UNIVERSITY OF CALIFORNIA,
IRVINE

Autonomic Software Tools that Discover and Quantify GPU Low Level Architectural Features and Machine Behaviors

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Fausto Artico

Thesis Committee:
Professor Alex Nicolau, Chair
Professor Alex V. Veidenbaum
Chancellor's Professor Nikil Dutt

2015

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CURRICULUM VITAE

Fausto Artico

INTERESTS

High Performance Computing
Algorithm Engineering
Supercomputing

ACCOMPLISHMENTS

In Italy, for the National Institute of Nuclear Physics (one of the preeminent CERN groups), Fausto developed cache-oblivious, auto-adaptive and fully parametric Fluid Dynamics codes for multi-threaded and multi-core, Power 5 and Power 3 IBM machines.

At the European Centre for Theoretical Studies in Nuclear Physics, he was one of the members of the Aurora Project. The Aurora Architectures, which use Intel Xeon E5s, became the most energy efficient HPC architectures in the world (http://www.green500.org/lists/green201306).

Fausto also ported on NVIDIA GPUs and consistently increased the speed of the LLNL Fast-J Core Code by more than 20 times. The Fast-J Core Code, created at UCI by the Nobel Peace Prize Michael J. Prather, requires more than 10 millions hours / year worldwide (likely an under-estimate).

For his PhD thesis at the Università degli Studi di Padova, he developed a design methodology that provides a priori peak performance guarantees (100% efficiency) for codes running on NVIDIA GPUs. This design methodology was tested using several flavors of genomic codes. In every case, for every run performed on a cluster of 12 Fermi GTX 475, the efficiencies of all the genomic codes were always greater than 96%.

Finally, as one of the members of the Variability Expedition (founded by NSF), for his PhD thesis at the University of California (Irvine), Fausto created a set of autonomic tools to reverse engineer the undisclosed NVIDIA ISAs. He also created a set of autonomic tools to discover, understand and quantify general undisclosed low level GPU architectural features and machine behaviors. This was necessary to facilitate code optimization processes.

EDUCATION

Ph.D.,	**Computer Science**	University of California, Irvine	USA	09/2015
M.S.,	**Computer Science**	University of California, Irvine	USA	06/2015
Ph.D.,	**IC Science and Techs**	Università degli Studi di Padova	IT	04/2014
M.S.,	**Computer Engineering**	Università degli Studi di Padova	IT	10/2009
B.S.,	**Computer Engineering**	Università degli Studi di Padova	IT	09/2006

COLLABORATIONS

European Centre for Studies in Nuclear Physics (Trento, IT) (07/2009 - 09/2011)
M.S. & Ph.D. Researcher. High Performance Computing Consultant (HPC Applications)

National Institute of Nuclear Physics (INFN Ferrara, IT) (10/2008 - 10/2009)
M.S. Researcher. Performance and Scalability Engineer (Fluid Dynamics and CPUs)

Information Manag. Systems Group (Dep. of Eng. DEI, IT) (09/2007 - 01/2009)
M.S. Researcher. Performance Engineer (Information Retrieval and Audio Fingerprints)

Numerical Analysis Group (Department of Mathematics, IT) (09/2007 - 09/2008)
M.S. Researcher. Performance Engineer (Isogeometric Analysis and FEA Methods)

Special Interest Group on Networking (Dep. of Eng. DEI, IT) (04/2008 - 07/2008)
M.S. Researcher. Software Engineer (Computer Science and Telecom Engineering)

Real Time Operating Systems Group (Dep. of Eng. DEI, IT) (01/2008 - 04/2008)
Software Engineer (Schedulers for Multi-Threaded and Multi-Processor Systems)

Artificial Intelligence Group (Dep. of Eng. DEI, IT) (09/2007 - 01/2008)
M.S. Researcher. Software Engineer (Neural Networks, Evolutionary and Genetic Algs)

POSITIONS

Center for Embedded Cyber-Physical Systems (09/2010 - 09/2015)
M.S. & Ph.D. Researcher. High Performance Computing Engineer (CPUs and GPUs)

High Performance Architectures and Compilers Group (09/2011 - 09/2015)
M.S. & Ph.D. Researcher. High Performance Computing Engineer (CPUs and GPUs)

Centre of Excellence on Advanced Computing Paradigms (10/2008 - 04/2014)
M.S. & Ph.D. Researcher. High Performance Computing Engineer (CPUs and GPUs)

ABSTRACT

Autonomic Software Tools that Discover and Quantify
GPU Low Level Architectural Features and Machine Behaviors

By

Fausto Artico

Doctor of Philosophy in Computer Science

University of California, Irvine, 2015

Professor Alex Nicolau, Chair

Graphics Processing Units (GPUs) are specialized hardware that was originally created to accelerate computer graphics and image processing [40]. However, their highly parallel structure [27] and their low costs per GFlop per Watt [52] make them attractive as energy efficient performing architectures that accelerate other computationally-intensive scientific tasks [47].

NVIDIA is the market leader for GPUs. To protect its market leadership position, NVIDIA does not disclose its real assembly (ELF) Instruction Set Architectures (ISAs), keeps the compiler code closed, and does not disclose many of its low level GPU architectural features and machine behaviors. Optimization processes therefore become very time consuming and involves trials and errors.

In this thesis we design and implement a set of autonomic tools to reverse engineer the ELF ISAs and design and implement another set of autonomic tools to discover and quantify the GPU low level architectural features and machine behaviors. We also show that, even when implementing kernels using NVIDIA virtual assembly (PTX), more than 40% of the total performance is lost. This is compared to implementing the same kernels using the ELF ISAs and exploiting the insight yielded from the discovery, understanding, and quantification of the GPU low level architectural features and machine behaviors.

Chapter 1

Introduction

In order to produce meaningful results, many codes will soon require code performance in the scale of PetaFlop and soon of ExaFlop [48]. Furthermore, such high performances must be reached within a reasonable power budget. Many of these codes are primarily already highly optimized for CPUs but not for Graphics Processing Units (GPUs) [40, 60], which are the best candidates to achieve the mentioned performance [34] and power [32] budget goals.

Accelerating codes on GPUs is difficult [56]. This is due to the fact that GPUs, when compared to CPUs, have many more functional units (hundreds versus less than 10 [14, 18]), less hierarchical memory levels (two instead of three [15, 16]), and smaller cache memories (an L2 cache of 256 KB instead of an L2 cache of 2 MB [13, 17]). Furthermore, GPUs are designed for high parallel arithmetic intensity and not for branching sequential code like CPUs are [28].

Because of the architectural differences between GPUs and CPUs, the optimization processes for GPUs are difficult. This is because, in order to protect its market leadership position, NVIDIA does not disclose its real assembly (ELF) Instruction Set Architectures (ISAs). In addition, further difficulties are present because NVIDIA keeps the compiler code closed and does not disclose many of its GPU low level architectural features and machine

behaviors.

Without the means to implement ELF kernels, we have no way to bypass the compiler. At the same time, because the compiler code is closed, we have no control over the optimizations applied by the compiler. For these reasons, our control over the code that is executed by the GPU is very limited and therefore it becomes difficult, if not impossible, to produce specific ELF codes. Without the ability to generate specific ELF codes, it becomes extremely difficult to accurately study, discover, understand and quantify the GPU low level architectural features and machine behaviors.

Furthermore, with the compiler code closed, we cannot modify its code and so cannot implement or modify the optimization rules of the compiler. It is also difficult to understand the optimization rules the compiler applies, how it applies them, and when, and therefore it becomes difficult to understand the compiler relative efficacy.

The fact that many GPU low level architectural features and machine behaviors are not disclosed makes it difficult, if not impossible, to understand why one code is performing much worse than another, to identify the most important architectural features and machine behaviors when optimizing a specific code, and to exploit them during the implementation process.

Taken together, the factors noted above make code optimizations into time consuming, trial and error processes. It also become difficult, if not impossible, to analyze ELF codes, to develop accurate performance models, and to develop systematic and scientific design methodologies that a priori guarantee peak performance.

Chapter 2

Useful Terms and Definitions

In this chapter we introduce the reader to useful terms and definitions that we use in this thesis. The discussions are valid for all GPUs that use the GF100 or later architectures - see chapter 3 for a description of the GF100 architecture and its main hardware components.

2.1 PTX: Virtual Machine and Virtual ISA

GPU codes can be edited using several "languages". One possibility is to use PTX. However, PTX, at its core, is also a Parallel Thread Execution virtual machine and a virtual ISA.

As reported in [44], the main aims of PTX are the following: 1) to provide a stable ISA that spans multiple GPU generations; 2) to achieve performance in compiled applications comparable to native GPU performance; 3) to provide a machine-independent ISA for C/C++ and other compilers to target; 4) to provide a code distribution ISA for application and middleware developers; 5) to provide a common source-level ISA for optimizing code generators and translators, which will map PTX to specific target machines; 6) to facilitate hand-coding of libraries, performance kernels, and architecture tests; and 7) to provide a scalable programming model that spans GPUs ranging 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 ELF ISA of the Tesla C2070 - the GPU we use in this thesis - because using PTX to edit GPU code allows us to skip several phases of the compiling chain executed by nvcc. By skipping these phases, we obtain a compiled code that has undergone a minor number of transformation phases.

2.2 The Nvcc Compiler

Any PTX code cannot be executed in its original form by a GPU. Before a 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 compiler. The nvcc source code is closed, so the only things we know about nvcc are the things written in its manual.

Nvcc accepts inputs from two different types of files. Both types of files contain code that we want to be executed by the GPU, but the code needs to be completely written a) only using PTX, or b) using one or more of the other programming languages allowed by NVIDIA. The two different types of files that nvcc can accept as input are the following:

- The .ptx, or parallel thread execution, files. The PTX files contain only GPU code and this code can only be PTX code. When the nvcc compiler accepts input from a PTX file, nvcc produces a fatbin file as output. The fatbin file contains the PTX code transformed into GPU assembly code - let us call the GPU assembly ELF, considering that when we use cuobjdump, 4.1, cuobjdump generates an interpretation text file of what it defines a fatbin ELF code.

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

  When a CUDA file is accepted by nvcc, it splits the CUDA file into one or more CPU and GPU parts. The GPU parts are first transformed by nvcc into PTX codes. Next,

the PTX codes are transformed by nvcc into ELF codes - this is done by considering the particular target GPU architecture where the PTX codes have to be executed. The ELF codes so obtained are a part of the fatbin files generated by nvcc during its compiling phase. 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 cubin file.

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

Each time a cubin file is launched, its GPU parts are executed by GPU threads. In cases where a fatbin file is produced starting with a PTX file, the fatbin file has to be called by a CUDA file and therefore the CUDA file will be processed to produce a cubin file as output. This is necessary because the processing always has to start on the CPU side, but a fatbin file produced starting with a PTX file will not have any C/C++ code (figure 2.1, page X).

