA file is given in input to nvcc, nvcc splits the CUDA file in one or more CPU parts and in one or more GPU parts. The GPU parts are first transformed by nvcc in PTX codes and next the PTX codes are transformed by nvcc in ELF codes - this is done considering the particular target GPU architecture where the PTX codes have to be executed. The ELF codes so obtained are one of the parts of the fatbin files generated by nvcc during the compiling phase - 6.2. After the CPU parts have been compiled using the C/C++ compiler of the CPU host machine nvcc, merges together the C/C++ compiled parts destined to be executed by the CPU and the GPU parts destined to be executed by the GPU. The final result is a fatbin file.

  The merge between the CPU parts and the GPU parts is necessary a) because when a fatbin file is executed its execution starts on the CPU side and b) because some CPU-GPU synchronizations could be necessary.

Each time a fatbin file is launched, its GPU parts are executed by GPU threads - 2.4. In the case the fatbin file is produced starting by a PTX file then the fatbin file has to be called by inside a CUDA file - the CUDA file has to be processed too to produce another fatbin file as output because the processing has always to start on the CPU side but a fatbin file produced starting by a PTX file do not have any C/C++ code.

Now we know that only some parts of each fatbin file are executed by the GPU, the GPU parts, while the other parts of a fatbin file are executed by the CPU. The GPU parts are executed by GPU threads. In the next section what each GPU thread executes of the GPU parts.

## 2.4 GPU Threads - Executions

We edit code we want executed by the GPU using PTX or one of the other available programming languages. Next we given the code in input to nvcc and we get as output a fatbin file that we later launch. When we launch a fatbin file the fatbin file starts to be executed by the CPU and later one or more of its parts are executed by the GPU.

In the parts executed by the GPU there are some subparts - 6.2 - completely composed of ELF code instructions - let us call such subparts completely composed of ELF code instructions $s_p$. The subparts $s_p$ - created by nvcc during the compiling process - correspond to the code that a) we wanted executed by the GPU and b) we wrote using PTX or one of the other available programming languages. Each time we launch a fatbin file the subparts $s_p$ are always executed by all the GPU threads we decide - at the moment of the fatbin launch - we want to execute the fatbin file. This does not mean that different threads executes different parts of the subparts $s_p$. Each GPU thread executes all the subparts $s_p$ of a fatbin file or in other words each one of the subparts $s_p$ of a fatbin file is always executed by each GPU thread.

Different fatbin files can be executed in parallel on the GPU but usually to execute a program only a fatbin file is running on the GPU at a given moment in time - this for GPU hardware synchronization problems that we face if we launch more different fatbin files in parallel.

All the GPU threads launched execute the same ELF code but the GPU threads can follow different paths - if possible - inside the same ELF code. If this happens for the GPU threads of a same warp - 3.2 - then we are in presence of a divergence phenomenon. Each divergence phenomenon implies a slow down, 10.3.

The GPU threads have however, in any case, to be logically organized before of each launch. In the next section we see what this logical organization is and the hardware limits that it has to satisfy to give a correct fatbin file execution.

## 2.5 Launch Configuration - Definition

Each time we launch a fatbin file we need to decide a) the number of GPU thread blocks, b) a two dimensional space distribution of the GPU thread blocks - logic GPU thread block distribution - c) the number of GPU threads that is the number of GPU threads of each GPU thread block - GPU thread block composition - and d) a three dimensional space distribution of the GPU threads of each GPU thread block - logic GPU thread block form - that has be the same for all the GPU thread blocks. From here, let us define the choice of these parameters a launch configuration.

We can launch a maximum of $2^{32}$ GPU thread blocks per launch, a maximum of GPU $2^{16}$ thread blocks along the x dimension of the two dimensional space and a maximum of $2^{16}$ GPU thread blocks along the y dimension of the two dimensional space. The GPU thread blocks have to be distributed starting from the origin ( 0 , 0 ) of the two dimensional space and to be contiguous along the x and y dimensions of the two dimensional space. Each GPU thread block can have a maximum of 1032 GPU threads. The GPU threads of each GPU thread block have to be distributed starting from the origin ( 0 , 0 , 0 ) of the three dimensional space and to be contiguous along the x, y and z dimensions of the three dimensional space.

Because at each fatbin file launch each GPU thread executes one or more instructions, in the next section we a) analyze how any instruction - PTX or ELF - can be executed and b) analyze the main instruction features used to discover, understand and quantify in 7 some not disclosed GPU behaviors.

## 2.6 Instruction Configuration

The GPU executes ELF instructions. At each PTX instruction corresponds one or more ELF instructions - 6.3 - and there are ELF instructions that do not correspond to any PTX instruction - 6.6.

For each PTX instruction we do not know: 1) the type and number of ELF instructions used to execute the PTX instruction, 2) the type and number of ELF registers used in the ELF instructions used to execute the PTX instruction and 3) the type and number of dependences among the ELF registers used in the ELF instructions used to execute the PTX instruction. Because these things are important for the discussions in the next chapters we introduce here the concept of instruction configuration. The discussions in the next subsections are done considering the PTX instructions but analog discussions are valid for the ELF instructions.

### 2.6.1 Definition

Any type of PTX instruction - add.s32, sub.u64, etc. - can be executed in two different modes: normal mode or conditional mode. The conditional mode can be executed if a guard is set at true or if a guard is set at false.

If the PTX instruction has some PTX registers then these PTX registers are usually used in previous PTX instructions and so they have - as result PTX register - a write-read dependence type to the last previous PTX instruction where they were written or - as the operand PTX registers - a read-read dependence type to the last previous PTX instruction where they were read.

The triplet ( type of PTX instruction , mode of execution , type of dependence considered ) is what we define being an instruction configuration.

### 2.6.2 Dependence Distance

The dependence of each PTX register has a distance of zero or more PTX instructions - zero only if the PTX register a) is read more times as operand in the same PTX instruction because it is used more times as operand PTX register in the PTX instruction or b) is read and over written in the same PTX instruction because it is used as operand PTX register and result PTX register in the PTX instruction.

### 2.6.3 Execution Time

An instruction configuration execution time is the time from the moment when, after its scheduling, the GPU hardware has to read the PTX registers used as operands in the instruction configuration to the moment when the result of the instruction configuration can be read or overwritten without waiting time caused by the writing due to the calculation of the result of the instruction configuration.

### 2.6.4 Useless Dependence Types

If in an instruction configuration the same PTX register is read more times as operand and is written as result then we have a read-write dependence at distance zero for the PTX register.

The read-write dependences at distance zero are not considered because the execution times of each instruction configuration are by definition greater than the waiting times due to read-write dependences at distance zero.

The read-write dependences at distance greater than zero are not considered because, among all the possible read-write dependences, the read-write dependences at distance zero are the dependences requiring the minimum number of clock cycles for waiting times and their waiting times are contained in the execution times of each instruction configuration. The write-write dependence are instead not considered because between any two writes of the same PTX register used for the results of some instruction configurations we read the register at least one time and so for us the write-write dependences are without importance.

### 2.6.5 Examples

In example 1 the evidence is on the sub.s32 PTX instruction. The sub.s32 PTX instruction is executed in normal mode and the type of dependences are write-read for the PTX register %result_1

Table 2.1: Example 1 of Instruction Configuration

| |
|---|
| div.s32 %result_0, %operand_0, %operand_2; |
| ..... |
| ..... |
| ..... |
| add.s32 %result_1, %operand_0, %operand_1; |
| ..... |
| ..... |
| sub.s32 %result_2, %result_1, %operand_2; |

and read-read for the PTX register %operand_2. The sub.s32 PTX instruction can be seen as two instruction configurations. The first instruction configuration considers the write-read dependence of the PTX register %result_1, this dependence has a distance in number of PTX instructions equal to 3. The second instruction configuration considers the the read-read dependence of the PTX register %operand_2, this dependence has a distance in number of PTX instructions equal to 7.

Table 2.2: Example 2 of Instruction Configuration

| |
|---|
| mul.wide.u32 %result_0, %operand_0, %operand_1; |
| ..... |
| add.su64 %result_1, %operand_1, %operand_2; |
| ..... |
| ..... |
| ..... |
| %guard_0 div.u64 %result_3, %operand_2, %result_0; |
| ..... |

In example 2 the evidence is on the div.u64 PTX instruction. The div.u64 PTX instruction is executed in conditional mode - the guard has to be set at true - and the type of dependences are read-read for the PTX register %operand_2 and write-read for the PTX register %result_0. The div.u64 PTX instruction can be seen as two instruction configurations. The first instruction configuration considers the read-read dependence of the PTX register %operand_2, this dependence has a distance in number of PTX instructions equal to 4. The second instruction configuration considers the the write-read dependence of the PTX register %result_0, this dependence has a distance in number of PTX instructions equal to 6.

Table 2.3: Example 3 of Instruction Configuration

| |
|---|
| ..... |
| @!%guard_1 popc.b32 %result_0, %operand_1; |
| ..... |
| cnot.b32 %result_1, %operand_0; |
| ..... |
| bfind.b32 %result_0, %result_1, %operand_0; |
| ..... |
| ..... |

In example 3 the evidence is on the bfind.b32 PTX instruction. The bfind.b32 PTX instruction is executed in normal mode and the type of dependences are write-read for the PTX register

%result_1 and read-read for the PTX register %operand_0. The bfind.b32 PTX instruction can be seen as two instruction configurations. The first instruction configuration considers the write-read dependence of the PTX register %result_1, this dependence has a distance in number of PTX instructions equal to 2. The second instruction configuration considers the the read-read dependence of the PTX register %operand_0, this dependence has a distance in number of PTX instructions equal to 2.

## 2.7 Summary

In this chapter we have introduced the reader to GPUs and in particular to all the GPUs using a GF100 or later architecture. The main points to remember from this chapter are the following:

- The PTX is a parallel thread execution virtual machine and instruction set architecture (isa) used to improve the portability of GPU code across several different GPU architectures. PTX is the lowest of the "high" level "programming languages" that we can use to edit GPU code we want executed by the GPU. The PTX code can not to be executed by the GPU in its original form, its has to be given in input to the NVIDIA compiler - nvcc - before of becoming GPU executable;
- Nvcc can take in input PTX or CUDA codes and always produce as output fatbin files. Inside each fatbin file there is the transformation in ELF code - the GPU assembly - of the code we want executed by the GPU. When a fatbin file is launched one or more parts of it are executed by the CPU and one or more parts of it are executed by the GPU using GPU threads. The GPU threads execute the ELF codes in the GPU parts;
- Each GPU thread used to execute a fatbin file has to execute all the ELF codes in the GPU parts of the fatbin file also whether, inside each one of the GPU parts, each GPU thread can follow different paths. The GPU threads executing a fatbin file has to be logically organized before of any fatbin file launch;
- The logical GPU thread organization has many degrees of freedom and these degrees are important because they determine the values of the some GPU behaviours used in the analysis/analyses of the ELF codes;
- Any PTX or ELF instruction can be executed in two different modes: a) normal or b) conditional - with guard set at true or with guard set at false. For each PTX or ELF instruction different types of dependences for the PTX or ELF register used in the instruction can be considered. The type of PTX or ELF instruction, its execution mode and the type of dependence considered in each single case are important to discover, understand and quantify the GPU behaviors used in the analysis/analyses of the codes and so all together they are called instruction configuration.

In the next chapter we describe the GF100 architecture, the architecture of the GPU Tesla C2070 that we use in this thesis. We start describing the main components of the GF100 architecture and next we focus on the streaming multiprocessors, the parts of the GF100 architecture necessary to execute scientific computing. Follow a theoretical analysis about the peak performances achievable by the Tesla C2070 and a paragraph of summary about the architectural features of the Tesla C2070.

# Chapter 3

# The GF100 Architecture

## 3.1 Introduction

In the previous chapter we have introduced the reader to the GPUs, in this we describe the GF100 architecture of the GPU Tesla C2070 that we use in the thesis.

The GF100 architecture is a modular architecture designed by NVIDIA and manufactured by TMC using a 40 nm productive process. The GF100 architecture has a die size of 529 $mm^2$ and a maximum of 3.2 billion of transistors.

Commercial GPUs using the GF100 architecture are the Fermi GTX 465, the Fermi GTX 470 and the Fermi GTX 480. Also whether gf means GPU Fermi two high end Tesla GPUs uses the GF100 architecture too. These Tesla GPUs are the Tesla C2050 and the Tesla C2070. What we know and explain in this chapter about the GF100 architecture is what NVIDIA discloses. We start describing the main components of the GF100 architecture and their features, later we move to describe particular parts of the GF100 architecture, the streaming multiprocessors - the streaming multiprocessors are the GPU parts where is executed the scientific computing. We analyze the main components of the streaming multiprocessors and get a first understanding of how such components interact one with the other. Next, considering the Tesla C2070 hardware limitations, we calculate its theoretical GPU peak performances and we conclude summarizing the disclosed architectural features of the Tesla C2070.

## 3.2 Main Components of the GF100 Architecture

The GF100 has off chip some private gddr 5 ram, on chip a l2 cache, a constant cache, a gigathread scheduler, 4 graphics processing cluster and a maximum of 6 memory controllers. Let see what is known about each one of these components:

- *Gddr 5 Ram:* Fermi GTX cards have 256MB attached to each of the enabled gddr5 memory controllers for a total of 1.00, 1.25 or 1.50 GB. The Tesla C2050 and C2070 have 6 controllers. The Tesla C2050 has 512 MB on each of the controllers for a total of 3 GB while the Tesla C2070 has 1024 MB on each of the controllers for a total of 6 GB. The Fermi GTX 465 has a bandwidth of 102.6 GB/s for its gddr 5 ram, the Fermi GTX 470 of 133.9 GB/s, the Fermi GTX 480 of 177.4 GB/s, the Tesla C2050 and the Tesla C2070 of 144 GB/s;

- *L2 Cache:* The l2 cache is on chip and is at maximum of 768 KB - 672 KB for GPUs like the Fermi GTX 470 and the Tesla C2070 with 14 streaming multiprocessors. The l2 cache is semi coherent because it has to keep the data present in the l1 caches but it is not necessary it keeps the data present in the shared memories and in the hardware registers.

- *Constant Cache:* The constant cache has a dimension of 64 KB and can be written only by the CPU. The GPU executes warps and each warp is always composed by 32 GPU threads. If all the 32 GPU threads of a warp read the same constant memory cell then all the accesses are satisfied in only one clock cycle and the data is broadcasted to all the 32 GPU threads. If instead the 32 GPU threads of a warp read 32 different constant memory cells, one for each GPU thread of the warp, at least 32 clock cycles are necessary to satisfy all the 32 different requests.

- *Gigathread Scheduler:* Each time a fatbin file is executed using a launch configuration - 2.5 - the gigathread scheduler has to assign the GPU thread blocks to the streaming multiprocessors and later to schedule the warps of each GPU thread block resident in a streaming multiprocessor during the whole execution of the fatbin file. The assignments and the schedulings are executed by the two gigathread scheduler levels:

  - Chip Level: The gigathread scheduler assigns the GPU thread blocks to the streaming multiprocessors. After that a GPU thread block is assigned to a streaming multiprocessor the GPU thread block can not migrate. The gigathread scheduler can manage on fly a maximum of 21504 GPU threads. The assignment of the GPU thread blocks is executed considering: 1) the hardware resources available per streaming multiprocessor, 2) the hardware resources required by each GPU thread block and 3) a series of concurrent hardware design limits which a) the maximum number of resident GPU thread blocks in a streaming multiprocessor - 8 - b) the maximum number of GPU threads a streaming multiprocessor can manage on fly - 1536 - and c) the total quantity of shared memory required by the potential set of GPU thread blocks resident in a streaming multiprocessor - this total quantity has to be smaller than 16 or 48 KB, 16 or 48 KB depends on how we set the GPU before the execution of the fatbin file, 3.3.
  - Streaming Multiprocessor Level: The gigathread scheduler in each streaming multiprocessor is represented by 2 warp schedulers. The 2 warps schedulers concurrently schedule warps on the hardware resources of the streaming multiprocessor. The 2 warp schedulers in each streaming multiprocessor can manage on fly at maximum 48 warps - 1536 GPU threads.

  The assignments and the schedulings are executed at a not disclosed clock frequency but it is reasonable to assume that the schedulings are executed at a clock frequency than is half the clock frequency of the function units - the CUDA cores, the load and store units and the special function units, 3.3 - in a streaming multiprocessor.

  This is reasonable because a) a warp is scheduled on only 1 of the 4 groups of function units in a streaming multiprocessor - or 1 of the 2 groups of CUDA cores, or the group of load and store units or the group of special function units, 3.3 - when a warp ELF instruction has to be executed for the warp b) the CUDA cores, the load and store units and the special function units have all the same clock frequency, 3.3, c) a warp is always composed by 32 GPU threads

and d) the maximum number of function units in each one of the 4 groups of function units is 16, 3.3, and therefore at least 2 function unit clock cycles are necessary to execute any warp ELF instruction for a warp.

If the clock frequency used for the schedulings is greater than half the clock frequency of the functions units of the 4 groups of function units in a streaming multiprocessor then there would be the possibility to get some queues in input to the 4 groups of function units in a streaming multiprocessor - this is however improbable considering a) the die area that the queues would require and b) the control logic that would be necessary for the management of the queues.

Furthermore, if the clock frequency used for the schedulings is smaller than half of the clock frequency of the function units of the 4 groups of function units in a streaming multiprocessor then the theoretical peak performance achievable per second would be determinate by the clock frequency of the warp schedulers and not by the clock frequency of the function units of the groups of function units and so part of the speed of the function units would be wasted.

For the previous reasons is therefore reasonable to assume that the clock frequency of the warp schedulers is exactly half of the clock frequency of the function units in each one of the 4 groups of function units in a streaming multiprocessor.

- *Graphics Processing Clusters:* Each graphics processing cluster has a raster engine and a maximum of 4 streaming multiprocessors. The Tesla C2070 have 14 streaming multiprocessors and so some graphic processing clusters have less than 4 streaming multiprocessors.

- *Raster Engines:* The main components of a raster engine are the edge setup, the rasterizer and the z-cull. The GF100 has a total of 40 Render Output Units but they are outside the streaming multiprocessors.

- *Streaming Multiprocessors:* Each streaming multiprocessor has 64 KB of private ram, $2^{15} = 32768$ hardware registers, 32 CUDA cores, 16 load and store units, 4 special function units, 2 warp schedulers, 2 instruction dispatch units, 4 texture mapping units, 1 texture cache, 1 polymorph engine, 1 interconnection network and 1 instruction cache. In the next section we describe the hardware components inside a streaming multiprocessor and how each one of the hardware components interacts with the others.

## 3.3 Main Components of a Streaming Multiprocessor

The streaming multiprocessors are the parts of the GF100 architecture where is usually executed the scientific computing. A description of each one of the main components of a streaming multiprocessor is the following:

- *L1 Cache:* We can choose only 2 configurations for the blocks of 64 KB of private ram of the streaming multiprocessors and the configuration has to be the same for all the streaming multiprocessors during the whole execution of a fatbin file. The dimension of the l1 cache and of the shared memory of each streaming multiprocessor are determined by the configuration that we choose. The configurations:

  - Configuration 1: Each one of the 64 KB of private ram is partitioned in 48 KB and 16 KB - the 48 KB are managed by the hardware of the GPU and are seen like l1 cache

while the other 16 KB has to be managed by the programmer and are seen like shared memory;

- Configuration 2: Each one of the 64 KB of private ram is partitioned in 16 KB and 48 KB - the 16 KB are managed by the hardware of the GPU and are seen like l1 cache while the other 48 KB has to be managed by the programmer and are seen like shared memory.

- *Shared Memory:* The shared memory is used to exchange data among GPU threads because the hardware registers assigned to each GPU thread are private - private hardware registers can not be used for data exchanges. The shared memory is divided in blocks of 4 bytes and works in the following way:

  - When more GPU threads want to read or to write the same shared memory blocks at the same time then each one of the shared memory blocks of 4 bytes involved in the read or the write will be serially read or written without any guarantee on the order of execution of the instructions that read or write the same shared memory block;
  - When more GPU threads want to read or to write different shared memory blocks at the same time then each one of the shared memory blocks of 4 bytes involved in the read or the write will be concurrently read or written at the same time.

  *Hardware Registers:* The hardware registers are 32 bits registers assigned to each single GPU thread. After to be assigned they became private of the GPU thread. The data in a hardware register need not to be in the l2 cache, l1 cache or in the shared memory.

  *CUDA Cores:* The CUDA cores are also called scalar processors or shader processors - this depends on the different manuals or white papers released by NVIDIA. Each CUDA core has a clock frequency of 1.15 GHz.

  Inside each CUDA core there is a dispatch port, an unit for the gathering of the operands, a floating point unit, an integer unit and a result queue. Each CUDA core can execute a fusion multiple and add per clock cycle - this is valid if the operands are at 32 bits.

  *Load and Store Units:* The load and store units are 16 and allow to load and store data from/to any memory address. The addresses are normally 64 bits addresses. The clock frequency of the load and store units is 1.15 GHz.

  *Special Functions Units:* The special function units are 4 and execute transcendental instructions as sin, cos, reciprocal and square root and have a clock frequency of 1.15 GHz.

  Each special function unit executes at maximum a transcendental instruction per GPU thread per clock cycle therefore when a warp is scheduled for the execution of a warp instruction on the group of 4 special function units is impossible the warp instruction is executed in less than 8 function unit clock cycles - every warp is composed by 32 GPU threads.

  Each special function unit pipeline is decoupled from the 2 dispatch units and so each dispatch unit can assign warp instructions to the other 3 groups of function units - the 2 group of 16 CUDA cores and the group of 16 load and store units - while the special function units are busy.

  *Warp Schedulers:* Are a part of the gigathread scheduler - streaming multiprocessor level. Each warp scheduler schedules the warps on the hardware resources of the streaming multiprocessor.

At each warp scheduler clock cycle a maximum of 2 warps are concurrently scheduled Each warp is scheduled on 1 of the 2 groups of 16 CUDA cores, or on the group of 16 load and store units or on the group of 4 special function units:

- If the warp is scheduled on 1 of the groups of 16 function units - 1 of the 2 groups of 16 CUDA cores or the group of 16 load and store units then the warp is executed as 2 half-warps - each half-warp composed by 16 GPU threads - in the next 2 function unit clock cycles;
- If the warp is assigned to the group of 4 special function units then the warp is executed as 8 eighth-warps - each eighter-warp composed by 4 GPU threads - in the next 8 function unit clock cycles.

*Instruction Dispatch Units:* The instruction dispatch units are 2, one per warp scheduler. When a warp is scheduled on the hardware resources of a streaming multiprocessor by 1 of the 2 warp schedulers an instruction dispatch unit determines the warp instruction that has to be executed for the warp. The 2 instruction dispatch units can dispatch at each warp scheduler clock cycle 2 different warp instructions.

*Texture Mapping Units:* The texture mapping units are 4. Each texture mapping unit has 4 texture filtering units. GPUs like the Tesla C2070 with 14 streaming multiprocessors have a total of 56 texture mapping units and 224 texture filtering units.

*Texture Cache:* The texture cache is of type l1 and has dimensions of 12 KB. Each texture cache is shared by the 4 texture mapping units of a streaming multiprocessor.

*Polymorph Engine:* Each polymorph engine executes the instructions of vertex fetch, tessellation and viewport transform and has an attribute setup unit and a streaming output unit.

Knowing the main hardware components of a streaming multiprocessor and how they interact among them makes it possible to calculate the theoretical GPU Tesla C2070 peak performances per second later used to determine the ELF code theoretical shortest execution time of a fatbin file - the ELF code theoretical shortest execution time of a fatbin file is useful to get an idea of the minimum quantity of time that is necessary to execute a fatbin file on the GPU.

## 3.4 Theoretical Tesla C2070 Peak Performances per Second

Our discussion is here restricted to consider the 4 group of function units - the 2 groups of 16 CUDA cores, the group of 16 store and load units and the group of 4 special function units - that are usually used to execute scientific computing - the texture mapping units and the polymorph engines are therefore not considered.

At each warp scheduler clock cycle not more than 2 warps can be scheduled per streaming multiprocessor and so not more than $2 \cdot 14 = 28$ warps can be scheduled on the whole GPU per clock cycle. Such warps can not be executed in less than 2 function unit clock cycles - a warp is composed by 32 GPU threads but each group of function units have not more than 16 function units. A single function unit with a clock frequency of 1.15 Ghz can therefore execute at maximum 1.15 G instructions per second and these 1.15 G instructions are a part of the instructions that is

necessary to execute for the maximum of $\frac{1.15 \cdot 10^9}{2} = 507.5$ M warp instructions per second that can be scheduled on the group of function units where is the function unit.

Furthermore, also whether in a clock cycle more than 32 function units per streaming multiprocessor can be executing instructions - 4.2 - in average not more than 32 function units per streaming multiprocessor per clock cycle can be executing instructions - 4.3. With not more than $14 \cdot 32 = 448$ function units executing instructions, in average, per clock cycle, the theoretical GPU peak performance of $14 \cdot 32 \cdot 1.15 \cdot 10^9 = 515.2$ GF/s is possible for instructions using 32 bit or smaller operands while, in average, per clock cycle, the the theoretical GPU peak performance of $\frac{515.2}{2} = 257.2$ GF/s is possible for instructions using 64 bit operands.

Knowing the number of ELF instructions inside a fatbin file and the launch configuration used for its launch we can calculate the total number of ELF instructions that is necessary to execute on the GPU - this number is the number of ELF instruction inside the fatbin file times the number of GPU threads of the launch configuration. The total number of ELF instructions that is necessary to execute on the GPU divided the theoretical GPU Tesla C2070 peak performance gives the minimum quantity of time that is necessary to execute the fatbin file on the GPU.

## 3.5 Summary of the Tesla C2070 Architectural Features

In this chapter we have described the main hardware components of the GF100 architecture and next we have analyzed the main hardware components of the streaming multiprocessors and the way how the hardware components of the streaming multiprocessors interact. These things are fundamental to calculate the theoretical GPU Tesla C2070 peak performances per second and so the minimum quantity of time that is necessary to execute a fatbin file on the GPU.

The Tesla C2070 has 6 GB of gddr 5 ram off chip with a bandwidth 144 GB/s, 14 streaming multiprocessors, a total of $32 \cdot 14 = 448$ CUDA cores with a clock frequency of 1.15 GHz, a total of $16 \cdot 14 = 224$ load and store units with a clock frequency of 1.15 GHz, a total of $4 \cdot 14 = 74$ special function units at 1.15 GHz with a clock frequency of 1.15 GHz, a total of $2^{15} \cdot 14 = 458752$ 32 bits hardware registers for the equivalent memory on chip of $458752 \cdot 4 = 1.8$ MB, 64 KB of constant cache on chip, 672 KB of l2 cache on chip, a total of $64 \cdot 14 = 896$ KB of ram memory on chip that can be partitioned in a total of $42 \cdot 14 = 672$ KB of l1 cache and $16 \cdot 14 = 224$ KB of shared memory or in a total of $16 \cdot 14 = 224$ KB of l1 cache and $42 \cdot 14 = 672$ KB of shared memory, a theoretical GPU peak performance of 515.2 GF/s for instructions using 32 bits or smaller operands and a theoretical GPU peak performance of 257.2 GF/s for instructions using 64 bits operands.

In the next chapter we talk about the possible types of performance we need to consider when we optimize ELF codes, we define them and we explain why, during the optimization phases, some of them are more important of others.

# Chapter 4

# Types of Performance

## 4.1 Introduction

In the previous chapter we have described the main components of the GF100 architecture and the main components of each streaming multiprocessor. Next we have understood how the main components of each streaming multiprocessor interact and we have calculated the theoretical GPU peak performances per second achievable by the Tesla C2070 GPU that we use in the thesis. Knowing the theoretical GPU peak performances per second achievable by the Tesla C2070 GPU we can calculate the minimum quantity of time that is necessary to execute the ELF code of the GPU parts of a fatbin file and so we can calculate the ELF code efficiency.

In this chapter we introduce several types of performance that we can consider during the optimization of ELF codes, from which parts of the hardware design they have origin, because it is important to differentiate one from the other and if and why each type of performance is possible in reality considering what we already know of the hardware design of the GF100 architecture - these things are important to understand a) which things of an ELF code we are going to optimize, b) why we are going to optimize them and c) how is possible to optimize them.

Our discussion considers the parts of the streaming multiprocessor that are usually used to execute scientific computing and not the parts of graphic processing used to execute particular types of graphic instructions.

## 4.2 Theoretical Streaming Multiprocessor Peak Performance Achievable in a Clock Cycle

Each streaming multiprocessor has 2 warp schedulers. If possible, at each warp scheduler clock cycle, each one of the 2 warps schedulers schedules 1 warp on 1 of the 4 groups of function units of the streaming multiprocessor. The 4 groups of functions units are composed by 2 groups of 16 CUDA cores, 1 group of 16 load and store units and 1 group of 4 special function units. Each warp when scheduled can be scheduled on only 1 of these 4 groups and can not migrate during its execution - see X. If 2 warps are scheduled at the same moment by the 2 warp schedulers then the 2 warps have to be scheduled on 2 different groups of function units - see X.

Each function unit of each one of the 4 groups of function units has a clock frequency of 1.15 Ghz. The warp schedulers schedule warps at a clock frequency that is half of the clock frequency

of the function units of the 4 groups of function units. Because each warp is composed by 32 GPU threads then at each function unit clock cycle 16 GPU threads - if the warp is executed on 1 of the groups with 16 function units - or 4 GPU threads - if the warp is executed on the only group with 4 special function units - are working per clock cycle.

Having clear the previous things the theoretical streaming multiprocessor peak performance achievable in a clock cycle is determinate with the following example. Suppose that at time 0 2 warps are scheduled, 1 on 1 of the 3 groups with 16 function units and 1 on the only group with 4 function units. Later at time 2 2 warps are scheduled, each one on 1 group of 16 function units. At time 2 the warp previously scheduled at time 0 on 1 of the 3 groups of 16 function units has completed its execution while the warp scheduled on the only group of 4 special function units has still to terminate to be executed because only 4 GPU threads per clock cycle are executed by the only group of 4 special function units. At time 2 and 3 we have therefore 2 of the 3 groups of 16 function units executing each one 16 GPU threads per clock cycle and the only group of 4 special function units executing 4 GPU threads per clock cycle - total $16 + 16 + 4 = 36$, the theoretical streaming multiprocessor peak performance achievable in a clock cycle.

However, considering the GPU hardware design, the theoretical streaming multiprocessor peak performance achievable in a clock cycle is not achievable, in average, per clock cycle, therefore we introduce the theoretical streaming multiprocessor best average performance per clock cycle used to calculate the theoretical minimum number of clock cycles that is necessary to execute an ELF code.

## 4.3 Theoretical Streaming Multiprocessor Best Average Performance per Clock Cycle

Because a) at each warp scheduler clock cycle not more than 2 warps can be scheduled, b) each warp is always composed by 32 GPU threads and c) at each function unit clock cycle at maximum only half of each one of the 2 warps is executed then not more than 32 function units, per clock cycle, can, in average, execute GPU threads. The theoretical streaming multiprocessor best average performance per clock cycle is therefore 32.

Also whether the theoretical streaming multiprocessor best average performance per clock cycle is 32 this does not mean that the theoretical streaming multiprocessor best average performance per clock cycle is reachable by every instruction configuration - 2.6. Because this is effectively the case, we explain why it happens in the next section where we introduce the theoretical instruction configuration streaming multiprocessor peak performance achievable in a clock cycle.

## 4.4 Theoretical Instruction Configuration Streaming Multiprocessor Peak Performance Achievable in a Clock Cycle

Each instruction configuration is always executed only by a type of function units - the CUDA cores, the load and store units or the special function units - and so by 1 of the 2 groups of 16 CUDA cores, by the group of load and store units or by the group of 4 special function units. Discovering the type of function units executing a particular instruction configuration we can calculate the theoretical instruction configuration streaming multiprocessor peak performance achievable in a clock cycle.

The theoretical instruction configuration streaming multiprocessor peak performance achievable in a clock cycle is equal to the number of streaming multiprocessor function units able to execute the instruction configuration and so 32 if the instruction configuration is executed by the CUDA cores - this because 2 different warps can be scheduled at the same time on the 2 different groups of 16 CUDA cores by the 2 warp schedulers and both the warps can require the execution of the same instruction configuration - 16 if the instruction configuration is executed by the load and store units or 4 if it the instruction configuration is executed by the special function units.

However, this is only a theoretical peak performance. For some instruction configurations the correspondent real instruction configuration streaming multiprocessor peak performance achievable in a clock cycle - 4.5 - is smaller than their theoretical instruction configuration streaming multiprocessor peak performance achievable in a clock cycle.

## 4.5 Real Instruction Configuration Streaming Multiprocessor Peak Performance Achievable in a Clock Cycle

One of our conjectures - verified being true in 7.6.2 - is the existence of not disclosed hardware resources shared by the 2 groups of 16 CUDA cores, hardware resources that because shared do not allow to get the theoretical instruction configuration streaming multiprocessor peak performance achievable in a clock cycle for some instruction configurations.

The real instruction configuration streaming multiprocessor peak performance achievable in a clock cycle is important because if we use the theoretical instruction configuration streaming multiprocessor peak performance achievable in a clock cycle for the calculation of the efficiency of an ELF code we could think, in some cases, that the efficiency of the ELF code is very low while instead it could be the case that the efficiency of the ELF code is very near to 1.

However, the real instruction configuration streaming multiprocessor peak performance achievable in a clock cycle can not be usually used in the calculation of the efficiency of an ELF code because usually an ELF code is composed by many different instruction configurations and so the 2 warp instruction configurations, that could be scheduled, at each warp scheduler clock cycle, for 2 different resident warps in a streaming multiprocessor, could be different.

Furthermore, because we prove the existence of not disclosed shared hardware resources between the 2 groups of 16 CUDA cores in each streaming multiprocessor, we can not exclude the existence of not disclosed shared hardware resources among any subset of the 4 groups of function units inside each streaming multiprocessor and so to take care of these things we introduce the real ELF code streaming multiprocessor average performance per clock cycle.

## 4.6 Real ELF Code Streaming Multiprocessor Average Performance per Clock Cycle

Because many different couples of instruction configurations can be scheduled at the same time by the 2 instruction dispatch units inside each streaming multiprocessor and usually an ELF code has many different instruction configurations then we have not defined a) the theoretical instruction configuration streaming multiprocessor best average performance per clock cycle and b) the real instruction configuration streaming multiprocessor best average performance per clock cycle, this because they are useless for the calculations of the efficiency of an ELF code. What instead we need

to calculate the efficiency of an ELF code is the real ELF code streaming multiprocessor average performance per clock cycle.

The real ELF code streaming multiprocessor average performance per clock cycle is determinate by - but not only - the warp scheduling. It is calculated, after the end of the execution of a fatbin file, dividing the number of ELF instructions executed by a streaming multiprocessor by the number of GPU clock cycles that have been necessary to execute the ELF code of the GPU parts of the fatbin file and it is what we want to optimize when we consider an ELF code.

One first thing that is interesting to understand it is whether an ELF code, with for example instruction configurations a) executed by the 2 groups of 16 CUDA cores and b) with a real instruction configuration streaming multiprocessor peak performance achievable in a clock cycle that is at least for some of them smaller than their theoretical instruction configuration streaming multiprocessor peak performance achievable in a clock cycle, can be executed reaching a real ELF code streaming multiprocessor average performance per clock cycle equal to the theoretical streaming multiprocessor best average performance per clock cycle and so being executed in the shortest possible time. We believed this was possible and in 7.6.2 we verify this conjecture being true.

Knowing that the conjecture is true it is important because it means a) that the not disclosed shared hardware resources between the 2 groups of 16 CUDA cores could be different at least for some couples of instruction configurations and b) that exists ELF codes executing at the theoretical best that is possible achieve, considering the GPU hardware design also, if some or all the instruction configurations of the ELF codes have singularly a real instruction configuration streaming multiprocessor peak performance achievable in a clock cycle smaller than the theoretical streaming multiprocessor best average performance per clock cycle.

With the conjecture proved true for the cases considering the 2 groups of 16 CUDA cores the same conjecture - this time not verified - can be extend at the cases considering all the possible subsets of the 4 groups of function units in each streaming multiprocessor. The possibility that the extended same conjecture, also whether not verified, is true, has to be considered during the calculations of the real ELF code streaming multiprocessor best average performance per clock cycle because we want to be sure that the real ELF code streaming multiprocessor best average performance per clock cycle we calculate is correct, precise and accurate to understand how much near the real ELF code streaming multiprocessor average performance per clock cycle is to the possible best, determinate by the GPU hardware design, for the ELF code.

## 4.7 Theoretical ELF Code Streaming Multiprocessor Best Average Performance per Clock Cycle

An ELF code can be executed with many different launch configurations and each couple ( ELF code , launch configuration ) can be executed with many different warp schedulings. The union of all these warp schedulings is the set $S$ of all the warp schedulings that could be used for the execution of the ELF code - the hardware design could not allow to the warp schedulers to choose some of the warp schedulings but for the purposes in this section this is not important.

The theoretical ELF code streaming multiprocessor best average performance per clock cycle is the greatest real ELF code streaming multiprocessor average performance per clock cycle that we would get for an ELF code if the couples of warp schedulers in each streaming multiprocessor would choose a scheduling among the subset of the bests - the warp schedulings that if used by the

warp schedulers give the greatest real ELF code streaming multiprocessor average performance per clock cycle, this independently of the fact that the GPU hardware design could not allow to the warp schedulers to choose some or all them.

If we can calculate the theoretical ELF code streaming multiprocessor best average performance per clock cycle - that could be smaller than the theoretical streaming multiprocessor best average performance per clock cycle - then we can calculate the minimum number of clock cycles necessary for the execution of an ELF code and so to calculate the theoretical best ELF code efficiency.

## 4.8 Real ELF Code Streaming Multiprocessor Best Average Performance per Clock Cycle

The real ELF code streaming multiprocessor best average performance per clock cycle is the greatest real ELF code streaming multiprocessor average performance per clock cycle that we can get for an ELF code considering the fact that the warp schedulers could be forced by the GPU hardware design to choose only by a subset $SS$ of the set $S$ considered in the previous section.

We get the real ELF code streaming multiprocessor best average performance per clock cycle if the couples of warp schedulers in each streaming multiprocessor choose one of the warp schedulings that is in the subset $SS$ and is one of the bests of the subset $SS$. The warp schedulings of this type therefore a) are in $SS$ and b) give the greatest real ELF code streaming multiprocessor average performance per clock cycle among those in $SS$ - notice that this could be smaller than the greatest real ELF code streaming multiprocessor average performance per clock cycle that we can get if the warp schedulers would choose one warp scheduling among the bests in $S$, the set of warp scheduling considered in the previous section.

If we can calculate the real ELF code streaming multiprocessor best average performance per clock cycle then we can calculate the minimum number of clock cycles necessary in reality for the execution of an ELF code and so to calculate the best real ELF code efficiency.

If we can calculate the real best ELF code efficiency then we can calculate the real ELF code efficiency of each execution of an ELF code dividing the real ELF code streaming multiprocessor average performance per clock cycle by the real ELF code streaming multiprocessor best average performance per clock cycle - the real ELF code efficiency is however usually different from execution to execution because the warp schedulers usually choose a different warp scheduling at each execution of the same ELF code, 7.6.1 and 7.6.2.

## 4.9 Summary

In this chapter we have introduced several types of performances and their definitions to make clear what we are going to optimize in an ELF code and what it is necessary to consider for the calculation of some of the performances of an ELF code to be sure that the results that we get are correct, precise and accurate. The main points to remember from this chapter are the following:

- The theoretical streaming multiprocessor peak performance achievable in a clock cycle is not reachable in each clock cycle at cause of the GPU hardware design. The best that we can get per clock cycle is therefore the theoretical streaming multiprocessor best average performance per clock cycle;

- There are some not disclosed hardware resources shared among the 2 groups of 16 CUDA cores in each streaming multiprocessor and so we can not exclude the presence of not disclosed shared hardware resources among the possible subsets of the 4 groups of function units in each streaming multiprocessor;
- For the presence of not disclosed shared hardware resources among the 2 groups of 16 CUDA cores, for some instruction configurations, executed by the 2 groups of CUDA cores, is not possible to get their theoretical instruction configuration streaming multiprocessor peak performance achievable in a clock cycle;
- ELF codes with instruction configurations with a real instruction configuration streaming multiprocessor peak performance achievable in a clock cycle smaller than their theoretical instruction configuration streaming multiprocessor peak performance achievable in a clock cycle can however get a real ELF code streaming multiprocessor average performance per clock cycle equal to the theoretical streaming multiprocessor best average performance per clock cycle;

In the next chapter we see a) how the scheduling of the warps on the 4 groups of function units in each streaming multiprocessor generates some effects that can not be foreseen and quantified a priori before the execution of a fatbin file, and b) why it is impossible to calculate the theoretical ELF code efficiency and the real best ELF code efficiency and so what we can instead do to get an idea of how near an execution of an ELF code is at the real best ELF code efficiency that the ELF code can achieve on the GF100 architecture.

# Chapter 5

# Lower Bound on the Real ELF Code Efficiency

## 5.1 Introduction

In the previous chapter we have introduced the several types of performance that we can consider when we want to analyze and optimize an ELF code while in this chapter we instead explain because it is not possible to calculate in a correct, precise and accurate way the theoretical ELF code streaming multiprocessor best average performance per clock cycle and the real ELF code streaming multiprocessor best average performance per clock cycle and so the theoretical best ELF code efficiency and the real best ELF code efficiency. Because it is not possible to calculate the real best ELF code efficiency then it is not possible to calculate the real ELF code efficiency and so what we instead calculate it is a lower bound on the real ELF code efficiency.

To understand how much tight is the lower bound, that we calculate at each execution of an ELF code, to the real ELF code efficiency, we talk of the impossibility to choose the warp schedulings on the not disclosed hardware resources shared among the 2 groups of 16 CUDA cores in each streaming multiprocessor. We therefore talk about the impossibility to determine a priori the warp scheduling influence on the real ELF code efficiency and of the impossibility to determine a priori the warp scheduling impact on the variance of the real ELF code efficiency from execution to execution of a fatbin file. Next we explain how we eliminate the ELF code execution time variance problem given by the warp scheduling from execution to execution of a fatbin file. We therefore describe the mechanism that has to be implemented in the GF100 architecture to make it possible to assign the resident warps in a streaming multiprocessor to the 4 groups of function units in the streaming multiprocessor. Having described the mechanism we therefore talk of how much tight is the lower bound, on the real ELF code efficiency, that we calculate at each execution of an ELF code, to the real ELF code efficiency of the ELF code and we conclude talking about the generality of the solution found for the lower bound.

## 5.2 Calculations to Determine the Lower Bound

When we consider an ELF code we can talk of theoretical best ELF code efficiency, real best ELF code efficiency and real ELF code efficiency. The theoretical best ELF code efficiency is the best

efficiency that the execution of an ELF code can get on the GF100 architecture if one of the best warp schedulings is chosen. The best real ELF code efficiency is the efficiency that the execution of an ELF code can get if, among the warp schedulings that the GPU hardware design allows to a warp scheduler to choose, one of the bests among them is chosen. The real ELF code efficiency is the efficiency of the execution of an ELF code and should be calculated considering the best ELF code efficiency - the ELF code efficiency can be equal to 100% only if, among the warp schedulings that the GPU hardware design allows to a warp scheduler to choose, one of the bests among them is chosen for the execution of the ELF code.

Results in 7.6.1 and 7.6.2 confirm that, also for the same couple ( fatbin file , launch configuration ), at each execution the warp scheduling is different. Considering a) that also for the same couple ( fatbin file , launch configuration ) at each execution the warp scheduling is different, b) that many launch configurations can be used to execute a fatbin file and c) the enormous quantity of warp schedulings that could be used to execute a couple ( fatbin file , launch configuration ), then we can not determine the theoretical ELF code streaming multiprocessor best average performance per clock cycle and the real ELF code streaming multiprocessor best average performance per clock cycle and so the theoretical best ELF code efficiency and the real best ELF code efficiency.

However, what we can calculate, it is the theoretical minimum number $N_t$ of clock cycles that would be necessary to all the streaming multiprocessors together to execute the ELF code supposing each streaming multiprocessor is going to get an average performance per clock cycle equal to its theoretical streaming multiprocessor best average performance per clock cycle - no performance per clock cycle can be greater than this. $N_t$ is equal to the total number of ELF instructions that is necessary to execute for the couple ( fatbin file , launch configuration ) divided by the number given by the theoretical streaming multiprocessor best average performance per clock cycle times the number of streaming multiprocessors in the GF100 architecture. Dividing $N_t$ by the number of clock cycles $N_r$ that were necessary to execute the ELF code we get a lower bound on the real ELF code efficiency.

Because a) the theoretical ELF code streaming multiprocessor best average performance per clock cycle got in the case one of the best warp schedulings is chosen can not be greater than the theoretical streaming multiprocessor best average performance per clock cycle, b) the best real ELF code streaming multiprocessor best average performance per clock cycle got in the case one of the best warp schedulings - among those that the GPU hardware design allows to the warp schedulers to choose - is chosen, can not be greater than the theoretical ELF code streaming multiprocessor best average performance per clock cycle and c) the real ELF code streaming multiprocessor average performance per clock cycle of an execution of an ELF code can not be greater than the real ELF code streaming multiprocessor best average performance per clock cycle, then the lower bound is a lower bound because also whether we can not calculate a) the minimum number $N_1$ of clock cycles necessary to execute the ELF code to get its theoretical best ELF code efficiency and b) the minimum number $N_2$ of clock cycles necessary to execute the ELF code to get its real best ELF code efficiency, the number of clock cycles necessary to execute the ELF code in these two cases can not be smaller than $N_t$ and greater than $N_r$ - $N_t < N_1 < N_2 < N_r$.

To understand what can be said about how much tight is this lower bound to the real ELF code efficiency we need first to describe the mechanics of the scheduling of the warps on the not disclose hardware resources shared among the possible subsets of the 4 groups of function units in each streaming multiprocessor and later to describe the mechanics of the more general case of the scheduling of the warps on the 4 groups of functions units in each streaming multiprocessor.

## 5.3 Warp Scheduling on the Not Disclosed Shared Hardware Resources

For each possible couple of instruction configurations a) executed by the CUDA cores and b) with a real instruction configuration streaming multiprocessor peak performance achievable in a clock cycle smaller than the theoretical streaming multiprocessor best average peak performance per clock cycle, it would be useful to determine whether some not disclosed hardware resources are shared, for the parallel execution of the couple of different instruction configurations, by the 2 groups of 16 CUDA cores. Note well that this conjecture is different from the conjecture in 4.6 because here we are interested to understand whether the same or different not disclosed hardware resources shared between the 2 groups of CUDA cores are used for the parallel execution of a couple of different instruction configurations and not whether there are some not disclosed hardware resources shared between the 2 groups of 16 CUDA cores - we effectively already know this last thing being true, see 4.5.

This is unfortunately impossible for couples composed by 2 different instruction configurations because we can not force the warp scheduling and we can not be sure which warp scheduling is chosen by the warp schedulers for the execution of an ELF code.

The warp scheduling influences the ELF code execution time because different executions of the same ELF code can give different execution times considering that the warp schedulers could schedule at the same moment, during different launch, some times 2 warps requiring the execution of instruction configurations that have to be executed by the 2 groups of 16 CUDA cores but require both the same type of not disclosed shared hardware resources - we verify this conjecture being true in 7.6.2 - and other times 2 warps requiring the execution of instruction configurations that have to be executed by the 2 groups of 16 CUDA cores but require different types of not disclosed shared hardware resources - this is the conjecture that we can not know whether it is true or not - or require the different or the same types of not disclosed or disclosed shared or not shared hardware resources without conflicts - think for example to 2 ELF instruction configurations that have to be executed by the 2 groups of 16 CUDA cores that are disclosed not shared hardware resources but that not require the use of any of the shared resources, disclosed or not disclosed, present between the 2 groups of 16 CUDA cores.

## 5.4 Warp Scheduling Influence on the ELF Code Execution Time

Not knowing or having the possibility, a priori, to choose the warp scheduling of the execution of an ELF code we can not quantify a priori the warp scheduling influence - influence due to the use of the not disclosed hardware resources shared between the 2 groups of 16 CUDA cores in each streaming multiprocessor - on the ELF code execution time and so on the real ELF code streaming multiprocessor average performance per clock cycle.

More small is the real ELF code streaming multiprocessor average performance per clock cycle compared to the theoretical streaming multiprocessor best average performance per clock cycle used in the calculation of the lower bound, less tight the lower bound is.

The challenge is therefore to understand a priori, before the execution of the ELF code, how much small will be the real ELF code streaming multiprocessor average performance per clock cycle

compared to the theoretical streaming multiprocessor best average performance per clock cycle, this to quantify how much tight is the lower bound on the real ELF code efficiency, thing however useful only in the case the lower bound is small because if instead it is near at 100%, for example 96%, then we have no incentive to quantify how much tight the lower bound is because we automatically know that we got in the reality more than the 96% of the theoretical absolute and so more than the 96% of the theoretical best and so more than the 96% of the real best - 5.2.

With what we know about the GPU hardware design this challenge does not seem solvable because as said we have no way to know before, and however also after, the ELF code execution, which warp scheduling the warp schedulers are going to choose or have chosen for the execution of the ELF code.

## 5.5 Elimination of the Warp Scheduler Variability

We can not decide the warp scheduling and the GPU behavior could be unknown and have some variabilities but because humans have designed the GPU hardware the GPU has to have a deterministic behavior and so also a set of rules determining the warp schedulings.

The fact that the GPU has a deterministic behavior means that if the ELF codes used are GPU hardware design free for all the aspects excluded the aspect generated by the warp scheduling for the use of the not disclosed shared hardware resources among the 2 groups of 16 CUDA cores in each streaming multiprocessor during the execution of an ELF code then, true that we can not decide the warp scheduling and so find the best warp scheduling to utilize for the use of the not disclosed shared hardware resources, but the warp scheduler behavior should have a very little variance from execution to execution of the same ELF code and so a regularity has to be present in the warp schedulings of different executions of the same ELF code and such regularity, also whether a priori it is not clear how it is going to influence the ELF code execution time, has to be reflected in the ELF code execution time of each execution of the ELF code and so has to give ELF code execution times with a difference between the maximum and the minimum very little.

This solve the problem of the variability of the warp schedulings but do not say anything about how, given an ELF code, we can calculate a priori how the warp scheduler behavior, also if regular, is going to influence the real average number of clock cycles necessary to execute an ELF code - and this real average number could be much greater than a) the theoretical minimum number of clock cycles necessary to execute the ELF code that instead we use in the calculation of the lower bound on the real ELF code efficiency and b) the real minimum number of clock cycles necessary to execute the ELF code that is the number of clock cycles achievable using any warp scheduling taken from the set of the best warp schedulings. Furthermore, there is no reason a priori to believe that exists a warp scheduling such that the ELF code can get a real ELF code streaming multiprocessor average performance per clock cycle equal to the theoretical streaming multiprocessor best average peak performance per clock cycle.

## 5.6 Warp Management Mechanism

During each warp scheduler clock cycle the 2 warp schedulers in a streaming multiprocessor have to have a way a) to check which resident warps in the streaming multiprocessor are available to be scheduled at the next warp scheduler clock cycle and b) to choose among them a maximum

of 2 warps - each one of the 2 warps schedulers in the streaming multiprocessor will schedule at maximum 1 warp at the beginning of the next warp scheduler clock cycle.

The resident warps in a streaming multiprocessor could be divided or not between the 2 warp schedulers in the streaming multiprocessor but 1 of the 2 warps schedulers has to choose as first a warp, if possible, and next the other warp scheduler has to choose as second another warp, if possible. The warps have always to be scheduled on 2 different groups of function units and so the 2 warp schedulers in a way or in another have to be able to communicate with each other or choose in a way interference free and so there has to be a way a) to communicate to the first warp scheduler, that has to decide which warp to schedule, at the next warp scheduler clock cycle, which groups of function units will be available and which warps will be available and can be scheduled on such groups of function units that will be available and b) later to communicate to the second warp scheduler, that has to decide which warp to schedule, at the next warp scheduler clock cycle, the remained groups of function units that will be available and which of the remained warps that was previously available to be scheduled can be scheduled on the remained groups of function units that will be available.

Somebody could think that maybe the 4 groups of function units in each streaming multiprocessor have some input queues but this is not probable because: a) the 2 warps have to be scheduled on 2 different groups of function units, b) a queue means overhead in management, data storage, etc. and c) the clock frequency of every function unit is twice the clock frequency of a warp scheduler and while 1 warp, if executed by 1 of the 2 groups of 16 CUDA cores in a streaming multiprocessor or by the group of 16 load and store units in a streaming multiprocessor, is executed in not less than 2 function unit clock cycles and so in not less than 1 warp scheduler clock cycle, 1 warp, if executed by the group of 4 special function units in a streaming multiprocessor, is execute in not less than 8 function unit clock cycles and so in not less than 4 warp scheduler clock cycles.

For the above reasons we do not believe there are queues in input to the 4 groups of function units in each streaming multiprocessor but that instead there is a way for the group of 4 special function units to signal that it is busy or will be busy for a given quantity of warp scheduler clock cycles that depends on the ELF instruction that it has to execute - this because different ELF instructions could have a different ELF instruction configuration streaming multiprocessor best average performance per clock cycle but the group of special function units has only 4 special function units and each warp has always 32 GPU threads.

In 7.6.2 we also prove that there are some ELF instructions that are executed by the 2 groups of 16 CUDA cores in a streaming multiprocessor but have a real ELF instruction configuration streaming multiprocessor best average performance per clock cycles equal to 8 and so that to execute a warp requiring the calculation of one of such ELF instructions are required 4 function unit clock cycles equivalent to 2 warp scheduler clock cycles.

Because the group of 4 special function units has to have a way to signal that it is busy or will be busy for a given quantity of warp scheduler clock cycles depending on the ELF instruction that has to be executed then considering what just said for some ELF instructions that has to be executed by 1 of the 2 groups of 16 CUDA cores it has to be true too that the 2 groups of 16 CUDA cores have to have a way to signal that they are busy or will be busy for a given quantity of warp scheduler clock cycles depending on the ELF instruction that has to be executed - no checks were done for the group of 16 load and store units but it is safe to assume that it too has to have a way to signal that it is busy or will be busy for a given quantity of warp scheduler clock cycles depending on the ELF instruction that has to be executed because we can not exclude there are not some load

and store ELF instructions with a real ELF instruction configuration streaming multiprocessor best average performance per clock cycle smaller than 16.

With the above mechanism that has to be implemented in a way or in another in the GF100 architecture the GPU hardware can determine a) which groups of function units will be available at the beginning of the next warp scheduler clock cycle or b) after how many warp scheduler clock cycles a group of function units will become available again, but this mechanism alone is not enough to make possible the scheduling of the warps on the 4 groups of function units in each streaming multiprocessor.

When 2 warp schedulers can only choose to schedule only warps requiring the use of the same not disclosed hardware resources shared between the 2 groups of 16 CUDA cores in a streaming multiprocessor, the results in 7.6.2 show a) that only 1, of the maximum 2 warps, that is possible to schedule at each warp scheduler clock cycle, is scheduled and b) that the next warp can be scheduled only after that the execution of the ELF instruction, of the previous warp scheduled, has been completed. However always the results in 7.6.2 also show that for fatbin files with a mix of ELF instructions executed by the 2 groups of 16 CUDA cores and with a real ELF instruction configuration streaming multiprocessor best average performance per clock cycle equal to 32 and 16 we get a real ELF code streaming multiprocessor best average performance per clock cycle equal to 32. Considering the previous two things is therefore not possible that, when a group of function units signals that it is busy or it will be busy for a given quantity of warp scheduler clock cycles, it makes all the other groups of function units, sharing the same not disclosed hardware resources the group of function units will use to execute the ELF instruction of the warp that has to be scheduled on it, signal that they are busy too or they will be busy too for the same given quantity of warp scheduler clock cycles, this because otherwise the previous result about the real ELF code streaming multiprocessor best average performance per clock cycle equal to 32 for fatbin files with a mix of ELF instructions would be impossible.

What therefore happens, when a group of function units signals that it is busy or it will be busy for a given quantity of warp scheduler clock cycles, it is that a check has to be done on the warps in a streaming multiprocessor and if the next ELF instruction of the warp needs for its execution the same not disclosed hardware resources used for the execution of the ELF instruction of the warp that is going to be scheduled, then independently of which groups of function units share such not disclosed hardware resources, the warp will be made not available to be scheduled a) in the next warp scheduler clock cycle - in this case the check has to be always executed between each two warp scheduler clock cycles - or b) for the given quantity of warp scheduler clock cycles that is necessary to execute the ELF instruction of the warp that is going to be scheduled at the next warp scheduler clock cycle - in this case there has to be a counter for each resident warp in a streaming multiprocessor that has to be decreased by one between each two warp scheduler clock cycles.

The above mechanism works well for all the possible cases, also for the cases where the execution of the ELF instruction of the warp does not require the use of some not disclosed hardware resources shared among the possible subsets of the 4 groups of function units in a streaming multiprocessor. To illustrate because this mechanism works in all the possible cases let us consider the following two cases among the many possible:

- Case 1): Some resident warps in a streaming multiprocessor - maybe all - are available to be scheduled by the 2 warp schedulers at the next warp scheduler clock cycle, but for the execution of the next ELF instruction of each one of the available warps, the use of the group

of 4 special function units is required. Let us also suppose the group of 4 special function units will be available to be used at the next warp scheduler clock cycle. In this case one of the warps that is available to be scheduled at the next warp scheduler clock cycles will be chosen and scheduled at the next warp scheduler clock cycle. All the previous warps that were available to be scheduled at the next warp scheduler clock cycle now becomes not available and so they can not be scheduled at the next warp scheduler clock cycle. Furthermore, all the warp, that were available, or not, to be scheduled at the next warp scheduler clock cycle, and that require for the execution of their next ELF instruction the group of 4 special function units, now can not be scheduled for the next T warp scheduler clock cycles where T is equal to ceil of 32 divided by the real ELF instruction configuration streaming multiprocessor best average performance per clock cycle of the ELF instruction of the warp that is going to be scheduled at the next warp scheduler clock cycle on the group of 4 special function units.

Case 1) does not require the use of some not disclosed hardware resources shared among the possible subsets of the 4 groups of functions units in a streaming multiprocessor and shows how it is possible that also for such case, if there are at least two resident warps in a streaming multiprocessor that are available to be scheduled at the next warp scheduler clock cycle, it could be possible that the 2 warp schedulers schedule less than 2 warps.

The fact that the warp schedulers schedule less than 2 warps at the next warp scheduler clock cycle, also whether at least 2 warps are available to be scheduled, it could be possible a) because there is not any warp scheduling, for the whole execution of the ELF code, among all the possible warp schedulings that the GPU hardware design allows to the warp schedulers to choose for the whole execution of the ELF code, able to avoid the fact that the 2 warps can not be scheduled at the next warp scheduler clock cycle or b) because the warp scheduling, chosen by the warp schedulers, till now, for the execution of the ELF code, has generated a local situation in the streaming multiprocessor such that, considering the GPU hardware design, the 2 warp schedulers are going to schedule less than 2 warps, at the next warp scheduler clock cycle, also whether in reality this case could be avoided if the 2 warp schedulers would choose a different warp scheduling for the execution of the ELF code - notice that however this does not mean that the 2 warp schedulers can choose such warp scheduling, this depends on the GPU hardware design, 4.8;

- Some resident warps in a streaming multiprocessor - maybe all - are available to be scheduled by the 2 warp schedulers at the next warp scheduler clock cycle. One of the 2 warp schedulers chooses a warp among the warps available to be scheduled at the next warp scheduler clock cycle. The ELF instruction of the warp is going to be executed by 1 of the 2 groups of 16 CUDA cores and has a real ELF instruction configuration streaming multiprocessor best average performance per clock cycle equal to 16. The other warp scheduler chooses another warp, among the remaining warps that are available to be scheduled at the next warp scheduler clock cycle on one of the remaining groups of functions units, in the streaming multiprocessor, that will be available at the next warp scheduler clock cycle - notice that more than one group of functions units could be not available at the next warp scheduler clock cycle beyond the group of 16 CUDA cores that is necessary to execute the ELF instruction of the warp that is going to be scheduled by the first warp scheduler that chose among the warps, this effectively depends on the ELF instructions of the warps scheduled by the 2 warp schedulers at the previous warp scheduler clock cycles. The warp scheduler that has still to choose find therefore

a warp available and decide that it is going to schedule the warp at the next warp scheduler clock cycle when also the other warp scheduler will schedule the warp that was chosen as first. The ELF instruction of the second warp that is going to be scheduled at the next warp scheduler clock cycle is going to be executed by the other group of 16 CUDA cores and has a real ELF instruction configuration streaming multiprocessor best average performance per clock cycle equal to 8. The fact that both the ELF instructions have a real ELF instruction configuration streaming multiprocessor best average performance per clock cycle smaller than 32, the number of CUDA cores in a streaming multiprocessor, indicated that there are some not disclosed shared hardware resources among the 2 groups of 16 CUDA cores in a streaming multiprocessor for the execution of each one of these 2 different ELF instructions but that because the 2 warps are going to be scheduled together these not disclosed hardware shared resources are different for the execution of the ELF instructions of the 2 warps. After the 2 warps have been scheduled with the same mechanism used for the Case 1) the architecture takes care to determine which resident warps in the streaming multiprocessor will not be available to be scheduled at the next warp scheduler clock cycle or to determine for how many warp scheduler clock cycle each resident warp in the streaming multiprocessor will not be available to be scheduled.

The mechanisms above described has to implemented and execute in a way or in another by the GPU architecture because the warp schedulers have to communicate between them or however choose in an interference free way a maximum of 2 warps to schedule always on 2 different groups of function units, disclosed or not disclosed hardware resources shared among any possible subset of the 4 groups of function units.

The different cases considered in this subsections shows that the scheduling of the warps - that we can not know or force - on the 4 groups of function units, independently of the not use or the use of the same or different not disclosed hardware resources shared among the possible subsets of the 4 groups of function units in a streaming multiprocessor, can slow down the execution of an ELF code and so lower the real ELF code streaming multiprocessor average performance per clock cycle making it smaller than the theoretical streaming multiprocessor best average performance per clock cycle.