2.3 GPU Thread Executions

Only the GPU parts some parts of each fatbin file are executed by the GPU, while the other parts of a fatbin file are executed by the CPU. The GPU parts are executed by GPU threads. When we launch a fatbin file it starts to be executed by the CPU, and afterward one or more of its parts are executed by the GPU.

In the parts executed by the GPU, there are some subparts - 4.1 - that are completely composed of ELF code instructions - let us call such subparts s_p. The s_p subparts - created by nvcc during the compiling process - correspond to the code that a) we want to be executed by the GPU, and that b) we wrote using PTX or one of the other available programming languages. Each time we launch a fatbin file, the s_p subparts are always executed by all the

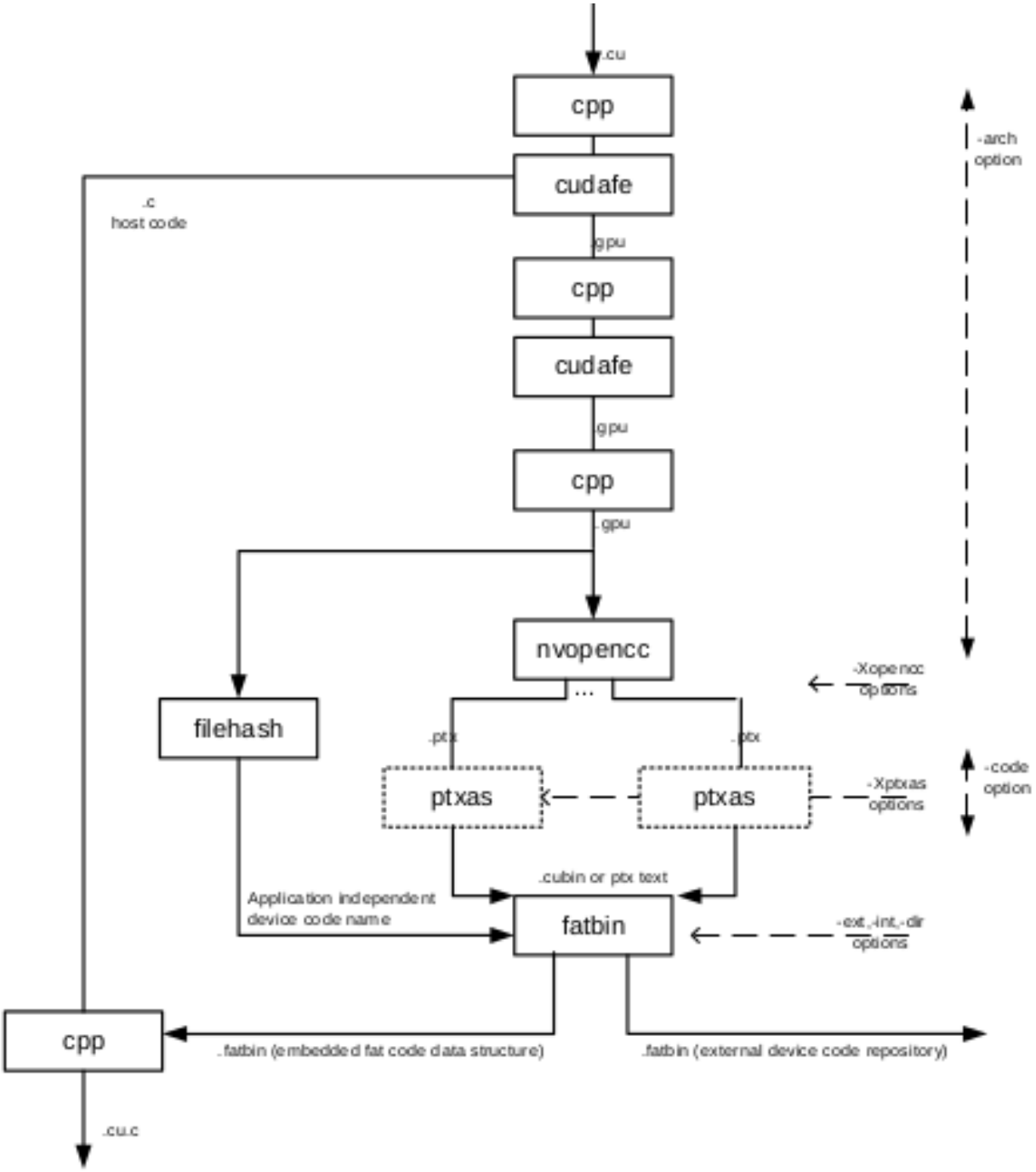

Figure 2.1: The nvcc compilation trajectory (courtesy of NVIDIA).

GPU threads we have decided to use at the moment of the fatbin launch. This does not mean that different threads execute different parts of the s_p subparts. Each GPU thread executes the entire s_p subparts of a fatbin file.

Different fatbin files can be executed in parallel on the GPU, but usually only one fatbin file is running on the GPU at a given moment in time. This is critical to avoid GPU hardware synchronization problems that we commonly face when launching different fatbin files in parallel.

All the launched GPU threads will execute the same ELF code, but the GPU threads can follow different paths inside the same ELF code. If this happens for the GPU threads of a warp - 3.1 - then a divergence phenomenon occurs. Each divergence phenomenon implies a slow down.

2.4 Launch Configurations

Each time we launch a fatbin file we need to decide a) the number of GPU thread blocks, b) the two dimensional space distribution of the GPU thread blocks, which we call the logic GPU thread block distribution, c) the number of GPU threads in each GPU thread block, which we call the GPU thread block composition, and d) the three dimensional space distribution of the GPU threads for each GPU thread block, which we call the logic GPU thread block form; this form has to be the same for all the GPU thread blocks. Each combination of values of the above parameters is a launch configuration - e.g. (6 , (2,3) , 1024 , (32,32,1)).

We can launch a maximum of 2^{30} GPU thread blocks per launch, with a maximum of 2^{15} GPU thread blocks along the x and y dimensions of the two dimensional space. The GPU thread blocks have to be distributed starting from the origin (0 , 0), and must 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), and must be contiguous

along the x, y and z dimensions of the three dimensional space.

Table 2.1: Some Possible Launch Configurations for a Fatbin File

```
Note: grid.z has always to be equal to 1

Launch Configuration 1:
.....................................................................
//  12 thread blocks     =  3 *  4 * 1
// 782 threads per block = 27 * 17 * 2
//  25 warps per block   = ceil of ( 782 / 32 )
.....................................................................
dim3    grid1;      grid1.x =  3;    grid1.y =  4;    grid1.z =  1;
dim3 threads1;   threads1.x = 27; threads1.y = 17; threads1.z =  2;

Launch Configuration 2:
.....................................................................
// 2^(32) thread blocks     = 2^(16) * 2^(16) * 1
//   1024 threads per block =     32 *     32 * 1
//     32 warps per block   = ceil of ( 1024 / 32 )
.....................................................................
dim3    grid1;      grid1.x = 2^(16);    grid1.y = 2^(16);    grid1.z =  1;
dim3 threads1;   threads1.x =     32; threads1.y =     32; threads1.z =  1;

Launch Configuration 3:
.....................................................................
//      1 thread blocks     = 1 * 1 * 1
//      1 threads per block = 1 * 1 * 1
//      1 warps per block   = ceil of ( 1 / 32 )
.....................................................................
dim3    grid1;      grid1.x =  1;    grid1.y =  1;    grid1.z =  1;
dim3 threads1;   threads1.x =  1;  threads1.y = 1; threads1.z =  1;

Using of the Launch Configurations:
.....................................................................
name_cuda_function <<< grid1 , threads1 >>> ( function's parameters );
name_cuda_function <<< grid2 , threads2 >>> ( function's parameters );
name_cuda_function <<< grid3 , threads3 >>> ( function's parameters );
.....................................................................
```

2.5 Instruction Configurations

The GPU executes ELF instructions. As we will see, each PTX instruction corresponds to one or more ELF instructions - 4.2 - and there are ELF instructions that do not correspond to any PTX instruction - 5.3.

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 to execute the PTX instruction; and 3) the type and number of dependences among the ELF registers used in the ELF instructions 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 consider PTX instructions but are interchangeable with ELF instructions.

2.5.1 Definition

Any type of PTX instruction - add.s32, sub.u64, etc. - can be executed in two different modes: normal mode (.n) or conditional mode (.c). The conditional mode can be executed if a guard is set either at true (.no) or false (.ne).

If the PTX instruction has some PTX registers, then these registers are usually also used in other PTX instructions. PTX registers can therefore exhibit read-read, read-write, write-write and write-read dependences.

We defined an instruction configuration as any set of specific values of three parameters - e.g. (add.u32 , n , write-read).

2.5.2 Dependence Distances

The read-read, read-write, write-write and write-read dependences of each PTX register have a distance of zero or more PTX instructions - zero only if the PTX register is reused in the same PTX instruction.

2.5.3 Examples

In example 1, the sub.s32 PTX instruction should be noted. The sub.s32 PTX instruction is executed in normal mode, and the dependences are write-read for the PTX register %result_1 and read-read for the PTX register %operand_2. The sub.s32 PTX instruction is therefore 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 equal to 3 PTX instructions. The second instruction configuration considers the read-read dependence of the PTX register %operand_2: this dependence has a distance equal to 7 PTX instructions.

Table 2.2: PTX Instruction Sub.s32 Executed in Normal Mode

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

Table 2.3: Ptx Instruction Div.u64 Executed in Conditional Mode (Guard True)

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 div.u64 PTX instruction should be noted. The div.u64 PTX instruction is executed in conditional mode where 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 equal to 4 PTX instructions. The second instruction configuration considers the write-read dependence of the PTX register %result_0: this dependence has a distance equal to 6 PTX instructions.

In example 3, the bfind.b32 PTX instruction should be noted. The bfind.b32 PTX instruction is executed in normal mode, and the type of dependences are write-write for the PTX register %result_0, 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 three instruction configurations. The first instruction configuration considers the write-write dependence of the PTX register %result_0: this dependence has a distance equal to 2 PTX instructions. The second instruction configuration considers the write-read dependence of the PTX register %result_1: this dependence has a distance equal to 4 PTX instructions. The third 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 6.

Table 2.4: Ptx Instruction Bfind.b32 Executed in Normal Mode

```
.....
@!%guard_1 popc.b32 %result_0, %operand_1;
.....
cnot.b32 %result_1, %operand_0;
.....
add.s32 %result_0, %operand_2, %operand_3; .....
bfind.b32 %result_0, %result_1, %operand_0;
.....
```

2.6 Summary

In this chapter we have introduced useful terms and definitions that we will use in this thesis, but that are also valid for all 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 virtual ISA. PTX is 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 that we want to be executed by the GPU. The PTX code cannot be executed by the GPU in its original form but rather has to be used as input for the NVIDIA compiler nvcc before becoming GPU executable.

- Nvcc can receive input from PTX or CUDA codes and always produces output as fatbin files. Inside each fatbin file, the code we want executed by the GPU is transformed into ELF code. When a fatbin file is launched, one or more of its parts are executed by the CPU, and one or more of its parts 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. However, inside each of the GPU parts, each GPU thread can follow different paths. The GPU threads executing a fatbin file have to be logically organized before any fatbin file is launched.

- The logical GPU thread organization has many degrees of freedom, and these degrees are important because, as we will see, they determine some GPU behaviors.

- Any PTX or ELF instruction can be executed in two different modes: normal, or conditional when the guard is set at either true or false. The PTX or ELF registers used by the PTX or ELF instructions can exhibit write-write, write-read, read-read or read-write dependences.

  The configurations of instruction, execution mode and dependence type are called instruction configurations, and these configurations are important for discovering, understanding and quantifying the GPU low level architectural features and machine behaviors.

Chapter 3

The GF100 Architecture

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 transistors.

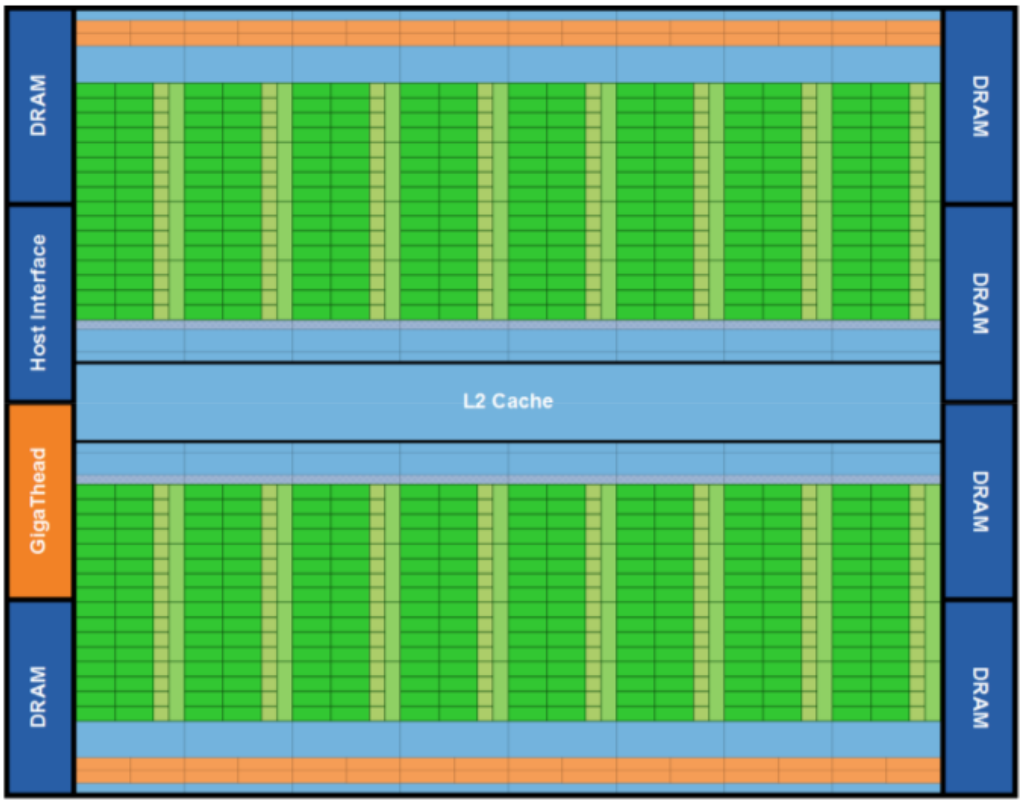

Figure 3.1: A high level view of the GF100 architecture. Note its modularity, its 6 global memory controllers, its gigathread scheduler and its shared L2 cache (courtesy of NVIDIA).

Commercial GPUs that use the GF100 architecture are the Fermi GTX 465, Fermi GTX 470 and Fermi GTX 480. In addition, 2 high-end Tesla GPUs also use the GF100 architecture: the Tesla C2050 and the Tesla C2070.

3.1 Main Components of the GF100 Architecture

Off chip, the GF100 has a private GDDR 5 RAM: on chip it has a L2 cache, a constant cache, a gigathread scheduler, 4 graphics processing clusters, and a maximum of 6 memory controllers. Here is what we know about each of these components:

- *GDDR 5 RAM:* Fermi GTX cards have 256 MB attached to each of the enabled GDDR 5 memory controllers, for a total of either 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. Bandwidths for the models are as follow: 102.6 GB/s for the Fermi GTX 465, 133.9 GB/s for the Fermi GTX 470, 177.4 GB/s for the Fermi GTX 480, and 144 GB/s for the Tesla C2050 and C2070.

- *L2 Cache:* The L2 cache is on chip and ranges between 768 and 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 store the data present in the L1 caches, but it is not necessary that the L2 cache stores the data that are present in the shared memories and hardware registers.

- *Constant Cache:* The constant cache has a dimension of 64 KB and can only be written only by the CPU. The GPU executes warps and each warp is always composed of 32 GPU threads. If all the 32 GPU threads that compose 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 32 GPU threads. If instead the 32 GPU threads read 32 different

constant memory cells, one for each GPU thread, at least 32 clock cycles are necessary to satisfy all 32 distinct requests.

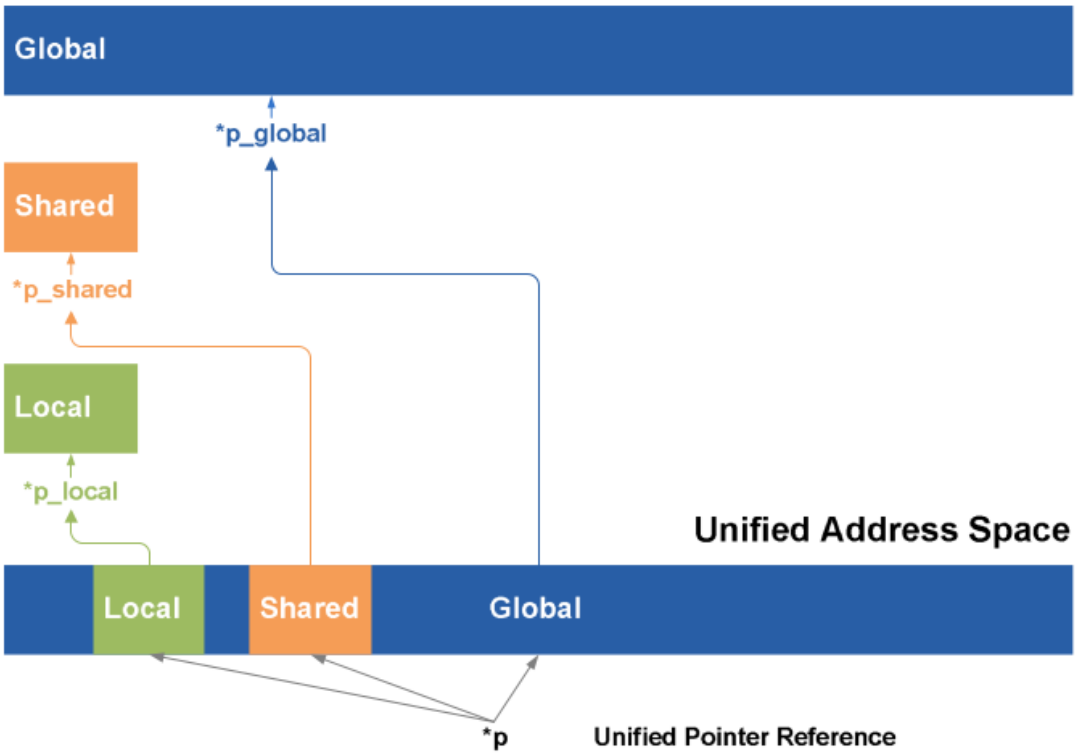

Figure 3.2: The GF100 architecture uses an unified address space. The shared L2 cache and the constant memory are not represented in this picture but for them the same principle is valid (courtesy of NVIDIA).

- *Gigathread Scheduler:* Throughout the execution of the fatbin file, using a launch configuration (see section 2.4), the gigathread scheduler has to assign the GPU thread blocks to the streaming multiprocessors and, later, must schedule the warps of each GPU thread block that reside in a streaming multiprocessor. The assignments and schedulings are executed by the two gigathread scheduler levels:

  – Chip Level: The gigathread scheduler assigns the GPU thread blocks to the streaming multiprocessors. After a GPU thread block is assigned to a streaming multiprocessor, the GPU thread block cannot migrate. The gigathread scheduler can manage a maximum of 21504 GPU threads on the fly.

The assignment of the GPU thread blocks is executed by 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.

These design limits are: a) the maximum number of GPU thread blocks that reside in a streaming multiprocessor (8); b) the maximum number of GPU threads a streaming multiprocessor can manage on the fly (1536); and c) the total quantity of shared memory required by the potential set of GPU thread blocks that reside in a streaming multiprocessor (this total quantity has to be smaller than either 16 or 48 KB, depending on how we set the GPU before the execution of the fatbin file).

– Streaming Multiprocessor Level: The gigathread scheduler in each streaming multiprocessor is represented by 2 warp schedulers. The 2 warp schedulers concurrently schedule warps on the hardware resources of the streaming multiprocessor. The 2 warp schedulers in each streaming multiprocessor can manage a maximum of 48 warps or 1536 GPU threads on the fly.

The assignments and schedulings are executed at an undisclosed 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 functional units (the CUDA cores, the load and store units and the special functional units) in a streaming multiprocessor.

This assumption is reasonable because: a) a warp is scheduled on only 1 of the 4 groups of functional units in a streaming multiprocessor when a warp ELF instruction has to be executed; b) the CUDA cores, the load and store units, and the special functional units all have the same clock frequency; c) a warp is always composed of 32 GPU threads; and d) the maximum number of functional units in each of the 4 groups of functional units is 16, and therefore at least 2 functional unit clock cycles are necessary

to execute any warp ELF instruction.