## 5.7 How much Tight Is the Lower Bound?

Considering what said in the previous subsection, at each warp scheduler clock cycle the warp schedulers will schedule, if possible, 2 warps - if this happens for the whole execution of an ELF code then we will get a real ELF code streaming multiprocessor average performance equal to 32, the theoretical streaming multiprocessor best average performance per clock cycle.

The real best ELF code efficiency is the efficiency achieved by an ELF code with the use of one of the warp schedulings, of the subset $SS$, see 4.8, that is one of the best warp schedulings in the subset $SS$ - these best warp schedulings are therefore the warp schedulings a) that the GPU hardware design allows to the warp schedulers to choose for the whole execution of an ELF code and b) that give the greatest real ELF code streaming multiprocessor average performance per clock cycle.

Because we can not modify or choose the warp scheduling and so which warps to assign, at each warp scheduler clock cycle, to the 4 groups of function units in each streaming multiprocessor, we

can therefore consider the lower bound on the real ELF code efficiency the more tight possible to the real best ELF code efficiency - this is true for each execution of each ELF code.

Nothing however can in general be said about the quantification of how much tight is the lower bound to the real ELF code efficiency, this because also if the warp schedulers in a streaming multiprocessor could be able to determine the best choice in a temporal horizon of one warp scheduler clock cycle - this supposing they are able to check all the possible couples of warps that are available to be scheduled at the next warp scheduler clock cycle - there is not proof that such choice is always the best if we consider the whole temporal horizon necessary for the execution of an ELF code.

## 5.8 Generality of the Solution Found for the Lower Bound

In this chapter we started our discussion considering only the not disclosed shared hardware resources between the 2 groups of 16 CUDA cores in each streaming multiprocessor - 5.3 - but later we expand the discussion considering potential not disclosed shared hardware resources among the possible subsets of the 4 groups of function units in each streaming multiprocessor - 5.6 - and showed how the fact that we can not choose the scheduling of the warps is going to be a problem in any case, not use or use of the same or different not disclosed hardware resources shared among the possible subsets of the 4 groups of function units - 5.6.

The goal to determine whether not disclosed hardware resources shared among the possible subsets of the 4 groups of function units in each streaming multiprocessor are used for the parallel execution of all the possible instruction configuration couples and triplets - triplets because in each moment a maximum of 3 groups of function units can be executing instruction configurations, 4.2 - faces the same not solvable challenges met considering the analog goal restricted to all the possible couples of instruction configurations that were considered in 5.3 for the discussion about the only 2 groups of 16 CUDA cores in each streaming multiprocessor - this happens because we can not choose or know the warp scheduling that the warp schedulers are going to choose or have chosen for the execution of an ELF code - but the goal is not important because the problem of the warp scheduling is always present in any case and so also when for the execution of the ELF instructions of the warps is not necessary the use of some not disclosed hardware resources shared among the possible subsets of the 4 groups of function units in a streaming multiprocessor - 5.6.

The lower bound we calculate on the real ELF code efficiency at each ELF code execution is therefore always the more thigh possible, this independently of the warp schedulings that the GPU hardware design allows to the warp schedulers to choose for the executions of an ELF code - remember, 1) usually the warp scheduling chosen by the warp schedulers will be different from execution to execution of the same ELF code in a fatbin file, this also in the case we use the same launch configuration, 7.6.1 and 7.6.2, and 2) the GPU hardware design could allow to the warp schedulers yes to choose one warp scheduling among the many possible but not among all the possible, 4.8.

## 5.9 Summary

In this chapter we have explained why we need to calculate a lower bound on the real ELF code efficiency at each execution of the ELF code and talked of how much tight is this lower bound to the real ELF code efficiency. The main points to remember from this chapter are the following:

- Also for the same couple ( fatbin file , launch configuration ) the warps schedulers in each streaming multiprocessor usually choose a different warp scheduling at each execution and the GPU hardware design allows to them to choose only warps in a subset of all the possible. Considering that it is not possible to determine the set of warp schedulings from which the GPU hardware design allows to the warp schedulers to choose and that it is not possible to determine which warp scheduling of this set the warp schedulers in each streaming multiprocessor will use at the next execution of the couple ( fatbin file , launch configuration ) then it is not possible to calculate a) the theoretical best ELF code efficiency and b) the real best ELF code efficiency, of the ELF code, in a fatbin file;

- Because it is not possible to calculate the theoretical best ELF code efficiency and the real best ELF code efficiency then what we calculate, at each execution of a couple ( fatbin file , launch configuration ), is a lower bound on the real ELF code efficiency - this is possible thanks to the use of the theoretical streaming multiprocessor best average performance per clock cycle;

- Because to calculate the lower bound on the real ELF code efficiency we use the theoretical streaming multiprocessor best average performance per clock cycle - notice that the real ELF code streaming multiprocessor best average performance per clock cycle can never be greater than the theoretical streaming multiprocessor best average performance per clock cycle - then, if the lower bound is around 90% or more, we do not really care about how much tight it is because we automatically know that we are already near to fully utilize the GF100 architecture at its absolute best;

- Because we can not know, choose or force the warp scheduling of the warps on the 4 groups of function units in each streaming multiprocessor, with what we know about the GF100 architecture, the lower bound, that we calculate on the real ELF code efficiency, can be more or less tight to the real ELF code efficiency but it is always the more tight possible;

In the next chapter we reverse engineer the real instruction set architecture a) to be able to get the wanted ELF algorithmic implementations because usually PTX codes are transformed in ELF codes that do not mirror the original PTX codes and b) to be able to modify fatbin files, if we want so, to optimize them and so increase their lower bounds on the efficiencies of their ELF codes.

# Chapter 6

# Reverse Engineering of the ISA and Modification of ELF Codes

## 6.1 Introduction

The lowest of the "high" level "programming languages" available to users to write GPU code is PTX but PTX is only a virtual instruction set architecture. Developing theories considering PTX codes, optimizing PTX codes and analyzing PTX codes is meaningless because the GPU architecture executes ELF code, not PTX code, and the ELF code produced by nvcc taking in input PTX code is usually very different compared to the PTX code a) for number, order and type of instructions - ELF instructions instead of PTX instructions - and b) for number, type and reuse of registers - ELF registers instead of PTX registers.

Also if we base our analyses on ELF codes, because the ELF codes are usually very different compared to the PTX codes given in input to nvcc, we need to be able to modify ELF codes to be able to optimize their executions, but because the real instruction set architecture and other features of the ELF codes - corresponding to the PTX codes - in the fatbin files, are not disclosed, it is impossible to modify the ELF codes without before to execute several reverse engineering procedures to uncover the necessary not disclosed information.

The goal of this chapter is therefore 1) to describe the reverse engineering procedures necessary to uncover the not disclosed information - a) on the real instruction set architecture and b) on the features of the ELF codes - necessary to modify the ELF codes, corresponding to PTX codes, in the fatbin files, and 2) describe the procedure that, using the results got about the not disclosed information, allows to modify the ELF code of a fatbin file and so to get any wanted ELF algorithmic implementation.

We start localizing in a fatbin file the positions of the ELF instructions corresponding to the PTX instructions of a PTX code. This is not easy because not knowing the real instruction set architecture we don't know the types of ELF instructions and their binaries and so we need first to understand, in a fatbin file, which are the binary codes of the ELF instructions corresponding to the PTX instructions of the PTX code and later to search them in the fatbin file using a robust procedure giving the guarantee that the positions that we find are really the positions of the ELF instructions necessary to execute the PTX instructions of the PTX code and not the positions of some other ELF instructions equal to the ELF instructions necessary to execute the

PTX instructions of the PTX code.

The PTX-ELF correspondences are therefore determined. The PTX-ELF correspondences imply a) the understanding of the number, order and type of ELF instructions necessary to execute each single PTX instruction, b) the understanding of the number, order and type of ELF registers used in each ELF instruction, c) the understanding of which ELF registers in the ELF instructions correspond to which PTX registers in the single PTX instructions and d) the understanding of the presence or not of ELF registers in the ELF instructions without corresponding PTX registers in the single PTX instructions.

A database of the ELF-PTX correspondences is built. This database stores the results discovered for the PTX-ELF correspondences. In this database the human readable text form representations of the types of ELF instructions necessary to execute each single type of PTX instruction are associated to each human readable text form representation of each single type of PTX instruction, this together at the results got at the points a), b), c) and d) of the previous paragraph for the PTX-ELF correspondences.

A database of the binary codes, of each possible human readable text form representation of each possible ELF instruction of interest, is generated. This database requires the reverse engineering of all the binary codes of all the possible human readable text form representations of the ELF instructions of interest and so of understanding which bits in each binary code correspond to which ELF registers used in each ELF instruction of interest and which bits instead correspond to other things like for example op codes, flags and other things not visible in the human readable text form representation of an ELF instruction.

Because in the given time frame it is not possible to determine a) the ELF instructions, if any, necessary to assign ELF registers to a fatbin file, b) the procedure to assign to each ELF register a different hardware register for each GPU thread used for the execution of a fatbin file and c) whether there are other things not evident from the analysis of the interpretation text file of a fatbin file - 6.2 - we create a procedure to generate fatbin files with at least the minimum number of resources later necessary to modify the fatbin files to get the wanted ELF algorithmic implementations.

We therefore describe another procedure that, taking in input the fatbin files with at least the minimum number of resources necessary for their modification, allows us to modify the fatbin files to get the wanted ELF algorithmic implementations and later further modifying them, if necessary, all the times and in all the ways we need or want.

## 6.2 Localization in Fatbin Files of the ELF Instructions Necessary to Execute the PTX Instructions of PTX Codes

Also whether a) the real instruction set architecture of the GF100 architecture is not disclosed and b) we do not know the binary codes corresponding to the ELF instructions used by the GF100 architecture to execute the PTX instructions of a PTX code, we can however interpret every fatbin file using a NVIDIA's tool - cuobjdump.

Cuobjdump gets in input a fabin file and gives as output an interpretation text file where we can see how the PTX code has been transformed in ELF code. Every line of the interpretation text file has three columns. In the first column there is a series of hexadecimal consecutive addresses with the address in the first row always equal to 0x0000, in the second column there are the binary representations of the ELF instructions at the addresses in the first column and in the third column

there are the human readable text form representations of the binary codes of the ELF instructions in the second column. The dimension of each ELF instruction is 8 bytes because in the second column of each interpretation text file there are always 16 hexadecimal digits.

We experimentally found that the dimension of each fatbin file is always bigger than the dimension in bytes of every ELF instruction - 8 - times the number of ELF instructions visible in the interpretation text file. To write/modify ELF codes or to extract the ELF instructions necessary to execute a PTX instruction is therefore necessary as first step to code a robust procedure able a) to localize in the fatbin files the correct position of each single ELF instruction visible in the interpretation text files and b) to understand whether the binary codes of the ELF instructions, showed in hexadecimal form, in the second column of the interpretation text files, are the real binary codes or permutations of the real binary codes of the ELF instructions.

We formulate two conjectures: 1) the binary codes of the ELF instructions showed in hexadecimal form in the second column of the interpretation text files are byte permutations of the real binary codes of the ELF instructions and 2) the ELF instructions visible in the interpretation text files are always ELF instructions that are consecutive in the fatbin files. To verify this two conjectures each time we analyze a fatbin file we extract the potential binary codes of the ELF instructions from the interpretation text file of the fatbin file. Next, for each one of the possible 8! permutations of the 8 bytes of each binary code of each ELF instruction, we execute the following procedure: 1) transformation of all the potential binary codes of the ELF instructions in the interpretation text file considering the chosen permutation, 2) alignment of the block of consecutive permuted binary codes of the ELF instructions visible in the interpretation fatbin file to every possible byte in the fatbin file, 3) calculation of the similarity score of every alignment, similarity score equal to the number of groups of 8 consecutive bytes, and so to the number of ELF instructions visible in the interpretation fatbin file, with a perfect match with the bytes in the fatbin file, given the chosen permutation. After the execution of the procedure for all the possible 8! permutations we 1) determine the maximum similarity score among all the similarity scores calculated, 3) check that the maximum similarity score is equal to the number of ELF instructions visible in the interpretation text file of the fatbin file and 3) check that the maximum similarity score appears only one time.

Thanks at the previous procedure we verified conjecture 1) and conjecture 2) being true. Conjecture 1) is true because the bytes of an ELF instruction in position 0, 1, 2, 3, 4, 5, 6, 7 on the hard disk are interpreted by cuobjdump as bytes in position 5, 6, 7, 8, 0, 1, 2, 3 for the hexadecimal representations of the binary codes of the ELF instructions visible in the interpretation text files of the fatbin files. Conjecture 2) is true because a) the maximum similarity score is always equal to the number of visible ELF instructions in the interpretation text file of a fatbin file and b) because it always appears only one time for each fatbin file.

Having verified a) conjecture 1) and 2) and b) storing the position in the fatbin file where the alignment of the block of consecutive real binary codes of the ELF instructions visible in the interpretation fatbin file gives us a perfect match, allows, after the analysis of the structure of the PTX and the fatbin files in 6.3.2 - the analysis is necessary to verify that, thanks at the editing guidelines used in 6.3.1 to write the PTX files, nvcc was forced to use the specific wanted transformation rules - to extract the number, order and type of ELF instructions necessary to execute a PTX instruction of interest - 6.3.3 - with the guarantee that the extracted ELF instructions are really the ELF instructions necessary to execute the PTX instruction.

## 6.3 PTX - ELF Correspondence Transformations

We can not exclude that each PTX instruction in a PTX code can be transformed by nvcc in more ELF instructions necessary for the execution of the PTX instruction. To be able to write or modify ELF codes is therefore necessary to understand the PTX - ELF correspondence transformations used by nvcc for each single PTX instruction.

### 6.3.1 Editing Guidelines To Edit PTX Files

Nvcc usually produces ELF codes very different from the input PTX codes but the nvcc code is not open and so it is hard to understand and to determine the nvcc transformation rules. Also supposing this is feasible, the whole procedure would be however very time and energy consuming and so not feasible in the given time frame.

A better choice is forcing nvcc to use only specific transformation rules. If we force nvcc to use only specific transformation rules and we verify each time, during the generation of a fatbin, that nvcc is really using only the specific transformations rules we want then 1) we can extract the number, order and type of ELF instructions used to execute each PTX instruction of interest and 2) we can be sure the number, order and type of ELF instructions used to execute each PTX instruction of interest are really corresponding to each single PTX instruction of interest instead of some other PTX instructions in the PTX files.

To force nvcc to use specific transformation rules we use a set of editing guidelines to edit a PTX file for each type of PTX instruction of interest. The editing guidelines are based on the assumptions that also whether the nvcc code is not open it is however reasonable that: a) nvcc is going to cut dead code, b) nvcc is going to save the greatest number of ELF registers it can save, c) nvcc is not going to remove two PTX synchronization barriers if between the two PTX synchronization barriers there is at least one useful PTX instruction, d) if there is only one PTX instruction between two PTX synchronization barriers and the PTX instruction is useful then nvcc is going to transform the PTX instruction between the two PTX synchronization barriers in one or more correspondent ELF instructions that nvcc put between the two ELF synchronization barriers corresponding to the two PTX synchronization barriers and e) the order between different couples of PTX synchronization barriers in the PTX file is preserved in the corresponding ELF code generated by nvcc.

Considering the previous assumptions here the editing guidelines: a) every single PTX instruction has to be written between two PTX synchronization barriers, b) for each not predicate PTX register used, a data is loaded in it, c) each data is loaded from a different GPU global memory address d) each data is loaded before the PTX instruction of interest, e) just after each data load, the not predicate PTX register, where is the data, is used as operand and as result of a PTX instruction, with the goal to modify the data, f) for each predicate PTX register used, a PTX instruction of setting, using as operands the not predicate PTX registers, is executed before the PTX instruction of interest, g) after the PTX instruction of interest, all the data, in all the PTX registers, are stored to different GPU global memory addresses, different among them and different from the GPU global memory addresses used to load the data in the not predicate PTX registers before the PTX instruction of interest.

The first motivation for the editing guidelines is that using them, if nvcc is forced to use the wanted transformation rules, then we can later extract from the interpretation text file of each fatbin file the number, order and type of ELF instructions necessary to execute each single PTX

instruction of interest - 6.3.3 - thing possible because:

- We know the order, number and type of each one of the possible 16 different PTX synchronization barriers we write in each PTX file;
- We divide each PTX file in different sections and in each section we use only one type of PTX synchronization barrier and the type of PTX synchronization barrier used in each one of the different sections is different - this implies that each PTX file can not have more than 16 different sections but this number is big enough for all our goals;
- We have manually checked that at each type of PTX synchronization barrier correspond only a single ELF instruction - this was done in 6.2, during the analysis of the interpretation text files of the fatbin files, generating 16 PTX files with only one type of PTX synchronization barrier per PTX file, type of PTX synchronization barrier written a number X of times, with us discovering a) that the number of PTX synchronization barriers in each PTX file was equal to the number of ELF instructions in each interpretation text file of the fatbin files generated and b) that all the ELF instructions in each fatbin file were always equal among them but different among different fatbin files;
- We have got the human readable text form representation of each one of the 16 single ELF synchronization barrier instructions - this was done always in 6.2 where we found that the human readable text form representations of each one of the possible 16 ELF synchronization barrier instructions corresponding to the 16 PTX synchronization barrier instructions is unique because uses a set of special registers and a set of constant values.

The second motivation for the editing guidelines is that using them, if nvcc is forced to use the wanted transformation rules, then we can later understand, analyzing the interpretation text files of the fatbin files, to which PTX register in a PTX instruction each ELF register used in an ELF instruction correspond, this because 1) we load a data in each not predicate PTX registers used, 2) we load the data from different GPU global memory addresses, 3) we load the data before of the PTX instruction of interest, 4) we modify each data, using only the PTX register where the data is, just after the data load, 5) we set, before of the PTX instruction of interest, the predicate PTX registers, this using as operands of the setting PTX instruction the not predicate PTX registers, 6) we store the data in each PTX register used in the PTX file, after the PTX instruction of interest, to different GPU global memory addresses - different among them and different from the GPU global memory addresses used to load the data in the not predicate PTX registers before the PTX instruction of interest - and so nvcc is forced to avoid to try to save ELF registers making clear, in the storing section, of an interpretation text file, of a fatbin file, that corresponds to the storing section of the PTX file, given in input to nvcc, to generate the fatbin file, which ELF registers correspond to which PTX registers for the ELF instruction of interest - this because of course we also know in which order we use the PTX registers of the PTX file in the storing section of the PTX file.

Also whether the editing guidelines are based on the assumptions a), b) c), d), e), f) and g), because we can not be sure the assumptions are true in the reality for every possible case - the nvcc code is not open - then it is necessary 1) to implement some automatic controls - 6.3.2 - to check that every time a fatbin file is generated by nvcc, the fatbin file is generated using the wanted transformation rules and 2) if this is not always the case, to discover and understand in which cases this does not happen, taking care to execute the consequent necessary actions - 6.3.2.

### 6.3.2 Analysis and Comparison of the PTX and Fatbin File Structures

To be sure that the editing guidelines, used to edit the PTX files, have forced nvcc, during the generation of the correspondent fatbin files, to use the transformation rules we want, we need to extract, analyze and compare the structures of each couple ( PTX file, ELF part corresponding to the PTX file ).

The structure of a PTX file is given by the order and type of PTX instructions in the PTX file while the structure, of the ELF part, corresponding to the PTX file, is given by the order and type of ELF instructions visible in the interpretation text file of the fatbin file.

To following checks and countermeasures, in the case nvcc has not used the transformation rules we wanted, also whether we used the editing guidelines to edit the PTX files, are therefore necessary to determine the reliability of each ELF part corresponding to each PTX file:

- a) Check on the number of each type of ELF synchronization barrier in the ELF part corresponding to the PTX file, this because we know how many PTX synchronization barriers we wrote for each type of PTX synchronization barrier in the PTX file but we want to be sure that nvcc, during the transformation of the PTX files in fatbin files, hasn't modified the number of synchronization barriers used for each type.

  If the number of also only one type of ELF synchronization barrier is not equal to the number of the correspondent type of PTX synchronization barrier then we can not trust the ELF part corresponding to the PTX file and so we need to discard the PTX and the fatbin file produced;

- b) Check that each type of ELF synchronization barrier has not other types of ELF synchronization barriers between its first and last exemplar. Because a) between each two PTX synchronization barriers we wrote a PTX instruction, b) we divided each PTX file in different sections using in each section only one type of PTX synchronization barrier and c) the type of PTX synchronization barrier used in each one of the different sections was different, if in the ELF part corresponding to the PTX file in a fatbin file we have one or more different types of ELF synchronization barriers between the first and last exemplar of any type of ELF synchronization barrier, then we automatically know that nvcc did not use the transformation rules we wanted, this also whether we edit the PTX files following the editing guidelines, and so in this case too we can not trust the ELF part corresponding to the PTX file in the fatbin file and so we need to discard the PTX and the fatbin file produced.

If the previous two checks are positive then we are sure that, thanks at the editing guidelines used to edit the PTX files, nvcc used the transformation rules we wanted and so, because we know a) between which type or types of PTX synchronization barriers we wrote the PTX instruction of interest and b) the index number of each one of the two PTX synchronization barriers - for example the fourth of type 2 and the first of type 3 - we are able 1) to localize in the interpretation text file generated by cuobjdump the number, order and type of ELF instructions corresponding to the PTX instruction of interest and 2) to check that among such ELF instructions there are not ELF instructions jumping to ELF instructions before or after the two ELF synchronization barriers delimiting the group - the ELF synchronization barrier corresponding to the fourth PTX synchronization barrier of type 2 and the ELF synchronization barrier corresponding to the first PTX synchronization barrier of type 3 - thing this necessary to be definitely sure that the ELF

instructions between the two ELF synchronization barriers are all the ELF instructions necessary to execute the PTX instruction.

### 6.3.3 Number, Type and Matching among PTX and ELF Registers

Being able to know the number and type of ELF instructions corresponding to a PTX instruction is not enough. For each ELF instruction used to execute a PTX instruction we need 1) to understand which ELF register correspond to which PTX register in the PTX instruction, 2) to check that the ELF registers used in the ELF instruction correspond to PTX registers used in the PTX instruction and not to some other PTX registers used in the PTX file and 3) to check whether the ELF instruction is using some ELF registers without any correspondent PTX register in the PTX file.

Knowing a) in which order we use the PTX registers of a PTX file to store the data resident in all the PTX registers used in the PTX file, b) the section of the PTX file where the storing procedure is executed - after the PTX instruction of interest - c) between which types of PTX synchronization barriers and index numbers each PTX register with a data to store is used, d) that the checks of the previous section on the comparison of the structure of the PTX file and the ELF part corresponding to the PTX file are satisfied and d) the position, in the human readable text form representations of the storing ELF instructions, of the ELF registers containing the data to store, then we can 1) check in the ELF code section corresponding to the storing section of the PTX file that the number of ELF registers, containing the data to store and used in the storing ELF instructions, is equal to the number of PTX registers used in the PTX file and 2) understand, for each ELF register, in the ELF code section, corresponding to the storing section in the PTX file, which is the corresponding PTX register in the PTX file.

We can affirm 2) because we use the editing rules to edit the PTX files and so it is very unlikely that in some parts of the ELF code an ELF register corresponds to a PTX register and in other parts of the ELF code the ELF register corresponds to another PTX register, this because a) we load, all the necessary data, from different parts of the GPU global memory, in the not predicate PTX registers, b) we modify such data and set the predicate PTX registers, before the PTX instruction of interest, c) after the PTX instruction of interest, we store the data, in all the PTX registers, to other different parts of the GPU global memory and d) it would be stupid whether the compiler would use more ELF instructions of the necessary to swap data between ELF registers for the execution of the fatbin file.

Knowing the matches, we can understand the position of each result and each operand of a PTX instruction in the human readable text form representations of the ELF instructions necessary to execute the PTX instruction. Manual checks also show that nvcc is always assigning two consecutive ELF registers to each 64 bits PTX register, this also whether, in the human readable text form representation of some of the ELF instructions, necessary to execute the PTX instruction, only one of the two ELF registers could be present.

Saving a) the name/s, type/s and position/s of the PTX register/s in the PTX instruction, b) the name/s, type/s and position/s of the ELF registers in the human readable text form representations of the ELF instructions necessary to execute the PTX instruction and c) the ELF registers correspondences, if any, to the PTX registers in the PTX instruction, allows us later, with different PTX-ELF register correspondences, to generate the right human readable text form representations of the ELF instructions necessary to execute each PTX instruction we want - 6.7 - and to search

in the database of the binary codes of the ELF instructions corresponding to the human readable text form representations - 6.5 - the binary codes of the ELF instructions, binary codes that we are going to use to modify the ELF parts corresponding to the PTX files, this to get the wanted ELF algorithmic implementations - 6.7.

However, before to save the previous data, two further checks are executed. The first check is executed to be sure that all the ELF registers, corresponding to the PTX registers used in the PTX instruction, appear in the ELF instructions necessary to execute the PTX instruction. The second check is executed to understand whether there are some ELF registers used in the ELF instructions necessary to execute the PTX instruction without any correspondent PTX register in the PTX instruction or in the PTX file. If one of the checks would be negative then something would be wrong but this never happens. If both the checks are instead positive, as it is always the case, then a) the name/s, type/ and position/s of the ELF register/s 1) used in the ELF instructions necessary to execute the PTX instruction and 2) without correspondent PTX register in the PTX instruction or in the PTX file, are stored - such data are useful when it is necessary to modify the ELF codes to get the wanted ELF algorithmic implementations, 6.7 - and b) we can extract the ELF instructions, necessary to execute the PTX instruction, being sure 1) that the ELF instructions correspond to the PTX instruction, 2) that the ELF registers that are used in the ELF instructions, necessary to execute the PTX instruction, have zero or more matches with the PTX registers used in the PTX instruction and 3) that such matches between ELF and PTX registers are correct and therefore can be used, when it is necessary, to modify an ELF code, to get the wanted ELF algorithmic implementations - 6.7.

## 6.4 Database of the Human Readable Text Form Representations

For each PTX instruction is important to store in a database 1) the number, order and type of human readable text form representations of the ELF instructions necessary to execute the PTX instruction, 2) the number, the type and the positions of the ELF registers used in the ELF instructions necessary to execute the PTX instruction and 3) the correspondences among ELF registers and PTX registers, if any. The previous three things are necessary because:

- a) When we modify the ELF codes, to get the wanted ELF algorithmic implementations, we need to understand how many ELF registers and types of ELF registers are used by the ELF instructions necessary to execute each PTX instruction we want the ELF code executes.

  Before and after our modifications, a fatbin file has to have at least a minimum number and type of ELF registers for its correct execution in its original and modified final form. If after its modification/s, the fatbin file uses ELF registers not originally to it assigned by nvcc during its creation, then we are always going to get a launch failure when we execute the fatbin file, this for violations due to the use of the hardware resources - hardware registers - to it not originally allocated - this has been experimentally proved by us thanks at the procedure that we describe in 6.6, procedure that we use to modify ELF codes.

  Knowing a) the number and type of PTX instructions we want executed by a fatbin file, b) the number, order and type of ELF instructions necessary for the execution of each PTX instruction, c) the number, the type and the position/s of each ELF register in each ELF

instruction, e) what is representing each ELF register in each ELF instruction, d) which ELF register corresponds to which PTX register, if any, e) the type of reuse we want for each ELF register - thing determining the type of dependence of each ELF register at each its reuse and the dependence distance between each two its consecutive uses in the ELF code - and e) having experimentally determined that a maximum of 64 ELF registers can be assigned to a fatbin file but that 4 of them are reserved for special uses in some ELF instructions and therefore it is safe to assume that they can not be substituted with one of the other remaining possible 60 ELF registers, then, because we can not write from scratch a fatbin file - see why in section 6.6 - but we can only modify fatbin files produced by nvcc, we can analyze a fatbin file, before to modify it, to understand whether it has at least the minimum number of ELF registers, per type of ELF register, necessary for its modification and if not, we execute a procedure, of creation and destruction of the fatbin file, to get this goal - section 6.6.

- b) Studying the human readable text form representations of the ELF instructions we found that 1) the ELF registers, used for the result and the operands of an ELF instruction, can be in different parts of the human readable text form interpretations of the ELF instructions and 2) in a order different from the order of the correspondent PTX registers in the PTX instruction - these two things are true too for the binary codes of the ELF instructions discovered using the procedure in 6.5.

  Knowing 1) which ELF register corresponds to which PTX register, 2) which ELF register is the register where will be written the result, which ELF registers are the operands of an ELF instruction and the order of the ELF registers compared to the order of the corresponding PTX registers in the PTX instruction, 3) whether in the ELF instructions necessary to execute a PTX instruction are used some ELF registers without correspondent PTX register in the PTX instruction and 4) whether an ELF register, used in the group of ELF instructions necessary to execute a PTX instruction, has to be present more times, in different positions, in the group of ELF instructions necessary to execute a PTX instruction, allows us a) to be sure that the execution of the fatbin file, after its modification, is logically correct, because we use the right types of ELF registers in each ELF instruction, in their right roles - result, operands, etc. - and read/write the data from/to the right ELF registers - role dependences in the single and in the group of ELF instructions necessary to execute each single PTX instruction - b) to modify the ELF code in such way to get the wanted dependence distances, in number of ELF instructions, between each couple of consecutive uses of each ELF register and c) to determine where and how many times each ELF register, assigned by nvcc to a fatbin file, can be used, during the fatbin file execution, without making the fatbin file execution logically incorrect.

We therefore build the database of the human readable text form representations where we store the results of the previous phases, for each PTX instruction of interest, and so a) the corresponding human readable text form representations of the ELF instructions necessary to execute a PTX instruction, b) the correspondences among the PTX registers used in the PTX instruction and the ELF registers, used in the ELF instructions necessary to execute the PTX instruction, together at their names, types and positions in the human readable text form representations of the ELF instructions and c) the possible ELF registers, used in the ELF instructions necessary to execute the PTX instruction, with no corresponding PTX register in the PTX instruction, together at their names, types and positions in the human readable text form representations of the ELF instructions.

Additional reasons to build the database of the human readable text form representations are the following: a) it is probable, that in PTX files different from the PTX files we edited and used to extract the human readable text form interpretations of the ELF instructions necessary to execute the single PTX instructions of interest, the PTX registers used in the PTX instructions of the PTX files have different names and b) it is probable that nvcc is going to assign each time different ELF registers to the different fatbin files produced for the different PTX files and that therefore we will have different ELF registers, that correspond to the analogous, also whether with different names, PTX registers, used in the different PTX files, to execute the ELF instructions necessary to execute the PTX instructions analogous to the single PTX instructions previously considered in the extraction phase.

The utility of the database of the human readable text form representations of the ELF instructions necessary to execute the single PTX instructions of interest is evident when we want to modify the ELF part corresponding to a PTX file with the goal to get the wanted ELF algorithmic implementation. In such cases we need 1) to build the correct human readable text form representations of the ELF instructions necessary to execute each PTX instruction in the PTX file 2) get the binary codes of such human readable text form representations and 3) overwrite one or more parts of the ELF code corresponding to the PTX file.

To be able to build the correct human readable text form representations of the ELF instructions necessary to execute each PTX instruction in the PTX file the database is fundamental because for each PTX instruction, that we want executed in the ELF part corresponding to a PTX file, we need: a) to match the new PTX registers used in the PTX instructions with the old PTX registers used in the PTX instructions during the extraction phase, b) to substitute in the human readable text form representations of the ELF instructions necessary to execute the PTX instructions, human readable text form representations got during the extraction phase, the old ELF registers corresponding to the old PTX registers with the new ELF registers corresponding to the new PTX registers and c) to take care of possible old ELF registers, without any correspondent old PTX register, substituting them with new ELF registers, new ELF registers that this time have a corresponding new PTX register in the PTX file given in input to nvcc to generate the fatbin file that we modify - see why in 6.6.

## 6.5 Database of the Binary Codes of the ELF Instructions

The analysis of an interpretation text file, generated by cuobjdump, about a generic fatbin file, shows that in the second column we see a permutation of the binary codes - 6.2 - permutation associated to the human readable text form representations of the ELF instructions in the third column. Nothing about the position and the number of bits of the opcode, of the ELF registers and of the other flags of the ELF instructions is known. In this section we take care to reverse engineer the position and the number of bits of each ELF register used in each ELF instruction and to discover the values of all the other remaining bits associated to each ELF instruction. This reverse engineering is fundamental to be able to modify the ELF parts corresponding to the PTX files - 6.7.

Because each ELF instruction is composed by 8 bytes - 6.2 - then it is not feasible to try all the possible $2^{64}$ binary combinations to understand a) which bit correspond to which ELF register of which ELF instruction, b) which bits corresponds to the opcodes of which ELF instructions, c) which bits corresponds to the remaining visible things in the human readable text form representations

of which ELF instructions and d) which bits corresponds to the other remaining not visible things in the human readable text form representations of which ELF instructions - we can not exclude, for example, that some bits are useful to set some flags that do not appear in the human readable text form representations of one or more ELF instructions.

We say that it is not feasible to try all the possible $2^{64}$ binary combinations not only for the big quantity of time that would be required but also because one of our conjectures, verified being true at the end of this reverse engineering phase, is that the binary format of the real instruction set architecture is not fixed and so, for example, the positions, of the bits, corresponding to the ELF register, used for the result, of one or more ELF instructions, are different from the positions, of the bits, corresponding to the ELF register, used for the result, of some other ELF instructions - the same it is true a) for all the other fields that can compose a binary code and b) the order of the fields.

To overcome these problems, for each fatbin file that nvcc each time produces a) we generate, using cuobjdump, the interpretation text file of the fatbin file, b) we transform each human readable text form representation, present in the interpretation text file generated by cuobjdump, in its abstract human readable text form representation where the substrings corresponding to the ELF registers used in each ELF instruction are substituted with more generic substrings indicating the type of ELF register and its index type in the ELF instruction - how many times such type of ELF register has already appeared from the beginning of the string of the abstract human readable text form representation - c) we get the binary code of each corresponding human readable text form representation from the interpretation text file, d) we create 64 different binary codes for each binary code selected in c), each one of the 64 binary codes, of 64 bits each one, got switching only 1 of the bits of the original binary code, e) we generate a copy of the fatbin file, f) using the 64 binary codes created at point e), for a number of times equal to the number of ELF instructions visible in the interpretation text file of the fatbin file, we overwrite the first 64 ELF instructions in the ELF part, corresponding to the PTX file, in the copy of the fatbin file, g) for each group of 64 overwrites, using cubojdump, we get the interpretation text file of the partially overwrited fatbin file, h) we extract the human readable text form representations of the first 64 ELF instructions in the interpretation text files of the partially overwrited fatbin file, i) we create the abstract human readable text form representations of the human readable text form representations extracted in h), j) we check which of these abstract human readable text form representations are equal to the abstract human readable text form representations of the original ELF instructions, k) for the binary codes with an abstract human readable text form representation equal to one of the abstract human readable text form representations of the original ELF instructions, we check, the modified bit, which of the ELF registers, used in the original ELF instruction, it is modifying and store the correspondence.

The abstract human readable text form representation of an ELF instruction, the positions of the bits modifying the ELF registers used in the ELF instruction, which ELF register, used in the ELF instruction, each one of such bits is modifying and the value of the remaining other bits not modifying any ELF register used in the ELF instruction, all together are an abstract representation of the ELF instructions or of its binary code.

For each ELF instruction, in the ELF part corresponding to a PTX file, we can determine its abstract representation following the previous procedure and check whether the abstract representation of the ELF instruction has already been discovered and therefore all the binary codes of all the human readable text form representations of the ELF instruction have already been determined

and are in the database of the binaries or not. If not then we need to execute the following procedure: a) generation of all the possible binary codes of the ELF instruction got modifying only the bits corresponding to the ELF registers in the ELF instruction, b) overwriting of the ELF part corresponding to a PTX file in the copy of a fatbin file with all or part of the binary codes generated at point a), c) generation, using cuobjdump, of the interpretation text file of the overwrited fatbin file, d) check that each binary code used to overwrite the copy of the fatbin file has an abstract representation equal to the abstract representation of the binary code of the original ELF instruction, e) check that the bit or group of bits modified for the creation of the binary code are modifying only the correspondent ELF registers used in the ELF instruction, f) updating of the database of the binaries with the couples ( human readable text form interpretation , binary code ) and g) return at point b) if there are other binary codes to use to overwrite the ELF part corresponding to a PTX file in the copy of the fatbin file because the number of ELF instructions in the ELF part corresponding to a PTX file in the copy of the fatbin file was smaller than the number of remaining binary codes generated in a) to use for the overwrites in b).

## 6.6 Fatbin File Generation Satisfying Resource Constraints

From 6.2 we know that the ELF part corresponding to a PTX file is only a part of a fatbin file. In every fatbin file, before and after the ELF part corresponding to a PTX file, there are other two parts that are not visible in the interpretation text file, generated by cuobjdump, for the fatbin file. Let us call A and C the two parts of a fatbin file not visible in its interpretation text file and B the part, visible in its interpretation text file, corresponding to a PTX file.

Analyzing the interpretation text files we discover that there are not ELF instructions corresponding to any of the PTX instructions used to declare the PTX registers in the PTX files and therefore the number and type of ELF registers used by the fatbin files generated by nvcc has to be declared in the parts A and C of the fatbin file.

Using particular flags we can force nvcc to let us know how many hardware registers each GPU thread will have to execute the fatbin file - such number of hardware registers is determined by nvcc, during the compiling phase, after having taken in input a PTX file, but before the generation of the corresponding fatbin file, and it can not change any more after the generation of the fatbin file, this independently of how many times and with which launch configurations the user will decide later to execute the fatbin file.

If we count the number of different text forms corresponding to the different ELF registers visible in the interpretation text file of a fatbin file and compare it to the number of hardware registers each GPU thread has to execute the fatbin file then the number of different text forms corresponding to the different ELF registers visible in the interpretation text file of a fatbin file is always equal to the number of hardware registers each GPU thread has to execute the fatbin file plus 1.

To test the possibility that there is a bug in nvcc we a) edit several PTX files using the editing guidelines in 6.3.1 and b) declare a number of PTX registers per PTX file greater than 64 - the maximum number of hardware registers that can be assigned to a fatbin file. The number of hardware registers returned as output by nvcc is always 63 instead of 64, but analyzing the interpretation text files of the fatbin files the number of different text forms corresponding to different ELF registers visible in the interpretation text file is always 64.

After this first verification, other PTX files are therefore edited using different numbers of PTX

registers per PTX file, each number smaller than 64, this time. Also for all these cases the number of different text forms corresponding to different ELF registers visible in the interpretation text file of a fatbin file is always equal to the number of hardware registers each GPU thread has to execute the fatbin file plus 1.

The two previous checks verify that there is probably a bug in nvcc, this because the two checks allow to speculate a) that the number of different text forms corresponding to the different ELF registers visible in the interpretation text file is probably the real number of hardware registers available to each GPU thread to execute a fatbin file, b) that there is probably a correspondence one to one between ELF registers and hardware registers per GPU thread and c) there are not probably hardware registers, available to each GPU thread for the execution of the fatbin file, not comparing as ELF registers in the interpretation text file.

Because the number of hardware registers available to each GPU thread for the execution of a fatbin file is always smaller than the number of different text forms corresponding to the different ELF registers visible in the interpretation text file of a fatbin file if the previous a) would not true then at least one hardware register should correspond to at least two different ELF registers, if the previous b) would not true then some hardware registers would correspond to more ELF registers and if the previous c) would not true then too some hardware registers would correspond to more ELF registers.

The probability that a), b) and c) are not true is very small because if some hardware registers would correspond to more ELF registers of the same GPU thread then there would be an additional overload in the determination of which warp each warp scheduler can schedule at each warp scheduler clock cycle, thing very unlikely considering that a maximum of 48 warps can be resident at each moment in each streaming multiprocessor, during the execution of a fatbin file, and that the 2 warp schedulers in each streaming multiprocessor have already to compete for the assignment of the warps on the 4 groups of function units of the streaming multiprocessor where the 2 warp scheduler are resident.

Considering a) the previous a), b) and c) true, b) that during the execution of a fatbin file each thread has to execute the B part of the fatbin file in its entirety, c) that the interpretation text file of a fatbin file is unique independently of how many GPU threads are going to execute the fatbin file during a launch and d) that the hardware registers of each GPU thread are private and can not be used/read/written by other GPU threads during the execution of a fatbin file then, in the parts A and C of a fatbin file, nvcc has to have generated some instructions that, using as operands 1) the parameters used in a launch configuration and 2) the distribution of the GPU thread blocks to the streaming multiprocessors after the beginning of the execution of the fatbin file, are able to assign the right number and type of hardware registers to each GPU thread for the execution of the B part of the fatbin file, at the same time avoiding to assign the hardware registers assigned to a GPU thread to another GPU thread.

Because a) we have not ELF instructions to declare the ELF registers we want and b) we do not know the instructions and the procedure, in the parts A and C of a fatbin file, necessary to assign, during the execution of the fatbin file, the hardware registers to the ELF registers in such way to get the guarantee that different hardware registers are used by each GPU thread during the execution of the fatbin file, then we need to develop a procedure to generate PTX files that compiled by nvcc give us fatbin files with the required number and type of ELF registers necessary to modify the fatbin files to get the wanted ELF algorithmic implementations.

If we are successful in this then, because we use the ELF registers that however appear in

the interpretation text file of the fatbin file before of its modification, we do not get any problem during the execution of the modified versions of the fatbin files, fatbin files now with their B parts containing the wanted ELF algorithmic implementations, this because we overwrite only the B parts of the fatbin files, B parts that we know correspond to PTX files and we know being without jumps to the parts A and C of the modified fatbin files as it was already the case in the original fatbin files and so independently of the mechanism used by the GPU architecture to execute a fatbin file, at a given point in time, during the execution of the fatbin files, the control has to be however passed in some ways to the beginnings of the B parts, B parts that will be executed in their entirety without jumps to the parts A and C as happen too in the case of the original fatbin files, B parts that therefore also whether overwrited will not give launch failures due to violations for the use of hardware resources - hardware registers - not initially assigned to them.

Knowing a) the number and type of PTX instructions necessary to execute a PTX file, b) the number and type of ELF registers in the ELF instructions necessary to execute each PTX instruction and c) how many ELF registers that appear in each ELF instruction we want reuse in the other ELF instructions necessary to execute the PTX instructions in the PTX file then, following the editing guidelines in 6.3.1, we can write PTX files that given in input to nvcc will produce as output fatbin files 1) with a minimum number of ELF registers, per type of ELF register, necessary for the overwrites and the modifications of the fatbin files and 2) with a number of ELF instructions in their B parts greater than the number of ELF instructions necessary to get the wanted ELF algorithmic implementations.

Because the editing guidelines are based on some assumptions - 6.3.1 - also whether such assumptions are reasonable we need in any case to check each time 1) the number of each type of ELF register in the interpretation text files of the fatbin files to be sure to have at least the minimum number of ELF registers, per type, necessary for the overwrites and the modifications of the fatbin files and 2) that the B parts of the fatbin files have a number of ELF instructions greater than the number of ELF instructions necessary to get the wanted ELF algorithmic implementations. If such checks for a fatbin file are not satisfied then we can continuously loop generating each time a PTX file with one more PTX register corresponding to one of the remaining ELF registers necessary to satisfy the checks.

Following the previous procedure we can always get the wanted fatbin file but it is also necessary that the following things, that we discovered, are satisfied: 1) the total number of ELF registers wanted for a fatbin file has to be smaller than 65, 2) the total number of ELF instructions wanted in the B part of a fatbin file has to be smaller than 8193, c) the number of ELF registers, starting with "P" or "p" - one type of ELF register - in a fatbin file, has to be smaller than 9 and d) the ELF registers RZ, R0, R1, pt have be used only in some ELF instructions and only in specific positions in such ELF instructions.

## 6.7 Wanted ELF Algorithmic Implementations

Thanks to the results in the previous sections we are now able to get any wanted ELF algorithmic implementation if we execute the following procedure: 1) determination of the type and order of PTX instructions necessary to execute an algorithm, 2) check, in the database of the human readable text form representations, of the number of ELF instructions necessary to execute each one of the PTX instructions, 3) determination of the number of ELF instruction necessary to execute the algorithm - this is possible considering the results got at the steps 1) and 2) - 4) check, in the

database of the human readable text form representations, the number and type of ELF registers necessary to execute each necessary ELF instruction, 5) determination of the total number and type of ELF registers we want use for the execution of the ELF instructions necessary to execute the algorithm - this is possible considering a) the results got at the step 1), b) the type and order of ELF instructions necessary to execute each PTX instruction, c) the results got at the step 4) and d) the frequency and type of reuse we want for each ELF register of the fatbin file - 6) creation of the fatbin file with a) at least the minimum number of ELF instructions and b) at least the minimum number of ELF registers, per type of ELF register, necessary to make possible the overwrites and the modifications of the fatbin file in the wanted way - step 6) is accomplished a) using the editing guidelines in 6.3.1, b) the procedure in 6.6 and c) PTX instructions completely different from the PTX instruction necessary to execute the algorithm, this because our goal here it is to get a fatbin file satisfying the resource constraints - 7) matching of the PTX registers, used in the PTX instructions necessary for the execution of the algorithm, with the ELF registers of the fatbin file - we assign each ELF register of the fatbin file to one or more PTX registers, this considering the order of the ELF instructions and the frequency and type of reuse we want for each ELF register of the fatbin file, things decided at the step 5) - 8) for each PTX instruction we match the PTX registers used in the PTX instruction with the original PTX register used in the PTX instruction during the extraction phase in 6.4, 9) substitution of the original ELF registers, used in the ELF instructions, necessary to execute the PTX instructions, during the extraction phase in 6.4, with the corresponding ELF registers - this is possible because a) we know each original ELF register to which original PTX register correspond and b) at the step 9) we have done the match among PTX registers now used in the PTX instructions and original PTX registers used in the PTX instructions during the extraction phase in 6.4, 10) substitution of the original ELF registers, used in the ELF instructions, necessary to execute the PTX instructions, without correspond original PTX register, with the ELF registers that at the step 5) we have decided to use for this - these are now ELF registers with a corresponding PTX register, declared in the PTX file, used for the generation of the fatbin file, thing done to get the guarantee to have how many ELF registers for type and number we want for these types of substitutions, 11) overwriting of the B part of the fatbin file, in order, with the ELF instructions with the wanted a) new ELF registers and b) dependences among the new ELF registers and 12) overwriting of the possible remaining original ELF instructions of the fatbin file with the exit ELF instruction, this because we generated a fatbin file with a number of ELF instructions in its B part at least equal to the number of ELF instructions necessary to get the wanted ELF algorithmic implementation, see step 6).

The modified fatbin file so obtained has in its B part the wanted ELF algorithmic implementation. In the B part of the modified fatbin file a) there are the ELF instructions necessary to execute, in the wanted order, the PTX instructions and b) the ELF registers, used in the ELF instructions, have the wanted dependence types - write-read, read-read, etc. - and the wanted dependence distances - in number of ELF instructions. Such modified fatbin files run without launch failures for every possible launch configuration and their execution is logically correct.

## 6.8 Summary

In this chapter we have described the procedures a) that were necessary to uncover not disclosed information about the real instruction set architecture and other features of the ELF codes and b) that are necessary to generate and modify fatbin files to get the wanted ELF algorithmic imple-

mentations. The most important points to remember from this chapter are the following:

- In a fatbin file, the ELF instructions, corresponding to the PTX code of a PTX file, are consecutive and occupy only a part of the fatbin file, the B part - the part A and C of the fatbin file are created by nvcc and it not given to know what they contain. We are able to localize the B part of each fatbin file, this is important because the B part is the part of a fatbin file that we need to modify to get the wanted ELF algorithmic implementations;

- To execute a PTX instruction one or more ELF instructions are necessary. The correspondence, between PTX instruction and number, type and order of ELF instructions necessary to execute the PTX instruction, has been determined for all the PTX instructions of interest. The correspondences, between ELF registers used in the ELF instructions necessary to execute a PTX instruction and PTX registers used in the PTX instruction, too have been determined so how which ELF registers used in the ELF instructions necessary to execute a PTX instruction do not have a correspondent PTX register in the PTX instruction.

  These results are useful when we want modify a fatbin file using a set of ELF registers different from the original one used in the ELF instructions. Knowing which ELF register is what in each ELF instruction allows us to modify an ELF code getting the guarantee that the execution of the modified ELF code is logically correct because we use each ELF register, in each ELF instruction, in the correct role;

- The reverse engineering of the real instruction set architecture has been executed so now we know that the binary codes of the real instruction set architecture have not a fixed format. With the results of the reverse engineering we are now able to generate all the binary codes of each ELF instruction of interest and overwrite the B part of a fatbin file using the binary codes corresponding to the ELF instructions we want with the ELF registers we want;

- There are not ELF instructions to assign ELF registers to a fatbin file so the ELF registers assigned to a fatbin file have to be assigned by nvcc during the compiling phase and later correspond to some hardware registers that has to be different for each GPU thread used for the execution of the fatbin file.

  The procedure for the assignment of hardware registers to the ELF registers of a fatbin file is not in the B part of a fatbin file so it has to be in the part/s A and/or C - parts A and C are not disclosed - because nvcc can not know the launch configuration that we are going to use at each different execution of the fatbin file and so nvcc can not know the number of GPU thread that each time is going to execute the fatbin file - the number of GPU threads could be effectively different each time.

  To get the wanted ELF algorithmic implementation its therefore necessary a) to write a PTX file using a particular set of editing guidelines - 6.3.1 - b) to give the PTX file in input to nvcc to get as output a fatbin file, that thanks to the use of the editing guidelines, has at least the minimum number of ELF registers, per type of ELF register, necessary for the modification of the fatbin file and c) to modify, in the fatbin file, the ELF code corresponding to the PTX file, to get the wanted ELF algorithmic implementations;

- For each wanted ELF algorithmic implementation we want, thanks to the previous results, we can always generate first a fatbin file having at least the minimum number of ELF registers, per type of ELF register, we want for the modification of the fatbin file and later we can

always modify the fatbin file overwriting its B part with the order, type and number of ELF instructions we want, each of them using the ELF registers that we want.

Such modified fatbin file a) have the wanted ELF algorithmic implementation, b) run without launch failures due to violations for the use of ELF registers not originally assigned to it and c) give us the guarantee that its execution its logically correct because we know the role of each ELF register that we use in each ELF instruction - whether result, operand, etc. - and the specific values of the bits, in the binary codes of each ELF instruction, for the determination of the particular ELF register we want in each role in each ELF instruction.

Being able to get the wanted ELF algorithmic implementations is however not enough to optimize the execution of the ELF code of the B part of a fatbin file. It is probable effectively that there are also some GPU behaviors - a) not disclosed, b) not quantified, c) determined by the GPU hardware design and d) beyond the control of the users - able to influence the execution time of a fatbin file. In the next chapter we therefore discover, understand and quantify such GPU behaviors.

# Chapter 7

# Discovery, Understanding and Quantification of Not Disclosed GPU Behaviors

## 7.1 Introduction

Being able to modify ELF codes is not enough to understand how to optimize them. There are surely not disclosed GPU behaviors, due to the GPU hardware design, that is necessary to discover, understand and quantify, to be able to optimize and analyze ELF codes.

Some not disclosed GPU behaviors are probably controllable and easily avoidable, others - as the warp scheduling mechanism used by the warp schedulers to schedule warps on the 4 groups of function units in each streaming multiprocessor - we already know are impossible to control or to avoid. Every ELF code has therefore to be structured and being launched in such a way that: a) the impact of GPU behaviors, that could have a negative impact but that can be avoided, is made null and b) the impact of GPU behaviors, that could have a negative impact but that can not be avoided, is minimized.

We divide the not disclosed GPU behaviors we want to verify, understand and quantify in two categories - global and local - and we explain a) which GPU architectural features we need to consider to verify whether the not disclosed GPU behaviors really exist and b) how we quantify each GPU architectural feature.

Next we talk of the structures of the ELF codes used for the quantifications. Such structures have to give the guarantee that the byte transfers among the different GPU memories can not slow down the executions of the ELF codes. We therefore describe a) the structures of the PTX codes and of the ELF codes used for the quantification of the GPU architectural features and b) the automatic procedure for the generation of all the necessary fatbin files necessary for the extraction of each GPU architectural feature of each instruction configuration.

Because each fatbin file can be launched in many different ways, understanding a) the launch configurations used for each one of the two categories of not disclosed GPU behaviors - global and local - and b) why the launch configurations have to be different, is useful to understand the logic behind the reliability of the ELF code execution times used in the quantification of the GPU architectural features.

The GPU architectural features are therefore extracted and quantified and the verified, understood and quantified GPU behaviors are explained. If a GPU behavior is avoidable and could have a negative impact on the execution times of an ELF code then an explanation on how to avoid it is given. If a GPU behavior is not avoidable and could have a negative impact on the execution times of an ELF code then its variability is studied and is considered during the analysis/analyses in the next chapters.

## 7.2 Not Disclosed GPU Behavior Categories

The not disclosed GPU behaviors we want to verify, understand and quantify can be divided in two categories:

- The category of the not disclosed global GPU behaviors The not disclosed global GPU behaviors are determined by the gigathread scheduler - 3.2 - behavior a) at chip level and b) at streaming multiprocessor level where some parts of the gigathread scheduler - the warp schedulers - are not synchronized among different streaming multiprocessors as instead are at chip level the parts of the gigathread scheduler distributing GPU thread blocks to the streaming multiprocessors.

  Verifying, understanding and quantifying the not disclosed global GPU behaviors and their variabilities is important a) for the global GPU load balancing analysis and b) to prove that the executions, of the ELF code, of the B part, of the fatbin file, of a couple ( fatbin file , launch configuration ), can not be slowed down by the bandwidths and the latencies of the GPU memories, when the fatbin file is executed using the launch configuration of the couple - 12.2;

- The category of the not disclosed local GPU behaviors The not disclosed local GPU behaviors are determined a) by the parts of the gigathread scheduler at the streaming multiprocessor level - the warp schedulers - and b) by the streaming multiprocessor hardware design.

  Verifying, understanding and quantifying the not disclosed local GPU behaviors and their variabilities is important a) for the local streaming multiprocessor load balancing analysis, b) to prove that the executions, of the ELF code, of the B part, of the fatbin file, of a couple ( fatbin file , launch configuration ), can not be slowed down by the bandwidths and the latencies of the GPU memories, when the fatbin file is executed using the launch configuration of the couple - 12.2 - and c) to prove that it is not possible, during the execution, of the ELF code, of the B part, of the fatbin file, of a couple ( fatbin file , launch configuration ), the generation of stalls, in the instruction pipelines of the streaming multiprocessors, due to the number of resident warps in each streaming multiprocessor, when the fatbin file is executed using the launch configuration of the couple - 12.3.

However, to be able to verify, understand and quantify the not disclosed global and local GPU behaviors is first necessary to quantify the not disclosed GPU architectural features.

## 7.3 GPU Architectural Features

The GPU architectural features are divided a) in global GPU assignment and scheduling architectural features - 7.5.1 - useful to verify, understand and quantify the not disclosed global GPU

behaviors and b) in local streaming multiprocessor PTX and ELF architectural features - 7.5.2 - useful to verify, understand and quantify the not disclosed local GPU behaviors

### 7.3.1 Global GPU Assignment and Scheduling Architectural Features

The global GPU assignment and scheduling architectural features are determined by GPU hardware limitations due to the GPU hardware design. Considering the different functions of the global architectural features, the global architectural features can be divided in two groups:

- The first group. The first group is useful to determine whether the gigathread scheduler is always evenly distributing the GPU thread blocks to the streaming multiprocessors. If the gigathread scheduler does not assign in an even way the GPU thread blocks to the streaming multiprocessors then this could vanify all the efforts done to optimize an ELF code because when a GPU thread block has been assigned to a streaming multiprocessor the GPU thread block can not migrate any more;

- The second group. The second group considers several time differences - in number of clock cycles - regarding the starting and ending warp scheduling phases of the resident warps on all the streaming multiprocessors.

  Because the GPU warp scheduling policies are not disclosed, the time differences are useful to understand how the resident warps on the whole GPU are made advancing after that they have been all scheduled at least one time.

  The quantification of these second group of global architectural features is useful to determine whether the execution, of the ELF code, of the B part, of the fatbin file, of a couple ( fatbin file , launch configuration ), can not be slowed down by the bandwidths and the latencies of the GPU memories, when the fatbin file is executed using the launch configuration of the couple - 12.2.

  If the execution, of the ELF code, of the B part, of the fatbin file, of a couple ( fatbin file , launch configuration ), could be slowed down by the bandwidths and the latencies of the GPU memories, when the fatbin file is executed using the launch configuration of the couple, then, to avoid this, we can modify the fatbin file of the couple in several different ways that we will explain in the next chapters.

  If instead the execution, of the ELF code, of the B part, of the fatbin file, of a couple ( fatbin file , launch configuration ), can not be slowed down by the bandwidths and the latencies of the GPU memories, when the fatbin file is executed using the launch configuration of the couple, then it is necessary to analyze what happens locally in each streaming multiprocessor. To analyze what happens locally in each streaming multiprocessor it is necessary to quantify the local streaming multiprocessor PTX and ELF architectural features.

### 7.3.2 Local Streaming Multiprocessor PTX and ELF Architectural Features

The local streaming multiprocessor PTX and ELF architectural features are features determined by the streaming multiprocessor hardware limitations due to the streaming multiprocessor hardware design. Considering the different function of the local streaming multiprocessor PTX and ELF

architectural features, the local streaming multiprocessor PTX and ELF architectural features can be divided in two groups:

- The first group. The first group is composed by the same time differences - in number of clock cycles - of the second group of features of the global GPU assignment and scheduling architectural features.

  The time differences are taken and determined locally for a single streaming multiprocessor and are useful to understand as the resident warps in a streaming multiprocessor are made advancing after that they have been all scheduled at least one time.

  The quantification of these second group of global architectural features is useful to determine whether the execution, of the ELF code, of the B part, of the fatbin file, of a couple ( fatbin file , launch configuration ), can not be slowed down by the bandwidths and the latencies of the GPU memories, when the fatbin file is executed using the launch configuration of the couple - 12.2.

  If the execution, of the ELF code, of the B part, of the fatbin file, of a couple ( fatbin file , launch configuration ), could be slowed down by the bandwidths and the latencies of the GPU memories, when the fatbin file is executed using the launch configuration of the couple, then, to avoid this, we can modify the fatbin file of the couple in several different ways that we will explain in the next chapters.

  If instead the execution, of the ELF code, of the B part, of the fatbin file, of a couple ( fatbin file , launch configuration ), can not be slowed down by the bandwidths and the latencies of the GPU memories, when the fatbin file is executed using the launch configuration of the couple, then it is necessary to consider the features of the second group of the local streaming multiprocessor PTX and ELF architectural features.

- The second group. The second group considers several local streaming multiprocessor PTX and ELF architectural features. The features are useful:

  - a) To quantify the real instruction configuration streaming multiprocessor best average performance per clock cycle;
  - b) To understand whether it is possible to get load unbalancing for the warp scheduling in a streaming multiprocessor if the gigathread scheduler has evenly distributed the GPU thread blocks to the streaming multiprocessors, and if yes, whether and how it is possible to get load balancing for the warp scheduling in a streaming multiprocessor;
  - c) To understand whether the warp schedulers have a scheduling waiting time. When a warp is scheduled, at cause of the GPU hardware design, it is possible that at least a minimum quantity of clock cycles has to pass before the warp schedulers can schedule the warp again and that this minimum quantity of time can be due to causes different a) by the write-read and read-read dependence waiting times - see d) below for an explanation - and/or b) the overhead time for the management of the warps - see e) below for an explanation. This minimum quantity of time is the scheduling waiting time.

    If the warp schedulers have a scheduling waiting time then it is necessary to understand whether the scheduling waiting time is different for different ELF instruction configurations and if yes then it is necessary to quantify the scheduling waiting time for each ELF instruction configuration;

- d) To quantify the number of clock cycles that is necessary to wait before to be able to read a data previously written in an ELF register - write-read dependence waiting time - for each ELF instruction configuration and the number of clock cycles that is necessary to wait before to be able to read a data previously read from an ELF register - read-read dependence waiting time - for each ELF instruction configuration;
- e) To understand whether there is an overhead time for the management of the warps, if yes then whether its increase is not linear when the number of resident warps in a streaming multiprocessor is linearly increasing, if yes then which can be the shape of a function expressing it;
- f) To determine the minimum number of resident warps necessary in a streaming multiprocessor to get the real instruction configuration best average performance per clock cycle of each ELF instruction configuration for each dependence distance.

The motivations to verify and quantify the local streaming multiprocessor PTX and ELF architectural features a), b) c), d), e) and f) are the following:

- Verifying and quantifying the real instruction configuration streaming multiprocessor best average performance per clock cycle is useful to get a first idea about the minimum number of clock cycles that is necessary to the GPU to execute some ELF codes and after the execution of the ELF codes to get a more accurate measure of their efficiencies.

  As limit case considers, for example, an ELF code of only sine ELF instructions with operands at 32 bits. The GPU theoretical peak performance is around 0.5 TF/s but not more than 4 sine ELF instructions can be executed in a clock cycle per streaming multiprocessor not more than 4 * 14 = 56 sine ELF instructions can be executed in a clock cycle by the GPU.

  Considering that the 4 special function units used to execute the sine ELF instructions have a clock frequency of 1.15 Ghz, this gives a GPU theoretical peak performance, for the ELF code, of about 0.06 TF/s instead of about 0.5 TF/s.