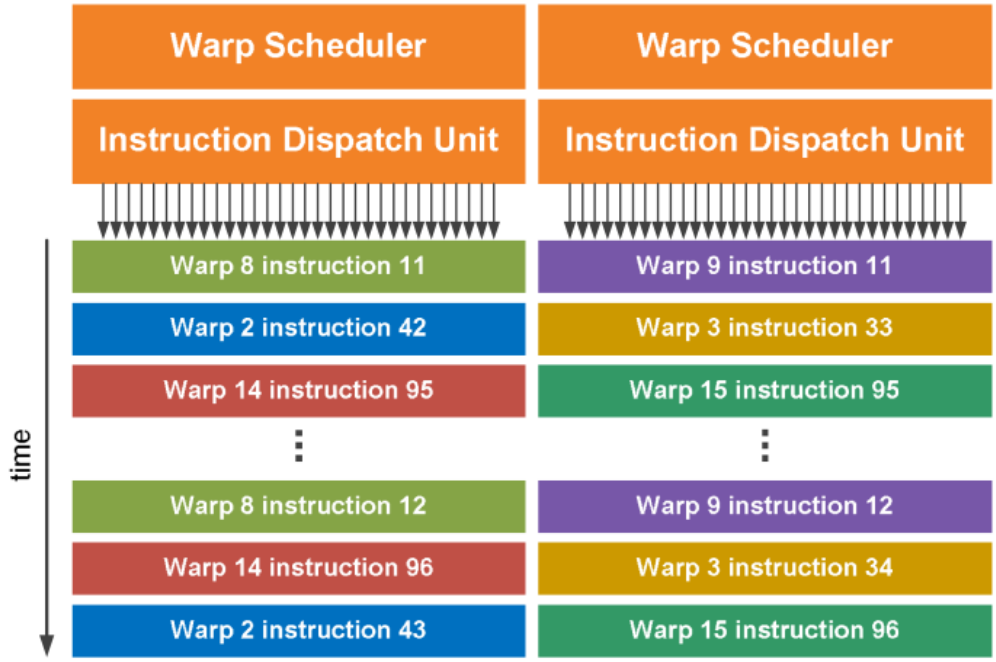

Figure 3.3: The warp schedulers schedule warps (courtesy of NVIDIA).

If the clock frequency used for the schedulings is greater than half the clock frequency of the functional units then it would be possible to get some input queues to the 4 groups of functional units in a streaming multiprocessor. However, this is improbable when considering a) the die area that the queues would require, and b) the control logic that would be necessary to manage of the queues.

Furthermore, if the clock frequency used for the schedulings is smaller than half of the clock frequency of the functional units then the theoretically achievable peak performance per second would be determined by the clock frequency of the warp schedulers and not by the clock frequency of the functional units. As a result, part of the speed of the functional units would be wasted.

For these reasons, it is therefore reasonable to assume that the clock frequency of the warp schedulers is exactly half of the clock frequency of the functional units.

- *Graphics Processing Clusters:* Each graphics processing cluster has a raster engine

and a maximum of 4 streaming multiprocessors. The Tesla C2070 has 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 of the streaming multiprocessors.

- *Streaming Multiprocessors:* Each streaming multiprocessor has 64 KB of private ram, $2^{15} = 32768$ 32-bit hardware registers, 32 CUDA cores, 16 load and store units, 4 special functional 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 these interacts with each other.

3.2 Main Components of a Streaming Multiprocessor

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

- *L1 Cache:* We can choose from only 2 configurations for the 64 KB blocks of private RAM in each streaming multiprocessor, and the chosen configuration has to be the same for all the streaming multiprocessors during the entire 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. The configurations are the following:

  - Configuration 1: Each one of the 64 KB blocks of private RAM is partitioned into 48 KB and 16 KB blocks. The 48 KB blocks are managed by the hardware of

the GPU as L1 cache blocks, while the 16 KB blocks have to be managed by the programmer as shared memory blocks.

  - Configuration 2: Each one of the 64 KB blocks of private RAM is partitioned into 16 KB and 48 KB blocks. The 16 KB blocks are managed by the hardware of the GPU as L1 cache blocks, while the 48 KB blocks have to be managed by the programmer as shared memory blocks.

- *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 cannot be used for data exchanges. The shared memory is divided into 4-byte blocks and works in the following way:

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

  - When more GPU threads want to read or write different shared memory blocks at the same time, then each one of the shared, 4-byte memory blocks, involved in the read or write, will be concurrently read or written at the same time.

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, a unit for the gathering of the operands, a floating point unit, an integer unit, and a result queue. Each CUDA core can execute a 32-bit fusion multiplication and add instruction per clock cycle.

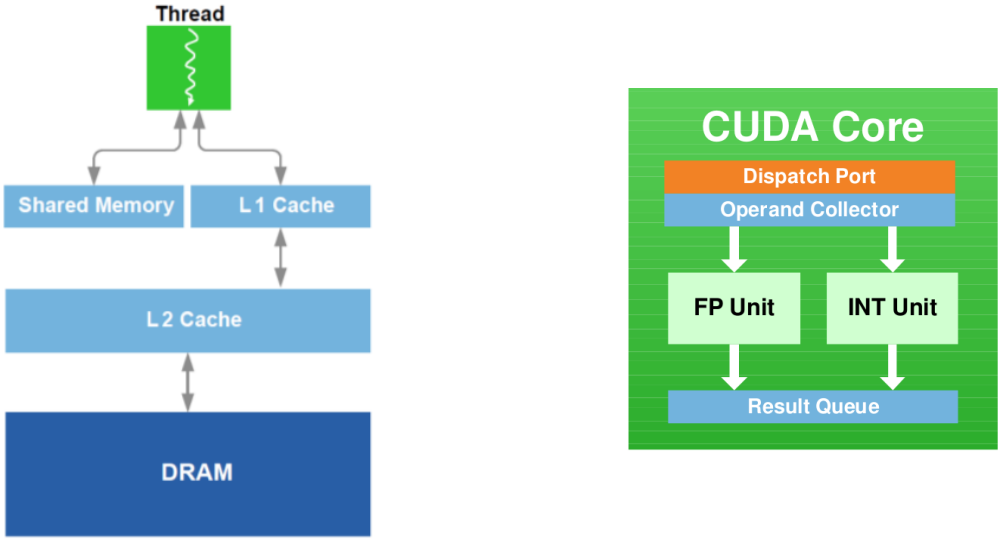

Figure 3.4: Memory hierarchy and CUDA core architecture (courtesy of NVIDIA).

Load and Store Units: There are 16 load and store units. Such units load and store data to and from any memory address. The addresses are normally 64-bit addresses. The clock frequency of the load and store units is 1.15 GHz.

Special Functional Units: There are 4 special functional units that execute transcendental instructions such as sin, cos, reciprocal and square root and have a clock frequency of 1.15 GHz.

Each special functional unit executes, at maximum, one 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 functional units, it is impossible for the warp instruction to be executed in less than 8 functional unit clock cycles, because every warp is composed of 32 GPU threads.

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

Hardware Registers: The hardware registers are 32-bit registers and are assigned to

each GPU thread. At the beginning of each execution, a register can be assigned to a GPU thread. During the execution, only that GPU thread can read and write the register. The data in a hardware register need not be in the L2 cache, L1 cache or in the shared memory.

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

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

- If the warp is scheduled on 1 of the groups of 16 functional units, then the warp is executed as 2 half-warps in the next 2 functional unit clock cycles.
- If the warp is scheduled on the group of 4 special functional units, then the warp is executed as 8 eighth-warps in the next 8 functional unit clock cycles.

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

Texture Mapping Units: There are 4 texture mapping units. Each texture mapping unit has 4 texture filtering units. GPUs like the Tesla C2070, which has 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 a dimension 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 vertex fetch, tessellation, and viewport transform instructions, and has an attribute setup unit and a streaming output unit.

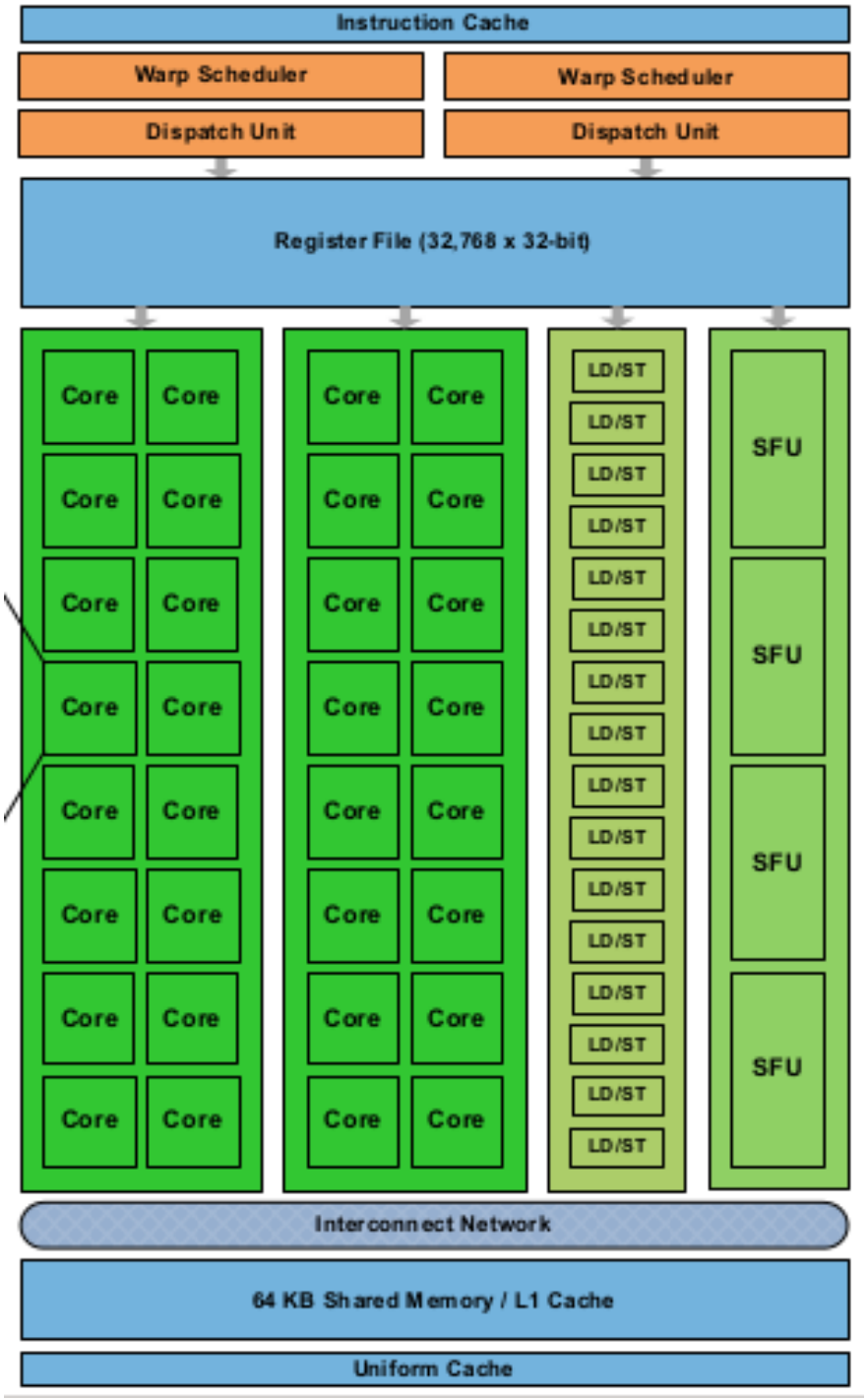

Figure 3.5: Main components of each streaming multiprocessor (courtesy of NVIDIA).