  The real instruction configuration streaming multiprocessor best average performance per clock cycle of the PTX and ELF instruction configurations is also important to understand which function units in a streaming multiprocessor are executing which PTX or ELF instruction configurations and to understand whether there are some not disclosed hardware resources, shared among the possible subsets of the 4 groups of function units in a streaming multiprocessor, for the parallel execution of all the possible couples or triplets of PTX or ELF instruction configurations;
- Verifying whether and when there is load unbalancing for the warp scheduling in a streaming multiprocessor, if the gigathread scheduler has evenly distributed the GPU thread blocks to the streaming multiprocessors, is important because, supposing the gigathread scheduler has evenly distributed the GPU thread blocks to the streaming multiprocessors, if we can not force load balancing for the warp scheduling in a streaming multiprocessor then the load unbalancing for the warp scheduling could have a very bad impact on the execution time of an ELF code.
- Verifying and quantifying the existence of the scheduling waiting time/times is important because if a) a couple ( fatbin file , launch configuration ) has an even GPU thread block

distribution on the streaming multiprocessors, b) we know how the warp schedulers are making advance the warps at the global and local level after all the warps have been scheduled at least one time, c) the execution, of the ELF code, of the B part, of the fatbin file, of the couple ( fatbin file , launch configuration ), can not be slowed down by bandwidths and latencies of the GPU memories, when the fatbin file is executed using the launch configuration of the couple and d) we can get load balancing, for the warp scheduling, in each streaming multiprocessor, then, given the ELF instruction configurations in the B part of the fatbin file, the scheduling waiting times for each ELF instruction configuration in the ELF code allow to determine, in the cases where the scheduling waiting time is the limiting factor, the minimum number of resident warps necessary locally in each streaming multiprocessor to avoid instruction pipeline stalls - due to the scheduling waiting times - during the execution, of the ELF code, of the B part, of the fatbin file, of a couple ( fatbin file , launch configuration ), when the fatbin file is executed using the launch configuration of the couple;

- Verifying and quantifying the write-read dependence waiting times and the read-read dependence waiting times is important because knowing a), b), c), d) of the previous paragraph and the ELF instruction configurations in the B part of a fatbin file, the write-read dependence waiting times and the read-read dependence waiting times of the ELF instruction configurations in the B part of a fatbin file allow to determine, in the cases where the dependence waiting times are the limiting factor, the minimum number of resident warps necessary locally in a streaming multiprocessor to avoid instruction pipeline stalls - due to the dependences waiting times - during the execution, of the ELF code, of the B part, of the fatbin file, of a couple ( fatbin file , launch configuration ), when the fatbin file is executed using the launch configuration of the couple and to determine the precise - and also average - number of clock cycles necessary to execute each ELF instruction configuration - 7.6.2;

- Verifying the existence of an overhead time for the management of the warps is useful to understand whether the real ELF code streaming multiprocessor average performance per clock cycle can be smaller than the theoretical streaming multiprocessor best average performance per clock cycle for causes different from the scheduling waiting times, the dependence waiting times and the scheduling of the warps on the not disclosed hardware resources shared among the subsets of the 4 groups of function units in each streaming multiprocessor.

  Verifying, whether the overhead time for the management of the warps is increasing not linearly as the number of resident warps in a streaming multiprocessor linearly increases, allows to understand that for the ELF instruction configurations the number of resident warps necessary locally in a streaming multiprocessor could have to be greater than the minimum got considering only the scheduling waiting time and the dependence waiting time of the ELF instruction configuration.

  Determining the shape of a function able to express the overhead time for the management of the warps allows to understand for which triplets ( ELF instruction configuration , dependence distance , number of resident warps in a streaming multiprocessor ) the overhead time for the management of the warps could be the limiting factor in getting the real ELF instruction configuration streaming multiprocessor best average performance

per clock cycle.

Considering the effects of the overhead time for the management of the warps allows to determine, in the cases where the overhead time for the management of the warps is the limiting factor, the minimum number of resident warps necessary locally in a streaming multiprocessor to avoid instruction pipeline stalls - due to the overhead time for the management of the warps - during the execution, of the ELF code, of the B part, of the fatbin file, of a couple ( fatbin file , launch configuration ), when the fatbin file is executed using the launch configuration of the couple;

– Determining the minimum number of resident warps necessary in a streaming multiprocessor to get the real instruction configuration best average performance per clock cycle of each ELF instruction configuration for each dependence distance is important because knowing a), b), c), d) and the ELF instruction configurations in the B part of a fatbin file, then the minimum number of resident warps necessary in a streaming multiprocessor to get the real instruction configuration best average performance of each ELF instruction configuration for each dependence distance allows to determine the possible numbers of resident warps necessary locally in a streaming multiprocessor to avoid instruction pipeline stalls - due to the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps - during the execution, of the ELF code, of the B part, of the fatbin file, of a couple ( fatbin file , launch configuration ), when the fatbin file is executed using the launch configuration of the couple - 7.6.2.

To verify and quantify each GPU PTX and ELF architectural feature, the execution times, of the ELF codes used, have to be accurate and reliable and so give an *a priori* guarantee that the byte transfers among the different GPU memories can not slow down the executions, of the ELF codes, of the B parts, of the fatbin files. To get an *a priori* guarantee that the byte transfers among the different GPU memories can not slow down the executions, of the ELF codes, of the B parts, of the fatbin files, it is necessary the ELF codes, of the B parts, of the fatbin files, have the structure indicated in the next section.

## 7.4 PTX and ELF Codes

The ELF codes, of the B parts, of the fatbin files, has to be written in such way to give an *a priori* guarantee that the byte transfers, among the different GPU memories, can not slow down the executions of one of their parts. This is important because in the quantification of the architectural features we need to have an *a priori* guarantee that the execution times of one of the parts of each ELF code are due only a) to the gigathread scheduler hardware limitations and b) to the streaming multiprocessor hardware limitations - warp schedulers, instruction pipeline depths, waiting times due to write-read and read-read dependences among ELF registers, not disclosed hardware resources shared among the possible subsets of the 4 groups of function units in each streaming multiprocessor, etc. .

### 7.4.1 *A Priori* Bandwidth and Latency GPU Memories Free Guarantee

The B part of every fatbin file has to be of limited length. For this reason the B part, of each fatbin file used for the quantification of the architectural features, has inside a for loop - in this

way we can iterate on the for loop more times and so make each GPU thread to execute at least the minimum number of ELF instructions we want.

The data, necessary to execute the ELF instructions inside the for loop, are loaded before the for loop. A written request to the same global memory location and a memory synchronization barrier are edited just before the beginning of the for loop. The GPU threads are released only after a) all them have met the synchronization barrier and b) all them have satisfied all the previous memory requests and so also the writing request to the same global memory location. Doing this we have the guarantee a) that all the necessary data, used inside the for loop, are in the ELF registers before the beginning of the execution of the for loop and b) that, during the execution of the for loop, the execution of the ELF instructions can not be slowed down by the byte transfers among the different GPU memories. Just after the synchronization barrier each GPU thread get the global GPU clock cycle and enter to execute the for loop. Later, just after the end of the for loop, each GPU thread get the GPU clock cycle, executes a writing request to the same global memory location and met another memory synchronization barrier.

Considering a) that the worst case global memory latency is not greater than 800 function unit clock cycles - [50, p. 87] and [56, p. 67] say 800 function unit clock cycles, [49, p. 47] and [55, p. 57] say 600 function unit clock cycles - and b) that the read and the write operations to the same global memory location are not atomic among GPU threads, such memory synchronization barriers are going to produce a incredibly small noise on the final calculation of the execution times necessary to execute the for loops because the number of ELF instructions executed for each for loop is of the order of magnitude of the millions.

### 7.4.2 Structure of the PTX and ELF Codes

In this subsection 7.4.2 and in the next subsection 7.4.3 we give only examples for the PTX cases, no example for the ELF cases is given but all the reasonings can be repeated in the same way considering ELF instructions instead of PTX instructions.

Table 7.1: Part of the for loop of the PTX file ( add.u32 , normal mode , write-read , 3 ) where ( add.u32 , normal mode , write-read ) is the PTX instruction configuration and 3 is the dependence distance considered for the write-read dependence type.

```
add.u32 %result_operand_0, %result_operand_0, %result_operand_0;
add.u32 %result_operand_1, %result_operand_1, %result_operand_1;
add.u32 %result_operand_2, %result_operand_2, %result_operand_2;

add.u32 %result_operand_0, %result_operand_0, %result_operand_0;
add.u32 %result_operand_1, %result_operand_1, %result_operand_1;
add.u32 %result_operand_2, %result_operand_2, %result_operand_2;

add.u32 %result_operand_0, %result_operand_0, %result_operand_0;
add.u32 %result_operand_1, %result_operand_1, %result_operand_1;
add.u32 %result_operand_2, %result_operand_2, %result_operand_2;

add.u32 %result_operand_0, %result_operand_0, %result_operand_0;
add.u32 %result_operand_1, %result_operand_1, %result_operand_1;
add.u32 %result_operand_2, %result_operand_2, %result_operand_2;
```

The PTX code is showing 4 groups of add.u32 PTX instructions executed in normal mode. Each group has the same 3 add.u32 PTX instructions - the name of the PTX registers determines whether two PTX instructions are equal. The type of dependence between each couple of two equal add.u32 PTX instructions is write-read at distance 3.

Table 7.2: Part of the for loop of the PTX file ( sub.s32 , conditional mode -> true , read-read , 2 ) where ( sub.s32 , conditional mode -> true , read-read ) is the PTX instruction configuration and 2 is the dependence distance considered for the read-read dependence type.

```
@%guard_0 sub.s32 %result_0, %operand_0, %operand_0;
@%guard_1 sub.s32 %result_1, %operand_1, %operand_1;

@%guard_0 sub.s32 %result_0, %operand_0, %operand_0;
@%guard_1 sub.s32 %result_1, %operand_1, %operand_1;

@%guard_0 sub.s32 %result_0, %operand_0, %operand_0;
@%guard_1 sub.s32 %result_1, %operand_1, %operand_1;

@%guard_0 sub.s32 %result_0, %operand_0, %operand_0;
@%guard_1 sub.s32 %result_1, %operand_1, %operand_1;
```

The PTX code is showing 4 groups of sub.s32 PTX instructions executed in a conditional way - the guard has to be true. Each group has the same 2 sub.s32 PTX add instructions. The type of dependence between each couple of two equal sub.s32 PTX instructions is read-read at distance 2.

Table 7.3: Part of the for loop of the PTX file ( xor.b32 , conditional mode -> false , write-read , 4 ) where ( xor.b32 , conditional mode -> false , write-read ) is the PTX instruction configuration and 4 is the dependence distance considered for the write-read dependence type.

```
!@%guard_0 xor.b32 %result_operand_0, %result_operand_0, %result_operand_0;
!@%guard_1 xor.b32 %result_operand_1, %result_operand_1, %result_operand_1;
!@%guard_2 xor.b32 %result_operand_2, %result_operand_2, %result_operand_2;
!@%guard_3 xor.b32 %result_operand_3, %result_operand_3, %result_operand_3;

!@%guard_0 xor.b32 %result_operand_0, %result_operand_0, %result_operand_0;
!@%guard_1 xor.b32 %result_operand_1, %result_operand_1, %result_operand_1;
!@%guard_2 xor.b32 %result_operand_2, %result_operand_2, %result_operand_2;
!@%guard_3 xor.b32 %result_operand_3, %result_operand_3, %result_operand_3;

!@%guard_0 xor.b32 %result_operand_0, %result_operand_0, %result_operand_0;
!@%guard_1 xor.b32 %result_operand_1, %result_operand_1, %result_operand_1;
!@%guard_2 xor.b32 %result_operand_2, %result_operand_2, %result_operand_2;
!@%guard_3 xor.b32 %result_operand_3, %result_operand_3, %result_operand_3;
```

The PTX code is showing 3 groups of xor.b32 PTX instructions executed in a conditional way - the guard has to be false. Each group has the same 4 xor.b32 PTX instructions. The type of dependence between each couple of two equal xor.b32 PTX instruction is write-read at distance 4.

To minimize the noise given by a) the increment of the variable of the number of for cycles executed, b) the setting of the guard for the execution of the next cycle of the for loop and c) the

conditional jump for possibly repeating the for loop, we put, a) the increment of the variable, of the number of for cycles executed, to the beginning of the for loop, b) the setting of the guard, for the execution of the next cycle of the for loop, in the middle of the for loop and c) the conditional jump, for possibly repeating the for loop, to the end of the for loop.

### 7.4.3 Construction of the PTX and ELF Codes

Using the results of the previous chapter we construct the B parts of the fatbin files necessary to extract the GPU PTX architectural features for each PTX instruction configuration of interest.

For the construction of the B parts of the fatbin files the following propositions are true: a) the B parts of the fatbin files have to be of limited length, b) the number of ELF registers, that each GPU thread can have, can not be greater than 64 and c) a limited number and type of ELF registers are necessary to execute the ELF instructions necessary to execute a PTX instruction configuration. Because a), b) and c) are true, for each PTX instruction configuration, it is possible to generate, in an automatic way, all the B parts of the fatbin files necessary to quantify the GPU PTX architectural features of the PTX instruction configuration.

Suppose, for example, that an add.u32 PTX instruction, executed in normal mode - see the first of the previous examples in 7.4.2 - requires 5 ELF registers, all of the same type - for example not predicate and so all at 32 bits. Considering that a) each GPU thread can not have more than 64 ELF registers and b) that - for example - 13 ELF registers has to be reserved to execute things different from the add.u32 PTX instructions, executed in normal mode, inside the for loop, this leaves 51 ELF registers to execute the add.u32 PTX instructions and so a maximum of 51 different add.s32 PTX instructions, to execute in normal mode, inside the for loop. For the PTX instruction configuration ( add.u32 , normal mode , write-read ), 51 fatbin files are therefore generated. The first fatbin file has groups with only an add.u32 PTX instruction - this fatbin file considers the distance 1 for the write-read dependence type of the PTX instruction configuration - the second has groups with only 2 add.u32 PTX instructions - this fatbin file considers the distance 2 for the write-read dependence type of the PTX instruction configuration - etc. . During the creation of these 10 fatbin files, the number of ELF instructions necessary to execute an add.u32 PTX instruction in normal mode determines the number of ELF instructions $NEI_g$ necessary to execute a group. The number of ELF instructions $NEI_r$ reserved in the fatbin file to execute add.u32 PTX instructions, in normal mode, with a write-read dependence type, inside the for loop, of the B part, of the fatbin file, determines the number of groups $N_g = \lfloor \frac{NEI_r}{NEI_g} \rfloor$ of add.u32 PTX instructions, to execute in normal mode, with a write-read dependence type, to edit inside the for loop, of the B part, of the fatbin file.

## 7.5 Launch Configurations

Considering we can use many possible launch configurations to execute a fatbin file - 2.5 - we always choose to use the minimum number of GPU thread blocks, in their simplest a) logic GPU thread block distribution, b) GPU thread block composition and c) logic GPU thread block form, to satisfy our distribution requirements on the streaming multiprocessors.

During the quantification of the GPU architectural features for each instruction configuration we calculate several tables. The rows of these tables represent the fatbin files constructed for the instruction configuration - one fatbin file for each one of the dependence distances of the dependence

type of the instruction configuration. The columns of these tables represent the launch configurations used to execute the fatbin files. Some of these tables contain only architectural features - for example time differences - while other contains values useful to quantify some architectural features - for example the real instruction configuration streaming multiprocessor best average performance per clock cycle.

### 7.5.1 Global GPU Assignment and Scheduling Architectural Features

The number of streaming multiprocessors, the maximum number of resident warps per streaming multiprocessor and the maximum number of warps per GPU thread block determine the minimum number of GPU thread blocks necessary to get on each GPU streaming multiprocessor the maximum number of resident warps per streaming multiprocessor - remember that are the GPU thread blocks that are assigned to the streaming multiprocessors, that the GPU thread blocks can not migrate after the assignment, that the warps are always composed by a fixed number of GPU threads, 32, and that are the warps that are scheduled by the 2 warps schedulers in each streaming multiprocessor and so our focus has to be on the number of GPU thread blocks and the number of warps per GPU thread block and not on the single GPU threads.

For our machine, the number of streaming multiprocessors is 14, the maximum number of resident warps per streaming multiprocessor is 48, the maximum number of warps per GPU thread block is 32 and so the minimum number of GPU thread blocks to get on each GPU streaming multiprocessor the maximum number of resident warps is $2*14 = 28$ because 2 GPU thread blocks, with $24 < 32$ warps per GPU thread block, on each one of the 14 streaming multiprocessors, give the maximum number of resident warps per streaming multiprocessor, 48.

For each fatbin file we have therefore a maximum of 24 launch configurations, 1 per possible number of warps per GPU thread block. Given a fatbin file, if the number of ELF registers per GPU thread does not allow to 2 GPU thread blocks of X warps each one to fit in a streaming multiprocessor the $1 \leq X \leq 24$ launch configuration for the fatbin file is not used. Every fatbin file is executed Y times for each one of the launch configurations.

### 7.5.2 Local Streaming Multiprocessor PTX and ELF Architectural Features

Only one of the GPU streaming multiprocessors is used and only a GPU thread block is used per launch configuration. With only a GPU thread block per launch configuration there is no logic GPU thread block distribution to choose. Because each GPU thread block can have a maximum of 1032 GPU threads and because each warp is always composed by 32 GPU threads, no GPU thread block can have more than 32 warps and so, because only a GPU thread block is used, each fatbin file can not be launched with more than 32 different launch configurations - 1 per possible number of warps of the only GPU thread block used. Every fatbin file is executed Y times for each one of the launch configurations.

For some fatbin files not all the launch configurations are used because a) each streaming multiprocessor has $2^{15}$ ELF registers and b) each GPU thread can not have more than 64 ELF registers, and so if the fatbin file has more than $\frac{2^{15}}{2^5 * 2^5} = \frac{2^{15}}{2^{10}} = 2^5 = 32$ ELF registers then less than 32 warps are necessary to fully occupy the hardware register resources of a streaming multiprocessor.

On the other side the fatbin files with less than $2^5 = 32$ ELF registers per GPU thread do not completely occupy all the hardware register resources of a streaming multiprocessor but this is

not a problem to quantify the real instruction configuration streaming multiprocessor best average performance per clock cycle of each instruction configuration - see why in 7.6.2.

## 7.6 GPU Architectural Feature Quantifications

In the next two subsections we quantify the global GPU assignment and scheduling architectural features - useful to verify, understand and quantify the not disclosed global GPU behaviors and their variabilities - and the local streaming multiprocessors PTX and ELF architectural features - useful to verify, understand and quantify the not disclosed local GPU behaviors and their variabilities.

### 7.6.1 Global GPU Assignment and Scheduling Architectural Features

For each couple ( fatbin file , launch configuration ), of the each instruction configuration, the first group of GPU architectural features is quantified. The GPU architectural features are: a) the possibility, for the gigathread scheduler, to assign, the GPU thread blocks, to the streaming multiprocessors, in a not even way, b) the number of fatbin file launches, c) the number of failed fatbin file launches, d) the number of not failed fatbin file launches, e) the number of not failed fatbin file launches with even distribution of the GPU thread blocks to the streaming multiprocessors and f) the number of not failed fatbin file launches with not even distribution of the GPU thread blocks to the streaming multiprocessors.

Several automatic checks are executed. A part of these automatic checks consider the number and type of launches per fatbin file - failed, not failed, with even or not even distribution of the GPU thread blocks to the GPU streaming multiprocessors:

- All the launches of a fatbin file with a given launch configuration can only fail or not fail - this is useful to understand whether something of wrong is happening on the GPU. If, when the same launch configuration is used to execute a fatbin file, we would have an hybrid of failed and not failed launches, then that would mean that there are problems about the bytes read and written from/to the GPU memories - launch failures due to try to read and/or write bytes from/to areas of the GPU memories not reserved to the fatbin file;

- The division between even and not even distribution of the GPU thread blocks to the GPU streaming multiprocessors is instead useful to test the conjecture that if the gigathread scheduler does not evenly distribute the GPU thread blocks to the GPU streaming multiprocessors - supposing an even distribution is not the only possible choice - then the gigathread scheduler is doing so for each execution of a fatbin file with a given launch configuration.

  Such conjecture - that we prove true - shows that the mistake of the gigathread scheduler is systematic - no analysis is instead done about the number of different types of not even distributions used by the gigathread scheduler.

Manual checks at the end of the extraction of this first group of architectural features has instead allowed to discover the following things:

- $T_1$) If the number of GPU thread blocks that we want assigned to each streaming multiprocessor requires a number of hardware registers that is smaller or equal than half of the number of hardware registers of a streaming multiprocessor - $\frac{2^{15}}{2}$ - and the number of GPU thread blocks launched is equal to 2 times the number of streaming multiprocessors then the

gigathread scheduler always assigns in a not even way the GPU thread blocks to the streaming multiprocessors;

- $T_2$) If the number of GPU thread blocks that we want assigned to each streaming multiprocessor requires a number of hardware registers that is greater than half of the the number of hardware registers of a streaming multiprocessor - $\frac{2^{15}}{2}$ - smaller or equal than the number of hardware registers of a streaming multiprocessor - $2^{15}$ - and the number of GPU thread blocks launched is equal to 2 times the number of streaming multiprocessors then the gigathread scheduler always assigns in an even way the GPU thread blocks to the streaming multiprocessors.

Also whether we are using only 24 launch configurations, instead of the possible many - 2.5 - $T_1$ experimentally proves a systematic not even distribution of the GPU thread blocks to the streaming multiprocessors by part of the gigathread scheduler - if the gigathread scheduler has such choice - and so we can not exclude this can not happen also for other launch configurations different from the 24 used. A not even distribution of the GPU thread blocks to the streaming multiprocessors that creates load unbalancing on the GPU can make useless any other thing done to optimize the execution of the ELF code of the B part of a fatbin file.

Without considering the type of not even GPU thread block distribution to the streaming multiprocessors because the result would be hardly generalizable considering we used only 24 launch configurations per fatbin file instead of the possible many, the main point, learned from $T_1$, is that it is always necessary to generate fatbin files in such way that knowing the number of GPU thread blocks that we want per streaming multiprocessor - number that has always to be less than 9 because a maximum of 8 GPU thread blocks can be resident in a streaming multiprocessor - the total number of hardware registers required by the number of GPU thread blocks that we want per streaming multiprocessor has always to be greater than half of the number of hardware registers of each streaming multiprocessor and smaller or equal than the number of hardware registers of each streaming multiprocessor. In this way we force the gigathread scheduler to assign the GPU thread blocks in an even way to the streaming multiprocessors because we do not leave to the gigathread scheduler any other possible choice.

The second group of global GPU assignment and scheduling architectural features is therefore quantified. The output data, from each execution, of the for loop, of each couple ( fatbin file , launch configuration ), are processed to calculate several types of time differences about the scheduling times of the warps on the GPU. The following conjectures were done before the beginning of this quantification phase:

- a) It could be that the warps can not be scheduled at any possible clock cycle - for example for warp management overhead, warp scheduler limitations, impossibility of warp scheduling 1) because the warp is waiting some results from previous ELF instructions or 2) because the warp is waiting some data for bandwidths or latencies memory problems - this last case is however impossible considering the structure of the B parts of the fatbin files used;

- b) Because each streaming multiprocessor has only 2 warp schedulers, if in a streaming multiprocessor there are more than 2 resident warps, all the resident warps can not be scheduled to execute the same ELF instruction at the same clock cycle and so the warps will execute the ELF instruction in different clock cycles;

- c) It is hard to believe that in the very simple case of only 2 resident warps per streaming multiprocessor, all the warps on the GPU start to execute the for loop at the same clock cycle, but also supposing this true, the number of warps per streaming multiprocessor has probably to be greater than 2, to get locally in a streaming multiprocessor the theoretical streaming multiprocessor best average performance per clock cycle.

If the conjectures a), b) and c) are true then there is an instant $t_1$, when a warp or a subset of all the resident warps on the GPU start to execute the first instruction of the for loop before all the others - let us call this warp or this subset of warps the leading warp or the leading subsets of warps - and an instant $t_2$, when a warp or a subset of all the resident warps on the GPU start to execute the first instruction of the for loop after all the others - let us call this warp or this subset of warps the last warp or the last subset of warps. Let us also define $(t_2 - t_1)$ the starting time difference in number of clock cycles. In the same way an analog reasoning can be done for the execution of the last ELF instruction of the for loop only that in this case the time difference takes the name of ending time difference in number of clock cycles.

So we have more resident warps in the GPU and all the warps can not execute the same ELF instruction at the same time. One or more warps execute first of all the others such ELF instruction. One or more warps execute after all the others such ELF instruction. The difference between these two moments is a time difference in number of clock cycles. The starting time difference helps to understand the scheduling variability at the beginning of the for loop. The ending time difference helps to understand the scheduling variability at the end of the for loop. Each one of these two time differences can be seen as a time window, time window that is necessary to wait to see all the resident warps, in the GPU, pass on a given ELF instruction after that such ELF instruction has been reached for the first time by a warp or a subset of warps.

The absolute difference, in number of clock cycles, between the ending and starting time differences, allows to understand how the warps are advancing. If the minimum number of clocks cycles, required to execute the local work on each streaming multiprocessor, is much greater than the starting and ending time differences, while the absolute difference between the starting and ending time differences is small, then we can say that from the moment when the last subset of warps start to execute the first ELF instruction of the for loop to the moment when the first subset of warps finish to execute the last ELF instruction of the for loop, the warp schedulers of the streaming multiprocessors make advance together all the warps. Knowing for which cases this is true is useful a) for the analysis of the ELF code of the B part of a fatbin file, b) for the eventual modifications of the ELF code of the B part of a fatbin file and c) for the generation of the launch configurations to use to execute a fatbin file.

At the end of the quantification of the global GPU assignment and scheduling architectural features several tables are calculated for each instruction configuration. As said previously the rows represent the fatbin files constructed for the instruction configuration - one fatbin file for each one of the dependence distances of the dependence type of the instruction configuration - while the columns represent the launch configurations used to execute the fatbin files. For each instruction configuration the tables can be classified in the following numbered groups:

- 1) The tables are the maximum absolute time differences between maximum ending time and maximum starting time differences of the couples ( fatbin file , launch configuration ), the maximum ending time differences of the couples ( fatbin file , launch configuration ), the maximum starting time differences of the couples ( fatbin file , launch configuration ), the

minimum absolute time differences between maximum ending time and maximum starting time differences of the couples ( fatbin file , launch configuration ), the minimum ending time differences of the couples ( fatbin file , launch configuration ) and the minimum starting time differences of the couples ( fatbin file , launch configuration );

- 2) The tables have the same name of the previous tables but are calculated only considering the couples ( fatbin file , launch configuration ) for which the gigathread scheduler has evenly distributed the GPU thread blocks to the streaming multiprocessors;
- 3) The tables have the same name of the previous tables but are calculated only considering the couples ( fatbin file , launch configuration ) for which the gigathread scheduler has not evenly distributed the GPU thread blocks to the streaming multiprocessors;

A further set of 3 summary tables is calculated for each instruction configuration. One table is calculated considering all the launch configurations with not failed launches, one table is calculated considering all the launch configurations with not failed launches and with an even GPU thread block distribution to the streaming multiprocessors and one table is calculated considering all the launch configurations with not failed launches and with a not even GPU thread block distribution to the streaming multiprocessors. Each one of these tables has 6 rows, one for each one of the possible 6 previous time differences, time differences this time calculated considering at the same time all the couples ( fatbin file , launch configuration ) of the instruction configuration. Each one of these tables has 3 columns: the first column is for the fatbin file identifier associated to the time difference at the given row, the second column is for the launch configuration associated to the time difference at the given row and the third column is for the value of the time difference considered at the given row.

At end of this phase, analog, time difference tables, in number and structure, to the time difference tables of the groups 1), 2), 3) and the set of 3 summary tables, are calculated considering at the same time all the couples ( fatbin file , launch configuration ) of all the instruction configurations. The only difference is for the tables of the set of 3 summary tables, tables that now have 4 columns instead of 3, the fourth containing the identifier of the instruction configurations associated to the time difference at the given row.

All the tables are useful to study the warp scheduler variabilities and to determine whether the byte transfers, among the different GPU memories, could slow down the execution, of the ELF code, of the B part, of the fatbin file, of a couple ( fatbin file , launch configuration ) - different from the couples used in this chapter - when the fatbin file is executed using the launch configuration of the couple.

### 7.6.2 Local Streaming Multiprocessor PTX and ELF Architectural Features

The first group of local streaming multiprocessor PTX and ELF architectural features is composed by the same time differences of the second group of global GPU assignment and scheduling architectural features. The group 1) of tables is calculated per instruction configuration, the group 2) is calculated considering only the couples ( fatbin file , launch configuration ) with an even number of warps on the only streaming multiprocessor used and the group 3) is calculated considering only the couples ( fatbin file , launch configuration ) with an odd number of warps on the only streaming multiprocessor used.

The further set of 3 summary tables is calculated for each instruction configuration only that this time one table is calculated considering all the launch configurations with not failed launches, one table is calculated considering all the couples ( fatbin file , launch configuration ) with not failed launches and with an even number of warps on the only streaming multiprocessor used and one table is calculated considering all the couples ( fatbin file , launch configuration ) with not failed launches and with an odd number of warps on the only streaming multiprocessor used. As in the previous case each one of these tables has 6 rows, one for each one of the possible 6 previous time differences, time differences this time calculated considering at the same time all the couples ( fatbin file , launch configuration ) of the instruction configuration. As in the previous case each one of these tables has 3 columns: the first column is for the fatbin file identifier associated to the time difference at the given row, the second column is for the launch configuration associated to the time difference at the given row and the third column is for the value of the time difference considered at the given row.

Also at end of this phase, analog, time difference tables, in number and structure to the time difference tables of the groups 1), 2), 3) and the set of 3 summary tables, are calculated considering at the same time all the couples ( fatbin file , launch configuration ) of all the instruction configurations. The only difference is, this time too, for the table of the set of 3 summary tables, tables that now have 4 columns instead of 3, the fourth containing the identifier of the instruction configurations associated to the time difference at the given row.

All these tables are useful - as in the previous case - to study the warp scheduler variabilities and to determine whether the byte transfers, among the different GPU memories, could slow down the execution, of the ELF code, of the B part, of the fatbin file, of a couple ( fatbin file , launch configuration ) - different from the couples used in this chapter - when the fatbin file is executed using the launch configuration of the couple.

The second group of local streaming multiprocessor PTX and ELF architectural features is therefore verified and/or quantified. Such group is composed by 1) the real instruction configuration streaming multiprocessor best average performance per clock cycle of the PTX and ELF instruction configurations, 2) the verification of whether it is possible to get load unbalancing for the warp scheduling in a streaming multiprocessor if the gigathread scheduler has evenly distributed the GPU thread blocks to the streaming multiprocessors, 3) the scheduling waiting times of the ELF instruction configurations, 4) the write-read and read-read dependence waiting times of the ELF instruction configurations, 5) the verification of the existence of an overhead time for the management of the warps, the verification of its not linear increase for a linear increase in the number of resident warps in a streaming multiprocessor, the determination of the shape of a function able to express it and 6) the minimum number of resident warps necessary in a streaming multiprocessor to get the real ELF instruction configuration streaming multiprocessor best average performance per clock cycle of each ELF instruction configuration for each dependence distance.

As the reader can notice the last four local streaming multiprocessor architectural features are verified and/or quantified only for the ELF instruction configurations. This happens because a) the GPU is going to execute ELF instruction configurations and not PTX instruction configurations, b) each PTX instruction configuration is transformed in one or more consecutive ELF instruction configurations and c) there are specific dependences among the ELF registers used by the consecutive ELF instruction configurations. The streaming multiprocessor hardware design together at the previous a), b) and c) determine the real PTX instruction configuration streaming multiprocessor best average performance per clock cycle while the verification or quantification, of the last

four local streaming multiprocessor architectural features, of each ELF instruction configuration, is useful for the optimization and the analysis of the ELF code of the B part of the fatbin file of a couple ( fatbin file , launch configuration ) - different from the couples used in this chapter - when the fatbin file is executed using the launch configuration of the couple.

*Real Instruction Configuration Streaming Multiprocessor Best Average Performance Per Clock Cycle:* We start calculating the total number of instruction configurations that has to be executed inside the for loops of each execution of each couple ( fatbin file , launch configuration ). Next we calculate the average number of instruction configurations executed per clock cycle by the single streaming multiprocessor during each launch of each couple ( fatbin file , launch configuration ) - the average is calculated because during the execution of a fatbin file there is no way to know which and how many instruction configurations are executed in each specific clock cycle. Considering the averages calculated for each execution of each couple ( fatbin file , launch configuration ), the average of the averages is calculated for each couple ( fatbin file , launch configuration ). Considering all the averages of the averages of all the couples ( fatbin file , launch configuration ) of an instruction configuration, the real instruction configuration streaming multiprocessor best average performance per clock cycle is the maximum, of the averages of the averages, rounded to the nearest bigger integer - a streaming multiprocessor has to have an integer number of function units to execute the instruction configuration, a maximum not integer is due at the presence of noise due to the warp scheduler variabilities and the three instructions inside each for loop used to check whether it is necessary to iterate on the for loop.

A check is executed on the real instruction configuration streaming multiprocessor best average performance per clock cycle achieved in this way. Such check is useful a) to get the guarantee that the real instruction configuration streaming multiprocessor best average performance per clock cycle is the true real instruction configuration streaming multiprocessor best average performance per clock cycle and b) to get the guarantee that the real instruction configuration streaming multiprocessor best average performance per clock cycle is the real instruction configuration streaming multiprocessor peak performance achievable in a clock cycle.

The check is executed separately for each instruction configuration. Here the check steps: 1) calculation, considering all the couples ( fatbin file , launch configuration ), of the maximum average of the averages, 2) selection of the fatbin files of the couples ( fatbin file , launch configuration ) with an average of the averages not smaller than 95% of the maximum average of the averages calculated at point 1), 3) check that the multiset so obtained has at least two different fatbin files.

Going more in detail on why is useful such check, we can say that because we do not know the GPU hardware limitations due to the GPU hardware design, we can not *a priori* know whether a) the number of fatbin files that is possible to generate for the instruction configuration and b) the number and type of launch configurations used to run these fatbin files, are sufficient to get the true real instruction configuration streaming multiprocessor best average performance per clock cycle - this because the real instruction configuration streaming multiprocessor best average performance per clock cycle of an instruction configuration could effectively depend on several different things.

The following are conjectures that we therefore need to consider and to verify during the quantification phase of the real instruction configuration streaming multiprocessor best average performance per clock cycle - and so of the real instruction configuration streaming multiprocessor peak performance achievable in a clock cycle - of each instruction configuration:

- 1) The real instruction configuration streaming multiprocessor best average performance per

clock cycle could depend on the instruction configuration. Different instruction configurations could have different real instruction configuration streaming multiprocessor best average performance per clock cycle in the case - for example - different instruction configurations require for their execution different function units;

- 2) A warp scheduler can not probably schedule a warp with 0 overhead clock cycles. After the scheduling of a warp it is more probable that the warp schedulers can not schedule such warp again before of a given number of clock cycles - scheduling waiting time - but not for the write-read and read-read dependence waiting times but a) for intrinsic hardware limitations of the warp scheduler or b) because the particular instruction configuration that has to be executed requires specific hardware resources - hardware paths, special registers shared among GPU threads, etc. - that has to be shared or used with a lower frequency compared to the frequency of other hardware resources;

- 3) The real instruction configuration streaming multiprocessor best average performance per clock cycle of an instruction configuration could depend on the write-read and read-read dependence waiting times of the instruction configuration;

- 4) There could be an overhead time for the management of the warps, its increasing could be not linear for a linear increase of the number of resident warps in a streaming multiprocessor and therefore there could be cases where the overhead time for the management of the warps could be the main factor in the determination of the real instruction configuration streaming multiprocessor best average performance per clock cycle;

- 5) The possible presence of not disclosed hardware shared resources between the 2 groups of 16 CUDA cores in a streaming multiprocessor could give a real instruction configuration streaming multiprocessor best average performance per clock cycle smaller than the theoretical instruction configuration streaming multiprocessor best average performance per clock cycle.

We can choose the instruction configuration and the number of resident warps in a streaming multiprocessor but we can not change the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps. Because we do not already know the scheduling waiting times, the dependence waiting times and the overhead times due to the management of the warps, we can not know *a priori* how the scheduling waiting times, the dependence waiting times and the overhead time for to the management of the warps are going to influence the real instruction configuration streaming multiprocessor best average performance per clock cycle of each instruction configuration. To avoid to believe that we have got the real instruction configuration streaming multiprocessor best average performance per clock cycle, when instead we have been limited by the scheduling waiting times, by the dependence waiting times or by the overhead time for the management of the warps, we need to get more fatbin files with one or more averages of the averages not smaller than 95% of the maximum average of the averages calculated considering all the couples ( fatbin file , launch configuration ) of the instruction configuration.

All the instruction configurations, considered in the extraction phase, satisfy this check, so we can safely say that we have determined the real instruction configuration streaming multiprocessor best average performance per clock cycle of each instruction configuration considered.

Instruction configurations not considered in the extraction phase are for example the load and store instructions. The load and store instructions are not considered in the extraction phase

because it is hard, whether not impossible, to time their execution without meeting great challenges in the proofs necessary to give an *a priori* guarantee that their execution times are not slowed down by the byte transfers among different GPU memories. However, because each streaming multiprocessor has only a group of 16 load and store units, we can safely assume that not more than 16 load or store ELF instruction configurations can be executed per clock cycle by a streaming multiprocessor.

PTX instruction configurations transformed by nvcc in only one ELF instruction configuration executed by the 2 groups of 16 CUDA cores and single ELF instruction configurations executed by the 2 groups of 16 CUDA cores, both with a real instruction configuration streaming multiprocessor best average performance per clock cycle smaller than 32, indicate the presence of not disclosed hardware resources shared between the 2 groups of 16 CUDA cores in each streaming multiprocessor - this verifies as true our conjecture in 4.5.

With the check at the beginning of the subsection and with what we know about the GPU hardware design, the real instruction configuration streaming multiprocessor best average performance per clock cycle of each instruction configuration is the real instruction configuration streaming multiprocessor peak performance achievable in a clock cycle of each instruction configuration because if we consider instruction configurations executed by the 2 groups of 16 CUDA cores, a) we get a real instruction configuration streaming multiprocessor best average performance equal to 32 whether there are not not disclosed hardware resources, shared between the 2 groups of 16 CUDA cores, for the parallel execution of the instruction configuration and c) we get a real instruction configuration streaming multiprocessor best average performance smaller than 32 whether there are not disclosed hardware resources, shared between the 2 groups of 16 CUDA cores, for the parallel execution of the instruction configuration, while if we consider instruction configurations executed by the group of 4 special function units, we get a real instruction configuration streaming multiprocessor best average performance equal to 4 for the execution of the instruction configuration.

Knowing the real instruction configuration streaming multiprocessor best average performance per clock cycle of each instruction configuration a) we can take one or more instruction configurations with a real instruction configuration streaming multiprocessor best average performance per clock cycle equal to 32 or 16 - in this way we are sure that the instruction configurations are executed by the 2 groups of 16 CUDA cores, this because we do not use load and store instruction configurations and the group of special function units has only 4 special function units and so the instruction configurations executed using the group of 4 special function units can not have a real instruction configuration streaming multiprocessor best average performance per clock cycle greater than 4 - b) we can mix them with other instruction configurations with a streaming multiprocessor best average performance per clock cycle equal to 16 - this to be sure that these instruction configurations too are executed by the 2 groups of 16 CUDA cores and that in the fatbin files that we are going to generate there are at least some instruction configurations with a real instruction streaming multiprocessor best average performance per clock cycle smaller than the theoretical streaming multiprocessor best average performance per clock cycle that is 32 - and c) using the procedures in 6.7, we generate some fatbin files using the same guidelines used in 7.4 for the generation of the fatbin files used for the quantification of the real instruction configuration streaming multiprocessor best average performance per clock cycle of the instruction configurations.

We therefore execute these fatbin files using launch configurations selected considering also the results of the next subsections, this to be sure, *a priori*, that the execution of the for loops of the B parts of the fatbin files are not slowed down by a) the load unbalancing, for the warp scheduling,

in the single streaming multiprocessor used, b) the scheduling waiting times, c) the write-read and read-read dependence waiting times and d) the overhead time for the management of the warps. At the end of each execution we calculate the real ELF code streaming multiprocessor average performance per clock cycle of the execution.

Checking the real ELF code streaming multiprocessor average performance per clock cycle of an execution and finding it equal to the theoretical streaming multiprocessor best average performance per clock cycle we prove being true our conjecture in 4.6 and so we know that fatbin files with ELF instructions, in their B parts, with a real ELF instruction best average performance smaller than the theoretical streaming multiprocessor best average performance per clock cycle - 32 - can however get a real ELF code streaming multiprocessor best average performance per clock cycle equal to the theoretical streaming multiprocessor best average performance per clock cycle.

*Possibility to Get Load Unbalancing for the Warp Scheduling in a Streaming Multiprocessor if the Gigathread Scheduler Has evenly Distributed the GPU Thread Blocks to the Streaming Multiprocessors:* If this is true and there is no way to get load balancing for the warp scheduling in a streaming multiprocessor if the gigathread scheduler has evenly distributed the GPU thread blocks on the streaming multiprocessors then the load unbalancing for the warp scheduling in a streaming multiprocessor can have a very bad impact on the execution time of an ELF code.

To verify this conjecture, for each instruction configuration, all its couples ( fatbin file , launch configuration ) are divided in two groups: the group of the couples with launch configurations with an even number of resident warps in the single streaming multiprocessor used and the group of the couples with launch configurations with an odd number of resident warps in the single streaming multiprocessor used. For each group, the couples, with a maximum average of the averages that is not smaller than 95% of the real instruction configuration streaming multiprocessor best average performance per clock cycle, are selected. If the number of couples, selected by the group with launch configurations with an even number of resident warps in the single streaming multiprocessor used, is always much greater than the number of couples selected by the group with launch configurations with an odd number of resident warps in the single streaming multiprocessor used, then we have load unbalancing for the warp scheduling inside the streaming multiprocessor. Manual checks says that this is always the case.

Understanding that each warp in a streaming multiprocessor has to be identifiable to be managed, to try to explain why it is possible to get load unbalancing for the warp scheduling inside a streaming multiprocessor, if the gigathread scheduler has evenly distributed the GPU thread blocks to the streaming multiprocessors, we formulated the conjecture that maybe one warp scheduler could only manage warps with an "even" identifier while the other warp scheduler could only manage warps with an "odd" identifier.

The verification of the conjecture was done executing the following procedure: a) calculation of the warp workloads - the workloads are always equal among warps because the ELF code in the for loop of the B part of each fatbin file is without divergences and so all the GPU threads execute always the same number of instruction configurations - b) determination, supposing the conjecture true, of the theoretical maximum efficiencies possible for the couples ( for loop in the B part of the fatbin file , launch configuration ) with launch configurations with an odd number of resident warps in the single streaming multiprocessor used and c) comparison of the theoretical maximum efficiencies to the real efficiencies. Because the real efficiencies were however always much greater than the correspondent theoretical maximum efficiencies the conjecture has been proved being false.

Manual checks evidence that, for each instruction configuration, the couples of the couples of

the type ( ( fatbin file X , launch configuration Y ) , ( fatbin file X , launch configuration Y + 1 ) ), with X > 2 and Y even, have starting time differences of the same order of magnitude while the ending time differences of the couples ( fatbin file X , launch configuration Y + 1 ), with an odd number of resident warps - Y + 1 - in the single streaming multiprocessor used, are of 1, 2 or 3 orders of magnitude greater than the ending time differences of the correspondent couples ( fatbin file X , launch configuration Y ), with an even number of resident warps - Y - in the single streaming multiprocessor used.

Considering a) the execution time of each execution of each couple ( fatbin file , launch configuration ) with an odd number of resident warps in the single streaming multiprocessor used and b) the ending time differences, we can say that 1) the lost in efficiency is due to the ending time differences and that 2) the starting and ending time differences show that for unknown reasons, when an odd number of warps is resident in a streaming multiprocessor, a situation of load unbalancing for the warp scheduling is created inside the streaming multiprocessor after the starting of the execution of the for loops in the B parts of the fatbin files and that such situation of load unbalancing for the warp scheduling survives till to the end of the execution of the for loops in the B parts of the fatbin files.

Also whether we have no way to know the causes generating the load unbalancing, the results show that is always important to make sure that the total number of warps that we want assigned to each streaming multiprocessor is always even - the total number of warps is given by the number of GPU thread blocks that we want assigned to each streaming multiprocessor times the number of warps of each GPU thread block.

*Scheduling Waiting Times of the ELF Instruction Configurations:* Each scheduling waiting time of each ELF instruction configuration is determined with the following procedure: 1) calculation of the maximum average of the average throughputs considering at the same time all the couples ( fatbin file X , launch configuration 1 ) - 1 <= X <= number of fatbin files generated for the the ELF instruction configuration while the launch configuration 1 has only 1 GPU thread block with only 1 warp - 2) check that more couples ( fatbin file X , launch configuration 1 ) have an absolute value of the difference, between their averages of the average throughputs and the maximum average of the average throughputs calculated at point 1), not bigger than 0.002 and 3) determination of the scheduling waiting time as the average number of clock cycles, rounded at the nearest smaller integer, necessary to execute an ELF instruction configuration of the couple ( fatbin file X , launch configuration 1 ) where X is the smallest number among the couples ( fatbin file X , launch configuration 1 ) satisfying the check at point 2).

Step 2) is necessary to have the guarantee that a) the average of the average throughputs of at least some couples ( fatbin file X , launch configuration 1 ) is limited by the scheduling waiting time and not by the dependence waiting time and b) that more couples ( fatbin file X , launch configuration 1 ) are limited by the scheduling waiting time because otherwise in presence of only a couple we can not be sure its average of the average throughputs was limited by the scheduling waiting time and so we can not be sure to have determined the real scheduling waiting time.

The average number of clock cycles is rounded to the nearest smaller integer because a streaming multiprocessor can only have an integer number of function units and the starting and ending time differences plus the three ELF instructions in each for loop necessary to iterate on it are introducing a small noise. In the next subsection however we will see that the average number of clock cycles necessary to execute an ELF instruction configuration is not always the real number of clock cycles necessary to execute an ELF instruction configuration but it will be clear that the average number

of clock cycles necessary to execute an ELF instruction configuration, selected at the end of the procedure above indicated, independently of the fact that it is or not the real number of clock cycles necessary to execute an ELF instruction configuration, is the scheduling waiting time.

If an ELF instruction configuration a) is executed by the 2 groups of 16 CUDA cores, b) has a real ELF instruction configuration streaming multiprocessor best average performance per clock cycle equal to 32 and c) we are not limited by the dependence waiting times and the overhead time for the management of the warps, then, with a number of warps equal to the scheduling waiting time - this because at each warp scheduler clock cycle the 2 warp schedulers in a streaming multiprocessor can schedule at maximum 2 warps but each warp is going to requires for its execution 2 function unit clock cycles and the clock frequency of the warp schedulers is half of the clock frequency of the function units - it should be possible to get an average throughput that is not smaller than 95% of the maximum average of the average throughputs calculated considering all the couples ( fatbin file , launch configuration ) of the ELF instruction configuration and this is effectively always the case.

*Write-Read and Read-Read Dependence Waiting Times of the ELF Instruction Configurations:* The write-read and read-read dependence waiting times of an ELF instruction configuration can be determined only in those cases where the scheduling waiting time is determined by the couples ( fatbin file X , launch configuration 1 ) with $X > 1$, thing indicating that for the couple ( fatbin file 1 , launch configuration 1 ) the executions of the ELF instruction configuration are limited by the dependence waiting time and not by the scheduling waiting time.

All the ELF instruction configurations in each fatbin file consider only one of the two dependence types of interest - write-read or read-read. To quantify the dependence waiting times we consider the average number of clock cycles necessary to execute an ELF instruction configuration, average that in some cases could be different from the real number of clock cycles necessary to execute an ELF instruction configuration, as we will see in this subsection.

For the cases where is possible to calculate the write-read and read-read dependence waiting times, the write-read and read-read dependence waiting times are equal to the average number of clock cycles necessary to execute an ELF instruction configuration of the couple ( fatbin file 1 , launch configuration 1 ), average rounded at the smaller nearest integer because a streaming multiprocessor can only have an integer number of function units and the starting and ending time differences plus the three ELF instructions in each for loop necessary to iterate on it are introducing a small noise.

In each code considering the read-read dependences there are also the write-write dependences, write-write dependences a) present between the results of each couple of two equal ELF instruction configurations - two ELF instruction configurations using the same ELF registers in the same roles - and b) not quantified because not interesting. The quantification of the read-read dependence waiting times is however not influenced by the write-write dependence waiting times. If a warp scheduler could schedule the warps in such a way to request the writing of the result of an ELF instruction configuration before the ELF register can be overwritten then some queues would be necessary, thing not probable considering a) the necessary management, b) the necessary die area and c) the fact that for the ELF codes used for the quantifications such queues would contain millions of results to write. The GPU architecture has therefore to be design in such a way that this is considered in the scheduling waiting times and so a warp scheduler can not schedule a warp in such a way that the request for the writing of the result in a ELF register can not be satisfied because a previous writing in the same ELF register has still to terminate. Considering this, the

quantification of the read-read dependence waiting times can not therefore be influenced by the write-write dependence waiting times.

Being the quantification of the read-read dependence waiting times not influenced by the write-write dependence waiting times and having experimentally proved that, for each ELF instruction configuration, the read-read dependence waiting time is smaller than the correspondent write-read dependence waiting time of the ELF instruction configuration and that, for some ELF instruction configurations, the read-read dependence waiting time is smaller than the correspondent scheduling waiting time of the ELF instruction configuration, we can say that, for some ELF instruction configurations, after a warp scheduler has seen to pass a quantity of time equivalent to that of a scheduling waiting time of the ELF instruction configuration, the warp scheduler is able to schedule again the same warp well before the ELF register, where maybe has still to be written the result of a previous equal ELF instruction configuration, can be overwritten again - however if the warp scheduler schedules again the warp this happens because the over writing of the ELF register will be for sure possible when necessary, see why in the previous paragraph. For this reason, the average number of clock cycles necessary to execute an ELF instruction configuration of the couple ( fatbin file 1 , launch configuration 1 ), when we consider read-read dependences, is not the precise number of clock cycles necessary to execute an ELF instruction configuration, but is smaller than this, and in some cases correspond to the scheduling waiting time.

Because we have 2 ELF instruction configurations - the ELF instruction configuration considering the write-read dependence and the ELF instruction configuration considering the read-read dependence - for each type of ELF instruction with a given execution mode - the normal, the conditional with guard set at true and the conditional with guard set at false - but both the 2 ELF instruction configurations consider the same type of ELF instruction given an execution mode, then the write-read dependence waiting time is instead the precise - and also average - number of clock cycles necessary to execute an ELF instruction, with the given execution mode, because a warp scheduler can not schedule a warp again before the result of a previous equal ELF instruction configuration can be read as operand. For the write-read dependences is however necessary to distinguish two cases:

- If, following the first procedure indicated to quantify the scheduling waiting time of an ELF instruction configuration, a couple ( fatbin file X , launch configuration 1 ), with $X > 1$, is selected, then the average number of clock cycles necessary to execute an ELF instruction configuration for the couple ( fatbin file 1 , launch configuration 1 ) is not determined by the scheduling waiting time of the ELF instruction configuration but by the dependence waiting time of the ELF instruction configuration. In this case the ELF instruction configuration has a write-read dependence waiting time greater than the scheduling waiting time of the ELF instruction configuration and the write-read dependence waiting time is equal to the average number of clock cycles, necessary to execute an ELF instruction configuration of the couple ( fatbin file 1 , launch configuration 1 ), rounded at the nearest smaller integer;

- If, following the first procedure indicated to quantify the scheduling waiting time of an ELF instruction configuration, a couple ( fatbin file X , launch configuration 1 ), with $X = 1$, is selected, then the average number of clock cycles necessary to execute an ELF instruction configuration for the couple ( fatbin file 1 , launch configuration 1 ) is not determined by the dependence waiting time of the ELF instruction configuration but by the scheduling waiting time of the ELF instruction configuration. In this case the ELF instruction configuration has

a write-read dependence waiting time smaller or equal than the scheduling waiting time of the ELF instruction configuration and it is not possible to determine the write-read dependence waiting time of the ELF instruction configuration but we can assume it equal to the scheduling waiting time of the ELF instruction configuration.

Write-read dependence waiting times smaller or equal than the scheduling waiting times happen for some ELF instruction configurations using not disclosed hardware resources. In PTX we can declare 64 bits PTX registers but the GF100 architecture has only 32 bits hardware registers. We also know a) that PTX instructions using 64 bits registers has to be executed using more than one ELF instruction - see results of the previous chapter - and b) that each ELF instruction has to use 32 bits hardware registers - [52, p. 12]. From a) and b) follow that carries and other partial results that can not be stored in the 32 bits hardware registers used in the ELF instructions necessary to execute a PTX instruction has temporary to be stored somewhere. Having stored the carries and the other partial results in hardware units different from the 32 bits hardware registers used in the ELF instructions necessary to execute a PTX instruction and being the ELF instructions necessary to execute the PTX instruction consecutive in the ELF code, it is therefore necessary for the GF100 architecture to be sure that it is not possible to schedule again the warp, to execute the next ELF instruction necessary to execute the PTX instruction, before the carries and the other partial results necessary to execute the next ELF instruction are written and can be read again.

Finally, also whether the ELF instruction configurations necessary to load and store data or results, of different sizes, to different GPU memories, have not been considered - this for the reasons explained in the part on the real instruction configurations streaming multiprocessor best average performance per clock cycle - we have planned some experiments that is our wish to carry on in future and that will give us an upper bound on the dependence waiting times of the operands of such ELF instruction configurations.

*Existence of an Overhead Time for the Management of the Warps, Verification of Its not Linear Increasing for a Linear Increase of the Number of Resident Warps in a Streaming Multiprocessor and Determination of the Shape of a Function able to Express It:* To verify the existence of an overhead time for the management of the warps we start considering the ELF instruction configurations 1) for which it was possible to determine the write-read dependence waiting time and 2) with a real ELF instruction configuration streaming multiprocessor best average performance per clock cycle equal to 32, checking that:

- a) The write-read dependence waiting time of each ELF instruction configuration divided 2 is greater than the corresponding scheduling waiting time of each ELF instruction configuration, this because being the real ELF instruction configuration streaming multiprocessor best average performance per clock cycle equal to 32, we are sure that the not disclosed hardware resources shared among the 2 groups of 16 CUDA cores in a streaming multiprocessor are not used for the execution of any of the ELF instruction configurations considered, and so to be sure of not being limited, in the next experiments, by the scheduling waiting time of the ELF instruction configurations, the only thing we need to check it is that the write-read dependence waiting time of each ELF instruction configuration divided 2 is greater than corresponding scheduling waiting time of each ELF instruction configuration, this because it is necessary that exactly 2 warps are scheduled at each warp scheduler clock cycle to get the real

ELF instruction configuration streaming multiprocessor best average performance per clock cycle of the ELF instruction configurations considered;

- b) Whether we get the real ELF instruction configuration streaming multiprocessor best average performance per clock cycle for the couple ( fatbin file 1 , launch configuration X ), where X is equal to the first even integer equal or greater than the write-read dependence waiting time - X is the number of warps in the single streaming multiprocessor used and we want it even to get load balancing inside the streaming multiprocessor, furthermore we use the write-read dependence waiting time to calculate X because at each warp scheduler clock cycle 2 warps have to be scheduled but each one of them requires 2 function unit clock cycles to be executed so a number of resident warps in a streaming multiprocessor equal to X is enough to be sure that the execution of a couple ( fatbin file 1 , launch configuration X ) is not slowed down by the write-read dependence waiting time of the ELF instruction configuration considered in the fatbin file.

Checks show that we do not get the real ELF instruction configuration streaming multiprocessor best average performance per clock cycle for the couples ( fatbin file 1 , launch configuration X ) with X equal to the first even integer equal or greater than the write-read dependence waiting time but that the real ELF instruction configuration streaming multiprocessor best average performance per clock cycle is got for the first time for the fatbin file 1 for an even number of warps Y greater than X - this experimentally proves the existence of an overhead time for the management of the warps.

Also whether moving from a number of warps equal to X to a number of warps equal to Y, using steps of 2 to get load balancing inside the streaming multiprocessor, we are not able to quantify the overhead time for the management of the warps, checking the values of the averages of the average throughputs, we understand that the rate of growth of the overhead time for the management of the warps is not linear with a linear increase of the number of warps in the single streaming multiprocessor used and we understand that also whether the overhead time for the management of the warps is the determinant factor for the average of the average throughputs, for the couples ( fatbin file 1 , launch configuration X <= Z <= Y ), the overhead time for the management of the warps has an influence exponentially decreasing moving from X to Y - X and Y depending on the ELF instruction configuration - with an null influence for a number of warps greater than Y.

At the same time, for ELF instruction configurations a) executed by the 2 groups of 16 CUDA cores, b) with an ELF instruction configuration streaming multiprocessor best average performance per clock cycle equal to 16 - thing showing that there are some not disclosed hardware resources shared among the 2 groups of 16 CUDA cores in a streaming multiprocessor and that such resources are been used for the execution of the ELF instruction configurations - c) for which it was not possible to determine the write-read dependence waiting time because their write-read dependence waiting times are smaller than their scheduling waiting times, this phenomenon does not happen for the first couple ( fatbin file 1 , launch configuration Z ) with Z equal to the scheduling waiting time of the ELF instruction configurations, if Z is enough "small". This experimentally prove that the effect of the overhead time for the management of the warps can be null if the number of warps is "small". For these types of ELF instructions, the overhead time is not going to be a problem even for a greater number of warps, in other words at the increase of the overhead time for the management of the warps, the number of warps a) is already enough big compared to the minimum number of warps necessary in a streaming multiprocessor to get the real ELF instruction configuration

streaming multiprocessor best average performance per clock cycle for the dependence distance 1 and b) always enough big compared to the value of the overhead time for the management of the warps, and so the overhead time for the management of the warps is never the limiting factor for this type of ELF instruction configurations, this independently of the growth rate of the overhead time for the management of the warps for such ELF instruction configurations.

Considering the previous two results, also whether we understand a) that the overhead time for the management of the warps is the limiting factor for some triplets ( ELF instruction configurations , dependence distance, number of resident warps in a streaming multiprocessor ) and b) that the shape of a function able to express the overhead time for the management of the warps could be a sigmoid with only positive values - starting with 0 for a number of warps equal to 0 and later becoming a constant C or however a set of values that approximate a constant C when the number of resident warps in a streaming multiprocessor is greater than Y, with Y and the values of the sigmoid like function however dependent on the ELF instruction configuration - we are not able to quantify the overhead time for the management of the warps for the triplets ( ELF instruction configuration , dependence distance, number of resident warps in a streaming multiprocessor ) and so, not being able to determine for some cases whether it is the scheduling waiting time, the dependence waiting time or the overhead time for the management of the warps the limiting factor for the average throughput that we get for the triplets ( ELF instruction configuration , dependence distance , number of resident warps in a streaming multiprocessor ), we need to store for each couple ( ELF instruction configuration , dependence distance ) the minimum number of resident warps necessary in a streaming multiprocessor to get an average throughput that is not smaller than 95% of the maximum average of the average throughputs calculated considering all the couples ( fatbin file , launch configuration ) of the ELF instruction configuration.

*Minimum Number of Resident Warps Necessary in a Streaming Multiprocessor to get the Real Instruction Configuration Streaming Multiprocessor Best Average Performance per Clock Cycle of Each ELF Instruction Configuration for Each Dependence Distance:* Considering that for some triplets ( ELF instruction configuration , dependence distance , number of resident warps in a streaming multiprocessor ) is not possible to determine whether it is the scheduling waiting time, the dependence waiting time or the overhead time for the management of the warps the limiting factor in the determination of the minimum number of resident warps necessary to get the real ELF instruction configuration streaming multiprocessor best average performance per clock cycle then it is important to store, for each couple ( ELF instruction configuration , dependence distance ), the minimum number of resident warps necessary in a streaming multiprocessor to get the real ELF instruction configuration streaming multiprocessor best average performance per clock cycle, this to be able to understand the minimum number of resident warps, necessary locally in each streaming multiprocessor used during the execution of a fatbin file, to avoid instruction pipeline stalls due to the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps.

## 7.7 Summary

In this chapter we have verified, understood and quantified the GPU hardware behaviors - due to the GPU hardware design - that could slow down the execution of the ELF code of the B part of a fatbin file. The main points to remember from this chapter are the following:

- When we execute a fatbin file we always choose launch configurations such that all the GPU thread blocks, we want to execute the fatbin file, are assigned, by the gigathread scheduler, to the streaming multiprocessors, at the beginning of the execution of the part of the fatbin file executed by the GPU. Doing this, the overhead due to the gigathread assignment - independently of what it is - is unique and paid only one time during the whole execution of the ELF code of the B part of the fatbin file, this because the GPU thread blocks will stay resident in the streaming multiprocessors till to the end of the execution of the ELF code of the B part of the fatbin file;

- A couple ( fatbin file , launch configuration ) has to be chosen in such a way that the total number of hardware registers required by the GPU threads of the GPU thread blocks we want assigned to each streaming multiprocessor is greater than half of the number of hardware registers of a streaming multiprocessor and smaller or equal than the number of hardware registers of a streaming multiprocessor. This is necessary because we have experimentally determined that otherwise the gigathread scheduler does not evenly distribute the GPU thread blocks to the streaming multiprocessors and this could make useless all the other efforts made to optimize the execution of the ELF code of the B part of a fatbin file;

- Also if the gigathread has evenly distributed the GPU thread blocks to the streaming multiprocessors the number of warps per GPU thread block times the number of GPU thread blocks that we want assigned to each streaming multiprocessor has to be even because we have experimentally determined that otherwise we do not get local load balancing for the warp scheduling in a streaming multiprocessor and this could have a very impact on the execution time of an ELF code;

- Not all the warps execute the same ELF instruction at the same time. There are some time differences, in number of clock cycles, that is necessary to wait from the moment when the leading warp or the leading subset of warps is scheduled to execute an ELF instruction to the moment when the last warp or the last subset of warps is scheduled to execute the same ELF instruction.

  We have experimentally determined such time differences, at the global GPU level and at the local streaming multiprocessor level, for the different possible cases of the triplets of values ( number of ELF registers per GPU thread , number of warps per GPU thread block , number of GPU thread blocks ), because they are useful to understand whether the byte transfers, among the different GPU memories, could slow down the execution of the ELF code of the B part of a fatbin file;

- The real instruction configuration streaming multiprocessor best average performance for each instruction configuration a) has been determined, b) correspond to the real instruction configuration streaming multiprocessor peak performance achievable in a clock cycle and c) has allowed to discover the presence of not disclosed hardware resources shared between the 2 groups of 16 CUDA cores in each streaming multiprocessor - for some instruction configurations, the not disclosed shared hardware resources do not allow to get the theoretical instruction configuration streaming multiprocessor best average performance per clock cycle;

- We have determined the scheduling waiting time of each ELF instruction configuration. The scheduling waiting time is the minimum number of clock cycles that has to pass before a warp

scheduler can consider to schedule the same warp again. The scheduling waiting times allow to determine, in the case they are the limiting factor, the minimum number of warps that has to be locally resident in a streaming multiprocessor to avoid stalls - due to the scheduling waiting times - in the instruction pipelines of the streaming multiprocessor;

- The write-read and the read-read dependence waiting times have been determined of each ELF instruction configuration. The write-read dependence waiting times are equal to the numbers of clock cycles that is necessary to wait before to be able to read a data from an ELF register previously written while the read-read dependence waiting times are equal to the numbers of clock cycles that is necessary to wait before to be able to read a data from an ELF register previously read.