3.3 Theoretical Peak Performances per Second

Our discussion is restricted here to considering the 4 groups of functional units usually used to execute scientific computing - the 2 groups of 16 CUDA cores, the group of 16 store and load units, and the group of 4 special functional units.

For each warp scheduler clock cycle, not more than 2 warps can be scheduled per streaming multiprocessor, and so no more than $2 \cdot 14 = 28$ warps can be scheduled on the whole GPU per clock cycle. Such warps cannot be executed in less than 2 functional unit clock cycles because a warp is composed of 32 GPU threads, but each group of functional units have is not larger that 16. A single functional unit with a clock frequency of 1.15 GHz can therefore execute a maximum of 1.15 G instructions per second. With not more than $14 \cdot 32 = 448$ functional units executing one instruction 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 the theoretical GPU peak performance of $\frac{515.2}{2} = 257.2$ GF/s is possible for instructions using 64-bit operands.

3.4 Summary

In this chapter we have described the main hardware components of the GF100 architecture, and have analyzed the main hardware components of the streaming multiprocessors, and have described how they interact. These tasks are necessary to calculate the theoretical GPU Tesla C2070's peak performances per second and to identify the amount of time that is necessary to execute a fatbin file on the GPU.

The Tesla C2070 has the following features: 6 GB of GDDR 5 RAM off chip with a bandwidth of 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 functional units at 1.15 GHz with a clock frequency of 1.15 GHz; a total of $2^{15} \cdot 14 = 458752$ 32-bit 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-bit or smaller operands and a theoretical GPU peak performance of 257.2 GF/s for instructions using 64-bit operands.

Chapter 4

Reverse Engineering of the ELF ISAs

The lowest of the "high level programming languages" available to users for writing GPU code is PTX, but PTX is only a virtual ISA. Developing theories that consider, optimize and analyze PTX codes is meaningless because the GPU architecture executes ELF code, not PTX code. We experimentally verified that the ELF codes produced by nvcc are usually very different when compared to the original PTX codes, in terms of their a) number, order and type of instructions, and b) number, type and reuse of registers.

To correctly and accurately discover and quantify the undisclosed GPU low level architectural features and machine behaviors, it is therefore necessary to design and implement ELF kernels, but to do this it is first necessary to design and implement an autonomic software tool to reverse engineer the ELF ISAs.

4.1 PTX $\rightarrow$ ELF: Localization

The real instruction set architecture of the GF100 is not disclosed and we do not know the binary codes, which correspond to the ELF instructions used by the GF100, needed to execute the PTX instructions of a PTX code. We can, however, interpret every fatbin file produced by nvcc using NVIDIA's cuobjdump.

Cuobjdump receives input from a fatbin file and produces an interpretation text file as

output, in which we can see how the input PTX code has been transformed by nvcc into ELF code. Every line of the interpretation text file has three columns. The order of the columns can be different for different cuobjdump's versions but the columns are the same.

Table 4.1: Interp. Text File generated by Cuobjdump for the Mad.rz.f64.c.ne PTX File

```
1
2  Fatbin elf code:
3  ................
4  code for sm_20
5     Function : function
6  ................
7  /*0000*/    /*0x00005de428004404*/    MOV R1, c [0x1] [0x100];
8  /*0008*/    /*0xfc1fdc03207e0000*/    IMAD.U32.U32 RZ, R1, RZ, RZ;
9  /*0010*/    /*0xffffdc0450ee0000*/    BAR.RED.POPC RZ, RZ;
10 /*0018*/    /*0xffffdc0450ee0000*/    BAR.RED.POPC RZ, RZ;
11 /*0020*/    /*0xfc1fdc0450ee8000*/    BAR.RED.POPC RZ, 0x1;
12 /*0028*/    /*0x80001de428004000*/    MOV R0, c [0x0] [0x20];
13 /*0030*/    /*0x20029c034801c000*/    IADD R10.CC, R0, 0x8;
14 /*0038*/    /*0x93f2dc4348004000*/    IADD.X R11, RZ, c [0x0] [0x24];
15 /*0040*/    /*0x00a01c8584000000*/    LD.E R0, [R10];
16 /*0048*/    /*0xfc1fdc0450ee8000*/    BAR.RED.POPC RZ, 0x1;
17 /*0050*/    /*0xfc1fdc0450ee8000*/    BAR.RED.POPC RZ, 0x1;
18 /*0058*/    /*0x10a09c8584000000*/    LD.E R2, [R10+0x12];
19 /*0060*/    /*0xfc1fdc0450ee8000*/    BAR.RED.POPC RZ, 0x1;
20 /*0068*/    /*0xfc2fdc0450ee8000*/    BAR.RED.POPC RZ, 0x2;
21 /*0070*/    /*0x0801dc23188e0000*/    ISETP.LT.AND P0, pt, R0, R2, pt;
22 /*0078*/    /*0xfc2fdc0450ee8000*/    BAR.RED.POPC RZ, 0x2;
23 /*0080*/    /*0xfc3fdc0450ee8000*/    BAR.RED.POPC RZ, 0x3;
24 /*0088*/    /*0xfc3fdc0450ee8000*/    BAR.RED.POPC RZ, 0x3;
25 /*0090*/    /*0x20a11ca584000000*/    LD.E.64 R4, [R10+0x16];
26 /*0098*/    /*0xfc3fdc0450ee8000*/    BAR.RED.POPC RZ, 0x3;
27 /*00a0*/    /*0xfc3fdc0450ee8000*/    BAR.RED.POPC RZ, 0x3;
28 /*00a8*/    /*0x40a19ca584000000*/    LD.E.64 R6, [R10+0x24];
29 /*00b0*/    /*0xfc3fdc0450ee8000*/    BAR.RED.POPC RZ, 0x3;
30 /*00b8*/    /*0xfc3fdc0450ee8000*/    BAR.RED.POPC RZ, 0x3;
31 /*00c0*/    /*0x60a21ca584000000*/    LD.E.64 R8, [R10+0x32];
32 /*00c8*/    /*0xfc3fdc0450ee8000*/    BAR.RED.POPC RZ, 0x3;
33 /*00d0*/    /*0x00031c0348000000*/    IADD R12, R0, R0;
34 /*00d8*/    /*0xfc4fdc0450ee8000*/    BAR.RED.POPC RZ, 0x4;
35 /*00e0*/    /*0x08201c0348000000*/    IADD R0, R2, R2;
36 /*00e8*/    /*0xfc4fdc0450ee8000*/    BAR.RED.POPC RZ, 0x4;
37 /*00f0*/    /*0x18429c0121900000*/    DFMA.RZ R10, R4, R6, R8;
```

```
/*00f8*/    /*0x08a09c0420100000*/    SEL R2, R10, R2, !P0;
/*0100*/    /*0x0cb0dc0420100000*/    SEL R3, R11, R3, !P0;
/*0108*/    /*0xfc5fdc0450ee8000*/    BAR.RED.POPC RZ, 0x5;
/*0110*/    /*0x80029de428004000*/    MOV R10, c [0x0] [0x20];
/*0118*/    /*0x9002dde428004000*/    MOV R11, c [0x0] [0x24];
/*0120*/    /*0x80a29c0348014000*/    IADD R10.CC, R10, c [0x0] [0x20];
/*0128*/    /*0x90b2dc4348004000*/    IADD.X R11, R11, c [0x0] [0x24];
/*0130*/    /*0x2cafdc03207e0000*/    IMAD.U32.U32 RZ, R10, R11, RZ;
/*0138*/    /*0xfc6fdc0450ee8000*/    BAR.RED.POPC RZ, 0x6;
/*0140*/    /*0x00a29ca594000000*/    ST.E.64 [R10+0x40], R10;
/*0148*/    /*0xfc6fdc0450ee8000*/    BAR.RED.POPC RZ, 0x6;
/*0150*/    /*0xe0a29ca594000000*/    ST.E.64 [R10+0x48], R10;
/*0158*/    /*0xfc6fdc0450ee8000*/    BAR.RED.POPC RZ, 0x6;
/*0160*/    /*0x30c31c0348000000*/    IADD R12, R12, R12;
/*0168*/    /*0xfc6fdc0450ee8000*/    BAR.RED.POPC RZ, 0x6;
/*0170*/    /*0x00a31c8594000000*/    ST.E [R10], R12;
/*0178*/    /*0xfc6fdc0450ee8000*/    BAR.RED.POPC RZ, 0x6;
/*0180*/    /*0x10a31c8594000001*/    ST.E [R10+0x52], R12;
/*0188*/    /*0xfc6fdc0450ee8000*/    BAR.RED.POPC RZ, 0x6;
/*0190*/    /*0x2cb2dc4348000000*/    IADD.X R11, R11, R11;
/*0198*/    /*0xfc6fdc0450ee8000*/    BAR.RED.POPC RZ, 0x6;
/*01a0*/    /*0x60a2800594000001*/    @P0 ST.E.U8 [R10+0x56], R10;
/*01a8*/    /*0xfc7fdc0450ee8000*/    BAR.RED.POPC RZ, 0x7;
/*01b0*/    /*0x80a09ca594000001*/    ST.E.64 [R10+0x57], R2;
/*01b8*/    /*0xfc7fdc0450ee8000*/    BAR.RED.POPC RZ, 0x7;
/*01c0*/    /*0xa0a11ca594000001*/    ST.E.64 [R10+0x65], R4;
/*01c8*/    /*0xfc7fdc0450ee8000*/    BAR.RED.POPC RZ, 0x7;
/*01d0*/    /*0xc0a19ca594000001*/    ST.E.64 [R10+0x73], R6;
/*01d8*/    /*0xfc7fdc0450ee8000*/    BAR.RED.POPC RZ, 0x7;
/*01f0*/    /*0xe0a21ca594000001*/    ST.E.64 [R10+0x81], R8;
/*01f8*/    /*0xfc7fdc0450ee8000*/    BAR.RED.POPC RZ, 0x7;
/*0200*/    /*0x00001de780000000*/    EXIT;
```

In the above example, in the first column, there is a series of consecutive hexadecimal addresses with the address in the first row always equal to 0. The second column lists the binary representations of the ELF instructions at the addresses in the first column, and the third column lists the human readable text form representations of the binary codes of the ELF instructions in the second column.

Even do NVIDIA does not disclose its ELF ISAs, we discover that each ELF instruction is 8 bytes in dimension by analyzing the second column of the interpretation text files generated

by cuobjdump. This is because there are always 16 hexadecimal digits in the second column of each interpretation text file generated by cuobjdump.