  The write-read and the read-read dependence waiting times are useful to determine, in the case they are the limiting factor, the minimum number of warps that has to be locally resident in a streaming multiprocessor to avoid stall - due to the write-read and the read-read dependence waiting times - in the instruction pipelines of the streaming multiprocessor. The write-read dependence waiting time of an ELF instruction configuration is also the number of clock cycles necessary to execute the ELF instruction configuration;

- The presence of an overhead time for the management of the warps has been validated as its not linear increase for a linear increase of the number of resident warps in a streaming multiprocessor. The shape of a function, able to express it, is a sigmoid, with only positive values, but an accurate quantification, of the overhead time for the management of the warps, for all the possible triplets ( ELF instruction configurations , dependence distance , number of warps ), was not possible.

  The overhead time for the management of the warps is important because its effects, also whether not quantifiable, allow to determine, in the case the overhead time for the management of the warps is the limiting factor, the minimum number of warps that has to be locally resident in a streaming multiprocessor to avoid stall - due to the overhead time for the management of the warps - in the instruction pipelines of the streaming multiprocessor;

- The minimum number of resident warps necessary in a streaming multiprocessor to get the real instruction configuration streaming multiprocessor best average performance per clock cycle of each couple ( ELF instruction configuration , dependence distance ) has been determined, considering concurrently a) the scheduling waiting time of the ELF instruction configuration, b) the dependence waiting time of the ELF instruction configuration and c) the overhead time for the management of the warps, overcoming in this way the problem to determine who of them is the limiting factor in each one of the single cases.

In the next chapter we describe the procedures to execute on an original fatbin file $F_{f_i}$. The procedures are useful to increase the probability to get a greater lower bound on the real ELF code efficiency and imply a) the generation of several different fatbin files equivalent to the original fatbin file $F_{f_i}$ and b) the generation of the sets of potential launch configurations that can be used to execute the fatbin files generated - one set of launch configurations for each one of the fatbin files generated.

# Chapter 8

# Modifications, Launch Configurations and Transformations

## 8.1 Introduction

In the previous chapter we have discovered, understood and quantified not disclosed GPU behaviors due to the GPU hardware design, things useful to understand how to transform a fatbin file - the goal of this chapter - to increase the probability to get a greater lower bound on its real ELF code efficiency.

We start describing a set of procedures that we use to modify single fatbin files. Next we explain the procedure to generate the set of launch configurations that has to be considered when we analyze a fatbin file. Finally, we explain how we transform a fatbin file, before to analyze it, to increase the probability to get a greater lower bound on its real ELF code efficiency.

## 8.2 Procedures to Modify Single Fatbin Files

To try to increase the probability to get a greater lower bound on the real ELF code efficiency of a fatbin file is necessary to modify the fatbin file. We find useful to give here the definitions of some procedures that we will use in the next sections to modify single fatbin files.

The first procedure generates a fatbin file for each logically correct permutation of the ELF instructions in the B part of a fatbin file. The second procedure modifies a fatbin file to give the guarantee that, when we use it with the wanted launch configuration, the gigathread scheduler is going to evenly distribute the GPU thread blocks to the streaming multiprocessors. The third procedure modifies the reading and/or writing mechanisms used by a fatbin file to allow us to run it with a greater number of launch configurations and to consider or not the possibilities of reuse of the local data.

### 8.2.1 Logically Correct Permutations of the ELF Instructions

Suppose that 1) we have a fatbin file $F_i$, 2) we want to determine all the possible logically correct permutations of its ELF instructions in its B part - we check the dependences between the ELF registers used in the ELF instructions and generate all the possible logically correct orders of

precedence among ELF instructions - and 3) we want to generate for each one of the possible orders a new fatbin file $F_o$ using the same number and type of hardware registers of the fatbin file $F_i$.

Let us call the procedure that makes these things possible procedure A - $P_A$. $P_A$ is feasible because a) we can analyze the interpretation text file of the fatbin file $F_i$ - 6.2 - to determine the number and type of ELF registers of the fatbin file, b) we can use the procedure described in 6.6 to generate fatbin files with a number and type of resources - ELF registers and ELF instructions used in their B parts - equal to the number and type of resources of the fatbin file $F_i$ and c) we can use the procedure described in 6.7 to overwrite each one of the fatbin files generated, this to get each one of the fatbin files $F_o$.

### 8.2.2 Even Distribution of the GPU Thread Blocks to the Streaming Multiprocessors

Suppose 1) we have a fatbin file $F_i$, 2) we want to execute the fatbin file $F_i$ using a specific launch configuration and 3) because the fatbin file $F_i$ has a number of ELF registers equal to $n_{er}^{F_i}$, we have not the guarantee that the gigathread scheduler is going to evenly distribute to the streaming multiprocessors the GPU thread blocks we want to use for the execution of the fatbin file.

Using procedure B - $P_B$ - we take in input the fatbin file $F_i$ and generate as output a fatbin file $F_o$ with, in its B part, the same number, type and order of the ELF instructions and the same dependences among ELF registers of those in the B part of the fatbin file $F_i$. $F_o$ however has a total number of ELF registers $n_{er}^{F_o}$ greater than $n_{er}^{F_i}$.

We choose $n_{er}^{F_o}$ in such a way that we get the guarantee that, when we execute the fatbin file $F_o$ using the specific launch configuration we wanted to use to execute the fatbin file $F_i$, the gigathread scheduler is going to evenly distribute to the streaming multiprocessors the GPU thread blocks - remember that each GPU thread block has the same number of warps - and so $n_{er}^{F_o}$ times the number of warps per GPU thread block times the number of GPU thread blocks that we want resident in each streaming multiprocessor, during the execution of the fatbin file $F_o$ - this is the total number of hardware registers required by the GPU thread blocks resident in a streaming multiprocessor - has to be greater than half the number of hardware registers of a streaming multiprocessor and smaller or equal than the number of hardware registers of a streaming multiprocessor - 8.3;

### 8.2.3 Modification of the Reading and/or Writing Mechanisms

Every fatbin file has to take in input the number of dimensions of the problem and the size of each one of these dimensions. To sort numbers, for example, we have only one dimension and the size of this dimension is the number of numbers we want to sort, while to multiply rectangular matrices the dimensions are three, each one with its size.

Each fatbin file is coded to be executed with one of the following possible four combinations: 1) fixed number of dimensions and fixed size of each dimension, 2) fixed number of dimensions but variable size of each dimension, 3) variable number of dimensions but fixed size of each dimension and 4) variable number of dimensions and variable size of each dimension. Let us call, the previous four combinations, problem structures in input to a fatbin file.

If each GPU thread executing a fatbin file is not going to read/write in the same order all the data/results in input/output from the first to the last then, each time we launch a fatbin file, each GPU thread has to calculate its global identifier to be able to determine from where read/write the data/results during the execution of the fatbin file.

**Types of Reading/Writing Mechanisms**

The reading/writing mechanisms can be implemented in different ways in a fatbin file but considering a) whether the fatbin file has to be executed using a fixed number of GPU threads or can be executed using a variable number of GPU threads and b) the four possible problem structures in input to the fatbin file, we can distinguish a total of three different categories of reading/writing mechanisms for all the possible couples ( fixed or variable number of GPU threads to use to execute a fatbin file , problem structure in input to the fatbin file ):

- $Cat_1$) The fatbin file is coded in such way that we can execute it only using a fixed number of GPU threads - the problem structure can be any. In this case we are very limited in the number of launch configurations we can use to execute the fatbin file because the number of GPU threads is fixed and the number of GPU threads in each warp is always equal to 32.

  One example of this category is a matrix multiplication implementation where given in input to the fatbin file two square matrices and their sizes, each GPU thread get a number of rows from the first matrix equal to the ceil of the number of rows of the first matrix divided the fixed number of GPU threads used to execute the fatbin file and a number of columns from the second matrix equal to the ceil of the number of columns of the second matrix divided the fixed number of GPU threads used to execute the fatbin file.

  Smaller the number of launch configurations we can use to execute a fatbin file smaller the probability to get a lower bound on the real ELF code efficiency near to 100% - the real ELF code efficiency is important is the more near that it is possible to 100% because we calculate it using the theoretical streaming multiprocessor best average performance per clock cycle seen that we can not know and calculate the theoretical best ELF code efficiency and the real best ELF code efficiency - 5.2.

- $Cat_2$) The fatbin file is coded in such way that we can execute it using a variable number of GPU threads - the problem structure can be any - but the subdivision of the work among the GPU threads does not consider the possibilities, of reuse of local data, generated by a) a different number of GPU threads used for the execution of the fatbin file and b) their assignment to the streaming multiprocessors - which GPU threads are where and near to which other GPU threads for the sharing of the resources of the GF100 architecture - specifically the resources of a streaming multiprocessor, 3.3.

  One example of this category is a matrix multiplication implementation, where, given in input, to the fatbin file, a) two square matrices, b) their sizes and c) the number of GPU threads that we want to use to execute the fatbin file, each GPU thread get a number of rows from the first matrix equal to the ceil of number of rows of the first matrix divided the number of GPU threads used to execute the fatbin file and a number of columns from the second matrix equal to the ceil of number of columns of the second matrix divided the number of GPU threads used to execute the fatbin file but 1) the GPU threads, in the same streaming multiprocessor, instead to use the same columns of the second matrix, are using different columns and/or 2) the rows and columns are so long that they can not fit in the number of hardware registers assigned to each GPU thread.

  Because the number of launch configurations that we can use to execute a fatbin file with a reading/writing mechanism of category $Cat_2$ is greater than the number of launch configurations that we can use to execute an analogous fatbin file with a reading/writing mechanism

of category $Cat_1$ then the probability to get a greater lower bound on the real ELF code efficiency for a fatbin file with a reading/writing mechanism of category $Cat_2$ is at least equal, if not greater, than that for the case with an analogous fatbin file with a reading/writing mechanism of category $Cat_1$, where the number of GPU threads used to execute the fatbin file is fixed;

- $Cat_3$) The fatbin file is coded in such way that we can execute it using a variable number of GPU threads - the problem structure can be any - and the subdivision of the work among the GPU threads considers the possibilities of reuse of local data generated by a) a different number of GPU threads used for the execution of the fatbin file and b) their assignment to the streaming multiprocessors;

  One example of this category is a matrix multiplication implementation where, given in input, to the fatbin file, a) two square matrices, b) their sizes and c) the number of GPU threads that we want to use to execute the fatbin file, each GPU thread get a different number of small square submatrices, from the first and the second matrix, considering 1) which other GPU threads are in the same multiprocessor with them, 2) the structure of the ELF code in the B part of the fatbin file and 3) the number of hardware registers assigned to each GPU thread, all this to try to maximize or at least to increase the probabilities of reuse of local data during the execution of the fatbin file.

  Because a) the number of launch configurations that we can use to execute a fatbin file with a reading/writing mechanism of category $Cat_3$ is greater than the number of launch configurations that we can use to execute an analogous fatbin file with a reading/writing mechanism of category $Cat_1$ and b) the ELF codes of the B parts of the fatbin files with a reading/writing mechanism of category $Cat_3$ try to maximize or increase the probabilities of reuse of local data during the execution of the fatbin files compared to the ELF codes of the B parts of analogous fatbin files with a reading/writing mechanism of category $Cat_1$ or $Cat_2$, the probability to get a greater lower bound on the real ELF code efficiency for a fatbin file using a reading/writing mechanism of category $Cat_3$ is at least equal, if not greater, than that for the case with an analogous fatbin file with a reading/writing mechanism of category $Cat_1$ or $Cat_2$.

  This happens not only because we can use several different launch configurations to execute the fatbin file - launch configuration greater in number of those that we can use to execute a fatbin file with a reading/writing mechanism of category $Cat_1$ - but also because the ability of a fatbin file with a reading/writing mechanism of category $Cat_3$, to trying to reuse local data, will require the transfer of an equal or smaller quantity of bytes than that required by an analogous fatbin file with a reading/writing mechanism of category $Cat_1$ or $Cat_2$, and smaller it is the quantity of bytes that is necessary to transfer during the execution of a fatbin file, greater the probability that it is possible to avoid slowdowns, during the execution of a fatbin file, due to the bandwidths and the latencies of the GPU memories.

Another advantage in using a fatbin file with a reading/writing mechanism of category $Cat_2$ or $Cat_3$ is that, having the possibility to be able to choose the number of GPU threads to use for the execution of the fatbin file, the probability to be able to get a greater lower bound on the real ELF code efficiency of the fatbin file, on the different models of the GF100 architecture, each one with different bandwidths and/or latencies for the GPU memories, is greater than the case when

we need to consider an analogous fatbin file with a reading/writing mechanism of category $Cat_1$. Several NVIDIA GPUs in fact use the GF100 architecture, but also whether the architecture is modular, each specific model has a) a different bandwidth between the GPU global memory and the chip, b) a different number of streaming multiprocessors and c) a different quantity of L2 cache on chip, and so 1) bigger the number of launch configurations - and so the number of GPU threads and their possible distributions on the streaming multiprocessors - that we can use to execute a fatbin file and 2) greater the ability of the fatbin file to try to increase the reuse of local data, greater the probability to be able to get a greater lower bound on the real ELF code efficiency of the same fatbin file, on the different models of the GF100 architecture, without necessity of further modifying the fatbin file.

#### Transformation Choices for the Reading/Writing Mechanisms

When we analyze a fatbin file with a reading/writing mechanism of the category $Cat_1$ one of our choices it is whether to transform the fatbin file in a fatbin file using a reading mechanism of category $Cat_2$ - let us call this procedure $P_1$.

Procedure $P_1$ requires simply a) to add to the parameters in input the number of GPU threads that are going to be used for the execution of the fatbin file, b) one or few more ELF registers necessary 1) to calculate the global identifier of the GPU thread and 2) to keep additional addresses or the advancement jumps, in the arrays, the vectors and the structures that contain the input data and that will contain the output results and c) few ELF instructions at the beginning of the B part of the fatbin file to partition the data in input among the GPU threads - this is done positioning the GPU threads in the arrays, the vectors and the structures that contain the input data and that will contain the output results.

Procedure $P_1$ is always possible because a) we know the data layout and can change it, b) we know the number of dimensions of the problem and their sizes and c) all the necessary ELF registers at exclusion maybe of one - the ELF register containing the global identifier of the GPU thread - are already present in the fatbin file. The only other cases, when more than one ELF register is necessary, are:

- 1) When the sizes of the dimensions of the problem are fixed. We find the possibility for such cases unlikely, because it is not in the GPU paradigm to create a fatbin file able to solve a problem with fixed sizes for its dimensions, because for matrix multiplication, for example, this would mean that, excluding the number of GPU threads we can use to execute the fatbin file, we can multiply only two matrices of fixed size;

- 2) When the user wants to make the GPU threads read and write the data and the results, in the arrays, the vectors and the structures, not in a consecutive way, but jumping from address to address, with a jump that can be represented with a constant but it is a function of something, for example the parameters of the launch configuration.

If we therefore want to transform a fatbin file with a reading/writing mechanism of category $Cat_1$ in a fatbin file with a reading/writing mechanism of category $Cat_2$ then we use the procedure $P_1$. The procedure $P_1$ utilizes the procedure 6.7 to generate a fatbin file as output a) that has a reading/writing mechanism of category $Cat_2$ and b) that is practically equal of the fatbin file with the reading/writing mechanism of category $Cat_1$ given in input to $P_1$.

The fatbin file as output, with the reading/writing mechanism of category $Cat_2$, has effectively a) the same ELF instructions, in the same order, with the same dependences among ELF registers of those of the fatbin file with the reading/writing mechanism of category $Cat_1$, b) the same number and type of ELF registers of those of the fatbin file with the reading/writing mechanism of category $Cat_1$ also whether the ELF registers could be different from those of the fatbin file with the reading/writing mechanism of category $Cat_1$, but c) one or few more ELF registers and few more ELF instructions at the beginning of its B part to partition the data in input among the GPU threads and allows to each GPU thread used to execute the fatbin file to position itself in the arrays, the vectors and the structures necessary for the execution of the fatbin file - the arrays, the vectors and the structures that contain the input data and that will contain the output results.

A fatbin file with a reading/writing mechanism of the category $Cat_1$ or $Cat_2$ can instead be transformed in one with a reading/writing mechanism of the category $Cat_3$ - let us call this procedure $P_2$ - but this procedure it could be more complex of the procedure $P_1$.

The types of transformations that the user wants to apply, to the fatbin file, with a reading/writing mechanism of category $Cat_1$ or $Cat_2$, in input to the procedure $P_2$, determines whether or not the output fatbin file of the procedure $P_2$ is practically equal or could be much different from the fatbin file in input:

- The fatbin file produced as output could be practically equal to the fatbin file in input because a) if the fatbin file in input to the procedure $P_2$ is a fatbin file with a reading/writing mechanism of category $Cat_1$ then we can apply the procedure $P_1$ to the fatbin file with a reading/writing mechanism of category $Cat_1$ in input to the procedure $P_2$ and later to try to increase the reuse of local data simply changing the positions of the data in the data layout or b) if the fatbin file in input to the procedure $P_2$ is a fatbin file with a reading/writing mechanism of category $Cat_2$ then we can directly change only the positions of the data in the data layout to try to increase the reuse of local data - let us call the set of transformations and changes necessary in all these cases $TAC_1$;

- The fatbin file produced as output could be instead very different from the fatbin file in input in the cases the user wants to modify a) the order and type of ELF instructions in the B parts of the fatbin files and/or b) the ELF registers used in the ELF instructions in the B part of the fatbin file and so maybe their number, dependence types and dependence distances and c) maybe the positions of the data in the data layout or the data layout - let us call the set of transformations and changes necessary in all these cases $TAC_2$.

Transforming a fatbin file with a reading/writing mechanism of category $Cat_1$ to a fatbin file with a reading/writing mechanism of category $Cat_2$ can be easily done with a procedure almost entirely or entirely automated, while transforming a fatbin file with a reading/writing mechanism of category $Cat_1$ or $Cat_2$, to a fatbin file with a reading/writing mechanism of category $Cat_3$, can instead become very hard - this happens if instead to try to increase the reuse of local data, as it is written in the definition of the reading/writing mechanism of category $Cat_3$, the user wants instead guarantees about the increase of the reuse of local data compared to the case where the user is going to use a fatbin file, analogous to the fatbin file with a reading/writing mechanism of category $Cat_3$, but with a reading/writing mechanism of category $Cat_1$ or $Cat_2$:

- If the user wants to apply the set of transformations and changes $TAC_1$ then a procedure almost entirely or entirely automated can be created to apply the set of transformations and

changes $TAC_1$ to a fabin file in input, this with the guarantee that the reuse of local data it is at least equal, whether not greater, than the reuse that the user would have if he/she would use the fatbin file with the reading/writing mechanism of category $Cat_1$ or $Cat_2$ in input to the procedure $P_2$ - however the guarantee is only possible a) if the reader believes the warp scheduling policy executed by the warps schedulers in the streaming multiprocessors is the cycling policy, see 9 for a description of the cycling policy, b) if the output fatbin file is one of the fatbin files in the subset $SS_{A_1}$, see 10 to understand when a fatbin file is in the subset $SS_{A_1}$ , and c) if the fatbin file has at least a launch configuration, between the possible, satisfying all the requirements of the analysis $A_1$, see 12 for a description of the analysis $A_1$, otherwise it is hard whether not impossible to give to the user an a priori guarantee about the increase of reuse of local data;

- If the user wants to apply the set of transformations and changes $TAC_2$ then we do not think 1) the choice of the transformations and of the changes in the set $TAC_2$ and 2) the application of the transformations and of the changes in the set $TAC_2$ to a fatbin file in input, can be automated - the job is highly complex already for an human being because a) it depends on the order of the ELF instructions in the B part of the fatbin file and by which ELF registers are used in each ELF instruction and b) it is hard to quantify how these two things, together at their many degrees of freedom, also just only for the simplest cases where we do not change the ELF instructions in the B part of a fatbin file, increase or decrease the reuse of local data compared to the reuse of local data the user would get if he/she would use the fatbin file with a reading/writing mechanism of category $Cat_1$ or $Cat_2$ in input to the procedure $P_2$, and so, as for the previous case, it is hard whether not impossible to give to the user an a priori guarantee about the increase of reuse of local data;

If the user decides that he/she wants to apply the set of transformations and changes $TAC_2$ to a fatbin file with a reading/writing mechanism of category $Cat_1$ or $Cat_2$ then there is a further disadvantage. Because in this case 1) the order of the ELF instructions in the B part of the fatbin file and 2) which ELF registers are used in each ELF instruction, are important, then, when an user decides to generate or transform a fatbin file with a reading/writing mechanism of category $Cat_1$ or $Cat_2$ in a fatbin file with a reading/writing mechanism of category $Cat_3$, he/she can not edit the GPU code in CUDA or PTX because the ELF code, of B part, of the fatbin file, produced by nvcc, is usually very different from that in CUDA or PTX for a) the order and type of ELF instructions, b) the number and type of ELF registers, c) the roles of the ELF registers used in the ELF instructions in the B part of the fatbin file, d) the dependence types of the ELF registers used in the ELF instructions in the B part of the fatbin file and e) the dependence distances among the ELF registers used in the ELF instructions in the B part of the fatbin file, and so if the user wishes to apply the set of transformations and changes $TAC_2$ to a fatbin file with a reading/writing mechanism of category $Cat_1$ or $Cat_2$ then he/she has to generate a fatbin file with the necessary resources and to modify the fatbin file to get the wanted ELF algorithmic implementation - these two things can be done using the procedures described in 6.6 and 6.7, but the whole process, also if partially automated, is very time consuming because the level of detail and focus required to the user to edit the necessary parts to implement his/her specific algorithmic are very demanding.

If instead the reader choose to use the set of transformations and changes $TAC_1$ then the user can a) simply give in input, to the procedure $P_2$, the fatbin file with a reading/writing mechanism of category $Cat_1$ or $Cat_2$, b) apply the procedure $P_1$ to the fatbin file and later c.1) use another

automated procedure to change the positions of the data in the data layout or c.2) reprogram the part on the CPU side that has to read the data from the hard disk and organize them in the arrays, the vectors and the structures, that is necessary to transfer from the CPU to the GPU, for the execution of the fatbin file.

Finally, if the user does not want a) to change the order of the ELF instructions in the B part of a fatbin file and b) the ELF registers used in the ELF instructions in the B part of a fabin file and their roles in the ELF instructions, then we definitely advise, if necessary, to use the procedure $P_1$ that can be automated. Our advice is to use the procedure $P_1$, if necessary and if the user does not want to change a) and b), because later in 8.4 we will show how, taking the output fatbin file of the procedure $P_1$ or $P_2$, we generate a set of fatbin files with the same ELF instructions in their B parts of the ELF instructions of the B part of the output fatbin file of the procedure $P_1$ or $P_2$, the same dependences among the ELF registers in their B parts of the ELF registers in the B part of the output fatbin file of the procedure $P_1$ or $P_2$, but with each fatbin file with one of the possible logically correct orders of the ELF instructions in the B part of the output fatbin file of the procedure $P_1$ or $P_2$.

## 8.3 Selection of the Launch Configurations

Not all the launch configurations that could be used to execute a fatbin file are used to analyze the fatbin file. To determine the launch configurations that can be used to analyze a fatbin file and so the potential launch configurations that can be used to execute a fatbin file we need in fact to consider the following things:

- The results in 7 about the time differences for the warp schedulings are got using launch configurations where the gigathread scheduler assigns, to the streaming multiprocessors, the GPU thread blocks used for the execution of the fatbin files, at the beginning of execution of the GPU code. It is in fact important that the gigathread scheduler a) does not assign any GPU thread block to the streaming multiprocessors during any other moment of the fatbin file execution at exclusion of the beginning of the execution of the GPU code and b) evenly distributes the GPU thread blocks to the streaming multiprocessors - in this way we got a first form of load balancing for the execution of the ELF code, of the B part, of a fatbin file.

  The launch configurations that can be considered to analyze a fatbin file have therefore a) to have a number of GPU thread blocks that can be assigned by the gigathread scheduler to the streaming multiprocessors at the beginning of the execution of the GPU code, b) to force the gigathread scheduler to evenly distribute the GPU thread blocks to the streaming multiprocessors and c) to avoid the presence of any GPU thread block to assign to the streaming multiprocessors during any other phase of the fatbin file execution different from the beginning of the execution of the GPU code - for example, if we are using thousands of GPU thread blocks for the execution of a fatbin file then the gigathread scheduler is surely going to assign some GPU thread blocks, to the streaming multiprocessors, after the beginning of the execution of the GPU code, because at each moment, each streaming multiprocessor can not have more than 8 resident GPU thread blocks;

- The number of resident warps in each streaming multiprocessor, during the execution of the GPU code of a fatbin file, can not be greater than 32 also whether the possible maximum

would be 48. The reason because we use 32 instead of 48 is that in 7.6.2 it is easy to execute fatbin files using only 1 GPU thread block and so, for every dependence distance of interest of each instruction configuration of interest, to study what happens, in a streaming multiprocessor, when the number of resident warps in the streaming multiprocessor is between 1 and 32 - remember that for every instruction configuration, fatbin file 1 was edited to study the dependence distance 1, fatbin file 2 was edited to study the dependence distance 2, etc. - but the things are harder to study for a number of resident warps between 33 and 48 because:

- 1) For prime numbers of warps like 37, 41 or 47 there does not exist any launch configuration able to give for the execution of the for loop, of the B part, of a fatbin file, 37, 41 or 47 warps in at least one streaming multiprocessor - all the GPU thread blocks have the same number of warps and the maximum number of warps that can have each GPU thread block is 32 so independently of an even or not even distribution of the GPU thread blocks to the streaming multiprocessors is not possible to get 37, 41 or 47 warps on any of the streaming multiprocessors for the whole execution of the for loop, of the B part, of a fatbin file;
- 2) Every fatbin file was created to study the dependence distance of a dependence type, write-read or read-read, of an instruction configuration. If the number of warps W between 33 and 48 that we want in each streaming multiprocessor times the number of GPU threads in each warp - 32 - times the number of ELF registers of the fatbin file is greater than the number of hardware registers of a streaming multiprocessor, then there is not launch configuration, that we can use, to study the dependence distance, of the dependence type, of the instruction configuration, with a number of warps between W and 48, because in a streaming multiprocessor, the hardware registers, necessary for the execution of the GPU code of the fatbin file, are not enough;
- 3) If the number of warps between 33 and 48 that we wants in each streaming multiprocessor times the number of threads in each warp - 32 - times the number of ELF registers of the fatbin file is smaller or equal than half the number of hardware registers of a streaming multiprocessor then to study the dependence distance, of the dependence type, of the instruction configuration, for a number of resident warps in a streaming multiprocessor between 33 and 48, we need to use $P_A$ to transform the fatbin file in such way that the addition of useless ELF registers makes the number of warps we wants in each streaming multiprocessor times the number of GPU threads in each warp times the number of ELF registers of the fatbin file greater than half the number of hardware registers of a streaming multiprocessor and smaller or equal than the number of hardware registers of a streaming multiprocessor.

  If this is the case then we need also a) to take care of the fact that we are not going to use only a number of GPU thread blocks equal to 1 or 2 times the number of streaming multiprocessors - for 33, for example, we can only use 3 GPU thread blocks per streaming multiprocessor, each GPU thread block with 11 warps, while for 35 we can use 5 GPU thread blocks per streaming multiprocessor, each GPU thread blocks with 7 warps, or 7 GPU thread blocks per streaming multiprocessor, each GPU thread block with 5 warps - and b) at the end of each execution, to find out which GPU thread blocks were on which streaming multiprocessors to be able to calculate the time differences.

Considering a) the quantity of work necessary, b) that the procedure $P_A$ is not automated for

this type of work, c) the fact that for some prime numbers of warps is however not possible to get any result at cause of the GPU hardware design that we can not modify, d) the fact that for some not prime numbers of warps we can not exclude too that it is however not possible to get any result at cause of the GPU hardware design that we can not modify and e) that we found that every dependence distance, of every dependence type, of every ELF instruction configuration, requires a minimum number of resident warps in a streaming multiprocessor smaller than 32, to get the real ELF instruction configuration streaming multiprocessor best average performance per clock cycle - this if the byte transfers, among the different GPU memories, can not slow down the executions of the ELF codes, of the B parts, of the fatbin files, as effectively it was also the case for the executions of the for loops, of the B parts, of the fatbin files used in 7 - we decide to use 32 instead of 48, for the selection of the launch configurations, also whether in future we plan to expand our work;

- Let us to consider the tables - calculated, during the extraction of the local streaming multiprocessor ELF architectural features in 7.6.2, considering, at the same time, all the ELF instruction configurations - $T_a$ of the maximum absolute time differences between maximum ending time and maximum starting time differences of the couples ( fatbin file , launch configuration ), $T_b$ of the maximum ending time differences of the couples ( fatbin file , launch configuration ) and $T_c$ of the maximum starting time differences of the couples ( fatbin file , launch configuration ).

  Because such tables were calculated considering at the same time all the ELF instruction configurations, in each one of them we have on the rows the dependence distances - fatbin file 1 corresponds to dependence distance 1, fatbin file 2 corresponds to dependence distance 2, etc. - without any consideration about a) the dependence types or b) the ELF instruction configurations of the dependence distances - and on the columns the launch configurations used to execute the fatbin files of each instruction configuration - launch configuration 1 has only 1 GPU thread block with 1 warp that is going to be resident on the only streaming multiprocessor used, launch configuration 2 has only 1 GPU thread block with 2 warps that are going to be resident on the only streaming multiprocessor used, etc. .

  All the maximum starting time differences in the table $T_c$ are ok because, independently of the quantity of work that each GPU thread has to execute in the for loop, of the B part, of a fatbin file, at each execution, the values in the table $T_c$ are smaller than 300 function unit clock cycles - this means that the starting time differences are not influenced by a) the quantity of work that each GPU thread has to execute in the for loop, of the B part, of the fatbin file, b) the number of resident warps in a streaming multiprocessor, c) the scheduling waiting times, d) the dependence waiting times, e) the overhead time due to the management of the warps, f) the dependence distance, g) the dependence type or h) the ELF instruction configuration, and remember that the executions of the for loops, of the B parts, of the fatbin files, can not be slowed down by the bandwidths and the latencies of the GPU memories.

  We instead already know that the maximum ending time differences in the table $T_b$ are not ok for launch configurations with an odd number of resident warps in a streaming multiprocessor and therefore, because table $T_c$ is completely ok, of consequence also the maximum absolute time differences between maximum ending time and maximum starting time differences of the couples ( fatbin file , launch configuration ) in the table $T_a$, corresponding to the couples ( fatbin file , launch configuration ) in the table $T_b$, are not ok.

In the tables $T_a$, $T_b$ and $T_c$, the columns represent the number of resident warps in the only streaming multiprocessor used during a fatbin file execution - column 1 1 warp, column 2 2 warps, etc. . While we use launch configurations, with only 1 GPU thread block, with a number of warps, for the only GPU thread block used, going from 1 to 32, to calculate the 32 columns of the tables $T_a$, $T_b$ and $T_c$, it is very important, for the next discussions, to understand that we can use more different launch configurations, to get the same number of resident warps, in each one of the streaming multiprocessors, for the execution of a fatbin file.

Because table $T_c$ is ok, it is checking the results in the table $T_b$ that we determine a first set of launch configurations that are not ok for the execution of a fatbin file - table $T_a$ does not matter because as we have said it is determined only by the results in table $T_b$, this because the whole table $T_c$ is ok.

A first set of launch configurations that are not ok for the execution of a fatbin file are all the launch configurations that imply an odd number of resident warps in each one of streaming multiprocessors - note that the launch configurations that are not ok, because implying an odd number of resident warps, in each one of the streaming multiprocessors used, could be different for different fatbin files, this because each one of the fatbin files could have a different number of ELF registers.

The set of launch configurations $S_{lc}$ that can be used to analyze a fatbin file, it would therefore seem to be a set composed by launch configurations with a number of GPU thread blocks that a) give us the guarantee that the GPU thread blocks are going to be evenly distributed by the gigathread scheduler to the streaming multiprocessors at the beginning of the execution of the GPU code of a fatbin file and b) have an even number of resident warps per streaming multiprocessor smaller or equal than 32, but this is not true.

Each launch configuration in $S_{lc}$ determines the number of resident warps in a streaming multiprocessor during the execution of a fatbin file. We separate the launch configurations of $S_{lc}$ in different subsets $SS_w$, each subset corresponding to a different number of warps W resident in a streaming multiprocessor during the execution of the fatbin file - the number of subsets are therefore 48 because a) not more than 48 warps can be resident in a streaming multiprocessor at any moment during the execution of a fatbin file, b) the launch configurations in $S_{lc}$ force the gigathread scheduler to assign all the GPU thread blocks at the beginning of the execution of the GPU code of the fatbin file and c) when a GPU thread block is assigned to a streaming multiprocessor the GPU thread block can not migrate to another streaming multiprocessor. Next we calculate the set of the dependence distances $S_{dd}$ that appear in the B part of the fatbin file. We therefore generate all the couples ( dependence distance in $S_{dd}$ , not empty subset $SS_w$ = number of resident warps in a streaming multiprocessor ). Finally we eliminate from $S_{lc}$ all the launch configurations in the subsets $SS_w$ of the couples ( dependence distance in $S_{dd}$ , not empty subset $SS_w$ ) that in the table $T_b$ have a maximum ending time difference greater than 300 function unit clock cycles - let us call the set composed by all these eliminated launch configurations $B_{lc}$, where B means bad. The launch configurations remained in $S_{lc}$ - the set of the potential launch configurations - are the launch configurations that we use to analyze the fatbin file.

The fact, that there are couples ( dependence distance , number of resident warps in a streaming multiprocessor ), that force the gigathread scheduler to evenly distribute the GPU thread blocks, to the streaming multiprocessors, at the beginning of the execution of GPU code of

a fatbin file and imply an even number of resident warps in each streaming multiprocessor, a maximum starting time difference smaller than 300 function unit clock cycles but a maximum ending time difference greater than 300 function unit clock cycles, is due to GPU hardware problems that we can not fix. The reasons because we say that this phenomenon is due to GPU hardware problems are the following:

- In $S_{lc}$, we have only launch configurations, generating couples ( dependence distance in $S_{dd}$ , launch configuration in $S_{lc}$ ), that, in the table $T_c$, have a maximum starting time difference, and, in the table $T_b$, have a maximum ending time difference, for the resident warps in each one of the single streaming multiprocessors used, a) of the same order of magnitude and b) smaller than 300 function unit clock cycles - these two things are true independently of the quantity of work that each GPU thread has to execute in the for loops, of the B parts, of the fatbin files.

  In $B_{lc}$ instead, we have only launch configurations, generating couples ( dependence distance in $S_{dd}$ , launch configuration in $B_{lc}$ ), that, in table $T_c$, have a maximum starting time difference, for the resident warps in each one of the single streaming multiprocessors used, smaller than 300 function unit clock cycles, independently of the quantity of work that each GPU thread has to execute in the for loops, of the B parts, of the fatbin files, but that, in table $T_b$, have a maximum ending time difference, for the resident warps in each one of the single streaming multiprocessors used, from 2 to 6 orders of magnitude greater than 300 function unit clock cycles, already for the executions of only 1 million - a very low number - of ELF instructions, in the for loops, of the B parts, of the fatbin files, per GPU thread, and the orders of magnitude increase at the increase of the quantity of work that each GPU thread has to execute in the for loops, of the B parts, of the fatbin files;

- Because the bandwidths and the latencies of the GPU memories can not slow down the execution of the for loops, of the B parts, of the fatbin files, used for the quantification of the local ELF architectural features, considering a generic couple ( dependence distance , number of resident warps in a streaming multiprocessor ) and taking all the ELF instruction configurations not limited by the scheduling waiting times, the dependence waiting times and the overhead time due to the management of the warps - we know which are such ELF instruction configurations for each couple ( dependence distance, number of resident warps in a streaming multiprocessor ) because in 7.6.2 we concurrently consider the scheduling waiting times, the dependence waiting times and the overhead time due to the management of the warps to determine the minimum number of warps necessary in a streaming multiprocessor to get an average throughput not smaller than the 95% of the maximum average of the average throughputs obtained, for the ELF instruction configuration, considering all the couples ( fatbin file , launch configuration ) used to analyze the ELF instruction configuration - we can say that the phenomenon is not due to some particular ELF instruction configurations and that, because the execution of the for loops, of the B parts, of the fatbin files, can not be slowed down by the bandwidths and the latencies of the GPU memories and for the cases here considered the execution of the for loops, of the B parts, of the fatbin files, is not slowed down by the scheduling waiting times, the dependence waiting times or the overhead time for the management of the warps, we can say that, for causes that we can not modify, a) during the execution

of the fatbin files, the warp schedulers prefer to schedule some warps instead of others and b) this phenomenon increases the maximum ending time differences at the increase of the quantity of work that each GPU thread has to execute in the for loops, of the B parts, of the fatbin files and so it is probably coming in play from the beginning of the executions of the for loops, of the B parts, of the fatbin files and is going to last almost for the whole executions of the for loops, of the B parts, of the fatbin files - we exclude the end of the executions of the for loops, of the B parts, of the fatbin files, where some warps are going to finish before of others and therefore force the warp schedulers to choose among a reduced number of resident warps, for a period of time that is greater than the cases for which the maximum ending time differences are smaller than 300 clock cycles;

We did not study the frequency of the phenomenon but we can not fix it in any case. Because the phenomenon can reduce the real ELF code efficiency of an ELF code we discard the launch configurations in $B_{lc}$ from $S_{lc}$ - in $B_{lc}$ there could be therefore launch configurations implying a) an even distribution of the GPU thread blocks to the streaming multiprocessors, b) an even number of resident warps in each streaming multiprocessor and c) a distribution of the GPU thread blocks, to the streaming multiprocessors, at the beginning of the execution of the GPU code of the fatbin file.

To summarize, to generate the set of launch configurations $S_{lc}$ to use for the analysis of a fatbin file it is necessary to calculate the following sets in the following order and in the following ways:

- The first set is composed by all the launch configurations a) with a number of GPU thread blocks smaller than 8 times the number of streaming multiprocessors - 8 because the maximum number of GPU thread blocks that can be resident at each moment during the execution of a fatbin file in a streaming multiprocessor is 8 - and b) with a number of warps per GPU thread block going from 1 to 32 - the maximum number of warps that a GPU thread block can have;

- From the first set, by elimination, we get a second set of potential launch configurations. The launch configurations eliminated by the first set are all the launch configurations, with a number of GPU thread blocks, that, divided by the number of streaming multiprocessors of the specific model of the GF100 architecture that we want use to execute the fatbin file, do not give us an integer number - this means that the gigathread scheduler can not evenly distribute the GPU thread blocks to the the streaming multiprocessors;

- From the second set, by elimination, we get a third set of potential launch configurations. The launch configurations eliminated by the second set are all the launch configurations that would have a number of resident warps per streaming multiprocessor greater than 32, this supposing the gigathread scheduler evenly distributes the GPU thread blocks to the streaming multiprocessors;

- From the third set, by elimination, we get a fourth set of potential launch configurations. The launch configurations eliminated by the third set are all the launch configuration that, supposing the gigathread scheduler evenly distributes the GPU thread blocks to the streaming multiprocessor, require a number of hardware registers per streaming multiprocessor - number of resident warps per streaming multiprocessor times the number of GPU threads per warp,

32, times the number of ELF registers of the fatbin file - that is smaller than half of the number of hardware registers in a streaming multiprocessor or greater than the number of hardware registers in a streaming multiprocessor - in this way we have the guarantee that, for the launch configurations not eliminated, a) the gigathread scheduler is going to evenly distribute the GPU thread blocks to the streaming multiprocessors and b) the gigathread scheduler is going to do this at the beginning of the execution of the GPU code of the fatbin file;

- From the third set, by elimination, we get a fourth set of potential launch configurations. The launch configurations eliminated by the third set are all the launch configurations with an odd number of resident warps in a streaming multiprocessor;

- From the fourth set, by elimination, we get the fifth set of potential launch configurations. The fifth set is the set of launch configurations $S_{lc}$ to use to analyze the fatbin file. The launch configurations eliminated by the fourth set are all the launch configurations that, considering the dependence distances that appear in the interpretation text file of the fatbin file, generate couples ( dependence distance in $S_{dd}$ , launch configuration = number of resident warps in a streaming multiprocessor ) that have in the table $T_b$ a maximum ending time difference greater than 300 function unit clock cycles.

## 8.4 Transformation of the Fatbin File to Analyze

Every fatbin file has a reading/writing mechanism of category $Cat_1$, $Cat_2$ or $Cat_3$. If the fatbin file has a reading/writing mechanism of category $Cat_1$ we can decide whether, for the analysis/analyses, to keep the fatbin file in the way it is or whether, using the procedure $P_1$, to transform the fatbin file in a fatbin file with a reading/writing mechanism of category $Cat_2$ or $Cat_3$. If the fatbin file has a reading/writing mechanism of category $Cat_2$ we can decide whether, for the analysis/analyses, to keep the fatbin file in the way it is or whether, using the procedure $P_2$, to transform the fatbin file in a fatbin file with a reading/writing mechanism of category $Cat_3$.

Let us call $F_{f_i}$ the original fatbin file in input to the previous potential procedure of transformation of its reading/writing mechanism and the output fatbin file of such potential procedure of transformation $F_{f_o}$ - $F_{f_o}$ can therefore being equal to $F_{f_i}$ or being the transformation of $F_{f_i}$.

$F_{f_o}$ has a number of ELF registers, let us say $n_{er}^{F_{f_o}}$. Using the procedure $P_b$ we generate a set of fatbin files, one for each integer number between $n_{er}^{F_{f_o}}$ and 64 - the maximum number of ELF registers that can have a fatbin file. Let us call the set of generated fatbin files $S_{F_f}^1$. Every one of the fatbin files in $S_{F_f}^1$ is analogous to $F_{f_o}$, maybe the name of the useful ELF registers used in the fatbin files in $S_{F_f}^1$ is different from the name of the ELF registers used in $F_{f_o}$ but this does not matter because each one of the fatbin files in $S_{F_f}^1$ has the same number and type of useful ELF registers of $F_{f_o}$, the dependences among the useful ELF registers in the B parts of the fatbin files in $S_{F_f}^1$ are equal to the dependences among the ELF registers in the B part of the fatbin file $F_{f_o}$ and the order and type of ELF instructions in the B parts of the fatbin files in $S_{F_f}^1$ are equal to the order and type of ELF instructions in the B part of fatbin file $F_{f_o}$ - the fatbin files in $S_{F_f}^1$ have also some additional useless ELF registers compared to $F_{f_o}$ but such useless ELF registers are not used in the B parts of the fatbin files in $S_{F_f}^1$. The generation of all these fatbin files is necessary to increase the probability that some launch configurations, not present in the $S_{lc}$, of the fatbin file $F_{f_o}$, are instead present in one or more $S_{lc}$s of the fatbin files in $S_{F_f}^1$.

Because the B parts of the fatbin files in $S^1_{F_f}$ are equivalent to the B part of the fatbin file $F_{f_o}$, we can use them, instead of $F_{f_o}$ alone, to analyze $F_{f_o}$, and because it is probable that some launch configurations not present in the $S_{lc}$ of the fatbin file $F_{f_o}$ are instead present in the $S_{lc}$s of the fatbin files in $S^1_{F_f}$, the probability, to find couples ( fatbin file in $S^1_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^1_{F_f}$ ), with a greater lower bound on the real ELF code efficiency, is equal, whether not greater, than the case where we consider only the couples ( fatbin file $F_{f_o}$ , launch configuration in the $S_{lc}$ of the fatbin file $F_{f_o}$ ) or only the couples ( original fatbin file $F_{f_i}$ , launch configuration in the $S_{lc}$ of the original fatbin file $F_{f_i}$ ), in the case we did not transform $F_{f_i}$.

Let us consider the cases when a) the number of ELF registers of the fatbin file $F_{f_o}$ is equal to $n_{er}^{F_{f_o}}$, b) we want a number of GPU thread blocks per streaming multiprocessor equal to $B$, c) the number of warps of each GPU thread block is equal to $W$ but e) $B \cdot W \cdot 32 \cdot n_{er}^{F_{f_o}}$ is smaller than half of the number of hardware registers in a streaming multiprocessor. If we use couples ( fatbin file , launch configuration ) satisfying the conditions a), b), c), d) and e), then it is not possible to get the guarantee that the gigathread scheduler is going to evenly distribute the GPU thread blocks to the streaming multiprocessors. Thanks to generation of $S^1_{F_f}$ is instead probable that there are one or more fatbin files, analogous to $F_{f_o}$, but with a number of ELF registers $n_{er}^{F_f}$ greater than $n_{er}^{F_{f_o}}$, such that $B \cdot W \cdot 32 \cdot n_{er}^{F_f}$ is greater than half of the number of hardware registers in a streaming multiprocessor but equal or smaller than the number of ELF registers in a streaming multiprocessor. If this is the case then we have the guarantee that the gigathread scheduler is going to evenly distribute the GPU thread blocks to the streaming multiprocessors when we use the couples ( fatbin file in $S^1_{F_f}$ , wanted launch configuration/s = number of resident warps in a streaming multiprocessor ) and that the gigathread scheduler is going to do this at the beginning of the execution of the GPU code of the fatbin file.

Some or all the fatbin files in $S^1_{F_f}$ could however have empty $S_{lc}$s - this depends on the $S_{dd}$ of each fatbin file in $S^1_{F_f}$. To avoid to loose good candidates for the analysis/analyses and for further increasing the probability to get a greater lower bound on the real ELF code efficiency of the original fatbin file $F_{f_i}$, we therefore use procedure $P_A$ on each one of the fatbin files in $S^1_{F_f}$.

Let us call $S_A$ each one of the sets of fatbin files generated by the procedure $P_A$ taking in input a fatbin file in $S^1_{F_f}$ - the number of $S_A$s generated is equal to the number of fatbin files in $S^1_{F_f}$ - and let us call $S^2_{F_f}$ the set containing all the fatbin files of all the sets $S_A$ that have been generated. The $S_{dd}$ of each fatbin file in a $S_A$ is probably different a) from the $S_{dd}$ of at least some other fatbin files in the $S_A$ and b) from the $S_{dd}$ of the fatbin file in $S^1_{F_f}$ used to generate $S_A$ and so the single fatbin files in each $S_A$ have a $S_{lc}$ that is probably different a) from the others $S_{lc}$s of the other fatbin files in the same $S_A$ and b) from the $S_{lc}$ of the fatbin file in $S^1_{F_f}$ used to generate $S_A$.

All the fatbin files in $S^2_{F_f}$ with an empty $S_{lc}$ are eliminated. The remaining files in $S^2_{F_f}$, with a not empty $S_{lc}$, are used to analyze the original fatbin file $F_{f_i}$ - this is done analyzing all the couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ). Generating $S^2_{F_f}$ is useful because also if the dependences, among the ELF registers, used in the ELF instructions, in the B parts, of the fatbin files, in $S^2_{F_f}$, are the same of the dependences, among the ELF registers, used in the ELF instructions, in the B part, of the fatbin file $F_{f_o}$, changing the order of the ELF instructions can increase the probability to get a greater lower bound on the real ELF code efficiency because:

- If a fatbin file in $S^1_{F_f}$ has an empty $S_{lc}$, at cause of its $S_{dd}$ used in the last step of the procedure used to generate its $S_{lc}$, then generating an $S_A$ for the fatbin file in $S^1_{F_f}$ increases

the probability that at least some fatbin files in its $S_A$ have a not empty $S_{lc}$. If at least some fatbin files in its $S_A$ have a not empty $S_{lc}$ then we have increased the number of couples ( fatbin file , launch configuration ) that we can use to analyze the original fatbin file $F_{f_i}$ and so we have made equal, whether not greater, the probability to get a greater lower bound on the real ELF code efficiency of the original fatbin file $F_{f_i}$;

- Different orders, of the ELF instructions, in the B parts, of the fatbin files, in $S^2_{F_f}$, imply the transfer of different byte quantities, among the different GPU memories, during the executions of the fatbin files.

  Considering that a) we can not choose the warp schedulings, b) we can not know the warp schedulings the GPU hardware design will allow to the warp schedulers to choose for the execution of a fatbin file and c) that the bandwidths and the latencies of the GPU memories are fixed for each specific model of the GF100 architecture then, for some byte quantities that it is necessary to transfer during the execution of a fatbin file - such quantities are due to the order and type of ELF instructions in the B part of the fatbin file, to the dependences among the ELF registers used in the ELF instructions in the B part of the fatbin file and to the warp schedulings that the GPU hardware design allows to the warp schedulers to choose for the execution of the B part of the fatbin file - the bandwidths and the latencies of the GPU memories could be a bottleneck, during the execution of the fatbin file, with some or all the warp schedulings that the GPU hardware design allows to the warp schedulers to choose for the execution of the B part of the fatbin file, and so the bandwidths and the latencies of the GPU memories could slow down the execution of the B part of the fatbin file.

  Greater the number of analogous possibilities that we can choose to execute a fatbin file $F_{f_o}$ - in other words fatbin files a) with the number and type of their ELF instructions in their B parts equal to number and type of the ELF instructions in the B part of the fatbin file $F_{f_o}$, b) with the dependences among the ELF registers used in the ELF instructions in their B parts equal to the dependences among the ELF registers used in the ELF instructions in the B part of the fatbin file $F_{f_o}$ but c) with a different logically correct order of the ELF instructions - better, because it is greater the probability that, for at least one of the orders of the ELF instructions in the B parts of the fatbin files, some or all the warp schedulings, that the GPU hardware design allows to the warp schedulers to choose for the execution of at least one of the fatbin files, make it impossible for the bandwidths and the latencies of the GPU memories to slow down the execution of the order of ELF instructions in the B part of at least one of the fatbin files however, if the bandwidths and the latencies of the GPU memories can not slow down the execution of the order of ELF instructions in the B part of a fatbin file, for only some of the warp schedulings that the GPU hardware design allows to the warp schedulers to choose for the execution of the B part of the fatbin file, then we need to make an experimental statistical study on the execution times of the B part of the fatbin file.

- Greater the probability to find at least one fatbin file in $S^2_{F_f}$ that, when executed with one of the launch configurations in its $S_{lc}$, with some or all the warp schedulings that the GPU hardware design allows to the warp scheduler to choose for the execution of the fatbin file, has an execution of its B part that can not be slowed down by the scheduling waiting times, the dependence waiting times and the overhead times due to the management of the warps - if this could happen for only some of the warp schedulings that the GPU hardware design

allows to the warp schedulers to choose for the execution of the B part of the fatbin file then we need here too to make an experimental statistical study on the execution times of the B part of the fatbin file.

To summarize, going from the original fatbin file $F_{f_i}$ to the fatbin file $F_{f_o}$, later, using the procedure $P_B$, from the fatbin file $F_{f_o}$ to $S^1_{F_f}$, and, using the procedure $P_A$, from $S^1_{F_f}$ to $S^2_{F_f}$, is a way to increase the probability to get a greater lower bound on the real ELF code efficiency of the original fatbin file $F_{f_i}$. The fatbin files of the couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) are fatbin files analogous to the fatbin file $F_{f_o}$ - this means that each one of the fatbin files in $S^2_{F_f}$ has a) the number and type of ELF instructions in its B part equal to the number and type of ELF instructions in the B part of the fatbin file $F_{f_o}$, b) the dependences among the ELF registers used in the ELF instructions in its B part equal to dependences among the ELF registers used in the ELF instructions in the B part of the fatbin file $F_{f_o}$ but c) a different logically correct order of the ELF instructions in its B part and d) a number of ELF registers equal or greater than the number of ELF registers of the fatbin file $F_{f_o}$. The total number of different launch configurations, obtained considering the sets $S_{lc}$ of the fatbin files in $S^2_{F_f}$, is greater than the number of launch configurations in the set $S_{lc}$ of $F_{f_i}$.

## 8.5 Summary

In this chapter we have described the steps necessary a) to transform an original fatbin file $F_{f_i}$ and b) to choose the launch configurations to use during the analysis/analyses, of each one of the fatbin files produced from the original one, this to increase the probability to get a greater lower bound on the real ELF code efficiency of the original fatbin file $F_{f_i}$. The main points to remember from this chapter are the following:

- We can modify the reading/writing mechanism that a fatbin file is using to read/write data/results a) to increase the number of launch configurations that we can use to analyze the fatbin file and b) to increase the number of launch configurations to try to reduce the quantity of bytes that is necessary to transfer during the execution of the B part of the fatbin file.

  Smaller the quantity of bytes that is necessary to transfer during the execution of a fatbin file, smaller the probability that the bandwidths and the latencies of the GPU memories can slow down the execution of the B part of the fatbin file, this independently of the warp scheduling - remember that we can not choose the warp scheduling or know which it will be at the next execution of the fatbin file, this also if we use the same launch configuration to execute the fatbin file;

- Let us call $F_{f_i}$ the original fatbin file and $F_{f_o}$ the output fatbin file of the possible, but sometimes not wanted, process of transformation, of the reading/writing mechanism of the original fatbin file $F_{f_i}$.

  $F_{f_o}$ - see 8.4 for an explanation for the following different points - a) is equal to the original fatbin file $F_{f_i}$ if no transformation process is used, b) is practically equal to the original fatbin file $F_{f_i}$, if the procedure $P_1$ is used, to transform, the original fatbin file $F_{f_i}$, with a reading/writing mechanism of category $Cat_1$, in a fatbin file, with a reading/writing mechanism of category $Cat_2$, c) is practically equal to the original fatbin file $F_{f_i}$, if the procedure $P_2$ is

used, to transform, the original fatbin file $F_{f_i}$, with a reading/writing mechanism of category $Cat_1$ or $Cat_2$, in a fatbin file, with a reading/writing mechanism of category $Cat_3$ and it necessary to apply, in the procedure $P_2$, the set of transformations and changes $TAC_1$ and d) could be very different from the original fatbin file $F_{f_i}$, if the the procedure $P_2$ is used, to transform the original fatbin file $F_{f_i}$, with a reading/writing mechanism of category $Cat_1$ or $Cat_2$, in a fatbin file, with a reading/writing mechanism of category $Cat_3$ and it is necessary to apply, in the procedure $P_2$, the set of transformations and changes $TAC_2$;

- Because the number of ELF registers $n_{er}^{F_f}$ of a fatbin file $F_f$ is fixed, it is not always possible to force the gigathread scheduler to evenly distribute the GPU thread blocks to the streaming multiprocessors - this happens when the number of GPU thread blocks $B$ we want in each streaming multiprocessor times the number of warpsW of each GPU thread block times the number of GPU threads of each warp times the number of ELF registers $n_{er}^{F_f}$ of the fatbin file $F_f$ is smaller than half the number of hardware registers in a streaming multiprocessor.

  Generating the set $S_{F_f}^1$ of fatbin files with their B parts equivalent to the B part of the fatbin file $F_{f_o}$ - equal number, type and order of ELF instructions and equal dependences among ELF registers - but a greater number of ELF registers - some of them will be dummy ELF registers - allow us to use more launch configurations than the case where we are only considering the fatbin file $F_{f_o}$.

  This happens because some launch configurations previously impossible to use with the original fatbin file $F_{f_i}$ and the fatbin file $F_{f_o}$ - $B \cdot W \cdot 32 \cdot n_{er}^{F_{f_i}}$ and $B \cdot W \cdot 32 \cdot n_{er}^{F_{f_o}}$ smaller than half of the number of hardware registers in a streaming multiprocessor - now, when used to execute one or more fatbin files in the set $S_{F_f}^1$, give us the guarantee that the gigathread scheduler is going to evenly distribute the GPU thread blocks to the streaming multiprocessors, this because $B \cdot W \cdot 32 \cdot n_{er}^{F_f}$ is greater than half the number of hardware registers in a streaming multiprocessor and smaller or equal the number of hardware registers in a streaming multiprocessor;

- It is not enough, for a launch configuration of a fatbin file, to be considered as one of the launch configurations that we use to analyze the fatbin file, to give us the guarantee a) that the gigathread scheduler is going to evenly distribute the GPU thread blocks to the streaming multiprocessor, b) that this even distribution is done by the gigathread scheduler at the beginning of the execution of the GPU code of the fatbin file and c) that each streaming multiprocessor will have an even number of resident warps.

  A launch configuration satisfying a), b) and c) determines the number of resident warps in each streaming multiprocessor during the execution of the fatbin file, but some couples ( dependence distance , number of resident warps in a streaming multiprocessor ), in the table - calculated during the extraction of the local streaming multiprocessor ELF architectural features in 7.6.2 - of the maximum ending time differences of the couples ( fatbin file = dependence distance , launch configuration = number of resident warps in a streaming multiprocessor ), and so also in the table - calculated during the extraction of the local streaming multiprocessor ELF architectural features in 7.6.2 - of the maximum absolute time differences between maximum ending time and maximum starting time differences of the couples ( fatbin file = dependence distance , launch configuration = number of resident warps in a streaming multiprocessor ), show the presence of a phenomenon of load unbalancing - a value greater

than 300 function unit clock cycles - for the warp scheduling in the streaming multiprocessors - load unbalancing that we can not fix and that is important to avoid.

Analyzing the B part of a fatbin file we therefore build its set of dependence distances $S_{dd}$. Given a launch configuration satisfying a), b) and c), we build the set of all the couples ( dependence distance in $S_{dd}$ , launch configuration = number of resident warps in a streaming multiprocessor ) and if also only one of the couples of this set is one of the couples that, in the table - calculated during the extraction of the local streaming multiprocessor ELF architectural features in 7.6.2 - of the maximum ending time differences of the couples ( fatbin file = dependence distance , launch configuration = number of resident warps in a streaming multiprocessor ) and so also in the table - calculated during the extraction of the local streaming multiprocessor ELF architectural features in 7.6.2 - of the maximum absolute time differences between maximum ending time and maximum starting time differences of the couples ( fatbin file = dependence distance , launch configuration = number of resident warps in a streaming multiprocessor ), shows the presence of the phenomenon of load unbalancing for the warp scheduling then we can not use the launch configuration to analyze the fatbin file.

The launch configurations - of a fatbin file - satisfying a), b) and c), for which all the couples ( dependence distance in $S_{dd}$ , launch configuration = number of resident warps in a streaming multiprocessor ) do not show the presence of a phenomenon of load unbalancing for the warp scheduling, compose the set of the launch configurations $S_{lc}$ that we use to analyze the fatbin file;

- Because we do not want to loose, at cause of their $S_{dd}$s, good candidates - fatbin files - for the analysis/analyses of an original fatbin file $F_{f_i}$, after the creation of a set of fatbin files $S^1_{F_f}$, to increase the number of launch configurations that we can use for the analysis/analyses - this compared to the case of those that is possible to use if we only consider the original fatbin file $F_{f_i}$ - we take, one at the time, each one of the fatbin files in the set $S^1_{F_f}$ and generate for each one of them a set $S_A$ of analogous fatbin files - each fatbin file of each set $S_A$ has in its B part one of the possible different logically correct orders of the ELF instructions in the B part of the fatbin file used as generator for the set $S_A$ and remember that the fatbin files in the set $S^1_{F_f}$ have their B parts equivalent to the B part of the fatbin file $F_{f_o}$. The fatbin files of each set $S_A$ probably have $S_{dd}$s different from the $S_{dd}$ of the original fatbin file $F_{f_i}$ and from the $S_{dd}$ of the fatbin file $F_{f_o}$ and so probably different from the $S_{dd}$s of the fatbin files in the set $S^1_{F_f}$. Different $S_{dd}$s increase the probability that, if a launch configuration could not be used with a fatbin file in the set $S^1_{F_f}$, now it can be used with at least one of the analogous fatbin files of the set $S_A$ created using as generator one of the fatbin files in the set $S^1_{F_f}$.

  The set, given by the union of all the set $S_A$, is the set $S^2_{F_f}$. For each fatbin file of the set $S^2_{F_f}$ we calculate its set of launch configuration $S_{lc}$ to use to analyze the fatbin file. The fatbin files with an empty $S_{lc}$ are eliminated by the set $S^2_{F_f}$.

Having 1) generated the set $S^2_{F_f}$ of fatbin files and 2) determined for each one of the fatbin files in the set $S^2_{F_f}$ its set of launch configurations $S_{lc}$ to use for its analysis, it is however not enough to start to analyze the fatbin files in the set $S^2_{F_f}$. To understand the possible analysis/analyses that can be executed on the fatbin files in the set $S^2_{F_f}$, it is first effectively necessary to talk of the possible warp scheduling policies executed by the warp schedulers in the streaming multiprocessors.

# Chapter 9

# Warp Scheduling Policies

## 9.1 Introduction

In the previous chapter we have described the steps necessary a) to transform the original fatbin file $F_{f_i}$ in a fatbin file $F_{f_o}$ - fatbin file $F_{f_o}$ that could have or not a reading/writing mechanism, 8.2.3, of category different from that of the reading/writing mechanism of the fatbin file $F_{f_i}$ - and b) to determine the launch configurations in the sets of launch configurations $S_{lc}$ - one set per fatbin file in the set $S^2_{F_f}$ - to use to analyze the fatbin files in the set $S^2_{F_f}$, set $S^2_{F_f}$ produced from the fatbin files $S^1_{F_f}$, set $S^1_{F_f}$ produced from the fatbin file $F_{f_o}$, fatbin file $F_{f_o}$ produced from the original fatbin file $F_{f_i}$. The points a) and b) increase the probability to get a greater lower bound on the real ELF code efficiency of the original fatbin file $F_{f_i}$.

The analysis/analyses to execute on each one of the the fatbin files in the set $S^2_{F_f}$ are determined a) by the fact that the fatbin file $F_{f_o}$ is or not in the subset $SS_{A_1}$ of all the possible fatbin files - if the fatbin file $F_{f_o}$ is in the subset $SS_{A_1}$ then the fatbin files in the set $S^2_{F_f}$ are eligible for the execution of the analysis $A_1$ described in 12 - and b) by the reader's goals. The factors that determine whether the fatbin file $F_{f_o}$ is or not in the subset $SS_{A_1}$ are five, the first of them is the warp scheduling policy executed by the warp schedulers in the streaming multiprocessors.