We also experimentally found that the dimension of each fatbin file is always bigger than the byte dimension of an ELF instruction when multiplied by the number of ELF instructions visible in any interpretation text file. Because we need to write/modify ELF codes to generate ELF kernels, this implies that we need to first design and implement a robust autonomic tool T_1 that is able to locate the correct position of each ELF instruction in the fatbin files that is also visible in the interpretation text files.

For the design and implementation of T_1, we formulated 2 conjectures: C_1) the binary codes of the ELF instructions displayed in hexadecimal form in the second column of the interpretation text files could be byte permutations of the real binary codes of the ELF instructions; and C_2) the ELF instructions visible in the interpretation text files are always consecutive ELF instructions in the fatbin files.

To verify C_1 and C_2, each time T_1 analyzes a fatbin file, it first extracts the potential binary codes of the ELF instructions from the interpretation text file of the fatbin file and next executes the below procedure.

Next, for each one of the possible 8! permutations of the 8 bytes of each binary code of each ELF instruction, T_1 executes the following procedures: 1) transformation of all the potential binary codes of the ELF instructions in the interpretation text file; 2) alignment of the block of consecutive permuted binary codes of the ELF instructions that are visible in the interpretation fatbin file with every byte position in the fatbin file; 3) for every alignment above, calculation of the similarity score as the number of ELF instructions with perfect matches.

After the execution of the procedure for all the possible 8! permutations, T_1 next: 1) determines the maximum similarity score among all the similarity scores calculated; 2) checks 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) checks that the maximum similarity score

appears only one time. T_1 verified that both C_1 and C_2 are true because of the following:

- C_1 is true because, for CUDA drivers' version 4.2, the ELF instruction bytes in positions 0, 1, 2, 3, 4, 5, 6, 7 on the hard disk are interpreted by cuobjdump as bytes in positions 5, 6, 7, 8, 0, 1, 2, 3 for the hexadecimal representations of the binary codes of the ELF instructions that are visible in the interpretation text files of the fatbin files. For CUDA drivers' version 6.5, they are simply in reverse order.

- C_2 is true because the maximum similarity score is always equal to the number of visible ELF instructions in the interpretation text file of a fatbin file, and because the maximum similarity score always appears only one time for each fatbin file.

Finally, T_1 stores the position P in the fatbin file, where the alignment of the block of consecutive real binary codes of the ELF instructions gives a perfect match, and other important information about the fatbin file.

Table 4.2: Features of the ELF Code generated for the Mad.rz.f64.c.ne PTX File

```
   1 start offset, in bytes, in the fatbin file, to get the visible ELF ins.
1057 pos, in bytes, in the fatbin file, to get the first visible ELF ins.
  33 num of potential ELF instructions preceding the first visible ELF ins.
  72 num of visible ELF instructions in the file generated by cuobjdump
```

Noting the above example for the mad.rz.f64.c.ne PTX instruction, we can see that even if there are 8 bytes in each ELF instruction, the ELF instructions are not aligned to frontiers that are multiples of 8 (the starting offset, first line, is different from 0 and it is not a multiple of 8). Furthermore, the ELF instructions are not located at the beginning of the fatbin file (the first byte of the first ELF instruction is the 1057 byte of the fatbin file), and the number of visible ELF instructions, when multiplied by 8 and then added to 1057 does not reach the end of the fatbin file.

T_1 stores these features because such features are necessary to modify or write ELF codes, but before modifying or writing these codes we need to use such features to discover how each individual PTX instruction is transformed into ELF instructions.

4.2 PTX → ELF: Instruction Correspondences

To write and modify ELF codes we need to discover and understand the ELF instructions that are necessary to execute a PTX instruction. To accomplish this goal, we developed 2 autonomic tools:

- T_2: this tool generates a file for each PTX instruction that we want to reverse engineer. T_2 automatically generates the files using specific guidelines that are necessary to satisfy in order to guarantee that the discovering phase is correctly executed, and so also to guarantee that the discovered ELF instructions for a PTX instruction are in fact the ELF instructions necessary to execute the PTX instruction.

- T_3: this tool checks that the structures of a fatbin file and a PTX file are equivalent. T_3 is used to check that the structures of the PTX files produced by T_2 are equivalent to the structures of the corresponding fatbin files generated by nvcc. This is necessary to correctly localize the ELF instructions that are necessary to execute a PTX instruction.

We experimentally verified that nvcc usually produces ELF codes very different from the input PTX codes but we need guarantees that the discovering phase is correctly executed, and therefore that the ELF instructions necessary to execute a PTX instruction are effectively correct.

To discover the correct ELF instructions, T_2 generates PTX files following specific guidelines, which force the compiler to avoid the saving or reuse of registers, or the application of optimizations that could destroy the original structure of the PTX file at the fatbin level. This makes it difficult, if not impossible, to correctly locate the ELF instructions corresponding to a PTX instruction. The guidelines have to be different for different types of PTX

instructions because there are many different PTX registers that can be used by the PTX instructions.

Table 4.3: Mad.rz.f64.c.ne PTX File

```
1
2  DIRECTIVES FOR THE COMPILER
3  .version X.X                                    // PTX version
4  .target sm_YY                                   // target architecture
5  .address_size ZZ                                // 32 or 64 bits binary
6  BEGINNING OF THE FUNCTION
7  .entry function( .param .u64 aii ){
8   DECLARATION OF THE REGISTER OF SUPPORT AND LOADING
9   .reg.u64 %ros ;                                          bar.sync 0 ;
10  ld.param.u64 %ros , [ aii ] ;                            bar.sync 0 ;
11  DECLARATION OF TWO AUXILIARY REGISTERS AND LOADINGS
12  .reg.s32 %reg_s32_au1 ;                                  bar.sync 1 ;
13  ld.global.s32 %reg_s32_au1 , [ %ros +  8 ];              bar.sync 1 ;
14  .reg.s32 %reg_s32_au2 ;                                  bar.sync 1 ;
15  ld.global.s32 %reg_s32_au2 , [ %ros + 12 ];              bar.sync 1 ;
16  DECLARATION OF A PREDICATE REGISTER AND ITS SETTING
17  .reg.pred %reg_p ;                                       bar.sync 2 ;
18  setp.lt.s32 %reg_p , %reg_s32_au1 , %reg_s32_au2 ;       bar.sync 2 ;
19  DECLARATION OF THE REGISTERS OF THE PTX INSTRUCTION AND LOADINGS
20  .reg.f64 %reg_f64_0 ;                                    bar.sync 3 ;
21  .reg.f64 %reg_f64_1 ;                                    bar.sync 3 ;
22  ld.global.b64 %reg_f64_1 , [ %ros + 16 ] ;               bar.sync 3 ;
23  .reg.f64 %reg_f64_2 ;                                    bar.sync 3 ;
24  ld.global.b64 %reg_f64_2 , [ %ros + 24 ] ;               bar.sync 3 ;
25  .reg.f64 %reg_f64_3 ;                                    bar.sync 3 ;
26  ld.global.b64 %reg_f64_3 , [ %ros + 32 ] ;               bar.sync 3 ;
27  UPDATING OF THE TWO AUXILIARY REGISTERS
28  add.s32 %reg_s32_au1 , %reg_s32_au1 , %reg_s32_au1 ; bar.sync 4 ;
29  add.s32 %reg_s32_au2 , %reg_s32_au2 , %reg_s32_au2 ; bar.sync 4 ;
30  THE PTX INSTRUCTION TO REVERSE ENGINEER
31  @!%reg_p mad.rz.f64 %reg_f64_0 , %reg_f64_1 , %reg_f64_2 , %reg_f64_3 ;
32                                                       bar.sync 5 ;
33  STORING THE VALUES OF SOME REGISTERS
34  add.u64 %ros , %ros , %ros ;                             bar.sync 6 ;
35  st.global.u64 [ %ros + 40 ] , %ros ;                     bar.sync 6 ;
36  add.s32 %reg_s32_au1 , %reg_s32_au1 , %reg_s32_au1 ; bar.sync 6 ;
37  st.global.s32 [ %ros + 48 ] , %reg_s32_au1 ;             bar.sync 6 ;
38  add.s32 %reg_s32_au2 , %reg_s32_au2 , %reg_s32_au2 ; bar.sync 6 ;
```

```
39  st.global.s32 [ %ros + 52 ] , %reg_s32_au2 ;          bar.sync 6 ;
40  add.u64 %ros , %ros , %ros ;                          bar.sync 6 ;
41  STORING THE VALUES OF THE REGISTERS OF THE PTX INTSRUCTION TO REV. ENG.
42  @%reg_p st.global.u8 [ %ros + 56 ] , %ros ;           bar.sync 7 ;
43  st.global.b64 [ %ros + 57 ] , %reg_f64_0 ;            bar.sync 7 ;
44  st.global.b64 [ %ros + 65 ] , %reg_f64_1 ;            bar.sync 7 ;
45  st.global.b64 [ %ros + 73 ] , %reg_f64_2 ;            bar.sync 7 ;
46  st.global.b64 [ %ros + 81 ] , %reg_f64_3 ;            bar.sync 7 ;
47  EXIT FROM THE FUNCTION
48  exit;
49 }
```

In the above example, T_2 generates the PTX file necessary to reverse engineer the mad.rz. f64.c.ne PTX instruction configuration. The guidelines used by T_2 to generate the PTX file are complex and were determined by generating many PTX kernels, transforming the PTX kernels into fatbin files, and then analyzing them.

Following the editing guidelines, T_2 generates one PTX file per PTX instruction configuration. Each PTX file has a unique structure. The PTX files have different sections, can load and store many times the values of their PTX registers, and have only one type of bar.sync (synchronization barriers) PTX instruction per section.

Thanks to their particular structure, the PTX files force nvcc to produce ELF codes that are structurally identical to the original PTX code. Since the editing guidelines used by T_2 were determined experimentally, T_3 analyzes and compares the structure of each PTX file and the corresponding ELF code for each instruction configuration. T_3 essentially verifies that the type, number and order of ELF instructions that correspond to the bar.sync PTX instructions are the same as the bar.sync instructions for each section.

For example, there are 6 identical bar.sync PTX instructions in the PTX file generated for the mad.rz.f64.c.ne PTX instruction (see Table 4.3), and so there has to be 6 identical ELF instruction in the ELF code generated by nvcc (see Table 4.1). These 6 identical ELF instructions all have to come after the ELF instructions that correspond to the bar.sync PTX instructions of type 0, 1, 2, 3, 4 and 5, but also must precede all of the bar.sync PTX

instructions of type 7.

The autonomic tools reverse engineered more than 400 PTX instruction configurations. T_2 found that the structure of each PTX file and of its corresponding ELF code, generated by nvcc, were always identical. This shows that the guidelines are robust.

Finally, we designed and implemented a different autonomic tool, T_4. After T_3's job is completed, T_4 extracts the ELF instructions necessary to reverse engineer the mad.rz.f64.c.ne PTX instruction configuration, which must be located between the ELF instructions that correspond to the last bar.sync 4 PTX instruction and the bar.sync 5 PTX instruction.

Table 4.4: PTX → ELF: Instruction Correspondences for the Mad.rz.f64.c.ne

```
mad.rz.f64.c.ne

-> DFMA.RZ [we do not know anything about these three ELF instructions]
-> SEL     [we only know that they are used to execute the mad.rz.f64.c.ne]
-> SEL     [that they are used in this order and SEL is executed twice]
```

4.3 PTX → ELF: Register Correspondences

Knowing what ELF instructions are necessary to execute a PTX instruction is not sufficient to modify ELF codes; we also need to determine the correspondence among PTX and ELF registers.

The autonomic tool T_5 discovers these correspondences, identifies whether of not the number of ELF registers necessary to execute a PTX instruction is greater than the number of PTX registers used in the PTX instruction, and whether or not ELF registers have no corresponding PTX register.

Table 4.5: PTX → ELF: Register Correspondences for the Mad.rz.f64.c.ne

```
%reg_p     (PTX predicate register) ->  P0      (ELF predicate register )
```

```
%reg_f64_0 (PTX    result register) ->  R2,  R3 (ELF    result registers)
%reg_f64_1 (PTX operand 1 register) ->  R4,  R5 (ELF operand 1 registers)
%reg_f64_2 (PTX operand 2 register) ->  R6,  R7 (ELF operand 2 registers)
%reg_f64_3 (PTX operand 3 register) ->  R8,  R9 (ELF operand 3 registers)
?                                   -> R10, R11
```

For the mad.rz.f64.c.ne PTX instruction configuration, T_5 discovers that the PTX predicate register $\%reg\_p$ is the ELF predicate register $P0$. This is possible because T_5 knows that the PTX predicate register $\%reg\_p$ is used in the unique PTX store instruction, which is located between the last PTX bar.sync 6 instruction and the first PTX bar.sync 7 instruction (see line 42 in Table 4.3), and because it knows that the ELF predicate registers are preceded by the reserved character @ (see line 59 in Table 4.1).

T_5 also discovers that the PTX 64-bit register $\%reg\_f64\_0$ that is used to store the result of the mad.rz.f64.c.ne PTX instruction has 2 corresponding ELF registers. In fact, it was experimentally determined that the compiler assigns 2 consecutive ELF registers to each PTX 64-bit register, in spite of the fact that often only the first of the 2 ELF registers appear in the text form interpretation of the ELF instructions that are necessary to execute the PTX instructions - e.g. $ST.E.64[R10 + 0x57], R2$; (line 61 in Table 4.1). This happens because the GF100 architecture only has 32-bit registers.

Finally, T_5 discovers that the ELF instructions necessary to execute the mad.rz.f64.c.ne PTX instruction configuration use 2 ELF registers, $R10$ and $R11$, without corresponding PTX registers being used in the PTX instruction.

The register correspondences are useful because NVIDIA does not disclose anything about the human representation of the ELF instructions, which can be read in the third column of the interpretation text files generated by cuobjdump. However, these correspondences, as well the instruction correspondence, will be necessary for the modification of ELF codes.

4.4 ELF Instructions: Binary Reverse Engineering

Now that we discovered the PTX → ELF instruction and register correspondences, we need to discover what bits of each ELF instruction correspond to the opcode of each ELF instruction, and what bits correspond to each ELF register used in each ELF instruction.

To accomplish this goal, we designed and implemented T_6. T_6 discovers that the ELF ISAs do not have a fixed format and creates a database of matches among the human representations of the ELF instructions and their binary codes.

When given an ELF instruction, T_6 takes its human representation and creates its corresponding abstract representation by utilizing the features of the different discovered ELF registers (see first frame below), and the algorithm indicated in the second frame below.

Table 4.6: Features of Discovered ELF Registers

Name	Type	Sub-type	Our Identifier
P0	predicate	not reserved	NT2
...	.....	.....	...
P7	predicate	not reserved	NT2
pt	predicate	reserved	ST2
R0	normal	not reserved	NT1
R1	normal	reserved	ST1
R2	normal	not reserved	NT1
...	.....	.....	...
R62	normal	not reserved	NT1
RZ	normal	reserved	ST1

Table 4.7: Algorithm To Create An Abstract Human Readable Text Form Representation

Human Readable Text Form Representation: SEL R2, R10, R2, !P0;

Registers	Our Identifiers	Order per Identifier	Abstract Forms
R2	NT2	0	NT2ER0
R10	NT2	1	NT2ER1

```
       R2              NT2                      2           NT2ER2
       P0              NT1                      0           NT1ER0

Example -> NT2ER1
N       -> sub-type of the ELF register: not reserved
T2      -> type of the ELF register: normal ELF register
ER1     -> from left to right, second normal not reserved ELF register

Abstract Forms   Frequencies   Features
           ST2             0          0
           ST1             0        0_0
           NT2             3      0_0_3
           NT1             1    0_0_3_1

Position Result Register: 0 -> first register appearing in the instruction
                          (this is not always true for other instructions)

Abstract Form Result Register: NT2ER0 (possibly anything for other instr.)

Human Readable .....      Abstract Human Readable Text Form Representation
SEL R2, R10, R2, !P0  SEL_NT2ER0,_NT2ER1,_NT2ER2,_!NT1ER0_0_0_3_1_0_NT2ER0
```

If the abstract representation has not already been discovered and reverse engineered, then T_6 flips each of the 64 single bits of the real binary code of the abstract interpretation. 64 new binary codes are therefore generated for the abstract interpretation.

Table 4.8: Binary Codes Generated For An Abstract Human Readable Text Form Interp.

```
ELF Instruction        One Of Its Human Readable Text Form Interpretations
SEL                                                  SEL R2, R10, R2, !P0

Human Readable .....      Abstract Human Readable Text Form Interpretation
SEL R2, R10, R2, !P0  SEL_NT2ER0,_NT2ER1,_NT2ER2,_!NT1ER0_0_0_3_1_0_NT2ER0

Binary Code Returned By Cuobjdump                        Real Binary Code
0x08a09c0420100000                                   0x2010000008a09c04

Bit Flipped         New Real Binary Code        Cuobjdump's Interpretation
         63           0xa010000008a09c04                0x08a09c04a0100000
         62           0x6010000008a09c04                0x08a09c0460100000
         ..           ..................                ..................
```

1	0x2010000008a09c06	0x08a09c0620100000
0	0x2010000008a09c05	0x08a09c0520100000

Next, using the 64 new binary codes, T_6 makes a copy of the original fatbin file, overwrites the 64 ELF instructions in the copy that correspond to the first 64 ELF instructions visualized in the interpretation text file of the copy, and uses cuobjdump to interpret the overwritten copy.

Table 4.9: The Overwriting of The Fatbin File Copy

Rel. Address	Original Binary	Bit Flipped	New Binary
0000	0x2800440400005de4	63	0xa010000008a09c04
0008	0x207e0000fc1fdc03	62	0x6010000008a09c04
....	.....	..	.....
....	.....	1	0x2010000008a09c06
....	.....	0	0x2010000008a09c05

Table 4.10: The Interpretation Of The Overwritten Fatbin File Copy

Bit Flipped	Previous Interpretation	Actual Interpretation
..	.....	.....
50	BAR.RED.POPC RZ, 0x2	SEL R2, R10, R2, !P4
49	BAR.RED.POPC RZ, 0x3	SEL R2, R10, R2, !P2
48	BAR.RED.POPC RZ, 0x3	SEL R2, R10, R2, !P1
..	.....	.....
30	MOV R11, c [0x0] [0x24]	SEL R2, R10, R34, !P0
29	IADD R10.CC, R10, c [0x0] [0x20]	SEL R2, R10, R18, !P0
28	IADD.X R11, R11, c [0x0] [0x24]	SEL R2, R10, R10, !P0
27	IMAD.U32.U32 RZ, R10, R11, RZ	SEL R2, R10, R6, !P0
26	BAR.RED.POPC RZ, 0x6	SEL R2, R10, R0, !P0
25	ST.E.64 [R10+0x40], R10	SEL R2, R10, R3, !P0
24	BAR.RED.POPC RZ, 0x6	SEL R2, R42, R2, !P0
23	ST.E.64 [R10+0x48], R10	SEL R2, R26, R2, !P0
22	BAR.RED.POPC RZ, 0x6	SEL R2, R2, R2, !P0
21	IADD R12, R12, R12	SEL R2, R14, R2, !P0
20	BAR.RED.POPC RZ, 0x6	SEL R2, R8, R2, !P0

```
19                    ST.E [R10], R12      SEL R2, R11, R2, !P0
18                BAR.RED.POPC RZ, 0x6     SEL R34, R10, R2, !P0
17                ST.E [R10+0x52], R12     SEL R18, R10, R2, !P0
16                BAR.RED.POPC RZ, 0x6     SEL R10, R10, R2, !P0
15                IADD.X R11, R11, R11      SEL R6, R10, R2, !P0
14                BAR.RED.POPC RZ, 0x6      SEL R0, R10, R2, !P0
13         @P0 ST.E.U8 [R10+0x56], R10      SEL R3, R10, R2, !P0
..   ................................    ....................
```

T_6 next takes the human readable text form interpretation of the 64 new binaries and creates corresponding abstract interpretations. An abstract interpretation of a new binary can be equal to or different from the abstract interpretation of the human representation of the original ELF instruction, e.g. SEL. If the abstract interpretation is different from the human one, then that means that the bit belongs to the opcode of the original ELF instruction. If the two are equal, then that means that the bit is one of those necessary to modify one of the ELF registers used by the original ELF instruction.