While we will talk of the last four, of the five factors, in the next chapter, in this chapter we talk of the first of the five factors, the warp scheduling policy executed by the warp schedulers in the streaming multiprocessors. We start explaining - thanks at the results that we got in 7.6.2 - what is reasonable to assume being true - and because other possibilities are unlikely - about the warp scheduling policy. Later we explain because it is impossible to know the implementation details of the warp scheduling policy executed by the warp schedulers in the streaming multiprocessors. Next we describe the mechanisms and the dynamics of the probable warp scheduling policy - the cycling policy - executed by the warp schedulers in the streaming multiprocessors, which are the reasons supporting the fact that the cycling policy is probably the warp scheduling policy executed by the warp schedulers in the streaming multiprocessors and how we justify the starting time differences we got in 7.6.2 for the quantification of the local streaming multiprocessor PTX and ELF architectural features, if it is true that the warp schedulers in the streaming multiprocessors execute the warp scheduling cycling policy. We therefore talk about the possibility that instead of the warp scheduling cycling policy other policies are executed, its consequences and why we believe this is unlikely. Finally we talk of the advantages and disadvantages of the warp scheduling cycling policy.

## 9.2 What is Reasonable to Assume being True

In 5.6 we describe the mechanism that has to be implemented in the GPU to manage the resident warps in a streaming multiprocessor but we do not explain in detail how the 2 warp schedulers in each streaming multiprocessor specifically select the warps among the warps that will be available to be scheduled at the next warp scheduler clock cycle.

To understand because it is not possible to get such implementation details and therefore what is reasonable to assume being true about the implementation details, it is necessary to talk of the starting and ending time differences that we got for the warp schedulings in 7.6.2.

### 9.2.1 Very Simple Fatbin Files

Each one of the B parts, of the fatbin files, used in 7, for the discovery, understanding and quantification of the not disclosed GPU behaviors, has a) in its for loop only one ELF instruction configuration, repeated many times, using the same or different ELF registers - there are also three ELF instructions in the for loop necessary to iterate on the for loop but these three ELF instructions are not important - b) a very simple structure - some ELF instructions before the for loop, a for loop, some others ELF instructions after the for loop - and c) a very simple control flow - no branches, only one for loop, all the GPU threads executing the same number and type of ELF instructions.

### 9.2.2 Executions with Load Balancing

Let us define $G_s$ - good set - the set of the couples ( dependence distance = fatbin file , number of resident warps in a streaming multiprocessor = launch configuration ) with all the three values, in the tables, created for the quantification of the local streaming multiprocessor ELF architectural features, $T_a$ of the maximum absolute time differences between maximum ending time and maximum starting time differences of the couples ( dependence distance , number of resident warps in a streaming multiprocessor ), $T_b$ of the maximum ending time differences of the couples ( dependence distance , number of resident warps in a streaming multiprocessor ) and $T_c$ of the maximum starting time differences of the couples ( dependence distance , number of resident warps in a streaming multiprocessor ), smaller than 300 function unit clock cycles. The couples in $G_s$ have load balancing at global level - same number of GPU thread blocks in each streaming multiprocessor and so same number of resident warps in each streaming multiprocessor - and load balancing at local level - the number of resident warps in each streaming multiprocessor is even and the combination ( dependence distance = fatbin file , number of resident warps in a streaming multiprocessor = launch configuration ) does not create load unbalancing, for the warp scheduling, during the execution of the fatbin file of the couple, using the launch configuration of the couple.

### 9.2.3 Probably True Things about the Warp Scheduling

Also whether it was not possible to get the state of advancement of the warps during the execution of the for loops of the B parts of the fatbin files - this because otherwise the quantification of the local streaming multiprocessor PTX and ELF architectural features would be been harder, whether not impossible, to prove correct - it is reasonable to assume that the warp schedulers, for the execution of the for loops of the very simple B parts of the fatbin files, used for the quantification of the local streaming multiprocessor PTX and ELF architectural features, move forward all the warps together

because all the three values, in the tables $T_a$, $T_b$ and $T_c$, of each couple ( dependence distance , number of resident warps in a streaming multiprocessor ) of the set $G_s$, are always smaller than 300 function unit clock cycles, this independently of the quantity of work that each GPU thread has to execute.

That, for all the couples ( dependence distance , number of resident warps in a streaming multiprocessor ) of the set $G_s$, the warp schedulers moves forward together all the resident warps in a streaming multiprocessor, it is in our opinion true also whether we could not study, during the executions, the advancement of the warps inside the for loops, of the B parts, of the fatbin files, because, given $W$ resident warps in a streaming multiprocessor, if, for all the possible couples of warps of the set $W$, we study the differences between the starting time difference of the 2 warps of each couple and the ending time differences of the 2 warps of the couple then we can see that a) such differences are only slightly different - few clock cycles in almost the totality of the cases and a little more in the remaining cases - and b) that the warp of each couple that starts first is going also to finish first and so, considering all the resident warps $W$ in a streaming multiprocessor, 1) that the first warp to start to execute the for loop, of the B part, of a fatbin file, it is also the first to finish to execute it and that the last warp to start to execute the for loop, of the B part, of a fatbin file, is also the last to finish to execute it and 2) that probably all the distances between all the warps are going to stay almost constant for the whole execution of the for loop, of the B part, of a fatbin file, considering that the starting time difference and the ending time difference, of the warps, of each couple, are equal or almost equal - the difference, between the starting time difference and the ending time difference, of the warps, of each couple, is effectively of few clock cycles, this independently of a) the ELF instruction configuration, b) the dependence distance and c) the fact that the execution of the for loop, of the B part, of the fatbin file is or not slowed down by the scheduling waiting times, the dependence waiting times or the overhead time for the management of the warps.

### 9.2.4 Because Other Possibilities are Unlikely

For the couples ( dependence distance , number of resident warps in a streaming multiprocessor ), of the set $G_s$ - such couples are generated using the fatbin files used for the quantification of the local streaming multiprocessor ELF architectural features in 7.6.2 - the possibility 1) that the warp schedulers were scheduling more often a subset of the resident warps in a streaming multiprocessor, after the beginning of the execution of the for loop. of the B part, of a fatbin file, for only later to make all the remaining resident warps in the streaming multiprocessor catch up the leading warps, repeating this accordion effect for the whole execution of the for loop and 2) that the warp schedulers would be able to make all the resident warps in the streaming multiprocessor finish the execution of the for loop, of the B part, of the fatbin file, in such way to get an ending time difference, among all the warps, and a starting time difference, among all the warps, both smaller than 300 function unit clock cycles, this independently of a) the quantity of ELF instructions executed by each GPU thread, b) the ELF instruction configuration and c) the fact that the execution of the B part of the fatbin file was or not slowed down by the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps, is very unlikely.

## 9.3 Impossibility of Knowing the Truth

Also whether we are able 1) to get the wanted ELF algorithmic implementations and 2) to program, in the real assembly, executed by the GF100 architecture, there is no way of knowing, also whether it seems reasonable to assume so, whether the things said in 9.2.4 are true in the real world. Because there is no way to know whether the previous things are true in the real word, we need to give to the reader the possibility of choice between two different cases. What the reader believes is true - which are the implementation details of the warp scheduling policy executed by the warp schedulers in the streaming multiprocessors and so which is the warp scheduling policy executed by the warp schedulers in the streaming multiprocessors - is one of the five factors that determines whether the fatbin file is $F_{f_o}$ is in the subset $SS_{A_1}$ and therefore partially determine - partially because it is only one of the five factors - whether the fatbin files in the set $S^2_{F_f}$ are eligible for the analysis $A_1$ - this is due to the way we generate the fatbin files in the set $S^2_{F_f}$.

We consider only one set $S^{wsp}_{id}$ of implementation details for the possible warp scheduling policies that could be executed by the warp schedulers in the streaming multiprocessors and the reader has only two possible choices, each one with its specific consequences:

- If the reader believes $S^{wsp}_{id}$ is the set of implementation details, implemented for the warp scheduling policy, executed by the warp schedulers in the streaming multiprocessors, at least in the cases when the execution, of the B part, of a fatbin file, is not be slowed down by the bandwidths and the latencies of the GPU memories, then the warp scheduling policy, executed by the warp schedulers in the streaming multiprocessors, is what we call the cycling policy;

- If the reader believes that also only one of the implementation details in the set $S^{wsp}_{id}$ is not implemented or is different for the warp scheduling policy executed by the warp schedulers in the streaming multiprocessors then all the possible warp scheduling policies, that could be executed by the warp schedulers in the streaming multiprocessors, do not matter which they are because the consequences, for the execution of any warp scheduling policy different from the cycling policy, are the same - see subsection X.

## 9.4 Cycling Policy - The Probable Warp Scheduling Policy

This is the easiest of the warp scheduling policies that it could be implemented and it is the warp scheduling policy that we believe it is executed at least in the cases when the execution, of the B part, of a fatbin file, is not be slowed down by the bandwidths and the latencies of the GPU memories.

We start describing the mechanisms executed by the 2 warp schedulers in a streaming multiprocessor and the dynamics, originated by the warp scheduling cycling policy, between the 2 warp schedulers in a streaming multiprocessor We therefore continue explaining the change, in the order of execution of the mechanisms, that can happen, between the 2 warp schedulers in a streaming multiprocessor, if one of the 2 warp scheduler in a streaming multiprocessor can schedule the warp that it is pointing and instead the other warp scheduler in the streaming multiprocessor can not. Finally we conclude the section explaining why we believe the warp scheduling cycling policy is the warp scheduling policy that is implemented and executed by the warp schedulers in the streaming multiprocessors.

### 9.4.1 Mechanisms and Dynamics of the Warp Scheduling Cycling Policy

What follows are the mechanisms executed by the 2 warp schedulers in a streaming multiprocessor and the dynamics, originated by the warp scheduling cycling policy, between the 2 warp schedulers in a streaming multiprocessor.

If the warp scheduling cycling policy is the warp scheduling policy executed by the warp schedulers in the streaming multiprocessors then the GPU is able to establish an order among the resident warps in a streaming multiprocessor. The 2 warp schedulers in a streaming multiprocessor, at each moment, consider 2 different warps in the order and there is a pointer that points to a warp in the order that is not any of the two warps in the order that are considered by the 2 warp schedulers in the streaming multiprocessor. When a warp scheduler schedules a warp, the pointer assigns the warp that it is pointing to the warp scheduler. If the pointer points to the last warp in the order then, after the assignment, the pointer points to the first warp in the order. If the pointer is not pointing to the last warp in the order then, after the assignment, the pointer is increased of one to point to the next warp in the order. After the beginning of a warp scheduler clock cycle, 1 of the 2 warp schedulers checks whether the warp that it is pointing is available to be scheduled at the next warp scheduler clock cycle:

- If the warp, that the warp scheduler is pointing, is not available to be scheduled at the next warp scheduler clock cycle then the warp scheduler does nothing and checks again the warp at the next warp scheduler clock cycle;
- If the warp, that the warp scheduler is pointing, is available to be scheduled at the next warp scheduler clock cycle then the warp scheduler decides to schedule the warp at the next warp scheduler clock cycle and updates the state of the other resident warps in the streaming multiprocessor. Possible examples:
  - If the warp, that the warp scheduler is pointing, has to be scheduled on the group of 4 special function units and the real ELF instruction streaming multiprocessor best average performance per clock cycle of its next warp ELF instruction, that has to be executed, is equal to 2, then 16 function units clock cycles - 32, the number of GPU thread in a warp, divided 2, the real ELF instruction streaming multiprocessor best average performance per clock cycle - are necessary to execute the next warp ELF instruction of the warp and so for the next 8 warp scheduler clock cycles - this because a warp scheduler clock frequency is half of a function unit clock frequency - the group of 4 special function units will not be available and therefore all the resident warps in the streaming multiprocessor, that require, for the execution of their next warp ELF instruction, the group of 4 special function units, will be made not available to be scheduled for the next 8 warp scheduler clock cycles;
  - If the warp, that the warp scheduler is pointing, has to be scheduled on 1 of the 2 groups of 16 CUDA cores and the real ELF instruction streaming multiprocessor best average performance per clock cycle of its next warp ELF instruction, that has to be executed, is equal to 4, then we know, from 7.6.2, that some not disclosed hardware resources shared among the 2 groups of 16 CUDA cores are used at least for the execution of warp ELF instructions equal to the warp ELF instruction that is necessary to execute for the warp - however these same not disclosed shared hardware resources could be used for the execution of other different warp ELF instructions too - and therefore all the resident

warps in the streaming multiprocessor, that require, for the execution of their next warp ELF instruction, the use of the same not disclosed shared hardware resources, will be made not available to be scheduled for the next 4 warp scheduler clock cycles - 32, the number of GPU threads, divided 4, the real ELF instruction streaming multiprocessor best average performance per clock cycle, divided 2, the warp scheduler clock frequency is half of a function unit clock frequency - independently of which of the 4 groups of function units in a streaming multiprocessor are necessary to execute their next warp ELF instruction.

After 1 of the 2 warp schedulers has executed the procedure, the same procedure is therefore repeated by the other warp scheduler. After the last warp scheduler has executed the procedure, each one of the 2 warp scheduler knows whether, at the next warp scheduler clock cycle, it will schedule the warp that it is pointing. If a warp scheduler will schedule the warp that it is pointing then, after the scheduling of the warp at the next warp scheduler clock cycle, the warp scheduler get from the pointer the new warp that the warp scheduler has to consider and repeat the procedure. If a warp scheduler will not schedule the warp that it is pointing then, at the next warp scheduler clock cycle, it will repeat the procedure for the same warp.

### 9.4.2 Change of the Order of Execution of the Mechanisms

The warp scheduler that executes as first the procedure described in the previous subsection could be the second to execute the same procedure at the next warp scheduler clock cycle. This depends on a) whether both the 2 warp schedulers in the streaming multiprocessor will schedule or not, at the next warp scheduler clock cycle, the warps that they are pointing or 2) whether 1 of the 2 warp schedulers in the streaming multiprocessor is going to schedule, at the next warp scheduler clock cycle, the warp that it is pointing, while the other warp scheduler no. The two possible mutually exclusive cases are therefore the following:

- If, at the next warp scheduler clock cycle, the 2 warp schedulers are both going or not to schedule the 2 warps that they are pointing then the order of precedence for the execution of the procedure described in the previous subsection is going to remain the same for the 2 warp schedulers;
- If, at the next warp scheduler clock cycle, 1 of the 2 warp schedulers is going to schedule the warp that it is pointing while the other warp scheduler is not going to schedule the warp that it is pointing then the warp scheduler that is not going to schedule the warp that it is pointing will execute as first the procedure described in the previous subsection.

### 9.4.3 Possibility of a Time Difference Between Warp Schedulers

In the discussions till now, we have supposed that the 2 warp schedulers in each streaming multiprocessors have their clock frequencies synchronized - the clock cycle $x$ of one of the two warp schedulers, $CC_x^{ws_1}$, is happening at the same moment of the clock cycle $x$ of the other warp scheduler, $CC_x^{ws_2}$. This could be true or not, but if not, it does not matter, every discussions done till now is however valid.

Because the clock frequency of the function units is twice the clock frequency of the warp schedulers, it could effectively be that, at the clock cycle $y$ of the function units - $CC_y^{fu}$ - only one

of the 2 warps schedulers in a streaming multiprocessor is at its $CC_x$ clock cycle. If this is the case, the warp scheduler schedules or not the warp that it is pointing, whether necessary get from the pointer the new warp that the warp scheduler has to consider, executes the procedure described in 9.4.1 and waits the function unit clock cycle $CC^{fu}_{y+2}$. At the same time, at the function unit clock cycle $CC^{fu}_{y+1}$, the other warp scheduler in the streaming multiprocessor could be at its $CC_z$ clock cycle. If this is the case, the other warp scheduler in the streaming multiprocessor schedules or not the warp that it is pointing, whether necessary get from the pointer the new warp that the warp scheduler has to consider, executes the procedure described in 9.4.1 and waits the function unit clock cycle $CC^{fu}_{y+3}$.

In this situation, the same warp scheduler clock cycle $CC_x$ does not happen at the same time for the 2 warp schedulers in a streaming multiprocessor but with a time difference greater than one function unit clock cycle.

All the previous discussions are valid also in this case. Furthermore, this case requires an hardware and/or software logic that is simpler than the case where the 2 warp schedulers in a streaming multiprocessor are supposed to have their clock frequencies synchronized and so this case a) probably requires a smaller die area to implement the necessary hardware logic and b) probably has a smaller probability to get some hardware and/or software bugs. The hardware and/or software logic is simpler than that required, to execute the procedure in 9.4.1, supposing the clock frequencies of the 2 warp schedulers in a streaming multiprocessor are synchronized, because:

- It is not necessary to switch the order of execution, of the procedure described in 9.4.1, between the 2 warp schedulers in a streaming multiprocessor, if at the previous warp scheduler clock cycle, the 2 warp schedulers do not have, both, scheduled or not, the 2 different warps that each one of them were pointing;
- The speed, necessary a) for the checks for the decisions of the warp schedulers and b) for the updating of the state of the resident warps a the streaming multiprocessor, could be half of the speed instead necessary in the case the clock frequencies of the 2 warp schedulers in a streaming multiprocessor are synchronized - think at the case where, at each function unit clock cycle, only 1 of the 2 warp schedulers in a streaming multiprocessor executes the procedure described in 9.4.1.

### 9.4.4 Supporting Reasons for the Warp Scheduling Cycling Policy

The warp scheduling cycling policy is the easiest warp scheduling policy to implement and in our opinion is the warp scheduling policy that is implemented because it is hard to believe that, implementing any other different warp scheduling policy, it is possible to get something better.

The reasons because we say that it is hard to believe that, implementing any other different warp scheduling policy, it is possible to get something better, it is because the designers of the GF100 architecture can not know 1) which ELF instructions a fatbin file will have in its B part and the order of the ELF instructions in the B part of the fatbin file, 2) the structure of the B part of a fatbin file - which types of loops?, how many?, nested?, etc. - 3) the control flow of the B part of a fatbin file - branches?, synchronizations?, etc. - 4) from where the GPU threads will read the data and to where the GPU threads will write the results - GPU global memory?, which cache level?, shared memory?, hardware registers? - and 5) how the GPU threads are going to read/write the data/results - consecutively?, using pointers?, etc. - and therefore:

- The designers would have a very hard time to prove why a different warp scheduling policy should give smaller execution times for the fatbin files - this because a) the number of different types of fatbin files that nvcc can generate and b) all the possible different launch configurations an user can use to execute a fatbin file, creates an incredibly huge number of couples ( fatbin file , launch configuration ) to consider for the proof;

- The designers would have a very hard time to show experimentally that a warp scheduling policy, different from the warp scheduling cycling policy, would give shorter execution times, for the B parts, of fatbin files, in general, this because, also supposing the designers could be able to show that a different warp scheduling policy could give shortest execution times for at least a small subset of all the possible couples ( B part of a fatbin file , launch configuration ), the designers would face big challenges about the generalization of the results to other couples;

- The designers would have a very hard time to prove that the cost payed for the increased complexity of the control logic, necessary to execute a warp scheduling policy different from the warp scheduling cycling policy, and so the cost payed for the inevitably bigger die area required for the implementation of a different, more complex, control logic than that required for the execution of the warp scheduling cycling policy, are worth.

### 9.4.5 Justifying the Starting Time Differences

From 9.2.2 we know that in a streaming multiprocessor, from the moment when the first warp or the first two warps in a streaming multiprocessor start to execute the for loop of the B part of each fatbin file, to the moment when the last warp or the last two warps in a streaming multiprocessor start to execute the for loop of the B part of the same fatbin file, a maximum of 300 function unit clock cycles passes for each couple ( dependence distance , number of resident warps in a streaming multiprocessor ) of the set $G_s$.

If the warp schedulers in the streaming multiprocessors would execute the warp scheduling policy then, because the maximum number of resident warps in a streaming multiprocessor, during the execution of each fatbin file used for the quantification of the local streaming multiprocessor PTX and ELF architectural features, is 32, it would seem that the maximum starting time differences should be much smaller than 300 clock cycles and so it is necessary to justify the values of the maximum starting time differences got for the couples ( dependence distance , number of resident warps in a streaming multiprocessor ) of the set $G_s$. The following three things, about the B parts of the fatbin files used in 7, are true:

- To synchronize the different number of GPU threads used for the executions of the B part of a fatbin file, we use a membar.gl ELF instruction - an ELF global memory barrier synchronization instruction;

- Just after the membar.gl ELF instruction, the same location, of the GPU global memory, that was overwritten with different random values, by all the GPU threads used for the execution of the B part of the fatbin file, just before the membar.gl ELF instruction, is read by all the GPU threads - this is necessary to force, all the GPU threads, used for the execution of the B part of the fatbin file, to become synchronized at the membar.gl ELF instruction;

- Just after the read ELF instruction and just before to start to execute the for loop, of the B part, of the fatbin file, each GPU thread, used to execute the B part of the fatbin file, gets the GPU global clock cycle - to get the GPU global clock cycle it is necessary to execute, for each GPU thread, a set of 6 different and consecutive ELF instructions, with 3, of the 6 ELF instructions, that a) are consecutive and b) use a total of 2 different special ELF registers that b.1) correspond probably to 2 different special hardware registers and b.2) could be shared by all the streaming multiprocessors in the GPU.

The previous things generate therefore the following three cases useful to explain because we got some maximum starting time differences, for the couples ( dependence distance , number of resident warps in a streaming multiprocessor ) of the set $G_s$, that would seem do not make sense, if the warp schedulers in the streaming multiprocessors are executing the warp scheduling cycling policy:

- Case $C_1$. The fact, that all the GPU threads, after the membar.gl ELF instruction, read the same location of the GPU global memory, implies the transfer, of the bytes, in that location, from the GPU global memory to the cache l2 and later from the l2 cache to the l1 cache of each one of the streaming multiprocessors used for the execution of the B part of the fatbin file.

  While for the movement of the bytes, from the GPU global memory to the l2 cache, the waiting time, due to the latency of the GPU global memory, will be shared by all the GPU threads used for the execution of the B part of the fatbin file, the waiting time, necessary to move the bytes, from the l2 cache to the l1 cache of each one of the streaming multiprocessors used for the execution of the B part of the fatbin file, could be different for the GPU threads in different streaming multiprocessors and so it could create a first time difference among the resident warps, in a streaming multiprocessor, used for the execution of the B part of the fatbin file - this case however is not applicable to the couples ( dependence distance , number of resident warps in a streaming multiprocessor ) of the set $G_s$ because the couples of the set $G_s$ are determined using the results got for the quantification of the local streaming multiprocessor PTX and ELF architectural features and for the quantification of the local streaming multiprocessor PTX and ELF architectural features only a single streaming multiprocessor is used for the execution of the B parts of the fatbin files;

- Case $C_2$. It is impossible to know the specific details that are used by the GF100 architecture to update and read the 2 special hardware registers used in 3 of the 6 ELF instructions that is necessary to execute for each GPU thread to get the GPU global clock cycle.

  Because the GPU global clock is at 64 bits but the GF100 architecture has only hardware registers at 32 bits then it could be that all the 3 or some of the 3 consecutive ELF instructions that a) use the 2 special ELF registers and b) are necessary to get the GPU global clock cycle, need to be executed consecutively for each warp and because a) each one of the 3 ELF instructions using the 2 special ELF registers has its scheduling waiting time, its dependence waiting times for its operands and its result and its overhead time for the management of the warps and b) the reading and writing times, for a same special ELF register, could be different for different ELF instructions, then the total number of clock cycles necessary for their execution could be not indifferent.

Whether to all this we add the fact that all the resident warps in a streaming multiprocessor have to execute the 3 consecutive ELF instructions, using the 2 special ELF registers, and that probably no other warp can execute them when they are executed for another warp then, when the number of resident warps in a streaming multiprocessor is approaching 32, we can very easily to get a maximum starting time difference of 300 function units clock cycles.

- Case $C_3$. After a membar.gl ELF instruction, it could be that not all the GPU threads used for the execution of the B part of the fatbin file are released at the same moment after that for the last warp or warps, used for the execution of the B part of the fatbin file, the membar.gl ELF instruction has been executed.

  It could be in fact possible that a quantity of time - with its variabilities - is required to release all the warps used for the execution of the B part of the fatbin file and that during the process the 2 warp schedulers in each streaming multiprocessor start to schedule the warps released in the streaming multiprocessor, cycling on them, as happen a) at the beginning of the execution of the B part of a fatbin file, when not all the GPU thread blocks, that should be assigned to a streaming multiprocessor, have been assigned by the gigathread scheduler to the streaming multiprocessor or b) when, always at the beginning of the execution of the B part of the fatbin file, all the warps of a GPU thread block, assigned to a streaming multiprocessor, have not yet been made available to be considered by the pointer - in the streaming multiprocessor - that assigns the warps, to the 2 warp schedulers, during the execution of the warp scheduling cycling policy.

## 9.5 The Possibility that Other Policies are Executed

If the reader believes a) that there is also only one, of the details, of the warp scheduling cycling policy, that is implemented in a different way or b) that there are other details implemented, but not considered, in the discussion of the warp scheduling cycling policy, that could imply a different advancement of the warps from that implied by the warp scheduling cycling policy, when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, then the warp scheduling policy, executed by the warp schedulers, in the streaming multiprocessors, would be different from the warp scheduling cycling policy, when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories.

To understand why this could be true and so why we can not be 100% sure that the warp scheduling cycling policy is the warp scheduling policy that is used by the 2 warp schedulers, to execute the B parts of general fatbin files, when the executions of the B parts of general fatbin files are not slowed down by the bandwidths and the latencies of the GPU memories, we need to consider the simplicity of the B parts of the fatbin files, used in 7, for the discovery, understanding and quantification of the not disclosed GPU behaviors

### 9.5.1 Generalization of Results about the Starting Time Differences

We know, thanks to results got in 7.6.2, that all the resident warps in a streaming multiprocessor, in all the possible cases - couples ( dependence distance , number of resident warps in a streaming multiprocessor ) in $G_s$ or not - are scheduled at least one time, in a time window smaller than 300

function unit clock cycles, after all them are forced to be synchronized and so are forced to being at the same point, of the ELF codes of the B parts of the fatbin files, just before the beginning of the for loop in the B parts of the fatbin files - this independently of a) the ELF instruction configuration and b) the fact that the execution of the B part of the fatbin file is or not slowed down by the scheduling waiting time, the dependence waiting time or the overhead time for the management of the warps.

This in our opinion is going to be the case also for the executions of the B parts of fatbin files very different from the B parts of the fatbin files used in 7 and therefore with a) many different ELF instructions, b) many different structures and c) many different control flows, compared to those of the B parts of the fatbin files used in 7.

### 9.5.2 Difficulty to Generalize the Results about the Ending Time Differences

If the warp scheduling policy executed by the 2 warp schedulers in a streaming multiprocessor, when the execution of the B part of the fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, is different by the warp scheduling cycling policy, then the results got in 7.6.2 for the couples ( dependence distances , number of resident warps in a streaming multiprocessor ) of the set $G_s$ - the couples immune by load balancing problems due to GPU the hardware design, 9.2.2 - however confirm that a) the maximum ending time differences of the couples are smaller than 300 function unit clock cycles and b) that the starting time differences and the ending time differences for each couple of resident warps in a streaming multiprocessors are practically the same - at maximum few clock cycle of difference. The points a) and b) show that, at least for the executions of very simple B parts as those of the fatbin files used for the quantification of the local streaming multiprocessor PTX and ELF architectural features in 7.6.2, it is reasonable to assume that the warp schedulers are moving forward all the warps together in the way implied by the warp scheduling cycling policy.

However we can not be sure that the maximum ending time differences of the couples ( dependence distances , number of resident warps in a streaming multiprocessor ) of the set $G_s$ - the couples immune by hardware design problems about the warp scheduling - are going to stay smaller than 300 function unit clock cycles also for the executions of B parts of fatbin files very different from the B parts of the fatbin files used in 7 and therefore that the results, about the points a) and b), got for the executions of the very simple B parts of the fatbin files, used in 7, for the quantification of the local streaming multiprocessor PTX and ELF architectural features, are generalizable to the executions of B parts of fatbin files with 1) many different ELF instructions, 2) many different structures and 3) many different control flows.

### 9.5.3 Consequences of the Reader's Choice

The results about the points a) and b) of the previous subsection are generalizable if the reader believes that the warp scheduling cycling policy is the warp scheduling policy that is executed by the warp schedulers in the streaming multiprocessors when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, but if the reader believes that the warp schedulers in the streaming multiprocessors instead execute another warp scheduling policy then we can not use, to analyze a fatbin file, the idea that the warp schedulers, in a streaming multiprocessor, are moving forward all the warps together in the way implied by the

warp scheduling cycling policy because, in fact, the reality could be very different from that, also whether the maximum starting times and the maximum ending times, of the couples ( dependence distances , number of resident warps in a streaming multiprocessor ) of the set $G_s$, seem to confirm that the warp schedulers, in a streaming multiprocessor, are moving forward all the warps together in the way implied by the warp scheduling cycling policy, this at least for the executions of very simple B parts as those of the fatbin files used in 7 for the quantification of the local streaming multiprocessor PTX and ELF architectural features.

### 9.5.4 Impossibility to Determine and Understand any Other Policy

Any type of forward movement of the resident warps in a streaming multiprocessor could be possible, if, when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, a warp scheduling policy different from the warp scheduling cycling policy is executed by the 2 warp schedulers in a streaming multiprocessor.

If, when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, the 2 warp schedulers in a streaming multiprocessor execute a warp scheduling policy different from the warp scheduling cycling policy then there is not way a) to prove which is the warp scheduling policy or b) supposing the warp scheduling policy can be determined and understood for a specific couple ( fatbin file , launch configuration ), being able to generalize the results to other couples ( fatbin file , launch configuration ), this because 1) the number of possible different fatbin files to consider is too big, 2) the number of possible different launch configurations that can be used to execute a fatbin file is too big, 3) there is no way to get the implementation details of the warp scheduling policy and 4) the different number of ELF instructions that each GPU thread has to execute - independently of the presence of divergences or not in the ELF code - could have it too, as the type of fatbin file and the type of launch configuration, an influence on the warp scheduling.

### 9.5.5 Why a Policy Different from the Cycling Policy is Unlikely

If, when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, the 2 warp schedulers in a streaming multiprocessor would execute a warp scheduling policy different from the warp scheduling cycling policy, then it could be possible to get a subset of resident warps in the streaming multiprocessor very forward in the execution of the ELF code and a subset of the resident warps in the streaming multiprocessor very behind in the execution of the ELF code. If some warps finish to execute the ELF code of the B part of a fatbin file first of others then greater the difficulty for the warps schedulers to be able to schedule, at each warp scheduler clock cycle, a couple of warps, this not only because, for the warp schedulers, it is harder, with a smaller number of warps, to hide the latencies of the GPU memories, but also because, for the warp schedulers, it is harder, with a smaller number of warps, to hide the scheduling waiting times and the dependence waiting times.

Furthermore, because the GF100 architecture can not know the next group of ELF instructions that is necessary to execute for a warp, if it is possible that some warps can be scheduled more times than others and so move forward in the execution of the ELF code of the B part of a fatbin file then it could be easy, for some warps, to get at a point of the execution, when the warps need to use a data, that the warps previously required to transfer from the GPU global memory, but the

data is not available. If this is the case then such warps can not be scheduled for the next N warp scheduler clock cycles. The following things are true about N:

- N is between 200 and 400 warp scheduler clock cycles, because 400 and 800 function unit clock cycles are the absolute minimum and the absolute maximum latencies of the GPU global memory - [50, p. 87] and [56, p. 67] say 400 and 800 function unit clock cycles, [49, p. 47] and [55, p. 57] say 400 and 600 function unit clock cycles - but a warp scheduler clock frequency is half of a function unit clock frequency.

  The exact value of the latency of the GPU global memory for the transfer of a data depends on the location of the data in the GPU global memory, however when we transfer, from the CPU to the GPU, the variables, the arrays, the vectors and the structures, necessary for the execution of a couple ( fatbin file , launch configuration ) - variables, arrays, vectors and structures that contain the input data and that will contain the output results - we have no way to choose or to force the locations, in the GPU global memory, of the variables, the arrays, the vectors and the structures and so the locations of the input data and the output results.

  The locations of the input data and the output results could be the same or not for the same or for different problem sizes but in any case we can not know, choose or force the locations and so to determine the maximum of the possible latencies of the GPU global memory that we can meet during the execution of the B part of a fatbin file - the maximum of the possible latencies could be smaller than the absolute maximum, this depends on the locations of the variables, the arrays, the vectors and the structures, used for the execution of the fatbin file, in the GPU global memory.

  For these reasons, in the analysis $A_1$, described in 12, when we execute, on a fatbin file, the subanalysis on the bandwidths and the latencies of the GPU memories, we need to use the greatest possible value for the latency of the GPU global memory, the absolute maximum, that is equal to 800 function unit clock cycles that are equivalent to 400 warp scheduler clock cycles;

- N depends on 1) the distance in number of ELF instructions between the ELF instruction of the warp that requires the transfer of the data from the GPU global memory and the first ELF instruction of the warp that needs to use that data, 2) the type and order of the ELF instructions between the ELF instruction of the warp that requires the transfer of the data from the GPU global memory and the first ELF instruction of the warp that needs to use that data - the type and the order determine a lower bound on the minimum number of clock cycles that has to pass before the warp can be scheduled for the execution of the ELF instruction that needs to use the data transfered from the GPU global memory - and 3) the warp scheduling history of all the resident warps in the streaming multiprocessor - number and type of GPU global memory requests to read/write data/results, moments when the GPU global memory requests happens, order and type of ELF instructions that is necessary to execute for the resident warps in the streaming multiprocessor, after that the warp, that requires the execution of the ELF instruction for the transfer of the data from the GPU global memory, has been scheduled for the execution of the ELF instruction for the transfer of the data from the GPU global memory.

Greater the quantity of warp scheduler clock cycles that some warps are not available to be

scheduled, greater the quantity of time that the warp schedulers have to be able to try to hide 1) the latency of the GPU global memory and the latencies of the GPU memories in general, 2) the scheduling waiting times and 3) the dependence waiting times. However, smaller the number of resident warps in a streaming multiprocessor that are available to be scheduled, harder the job of the warp schedulers and so greater the probability that the execution of the B part of a fatbin file is going to get some slowdowns.

## 9.6 Advantages and Disadvantages of the Cycling Policy

The warp scheduling cycling policy present some advantages, compared to any other warp scheduling policy, at the increase of the number of resident warps in a streaming multiprocessor.

Greater the number of warps for a warp scheduling policy different from the warp scheduling cycling policy, greater the potential disadvantage of the warp scheduling policy because it could be easier for the 2 warp schedulers to generate the situations described in the previous subsection therefore increasing the probability that the execution of the B part of a fatbin file is going to get some slowdowns.

If instead the 2 warp schedulers in a streaming multiprocessor execute the warp scheduling cycling policy then, greater the number of resident warps in a streaming multiprocessor, greater the probability for the warp schedulers in a streaming multiprocessor to be able to avoid slowdowns due to 1) the latencies of the GPU memories, 2) the scheduling waiting times of the ELF instructions, 3) the dependence waiting times of the ELF instructions and 4) the overhead for the management of the warps - see below why.

To understand why this happens we need to consider the following things: 1) at each warp scheduler clock cycle not more than 2 warps can be scheduled and 2) each warp ELF instruction that has to be executed for a warp can not be executed in less than 2 function units clock cycles - this because every warp always has 32 GPU thread but each one of the 4 groups of functions units in each streaming multiprocessor has not more than 16 function units.

Let us consider a generic fatbin file - the distances in number of ELF instructions, between ELF instructions that require the transfer of data from the GPU global memory and the first ELF instructions using those data, are therefore constant from launch to launch, but let us execute the generic fatbin file with a different number of resident warps per streaming multiprocessor from launch $L_1$ to launch $L_2$ - in $L_2$ a greater number of warps will be resident in each streaming multiprocessor during the execution of the B part of the fatbin file. The number of function units clock cycles that has to pass, from the moment when a warp is scheduled and needs the execution of one of the ELF instructions that require the transfer of a data from the GPU global memory, to the moment when the warp could be available to be scheduled and needs the execution of the first ELF instruction that requires the use of the data, is not smaller than the distance between the two ELF instructions - the ELF instruction that requires the transfer of the data from the GPU global memory and the first ELF instruction that requires the use of the data - times the number of resident warps in the streaming multiprocessor - this because at the best case 2 warps can be scheduled at each warp scheduler clock cycle, but each warp ELF instruction can not be executed in less than 2 function units clock cycles.

Independently of a) the type of fatbin file - with or without divergences, with or without loops, etc. etc. - and b) the distances between the ELF instructions that require the transfer of data from the GPU global memory and the ELF instructions that require their use, greater the number

of resident warps in a streaming multiprocessor, greater the minimum number of function units clock cycles that has to pass before a warp can be scheduled again and so greater the number of function units clock cycle that has to pass before the warp is available to be scheduled and requires the execution of one of the ELF instructions that have to use the data and therefore greater the probability, for the warp schedulers in a streaming multiprocessor, to be able to hide the latencies of the GPU memories. At the same time, greater the number of resident warps in a streaming multiprocessor, greater the probability, for the warp schedulers in a streaming multiprocessor, to be able to hide the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps - remember that the overhead time for the management of the warps is not growing linearly and also whether it is the most important factor for the determination of the real ELF instruction configuration streaming multiprocessor average performance per clock cycle for some triplets ( ELF instructions configuration , dependence distance , number of resident warps in a streaming multiprocessor ), its influence is null beyond a given number of warps, number of warps that however dependent on the couple ( ELF instructions configuration , dependence distances ), see 7.6.2.

What it could be greater too, at the increase of the number of resident warps in a streaming multiprocessor - but this is a disadvantage for the warp scheduling cycling policy - it is the probability that the bandwidths of the GPU memories will slow down the execution of the B part of the fatbin file. This happens because, greater the number of resident warps in a streaming multiprocessor, greater it could be the number of data transfers required from the moment when a warp is scheduled and needs the execution of an ELF instruction that requires the transfer of a data from one of the GPU memories to the moment when the warp could be available to be scheduled and needs the execution of the first ELF instruction that requires the use of the data, this because all the warps are moved forward all together with the mechanisms explained in 9.4.1 and 9.4.2 and so the warp schedulers have to cycle on more warps, thing that increases the probability that a greater number of transfers could be necessary compared to the case when the B part of the same fatbin file is executed with a smaller number of resident warps in each streaming multiprocessor.

## 9.7 Summary

In this chapter we have explained that also whether, thanks to the results in 7.6.2, it is reasonable to assume specific things about the warp scheduling, because it is impossible to get the implementation details about the warp scheduling policy executed by the warp schedulers in the streaming multiprocessors of the GF100 architecture, then the generalization or not a) of the results in 7.6.2 about the warp scheduling and b) of what it is reasonable to assume about the warp scheduling, depends on what the reader believes . The main points to remember from this chapter are:

- With what we know from 7.6.2, it is reasonable to assume that, at least for the very simple fatbin files used in that chapter, if the fatbin file is executed with a launch configuration such that the couple ( dependence distance = fatbin file , number of resident warps in a streaming multiprocessor = launch configuration ) does not have any balancing problem - this means that the couple is in the set $G_s$, 9.2.2 - the warp schedulers, in a streaming multiprocessor, are moving forward all the resident warps in the streaming multiprocessor together, this at least in the case when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, as it effectly happens for the executions

of the for loops of the B parts of the fatbin files used in 7;

- It is impossible to get the implementation details of the warp scheduling policy used by the warp schedulers in the streaming multiprocessors, so we can not know for sure whether, when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, the 2 warp schedulers in a streaming multiprocessor are moving forward all the resident warps in the streaming multiprocessor together or whether, because there is no way to study the advancement of the warps, inside the for loops, during the executions of the B parts of the fatbin files, without to make useless all the output results, this is an illusion given to us from the fact that we analyze, for each resident warp in a streaming multiprocessor, only the 2 moments corresponding to the moments when the warp is going to enter in and has just left, the for loops, of the B parts, of fatbin files;

- The reader has to choose whether, at least in the case when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, to believe that the warp schedulers are executing the warp scheduling cycling policy - 9.4 - or not. The reader's choice has different consequences about the possibility or not to generalize the results we got about the warp scheduling in 7.6.2 and so it is one of the five factors that determines whether the fatbin file $F_{f_o}$ is in the subset $SS_{A_1}$ and therefore partially determine - partially because it is only one of the five factors - whether the fatbin files in $S^2_{F_f}$ are eligible for the execution of the analysis $A_1$ - this is due to the way we generate the fatbin files in the set $S^2_{F_f}$;

- The warp scheduling cycling policy is, in our opinion, the warp scheduling policy executed by the warp schedulers in the streaming multiprocessors, this considering a) all the types of advantages and b) the only potential disadvantage that the execution of the warp scheduling cycling policy gives compared to the numerous and heavy disadvantages that instead any other policy would generate whether executed by the warp schedulers in the streaming multiprocessors of an architecture like the GF100 architecture;

The reader's choice determines that type of warp scheduling policy, whether cycling or not, we need to consider as one of the five factors necessary to determine whether the fatbin file $F_{f_o}$ is in the subset $SS_{A_1}$ and therefore partially determine - partially because it is only one of the five factors - whether the fatbin files in the set $S^2_{F_f}$ are eligible for the execution of the analysis $A_1$ - this is due to the way we generate the fatbin files in the set $S^2_{F_f}$. The other three factors that is necessary to consider to determine whether the fatbin file $F_{f_o}$ is in the subset $SS_{A_1}$ are 1) the presence of branches in the B part of the fatbin file $F_{f_o}$, 2) which, the read believes, are the eviction policies for the l2 cache and the l1 caches, 3) the possibility to know a priori, before the execution of the fatbin file $F_{f_o}$, which are the positions, in the arrays, the vectors and the structures, in the GPU global memory, of the data/results that will be read/written, by each GPU thread used for the execution of the B part of the fatbin file $F_{f_o}$, during the execution of the B part of the fatbin file $F_{f_o}$ and 4) the presence of ELF instructions of synchronization in the B part of the fatbin file $F_{f_o}$. In the next chapter we talk of these four factors.

# Chapter 10

# Taxonomy for Fatbin Files

## 10.1 Introduction

In the previous chapter we have described the warp scheduling policy - the cycling policy - that is probably executed by the warp schedulers in the streaming multiprocessors of the GF100 architecture. Also whether we give many supporting reasons on why the warp scheduling cycling policy is, in our opinion, the warp scheduling policy executed by the warp schedulers in the streaming multiprocessors of the GF100 architecture, there is no way we can be 100% sure of this, because it is impossible to get the implementations details of the warp scheduling policy. Because we can not be 100% sure that the warp scheduling cycling policy is the warp scheduling policy executed by the warps in the streaming multiprocessors then we need to give to the reader the choice to believe whether, at least when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, the warp scheduling cycling policy is the warp scheduling policy executed or not by the warp schedulers in the streaming multiprocessors of the GF100 architecture.

The reader's choice about the warp scheduling policy is one of the four factors that determines the place of a fatbin file in the taxonomy that we introduce in this chapter. The taxonomy is simple. In the taxonomy a fatbin file can only be in the subset $SS_{A_1}$ or in its complement. The fact that a fatbin file is or not in the subset $SS_{A_1}$ depends on 1) which warp scheduling policy the reader believes is executed by the warp schedulers in the streaming multiprocessors when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, 2) the presence of branches in the B part of the fatbin file, 3) which, the read believes, are the eviction policies used for the l2 cache and the l1 caches, 4) the possibility to know *a priori*, before the execution of the fatbin file, which are the positions, in the arrays, the vectors and the structures, in the GPU global memory, of the data/results that will be read/written by each GPU thread used to execute the B part of the fatbin file, during the execution of the B part of the fatbin file and 5) the presence of ELF instructions of synchronization in the B part of the fatbin file.

The reader's choices about what to believe a) it is the warp scheduling policy executed by the warp schedulers in the streaming multiprocessors when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories and b) which are the eviction policies used for the l2 cache and l1 caches, determines whether the fatbin file $F_{f_o}$ and so the fatbin files in the set $S^2_{F_f}$ are in the complement of the subset $SS_{A_1}$ or whether instead they could be in the subset $SS_{A_1}$ - to be sure the fatbin file $F_{f_o}$ and the fatbin files in the set $S^2_{F_f}$ are

in the subset $SS_{A_1}$ it is necessary to check also the others three factors - factor 2), factor 4) and factor 5) - that do not depend on the reader's choices but instead depend on the B parts of the fatbin files and could depend on the values of the input data read by the GPU threads used for the executions of the B parts of the fatbin files.

## 10.2 Warp Scheduling Policy

In the previous chapter we subdivided all the possible warp scheduling policies that could be executed by the warp schedulers in the streaming multiprocessors in two set: the set composed by only the warp scheduling cycling policy and the set composed by all the other possible warp scheduling policies. The fact that a fatbin file is or not in the subset $SS_{A_1}$ partially depends on which warp scheduling policy the reader believes is executed by the warp schedulers in the streaming multiprocessors when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories:

- If the reader believes that the warp scheduling policy executed by the warp schedulers in the streaming multiprocessors, when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, is not the warp scheduling cycling policy, then the fatbin file is not in the subset $SS_{A_1}$ and therefore the fatbin files in the set $S^2_{F_f}$ are not eligible for the execution of the analysis $A_!$ but only for the analysis $A_2$ described in 11;
- If the reader believes that the warp scheduling policy executed by the warp schedulers in the streaming multiprocessor, when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, is the warp scheduling cycling policy, then the fatbin file could be in the subset $SS_{A_1}$, this depends on the other four factors, that we describe in the next the four sections.

## 10.3 Branches

Let us suppose that a fatbin file has some branches in its B part. If a fatbin file has some branches then, usually, during the execution of the fatbin file, which subparts, of the B part of the fatbin file, each GPU thread of a warp is going to execute, depend on the values of the input data that the GPU thread is going to read during the execution of the B part of the fatbin file.

Because a) the subparts, of the B part of the fatbin file, that each GPU thread of a warp is going to execute, are determined by the values of the input data that the GPU thread is going to read during the execution of the B part of the fatbin file and b) it is usually impossible to know, *a priori*, which will be the values of the input data, that at the next execution of the B part of the fatbin file, a GPU thread is going to read, it follows that it is usually impossible to know *a priori* which subparts, of the B part of the fatbin file, a GPU thread will execute during the next execution of the fatbin file and it could be impossible too to know *a priori* the order of execution of such subparts.

When the GPU threads of a warp execute different subparts of the B part of a fatbin file we are in presence of a phenomenon known as divergence. Let us suppose the 32 GPU threads of a warp require the execution of 4 different ELF instructions - this means that they are executing 4 different subparts of the B part of a fatbin file. Because the GPU threads of the warp require the execution

of 4 different ELF instructions, the 32 GPU threads of the warp can be subdivided in 4 subsets, one subset for each one of the 4 different ELF instructions that is necessary to execute for the warp. During the next 4 times the warp will be scheduled, the first time only 1, of the remaining 4 of the 4 different ELF instructions, that is necessary to execute for the warp, is executed, the second time only 1, of the remaining 3 of the 4 different ELF instructions, that is necessary to execute for the warp, is executed, the third time only 1, of the remaining 2 of the 4 different ELF instructions, that is necessary to execute for the warp, is executed and the fourth time the last, of the 4 different ELF instructions, that is necessary to execute for the warp, is executed. At this point the 32 GPU threads of the warp could or not to point the same ELF instruction in the B part of the fatbin file:

- If the 32 GPU threads of the warp point different ELF instructions, in the B part of the fatbin file, then the number of different ELF instructions could be smaller than 4, equal to 4 or greater than 4, but in any case, also if the degree of divergence could be diminished - smaller than 4 - equal - equal to 4 - or increased - greater than 4 - compared to the previous case, the previous procedure will be repeated when a) the warp will be available to be scheduled again - all the data necessary for the execution of the next different ELF instructions pointed by the 32 GPU threads of the warp are available and can be read - and b) one of the two warp schedulers in the streaming multiprocessor will decide to schedule the warp.
- If the 32 GPU threads of the warp point the same ELF instruction, in the B part of the fatbin file, then all the 32 GPU threads of the warp require now the execution of the same ELF instruction and so the previous divergence phenomenon, as the slow down to it associated, will be both absent the next time the warp will be scheduled

Because it is usually impossible to know *a priori* which subparts, of the B part of a fatbin file, the GPU threads are going to execute, during the next execution of the B part of the fabin file, it is usually impossible to determine, to which ELF instructions in the B part of a fatbin file, the GPU threads of a warp, and more in general the GPU threads of all the warps, will point at different moments during the execution of the B part of the fatbin file and so a) to force load balancing among the streaming multiprocessors, b) to determine the slow downs generated by the divergence phenomenons, c) to determine the quantities of bytes that could be necessary to read/write from/to the GPU global memory and to transfer among different GPU memory levels. For these reasons:

- If a) a fatbin file has some branches in its B part and b) it is possible that the GPU threads of a warp can follow a different path during the execution of the B part of the fatbin file - different from the path that could be followed by the other GPU threads of the warp and different from the path that could be followed by the GPU threads of other warps - then the fatbin file is not in the subset $SS_{A_1}$;
- If a) a fatbin file has some branches in its B part and b) it is impossible that the GPU threads of a warp can follow a different path during the execution of the B part of the fatbin file - different from the path that could be followed by the other GPU threads of the warp and different from the path that could be followed by the GPU threads of other warps - then the fatbin file could be in the subset $SS_{A_1}$;
- If a fatbin file has not branches in its B part then the fatbin file could be in the subset $SS_{A_1}$.

There could be however some cases where, a fatbin file, with some branches in its B part, is not in the subset $SS_{A_1}$, only because for some combinations ( input , launch configuration ), that

could be used to execute the B part of the fatbin file, some of the GPU threads, used to execute the B part of the fatbin file, follow different paths - this also whether maybe there could be instead other combinations ( input , launch configuration ), that could be used to execute the B part of the fatbin file, for which all the GPU threads, used to execute the B part of the fatbin file, follow the same path.

In these cases, before to start to analyze the couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), to get more couples eligible for the execution of the analysis $A_1$, we can substitute each couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) considered not eligible for the execution of the analysis $A_1$ because for some inputs, some of the GPU threads, used to execute the B part of the fatbin file, follow different paths, with the triplets ( subset of inputs of the set of inputs used for the original fatbin file $F_{f_i}$ , fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) forcing all the GPU threads, used to execute the B part of the fatbin file, to follow the same path - each one of the sets of possible inputs of interest forces all the GPU threads, used to execute the B part of the fatbin file, to follow the same path but this path is different from all the other paths that all the other sets of possible inputs of interest force all the GPU threads, used to execute the B part of the fatbin file, to follow;

That a) the reader believes that the warp scheduling policy executed by the warp schedulers in the streaming multiprocessors, when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, is the warp scheduling cycling policy and b) the B part of a fatbin file is without branches or c) the B part of a fatbin file has some branches but all the GPU threads of all the warps used to execute the B part of the fatbin file follow the same path during the execution of the B part of the fatbin file, it is however not enough to guarantee that the fatbin file is in the subset $SS_{A_1}$. Three other factors, together a), b) and c), effectively determine whether a fatbin file is or not in the subset $SS_{A_1}$. The first of these three other factors, the eviction policies used for the l2 cache and the l1 caches, is discussed in the next section.

## 10.4 Eviction Policies Used for the L2 Cache and the L1 Caches

The eviction policies used for the l2 cache and the l1 caches of the GF100 architecture are not disclosed. We believe that, knowing what we now know about the GF100 architecture, it is reasonable to assume that the eviction policies, when one of the caches is full and it is necessary to substitute some cache lines, will substitute the cache lines last recently used (LRU).

In our time frame we had no time to plan some experiments to validate whether the eviction policies used for the l2 cache and the l1 caches of the GF100 architecture are LRU policies but in any case we think that is probably impossible to build some experiments able to validate or discover the eviction policies used for the l2 cache and the l1 caches of the GF100 architecture - this is due to the impossibility to choose/force the warp scheduling and understand before of the execution of the B part of a fatbin file which it will be or after the execution of the B part of the fatbin file which it has been.

Because it was not possible to build some experiments to validate or discover the eviction policies used for the l2 cache and the l1 caches of the GF100 architecture, then, also in this case, like in the case for the warp scheduling policy, the reader has to choose whether to believe or not that the

eviction policies used for the l2 cache and the l1 caches of the GF100 architecture are LRU policies:

- If the reader believes that the eviction policies used for the l2 cache and the l1 caches of the GF100 architecture are not LRU policies then the fatbin file is not in the subset $SS_{A_1}$;
- If the reader believes that the eviction policies used for the l2 cache and the l1 caches of the GF100 architecture are LRU policies then if the reader also believes that the warp scheduling policy executed by the warp schedulers in the streaming multiprocessors is the warp scheduling cycling policy, if the fatbin file has no branches in its B part or if the fatbin file has some branches in its B part but that all the GPU threads of all the warps used to execute the B part of the fatbin file follow the same path during the execution of the B part of the fatbin file, then the fatbin file could instead be in the subset $SS_{A_1}$ - this depends on the last the two factors, a) the possibility to know *a priori*, before the execution of the fatbin file, which are the positions, in the arrays, the vectors and the structures, in the GPU global memory, of the data/results that will be read/written by each GPU thread used to execute the B part of the fatbin file, during the execution of the B part of the fatbin file and b) the presence of ELF instructions of synchronization in the B part of the fatbin file.

## 10.5 Reading and Writing - Which and Where

The possibility to know *a priori*, before the execution of the fatbin file, which are the positions, in the arrays, the vectors and the structures, in the GPU global memory, of the data/results that will be read/written by each GPU thread used to execute the B part of the fatbin file, during the execution of the B part of the fatbin file, is fundamental - together a) at the fact that the reader believes that the warp scheduling policy executed by the warps in the streaming multiprocessors, when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, is the warp scheduling cycling policy, b) at the fact that the fatbin file has no branches in its B part or at the fact that the fatbin file has some branches in its B part but that all the GPU threads of all the warps used to execute the B part of the fatbin file follow the same path during the execution of the B part of the fatbin file and c) at the fact that the reader believes that the eviction policies used for the l2 cache and the l1 caches of the GF100 architecture are LRU policies - to determine, the quantities of bytes that it is necessary to transfer, from/to off-chip to/from on-chip, during the execution of the B part of a fatbin file - we will explain why in 12.2 - and therefore:

- If it is not possible to know *a priori*, before the execution of the fatbin file, which are the positions, in the arrays, the vectors and the structures, in the GPU global memory, of the data/results that will be read/written by each GPU thread used to execute the B part of the fatbin file, during the execution of the B part of the fatbin file, then the fatbin file is not in the subset $SS_{A_1}$;
- If it is possible to know *a priori*, before the execution of the fatbin file, which are the positions, in the arrays, the vectors and the structures, in the GPU global memory, of the data/results that will be read/written by each GPU thread used to execute the B part of the fatbin file, during the execution of the B part of the fatbin file, if the reader believes that the warp scheduling policy executed by the warps in the streaming multiprocessors, when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the

GPU memories, is the warp scheduling cycling policy, if the fatbin file has no branches in its B part or if the fatbin file has some branches in its B part but that all the GPU threads of all the warps used to execute the B part of the fatbin file follow the same path during the execution of the B part of the fatbin file, if the reader believes that the eviction policies used for the l2 cache and the l1 caches of the GF100 architecture are LRU policies, then the fatbin file could be in the subset $SS_{A_1}$ - this depends on the last factor, the presence of ELF instructions of synchronization in the B part of the fatbin file.

## 10.6 ELF Instructions of Synchronization

In 9.4.5 we explain why, also if the warp schedulers in the streaming multiprocessors are executing the warp scheduling cycling policy when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, it is possible to get some maximum starting time differences, for some of the couples ( dependence distance , number of warps resident in a streaming multiprocessor ) of the set $G_s$ - 9.2.2 - that would seem impossible to get supposing the warp scheduling policy executed by the warps in the streaming multiprocessors, when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, is the warp scheduling cycling policy.

One of the possibilities that explains why this is however possible it is described in the case $C_3$ - 9.4.5. After an ELF instruction of synchronization, the warps, that become synchronized, could be released at different moments, forcing, in this way, for a period of time, the pointer in each streaming multiprocessor, and so the 2 warp schedulers in each streaming multiprocessor, to cycle only on a limited also whether increasing number of warps. Because there is no way to verify if this is true or not then we can not exclude this is the case and therefore:

- If the fatbin file has some ELF instructions of synchronization in its B part then the fatbin file is not in the subset $SS_{A_1}$. If a fatbin file has some ELF instructions of synchronization in its B part and it is possible that all the warps resident in a streaming multiprocessor are not made available again at the same moment to the pointer that assigns them to the 2 warp schedulers in the streaming multiprocessor then for the fact that the pointer, and so the 2 warp schedulers in the streaming multiprocessor, are cycling on a reduced number of warps compared to all those resident in the streaming multiprocessor, some slowdowns, due to the scheduling waiting times, the dependence waiting times and the overhead for the management of the warp, could be generated, and each one of these slowdowns could generate some other slowdowns a) of different nature - and so due to the warp scheduling and/or to the bandwidths and the latencies of the GPU memories - b) of the same nature - and so due to the scheduling waiting times and/or to the dependence waiting times and/or to the overhead time for the management of the warps - or c) due to a mix of the previous ones in a) and b). Because we can not exclude this avalanche effect and it is impossible 1) to quantify the number and type of slowdowns that the avalanche effect would generate, 2) the moment when each one of the slowdowns would be generated and c) to quantify the slowdown times of the slowdowns, then we need to take care that the analysis $A_1$ is executed only on fatbin files without ELF instructions of synchronization in their B parts.

- If the fatbin file instead has not ELF instructions of synchronization in its B part then, if the reader also believes that the warp scheduling policy executed by the warp schedulers in the

streaming multiprocessors, when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, is the warp scheduling cycling policy, if the fatbin file has no branches in its B part or if the fatbin file has some branches in its B part but that all the GPU threads of all the warps used to execute the B part of the fatbin file follow the same path during the execution of the B part of the fatbin file, if the reader believes that the eviction policies used for the l2 cache and the l1 caches of the GF100 architecture are LRU policies and if it is possible to know *a priori*, before the execution of the fatbin file, which are the positions, in the arrays, the vectors and the structures, in the GPU global memory, of the data/results that will be read/written by each GPU thread used to execute the B part of the fatbin file, during the execution of the B part of the fatbin file, then the fatbin file is in the subset $SS_{A_1}$.

## 10.7 Fatbin Files Generated for the Optimizations

If a fatbin file is in the subset $SS_{A_1}$ then the fatbin file is eligible for the execution of the analysis $A_1$. Given the procedure used to generate the fatbin files in the set $S^2_{F_f}$ from the fatbin files in the set $S^1_{F_f}$ and the procedure to generate the fatbin files in the set $S^1_{F_f}$ from the fatbin file $F_{f_o}$, if the fatbin file $F_{f_o}$ is in the subset $SS_{A_1}$ then automatically the fatbin files in the set $S^2_{F_f}$ are in the subset $SS_{A_1}$. For the fatbin file $F_{f_o}$ we need instead to distinguish the two following cases:

- If the fatbin file $F_{f_o}$ a) is equal to the original fatbin file $F_{f_i}$ or b) is generated from the original fatbin file $F_{f_i}$ using the procedure $P_1$ or c) is generated from the original fatbin file $F_{f_i}$ using the procedure $P_2$ and the set of transformations and changes $TAC_1$ - 8.2.3 - then, if the original fatbin file $F_{f_i}$ is in the subset $SS_{A_1}$ then the fatbin file $F_{f_o}$ is in the subset $SS_{A_1}$, while if the original fatbin file $F_{f_i}$ is not in the subset $SS_{A_1}$ then the fatbin file $F_{f_o}$ is not in the subset $SS_{A_1}$;

- If the fatbin file $F_{f_o}$ is instead generated from the original fatbin file $F_{f_i}$ using the procedure $P_2$ and the set of transformations and changes $TAC_2$ - 8.2.3 - then it does not matter whether the fatbin file $F_{f_i}$ is or not in the subset $SS_{A_1}$, the fatbin file $F_{f_o}$ could be or not in the subset $SS_{A_1}$ independently of the fact that the original fatbin file $F_{f_i}$ is or not in the subset $SS_{A_1}$.

## 10.8 Summary

In this chapter we have introduced a taxonomy for fatbin files. The taxonomy is very simple, every fatbin file can only be in the subset $SS_{A_1}$ or in its complement. If a fatbin file is in the subset $SS_{A_1}$ then it is eligible for the execution of the analysis $A_1$ described in 12. The main points to remember from this chapter are the following:

- The fact that a fatbin file is or not in the subset $SS_{A_1}$ depends on 1) which warp scheduling policy the reader believes is executed by the warps in the streaming multiprocessor when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, 2) the presence of branches in the B part of the fatbin file, 3) which, the read believes, are the eviction policies used for the l2 cache and the l1 caches, 4) the possibility to know *a priori*, before the execution of the fatbin file, which are the positions, in the arrays, the vectors and the structures, in the GPU global memory, of the data/results

that will be read/written by each GPU thread used to execute the B part of the fatbin file, during the execution of the B part of the fatbin file and 5) the presence of ELF instructions of synchronization in the B part of the fatbin file;

- A fatbin file is in the subset $SS_{A_1}$, and so eligible for the execution of the analysis $A_1$, described in 12, if all the following conditions are true: a) the reader believes that the warp scheduling policy executed by the warps in the streaming multiprocessors, when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, is the warp scheduling cycling policy, b) the fatbin file has no branches in its B part or the fatbin file has some branches in its B part but that all the GPU threads of all the warps used to execute the B part of the fatbin file follow the same path during the execution of the B part of the fatbin file, c) the reader believes that the eviction policies used for the l2 cache and the l1 caches of the GF100 architecture are last recently used policies, d) it is possible to know *a priori*, before the execution of the fatbin file, which are the positions, in the arrays, the vectors and the structures, in the GPU global memory, of the data/results that will be read/written by each GPU thread used to execute the B part of the fatbin file, during the execution of the B part of the fatbin file and e) the fatbin file has no ELF instructions of synchronization in its B part;

- There could be some cases where, a fatbin file, with some branches in its B part, is not in the subset $SS_{A_1}$, only because for some combinations ( input , launch configuration ), that could be used to execute the B part of the fatbin file, some of the GPU threads, used to execute the B part of the fatbin file, follow different paths - this also whether maybe there could be instead other combinations ( input , launch configuration ), that could be used to execute the B part of the fatbin file, for which all the GPU threads, used to execute the B part of the fatbin file, follow the same path.

  In these cases, before to start to analyze the couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), to get more couples eligible for the execution of the analysis $A_1$, we can substitute each couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) considered not eligible for the execution of the analysis $A_1$ because for some inputs, some of the GPU threads, used to execute the B part of the fatbin file, follow different paths, with the triplets ( set of possible inputs of interest , fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) forcing all the GPU threads, used to execute the B part of the fatbin file, to follow the same path - each one of the sets of possible inputs of interest forces all the GPU threads, used to execute the B part of the fatbin file, to follow the same path but this path is different from all the other paths that all the other sets of possible inputs of interest force all the GPU threads, used to execute the B part of the fatbin file, to follow;

- If the fatbin file $F_{f_o}$ a) is equal to the original fatbin file $F_{f_i}$ or b) is generated from the original fatbin file $F_{f_i}$ using the procedure $P_1$ or c) is generated from the original fatbin file $F_{f_i}$ using the procedure $P_2$ and the set of transformations and changes $TAC_1$ - 8.2.3 - then, if the original fatbin file $F_{f_i}$ is in the subset $SS_{A_1}$ then the fatbin file $F_{f_o}$ is in the subset $SS_{A_1}$, while if the original fatbin file $F_{f_i}$ is not in the subset $SS_{A_1}$ then the fatbin file $F_{f_o}$ is not in the subset $SS_{A_1}$.

  If the fatbin file $F_{f_o}$ is instead generated from the original fatbin file $F_{f_i}$ using the procedure

$P_2$ and the set of transformations and changes $TAC_2$ - 8.2.3 - then it does not matter whether the fatbin file $F_{f_i}$ is or not in the subset $SS_{A_1}$, the fatbin file $F_{f_o}$ could be or not in the subset $SS_{A_1}$ independently of the fact that the original fatbin file $F_{f_i}$ is or not in the subset $SS_{A_1}$.

- Given the procedure used to generate the fatbin files in the set $S^2_{F_f}$ from the fatbin files in the set $S^1_{F_f}$ and the procedure to generate the fatbin files in the set $S^1_{F_f}$ from the fatbin file $F_{f_o}$, if the fatbin file $F_{f_o}$ is in the subset $SS_{A_1}$ then automatically the fatbin files in the set $S^2_{F_f}$ are in the subset $SS_{A_1}$.

The analysis or the analyses that are executed on a fatbin file depend on a) the fact that the fatbin file is or not in the subset $SS_{A_1}$ and b) the reader's goals. In the next chapter, considering a) whether the fatbin files in the set $S^2_{F_f}$ are or not in the subset $SS_{A_1}$ and b) the reader's goals, we describe the analysis or the analyses that are executed on each fatbin file in the set $S^2_{F_f}$.

# Chapter 11

# Analysis/Analyses Selection

## 11.1 Introduction

In the previous chapter we have explained how to determine whether the fatbin file $F_{f_o}$ is in the subset $SS_{A_1}$ or not and so whether the fatbin files in the set $S^2_{F_f}$ are in the subset $SS_{A_1}$ or not and therefore whether the fatbin files in the set $S^2_{F_f}$ are eligible for the execution of the analysis $A_1$ or not.

The analysis/analyses that can be executed on the fatbin files in the set $S^2_{F_f}$ is/are determined by a) the fact that the the fatbin file $F_{f_o}$ is in the subset $SS_{A_1}$ or not and b) the reader's goals. In this chapter we therefore distinguish the possible different cases given by the combinations of a) and b) and describe the analysis $A_2$ - the analysis $A_1$ will be instead described in 12.

## 11.2 Analysis/Analyses Selection

The selection of the analysis/analyses to execute on the fatbin files in the set $S^2_{F_f}$ depends on two factors: a) whether the fatbin file $F_{f_o}$ is in the subset $SS_{A_1}$ or not and b) by the reader's goals. We can distinguish the following cases:

- If the fatbin file $F_{f_o}$ is in the subset $SS_{A_1}$ then we can execute the analysis $A_1$, described in 12, on each one of the fatbin files in the set $S^2_{F_f}$. At the end of this phase only one of the two following mutually exclusive subcases is possible:

  - At least one couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) satisfies all the requirements of the analysis $A_1$. In this case we give *a priori* ELF code shortest execution time guarantees for the execution, of the ELF codes, of the B parts, of the fatbin files, of the couples, satisfying all the requirements of the analysis $A_1$.

    Thanks to the *a priori* ELF code shortest execution time guarantees, we have the guarantee that the executions of the ELF codes, of the B parts, of the fatbin files, of the couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), satisfying all the requirements of the analysis $A_1$, will be never slowed down by 1) the bandwidths of the GPU memories, 2) the latencies of the GPU memories, 3) the scheduling waiting times, 4) the dependence waiting times and 5) the overhead time for the management of the warps.

Furthermore, for the couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), satisfying all the requirements of the analysis $A_1$, we also get the *a priori* guarantees that a) the only thing that can slow down the executions of the ELF codes, of the B parts, of the fatbin files, of the couples, is the warp scheduling - and so sometimes it could be possible that less than 2 warps are scheduled at a warp scheduler clock cycle, but this not for causes due to 1), 2), 3), 4) and 5), but for the warp scheduling, that we can not choose/force or know, before and after too, the execution of the B part of a fatbin file - and b) that independently of the warp scheduling, the slowdowns, that the warp scheduling can generate, are not going to create some local streaming multiprocessor states or global GPU states such that slowdowns due to 1), 2), 3), 4) and 5) become possible.

Also if we get an *a priori* ELF code shortest execution time guarantee for some couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), some of the couples could have an average execution time, for the B part, of their fatbin file, that is smaller than the average execution time, for the B part, of the fatbin file, of others, because a) we can not choose/force the warp scheduling, b) the GPU allows to the warp schedulers to choose the warp schedulings to use for the executions of the B part of a fatbin file in $S^2_{F_f}$, of a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), from a subset of all the possible warp schedulings that could be used, c) the subset of warp schedulings from where the GPU allows to the warp schedulers to choose the warp schedulings for the executions of the B part of the fatbin file in $S^2_{F_f}$, of a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), could be determined not only on the type of fatbin file but also on the launch configuration, the dimensions of the variables, the arrays, the vectors and the structures, in the GPU global memory, necessary to execute the couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) - variables, arrays, vectors and structures that contain the input data and that will contain the output results - and the location of the data/results in the GPU global memory - location that we can not know or force, see 9.5.5 - and d) we have no way to determine the subset of warp schedulings from where the GPU allows to the warp schedulers to choose the warp scheduling for the executions of the B part, of the fatbin file in $S^2_{F_f}$, of a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) - note that this is the simplest possible case because the fatbin file is fixed, the launch configuration is fixed and that the dimensions of the variables, the arrays, the vectors and structures could be fixed because the launch configuration determines the number of GPU threads used to execute the B part of the fatbin file and this number will be always the same from execution to execution.

Because also if we get an *a priori* ELF code shortest execution time guarantee for some couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), some of the couples could have an average execution time, for the B part, of their fatbin file, smaller than the average execution time, for the B part, of the fatbin file, of others, considering the reader's goals and so the quantity of time that the reader is willing to dedicate to the analysis and the optimization of the original fatbin file $F_{f_i}$, the reader has two mutually exclusive choices:

* For the reader is enough to get the *a priori* ELF code shortest execution time guarantee for at least one couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$

of the fatbin file in $S^2_{F_f}$ ).

In this case, during the analysis $A_1$, the first couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), that satisfies all the requirements of the analysis $A_1$, is the couple that the reader can choose to use to solve the same problem that the reader would solve using the original fatbin file $F_{f_i}$. However, whether the reader would use the original fabin file $F_{f_i}$, the reader would probably get an average execution time, for the B part, of the original fatbin file $F_{f_i}$, greater than the average execution time, of the B part, of the fatbin file, of the first couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) found to satisfy all the requirements of the analysis $A_1$.

If the reader would be willing to spend more time for the analysis, supposing there are more couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) satisfying all the requirements of the analysis $A_1$, then more couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), satisfying all the requirements of the analysis $A_1$, will be found, but because there is no way to calculate *a priori* the execution times of the B part of the fatbin files of each one of these couples - this because a) there is no way to know *a priori* the warp scheduling that will be used by the warp schedulers in the streaming multiprocessors from execution to execution of each single couple, b) there is now way to choose/force the warp scheduling and c) usually the warp scheduling, also for the same couple, changes from execution to execution - then there is no way, *a priori*, to differentiate for execution time, of the B part, of the fatbin files, the couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), satisfying all the requirements of the analysis $A_1$, and so the reader can simply choose, during the analysis $A_1$, the first couple satisfying all the requirements and terminate the analysis $A_1$;

  * For the reader is not enough to get the *a priori* ELF code shortest execution time guarantee for at least a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ). In this case a second analysis, the analysis $A_2$ - that we explain in the following paragraphs below - is executed on all the couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), satisfying all the requirements of the analysis $A_1$, and the best couple - best could mean different things considering the possibly different reader's goals - is the couple that is used by the reader to solve the same problem that the reader could solve using the couple ( original fatbin file $F_{f_i}$ , a possible launch configuration for the original fatbin file $F_{f_i}$ that is not necessarily in the set $S_{lc}$ of the original fatbin file $F_{f_i}$ ).

- No couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) satisfies all the requirements of the analysis $A_1$. In this case we can not give an *a priori* ELF code shortest execution time guarantee for the execution of the ELF code, of the B part, of any of the fatbin files in the set $S^2_{F_f}$. If there is not any couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) satisfying all the requirements of the analysis $A_1$ then we execute the analysis $A_2$ on each one of the fatbin files in the set $S^2_{F_f}$.

  When the analysis $A_2$ is executed on a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), the fatbin file in $S^2_{F_f}$ of the couple has to be executed a

given minimum number of times T using the specific launch configuration of the couple. T could be different for different couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) and depends on how much fast we can get an accurate distribution of the execution times, of the B part, of the fatbin file, of the couple, this, of course, when the fatbin file of the couple is executed using the launch configuration of the couple - we will not repeat this anymore in this chapter, but when we talk of execution time, of the B part, of the fatbin file, of a couple, we always imply that the execution time, of the B part, of the fatbin file, of the couple, is obtained executing the fatbin file of the couple using the launch configuration of the couple - but because a) we can not *a priori* know the distribution, b) the distribution depends on factors that we can not control - for example the warp scheduling - and c) the behaviors of the factors can change, also for the same couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), from execution to execution, then T can be determined only at run time. The analysis $A_2$ allows us 1) to study and determine the minimum and maximum execution times, of the B part, of the fatbin file, of each couple, 2) to study and determine the distribution of the execution times, of the B part, of the fatbin file, of each couple, between the minimum and the maximum execution times, of the B part, of the fatbin file, of each couple, 3) to calculate the average execution time, the median execution time, the variance of the execution time and other statistical parameters on the execution times, of the B part, of the fatbin file, of each couple and 4) to determine, among all the couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), the best couple - for example the couple $Couple_1$ with an average execution time, of the B part, of its fatbin file, that is the minimum among all the average execution times, of the B parts, of the fatbin files, of all the couples or, if we instead want something in high probability, for example the couple $Couple_2$ that, among all the couples, is the couple having the higher probability $P$, greater than a threshold probability $P_t$, that the B part, of its fatbin file, will be executed in a time smaller than that average time necessary to execute the B part, of the fatbin file, of the couple $Couple_1$.

- The fatbin file $F_{f_o}$ is not in the subset $SS_{A_1}$. If this is true then all the fatbin files in the set $S^2_{F_f}$ are not in the subset $SS_{A_1}$. If this is the case then the fatbin files in the set $S^2_{F_f}$ are not eligible for the execution of the analysis $A_1$ but we always can, in any case, to execute the analysis $A_2$ on the fatbin files in the set $S^2_{F_f}$ and this is, in fact, the only thing that we can do, supposing a) we do not want to modify the B part of the original fatbin file $F_{f_i}$, to move it, if possible, in the subset $SS_{A_1}$ and therefore generate from the original fatbin file $F_{f_i}$ so modified a new fatbin file $F_{f_o}$ - fatbin file $F_{f_o}$ that this time will be in the subset $SS_{A_1}$ thanks to the modifications in the B part of the original fatbin file $F_{f_i}$, but this could be impossible sometimes, it depends whether some modifications exist able to transform the original fatbin file $F_{f_i}$ in a fatbin file that is in the subset $SS_{A_1}$ and which procedure, whether $C_1$ or $C_2$, and/or which set of transformations and changes, whether $TAC_1$ or $TAC_2$, is/are used to generate the new fatbin file $F_{f_o}$ from the original fatbin file $F_{f_i}$ so modified - or b) we do not want to modify the B part of the fatbin file $F_{f_o}$ to move - if possible - the fatbin file $F_{f_o}$ in the subset $SS_{A_1}$.

  If however a) or b) is/are possible and applied, then we can generate another time the set of fatbin files $S^1_{F_f}$ and from the set of fatbin files $S^1_{F_f}$ the set of fatbin files $S^2_{F_f}$, fatbin files in

the set $S^2_{F_f}$ that this time will be in the subset $SS_{A_1}$ and on which therefore we can apply the analysis $A_1$.

Independently of the analysis/analyses executed on the couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), at the end of the analysis/analyses, the analysis $A_2$ is executed on the couples ( original fatbin file $F_{f_i}$ , a possible launch configuration for the original fatbin file $F_{f_i}$ that is not necessarily in the set $S_{lc}$ of the original fatbin file $F_{f_i}$ ) - the launch configurations are the launch configurations that the reader would use without knowing the procedure used to generate the sets of launch configurations $S_{lc}$s of the fatbin files in the set $S^2_{F_f}$, procedure therefore that could be used also to generate the set of launch configurations $S_{lc}$ for the original fatbin file $F_{f_i}$.

The execution times, of the B part, of the fatbin file, of the best couple ( original fatbin file $F_{f_i}$ , a possible launch configuration for the original fatbin file $F_{f_i}$ that is not necessarily in the set $S_{lc}$ of the original fatbin file $F_{f_i}$ ), got using the analysis $A_2$, are compared to the execution times, of the B part, of the fatbin file, of the best couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) - in the case only the analysis $A_1$ is used on the fatbin files in the set $S^2_{F_f}$ the couple could not be the best but one of the bests - to quantify the improvement obtained, for the execution time, of the B part, of the original fatbin file $F_{f_i}$, by the whole optimization process.

## 11.3 Summary

In this chapter we have described the different cases we can get about the selection of the analysis/analyses to execute on the fatbin files in the set $S^2_{F_f}$. The different cases depends on a) the fact that the fatbin file $F_{f_o}$ is in the subset $SS_{A_1}$ or not and b) the reader's goals. The main points to remember from this chapter are the following:

- If the fatbin file $F_{f_o}$ is in the subset $SS_{A_1}$ then we can execute the analysis $A_1$ on all the couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ):
  - If there is at least a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) satisfying all the requirements of the analysis $A_1$ then we can give an *a priori* ELF code shortest execution time for the execution, of the ELF code, of the B part, of the fatbin file, of at least a couple. After this, considering the reader's goals, the analysis $A_2$ could be executed or not, on the couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), satisfying all the requirements of the analysis $A_1$;
  - If instead there is not couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) satisfying all the requirements of the analysis $A_1$ then we need to execute on each couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) the analysis $A_2$.
- If the fatbin file $F_{f_o}$ is not in the subset $SS_{A_1}$ then a) we can execute on the fatbin files of the set $S^2_{F_f}$ the analysis $A_2$ or b) we can modify, the original fatbin file $F_{f_i}$, to move it, if possible, in the subset $SS_{A_1}$, generate from the original fatbin file $F_{f_i}$ the fatbin file $F_{f_o}$, taking care, of what we do, if to do this, we use the procedure $C_2$ with the set of transformations and changes $TAC_2$ - this to be sure that the fatbin file $F_{f_o}$ too is in the subset $SS_{A_1}$ - and therefore repeat

the procedures described in 8 to generate the new set of couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) on which to execute the analysis/analyses or c) we can directly modify the fatbin file $F_{f_o}$, to move it, if possible, in the subset $SS_{A_1}$, and therefore repeat the procedures described in 8 to generate the new set of couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) on which to execute the analysis/analyses;

- When the analysis $A_2$ is executed on a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), the fatbin file in $S^2_{F_f}$ of the couple has to be executed a given minimum number of times T using the specific launch configuration of the couple. T could be different for different couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) and depends on how much fast we can get an accurate distribution of the execution times, of the B part, of the fatbin file, of the couple, but because a) we can not *a priori* know the distribution, b) the distribution depends on factors that we can not control - for example the warp scheduling - and c) the behaviors of the factors can change, also for the same couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), from execution to execution, then T can be determined only at run time.

While in this chapter we have described the analysis $A_2$ we do not have described the analysis $A_1$. In the next chapter we therefore describe the analysis $A_1$ and we explain why it is possible, for each couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), satisfying all the requirements of the analysis $A_1$, to give an *a priori* ELF code shortest execution time guarantee, for the execution, of the ELF code, of the B part, of the fatbin file, of each one of such couples.

# Chapter 12

# Guaranteeing *A Priori* ELF Code Shortest Execution Times

## 12.1 Introduction

In the previous chapter, given the possible combinations generated by a) the fact that the fatbin files in the set $S^2_{F_f}$ are in the subset $SS_{A_1}$ - 10 - or not and b) the reader's goals, we have explained the procedure to determine the analysis/analyses that can be executed on the fatbin files in the set $S^2_{F_f}$ and we have describe one of the analyses, the analysis $A_2$.

In this chapter we describe the analysis $A_1$. When the analysis $A_1$ is executed on a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), if the couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) satisfies all the requirements a) of the subanalysis on the bandwidths and the latencies of the GPU memories and b) of the subanalysis on the number of resident warps in each streaming multiprocessor, then we get an *a priori* ELF code shortest execution time guarantee for the execution of the ELF code, of the B part, of the fatbin file, of the couple, when the fatbin file is executed using the launch configuration of the couple, and so the guarantee that the execution of the ELF code, of the B part, of the fatbin file, of the couple, when the fatbin file is executed using the launch configuration of the couple, can be slowed down only by the warp scheduling - that we can not choose/force - and not by the bandwidths and the latencies of the GPU memories, the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps.

## 12.2 Bandwidths and Latencies of the GPU Memories

The subanalysis on the bandwidths and the latencies of the GPU memories is necessary to get the guarantee that, independently of the warp scheduling, when some data are read from GPU memories different from the hardware registers present in the streaming multiprocessors, the data will be in the hardware registers when necessary, therefore giving us the guarantee that it will be impossible for the warps in a streaming multiprocessor not being available to be scheduled, when necessary, at cause of the fact that some data, that the warps need for the execution of their next warp ELF instruction, are not yet in the hardware registers, that will be used as operands, because the data are still moving among different GPU memories.

### 12.2.1 Reading and Writing - Positions and Locations

If we execute the analysis $A_1$ on a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) then this means that the fatbin file of the couple is in the subset $SS_{A_1}$. Because the fatbin file of the couple is in the subset $SS_{A_1}$ then we know from where and to where, in the variables, the arrays, the vectors and the structures in the GPU global memory, each GPU thread, used to execute the fatbin file of the couple, reads and writes the data and the results, when the fatbin file is executed using the launch configuration of the couple.

We can not know, choose or force the location of the variables, the arrays, the vectors and the structures are in the GPU global memory - 9.5.5 - but we know that they are aligned to frontiers of 128 bytes, the dimension in bytes of a line of l1 cache - [50, p. 163] and [55, p. 32].

When the GPU threads of a warp read some data that a) are in the GPU global memory b) but are between more than two consecutive frontiers, then more l1 cache lines will be transfered. For example, if, to execute a warp ELF instruction, a) the 32 GPU threads of the warps read data that are between the positions 4 and 28, 45 and 51 and 68 and 90 in an array of data, b) each data is at 4 bytes and c) the data are in the GPU global memory, then three l1 cache lines have to be transfered from the GPU global memory to the l1 cache.

When the 32 GPU threads of a warp read or write results, if we know a) whether the data or the results are or not in the l2 cache or in the l1 cache and b) the positions, of the data and the results, to read and to write, in the variables, the arrays, the vectors and the structures in the GPU global memory, then it is possible to determine the number of l1 cache lines that will be transfered, for the read or the write, between the GPU global memory and the l1 cache, and so from off-chip to on-chip and/or from on-chip to off-chip, when the data or the results have to be read or written from/to the GPU global memory.

Let us suppose, for example, a warp ELF instruction requires to read some data and 1) that 8, of the 32 GPU threads, of the warp, read, from the GPU global memory, a) the same data at 16 bits, b) that the data is only in the GPU global memory and c) the data is at the position 0 of an array A, 2) that 15, of the 32 GPU threads, of the warp, read, from the l1 cache, other data all present in the l1 cache and 3) that 9, of the 32 GPU threads, of the warp a) read 9 different data, b) that all the 9 data are all between the positions 64 and 127 of the array A and c) that all the 9 data are only in GPU global memory. In these conditions, when such warp ELF instruction is executed, 2 l1 cache lines are transfered by the GPU global memory to the l1 cache and so from off-chip to on-chip - this happens because one l1 cache line has to be transfered for the data, in position 0, of the array A, that 8, of the 32 GPU threads, of the warp, read and another l1 cache line has to be transfered for the 9 different data, that 9, of the 32 GPU threads, of the warp, read - these 9 data are in positions between the 64 and the 127 in the array A and so between the bytes in position 128 and 255 from the beginning of the array A, array A that we know being aligned to frontiers of 128 bytes in the GPU global memory.

### 12.2.2 Difficulties in the Determination of the Cache Lines to Transfer

If there would be only one GPU thread used for the execution of the fatbin file of a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) then, knowing 1) from where and to where, in the variables, the arrays, the vectors and the structures in the GPU global memory, the GPU thread, reads and writes, the data and the results, during the execution of the fatbin file, 2) the eviction policies used for the l2 cache and the l1 caches and 3) that is not necessary that a

cache line that is in a l1 cache is also in the l2 cache and that therefore there are some updating mechanisms, when a warp tries to write some results to the GPU global memory and a l1 cache line is only partially overwritten, that could imply a new transfer of the l1 cache line, from the GPU global memory to the l1 cache, it would be easy to determine a) the number of l1 cache lines that is necessary to transfer from off-chip to on-chip and from on-chip to off-chip and b) when these cache lines are transfered between different GPU memories, for the execution of a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ).

However, 1) many GPU threads are used for the execution of the fatbin file of a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) and 2) also for launch configurations a) forcing the gigathread scheduler to evenly distribute the GPU thread blocks to the streaming multiprocessors and b) without warp scheduling balancing problems at local level - this means b.1) that there is an even number of warps in each streaming multiprocessor, b.2) that the number of warps is the same for all the streaming multiprocessors and b.3) that all the couples ( dependence distance in the set $S_{dd}$ of the fatbin file in $S^2_{F_f}$ , number of resident warps in a streaming multiprocessor ) are in the good set $G_s$, 9.2.2 - the maximum starting time differences, for the couples in the set $G_s$, in the table $T_c$, built, for the global level, considering all the streaming multiprocessors at the same time, are of the order of the millions of function unit clock cycles.

What said in 1) and 2) make it impossible accurately to determine to which ELF instructions of the ELF code, of the B part, of a fatbin file, the many GPU threads, used for the execution of the B part of a fatbin file, are pointing, during the execution of the B part of the fatbin file, and so to determine a) the number of l1 cache lines that is necessary to transfer from off-chip to on-chip and from on-chip to off-chip and b) when these l1 cache lines are transfered between different GPU memories, during the execution of a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ).

### 12.2.3 Supposing the GF100 Architecture Without the L2 Cache

Let us instead suppose the GF100 architecture is without the l2 cache. If we suppose the GF100 architecture is without the l2 cache then the data and the results, whether not in the hardware registers in the streaming multiprocessors, can only be in the GPU global memory, in the l1 caches, in the shared memories, in the constant memory or in the texture memories - the constant memory and the texture memories are not considered in the following discussion because the constant memory can only be managed by the CPU side before to execute the B part of a fatbin file and the texture memories are not of interest.

Without l2 cache, the data can only be moved 1) from the GPU global memory to the l1 caches or to the shared memories or 2) from the l1 caches or from the shared memories to the GPU global memory. Because each streaming multiprocessor has its l1 cache and its shared memory then, when we need to determine a) the number of l1 cache lines that is necessary to transfer from off-chip to on-chip and from on-chip to off-chip and b) when these l1 cache lines are transfered between the different GPU memories, for the execution of a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), instead to consider the maximum of the maximum starting time differences of the couples ( dependence distance in the set $S_{dd}$ of the fatbin file in $S^2_{F_f}$ , number of resident warps in a streaming multiprocessor ) in the table $T_c$, built for the global level, we can consider the maximum of the maximum starting time differences of the couples ( dependence distance in the set $S_{dd}$ of the fatbin file in $S^2_{F_f}$ , number of resident warps in a streaming multiprocessor ) in the table

$T_c$, built for the local level, considering only a single streaming multiprocessor, instead of all the streaming multiprocessors at the same time.

The quantity of data that is necessary to transfer, from off-chip to on-chip and from on-chip to off-chip, during the execution of a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), supposing the GF100 architecture is without l2 cache, is at least equal whether not greater to the quantity of data that is really necessary to transfer, from off-chip to on-chip and from on-chip to off-chip, during the execution of the couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) - this because the GF100 architecture has a l2 cache - and therefore is an upper bound on the quantity of data that is really necessary to transfer, from off-chip to on-chip and from on-chip to off-chip, during the execution of the couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ).

### 12.2.4 Maximum Distance in Number of Warp ELF Instructions

To calculate the previous upper bound, on the quantity of data that is necessary to transfer, from off-chip to on-chip and from on-chip to off-chip, during the execution of a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), we need to determine the number of l1 cache lines that is necessary to transfer, from off-chip to on-chip and from on-chip to off-chip, during the execution of a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ).

To try to determine the number of l1 cache lines that is necessary to transfer, from off-chip to on-chip and from on-chip to off-chip, during the execution of a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), we first calculate the maximum distance, in number of warp ELF instructions, that we can get, during the execution of the B part of the fatbin file of the couple, when the fatbin file of the couple is executed with the launch configuration of the couple, between the leading subset of resident warps in a streaming multiprocessor and the last subset of resident warps in the streaming multiprocessor.

If the couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) would not satisfy all the requirements of the analysis $A_1$ then a) there is no way to calculate the maximum distance, in number of warp ELF instructions, between the leading subset of resident warps in a streaming multiprocessor and the last subset of resident warps in the streaming multiprocessor, that is possible during the execution of the B part of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple and b) the distances, in number of warp ELF instructions, among resident warps in a streaming multiprocessor, are not going to stay constant or almost constant but can change abruptly during the execution of the B part of the fatbin file of the couple, when the fatbin file of the couple is executed with the launch configuration of the couple.

If instead we suppose *a priori* the couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) satisfies all the requirements of the analysis $A_1$ and so the execution of the B part of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple, can be slowed down only by the warp scheduling and not by the bandwidths and the latencies of the GPU memories, the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps then a) it is possible to calculate the maximum distance, in number of warp ELF instructions, between the leading subset of resident warps in a streaming multiprocessor and the last subset of resident warps in the streaming multiprocessor, that

is possible during the execution of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple and b) the distances, in number of warp ELF instructions, among resident warps in a streaming multiprocessor, are going to stay practically constant, during the execution of the fatbin file of the couple, when the fatbin file of the couple is executed with the launch configuration of the couple.

To calculate the maximum distance, in number of warp ELF instructions, between the leading subset of resident warps in a streaming multiprocessor and the last subset of resident warps in the streaming multiprocessor, we determine the maximum of the maximum starting time differences $M_{mstd}$ of the couples ( dependence distance in the set $S_{dd}$ of the fatbin file in $S^2_{F_f}$ , number of resident warps in a streaming multiprocessor ) in the table $T_c$, built for the local level, and remember the following things: 1) in each moment not more than 2 warps can be scheduled at each warp scheduler clock cycle in a streaming multiprocessor, 2) because a) each one of the 4 groups of function units in a streaming multiprocessor have no more than 16 function units and b) each warp is always composed by 32 GPU threads, then the execution of a warp ELF instruction requires at least 2 function unit clock cycles, 3) the launch configuration of the couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) determines the number of resident warps per streaming multiprocessor - W - 4) the minimum scheduling waiting time we discovered is of 4 function units clock cycles, 5) because 4 - the minimum scheduling waiting time we discovered - is greater than 2 - the minimum number of function unit clock cycles that are necessary to execute a warp ELF instruction - then in $M_{mstd}$ function unit clock cycles, the 2 warp schedulers in a streaming multiprocessor can not schedule the same warp more times than $\lfloor \frac{M_{mstd}}{4} \rfloor$ and so in $M_{mstd}$ function unit clock cycles not more than $\lfloor \frac{M_{mstd}}{4} \rfloor$ warp ELF instructions can be executed for the same warp - this because a) the execution of a warp ELF instruction require at least 2 function unit clock cycles and b) 1 warp scheduler clock cycle is equivalent to 2 function unit clock cycles. It would therefore seem, that at the moment when the last or the last 2 resident warps in a streaming multiprocessor are scheduled for the first time for the execution, of the B part, of the fatbin file, of a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), the maximum distance, in number of warp ELF instructions, between the leading subset of resident warps in a streaming multiprocessor and the last subset of resident warps in a streaming multiprocessor, can be not more than $MD_{nwei} = \lfloor \frac{M_{mstd}}{4} \rfloor$ - this because in $M_{mstd}$ function unit clock cycles the minimum number of warp ELF instructions that can be scheduled for a warp is 0 while the maximum its $\lfloor \frac{M_{mstd}}{4} \rfloor$.

### 12.2.5 Introduction of ELF Instructions of Synchronization

We do not know how much time is necessary to the gigathread scheduler to distribute the GPU thread blocks to the streaming multiprocessors and so possibly more GPU thread blocks to a streaming multiprocessor. For this reason, when we determine the $M_{mstd}$ for a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), to be sure that the $M_{mstd}$, determined for the couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), is really an upper bound on the starting time difference that we can get for the execution of the B part of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple, we need to modify the fatbin files in the set $S^2_{F_f}$.

The table $T_c$, built in 7, for the local level, is built using the results got for the fatbin files used in that chapter. To be sure that, using the data of the table $T_c$, built for the local level, the $M_{mstd}$ we

determine for a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), is really an upper bound on the starting time difference that we can get for the execution of the B part of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple, we modify the fatbin files in the set $S^2_{F_f}$, introducing at the beginning of their B parts, the same three ELF instructions, we used to synchronize the GPU threads used for the executions of the fatbin files used in 7 and therefore 1) a write ELF instruction, that writes, the data in one of the ELF registers of the fatbin file, to a GPU global memory address common to all the GPU threads used for the execution of the B part of the fatbin file, 2) a membar.gl ELF instruction and 3) a read ELF instruction that reads the data from the GPU global memory address common to all the GPU threads used for the execution of the B part of the fatbin file and load it back to the ELF register used for the writing in 1). Doing this we are sure that the $M_{mstd}$ that we calculate for each couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) is really an upper bound on the starting time difference that we can get for the execution of the B part of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple.

Knowing that the $M_{mstd}$, that we calculate for a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), is really an upper bound on the starting time difference that we can get for the execution of the B part of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple, we can be sure that, at the warp scheduler clock cycle $WSCC$ - when, for the first time, after that all the resident warps in a streaming multiprocessor have been scheduled for the execution of the three warp ELF instructions used to synchronize all the resident warps in all the streaming multiprocessors, all the resident warps in the streaming multiprocessor have been scheduled at least another time - the distance, in number of warp ELF instructions, between the leading subset of resident warps in the streaming multiprocessor and the last subset of resident warps in the streaming multiprocessor, is really not more than $MD_{nwei} = \lfloor \frac{M_{mstd}}{4} \rfloor$. To give some examples, considering the values in the table $T_c$, built for the local level:

- Because the maximum of the maximum starting time difference of all possible couples ( dependence distance , number of resident warps in a streaming multiprocessor = 8 ), in the set $G_s$, is 56, then, the $MD_{nwei}$, of every fatbin file in the set $S^2_{F_f}$, when the fatbin file in the set $S^2_{F_f}$ is executed with a launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ that forces the gigathread scheduler to assign to each streaming multiprocessor a number of warps equal to 8, can not be greater than 14 - it could be smaller, this depends on the dependence distances in the set $S_{dd}$ of the fatbin file in $S^2_{F_f}$;

- Because the maximum of the maximum starting time differences of all possible couples ( dependence distance , number of resident warps in a streaming multiprocessor = 32 ), in the set $G_s$, is 296, then, the $MD_{nwei}$, of every fatbin file in the set $S^2_{F_f}$, when the fatbin file in the set $S^2_{F_f}$ is executed with a launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ that forces the gigathread scheduler to assign to each streaming multiprocessor a number of warps equal to 32, can not be greater than 74 - it could be smaller, this depends on the dependence distances in the set $S_{dd}$ of the fatbin file in $S^2_{F_f}$.

### 12.2.6 Constancy, of the Distances, in Number of Warp ELF Instructions

After the three ELF instructions, used to synchronize the warps, at the beginning of the B part, of the fatbin file, of a couple, it could be that the warp schedulers in a streaming multiprocessor are going to cycle, for a period of time, on a number of warps that is smaller than the number of resident warps in the streaming multiprocessor - case $C_3$ in 9.4.5.

During the execution of the B part of the fatbin file of a couple, we however reach a warp scheduler clock cycle $WSCC$ when, for the first time, after that all the resident warps in a streaming multiprocessor have been scheduled for the execution of the three warp ELF instructions used to synchronize all the resident warps in all the streaming multiprocessors, all the resident warps in the streaming multiprocessor have been scheduled at least another time.

What we want prove in this subsection is that if a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) satisfies all the requirements of the analysis $A_1$ and so the execution of the B part of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple, can only be slowed down by the warp scheduling, then each distance - calculated at the warp scheduler clock cycle $WSCC$ - in number of warp ELF instructions, between each couple of resident warps in a streaming multiprocessor, can oscillate not more than plus two minus two for almost the whole execution of the remaining B part of the fatbin file of the couple.

If it is true that the execution of the B part of the fatbin file of a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) can only be slowed down by the warp scheduling, then, after the warp scheduler clock cycle $WSCC$, for almost the whole execution of the remaining B part of the fatbin file of a couple, it impossible that the 2 warp schedulers in a streaming multiprocessors are pointing to 2 warps that are more distant than 3 warps, in the order, established by the GF100 architecture, for the resident warps in the streaming multiprocessor, order on which the pointer in the streaming multiprocessor, and so the 2 warp schedulers in the streaming multiprocessor, cycle - 9.4.

The warp ELF instructions that require the greatest number of function units clock cycles to be executed are the warp ELF instructions that require for their execution the use of the group of 4 special function units. For the execution of one of such warp ELF instructions, because the number of GPU threads in a warp is always 32, 8 function unit clock cycles - equivalent to 4 warp scheduler clock cycles - are necessary. Furthermore, at any moment, of the execution of the B part of the fatbin file of a couple, each one of the 2 warp schedulers in a streaming multiprocessor is pointing to one of the warps $w_{sm}$ resident in the streaming multiprocessor.

Let us now suppose: 1) that one of the 2 warp schedulers is pointing the warp $w_x$ and the other warp scheduler is pointing the warp $w_y$, 2) that the next warp, in the order, that will be considered by the pointer in the streaming multiprocessor, is the warp $w_{(y+1)\%w_{sm}}$, 3) that both the warps $w_x$ and $w_y$ need the execution of a warp ELF instruction that requires the use of the group of 4 special function units and 4) that the warp $w_x$ is scheduled at the warp scheduler clock cycle $t$ making therefore impossible to schedule the warp $w_y$ at warp scheduler clock cycles $t$, $t+1$, $t+2$ and $t+3$.

Also whether 1 of the 2 warp schedulers can not schedule the warp $w_y$ at the warp scheduler clock cycles $t$, $t+1$, $t+2$ and $t+3$, at the warp scheduler clock cycles $t+1$, $t+2$ and $t+3$, the other warp scheduler could however continue to schedule one or more warps - these warps would be, in the order, the warp/warps $W_{(y+1)\%w_{sm}}$, $W_{(y+2)\%w_{sm}}$ and $W_{(y+3)\%w_{sm}}$. At the warp scheduler clock cycle $t+3$, the distance, in number of warps, between the 2 warps, pointed by the 2 warp

schedulers in the streaming multiprocessor, can not be greater than 3 - if equal to 3 then, at the warp scheduler clock cycles $t+1$, $t+2$ and $t+3$, 1 of the 2 warp schedulers has scheduled the warps $W_{(y+1)\%w_{sm}}$, $W_{(y+2)\%w_{sm}}$ and $W_{(y+3)\%w_{sm}}$.

At $t+4$ the warp scheduler that is pointing the warp $w_y$ schedules the warp $w_y$ - this for sure because the warp scheduler that is pointing the warp $w_y$ has the precedence on the other warp scheduler, 9.4.2 - and get from the pointer in the streaming multiprocessor the next warp in the order. The group of 4 special function units will be available again only at the warp scheduler clock cycle $t+8$. Because at the warp scheduler clock cycles $t+4$, $t+5$, $t+4$ and $t+7$, not more than 2 warps can be scheduled at each warp scheduler clock cycle, then, if there are at least 11 = 3 + 8 resident warps in the streaming multiprocessor, we get the guarantee that each time a warp $w$ is scheduled, the other warp scheduler is pointing to a warp that, in the order, is the warp $W_{(y-3)\%w_{sm}}$, $W_{(y-2)\%w_{sm}}$, $W_{(y-1)\%w_{sm}}$, $W_{(y+1)\%w_{sm}}$, $W_{(y+2)\%w_{sm}}$ or $W_{(y+3)\%w_{sm}}$.

If instead the number of resident warps in a streaming multiprocessor would be less than 11 then, it is possible, if the warp $w_y$ would require the execution of a series of warp ELF instructions each one using the group of 4 special function units, the generation of a compound effect, where the group of 4 special function units is not available, when the warp $w_y$ is available to be scheduled, not because busy to execute the warp ELF instruction of another warp, but because busy to execute a previous warp ELF instruction for the warp $w_y$. If this would be the case then the distances, in number of warp ELF instructions, between the 2 warps of some of the couples of resident warps in the streaming multiprocessor, could grow and grow, without any limit, during the execution of the B part of the fatbin file of a couple, when the fatbin file is executed using the launch configuration of the couple, but we want to be sure that this does not happen because also whether our next proofs allow to the distances, in number of warp ELF instructions, between the 2 warps of all the possible couples of resident warps in a streaming multiprocessor, to change during the execution of the B part of the fatbin file of a couple, when the fatbin file is executed using the launch configuration of the couple, the value of each distance, in number of warp ELF instructions, between the 2 warps of each couple of resident warps in a streaming multiprocessor, has to oscillate, during the execution of the B part of the fatbin file of a couple, when the fatbin file is executed using the launch configuration of the couple, around the initial value of the distance, calculated at the warp scheduler clock cycle $WSCC$, of a maximum quantity that has to be quantifiable.

At the warp scheduler clock cycle $WSCC$, only the two following cases are possible: 1) only 1 of the 2 warp schedulers in the streaming multiprocessor get a new warp from the pointer in the streaming multiprocessor or 2) both the warps schedulers in the streaming multiprocessor get a new warp from the pointer, in the streaming multiprocessor, that assigns the resident warps in the streaming multiprocessor to the 2 warp schedulers in the streaming multiprocessor. Independently of 1) and 2), after the assignment/s, the 2 warp schedulers in the streaming multiprocessor are pointing to 2 warps a) that have to be consecutive in the order and b) without any other warp, in the order, between them. Let us define the warp, of the 2, that comes first in the order, the first warp of the order.

Considering how the warp scheduling cycling policy cycles on the resident warps in the streaming multiprocessor - 9.4.1 and 9.4.2 - and that, supposing there are at least 11 resident warps in a streaming multiprocessor, each time a warp is scheduled by 1 of the 2 warp schedulers in a streaming multiprocessor the other warp scheduler is pointing to another warp that is distant not more than 3 in the order of the resident warps in the streaming multiprocessor, then, during almost the whole execution of the B part of the fatbin file of a couple, the initial values of the distances,

calculated at the warp scheduler clock cycle $WSCC$, between a resident warp $w$ in a streaming multiprocessor and the other resident warps in the streaming multiprocessor, can only change in one of the following three ways:

- The value of the distance, in number of warp ELF instructions, between the warp $w$ and another resident warp in the streaming multiprocessor, it's equal to the value of the distance, between the 2 warps, calculated at the warp scheduler clock cycle $WSCC$.

  In this case, the value of the distance, in number of warp ELF instructions, between the warp $w$ and the other resident warp in the streaming multiprocessor, can a) remain constant or b) decrease of one or c) decrease of two - this is only possible whether only 1 of the 2 warps, the warp $w$ and the other resident warp in the streaming multiprocessor, is one of the three last warps in the order - or d) increase of one or e) increase of two - this is only possible if only 1 of the 2 warps, the warp $w$ and the other resident warp in the streaming multiprocessor is one of the first three warps in the order;

- The value of the distance, in number of warp ELF instructions, between the warp $w$ and another resident warp in the streaming multiprocessor, it's equal to the value of the distance, between the 2 warps, calculated at the warp scheduler clock cycle $WSCC$ minus one or two - two is only possible if only 1 of the 2 warps, the warp $w$ and the other resident warp in the streaming multiprocessor, is one of the last three warps in the order.

  In this case, the value of the distance, in number of warp ELF instructions, between the warp $w$ and the other resident warp in the streaming multiprocessor, can a) remain constant or b) increase of one or c) increase of two - this is only possible if only 1 of the 2 warps, the warp $w$ and the other resident warp in the streaming multiprocessor, is one of the last three warps in the order;

- The value of the distance, in number of warp ELF instructions, between the warp $w$ and another resident warp in the streaming multiprocessor it's equal to the distance between the 2 warps calculated at the warp scheduler clock cycle $WSCC$ plus one or two - two is only possible if only 1 of the 2 warps, the warp $w$ or the other resident warp in the streaming multiprocessor, is one of the first three warps in the order.

  In this case, the value of the distance, in number of warp ELF instructions, between the warp $w$ and the other resident warp in the streaming multiprocessor, can a) remain constant or b) decrease of one or c) decrease of two - this is only possible if only 1 of the 2 warps, the warp $w$ and the other resident warp in the streaming multiprocessor, is one of the first three warps in the order.

For the previous three cases, after the warp scheduler clock cycle $WSCC$, the value of the distance, in number of warp ELF instructions, between 2 generic resident warps in a streaming multiprocessor, can only increase at maximum of two and decrease at maximum of two, compared to the value of the distance, calculated at the warp scheduler clock cycle $WSCC$, between the same 2 generic resident warps in the streaming multiprocessor, and therefore the value of the distance, in number of warp ELF instructions, between the 2 warps of each possible couple of resident warps in a streaming multiprocessor, is going to stay practically constant - plus two minus two - for almost the whole execution of the B part of the fatbin file of a couple - if the couple satisfies all the requirements of the analysis $A_1$.

What said is true for almost the whole execution of the B part of the fatbin file of a couple satisfying all the requirements of the analysis $A_1$ because near the end of the execution of the B part of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple, some warps will finish first of others to execute the B part of the fatbin file of the couple, leaving progressively a smaller and smaller number of resident warps in the streaming multiprocessor - at this point in time some slowdowns due to the bandwidths and the latencies of the GPU memories, the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps could be possible but they can not last more of the quantity of time necessary to execute a number of warp ELF instructions that can not be greater than $MD_{nwei}$ warp ELF instructions times the number of resident warps in a streaming multiprocessor and so usually a quantity of time that it is infinitesimal compared to the execution time, of the B part, of the fatbin file, of the couple, when the fatbin file is executed using the launch configuration of the couple.

Because the value of the distance, in number of warp ELF instructions, between the 2 warps of each possible couple of resident warps in a streaming multiprocessor can only oscillate, after the warp scheduler clock cycle $WSCC$, at maximum, of plus 2 minus 2, for almost the whole execution of the B part of the fatbin file of a couple, then, this means that, after the warp scheduler clock cycle $WSCC$, for almost the whole execution of the B part of the fatbin file of a couple, the maximum distance, in number of warp ELF instructions, between the leading warp or the leading subset of warps in a streaming multiprocessor and the last warp or the last subset of warps in a streaming multiprocessor, is not equal or smaller than $\lfloor \frac{M_{mstd}}{4} \rfloor$ - 12.2.4 - but instead is equal or smaller than $MD_{nwei} = \lfloor \frac{M_{mstd}}{4} \rfloor + 2$.

Before to execute the analysis $A_1$ on a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), because we can scan the B part of the fatbin file of the couple, we can check in an automatic way all the ELF instructions in the B part of the fatbin file and get the number of warp scheduler clock cycles that are necessary to execute the most "expensive" warp ELF instruction $wei_{me}$ of the B part of the fatbin file - this is possible thanks to the results got in 7.6.2.

Substituting in the proof of this subsection the value 4 - used supposing that the most "expensive" warp ELF instruction, in the B part of the fatbin file of the couple to analyze, would be a warp ELF instruction requiring the use of the group of 4 special function units - with the number of warp scheduler clock cycles necessary to execute the warp ELF instruction $wei_{me}$ - we determine the minimum number of warps $min^1_{w_{sm}}$, that is necessary are resident in a streaming multiprocessor, to avoid the compound effect that would not allow to quantify the maximum distance, in number of warp ELF instructions, between the leading warp or the leading subset of warps in a streaming multiprocessor and the last warp or the last subset of warps in a streaming multiprocessor, during the execution of the B part of the fatbin file of a couple - satisfying all the requirements of the analysis $A_1$ - when the fatbin file is executed using the launch configuration of the couple.

If for a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) $min^1_{w_{sm}}$ is equal or greater than the number of resident warps in each streaming multiprocessor, that we obtain when the fatbin file of a couple is executed using the launch configuration of the couple, then we need to discard the couple because the distance, in number of warps ELF instructions, between the leading warp or the leading subset of resident warps in a streaming multiprocessor and the last warp or the last subset of resident warps in the streaming multiprocessor, could grow and grow making impossible any proof on the bandwidths and the latencies of the GPU memories.

### 12.2.7 Warp ELF Instructions Implying Off-Chip ↔ On-Chip Transfers

If a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) satisfies all the requirements of the analysis $A_1$ then the execution of the B part of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple, can only be slowed down by the warp scheduling, and in these conditions we know that, during almost the whole execution of the B part of the fatbin file of the couple, also whether we can not calculate, and therefore know, the values of the single distances, in number of warp ELF instructions, between the 2 warps of each possible couple of resident warps, in any of the streaming multiprocessors, the distance, in number of warp ELF instructions, between the leading warp or the leading subset of warps in a streaming multiprocessor and the last warp or the last subset of warps in the same streaming multiprocessor, is equal or smaller than $MD_{nwei} = \lfloor \frac{M_{mstd}}{4} \rfloor + 2$ - 12.2.6.

To try to determine the number of l1 cache lines that is necessary to transfer during the execution of the B part of the fatbin file of a couple, when the fatbin file is executed using the launch configuration of the couple, we generate the set $S_{upv}$ of the unrolled path versions of the B part of the fatbin file of the couple. The original fatbin file $F_{f_i}$ has a set of inputs $S_i$. Each one of the unrolled path versions of the B part of the fatbin file of the couple is the path that a) is determined by each single input of a subset $SS_i$ of inputs - more inputs could force all the GPU threads used to execute the B part of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple, to follow the same path, same because the fatbin file of the couple is in the subset $SS_{A_1}$ and so it is impossible that there are some GPU threads, among those used to execute the B part of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple, following different paths, 10.3 - and b) is unrolled, some loops could be present in the B part of the fatbin file of the couple.

For the generation of the set of the unrolled path versions of the B part of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple, we need to consider the following things:

- If it is possible to determine/know which inputs will be used for the original fatbin file $F_{f_i}$ then it is possible to determine each single path - supposing there are more possible paths in the B part of the fatbin file of the couple - that all the GPU threads, used to execute the B part of the fatbin file, will follow in the B part of the fatbin file of the couple, each time the fatbin file is executed using the launch configuration of the couple, for each one of the inputs of one of the subsets $SS_i$ of inputs - each single path is a path version of the B part of the fatbin file of the couple;

- If it is not possible to determine/know which inputs will be used for the original fatbin file $F_{f_i}$ then it is not possible to determine which single path/paths - supposing there are more possible paths in the B part of the fatbin file of the couple - all the GPU threads, used to execute the B part of the fatbin file, will not follow in the B part of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple, but in this case, we can simply generate all the possible paths that a GPU thread can follow to execute the B part of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple - each single path is a path version of the B part of the fatbin file of the couple;

- If it is possible to determine/know the number of times, that each one of the loops, in the B

part of the fatbin file of the couple, has/have to be executed, for an input or for a path version of the B part of the fatbin file of the couple, then we generate the unrolled path version of B part of the fatbin file of the couple for the input - some inputs could give the same unrolled path version of the B part of the fatbin file - or for the path version of the B part of the fatbin file of the couple;

- If it is not possible to determine/know the number of times, that one or more loops, in the B part of the fatbin file of the couple, has/have to be executed, for an input or for a path version of the B part of the fatbin file of the couple then, we a) can assume different upper bounds on the number of times that each one of the loops, in the B part of the fatbin file of the couple, has/have to be executed, when the fatbin file is executed using the launch configuration of the couple, for the input or for the path version of the B part of the fatbin file of the couple, and b) generate an unrolled path version of the B part of the fatbin file of the couple for each one of the possible combinations of times that each one of the loops in the B part of the fatbin file, could be executed, when the fatbin file is executed using the launch configuration of the couple, for the input or for the path version of the B part of the fatbin file of the couple, but if this is case then we need to prove that it is impossible that any of the other unrolled path versions of the B part of the fatbin file of the couple, that 1) we do not generate and that correspond to potential executions of the B part of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple, for the input or for the path version of the B part of the fatbin file of the couple, and 2) where one or more of the loops is/are executed a number of times greater than its/their upper bound/bounds, can give executions of the B part of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple, for the input or for the path version of the B part of the fatbin file of the couple, that could be slowed down by things different by the warp scheduling - in other words we need to prove that each one of all the other possible unrolled path versions of the B part of the fatbin file of the couple that we do not generate for the potential executions of the B part of the fatbin file of the couple, that could happen, when the fatbin file is executed using the launch configuration of the couple, for the input or for the path version of the B part of the fatbin file of the couple, satisfies all the requirements of the analysis $A_1$.

We therefore overlap a window of size $MD_{nwei}$ to each one of the unrolled path versions of the B part of the fatbin file of the couple and we consecutively align the lower side of the window to each one of the ELF instructions of each one of the unrolled path versions of the B part of the fatbin file of the couple - in other words we make slide the window on all the unrolled path versions of the B part of the fatbin file of the couple.

Next, for each couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), we want to analyze the warp ELF instructions $wei_{ls}$ - the warp ELF instructions that require the load and/or the store of data and/or results - of each one of the single unrolled path versions in the sets $S_{uvp}$ generated for the couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) - one set $S_{uvp}$ per couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ).

For the same warp ELF instruction $wei_{ls}$, of an unrolled path version of the B part of the fatbin file of a couple a) in a set $S_{upv}$ and b) generated by each one of the inputs of one of the subsets $SS_i$ of inputs - the union of the subsets $SS_i$ of inputs give the set $S_i$ of inputs of the original fatbin file

$F_{f_i}$ used to analyze a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) - we can distinguish the following two cases:

- The warp ELF instruction $wei_{ls}$, of the unrolled path version of the B part of the fatbin file of the couple, load/store data/results from/to off-chip to/from on-chip because a) the data to read are transfered from the GPU global memory to the l1 caches of the streaming multiprocessors or to the shared memories of the streaming multiprocessors, b) the results to store are transfered from the l1 caches of the streaming multiprocessors or the shared memories of the streaming multiprocessors to the GPU global memory or c) the results to store imply the transfer of data from the the GPU global memory to the l1 caches of the streaming multiprocessors, the updating of the data and the transfer of some or all the data back to the GPU global memory, for some or all the inputs, of the subset $SS_i$, of inputs generating the unrolled path version of the B part of the fatbin file of the couple - remember that a) we are supposing the GF100 architecture is without l2 cache and b) we do not consider the constant memory and the texture caches;

- The warp ELF instruction $wei_{ls}$, of the unrolled path version of the B part of the fatbin file of the couple, does not load/store data/results from/to off-chip to/from on-chip because 1) the data are loaded from a) the l1 caches of the streaming multiprocessors, b) the shared memories of the streaming multiprocessors or c) the hardware registers of the streaming multiprocessors or 2) the results are stored a) from the hardware registers in the streaming multiprocessors to the shared memories of the streaming multiprocessors, or b) in the hardware registers of the streaming multiprocessors, for some or all the inputs, of the subset $SS_i$, of inputs generating the unrolled path version of the B part of the fatbin file of the couple.

What we want to understand, given an unrolled path version of the B part of the fatbin file of the a couple, which of its warp ELF instructions $wei_{ls}$ are warp ELF instructions $wei_{it}$ or in other words are warp ELF instructions that imply a) the transfer of a quantity of bytes from off-chip to on-chip, or b) the transfer of a quantity of bytes from on-chip to off-chip, or c) the transfer of a quantity of bytes from off-chip to on-chip and the transfer of a quantity of bytes from on-chip to off-chip - see $Possibility_2$ of the $Example_3$ in the next subsection to understand when this can happen - and we want understand this for each one of the inputs, of the subset $SS_i$, of inputs generating the unrolled path version of the B part of the fatbin file of the couple.

We therefore substitute each couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) with the set of all the possible quadruplets ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ , unrolled path version of the B part of the fatbin file in $S^2_{F_f}$ , an input, of one of the subsets $SS_i$, of inputs generating the unrolled path version of the B part of the fatbin file in $S^2_{F_f}$ ) and instead of each couple we analyze each set of quadruplets. Each set of quadruplets has one or more quadruplets for each one of the inputs, of the set $S_i$ of inputs, of the original fatbin file $F_{f_i}$ - this because a single input could generate more unrolled path versions of the B part of the fatbin file of a couple. During the analysis of each set of quadruplets, for each quadruplet, we can distinguish the following two mutually exclusive cases:

- Case 1. It is possible to determine which warp ELF instructions $wei_{ls}$, of the quadruplet, are warp ELF instructions $wei_{it}$, independently of a) which ELF instructions, the resident warps in a streaming multiprocessor, point at the warp scheduler clock cycle $WSCC$ - the warps have however to point to some ELF instructions inside the window going from the

first ELF instruction $fei_x$, after the three ELF instructions, used at the beginning of the B part of the fatbin file of the couple, to synchronize all the resident warps in all the streaming multiprocessors, to the ELF instruction $ei_x = fei_x + MD_{nwei}$ - b) the order of the resident warps in the streaming multiprocessor and c) which of the 2 warp schedulers in the streaming multiprocessor will have the priority of scheduling at the end of the warp scheduler clock cycle $WSCC$.

- Case 2. It is not possible to determine which warp ELF instructions $wei_{ls}$, of the quadruplet, are warp ELF instructions $wei_{it}$. Considering how we try to identify the warp ELF instruction $wei_{it}$ - with the help of a window of size $MD_{nwei}$ that we make slide on the unrolled path version of the B part of the fatbin file of the quadruplet - it can happen that it is not possible to determine which warp ELF instructions $wei_{ls}$ are warp ELF instructions $wei_{it}$ because there could be some cases where the warp ELF instructions $wei_{it}$ are a function of the previous a), b) and c) described in case 1, and so the warp ELF instructions $wei_{it}$ could be different from execution to execution of the fatbin file of a quadruplet, when the fatbin file of the quadruplet is executed using the launch configuration of the quadruplet, for the unrolled version of the quadruplet, for the input of the quadruplet. If for a quadruplet we are in this case then we have the following three choices:

  - $Choice_1$) We consider the couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), originating the set of quadruplets of the quadruplet that we are analyzing, as one of the couples that does not satisfy all the requirements of the analysis $A_1$ and therefore we discard the couple;
  - $Choice_2$) We discard the quadruplet and we continue to analyze the other quadruplets, of the set of quadruplets originated by the couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) that we are analyzing, to check which of them satisfy all the requirements of the analysis $A_1$;
  - $Choice_3$) We a) modify the fatbin file of the quadruplet taking care that the same unrolled version of the B part of the fatbin file of the quadruplet is generated again for the input of the quadruplet and analyze a second time the quadruplet, or b) modify the fatbin file of the couple generating the set of quadruplets, generate another time the set of quadruplets and start to analyze again each quadruplet of the new set of quadruplets, or c) modify the original fatbin file $F_{f_i}$ and repeat the whole procedure of generation 1) of the fatbin file $F_{f_o}$, 2) of the set $S^1_{F_f}$ of fatbin files, 3) of the set $S^2_{F_f}$ of fatbin files, 4) of the sets of launch configurations for the fatbin files in the set $S^2_{F_f}$ - one set for each fatbin file - and 5) of the set of quadruplets for each new couple couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) - but if we modify the original fatbin file $F_{f_i}$ then, if for the generation of the fatbin file $F_{f_o}$ - 8.2.3 - we use the procedure $C_2$ and the set of transformations and changes $TAC_2$, we need to be careful to how we generate the fatbin file $F_{f_o}$ because we could nullify the benefits of the modifications introduced in the original fatbin file $F_{f_i}$ and so find us, during the execution of the analysis $A_1$, in the case 2 again, for many of the new quadruplets generated.

    If we decide to modify the original fatbin file $F_{f_i}$, we do this to move from case 2 to case 1 as many couples ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file

in $S^2_{F_f}$ ) or as many quadruplets as possible - if the reader knows the distribution of the inputs then it could be important to move from case 2 to case 1 only few quadruplets.

A simple way to try to accomplish this goals is to substitute one or more ELF instructions, that load/store data/results using the l1 caches of the streaming multiprocessors, with ELF instructions that load/store data/results from/to the shared memories of the streaming multiprocessors to/from the hardware registers of the streaming multiprocessor and with ELF instructions that load/store the data/results from/to the shared memories of the streaming multiprocessors to/from the GPU global memory.

Because the shared memory has to be managed by the programmer, the ELF instructions, to load/store data/results from/to the shared memories of the streaming multiprocessors to/from the GPU memory, a) always imply the transfer of data/results and b) are always explicit about the quantity of bytes that is necessary to load/store.

If therefore the B part of a fatbin file would only have a) ELF instructions that load/store data/results from/to the shared memories of the streaming multiprocessors to/from the hardware registers of the streaming multiprocessors and b) ELF instructions that load/store data/results from/to the shared memories of the streaming multiprocessors to/from the GPU global memory, then all the quadruplets generated by the fatbin file would have a probability equal to zero of being in case 2 and so for them we would be in case 1.

Reducing the number of ELF instructions that load/store data/results using the l1 caches of the streaming multiprocessors and at the same time increasing the number of ELF instructions a) that load/store data/results from/to the shared memories of the streaming multiprocessors to/from the hardware registers of the streaming multiprocessors and b) that transfer data/results from/to the shared memories of the streaming multiprocessors to/from the GPU global memory, we make the probability, that when we analyze a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), we are in case 1 instead of case 2, at least equal whether not greater than that when we consider the couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) without the modifications.

Before each execution of a fatbin file, we need always to set the dimensions of the l1 cache and of the shared memory for all the streaming multiprocessors - the dimensions have to be the same for all the streaming multiprocessors, 3.3. With a total of 64 KB per streaming multiprocessor, partitioned as 48 KB of l1 cache and 16 KB of shared memory or 16 KB of l1 cache and 48 KB of shared memory, before each execution of the fatbin file of a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), and the fact that each couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) is not executed with more than 32 resident warps per streaming multiprocessor - 8.3 - then, in average, each warp, resident in a streaming multiprocessor, has at least 2 of the 64 KB of memory - 2 KB correspond to a quantity of memory equivalent to that of 16 lines of l1 cache. With the shared memory set to 48 KB before the execution of a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), each warp - supposing it is working on data completely different from the data on which all the other resident warps in the streaming multiprocessor are working - will have, in average, a quantity of shared memory, for its personal use,

equivalent to at least 1.5 KB - 1.5 KB correspond to 12 lines of l1 cache.

If we decide to modify a) the fatbin file of a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), b) the fatbin file of a quadruplet ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ , unrolled path version of the B part of the fatbin file in $S^2_{F_f}$ , an input, of one of the subsets $SS_i$, of inputs generating the unrolled path version of the B part of the fatbin file in $S^2_{F_f}$ ) or c) the original fatbin file $F_{f_i}$, then, because we are in case 2, when we analyze the B part of the fatbin file, only one of the two following things can be true:

* The B part of the fatbin file already uses the shared memories. If this is true then, because we are in case 2, this means that the B part of the fatbin file has ELF instructions that load/store data/results using the l1 caches of the streaming multiprocessors - if this would not be the case then we would be in case 1 because if there would not be ELF instructions, in the B part of the fatbin file, that load/store data/results using the l1 caches of the streaming multiprocessors then there would be only a) ELF instructions that load/store data/results from/to the shared memories of the streaming multiprocessors to/from the hardware registers of the streaming multiprocessors and b) ELF instructions that load/store data/results from/to the shared memories of the streaming multiprocessors to/from the GPU global memory, and as we know the ELF instructions that load/store data/results from/to the shared memories of the streaming multiprocessors to/from the GPU global memory always imply the transfer of data/results from/to on-chip to/from off-chip and are always explicit about the quantity of bytes to load/store.
  We can therefore further reduce the number of ELF instructions that load/store data/results using the l1 caches of the streaming multiprocessors substituting them a) with ELF instructions that load/store data/results from/to the shared memories of the streaming multiprocessors to/from the hardware registers of the streaming multiprocessors transfer and b) with ELF instructions that load/store data/results from/to the shared memories of the streaming multiprocessors to/from the GPU global memory;
* The B part of the fatbin file does not use the shared memory. In this case too we can substitute one or more ELF instructions that load/store data/results using the l1 caches of the streaming multiprocessors a) with ELF instructions that load/store data/results from/to the shared memories of the streaming multiprocessors to/from the hardware registers of the streaming multiprocessors and b) with ELF instructions that load/store data/results from/to the shared memories of the streaming multiprocessors to/from the GPU global memory.

At the end of this phase a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) is discarded or we know which warp ELF instructions $wei_{ls}$, of each one of the quadruplets ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ , unrolled path version of the B part of the fatbin file in $S^2_{F_f}$ , an input, of one of the subsets $SS_i$, of inputs generating the unrolled path version of the B part of the fatbin file in $S^2_{F_f}$ ) of the set of quadruplets substituting the couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), are warp ELF instructions $wei_{it}$.

### 12.2.8 Slowdowns due to the Bandwidths and the Latencies

If we know which are the warp ELF instructions $wei_{it}$ of a quadruplet ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ , unrolled path version of the B part of the fatbin file in $S^2_{F_f}$ , an input, of one of the subsets $SS_i$, of inputs generating the unrolled path version of the B part of the fatbin file in $S^2_{F_f}$ ) then, because we know a) from/to which GPU memories the data/results are loaded/stored and b) the positions of the data/results in the variables, the arrays, the vectors and the structures in the GPU global memory, we can calculate an upper bound on the quantity of bytes that the GF100 architecture has to transfer from off-chip to on-chip and/or from on-chip to off/chip for the execution of each warp ELF instruction $wei_{it}$ - the case from off-chip to on-chip and from on-chip to off/chip can happen when, for example, a result, in a hardware register in a streaming multiprocessor, have to be written in the GPU global memory, but it is necessary to transfer from off-chip to on-chip a l1 cache line in the l1 cache of a streaming multiprocessor, update part of the l1 cache line and transfer back the l1 cache line from on-chip to off-chip.

Furthermore, for each ELF instruction, that store/load data/results from/to the shared memories in a streaming multiprocessor to/from the GPU global memory, the positions, of the data/results to load/store, have always to be consecutive in the shared memories and the GPU global memory.

Some examples about the quantity of bytes that the GF100 architecture has to transfer from off-chip to on-chip and/or from on-chip to off/chip for the execution of some warp ELF instructions $wei_{it}$ are the following:

- $Example_1$) To load 129 consecutive bytes from the GPU global memory to the l1 cache of a streaming multiprocessor, the GF100 architecture transfers from off-chip to on-chip 256 bytes, this independently of the alignment of the first byte to the frontiers of 128 bytes of the GPU global memory - this happens because a l1 cache line has 128 bytes. The first l1 cache line is transfered for the first 128 bytes while the second l1 cache line is transfered for the last byte, the byte 129;

- $Example_2$) To store 128 consecutive bytes, that are in a l1 cache line in a l1 cache of a streaming multiprocessor, to consecutive locations in the GPU global memory:

  - $Possibility_1$) The 128 bytes will be aligned to one of the frontiers of 128 bytes of the GPU global memory. In this case the GF100 architecture transfers, from the l1 cache of a streaming multiprocessor to the GPU global memory, 128 bytes, and therefore for this case the total quantity of bytes transfered is 128;

  - $Possibility$ : 2) The 128 bytes will not be aligned to one of the frontiers of 128 bytes of the GPU global memory. In this case the GF100 architecture has to a) transfer from the global memory to the l1 cache of the streaming multiprocessor 2 l1 cache lines, b) update parts of each one of the 2 l1 cache lines and c) transfer back the 2 l1 cache lines from the l1 cache of the streaming multiprocessor to the GPU global memory, and therefore for this case the total quantity of bytes transfered is $128 \cdot 4 = 512$.

- $Example_3$) To store 129 consecutive bytes from the shared memory of a streaming multiprocessor to the GPU global memory, supposing the position of the first of these 129 consecutive bytes is aligned to one of the frontiers of 128 bytes of the GPU global memory, the GF100 architecture transfers not more than $128 + 128 = 256$ bytes. For this case we have two possibilities:

- $Possibility_1$) 129 bytes are transfered directly from the shared memory to the GPU global memory and exactly 129 bytes are transfered from the GF100 architecture. In our opinion this does not happen. We say this considering a) how works the transfers between the off-chip GPU global memory and the on-chip l1 cache memories - the transfers are done at groups of 128 bytes for the l1 cache - and b) the fact that the 64 KB of memory of each streaming multiprocessor are configurable, from execution to execution, as 48 KB of l1 cache and 16 KB of shared memory or 16 KB of l1 cache and 48 KB shared memory;
- $Possibility_2$) 129 bytes are transfered directly from the shared memory to the GPU global memory but because the transfers for the l1 cache are done at groups of 128 bytes - the dimension in bytes of a l1 cache line - then the data paths used to transfer bytes at the worst case will be used to transfer the 129 bytes as whether the 129 bytes would be 256 bytes. In our opinion this is what happens considering the fact that the 64 KB of memory of each streaming multiprocessor are configurable, from execution to execution, as 48 KB of l1 cache and 16 KB of shared memory or 16 KB of l1 cache and 48 KB shared memory.

If a quadruplet has warp ELF instructions $wei_{it}$ that to write to the GPU global memory has to a) transfer bytes from the GPU global memory to the l1 caches of the streaming multiprocessors and b) transfer back some or all the bytes from the l1 caches of the streaming multiprocessors to the GPU global - see $Example_2$ above for one of these warp ELF instructions - then this is a problem for a proofs in this subsection. To avoid these problems, we repeat the procedure described in $Choice_3$ in the previous subsection, this to modify a) the fatbin file of the quadruplet, or b) the fatbin file of the couple originating the quadruplet, or c) the original fatbin file, to eliminate these warp ELF instructions $wei_{ls}$ - this can be done substituting them 1) with warp ELF instructions that transfers bytes from the GPU global memory to the the shared memories of the streaming multiprocessors and b) with warp ELF instructions that transfer bytes from the shared memories of the streaming multiprocessors to the GPU global memory. If the reader does not wish to do the modifications then other different proofs have to be used but these proofs will be discussed in another place at an another time.

For each warp ELF instruction $wei_{it}$, of each quadruplet ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ , unrolled path version of the B part of the fatbin file in $S^2_{F_f}$ , an input, of one of the subsets $SS_i$, of inputs generating the unrolled path version of the B part of the fatbin file in $S^2_{F_f}$ ), of each set of quadruplets, we calculate an upper bound $UB^{wei_{it}}_{qbt}$ on the quantity of bytes that the GF100 architecture has to transfer, from/to off-chip to/from on-chip, for the execution of the warp ELF instruction $wei_{it}$. All the upper bounds are calculated supposing that, the bytes that is necessary to transfer, by the GF100 architecture, from/to off-chip to/from on-chip, for the execution of each single warp ELF instruction $wei_{it}$, are always transfered in quantities that are multiples of 128 bytes, the dimension in bytes of a l1 cache line - see why we want this reading the possibility 2 of the $Example_3$ above - and this upper bound is the most tight possible.

To verify whether the execution of the B part of the fatbin file of a quadruplet, when the fatbin file is executed using the launch configuration of the quadruplet, for the unrolled path version of the quadruplet, for the input of the quadruplet, can not be slowed down by the bandwidths and the latencies of the GPU memories, we need to execute the following procedure on each one of the

warp ELF instructions $wei_{it}$ of the quadruplet:

- Step 1. Let us suppose a) that there is a warp $w$, among the warps W resident in a streaming multiprocessor, that requires the execution of a warp ELF instruction $wei_{it}$, b) that the warp $w$ is scheduled, for the execution of such warp ELF instruction $wei_{it}$, at the warp scheduler clock cycle $t$ and c) that, in the same streaming multiprocessor where the warp $w$ is resident, between the warp scheduler clock cycles between $t - W$ and $t$, W warp ELF instruction $wei_{it}$ have been scheduled - this as 1) one warp ELF instruction $wei_{it}$ scheduled per warp or 2) more than one warp ELF instruction $wei_{it}$ scheduled for some warps and zero warp ELF instructions $wei_{it}$ scheduled for some other warps. Because we can not know to which ELF instructions in the B part of a fatbin file, during the execution of the B part of the fatbin file, the warps W, resident in a streaming multiprocessor, are pointing, step 1 is useful because:

  - It is the worst case scenario about the number of warp ELF instructions $wei_{it}$ that can be scheduled in the last $W$ warp scheduler clock cycles in a streaming multiprocessor. A number of warp ELF instructions $wei_{it}$ equal to W is in fact the maximum number of warp ELF instructions $wei_{it}$ that can be scheduled in a time span of $W$ warp scheduler clock cycles because in each streaming multiprocessor there is only one group of 16 load/store function units to execute ELF instructions that load/store data/results - 3.3;
  - It is the worst case scenario about the warp scheduler clock cycle when the warp $w$ is scheduled for the execution of a warp ELF instruction $wei_{it}$ because the last warp ELF instruction $wei_{it}$ - the last warp ELF instruction $wei_{it}$ is the warp ELF instruction $wei_{it}$ to execute for the warp $w$ - is scheduled at the warp scheduler clock cycle $t$ instead of one of the previous warp scheduler clock cycles.

- Step 2. Let us suppose that between the warp scheduler clock cycles $t - W$ and $t$, also in each one of the other streaming multiprocessors, W warp ELF instructions $wei_{it}$ have been scheduled.

  Because we can not know to which ELF instructions of the B part of a fatbin file all the warps, not only the resident warps in a streaming multiprocessor, are pointing, during the execution of the B part of the fatbin file, and therefore which ELF instructions have been scheduled in the last $W$ warp scheduler clock cycles by the warp schedulers in the streaming multiprocessors, then, in a time span of $W$ warp scheduler clock cycles, remembering the procedure, for a single streaming multiprocessor, described in step 1, it is not possible that more than W warp ELF instructions $wei_{it}$ have been scheduled per streaming multiprocessor and so that more than a total of $S \cdot W$ warp ELF instructions $wei_{it}$ have been scheduled in the whole GPU - S is the number of streaming multiprocessors of the GPU.

  Let us also suppose that the data/results that needs to be loaded/stored for the execution of the warp ELF instruction $wei_{it}$, scheduled for the warp $w$, at the warp scheduler clock cycle $t$, a) will be the last data/results that will be loaded/stored and b) will be the last data/results, of all the $S \cdot W$ warp ELF instructions $wei_{it}$, that will arrive to the l1 cache of the streaming multiprocessor, to the shared memory of the streaming multiprocessor or to the GPU global memory.

- Step 3. We can not determine the total quantity of bytes that is necessary to load/store for the whole set of $S \cdot W$ warp ELF instructions $wei_{it}$ because also supposing the window,

that we make slide on the unrolled path versions of a quadruplet, is enough small to allow us to determine which warp ELF instructions $wei_{it}$ have been scheduled, in the last $W$ warp scheduler clock cycles, in the streaming multiprocessor where the warp $w$ is resident, we can not know which warp ELF instructions $wei_{it}$ have been scheduled, in the last $W$ warp scheduler clock cycles, for all the other resident warps in all the other streaming multiprocessors, this because the starting time differences, at global level, are of the order of the millions of function unit clock cycles, 12.2.2 - one million of function unit clock cycles is equivalent to half million of warp scheduler clock cycles - and therefore each one of the other warps in the other streaming multiprocessors could point to any ELF instruction of the B part of the fatbin file of the couple, during the execution of the B part of the fatbin file of the couple.

For this reason, we need to assume we are in the worst case scenario and so that for the execution of each one of the warp ELF instruction $wei_{it}$, scheduled in the last $W$ warp scheduler clock cycles, the GF100 architecture has to transfer, from/to off-chip to/from on-chip, the maximum possible quantity of bytes. Among all the upper bounds, on the quantity of bytes that is necessary for the GF100 architecture to transfer, from/to off-chip to/from on-chip, for the execution of each one of the warp ELF instructions $wei_{it}$, we therefore select the maximum $UB_{qbt_{max}}^{wei_{it}}$.

Because $UB_{qbt_{max}}^{wei_{it}}$ is an upper bound, on the quantity of bytes, that the GF100 architecture transfers, from/to off-chip to/from on-chip, for the execution of any warp ELF instruction $wei_{it}$, an upper bound, on the total quantity of bytes, that the GF100 architecture transfers, from/to off-chip to/from on-chip, for the execution of the whole set of $S \cdot W$ warp ELF instructions $wei_{it}$, is $UB_{tqbt} = S \cdot W \cdot UB_{qbt_{max}}^{wei_{it}}$.

- Step 4. For the discussion that follow a) we consider the GF100 architecture without l2 cache - 12.2.3 - and b) we consider that the bandwidths between the different types of GPU memories on-chip are greater than the bandwidth between the GPU global memory off-chip and the l2 cache on-chip.

  The maximum latency, in number of warp scheduler clock cycles, for the GPU global memory, is 400 warp scheduler clock cycles - [50, p. 87] and [56, p. 67] say 800 function unit clock cycles, [49, p. 47] and [55, p. 57] say 600 function unit clock cycles. Because the maximum latency, in number of warp scheduler clock cycles, for the GPU global memory, is 400, let us put ourself in the worst case scenario about the latency and so that the quantity of bytes, that the GF100 architecture has to transfer from/to off-chip to/from on-chip for the execution of each one of the warp ELF instructions $wei_{it}$, is always facing a latency of 400 warp scheduler clock cycles and therefore that if a warp ELF instruction $wei_{it}$ is scheduled at the warp scheduler clock cycle $x$ then the quantity of bytes that the GF100 architecture has to transfer from/to off-chip to/from on-chip and/or from on-chip to off-chip for the execution of the warp ELF instruction $wei_{it}$ can not being loaded/stored before of the warp scheduler clock cycle $x + 400$.

  Knowing the bandwidth in bytes per second used to transfer bytes from/to the off-chip GPU global memory to/from the on-chip memories - we know such bandwidth for every GPU model using the GF100 architecture, for the Tesla C2070 it is 144 GB/s - we calculate the bandwidth in bytes per warp scheduler clock cycle - $B_{off-on}^{b/wscc}$ - to transfer bytes from/to the off-chip GPU global memory to/from the on-chip memories.

If the execution of other warp ELF instructions $wei_{it}$, scheduled before the warp scheduler clock cycle $t - W$, is not interfering, for the use of the bandwidth used to transfer bytes from/to off-chip to/from on-chip, with execution of the warp ELF instructions $wei_{it}$, scheduled between the warp scheduler clock cycle $t - W$ and $t$, then an upper bound on the number of warp scheduler clock cycles that is necessary at the GF100 architecture to transfer from/to off-chip to/from on-chip the total quantity of bytes $UB_{tqbt} = S \cdot W \cdot UB_{qbt_{max}}^{wei_{it}}$ is $UB_{wscc}^{tqbt} = 400 + \lceil \frac{UB_{tqbt}}{B_{off-on}^{b/wscc}} \rceil$.

Considering the upper bound $UB_{wscc}^{tqbt}$ on the number of warp scheduler clock cycles that is necessary at the GF100 architecture to transfer from/to off-chip to/from on-chip the total quantity of bytes $UB_{tqbt}$ then, if the execution of other warp ELF instructions $wei_{it}$, scheduled before the warp scheduler clock cycle $t - W$, is not interfering, for the use of the bandwidth used to transfer bytes from/to off-chip to/from on-chip, with the execution of the warp ELF instructions $wei_{it}$, scheduled between the warp scheduler clock cycle $t - W$ and $t$, the quantity of bytes that the GF100 architecture has to transfer for the execution of the warp ELF instruction $wei_{it}$, of the warp $w$, scheduled at the warp scheduler clock cycle $t$, will be a) in the ll cache of the streaming multiprocessor where the warp $w$ is resident, b) in the shared memory of the streaming multiprocessor where the warp $w$ is resident or c) in the GPU global memory, not later than the warp scheduler clock cycle $t + UB_{wscc}^{tqbt}$.

- Step 5. Some of the bytes that the GF100 architecture has to transfer for the execution of the warp ELF instruction $wei_{it}$, scheduled at the warp scheduler clock cycle $t$, for the warp $w$, could be used by some other warp ELF instructions in the B part of the fatbin file of the couple. This happens when some or all the bytes that the GF100 architecture has to transfer for the execution of the warp ELF instruction $wei_{it}$, scheduled at the warp scheduler clock cycle $t$, for the warp $w$, have to be read for the execution of some other warp ELF instructions - think for example a) to some data, transfered from the GPU global memory to the l1 cache, that are going to be used for the execution of some warp ELF instructions or b) some partial results that are transfered from the l1 cache or the shared memory of a streaming multiprocessor to the GPU global memory but later need to be transfered again from/to off-chip to/from on-chip.

  Let us a) call $wei_f$ the first of these warp ELF instructions, after the warp ELF instruction $wei_{it}$, scheduled at the warp scheduler clock cycle $t$, for the warp $w$ and b) calculate the distance $D_{nwei}$, in number of warp ELF instructions, between the warp ELF instruction $wei_{it}$ and the warp ELF instruction $wei_f$, in the unrolled path version of the B part of the fatbin file of the couple.

  At each warp scheduler clock cycle not more than 2 warps can be scheduled in a streaming multiprocessor. A lower bound, on the number of warp scheduler clock cycles, that has to pass after the warp scheduler clock cycle $t$, before the warp $w$ can be considered again for the scheduling of the warp ELF instruction $ei_f$, is therefore $LB_{wscc}^{t_1} = D_{nwei} \lfloor \frac{W}{2} \rfloor$.

  That the execution of other warp ELF instructions $wei_{it}$, scheduled before the warp scheduler clock cycle $t - W$, is not interfering, for the use of the bandwidth, used by the GF100 architecture to transfer bytes from/to off-chip to/from on-chip, with the execution of the warp ELF instructions $wei_{it}$, scheduled between the warp scheduler clock cycle $t - W$ and $t$, is important because otherwise the bytes that the GF100 architecture has to transfer for the

execution of the warp ELF instruction $wei_{it}$, scheduled at the warp scheduler clock cycle $t$, for the warp $w$, could not be where they need to be - and so a) in the l1 cache of the streaming multiprocessor where the warp $w$ is resident, b) in the shared memory of the streaming multiprocessor where the warp $w$ is resident or c) in the GPU global memory - at the warp scheduler clock cycle $t + LB^{t_1}_{wscc}$ - we check whether this is the case in step 6.

Supposing there is no interference, if $t + LB^{t_1}_{wscc}$ is equal or greater than $t + UB^{tqbt}_{wscc}$ then the bandwidths and the latencies of the GPU memories can not slow down the execution of the B part of the fatbin file of the couple, when the fatbin file is executed using the launch configuration of the couple, for any of the input that generates the unrolled path version of the B part of the fatbin file of the couple, at cause of the transfer of the bytes necessary for the execution of the warp ELF instruction $wei_{it}$, scheduled at the warp scheduler clock cycle $t$, for the warp $w$.

- Step 6. We need to be sure that the execution of other warp ELF instructions $wei_{it}$, scheduled before the warp scheduler clock cycle $t - W$, is not interfering, for the use of the bandwidth, used by the GF100 architecture to transfer bytes from/to off-chip to/from on-chip, with the execution of the warp ELF instructions $wei_{it}$, scheduled between the warp scheduler clock cycle $t - W$ and $t$.

  In the unrolled path version of the quadruplet, we determine the distances, in number of warp ELF instructions, between all the consecutive couples of warp ELF instructions $wei_{it}$ and among these distances we select the minimum distance $md_{wei_{it}}$. Considering the position, in the unrolled path version of the quadruplet, of the warp ELF instruction $wei_{it}$, that is scheduled, as warp ELF instruction, at the warp scheduler clock cycle $t$, for the warp $w$, we know that it is impossible that in the previous $md_{wei_{it}} - 1$ warp ELF instructions, in the unrolled path version of the quadruplet, there are some ELF instructions $wei_{it}$.

  Because at each warp scheduler clock cycle not more than 2 warps can be scheduled in a streaming multiprocessor, a lower bound, in number of warp scheduler clock cycles, required to the 2 warp schedulers, resident in the streaming multiprocessor where the warp $w$ is resident, to move the warp $w$ from the execution of a warp ELF instruction $x$ to a warp ELF instruction $x + md_{wei_{it}}$, is $LB^{t_2}_{wscc} = md_{wei_{it}} \lfloor \frac{W}{2} \rfloor$ and therefore the warp scheduler clock cycle at which the warp $w$ could have been scheduled for the execution of the warp ELF instruction $wei_{it}$, that precede the warp ELF instruction $wei_{it}$, for which the warp $w$ is scheduled at the warp scheduler clock cycle $t$, can not be a warp scheduler clock cycle after the warp scheduler clock cycle $t - LB^{t_2}_{wscc}$.

  If the execution of other warp ELF instructions $wei_{it}$, scheduled before the warp scheduler clock cycle $t - W$, interferes, for the use of the bandwidth, used by the GF100 architecture to transfer bytes from/to off-chip to/from on-chip, with the execution of the warp ELF instructions $wei_{it}$, scheduled between the warp scheduler clock cycle $t - W$ and $t$, then their interference can be not greater than the case when all them - the other warp ELF instructions $wei_{it}$ - are scheduled between the warp scheduler clock cycles $t - LB^{t_2}_{wscc} - W$ and $t - LB^{t_2}_{wscc}$.

  If $t - LB^{t_2}_{wscc} + UB^{tqbt}_{wscc}$ is equal or smaller than $t + 400$ then we have the guarantee that the execution of the warp ELF instructions $wei_{it}$, scheduled between the warp scheduler clock cycle $t - LB^{t_2}_{wscc} - W$ and $t - LB^{t_2}_{wscc}$, is not interfering, for the use of the bandwidth, used by the GF100 architecture to transfer bytes from/to off-chip to/from on-chip, with the

execution of the warp ELF instructions $wei_{it}$, scheduled between the warp scheduler clock cycle $t - W$ and $t$, and so the bytes transfered for the execution of the warp ELF instruction $wei_{it}$, scheduled at the warp scheduler clock cycle $t$, for the warp $w$, will be of course a) in the l1 cache of the streaming multiprocessor where the warp $w$ is resident, b) in the shared memory of the streaming multiprocessor where the warp $w$ is resident or c) in the GPU global memory, not later than the warp scheduler clock cycle $t + UB^{tqbt}_{wscc}$ and so not later than the warp scheduler clock cycle $t + LB^{t_1}_{wscc}$.

Furthermore, if $t - LB^{t_2}_{wscc} + UB^{tqbt}_{wscc}$ is equal or smaller than $t + 400$ then it is not necessary to repeat the previous procedure backward till at the beginning of the unrolled path version of the quadruplet because it is impossible that the execution of previous groups of warp ELF instructions $wei_{it}$, scheduled in groups of $W$ warp scheduler clock cycles as indicated, can make possible that the bytes, that the GF100 architecture has to transfer from/to off-chip to/from on-chip, for the execution of the warp ELF instruction $wei_{it}$, scheduled at the warp scheduler clock cycle $t$, for the warp $w$, are not, after the warp scheduler clock cycle $t+UB^{tqbt}_{wscc}$ and so at the warp scheduler clock cycle $t + LB^{t_1}_{wscc}$, a) in the l1 cache of the streaming multiprocessor where the warp $w$ is resident, b) in the shared memory of the streaming multiprocessor where the warp $w$ is resident or c) in the GPU global memory.

If a warp ELF instruction $wei_{it}$ is scheduled for a warp $w$ resident in a streaming multiprocessor at the warp scheduler clock cycle $t$ then, at the warp scheduler clock cycles between $t - W$ and $t - 1$, because the proofs consider the worst case scenarios for all the factors involved, it is not necessary that a) in the same streaming multiprocessor, a warp ELF instruction $wei_{it}$ has been scheduled at each one of the warp scheduler clock cycles between $t - W$ and $t - 1$ and b) in each one of the other streaming multiprocessors, a warp ELF instruction $wei_{it}$ has been scheduled at each one of the warp scheduler clock cycles between $t - W$ and $t$, and therefore, without further calculations, if $t - LB^{t_2}_{wscc} + UB^{tqbt}_{wscc}$ is equal or smaller than $t + 400$ then we get the guarantee that, independently of when the warp ELF instructions $wei_{it}$ of the B part of the fatbin file of the quadruplet, the bandwidths and the latencies of the GPU memories can not slow down the execution of the B part of the fatbin file of the quadruplet, when the fatbin file is executed with the launch configuration of the quadruplet, for the input of the quadruplet.

If instead $t - LB^{t_2}_{wscc} + UB^{tqbt}_{wscc}$ is greater than $t + 400$ then we can not have the guarantee that the execution of other warp ELF instructions $wei_{it}$, scheduled before the warp scheduler clock cycle $t - W$, is not interfering, for the use of the bandwidth, used by the GF100 architecture to transfer bytes from/to off-chip to/from on-chip, with the execution of the warp ELF instructions $wei_{it}$, scheduled between the warp scheduler clock cycle $t - W$ and $t$, and so the bytes, that the GF100 architecture has to transfer from/to off-chip to/from on-chip, for the execution of the warp ELF instruction $wei_{it}$, scheduled at the warp scheduler clock cycle $t$, for the warp $w$, could not be a) in the l1 cache of the streaming multiprocessor where the warp $w$ is resident, b) in the shared memory of the streaming multiprocessor where the warp $w$ is resident or c) in the GPU global memory, at the warp scheduler clock cycle $t + LB^{t_1}_{wscc}$.

If this would be the case then we need to discard the quadruplet because the possibility, that the bytes, that GF100 architecture has to transfer from/to off-chip to/from on-chip, for the execution of the warp ELF instruction $wei_{it}$, scheduled at the warp scheduler clock cycle $t$,

for the warp $w$, are or not a) in the l1 cache of the streaming multiprocessor where the warp $w$ is resident, b) in the shared memory of the streaming multiprocessor where the warp $w$ is resident or c) in the GPU global memory, at the warp scheduler clock cycle $t + LB^{t_1}_{wscc}$, depends on factors that we can not know, quantify, choose or force.

If also only one of the warp ELF instructions $wei_{it}$ of a quadruplet does not pass the previous test, also if the quadruplet could satisfy all the requirements of the subanalysis on the number of resident warps in each streaming multiprocessor, the execution of the B part of the fatbin file of the quadruplet, when the fatbin file is executed using the launch configuration of the quadruplet, for the unrolled path version of the quadruplet, for the input of the quadruplet, could be slowed down by the bandwidths and the latencies of the GPU memories - there is no really way to know this.

If for any reason, during the execution of the B part of the fatbin file of a quadruplet, when the fatbin file is executed using the launch configuration of the quadruplet, for the unrolled path version of the quadruplet, for the input of the quadruplet, also only one slowdown is generated by the bandwidths and the latencies of the GPU memories then a) each single slowdown generated by the bandwidths and the latencies of the GPU memories can generate any other number and type of slowdowns - slowdowns due to the warp scheduling, slowdowns due to the scheduling waiting times, slowdowns due to the dependence waiting times and/or slowdowns due to the overhead time for the management of the warps - and b) each one of the new slowdowns can generate any other number and type of slowdowns - avalanche effect.

If instead all the warp ELF instruction $wei_{it}$ of a quadruplet pass the previous test then we have the guarantee that, if the quadruplet also satisfies all the requirements of the subanalysis on the number of resident warps in each streaming multiprocessor, the execution of the B part of the fatbin file of the quadruplet, when the fatbin file is executed with the launch configuration of the quadruplet, for the unrolled path version of quadruplet, for any input - not only the input of the quadruplet - a) considered or not in the analysis $A_1$, b) generating the unrolled path version of the B part of the fatbin file of the quadruplet and c) generating an $UB^{wei_{it}}_{qbt_{max}}$ equal or smaller than the $UB^{wei_{it}}_{qbt_{max}}$ of the quadruplet, can be slowed down a) only by the warp scheduling - that we can not know, choose or force - and b) that each single slowdown generated by the warp scheduling is not going to generate any other slowdown.

However, at the beginning of the execution of the B part of the fatbin file of a quadruplet, between the warp scheduler clock cycle when all the warps used to execute the fatbin file are synchronized and the warp scheduler clock cycle $WSCC$, the 2 warps schedulers in each one of the streaming multiprocessors could cycle on a number of warps that is smaller than the number of warps that is resident in the streaming multiprocessor where the 2 warp schedulers are - case $C_3$ in 9.4.5 - and so to be sure the proofs given about the bandwidths and the latencies of the GPU memories are correct, if the unrolled path version of the quadruplet satisfies the requirements of the subanalysis on the bandwidths and the latencies of the GPU memories then we need to modify the B part of the fatbin file of the quadruplet introducing, just after the three ELF instruction used to synchronizes the warps, $MD_{nwei}$ nop - not operation - ELF instructions, this to be sure that no warp ELF instruction $wei_{it}$ is scheduled before of the warp scheduler clock cycle $WSCC$ after which we are sure that the 2 warps schedulers in each one of the streaming multiprocessors are cycling on all the resident warps in the streaming multiprocessor where the 2 warp schedulers are.

## 12.3 Number of Resident Warps in Each Streaming Multiprocessor

Each ELF instruction in the B part of the fatbin file of a quadruplet is using some ELF registers. Considering the B part of the fatbin file of a quadruplet, each ELF register of each ELF instruction has one or more dependence distances equal to the number of ELF instructions, in the unrolled path version of the quadruplet, between the ELF instruction where the ELF register is used and the next ELF instruction that uses the register.

For each unrolled path version of a quadruplet, for each ELF register of an ELF instruction, we create the corresponding couples ( instruction configuration , dependence distance ). The ELF register used as result gives us the dependence distance for the ELF instruction configuration, of the ELF instruction, where the ELF register, used as result, of the ELF instruction, has a type of dependence write-read, while the ELF registers used as operands give us the dependence distances for the ELF instruction configuration, of the ELF instruction, where the ELF registers, used as operands, have a type of dependence read-read.

For each ELF instruction, we therefore retrieve the minimum number of warps $min^{ei}_{w_{sm}}$, resident in a streaming multiprocessor, that is necessary to get the real ELF instruction streaming multiprocessor best average performance per clock cycle. To do this we first retrieve the minimum of each couple ( instruction configuration , dependence distance ) of the two ELF instruction configurations corresponding to the ELF instruction - two because one is for the dependence type write-read and the other is for the dependence type read-read but note that if more different ELF registers are used as operands of the ELF instruction then we have more couples ( instruction configuration , dependence distance ) with a dependence type read-read - and next we take the maximum - $min^{ei}_{w_{sm}}$ - of the minimums - the maximum $min^{ei}_{w_{sm}}$ is the minimum number of warps $min^{ei}_{w_{sm}}$, resident in a streaming multiprocessor, that is necessary to get the real ELF instruction streaming multiprocessor best average performance per clock cycle considering at the same time all the dependence distances of all the dependence types of all the ELF registers used in the ELF instruction.

At this point we take the maximum of the maximums $min^{ei}_{w_{sm}}$ - we have a maximum $min^{ei}_{w_{sm}}$ for each one of the ELF instructions in the unrolled path version of the quadruplet. Let us call the maximum of the maximums $min^{2}_{w_{sm}}$. $min^{2}_{w_{sm}}$ is a lower bound on the minimum number of warps that have to be resident in each streaming multiprocessor, during the execution of the fatbin file of the quadruplet, when the fatbin file is executed using the launch configuration of the quadruplet, for the input of the quadruplet, to get the guarantee that the execution of the B part of the fatbin file of the quadruplet, when the fatbin file is executed using the launch configuration of the quadruplet, for the input of the quadruplet, can not be slowed down by the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps - this supposing the unrolled path version of the B part of the fatbin file of the couple satisfies all the requirements of the subanalysis on the bandwidths and the latencies of the GPU memories.

Because we have calculated two of these types of lower bounds - $min^{1}_{w_{sm}}$ in 12.2.6 and $min^{2}_{w_{sm}}$ here - we take the maximum - $min_{w_{sm}} = max(min^{1}_{w_{sm}}, min^{2}_{w_{sm}})$ - of the two. $min_{w_{sm}}$ is the minimum number of warps that have to be resident in each streaming multiprocessor to get the guarantee, if the unrolled path version of the B part of the fatbin file of the quadruplet satisfies also all the requirements of the subanalysis on the bandwidths and the latencies of the GPU memories, a) that the execution of the B part of the fatbin file of the quadruplet, when the fatbin file is executed using the launch configuration of the quadruplet, for the unrolled path version of the

quadruplet, for the input of the quadruplet, can not be slowed down by the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps but can only be slowed down by the warp scheduling - that we can not know, choose or control - and b) that each single slowdown generated by the warp scheduling can not generate any other slowdown.

If instead $min_{w_{sm}}$ is greater than the number of resident warps in a streaming multiprocessor that we obtain, using the launch configuration of the quadruplet, for the execution of the B part of the fatbin file of the quadruplet, for the unrolled path version of the quadruplet, for the input of the quadruplet, then the execution of the B part of the fatbin file of the quadruplet, when the fatbin file is executed using the launch configuration of the quadruplet, for the unrolled path version of the quadruplet, for the input of the quadruplet, could be slowed down by the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps. Each single slowdown of one of these three types could generate any other number and type of one of these three types of slowdowns and wrap scheduling slowdowns and each one of the new slowdowns generated could originate further slowdowns of one of these four types - avalanche effect - but if the quadruplet satisfies all the requirements of the subanalysis on the bandwidths and the latencies of the GPU memories then it is impossible for any of the slowdowns, due to the warp scheduling, the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps, to generate slowdowns due to the bandwidths and the latencies of the GPU memories.

The reader could notice that when we quantified the dependence waiting times, the scheduling waiting times and the overhead time for the management of the warps for each ELF instruction of interest we used several fatbin files - 7.4 - but that each one of them have only one type of ELF instruction configuration in the single for loop in the B part of each one of the fatbin files - this if we exclude the three ELF instructions to check whether it is necessary to iterate on the for loop. The reader could therefore ask how it possible that we can generalize the quantifications of the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps to fatbin file that in their B parts can be very different from the fatbin files used in 7.

Each ELF instruction requires a given number of hardware resources to be executed. Some of these hardware resources are visible in the human readable text form representation of the ELF instruction while others no - think, for example, to the group or to the groups of function units that can be used to execute a warp ELF instruction or at the case when an ELF instruction requires the use of some not disclosed hardware resources different from the special registers that sometimes are visible in the human readable text form representations of the ELF instructions.

The values of the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps are the worst possible for each one of the ELF instruction configurations considered in 7. This is true because the ELF instruction configurations, in the for loop, of the B part, of each one of the fatbin files, used in 7, to quantify the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps of an ELF instruction, are the same - this if we exclude the three ELF instructions to check whether it is necessary to iterate on the for loop - and so - excluded the ELF registers, that are visible in the human readable text form representation of each one of the ELF instruction configurations, and that each one of the ELF instruction configurations could have different - each one of the ELF instruction configurations in the for loop of a single fatbin file requires the use of the same hardware resources when singularly executed.

When we have a fatbin file with many different ELF instructions in its B part, independently of which ELF instructions in its B part the resident warps in a streaming multiprocessor are pointing,

the set of hardware resources - different from the hardware registers that correspond to the ELF registers - necessary to execute the 2 warp ELF instructions of the maximum of 2 warps that can be scheduled by the 2 warp schedulers at a warp scheduler clock cycle, can give only three cases: 1) the sets of hardware resources required for the execution of each one of the 2 warp ELF instructions are completely different, 2) the sets of hardware resources required for the execution of each one of the 2 warp ELF instructions have an intersection that is not empty, 3) the sets of hardware resources required for the execution of each one of the 2 warp ELF instruction are the same.

For all the fatbin files in 7 we are in case 3. Case 3 is the worst case that we can have for the reuse of the hardware resources used to execute a warp ELF instruction and so the worst case for the quantifications a) of the scheduling waiting times, b) the dependence waiting times and c) the overhead time for the management of the warps. The reason why case 3 is the worst case, it is due to the fact that all the hardware resources - at exclusion of the hardware registers that correspond to the ELF registers - used for the execution of a warp ELF instruction, have to be used for the execution of the next warp ELF instruction. For this reason, the quantifications of the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps got in 7 are all upper bounds for each one of the ELF instruction configurations. Considering also that a) the quantification about the minimum number of resident warps in a streaming multiprocessor, necessary to get the real ELF instruction configuration streaming multiprocessor best average performance per clock cycle, for each dependence distance of each ELF instruction configuration, is always determined considering, at the same time, the influence of all three these concurrent factors - the warp scheduling time, the dependence waiting time and the overhead time for the management of the warps - b) that we are always using the values that these three factors have for the dependence distance of the ELF instruction configuration and c) that each of these three factors can not be greater than what it was - this because we are in case 3 and therefore each one of their quantifications for each one of the dependence distances of each one of the ELF instruction configurations is always an upper bound - then, because 1) we execute the subanalysis on the number of resident warps in each streaming multiprocessor using $min_{w_{sm}}$ and 2) we execute the subanalysis on the bandwidths and the latencies of the GPU memories considering that all the warps, used for the execution of the B part of the fatbin file of a quadruplet, could point to any ELF instructions in the B part of the fatbin file of a quadruplet - this also whether we know that, when the quadruplet satisfies all the requirements of the analysis $A_1$, at least all the resident warps in the same streaming multiprocessor are always not more distant than $MD_{nwei}$ warp ELF instructions - we are sure that cases worst of those present during the execution of the B part of the fatbin files used in 7 can never happen during the execution of B parts of fatbin files with many different ELF instructions.

## 12.4 Summary

In this chapter we have described the procedures that it is necessary to execute to verify whether a couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) satisfies all the requirements of the analysis $A_1$. We can distinguish two cases:

- A couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) satisfies all the requirements of the analysis $A_1$. In this case we give an *a priori* ELF code shortest execution time guarantee for the execution of the B part of the fatbin file of the couple, when

the fatbin file of the couple is executed using the launch configuration of the couple, for any input a) considered or not in the analysis $A_1$, b) generating one of the unrolled path versions - 12.2.7 - generated by the inputs of the set $S_i$ of inputs - 12.2.7 - and c) with a $UB^{wei_{it}}_{qbt_{max}}$ - 12.2.8 - smaller or equal of the greatest $UB^{wei_{it}}_{qbt_{max}}$ of the unrolled path versions generated by the inputs of the set $S_i$ of inputs;

- A couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ) does not satisfy all the requirements of the analysis $A_1$. In this case it could be that some of the quadruplets ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ , unrolled path version of the B part of the fatbin file in $S^2_{F_f}$ , an input, of one of the subsets $SS_i$, of inputs generating the unrolled path version of the B part of the fatbin file in $S^2_{F_f}$ ) - 12.2.7 - generated to analyze the couple ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ ), satisfy all the requirements of the analysis $A_1$. If this is the case then the quadruplets satisfying all the requirements of the analysis $A_1$ can be subdivided in subsets, a subset for each different unrolled path version of the quadruplets. In each subset one or more quadruplets have the greatest $UB^{wei_{it}}_{qbt_{max}}$. For any input a) considered or not in the analysis $A_1$ and b) generating an unrolled path version of the B part of the fatbin file of the couple that 1) is an unrolled path version representing one of the subsets of quadruplets generated using the inputs of the set $S_i$ of inputs and 2) has a $UB^{wei_{it}}_{qbt_{max}}$, for the input, smaller or equal than the greatest $UB^{wei_{it}}_{qbt_{max}}$ of the subset with the unrolled path version equal to the unrolled path version generated by the input, we give an *a priori* ELF code shortest execution time guarantee;

The more important points to remember from this chapter are the followings:

- If a quadruplet satisfies all the requirements of the analysis $A_1$ then, also we can not know to which ELF instructions, in the B part of the fatbin file of the quadruplet, the resident warps in a streaming multiprocessor are pointing during the execution of the B part of the fatbin file, when the fatbin file is executed using the launch configuration of the quadruplet, for the unrolled version of the quadruplet, for the input of the quadruplet, we prove that the leading warp or the leading subset of resident warps in a streaming multiprocessor can not be more distant than $MD_{nwei}$ warp ELF instructions from the last warp or the last subset of resident warps in the same streaming multiprocessor, this for almost the whole execution of the B part of the fatbin file - the beginning and the ending are excluded because not all the resident warps in a streaming multiprocessor start and finish together but instead some warps will start first of others and will finish first of others.

  $MD_{nwei}$ depends on a) the set $S_{dd}$ of dependence distances of the fatbin file of the quadruplet and b) the launch configuration of the quadruplet - the launch configuration determines the number of resident warps in each streaming multiprocessor during the execution of a fatbin file. Considering the results got for the starting time differences in 7 $MD_{nwei}$ can go from a minimum of 0 to a maximum of 74 - 74 can be reached when there are 32 resident warps in a streaming multiprocessor;

- If a quadruplet satisfies all the requirements of the analysis $A_1$ then, also whether we can not know to which ELF instructions in the B part of the fatbin file of the quadruplet, the resident warps in a streaming multiprocessor are pointing during the execution of the B part of the fatbin file, when the fatbin file is executed using the launch configuration of the quadruplet,

for the unrolled version of the quadruplet, for the input of the quadruplet, we prove that, for almost the whole execution of the B part of the fatbin file, if there is a number of resident warps in each streaming multiprocessor greater than $min^1_{w_{sm}}$ - $min^1_{w_{sm}}$ depends on the ELF instructions in the unrolled path version of the quadruplet - the value of the distance, in number of warp ELF instructions, between the 2 warps of each possible couple of resident warps in a streaming multiprocessor, is going to oscillate not more than plus minus 2 around the value that the distance of the couple has at the first warp scheduler clock cycle when all the resident warps in the streaming multiprocessor have been scheduled at least one time after their synchronization at the beginning of the B part of the fatbin file;

- Knowing that, if a quadruplet satisfies all the requirements of the analysis $A_1$ then the leading warp or the leading subset of resident warps in a streaming multiprocessor can not be more distant than $MD_{nwei}$ warp ELF instructions from the last warp or the last subset of resident warps in the same streaming multiprocessor, this for almost the whole execution of the B part of the fatbin file of the quadruplet, when the fatbin file is executed using the launch configuration of the quadruplet, for the unrolled path version of the quadruplet, for the input of the quadruplet, we explain a) how to determine the warp ELF instructions $wei_{it}$, in the unrolled path version of the quadruplet, that imply the transfer of bytes from off-chip to on-chip and/or from on-chip to off-chip during the execution of the B part of the fatbin file of the quadruplet, when the fatbin file is executed with the launch configuration of the quadruplet, for the unrolled path version of the quadruplet, for the input of the quadruplet and b) what to do if instead these warp ELF instructions $wei_{it}$ can not be determined because they could be different by execution to execution of the B part of the fatbin file of the quadruplet, when the fatbin file is executed with the launch configuration of the quadruplet, for the unrolled path version of the quadruplet, for the input of the quadruplet.

- Knowing the warp ELF instruction $wei_{it}$, in the unrolled path version of the quadruplet, that imply the transfer of bytes from off-chip to on-chip and/or from on-chip to off-chip during the execution of the B part of the fatbin file of the quadruplet, when the fatbin file is executed with the launch configuration of the quadruplet, for the unrolled path version of the quadruplet, for the input of the quadruplet, we prove whether the execution of the B part of the fatbin file of the quadruplet, when the fatbin file is executed with the launch configuration of the quadruplet, for the unrolled path version of the quadruplet, for the input of the quadruplet, can not be slowed down by the bandwidths and the latencies of the GPU memories;

- If a quadruplet satisfies all the requirements of the subanalysis on the bandwidths and the latencies of the GPU memories then, knowing the number of resident warps in each streaming multiprocessor for the execution of fatbin file of the quadruplet, when the fatbin file is executed using the launch configuration of the quadruplet, for the unrolled path version of the quadruplet, for the input of the quadruplet, we prove whether the execution of the B part of the fatbin file of the quadruplet, when the fatbin file is executed with the launch configuration of the quadruplet, for the unrolled path version of the quadruplet, for the input of the quadruplet, can not be slowed down by the scheduling waiting times, the dependence waiting times and the overhead for the management of the warps.

In the next chapter we explain a) how with this thesis we have solved several challenges that nobody - at the best of our knowledge - had solved or addressed in papers in literature, and b)

because in our opinion it is important that we have addressed and solved these challenges.

# Chapter 13

# Contributions of the Thesis

## 13.1 Introduction

In the previous chapter we have described the analysis $A_1$ and explained what it is necessary for a quadruplet ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ , unrolled path version of the B part of the fatbin file in $S^2_{F_f}$ , an input, of one of the subsets $SS_i$, of inputs generating the unrolled path version of the B part of the fatbin file in $S^2_{F_f}$ ) to satisfy all the requirements of the analysis $A_1$ and so allowing us to give an *a priori* ELF code shortest execution time guarantee for the execution of the ELF code, of the B part, of the fatbin file, of the quadruplet, when the fatbin file is executed using the launch configuration of the quadruplet, for any input - not only the input of the quadruplet - a) considered or not in the analysis $A_1$, b) generating the unrolled path version of the B part of the fatbin file of the quadruplet - 12.2.7 - and c) generating an $UB^{wei_{it}}_{qbt_{max}}$ equal or smaller than the $UB^{wei_{it}}_{qbt_{max}}$ of the quadruplet - 12.2.8.

In the this chapter we explain a) how with this thesis we have solved several challenges that nobody - at the best of our knowledge - had solved or addressed in papers in literature, and b) because in our opinion it is important that we have addressed and solved these challenges.

In 13.2 we explain because it was important to reverse engineer the real ISA and being able to modify ELF codes to get the wanted ELF algorithmic implementations - for a greater quantity of details see the summary of 6 and 6. We describe in 13.2.1 how we are able to localize in a fatbin file - see 2.3 to understand what it is a fatbin file - the ELF code that corresponds to the PTX code - see 2.2 for the definition of PTX - given in input to nvcc - see 2.3 to understand what it is nvcc. Next in 13.2.2 we explain because it is important to use the editing rules that we have given to force nvcc to generate fatbin files. We therefore talk in 13.2.3 of the PTX-ELF correspondences that we have discovered between PTX and ELF instructions. In 13.2.4 we instead give an explanation of because it was important to reverse engineer the real instruction set architecture and we explain that we have found that the real instruction set architecture is not at fixed format but that this is not a problem because we have successfully reverse engineered the binary code of each possible ELF instruction we need to modify any fatbin file. In 13.2.5, we explain that thanks to these results we are now able to get any wanted ELF algorithmic implementation we want executed by a GF100 architecture.

In 13.3 we explain because it was important to discover, understand and quantify some not disclosed GPU behaviors. We start talking of the importance that the B parts of the fatbin files - used for the discoveries, the understanding and the quantifications of the not disclosed GPU

behaviors - are generated in specific ways - the B part of a fatbin file, 6.6, is the part of the fatbin file that is composed by the ELF code that corresponds to the PTX code given in input to nvcc. In 13.3.1 we explain what we have discovered about the advancement of the resident warps in a streaming multiprocessor - see 3.2 to understand what it is a warp and 3.3 to understand what it is a streaming multiprocessor. Next in 13.3.2 we explain what it is necessary to do to get the guarantee that the gigathread scheduler is going to evenly distribute to the streaming multiprocessors the GPU thread blocks used to execute the B part of a fatbin file. We therefore talk in 13.3.3 of what it is necessary to do to avoid warp scheduling load unbalancing in a streaming multiprocessor. Finally in 13.3.4 we talk of the importance of the discovery, understanding and quantification - see 7.5.2 - of the following local streaming multiprocessor PTX and ELF architectural features: a) the real instruction configurations streaming multiprocessor best average performance per clock cycle of the PTX and ELF instruction configurations - see 2.6 for the definition of instruction configuration - b) the scheduling waiting times of the ELF instruction configurations - see 7.5.2 for the definition of scheduling waiting time - c) the dependence waiting times of the ELF instruction configurations - see 7.5.2 for the definition of dependence waiting time - d) the overhead time for the management of the warps and e) the minimum number of resident warps necessary in a streaming multiprocessor to get the real instruction configuration streaming multiprocessor best average performance per clock cycle of each ELF instruction configuration for each dependence distance - see 2.6.2 for the definition of dependence distance.

In 13.4 we explain because it is important a) to transform an original fatbin file, that we want to optimize, in a set of fatbin files equivalent to the original fatbin file - 8.4 - and b) to generate a set of launch configurations for each one of the fatbin files generated - see 8.3 for the procedure and 2.5 for the definition of launch configuration - to increase the probability to get shorter execution times for the B parts of the fatbin files generated - this compared to the execution times that the reader would get if he/she would execute only the B part of the original fatbin file with a launch configuration of his/her choice.

In 13.5 we explain because it is important to analyze a fatbin file considering several things. In 13.5.1 we explain that some of the previous things determine the position of the fatbin file in a taxonomy for fatbin files that we have created - 10. In 13.5.2 we explain that the position of a fatbin file in the taxonomy is one of the two things that determine the analysis/analyses - the empirical one and/or the theoretical one, see 11 - that can be executed on the fatbin file. Finally in 13.5.3 we explain that it is possible, executing the theoretical analysis that we have devised - 12 - to give an *a priori* ELF code shortest execution time guarantee for the execution of the ELF code of the B part of a fatbin file.

## 13.2 Real ISA and ELF Codes

To be able to get the wanted ELF algorithmic implementations it is necessary to reverse engineer several aspects of the GF100 architecture. Being able to get the wanted ELF algorithmic implementations is important because if we can only use CUDA or PTX to edit code then nvcc can each time completely ruin the modifications and/or transformations that we apply to a fatbin file during the optimization process - the modifications and/or transformations have as goal the reduction of the execution time of the ELF part/parts of the fatbin file that will be executed by the target GPU for which the fatbin file has been compiled.

### 13.2.1 Localization in Fatbin Files of the ELF Instructions Necessary to Execute the PTX Instructions of PTX Codes

We are not able to find in literature any paper that shows or studies the structure of the fatbin files generated as output by nvcc. We have found that also in the simplest case when we give in input to nvcc files without code that has to be executed by the CPU - PTX files for example - the fatbin files produced by nvcc - fatbin files that therefore contain only code that is executed by the GPU - have many more ELF instructions of the ELF instructions generated by the procedure of transformation of the PTX code in ELF code. We know this because analyzing the interpretation text files generated by cuobjdump - a NVIDIA tool able to "interpret a fatbin file" - we have found that the number of ELF instructions in the interpretation text files times 8 - the number of bytes of each ELF instruction - is always smaller than the dimension in bytes of the fatbin file - 6.2.

We devised a robust procedure - 6.2 - able to always individuate in a fatbin file the ELF instructions generated by the procedure of transformation of the PTX code, thing not easy because a) the real instruction set architecture is not disclosed and b) cuobjdump in reality shows a permutation of the 8 bytes of each one of the ELF instructions generated by the procedure of transformation of the PTX code - we discovered this because searching in any fatbin file the binary codes showed by cuobjdump we were not able to find them.

Being able to understand the real position of the bytes, of the binary codes showed by cuobjdump, in the binary code of each ELF instruction, is important to be able to individuate in which part of the fatbin file are the ELF instructions generated by the procedure of transformation of the PTX code - this is the part of the fatbin file that we have defined as the B part and that is composed only from the ELF instructions a) generated by the procedure of transformation of the PTX code and b) visible in the interpretation text file generated by cuobjdump for the fatbin file, 6.6.

Being able to individuate the B part of a fatbin file it is instead important to be able to modify the B part of the fatbin file to get the wanted ELF algorithmic implementation.

### 13.2.2 Editing Rules to Force Nvcc

NVIDIA does not allow to users to write in the assembly - the ELF, see 2.3 - executed by the GPU. When we write a PTX code, the ELF code - corresponding to the PTX code - executed by the GPU, is usually very different by the PTX code a) for number, order and type of instructions and b) for number, type and reuse of registers - 6.1. The fact that the ELF code, corresponding to a PTX code, is very different from the PTX code, it is usually - for not saying pretty much always - overlooked in the papers in literature - this happens for papers considering CPUs too.

If the analyses of a code, that has to be executed on a machine, are based on something of different from the assembly representation of the code that has to be executed on the machine, then the analyses are usually meaningless and not correct because the assembly representation of the code executed by the machine - CPU or GPU - is usually very different from the higher representation of the code written by an user, while for the few cases when the assembly representation of the code executed by the machine - CPU or GPU - is mirroring the higher representation of the code written by an user a) for number, order and type of instructions and b) for number, type and reuse of registers, the results are not generalizable - you could get a completely different assembly code 1) changing the version of the compiler, 2) changing the compiler, 3) changing the value of any of the flags used to compile the code, 4) adding/removing some of the flags, 5) changing the drivers

of the part of the machine where has to be executed the assembly code, 6) changing the version of the operative system of the machine or 7) changing the operative system of the machine.

The first thing to do it is therefore to base any analysis of any code to execute on a machine on the assembly representation of the code that has to be executed on the machine. The second thing to do, in the case it is not possible to edit the code using the assembly of the machine - as it happens in our case - it is a) to force the compiler to generate the wanted assembly algorithmic implementation or b) to force the compiler to generate an assembly file with at least the minimum number of resources later necessary to modify the assembly file to get the wanted assembly algorithmic implementation.

We are not able, at this moment, to force nvcc to generate the wanted ELF algorithmic implementation, this because a) the nvcc code is not open and b) we believe that also if we could be able to do that then it would be hard to generalize the results to other versions of nvcc. For the last reason, we therefore believe it is better to force nvcc to generate a fatbin file with at least the minimum number of resources later necessary to modify the fatbin file to get the wanted ELF algorithmic implementation. This, in our opinion, is an easier - but however difficult goal - compared to that of forcing nvcc to generate a fatbin file with the wanted ELF algorithmic implementation - thing that would be however hardly generalizable, as explained.

Anyway, we need however to force nvcc to generate fatbin files with at least the minimum number of resources later necessary to modify the fatbin files, this because in the interpretation text files produced by cuobjdump a) there are not the ELF instructions necessary to declare the ELF registers, 6.6 - this also whether we need to write the PTX instructions, in the PTX codes, to declare the PTX registers that later will get some corresponding ELF registers - and b) there are not ELF or not ELF instructions assigning hardware registers to the ELF registers, 6.6. Not having way to know a) which are such - ELF or not ELF - instructions and b) where they are in the other A and C parts of the fatbin files different from the B parts that are composed by the consecutive ELF instructions generated by nvcc to execute the PTX codes - this because the real instruction set architecture is not disclosed and cuobjdump does not interpret such instructions that are not with all the others that it instead interprets in the fatbin file - we need always to force nvcc to generate fatbin files with at least the minimum number of resources later necessary to modify the fatbin files, this because, considered what said, we can not force nvcc to give us exactly the resources we want - for example the ELF registers $R2$ and $R10$ - or generate the procedure of assignment of the hardware registers to the ELF registers, but we can force nvcc to give us at least the minimum number and type of ELF registers we want and we can force nvcc to generate the procedure of assignment of the hardware registers to the ELF registers - also whether we do not know which it is - and later use the resources assigned by nvcc to the fatbin file - for example the ELF registers $R37$ and $R49$ instead of the ELF registers $R2$ and $R10$ that have not been assigned to the fatbin file - to modify the fatbin file to get the wanted ELF algorithmic implementation, wanted ELF algorithmic implementation that will not crash because a) to modify the fatbin file we have used only resources assigned to the fatbin file and b) there is no jump, in the part of the fatbin file interpreted by cuobjdump - the B part - to some other parts of the fatbin file different from the B part, and therefore independently of which can be the procedure of assignment of the hardware registers to the ELF registers, when the control is passed to the beginning of the part of the fatbin file interpreted by cuobjdump - the B part - the part is executed in its wholeness, using only the resources assigned by nvcc to the B part of the fatbin file.

We give a set of guidelines to force nvcc to generate fatbin files with at least the minimum number of resources later necessary to modify the fatbin files to get the wanted ELF algorithmic

implementations -6.3.1. Our set of guidelines is based on the assumptions that nvcc, when compiles a PTX file, $a_1$) tries to save as many registers as possible, $a_2$) does not remove 2 PTX synchronization barriers if there are some useful PTX instructions between the 2 PTX synchronization barriers and $a_3$) transform the PTX instructions between each couple of PTX synchronization barriers in ELF instructions that will be between the 2 ELF synchronization barriers that correspond to the couple of PTX synchronization barriers that contains the PTX instructions However, because we can not know whether the assumptions are really true in reality - this because the nvcc code is not open - we always check that this is in fact the case each time a fatbin file is generated, 6.3.2 - this is done checking the structure of the PTX file given in input to nvcc and the structure of the part of fatbin file that corresponds to the PTX instructions transformed in ELF instructions, in other words the part of the fatbin file that is interpreted by cuobjdump, the B part of the fatbin file.

In out situation the code of our compiler - nvcc - is not open, but also supposing the code of a generic compiler would be open, usually people do not spend time to study a compiler code to force the compiler to generate the wanted assembly algorithmic implementations a) because it is very time consuming and b) because it would be hard to generalize the results to other versions of the same or a different compiler in the same or a different environment - operative system, hardware architecture, drivers, etc. . The only other possible choice, in this case too, it would be therefore to force the compiler to generate an assembly file with the minimum number and type of resources later necessary to modify it to get the wanted assembly algorithmic implementation.

If the people writing compilers can give a simple function to generate assembly files with a given number and type of resources then that would be great, if not, users can simply implement our procedure for another machine, this because our procedure works for any version of any compiler - CPU or GPU - with or not open code, in any environment, but if the real instruction set architecture is not know - this means that we do not know the binary codes of the assembly instructions and which bits in the binary code of an assembly instruction represent what - then there has to be a tool - as cuobjdump - that returns an interpretation text file where we can at least read an human readable text form representation of the assembly instructions - for example $ADD.B32\ R34,\ R5,\ R17$.

We are not able to find in literature any paper that gives editing guidelines to write PTX codes in such way to force nvcc to generate fatbin files with at least a wished minimum number and type of resources. Considering the importance of this also in the more general case we hope people writing compilers consider the possibility of making available to users functions to generate assembly files with the wished minimum number and type of resources.

### 13.2.3 PTX-ELF Correspondences

The real instruction set architecture is not disclosed so to be able to modify the B parts of fatbin files to get the wanted ELF algorithmic implementations it is necessary to understand in which ELF instructions is transformed each single PTX instruction, in which order are such ELF instructions and which ELF registers in such ELF instructions correspond to which PTX registers in the single PTX instructions.

We use the set of guidelines - 6.3.1 - to generate a series of fatbin file - a fatbin file per single PTX instruction of interest. The same set of guidelines that allows us to generate fatbin files with at least a minimum number and type of resources allows us also to understand, analyzing the interpretation text file of a fatbin file generated giving in input to nvcc a PTX file edited following the editing guidelines, a) which ELF instruction/instructions are used to execute each single PTX

instruction, 6.3.2 - there are many single PTX instructions that are transformed in set of consecutive ELF instructions - b) which ELF register correspond to a PTX register of the PTX code used to generate the fatbin file - 6.3.3 - and c) whether there are some ELF registers, used in the ELF instruction/instructions used to execute each single PTX instruction, without corresponding PTX register in the single PTX instruction - 6.3.3.

Understanding these things is important because when we modify the B parts of the fatbin files - 6.7 - for each PTX instruction we want to transform in the ELF executed by the GPU, we need a) to associate the PTX registers - for example $\%reg_3$, $\%reg_4$ and $\%reg_5$ - used in the PTX instruction - for example a mul.s32 PTX instruction - in our PTX code, with the original PTX registers - for example $\%result$, $\%operand_1$ and $\%operand_2$ - used in the PTX instruction when we extracted the PTX instruction, b) to understand which ELF registers in our fatbin file generated for our PTX code correspond to the PTX registers used in the PTX instruction in our code - for example $R27$ for $\%reg_3$, $R34$ for $\%reg_4$ and R47 for $\%reg_5$ - c) to build the human readable text form representation/representations of the ELF instruction/instructions necessary to execute the PTX instruction, taking care to substitute the original ELF registers - for example $R23$, $R27$ and $R59$ - that corresponds to the original PTX registers - $\%result$, $\%operand_1$ and $\%operand_2$ - with the ELF registers of our fatbin file - $R27$, $R34$ and $R47$ - that correspond to the PTX registers of our PTX instruction - $\%reg_3$, $\%reg_4$ and $\%reg_5$ - d) to take care to use some of the ELF registers of our fatbin file if in the ELF instruction/instructions necessary to execute the PTX instruction there are some original ELF registers without corresponding original PTX register in the original PTX instruction, and e) finally retrieve the binary codes/codes of the human readable text form representation/representations, of the ELF instruction/instructions, so built.

Considering the complexity of the GPUs we can not exclude that the same PTX instruction, using the same PTX registers, in the same roles in the PTX instruction, can be transformed by nvcc in a different number and type of ELF instructions that depend on the following 5 things: a) the NVIDIA drivers and their versions, b) the version of nvcc, c) whether the code is compiled for 32 bits or 64 bits, d) which has to be the PTX version of the intermediate PTX files generated by nvcc for the generation of the output fatbin file, e) for which GPU architecture has to be produced the output fatbin file and f) the operative system running on the CPU and its version.

For this reasons, 1) our framework is able to find the PTX-ELF correspondences of any of the specific GPUs using the GF100 architecture - this for each one of the possible combinations of values of the previous 5 things - and 2) independently of which GPU architecture the Kepler GPUs use - equal or different from the GF100 architecture - the framework is able to find the PTX-ELF correspondences of any specific Kepler GPU - this is possible because cuobjdump is supported also for Kepler GPUs.

We are not able to find any paper in literature where are indicated the PTX-ELF correspondences but such correspondences are important to understand how we need to modify the B part of a fatbin file to get the wanted ELF algorithmic implementations, with the right ELF instructions necessary to execute each PTX instruction, with the right ELF registers in their different roles in each one of the ELF instructions and with the right dependences among ELF registers used in the ELF instructions - see 6.4 for further details.

### 13.2.4 Reverse Engineering of the Real Instruction Set Architecture

We are not able to find in literature any paper that reverse engineers the real instruction set architecture used by the GF100 architecture. We reverse engineer the real instruction set used by the GF100 architecture - 6.5 - because it is important a) to understand some details of the GF100 architecture and b) to be able to modify the B parts of the fatbin files to get the wanted ELF algorithmic implementations.

Having reverse engineered the real instruction set architecture used by the GF100 architecture and knowing the human readable text form representations of the ELF instructions we want the GPU executes, we can retrieve or generate the binary codes corresponding to the human readable text form representations of the ELF instructions that we want the GPU executes and overwrite the B parts of the fatbin files that we know correspond to the PTX codes given in input to nvcc.

Considering the complexity of the GPUs we can not exclude that the same ELF instruction, using the same ELF registers, in the same roles in the ELF instruction, can be transformed by nvcc in different binary codes that depend on the following 5 things: a) the NVIDIA drivers and their versions, b) the version of nvcc, c) whether the code is compiled for 32 bits or 64 bits, d) which has to be the PTX version of the intermediate PTX files generated by nvcc for the generation of the output fatbin file, e) for which GPU architecture has to be produced the output fatbin file and f) the operative system running on the CPU and its version.

For this reasons, as already previously said, 1) our framework is able to find the PTX-ELF correspondences of any of the specific GPUs using the GF100 architecture - this for each one of the possible combinations of values of the previous 5 things - and 2) independently of which GPU architecture the Kepler GPUs use - equal or different from the GF100 architecture - the framework is able to find the PTX-ELF correspondences of any specific Kepler GPU - this is possible because cuobjdump is supported also for Kepler GPUs.

We have found that the real instruction set architecture has not a fixed format. For each ELF instruction we now know which groups of bits in the 8 bytes that represent the binary code of the ELF instruction correspond to which of the ELF registers that appear in the human readable text form representation of the ELF instruction. For example, for the human readable text form representation of the ELF instruction $@P0\ SET.B32.GT\ P1,\ pt,\ R5,\ R37,\ pt$, we know which bits allow us to modify the normal predicate ELF registers $P0$ and $P1$ and the special predicate ELF register $pt$, and which bits allow us to modify the normal not predicate ELF registers $R5$ and $R37$. All the bits that do not modify any of the ELF registers that appear in the human readable text form representation of an ELF instruction are instead considered necessary to generate the ELF instruction.

### 13.2.5 Getting the Wanted ELF Algorithmic Implementations

If a) we can not write in the assembly executed by the machine or b) we can not force the compiler to generate the wanted assembly algorithmic implementation or c) we can not modify the assembly code produced by the compiler to get the wanted assembly algorithmic implementation, then all our efforts in editing code could be useless because usually the assembly code generated by a compiler is very different from the higher code edited by an user and this is usually true not only for GPUs but also for CPUs.

We are not able to find in literature any paper showing how to get the wanted ELF algorithmic implementations for GPUs using the GF100 architecture. To eliminate this lack of control, thanks

to the results of the previous phases, we devise a procedure to get any wanted ELF algorithmic implementation - 6.7 - to be able, also whether we can not directly write in ELF, of modifying the B part of any fatbin file, a) with the number and type of ELF instructions that we want, b) with the ELF registers, in each one of the ELF instructions, that we want, in the positions that we want, c) with the dependences among ELF registers that we want, d) with the guarantee that the B part of the fatbin file will not crash during its execution and e) that the execution of the B part of the fatbin file will be logically correct - this because we know the correspondences among PTX registers and ELF registers. For example, for a PTX instruction $@\%reg_0\ sub.s32\ \%res,\ \%fo,\ \%so$, the corresponding human readable text form representation of the ELF instruction necessary to execute it could be $SUB.B32\ R28,\ R6,\ R59,\ pt,\ @P4$, where 1) the PTX normal predicate register $\%reg_0$ appear at the end of the human readable text form representation of the ELF instruction as the normal predicate ELF register $P4$ and 2) there is the use of the special predicate ELF register $pt$ that does not appear anywhere in the PTX instruction.

The procedure gives us control on the ELF code that is executed by the GF100 architecture and we can get the same results for any different combination of values of the 5 things previously listed - and so a) the NVIDIA drivers and their versions, b) the version of nvcc, etc. - that could require different binary codes, also for the same ELF instruction, using the same ELF registers.

## 13.3 Not Disclosed GPU Behaviors

We foreseen that there are some not disclosed GPU behaviors able to slow down the execution of the B part of a fatbin file. Discovering, understanding and quantifying these not disclosed GPU behaviors is important to understand how to analyze the B part of a fatbin file and to understand how to modify it - if necessary - to optimize its execution time.

To discover, understand and quantify the not disclosed GPU behaviors, we generate several fatbin files, a fatbin file per couple ( instruction configuration , dependence distance ), 7.4 - we do this using the results described in the previous section. The fatbin files are at our knowledge the only fatbin files - used to get this goal - with their B parts built having a complete control on the ELF of the GF100 architecture - this is important to validate the results that we get for the discovering, the understanding and the quantification of the not disclosed GPU behaviors. In all the papers we found in literature instead, also whether all the results are obtained - for a similar or different goal - executing fatbin files produced by nvcc, there is no sign of which is the ELF executed by the GF100 architecture and whether the B parts of the fatbin files are equal or different to the codes - CUDA or PTX - given in input to nvcc.

About the for loop in the B parts of the fatbin files a) to understand why the executions of the for loop in the B parts of the fatbin files generated for the discovery, understanding and quantification of the not disclosed GPU behaviors can not be slowed down by the bandwidths and the latencies of the GPU memories see 7.4.1, b) to understand the structure of the for loop in the B parts of the fatbin files see 7.4.2 and c) to understand the procedure that we have used to generate the B parts of the fatbin files see 7.4.3 - to instead better understand the procedures used for the discovery, the understanding and the quantification of the not disclosed GPU behaviors see 7.6.

### 13.3.1 Advancement of the Resident Warps in a Streaming Multiprocessor

Studying the advancement of the resident warps in a streaming multiprocessor, during the execution of the for loops of the B parts of the fatbin files generated for the discovery, understanding and quantification of the not disclosed GPU behaviors, we have strong evidence that the warps are moved forward all together by the warp schedulers in a streaming multiprocessor. We think this is the case also for the execution of the B part of any fatbin file - equal or different from the fatbin files used - this at least in the cases when the execution of the B part of the fatbin file can not be slowed down by the bandwidths and the latencies of the GPU memories - see 9 for the several supporting reasons.

Understanding this is important because the warp scheduling policies executed by the warp schedulers in the streaming multiprocessors are not disclosed but we need to understand how the resident warps in a streaming multiprocessor are made to advance because, for example, this allows us to correctly calculate some lower bounds on the minimum quantity of time that it is necessary for a warp to move from the execution of a warp ELF instruction $x$ to a warp ELF instruction $y$ when there are $w$ warp resident in the streaming multiprocessor where the warp is resident.

### 13.3.2 Even Distribution of the GPU Thread Blocks

We are not able to find in literature any paper that accurately studies the distribution of the GPU thread blocks to the streaming multiprocessors of the GF100 architecture. If the gigathread scheduler is not evenly distributing the GPU thread blocks to the streaming multiprocessors then all our efforts to optimize the B part of a fatbin file could be made useless by a not even distribution of the GPU thread blocks to the streaming multiprocessors - think for example to the load unbalancing that could be created.

Thanks to our study we discover that the distributions of the GPU thread blocks to the streaming multiprocessors depend on a) the number of ELF registers of a fatbin file and b) the launch configuration used to execute the fatbin file - a fatbin file can be usually executed using more than only one launch configuration - 2.5. Because the number of launch configurations that could be used to execute a fatbin file is huge - 2.5 - we study a subset of all the possibilities - this because not more than 48 warps can be resident in each moment, during the execution of the B part of a fatbin file, in a streaming multiprocessor. The results of the study are the following:

- $R_1$) If the number of GPU thread blocks that we want assigned to each one of the streaming multiprocessors times the number of warps per GPU thread block times the number of ELF registers of the fatbin file is smaller than half of the number of hardware registers of a streaming multiprocessor then the gigathread scheduler is never evenly distributing the GPU thread blocks to the streaming multiprocessors;

- $R_2$) If the number of GPU thread blocks that we want assigned to each one of the streaming multiprocessors times the number of warps per GPU thread block times the number of ELF registers of the fatbin file is greater than half of the number of hardware registers of a streaming multiprocessor and smaller or equal than the number of hardware registers of a streaming multiprocessor then the gigathread scheduler is always evenly distributing the GPU thread blocks to the streaming multiprocessors.

Beyond the fact that only 48 warps can be resident at each moment, during the execution of the B part of a fabin file, in a streaming multiprocessor, another reason to use a subset of all the possible launch configurations it is that we want to avoid, during the execution of the B part of the fatbin file, the overheads due to the assignment of the GPU thread blocks to the streaming multiprocessors. If the number of GPU thread blocks we want per streaming multiprocessor - smaller than 48 - times the number of warps per GPU thread block times the number of ELF registers of the fatbin file is greater than half of the number of hardware registers of a streaming multiprocessor and smaller or equal than the number of hardware registers of a streaming multiprocessor then, for any launch configuration a) satisfying the previous requirement and b) implying, supposing an even distribution of the GPU thread blocks, a number of GPU thread blocks per streaming multiprocessor smaller than 48, we get the guarantee that the gigathread scheduler is always going to evenly distribute the GPU thread blocks to the streaming multiprocessors and that it will do this only one time during the execution of the B part of the fatbin file.

### 13.3.3 Warp Scheduling Load Unbalancing

We are not able to find in literature any paper that accurately studies the warp scheduling at local level - and so in the streaming multiprocessors - where the authors take care to create in ELF some specific B parts for the fatbin files used in the studies.

Timing the execution of the many different for loops of the many different fatbin files we generated for the discovery, the understanding and the quantification of the not disclosed GPU behaviors, we find that if the number of resident warps in a streaming multiprocessor is odd then some phenomenons of warp scheduling load unbalancing are present and so to execute a fatbin file it is better to avoid the use of launch configurations that, also whether imply an even distribution of the GPU thread blocks to the streaming multiprocessors, force an odd number of resident warps in each streaming multiprocessor.

The number of resident warps in a streaming multiprocessors is not however the only thing that can create warp scheduling load unbalancing in a streaming multiprocessor, another are the dependence distances between the ELF registers used in the ELF instructions of the B part of a fatbin file - 8.3 thanks to the results in 7.6.2.

### 13.3.4 Local Streaming Multiprocessor PTX and ELF Architectural Features

The discovery, understanding and quantification of the local streaming multiprocessor PTX and ELF architecture features it is important to understand how to optimize and how to analyze the B part of a fatbin file.

The local streaming multiprocessor PTX and ELF architectural features are: a) the real instruction configurations streaming multiprocessor best average performance per clock cycle of the PTX and ELF instruction configurations, b) the scheduling waiting times of the ELF instruction configurations, c) the dependence waiting times of the ELF instruction configurations, d) the overhead time for the management of the warps and e) the minimum number of resident warps necessary in a streaming multiprocessor to get the real instruction configuration streaming multiprocessor best average performance per clock cycle of each ELF instruction configuration for each dependence distance.

*Real Instruction Configuration Streaming Multiprocessor Best Average Performance per Clock Cycle:* We are not able to find in literature any paper that studies the real instruction configuration streaming multiprocessor best average performance per clock cycle, of the PTX and ELF instruction configurations of interest, with our level of accuracy. We not only determine the real PTX and ELF instruction configuration streaming multiprocessor best average performance per clock cycle of each PTX and ELF instruction configuration of interest - 7.6.2 - but we also discover the presence of some not disclosed hardware resources shared among the 2 groups of 16 CUDA cores in each streaming multiprocessor, not disclosed shared hardware resources that make impossible for some PTX and some ELF instruction configurations to have a real PTX or ELF instruction configuration streaming multiprocessor best average performance per clock cycle equal to the theoretical streaming multiprocessor best average performance per clock cycle - 4.3. Knowing the real PTX and ELF instructions streaming multiprocessor best average performance per clock cycle allows the real ELF code efficiency of the B part of some fatbin files to be better calculated;

*Scheduling Waiting Times:* We are not able to find any paper in literature that accurately studies the warp scheduling waiting times. We foreseen that the warp scheduling waiting time could be different for different ELF instruction configurations. Verified that the warp scheduling waiting time is different for different ELF instruction configurations, we quantify the scheduling waiting time of each ELF instruction configuration of interest - 7.6.2.

Knowing the scheduling waiting times is important because, considering the B part of a fatbin file, the scheduling waiting times of the ELF instructions in the B part of the fatbin file, supposing the execution of the B part of the fatbin file can not be slowed down by the bandwidths and the latencies of the GPU memories, allows us to determine the minimum number of resident warps, that are necessary in each streaming multiprocessor, during the execution of the B part of the fatbin file, to avoid pipeline stalls due to the scheduling waiting times.

*Dependence Waiting Times:* We are not able to find any paper in literature that accurately studies the dependence waiting times. We foreseen that the dependence waiting time could be different for different ELF instruction configurations. Verified that the dependence waiting time is different for different ELF instruction configurations, we quantify the dependence waiting times of each ELF instruction configuration of interest - 7.6.2.

Knowing the dependence waiting times is important because, considering the B part of a fatbin file, the dependence waiting times of the ELF instructions in the B part of the fatbin file, supposing the execution of the B part of the fatbin file can not be slowed down by the bandwidths and the latencies of the GPU memories, allows us to determine the minimum number of resident warps, that are necessary in each streaming multiprocessor, during the execution of the B part of the fatbin file, to avoid pipeline stalls due to the dependence waiting times.

*Overhead Time for the Management of the Warps:* We are not able to find any paper in literature that accurately studies the overhead time for the management of the warps. We foreseen its existence and that it is possible that it is not linearly increasing with a linear increase of the number of resident warps in a streaming multiprocessor. Verified its existence and its not linear increase at a linear increase of the number of resident warps in a streaming multiprocessor, we study the overhead time for the management of the warps and take in account its effects for the different triplets ( ELF instruction configuration , dependence distance , number of resident warps in a streaming multiprocessor ) - 7.6.2.

Knowing the overhead time for the management of the warps is important because, considering the B part of a fatbin file, the overhead time for the management of the warps, supposing the

execution of the B part of the fatbin file can not be slowed down by the bandwidths and the latencies of the GPU memories, allows us to determine the minimum number of resident warps that are necessary in each streaming multiprocessor, during the execution of the B part of the fatbin file, to avoid pipeline stalls due to the overhead time for the management of the warps.

*Minimum Number of Resident Warps Necessary in a Streaming Multiprocessor to get the Real Instruction Configuration Streaming Multiprocessor Best Average Performance per Clock Cycle of Each ELF Instruction Configuration for Each Dependence Distance:* Considering concurrently the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps, we determine the minimum number of resident warps necessary in a streaming multiprocessor to get the real instruction configuration streaming multiprocessor best average performance per clock cycle of each ELF instruction configuration for each dependence distance - 7.6.2.

Knowing the the minimum number of resident warps necessary in a streaming multiprocessor to get the real instruction configuration streaming multiprocessor best average performance per clock cycle of each ELF instruction for each dependence distance is important because, considering the B part of a fatbin file, the minimum number of resident warps necessary in a streaming multiprocessor to get the real instruction configuration streaming multiprocessor best average performance per clock cycle of each ELF instruction for each dependence distance, supposing the execution of the B part of the fatbin file can not be slowed down by the bandwidths and the latencies of the GPU memories, allows us to determine the minimum number of resident warps necessary in each streaming multiprocessor, during the execution of the B part of the fatbin file, to avoid pipeline stalls due to the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps.

## 13.4 Transformations and Launch Configurations

While thanks to the results in 13.2 we are able to get the wanted ELF algorithmic implementations and so being sure that nvcc can not ruin our efforts, thanks to the results got in 13.3 we understand how to modify the B part of any fatbin file to optimize its execution time. The process of optimization start with a) the transformation of the original fatbin file that we want to optimize - the transformation implies 1) the possible modification of the B part of the original fatbin file that we want to optimize and 2) the generation of a set of fatbin files with their B parts equivalent to the B part of the original fatbin file, see 8.4 - and b) the generation of a set of launch configurations for each one of the equivalent fatbin files generated - the launch configurations, in the set of launch configurations generated for an equivalent fatbin file, 1) are only some of the launch configurations that could be used to execute the fatbin file and 2) will be the only launch configurations that can be considered during the analysis/analyses of the B part of the fatbin file, see 8.3.

We are not able to find any paper in literature where some modifications are applied - at ELF level - to the B part of a fatbin file that has to be executed by GPUs using the GF100 architecture. We instead are able to do this and we do it because generating many fatbin files with their B parts equivalent to the B part of the original fatbin file, we increase the probability that we can use at least a launch configuration, to execute at least one of the equivalent fatbin files, without problems of load unbalancing - this is far from banal, in fact we are not able to find in literature any paper that a) studies accurately the process of distribution of the GPU thread blocks to the streaming multiprocessors - 7.6.1 - b) discovers that there is warp scheduling load unbalancing in the case

there is an odd number of resident warps in a streaming multiprocessor - 7.6.2 - and c) discovers that there can be warp scheduling load unbalancing whether, when the fatbin file is executed using a launch configuration, the B part of a fatbin file, together at the launch configuration used to execute the fatbin file, generates some specific couples ( dependence distance , number of resident warps in a streaming multiprocessor ) - 8.3 thanks to the results in 7.6.2.

### 13.4.1 Transformation of the Original Fatbin File to Be Optimized

After the possible modification of an original fatbin file to be optimized we generate many equivalent copies of the original fatbin file, each one a) with a different number of ELF registers and b) with a different logically correct order of the ELF instructions of the B part of the original fatbin file - see 8.4 for the whole procedure of transformation of the original fatbin file.

This procedure allows us, having many equivalent copies of the original fatbin file with only a) a different number of ELF registers and b) a different logically correct order of the ELF instructions of the B part of the original fatbin file, to consider for the analysis/analyses many more launch configurations of those that we could consider for the analysis/analyses of only the B part of the original fatbin file - each launch configuration used to analyze a fatbin file has to satisfy specific requirements, see next subsection.

### 13.4.2 Selection of the Launch Configurations

The launch configurations in each set of launch configurations of each fatbin file are generated in such a way that, thanks to the results on the discovery, understanding and quantification of the not disclosed GPU behaviors, we have an *a priori* guarantee that each time an equivalent fatbin file is executed using one of the launch configurations in the set of launch configurations generated for it, a) the gigathread scheduler evenly distributes the GPU thread blocks to the streaming multiprocessors, b) the number of resident warps in each streaming multiprocessor is even and c) phenomenons of warp scheduling load unbalancing, due to the presence of some dependence distances in the B part of the fatbin file that are bad - see 8.3 - for the number of resident warps, in each streaming multiprocessor, implied by the launch configuration used to execute the fatbin file, are absent - this last thing is due to the fact that to generate the set of launch configurations for the equivalent fatbin file, all the couples ( dependence distance , number of resident warps in a streaming multiprocessor ), generated by the combination ( B part of the equivalent fatbin file generated , potential launch configuration in the set of launch configurations generated for the equivalent fatbin file ), are considered.

## 13.5 Analysis of the Equivalent Fatbin Files Generated

To be able to complete the optimization process of the B part of an original fatbin file we need to analyze the couples ( equivalent fatbin file generated , launch configuration in the set of launch configurations generated for the equivalent fatbin file ).

To be able to accurately analyze the couples, we highlight the importance of differentiating fatbin files and therefore create a taxonomy per fatbin files that allows us to classify them - 10. Next, considering the position of the fatbin files in the taxonomy, we explain the analysis/analyses that are executable on the fatbin files - 11. Finally, if a fatbin file is in a particular position in the taxonomy then we explain why we are able to execute on it a theoretical analysis that we have

devised. If the theoretical analysis is executed on an equivalent fatbin file and if at least one of the couples for it generated satisfies all the requirements of the theoretical analysis then we give an *a priori* ELF code shortest execution time guarantee for the execution of the B part of the equivalent fatbin file when the equivalent fatbin file is executed using the launch configuration of the couple - 12.

### 13.5.1 Taxonomy for Fatbin Files

To differentiate fatbin files, and so to understand which analysis/analyses are executable on them, we create a taxonomy for fatbin files. We are not able to find any paper in literature where the fatbin files are classified considering - as instead we do for our taxonomy - 1) which warp scheduling policy the reader believes is executed by the warp schedulers in the streaming multiprocessors when the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, 2) the presence of branches in the B part of the fatbin file, 3) which, the read believes, are the eviction policies used for the l2 cache and the l1 caches, 4) the possibility to know *a priori*, before the execution of the fatbin file, which are the positions, in the arrays, the vectors and the structures, in the GPU global memory, of the data/results that will be read/written by each GPU thread used to execute the B part of the fatbin file, during the execution of the B part of the fatbin file and 5) the presence of ELF instructions of synchronization in the B part of the fatbin file.

The classification of the fatbin file using the taxonomy is important because considering how the fatbin files are classified using the taxonomy, the fatbin files are eligible for the execution of different types of analyses - empirical and/or theoretical - thing instead usually not considered in the papers in literature where the analysis/analyses a) is/are empirical studies, b) is/are executed running some microkernels and c) does/do not consider the specific structure/structures of the B parts of fatbin files on which the microkernels could be embedded by users.

### 13.5.2 Analysis/Analyses Selection

Considering the positions of the fatbin files in the taxonomy we explain because some analyses are possible instead of others. For the fatbin files in a given position in the taxonomy we explain the problems given from the fact that it is impossible to know the locations, in the GPU global memory, where the ELF instructions, of the B part of any fatbin file, read/write data/results, this also if it could be possible to determine a) which ELF instructions in the B part of a fatbin file imply some byte transfers and b) in which positions in the arrays, the vectors and the structures that contains the data in input and that will contain the output results, the GPU threads, used to execute the B part of a fatbin file, are going to read/write the data/results, during the execution of the B part of the fatbin file.

### 13.5.3 Guaranteeing *A Priori* ELF Code Shortest Execution Times

At the best of our knowledge we are the only one to theoretically study the B part of a fatbin file a) considering its features - in other words its position in a taxonomy - and b) creating several unrolled path versions of its B part that are analyzed considering the inputs that generate them.

Each couple ( fatbin file generated , launch configuration in the set of launch configurations of the fatbin file generated ) is substituted with a set of quadruplets - see 12.2.7 for a greater quantity

of details about the process of generation and the reasons because it is necessary.

For each quadruplet satisfying all the requirements of the theoretical analysis we can give an *a priori* ELF code shortest execution time guarantee - see 12.4 for a brief explanation of the consequences of this. For the other results that we proved to be able to execute the theoretical analysis on the B part of a fatbin file we invite the reader to read 12.4 while for an explanation and the verification of the correctness of the theoretical analysis we invite the reader to check 12.

## 13.6 Summary

In this chapter we have reviewed the main contributions of the thesis. The users can only use CUDA or PTX to edit code but it is very easy that nvcc can completely ruin all the efforts done at CUDA or PTX level to optimize a code. Also whether a) the real instruction set architecture used by the G100 architecture is not disclosed, b) the nvcc code is not open and c) it is not possible to edit code in the assembly - the ELF - executed by the GPUs using the GF100 architecture, thanks to the results in this thesis, now users a) can get the wanted ELF algorithmic implementation and b) can optimize the execution time of ELF code of fatbin files with a much greater degree of accuracy compared to what it was possible before - this is possible not only because we give to users the tools to get the wanted ELF algorithmic implementations but also because 1) we discover, understand and quantify some not disclosed GPU behaviors that could slow down the execution of the B part of a fatbin file and 2) we devise an optimization process that, considering what we have discovered, understood and quantified about the not disclosed GPU behaviors, transforms the original fatbin file, classifies the equivalent fatbin files generated and executes different types of analyses, empirical or theoretical, on the equivalent fatbin files generated.

Thanks to the results in 13.2 - real ISA and ELF codes - 1) we are able to localize in a fatbin file the ELF instructions that correspond to the PTX code given in input to nvcc for the generation of the fatbin file, 2) we give a set of guidelines to force nvcc to generate fatbin files with at least the minimum number and type of resources later necessary to modify the B part of the fatbin file to get the wanted ELF algorithmic implementations, 3) we discover the PTX-ELF correspondences and so for each single PTX instruction we know the number, type and order of ELF instructions necessary to execute the PTX instruction and which ELF registers, in the ELF instructions necessary to execute the PTX instruction, correspond to which PTX registers in the PTX instruction, 5) we reverse engineer the real instruction set architecture and 6) we are able to get any wanted ELF algorithmic implementations - we can get all these results not only for any GPU using a GF100 architecture but also for any NVIDIA GPU using a later architecture and so, for example, Kepler GPUs.

Thanks to the results in 13.3 - not disclosed GPU behaviors - 1) we know how the resident warps in a streaming multiprocessor are made to advance , this at least in the case the execution of the B part of a fatbin file is not slowed down by the bandwidths and the latencies of the GPU memories, 2) we know how to force the gigathread scheduler to always evenly distribute the GPU thread blocks to the streaming multiprocessors, 3) we know how to avoid warp scheduling load unbalancing in the streaming multiprocessors, 4) we discover, understand and quantify the local streaming multiprocessor PTX and ELF architectural features and so a) the real instruction configuration streaming multiprocessor best average performance per clock cycle of the PTX and ELF instruction configurations, b) the scheduling waiting times of the ELF instruction configurations, c) the dependence waiting times of the ELF instruction configurations, d) the overhead time for

the management of the warps and e) the minimum number of resident warps necessary in a streaming multiprocessor to get the real instruction configuration streaming multiprocessor best average performance per clock cycle of each ELF instruction configuration for each dependence distance.

Thanks to the results in 13.4 - transformations and launch configurations - 1) we know how to transform the B part of an original fatbin file that we want to optimize and 2) we know how to generate the sets of launch configurations, a set of launch configurations for each one of the equivalent fatbin files generated from the original fatbin file - the set of launch configurations of each equivalent fatbin file is used for the analysis/analyses.

Thanks to the results in 13.5 - analysis of the equivalent fatbin files generated - 1) we classify the equivalent fatbin files, generated from the original fatbin file, using a taxonomy, 2) considering the positions, in the taxonomy, of the equivalent fatbin files generated, we determine the analysis/analyses that can be executed on their B parts and 3) we give *a priori* ELF code shortest execution time guarantees if the equivalent fatbin files generated from the original fatbin file are eligible for the execution of the theoretical analysis and one or more of them satisfy all the requirements of the theoretical analysis.

In the next chapter we review the previous work and considering the contributions of the thesis we highlight the problems that afflict all the results of all the papers that we were able to find.

# Chapter 14

# Previous Work and its Problems

## 14.1 Introduction

In the the previous chapter we have explained a) how with this thesis we have solved several challenges that nobody - at the best of our knowledge - had solved or addressed in papers in literature, and b) because in our opinion it is important that we have addressed and solved these challenges. In the this chapter we review the previous work and considering the contributions of the thesis we highlight the problems that afflict all the results of all the papers that we were able to find.

## 14.2 Previous Work

The papers about NVIDIA GPUs can be subdivided in several categories and a paper can be at the same time in more categories. For these reasons here we present only one of the possible way of classification, this to explain the evolution of the state of the art for different topics.

In the auto-tuning category we have tools to transform in an automatic way C codes in CUDA codes [42, 2010], more specific auto-tuning tools to optimize dense matrix multiplications for GPGPU (General Purpose GPUs) with caches [78, 2010], model-driven auto-tuning tools for the sparse matrix-vector multiplication [35, 2010], tools to auto-tune CUDA parameters for the sparse vector multiplication [23, 2010], automatic tools for the generation of BLAS - basic linear algebra subprograms - libraries [24, 2011], auto-tuning tools for GEMM - general matrix multiplication - kernels, specifically for Fermi GPUs - and therefore for GPUs using the GF100 architecture - [32, 2012], auto-tuning tools for dense vector and matrix-vector operations for Fermi GPUs [70, 2012] and auto-tuning tools for the sparse matrix vector product based on the ELLR-T approach [17, 2012].

In the matrix-multiplication category we have, for GPUs using an architecture different from the GF100 architecture, general studies as [43, 2008] and [6, 2009] that optimize the sparse matrix-vector multiplication, while studies that also consider GPUs using the GF100 architecture are [58, 2010] that improves the Magma GEMM , [59, 2011] that optimizes the symmetric dense matrix vector multiplication and [75, 2012] that optimizes the sparse matrix-vector multiplication using cache blocking methods.

In the CUDA optimization category we have papers that study optimization principles and evaluate performance [65, 2008], papers that explain how to reduce the GPU programming complexity

[67, 2008], papers that propose mapping paths for multi-GPGPU accelerated computers starting from portable high level programming abstractions [4, 2010] and papers that explain the impact of the CUDA tuning techniques for Fermi GPUs [82, 2011]. Papers that consider more specific optimization techniques are instead for example papers that propose control-structures to optimize GPGPU codes [62, 2009] and papers that propose on-the fly elimination of dynamic irregularities for GPU computing [16, 2011].

In the framework category where a framework can be used to translate, optimize and/or analyze GPU code, we have papers that propose frameworks for an efficient implementation of CUDA kernels on multi-cores, papers that propose compiler frameworks for the optimization of affine loop nests [44, 2008], papers that propose compiler frameworks for the automatic translation and optimization of OpenMP code to GPGPU code [64, 2009], papers that propose cross-input adaptive frameworks for the optimization of GPU code [79, 2009], papers that propose to optimize compilers for GPGPU with input-data sharing [81, 2010], papers that propose dynamic optimization frameworks - as for example Ocelot - for bulk-synchronous applications in heterogeneous systems [18, 2010], papers that propose frameworks able to predict the GPU performance considering CPU code skeletons that are translated by the frameworks [34, 2011], papers that propose frameworks to dynamically instrument - within Ocelot - GPU applications [48, 2011], papers that propose frameworks to port shared memory GPU applications to multi-GPUs [11, 2012] and papers that propose different optimization strategies using llCoMP [61, 2012].

In the performance model category we have papers that propose performance prediction models for CUDA GPGPU platforms [36, 2009], papers that propose performance modeling and automatic ghost zone optimizations for iterative stencil loops [45, 2009], papers that propose adaptive performance modeling tools [66, 2010], theses that propose performance prediction using parametrized models [60, 2011] and papers that propose GPU performance models for effective control flow divergence optimizations [83, 2012].

In the sorting category we have papers that consider the use of GPUs to manage large databases [21, 2006], papers that implement adaptive bitonic sorting on GPUs [22, 2006], papers using hybrid algorithms on GPUs [69, 2008], papers that explain the design of efficient sorting algorithms for GPUs [68, 2009], papers that explain the design and the implementation in CUDA of algorithms to sort integers [37, 2011] and papers that implement and study the performance of radix sort algorithms [46, 2011].

In the design and evaluation category beyond some of the papers in the sorting category we have papers that consider the design and implementation of visual computing algorithms [29, 2009] and papers that consider the design and performance evaluation of image processing algorithms [28, 2011], while for more specific types of analysis and characterizations we have papers that consider PTX kernels [2, 2009].

In the modeling category we have papers that devise analytical models for GPU architectures with memory-level and thread-level parallelism awareness [27, 2009] and papers that instead model CPU-GPU workloads and systems [3, 2010], in the simulator category we have papers that analyze CUDA workloads [1, 2009] and papers that propose modular function GPU simulators like Barra [63, 2009] while in the API category we have papers that propose to optimize the memory bandwidths of GPUs via warp specialization [40, 2011].

Finally in the various fields category we have papers that consider the high performance computation and iterative display of molecular orbitals [71, 2009], papers that consider the optimization of data intensive computations for DNA sequence alignment [72, 2009], papers that consider the

heap based k-nearest neighbor search on GPUs [5, 2010], papers that solve on GPUs lattice QCD systems of equations using mixed precision solvers [13, 2010], papers that use GPUs to execute multi-spin monte carlo simulations of the 2D ising model [8, 2010], papers that use GPUs to simulate fluid flows in complex geometries using lattice Boltzmann codes [7, 2010], papers that consider the use of GPUs for the direct aperture optimization for online adaptive radiation therapy [12, 2010], papers that implements CUDA algorithms for cone beam reconstruction [80, 2010], papers that describe optimization strategies and study the performance of lattice Boltzmann CUDA kernels [31, 2011], papers that evaluate the optimizations applied to parallel particle swarm algorithms [38, 2011], papers that implements molecular dynamics on hybrid high performance computers [10, 2011], papers that consider molecular dynamics simulations of the relaxation processes in the condensed matter [47, 2011], papers that consider the performance potential for simulating spin models on GPUs [77, 2012], papers that consider CUDA optimization strategies for compute - and memory - bound neuroimaging algorithms [14, 2012], papers that consider the optimization of linked list prefix computations on multithreaded GPUs using CUDA [76, 2012], papers that consider the haralick's texture features computation accelerated by GPUs for biological applications [41, 2012], papers that consider co-evolutionary differential evolution algorithm for solving min-max optimization problems implemented on GPUs using CUDA [39, 2012], papers that consider the use of GPUs to solve knapsack problems [73, 2012], papers that study the q-state plotts using monte carlo algorithms implemented and optimized in CUDA [15, 2012] and papers that enhance data parallelism for ant colony algorithms [33, 2013].

Three papers, among all those that we have been able to find, consider for their studies - executed on GPU architectures pre-GF100 - the real assembly produced by nvcc, instead of CUDA or PTX code:

- In 2008, V. Volkov and J. W. Demmel presented in [74] the performance results for dense linear algebra for the NVIDIA GeForces GTX280, 9800GTX, 8800GTX and 8600GTS. For their studies they use Decuda to inspect the binaries produced by nvcc and as they report *Decuda is a third-party disassembler of GPU binaries based on the reverse engineering of the real instruction set architecture used by the GPU architectures studied in the paper*;

- In 2010 H. Wong, M. M. Papadopoulou, M. S. Alvandi and A. Moshovos in [25] demystified part of the NVIDIA GT200 (GTX280) GPU microarchitecture trough microbenchmarking. In their paper they too use Decuda to inspect the binary code generated by nvcc for the CUDA kernels that they implemented and used;

- In 2011 Y. Zhang and J. D. Owens in [84] describe a quantitative performance analysis model for the NVIDIA GT200 GeForce 200-series GPUs. In their paper with the assistance of Decuda they *buid a tool to modify the original binary instructions, assemble the modified instructions back to the binary code sequence, and finally embed the modified code into the execution file.* This is an improvement compared to the previous papers because this means that they got control on the assembly executed by the GPU architecture.

All the three previous papers however consider GPU architectures pre-GF100 - it is not possible to use Decuda to modify the binary code that has to be executed by GPUs using the GF100 architecture.

The authors - G. Tany, L. Liy, S. Triechlez, E. Phillipsz, Y. Baoy and N. Suny - of another paper - [19, 2011] - instead implement DGEMM specifically for Fermi GPUs and study what happens for

a Tesla C2050 GPU - Tesla C2050 GPUs use the GF100 architecture as Fermi GPUs.

At page 8 of their paper they report that *the proposed optimization strategies are involved with the exact selection and scheduling of instructions and therefore they cannot be achieved at the level of either CUDA C or PTX language because the programs of CUDA C/PTX are transformed to the native machine instructions by nvcc.* They next claim that *with NVIDIA's internal tool-chain, they implemented Algorithm 3 using Fermi's native machine language on NVIDIA Tesla C2050.* We would like to understand how they were able to get this result because we are not able to find in the paper any further explanation.

We know that there is AsFermi - http://code.google.com/p/asfermi/ - an assembler for the NVIDIA Fermi ISA but a) the last update to the AsFermi project was done during January 2012, b) the set of instructions reversed engineered - http://code.google.com/p/asfermi/wiki/Instructions - is much smaller than our where we also consider the size and the type of the operands and of the results of the PTX instructions that are transformed, c) we can not find any explanation about the PTX-ELF correspondences that we have found and that are necessary to produce correct ELF code and d) we know that there are some known issues with AsFermi - http://code.google.com/p/asfermi/wiki/KnownIssues.

Because we can not read in [19] anything about AsFermi then we are curious to know how the authors of that paper were able to do what they say - Sean Triechlez and Everett Phillips were working for NVIDIA at that time but also supposing they knew and did what it was necessary, the method that they used, to get the wanted ELF algorithmic implementations to execute on the Tesla C2050, has not been disclosed at the best of our knowledge.

## 14.3 Problems with the Previous Work

All the results of all the papers that we have been able to find are afflicted by one or more problems. We highlight here the most important of these problems, problems that show how the results got in the papers could be not correct - the results could be due to causes different from those thought by the authors of the papers - or not generalizable - change also only the version of the NVIDIA drivers and the results could be completely different. Here therefore the description of the most important problems afflicting the papers that we have been able to find:

- The results are got considering CUDA or PTX codes. The results in this case could be incorrect because the authors of the papers consider that the effects - the results - that they got, are due to the features that they implemented at CUDA or PTX level.

  The ELF code produced in output by nvcc is usually very different from the input CUDA or PTX code and therefore the features implemented at CUDA or PTX level could be transformed in something of completely different by nvcc during the generation of the output fatbin files corresponding to the input CUDA or PTX codes and so being absent or almost absent at ELF level.

  If this is the case then a) the results could be useless because the cause-effect principle behind the explanation of the results could be wrong - this because the causes of the effects, and so of the results, could be different from the features implemented at the CUDA or PTX level - or b) supposing the results correct, the results are not generalizable because the same CUDA or PTX code could be transformed in a completely different way at the change of also only one of the following 5 things: a) the NVIDIA drivers and their versions, b) the version of nvcc,

c) whether the code is compiled for 32 bits or 64 bits, d) which has to be the PTX version of the intermediate PTX files generated by nvcc for the generation of the output fatbin file, e) for which GPU architecture has to be produced the output fatbin file and f) the operative system running on the CPU and its version;

- The not disclosed GPU behaviors that we have discovered, understood and quantified are not considered. This is reasonable because to discover, to correctly understand and to accurately quantify the not disclosed GPU behaviors it is necessary to be able to produce the wanted ELF algorithmic implementations but this was not possible for the GF100 architecture before of the procedures we devised to get such goal;

  If the not disclosed GPU behaviors are not discovered, not correctly understood or not accurately quantified then it is not possible to get a reliable, correct and accurate model of the GPU architecture on which the fatbin files are executed, thing instead necessary to understand how to optimize the B part of a fatbin file and therefore a) to transform it, b) to select for it a set of launch configurations, c) to classify it and d) to understand how to develop the empirical and theoretical analysis/analyses to execute on it;

- The warp scheduling policies executed by the warp schedulers in the streaming multiprocessors are not studied. Studying how the warps are moved forward by the warp schedulers in the streaming multiprocessors is important for the whole process of optimization and to avoid phenomenon of warp scheduling load unbalancing during the execution of the B part of a fatbin file.

  A phenomenon of warp scheduling load unbalancing if not recognized and avoided could make to interpret in the wrong way the results - the warp scheduling load unbalancing could be attributed to the wrong cause or set of causes that instead does/do not have any connection with the generation of the phenomenon of warp scheduling load unbalancing;

- The distributions of the GPU thread blocks to the streaming multiprocessors is not studied. Understanding how the gigathread scheduler is going to distribute the GPU thread blocks to the streaming multiprocessors for the execution of the B part of a fatbin file is important to avoid phenomenons of workload unbalancing.

  If the phenomenon of workload unbalancing is not recognized and understood then it is possible to attribute the wrong causes to the poor efficiency that the B part of a fatbin file is getting while if the phenomenon of workload unbalancing is recognized and understood then it is possible to force the gigathread scheduler to evenly distribute the GPU thread blocks to the streaming multiprocessors;

- The minimum number of resident warps that is necessary in a streaming multiprocessor to avoid pipeline stalls, due to the warp scheduling times, the dependence waiting times and the overhead time for the management of the warps, during the execution of the B part of a fatbin file is not studied with our level of accuracy or is not proved or determined and so it is not possible to understand whether the efficiency of the ELF code is good because its only slowdowns are due to the warp scheduling or instead the efficiency of the ELF code is not good because its slowdowns are due to causes a) also different from the warp scheduling and b) that could be however avoided;

The fact that previously at this thesis was not possible to get the wanted ELF algorithmic implementations made difficult to accurately determine it for the B part of a fatbin file but now we can determine it not only for the B part of a fatbin file but also for each dependence distance of each ELF instruction.

- The analysis/analyses are executed as empirical studies where some microkernels are run without considering the structure of the B parts of fatbin files on which an user could embed the microkernels, thing instead very important to consider to prove that the results got for the microkernels are generalizable to other cases.

  It is very easy, for example, to embed a microkernel with an high efficiency on the B part of a fatbin file that can completely ruin the efficiency of the microkernel - think to the generation of slowdowns, during the execution of the B part of a fatbin file, due to the bandwidths and latencies of the GPU memories, this at cause of the order and type of ELF instructions of the B part of the fatbin file and the dependences among the ELF registers used in the ELF instructions of the B part of the fatbin file;

- There is not theoretical study considering the quadruplets ( fatbin file in $S^2_{F_f}$ , launch configuration in the $S_{lc}$ of the fatbin file in $S^2_{F_f}$ , unrolled path version of the B part of the fatbin file in $S^2_{F_f}$ , an input, of one of the subsets $SS_i$, of inputs generating the unrolled path version of the B part of the fatbin file in $S^2_{F_f}$ ), thing instead important to be able to give an a priori ELF code shortest execution time guarantee for the execution of the B part of a fatbin file.

  We show that it is important to classify fatbin files to determine the type of analysis/analyses that can be executed on them and that it is always important to generate the unrolled path version of the B part of a fatbin file considering the launch configuration and the input given to the fatbin file because this two things can make the difference for the generation of slowdowns due to causes different from the warp scheduling;

- There is not correct theoretical analysis able to give an a priori guarantee that the execution of the B part of the fatbin file can not be slowed down by the bandwidths and the latencies of the GPU memories, the warp scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps, and so that the only thing that can slow down the execution of the B part of a fatbin file is the warp scheduling - warp scheduling that we can not force, choose or control.

## 14.4 Summary

In the this chapter we have reviewed the previous work and considering the contributions of the thesis we have highlighted the problems that afflict all the results of all the papers that we were able to find. In the next chapter we briefly summarize the results of the thesis and highlight some of the possible future research directions.

# Chapter 15

# Conclusions and Future Research Directions

## 15.1 Introduction

In the previous chapter we have reviewed the previous work and considering the contributions of the thesis we have highlighted the problems that afflict all the results of all the papers that we have been able to find. In this chapter we briefly summarize the results of the thesis and highlight some of the possible future research directions.

## 15.2 Conclusions

Considering the impossibility to edit code using the real instruction set architecture a) we reversed engineered it - 6 - b) we devised a procedure - 6.6 - to generate fatbin files with at least the minimum number of resources later necessary to modify their B parts, 6.6 - this was necessary because it was not possible to bypass the compiler - and c) we devised another procedure to get the wanted ELF algorithmic implementations - 6.7.

We therefore discovered, understood and quantified some not disclosed GPU behaviors - 7 - that could slow down the execution of the B part of a fatbin file. This was necessary for the optimization process and so to understand a) how to transform an original fatbin file - 8 - b) how to classify fatbin files - 10 - c) which analysis/analyses can be executed on them - 11 - and d) how to execute the analysis/analyses - 11 and 12.

Next we devised a) a procedure that transforms an original fatbin file that we want to optimize in a set of equivalent fatbin files - 8.4 - and b) a procedure that generates for each one of the equivalent fatbin files a set of launch configurations - 8.3 - to use during the analysis/analyses of the equivalent fatbin files - because many different launch configurations could be used to execute the B part of a fatbin file, 2.5, and because the launch configurations in each set has to satisfy some requirements, then the launch configurations in a set are usually only a subset of all those possible.

We have therefore showed the importance of classifying fatbin files using a taxonomy for fatbin files that we have devised - 10 - and why the position of a fatbin file in the taxonomy determine the analysis/analyses - empirical and/or theoretical - that can be executed on the fatbin file - 11.

Finally, we devised a theoretical analysis - 12 - that, if it is applicable to a fatbin file, allows us

to give an *a priori* ELF code shortest execution time guarantee for the execution of the B part of the fatbin file - this supposing the fatbin file satisfies all the requirements of the theoretical analysis.

## 15.3 Future Research Directions

We focused on the optimization of the B part of a fatbin file - we want the execution time of the B part of a fatbin file the most short possible - but we want to extend our work considering also a) the part/parts of a fatbin file executed by a CPU and b) the part/parts of a fatbin file that implies/imply byte transfers between the CPU and the GPU of the machine where the fatbin file is executed. Possible future research directions, for the topics already considered in the thesis, can be instead the following:

- *Reverse engineering of the real instruction set:* The reverse engineering of the real instruction set for GPUs using the GF100 architecture is almost complete. We can reverse engineer some particular ELF instructions as, for example, the ELF instructions using the texture memories, the ELF instructions executing atomic updatings or the ELF instruction executing reduction operations, but the reverse engineering of all the other ELF instructions is already implemented in our framework. Completed the reverse engineering for all the ELF instructions of the real instruction set architecture used by the G100 architecture we wish to repeat the whole procedure for Kepler GPUs - we do not need to change anything in our framework;

- *Modification of ELF code:* Also whether we have complete control on the B part of any fatbin file and so we can get any wanted ELF algorithmic implementation, we are going to try to reverse engineer a) the procedure of generation applied by nvcc for the assignment of ELF registers to fatbin files, b) the procedure of assignment of hardware registers, to ELF registers, during the execution of the B part of a fatbin file, and c) the other parts of a fatbin file that we know are executed by a GPU but that are not visible in the interpretation text file generated by cuobjdump - 6.2;

- *Not disclosed GPU behaviors:* We want to discover, understand and quantity the not disclosed GPU behaviors of the architecture/architectures used by Kepler GPUs. This will require some work on the framework because also whether the architecture/architectures used by Kepler GPUs is/are similar to the GF100 architecture, there will be however some differences that it is necessary to consider to repeat, in the correct way, the whole procedure;

- *Transformation and launch configurations:* We want to automate the procedures described in 8. While in this thesis we devised a procedure to generate fatbin files that are equivalent to the original fatbin file, in future we will instead devise procedures able to generate fatbin files very different from the original fatbin file - it could be hard to discover a good process of generation because if the B part of a fatbin file generated is very different from the B part of the original fatbin file then it could be hard to be able to prove that there are good possibilities to execute the B part that is very different in a shorter quantity of time than that necessary to execute the B part of the original fatbin file;

- *Analysis of the equivalent fatbin files:* We want to expand the framework to make the execution of the empirical analysis automatic, but the parameters in input to the fatbin files are usually different for different fatbin files and therefore the preparation of the stack and the generation

of the variables, arrays, vectors and structures, that contain the input data and that will contain the output results for the execution of the B part of the fatbin file, in any case, are usually different from problem to problem and therefore can not be automated.

We want also to automate the procedure necessary for the execution of the theoretical analysis but we already know that the procedure, to determine the minimum number of resident warps that is necessary in each streaming multiprocessor to avoid pipeline stalls due to the scheduling waiting times, the dependence waiting times and the overhead time for the management of the warps, will be much easier to automate than the procedure necessary to determine whether the execution of the B part of the fatbin file can or not be slowed down by the bandwidths and the latencies of the GPU memories.

# Bibliography


[1] W. W. L. Fung H. Wong A. Bakhoda, G. Yuan and T. M. Aamod. Analyzing CUDA Workloads using a Detailed GPU Simulator. *Proceedings of the 36th annual international symposium on Computer architecture*, pages 152–163, 2009.

[2] G. Diamos A. Kerr and S. Yalamanchili. A Characterization and Analysis of PTX Kernels. *IEEE International Symposium on Workload Characterization*, pages 3–12, 2009.

[3] G. Diamos A. Kerr and S. Yalamanchili. Modeling GPU-CPU Workloads and Systems. *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 31–42, 2010.

[4] B. Meister M. Baskaran D. Wohlford C. Bastoul A. Leung, N. Vasilache and R. Lethin. A Mapping Path for Multi-GPGPU Accelerated Computers from a Portable High Level Programming Abstraction. *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 51–61, 2010.

[5] R.J. Barrientos, J.I. GÃşmez, C. Tenllado, and M. Prieto. Heap Based k-Nearest Neighbor Search on GPUs. *Conference Proceedings, XXI Jornadas de Paralelismo*, pages 559–566, 2010.

[6] N. Bell and M. Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. *Proceedings of 2011 Conference on Supercomputing*, 2009.

[7] M. Bernaschi1, M. Fatica, S. Melchionna, S. Succi, and E. Kaxiras. A Flexible High-Performance Lattice Boltzmann GPU Code for the Simulations of Fluid Flows in Complex Geometries. *Concurrency and Computation: Practice and Experience*, 22(1):1–14, 2010.

[8] B. Block, P. Virnau, and T. Preis. Multi-GPU Accelerated Multi-Spin Monte Carlo Simulations of the 2D Ising Model. *Computer Physics Communications*, 181(9):1549–1556, 2010.

[9] N. Brookwood. NVIDIA Solves the GPU Computing Puzzle. `http://www.nvidia.com/object/fermi-architecture.html`, 2010. [Online; accessed 23-October-2011].

[10] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington. Implementing Molecular Dynamics on Hybrid High Performance ComputersâĂŞShort Range Forces. *Computer Physics Communications*, 182(4):898–911, 2011.

[11] C. WenGuang C. DeHao and Z. WeiMin. CUDA-Zero: a Framework for Porting Shared Memory GPU Applications to Multi-GPUs. *Science in China - Information Sciences*, 55(2):663–676, 2012.

[12] X. Jia C. Men and S. B. Jiang. GPU-Based Ultra-Fast Direct Aperture Optimization for Online Adaptive Radiation Therapy. *Physics in Medicine and Biology*, 55:4309–4319, 2010.

[13] M.A. Clark, R. Babich, K. Barros, R.C. Brower, and C. Rebbi. Solving Lattice QCD Systems of Equations using Mixed Precision Solvers on GPUs. *Computer Physics Communications*, (181):1517–1528, 2010.

[14] B. Dongb B. Gutmana I. Yanovskyc D. Leea, I. Dinova and A. W. Togaa. CUDA Optimization Strategies for Compute - and Memory - Bound Neuroimaging Algorithms. *Computer Methods and Programs in Biomedicine*, 106(3):175–187, 2012.

[15] N. Wolovick E. E. Ferrero, J. P. De Francesco and S. A. Cannas. Q-State Potts Model Metastability Study using Optimized GPU-Based Monte Carlo Algorithms. *Computer Physics Communications*, 183(8):1578–1587, 2012.

[16] Z. Guo K. Tian E. Z. Zhang, Y. Jiang and X. Shen. On-the-Fly Elimination of Dynamic Irregularities for GPU Computing. *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 369–380, 2011.

[17] E.M. Garzon F. Vazqueza, J.J. Fernandeza. Automatic Tuning of the Sparse Matrix Vector Product on GPUs Based on the ELLR-T Approach. *Parallel Computing*, 38(8):408–420, 2012.

[18] S. Yalamanchili G. F. Diamos, A. R. Kerr and N. Clark. Ocelot: a Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems. *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 353–364, 2010.

[19] S. Triechlez E. Phillipsz Y. Baoy G. Tany, L. Liy and N. Suny. Fast Implementation of DGEMM on Fermi GPU. *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[20] P. Glaskowsky. NVIDIAs Fermi: the First Complete GPU Computing Architecture. `http://www.nvidia.com/object/fermi-architecture.html`, 2010. [Online; accessed 19-July-2011].

[21] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GpuTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. *SIGMOD - ACM*, pages 1–12, 2006.

[22] A. Greb and G. Zachmann. GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. *20th International Parallel and Distributed Processing Symposium*, 2006.

[23] P. Guo and L. Wang. Auto-Tuning CUDA Parameters for Sparse Matrix-Vector Multiplication on GPUs. *2010 International Conference on Computational and Information Sciences*, pages 1154–1157, 2010.

[24] J. Xue Y. Yang H. Cui, L. Wang and X. Feng. Automatic Library Generation for BLAS3 on GPUs. *Parallel and Distributed Processing Symposium*, 2011.

[25] M. S. Alvandi H. Wong, M. M. Papadopoulou and A. Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 235–246, 2010.

[26] T. R. Halfhill. Looking Beyond Graphics. `http://www.nvidia.com/object/fermi-architecture.html`, 2010. [Online; accessed 27-June-2011].

[27] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. *Proceedings of the 36th annual international symposium on Computer architecture*, pages 152–163, 2009.

[28] M. H. Lee S. Cho I. K. Park, N. Singhal and C. W. Kim. Design and Performance Evaluation of Image Processing Algorithms on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):91–104, 2011.

[29] M. H. LeeâĂă I. K. Park, N. Singhal and S. Cho. Efficient Design and Implementation of Visual Computing Algorithms on the GPU. *16th IEEE International Conference on Image Processing*, pages 2321–2324, 2009.

[30] S. S. Stone J. A. Stratton and W. W. Hwu. MCUDA: an Efficient Implementation of CUDA Kernels on Multi-Cores. *Book on Languages and Compilers for Parallel Computing*, pages 16–30, 2008.

[31] G. Hager J. Habich, T. Zeiser and G. Wellein. Performance Analysis and Optimization Strategies for a D3Q19 Lattice Boltzmann Kernel on NVIDIA GPUs using CUDA. *Advances in Engineering Software*, 42(5):266–272, 2011.

[32] S. Tomov J. Kurzak and J. Dongarra. Autotuning GEMM Kernels for the Fermi GPU. *Ieee Transactions on Parallel and Distributed Systems*, 23(11):2045–2057, 2012.

[33] A. Nisbet M. Amosb J. M. Cecilia, J. M. Garcia and M. Ujaldon. Enhancing Data Parallelism for Ant Colony Optimization on GPUs. *Journal of Parallel and Distributed Computing*, 73(1):42–51, 2013.

[34] K. Kumaran V. Vishwanath J. Meng, V. A. Morozov and T. D. Uram. GROPHECY: GPU Performance Projection from CPU Code Skeletons. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[35] A. Singh J. W. Choi and R. W. Vuduc. Model-driven Autotuning of Sparse Matrix-Vector Multiply on GPUs. *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 115–126, 2010.

[36] M. S. Rehman S. Patidar P. J. Narayanan K. Kothapalli, R. Mukherjee and K. Srinathan. A Performance Prediction Model for the CUDA GPGPU Platform. *International Conference on High Performance Computing*, pages 463–472, 2009.

[37] V. Kolonias, A. G. Voyiatzis, G. Goulas, and E. Housos. Design and Implementation of an Efficient Integer Count Sort in CUDA GPUs. *Concurrency and Computation: Practice and Experience*, (23):2365–2381, 2011.

[38] S. Cagnoni L. Mussi, F. Daolio. Evaluation of Parallel Particle Swarm Optimization Algorithms within the CUDA Architecture. *Information Sciences*, 18(1):4642âĂŞ4657, 2011.

[39] S. Cagnoni L. Mussi, F. Daolio. A Co-Evolutionary Differential Evolution Algorithm for Solving Min-Max Optimization Problems Implemented on GPU using C-CUDA. *Expert Systems with Applications*, 39(12):10324âĂŞ10333, 2012.

[40] H. Cook M. Bauer and B. Khailany. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[41] N. Harder A. Suratanee K. Rohr R. Konig M. Gipp, G. Marcus and R. Manner. Haralick's Texture Features Computation Accelerated by GPUs for Biological Applications. *Modeling, Simulation and Optimization of Complex Processes*, page 127âĂŞ137, 2012.

[42] J. Ramanujam M. M. Baskaran and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, pages 244–263, 2010.

[43] R. Bordawekar M. M. Baskaran. Optimizing Sparse Matrix-Vector Multiplication on GPUs. *Technical Report, IBM T.J.Watson Research Center*, 2008.

[44] S. Krishnamoorthy J. Ramanujam A. Rountev M. M. Baskaran, U. Bondhugula and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. *Proceedings of the 22nd annual international conference on Supercomputing*, pages 225–234, 2008.

[45] J. Meng and K. Skadron. Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs. *Proceedings of the 23rd international conference on Supercomputing*, pages 256–265, 2009.

[46] D. Merrill and A. S. Grimshaw. High Performance and Scalable Radix Sorting: a Case Study of Implementing Dynamic Parallelism for GPU Computing. *Parallel Processing Letters*, 21(2):245–272, 2011.

[47] I.V. Morozov, A.M. Kazennov, R.G. Bystryi, G.E. Norman, V.V. Pisarev, and V.V. Stegailov. Molecular Dynamics Simulations of the Relaxation Processes in the Condensed Matter on GPUs. *Computer Physics Communications*, 182(9):1974–1978, 2011.

[48] G. Diamos S. Yalamanchili N. Farooqui, A. Kerr and K. Schwan. A Framework for Dynamically Instrumenting GPU Compute Applications within GPU Ocelot. *4th Workshop on General-Purpose Computation on Graphics Procesing Units*, 2011.

[49] Nvidia. CUDA C Best Practices Guide Version 3.1. `http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation`, 2010. [Online; accessed 04-July-2011].

[50] Nvidia. CUDA C Programming Guide Version 3.1. `http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation`, 2010. [Online; accessed 11-January-2011].

[51] Nvidia. Fermi Compute Architecture. `www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf`, 2010. [Online; accessed 10-September-2012].

[52] Nvidia. PTX (Parallel Thread Execution) ISA Version 2.2. `http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation`, 2010. [Online; accessed 27-June-2011].

[53] Nvidia. The CUDA Compiler Driver Nvcc. `http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation`, 2010. [Online; accessed 07-May-2011].

[54] Nvidia. Cuobjdump (An Interpreting Tool for Fatbin Files). `http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation`, 2011. [Online; accessed 03-April-2011].

[55] Nvidia. CUDA C Best Practices Guide Version 4.1. `http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation`, 2012. [Online; accessed 07-September-2012].

[56] Nvidia. CUDA C Programming Guide Version 4.1. `http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation`, 2012. [Online; accessed 13-September-2012].

[57] D. Patterson. The Top 10 Innovations in the New NVIDIA Fermi Architecture and the Top 3 Next Challenges. `http://www.nvidia.com/object/fermi-architecture.html`, 2010. [Online; accessed 15-February-2011].

[58] S. Tomov R. Nath and J. Dongarra. An Improved Magma GEMM for Fermi Graphics Processing Units. *International J. of High Performance Computing Application*, 24(4):511–515, 2010.

[59] T. T. Dong R. Nath, S. Tomov and J. Dongarra. Optimizing Symmetric Dense Matrix-Vector Multiplication on GPUs. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[60] A. Resios. GPU Performance Prediction using Parametrized Models.pdf. *Master's thesis - Utrecht University*, 2011.

[61] R. Reyes and F. de Sande. Optimization Strategies in Different CUDA Architectures using LlCoMP. *Microprocessors and Microsystems*, 36(2):78–87, 2012.

[62] J. Siegel S. Carrillo and X. Li. A Control-Structure Splitting Optimization for GPGPU. *Proceedings of the 6th ACM conference on Computing frontiers*, pages 147–150, 2009.

[63] D. Defour S. Collange and D. Parello. Barra, a Modular Functional GPU Simulator for GPGPU. *Technical Report*, 2009.

[64] S. Min S. Lee and R. Eigenmann. OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization. *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110, 2009.

[65] S. S. Baghsorkhi S. S. Stone D. B. Kirk S. Ryoo, C. I. Rodrigues and W. W. Hwuy. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, 2008.

[66] S. J. Patel W. D. Gropp S. S. Baghsorkhi, M. Delahaye and W. W. Hwu. An Adaptive Performance Modeling Tool for GPU Architectures. *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 105–114, 2010.

[67] S. S. Baghsorkhi S. Ueng, M. Lathara and W. W. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. *Languages and Compilers for Parallel Computing*, pages 1–15, 2008.

[68] N. Satish, M. Harris, and M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. *Parallel and Distributed Processing Symposium - IEEE*, pages 1–10, 2009.

[69] E. Sintorn and U. Assarsson. Fast Parallel GPU-Sorting using a Hybrid Algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381–1388, 2008.

[70] H. H. B. Sorensen. Auto-Tuning Dense Vector and Matrix-Vector Operations for Fermi GPUs. *Parallel Processing and Applied Mathematics - Lecture Notes in Computer Science*, 72(3):619–629, 2012.

[71] J. E. Stone, J. Saam, D. J. Hardy, K. L. Vandivort, W. W. Hwu, and K. Schulten. High Performance Computation and Interactive Display of Molecular Orbitals on GPUs and Multi-Core CPUs. *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 9–18, 2009.

[72] C. Trapnell and M. C. Schatz. Optimizing Data Intensive GPGPU Computations for DNA Sequence Alignment. *Parallel Computing*, 35(8):429–440, 2009.

[73] M. Elkihela V. Boyera, D. El Baza. Solving Knapsack Problems on GPU. *Computers and Operations Research*, 39(1):42–47, 2012.

[74] V. Volkov and J. W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2008.

[75] S. Jiao D. Wang F Song W. Xu, H. Zhang and Z. Liu. Optimizing Sparse Matrix Vector Multiplication Using Cache Blocking Method on Fermi GPU. *ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2012.

[76] Z. Wei and J. Jaja. Optimization of Linked List Prefix Computations on Multithreaded GPUs using CUDA. *Parallel Process. Letters*, 22(4), 2012.

[77] M. Weigel. Performance Potential for Simulating Spin Models on GPU. *Journal of Computational Physics*, 231(8):3064–3082, 2012.

[78] C. Zhang X. Cui, Y. Chen and H. Mei. Auto-Tuning Dense Matrix Multiplication for GPGPU with Cache. *International Conference on Parallel and Distributed Systems*, 2010.

[79] E. Z. Zhang Y. Liu and X. Shen. A Cross-Input Adaptive Framework for GPU Program Optimizations. *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10, 2009.

[80] F. Ino Y. Okitsu and K. Hagihara. High-Performance Cone Beam Reconstruction using CUDA Compatible GPUs. *Parallel Computing*, 36(2):129–141, 2010.

[81] J. Kong Y. Yang, P. Xiang and H. Zhou. An Optimizing Compiler for GPGPU Programs with Input-Data Sharing. *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 343–344, 2010.

[82] Arturo Gonzalez-Escribano Yuri Torres and Diego R. Llanos. Understanding the Impact of CUDA Tuning Techniques for Fermi. *High Performance Computing Symposium*, pages 631–639, 2011.

[83] K. Rupnow Z. Cui, Y. Liang and D. Chen. An Accurate GPU Performance Model for Effective Control Flow Divergence Optimization. *IEEE 26th International Parallel and Distributed Processing Symposium*, pages 83–84, 2012.

[84] Y. Zhang and J. D. Owens. A Quantitative Performance Analysis Model for GPU Architectures. *IEEE 17th International Symposium on High Performance Computer Architecture*, pages 382–393, 2011.