Table 4.11: The Features Of An Abstract Human Readable Text Form Representation

```
Example of Abstract Human Readable Text Form Representation

SEL_NT2ER0,_NT2ER1,_NT2ER2,_!NT1ER0_0_0_3_1_0_NT2ER0

The SEL ELF instruction does not use any reserved ELF register. It instead
uses 3 normal ELF registers and 1 normal predicate (negated) ELF register.

------------------------------------------------------------------------

This SEL ELF instruction does not require the use of reserved ELF regi-
sters. However, reserved ELF registers are not considered for the calcula-
tion of the order of the identifiers. The bits of the reserved registers
are in fact considered fixed as the bits of the opcode of the instruction.

------------------------------------------------------------------------

Bit Flipped   Bit Value    Field  ELF Register Type  Order Per Identifier

         63           0   opcode                \                     \
```

```
..          .   opcode                \                \
52          1   opcode                \                \

51        0\1   NT1ER0          predicate                3
50        0\1   NT1ER0          predicate                3
49        0\1   NT1ER0          predicate                3

48          0   opcode                \                \
..          .   opcode                \                \
32          0   opcode                \                \

31        0\1   NT2ER2             normal                2
30        0\1   NT2ER2             normal                2
29        0\1   NT2ER2             normal                2
28        0\1   NT2ER2             normal                2
27        0\1   NT2ER2             normal                2
26        0\1   NT2ER2             normal                2

25        0\1   NT2ER1             normal                1
24        0\1   NT2ER1             normal                1
23        0\1   NT2ER1             normal                1
22        0\1   NT2ER1             normal                1
21        0\1   NT2ER1             normal                1
20        0\1   NT2ER1             normal                1

19        0\1   NT2ER0             normal                0
18        0\1   NT2ER0             normal                0
17        0\1   NT2ER0             normal                0
16        0\1   NT2ER0             normal                0
15        0\1   NT2ER0             normal                0
14        0\1   NT2ER0             normal                0

13          0   opcode                \                \
..          .   opcode                \                \
 0          0   opcode                \                \
```

Next, T_6 generates the binary codes for all the possible human representations that have abstract representations equal to the abstract representation of the ELF instruction that T_6 is reverse engineering, e.g. SEL R2, R10, R2, !P0. Finally, T_6 verifies that the binaries are in fact correct, updates the database of the human representations, and updates the binary database.

Table 4.12: Other Types Of Discovered SEL ELF Instructions

During the entire reverse engineering process ,the autonomic tools discover several types of SEL ELF instructions. Note that the reverse engineering procedure described in this thesis was only for (1). (2), (3), (4) and (5) are instead discovered to be feeding the autonomic tools with many different fatbin files.

SEL ELF Instruction Used In The Examples
SEL R2, R10, R2, !P0 (1)

Its Partial Abstract Human Readable Text Form Representation
(PA1) SEL_NT2ER0,_NT2ER1,_NT2ER2,_!NT1ER0

Different Discovered SEL ELF Instructions		Differences Compared to (1)
SEL R32, R10, R12, P7	(2)	P7 is not negated (!) like P0
SEL R27, RZ, RZ, P7	(3)	RZ is a reserved register
SEL R3, RZ, 0x8, !P1	(4)	RZ, constant 0x8, and (!)
SEL R7, RZ, 0x2, P3	(5)	RZ and constant 0x2

Their Partial Abstract Human Readable Text Form Representations
(PA2) SEL_NT2ER0,_NT2ER1,_NT2ER2,_NT1ER0
(PA3) SEL_NT2ER0,_ST2ER0,_ST2ER1,_NT1ER0
(PA4) SEL_NT2ER0,_ST2ER0,_XxX0,_!NT1ER0
(PA5) SEL_NT2ER0,_ST2ER0,_XxX0,_NT1ER0

Id.				Binary
(PA1)	--+--------+333	----------------2222211111000000	-+++-------+--	
(PA2)	--+---------333	----------------2222211111000000	-+++-------+--	
(PA3)	--+---------333	----------------++++++++++000000	-+++-------+--	
(PA4)	--+--------+333	-++2222222222222222222	+++++000000	-+++-------+--
(PA5)	--+---------333	-++2222222222222222222	+++++000000	-+++-------+--

Definition of Sensible Field: Any not reserved ELF register or constant)

Legend:
-: bit equal to 0
+: bit equal to 1
0: bit positions associated to the first sensible field from left
1: bit positions associated to the second sensible field from left
2: bit positions associated to the third sensible field from left
3: bit positions associated to the forth sensible field from left

Note: The normal reserved ELF register RZ has a binary value of 111111;

```
see PA3 bit positions 30 to 23 and 22 to 15, PA4 bit positions 22 to 15,
or PA5 bit positions 22 to 15. The negation of the ELF predicate register
is instead determined by the value 1 of the bit in position 51.
```

4.5 Summary

In this chapter we have described the procedures a) that were necessary to uncover undisclosed 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 desired ELF algorithmic implementations. The central points of this chapter are as follows:

- In a fatbin file, the ELF instructions that correspond to the PTX code of a PTX file are consecutive, and occupy only the B part of the fatbin file. In contrast, parts A and C of the fatbin file are created by nvcc and we cannot know what they contain.

  We are able to locate 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 produce the desired ELF algorithmic implementations.

- One or more ELF instructions are necessary to execute a PTX instruction. The correspondences (between each PTX instruction and the number, type and order of ELF instructions necessary to execute that PTX instruction) have been determined for all the PTX instructions of interest.

  We have also determined the correspondences between the ELF registers used in the ELF instructions, which are necessary to execute a PTX instruction, and the PTX registers used in the PTX instruction. In addition, we determined what ELF registers do not have a corresponding PTX register and the cases in which this lack of correspondence occurs.

  These results are useful for modifying a fatbin file using a set of ELF registers that are different from the original ones used in the ELF instructions.

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

- There are not ELF instructions to assign ELF registers to a fatbin file, so the ELF registers have to be assigned by nvcc during the compiling phase. These registers will correspond to some hardware registers, which have to be different for each GPU thread that is 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 contained in the B part of a fatbin file, so it has to be in parts A and/or C, both of which are not disclosed. This is because nvcc cannot know the launch configuration that we are going to use for each execution of the fatbin file, and so nvcc cannot know the number of GPU threads that will be used to execute the fatbin file - this number could be different each time.

  To produce the desired ELF algorithmic implementations, it is therefore necessary to: a) write a PTX file using a particular set of editing guidelines; b) input the PTX file into nvcc to produce a fatbin file, which (thanks to the use of the editing guidelines) has at least the minimum number and types of ELF registers necessary to modify the fatbin file; and to c) modify the ELF code that correspond to the PTX file, in order to produce the desired ELF algorithmic implementations.

- For each desired ELF algorithmic implementation, thanks to the previous results, we can always begin by generating a fatbin file that has at least the minimum number and types of ELF registers necessary to modify the fatbin file. We can always modify the fatbin file later by overwriting its B part with the order, type and number of ELF instructions we want, each of which will use the ELF registers that we want.

Such a modified fatbin file will a) have the desired ELF algorithmic implementation, b) run without launch failures due to violations from using ELF registers that were not originally assigned to it, and c) guarantee that its execution is logically correct. We know that the executions are correct because we know the role of each ELF register in each ELF instruction (e.g. result, operand), and the specific values of the bits in the binary codes of each ELF instruction.

Chapter 5

Generation of Fatbin Shells

Reverse engineering the ELF ISAs is not enough to generate assembly codes. This is because, in the interpretation text files, there are not ELF instructions that correspond to the PTX instructions used to declare PTX registers. It is therefore necessary to force the nvcc compiler to generate fatbin files with certain hardware resources (e.g. a specific number and type of ELF registers), and next overwrite specific parts of the generated fatbin files to produce the desired micro-benchmarking assembly kernels. These kernels are then used to correctly and accurately discover and quantify the undisclosed GPU low level architectural features and machine behaviors.

5.1 Register Mismatch

During the PTX $\rightarrow$ ELF compilation phase, by using particular flags like -Xptxas=-v, we can force nvcc to disclose the number of hardware registers that each GPU thread will use to execute the generated fatbin file. This number is determined by nvcc using its undisclosed compiling rules and cannot change after the generation of the fatbin file. This number remains independent of the number of executions and launch configurations.

If we count the number of different text forms that correspond to the different ELF registers that are visible in the interpretation text file of a fatbin file (e.g. R10, P3), and

compare it to the number of hardware registers that are assigned to each GPU thread, then the number of different text forms that correspond to the different ELF registers visible in the interpretation text file of a fatbin file should always be equal to the number of hardware registers assigned to each GPU thread. However, we found that the number of different text forms that correspond to the different ELF registers visible in the interpretation text file of a fatbin file is always equal to 1 plus the number of hardware registers assigned to each GPU thread.

To test the possibility that this is the result of a bug in nvcc, we therefore edited several PTX files using the editing guidelines described in the previous chapter, and we declared more than 64 PTX registers per PTX file file (64 is 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 by analyzing the interpretation text files of the fatbin files, we see that the number of different text forms that correspond to different ELF registers visible in the interpretation text file is always 64.

After this first verification, we created other PTX files using different numbers of PTX registers per PTX file, with each number smaller than 64. in addition, for all these cases, the number of different text forms that correspond to different ELF registers visible in the interpretation text file of a fatbin file is always equal to 1 plus to the number of hardware registers assigned to each GPU thread.

The two previous checks verify that there is a bug in nvcc. Specifically, the two checks allow to speculate that: s_1) 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; s_2) that there is probably a one-to-one correspondence between the ELF registers and hardware registers of each GPU thread; and s_3) some hardware registers are probably not available to each GPU thread for the execution of the fatbin file, and these registers do not appear as ELF registers in the interpretation text file.

5.2 ELF → Hardware: Register Mapping

We verified that 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 that correspond to the different ELF registers visible in the interpretation text file of a fatbin file. Because of this, if the previous s_1 is not true, then at least one hardware register should correspond to at least two different ELF registers; if the previous s_2 is not true, then some hardware registers would correspond to multiple ELF registers; and if the previous s_3 is not true, then some hardware registers would correspond to multiple ELF registers as well.

The probability that s_1, s_2 and s_3 are not true is very small because, if some hardware registers can correspond to more than one ELF register of the same GPU thread, then there would be an additional overload in the determination of what warp each warp scheduler can schedule per warp scheduler's clock cycle. This is very unlikely considering that a maximum of 48 warps can reside at one time in each streaming multiprocessor during the execution of a fatbin file, and given that the 2 warp schedulers in each streaming multiprocessor already have to compete for the assignment of the warps.

In each interpretation text file we only get the interpretation of the ELF part of the fatbin file, which corresponds to a PTX file. In every fatbin file, before and after the aforementioned ELF part, there are therefore two other parts that are not interpreted by cuobjdump. Let us call these two uninterpreted parts A and C, and call the interpreted part B.

Let us assume that the previous s_1, s_2 and s_3 are true, and furthermore we know a) that during the execution of a fatbin file each thread has to execute the B part of the fatbin file in its entirety, b) that the interpretation text file of a fatbin file is unaffected by how many GPU threads are used to execute the fatbin file during a launch, and c) that the hardware registers of each GPU thread are private and cannot be used/read/written by other GPU threads during the execution of a fatbin file. Therefore in parts A and C of a fatbin file, nvcc must have generated some instructions that use both the parameters used in a launch configuration and the distribution of the GPU thread blocks to the streaming

multiprocessors after the beginning of the execution of the fatbin file as operands. In doing so, these instructions 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, while at the same time they avoid re-assigning the hardware registers to another GPU thread.

5.3 Missing ELF Instructions

By analyzing several hundred thousands interpretation text files, we found that the PTX instructions used to declare PTX registers in the PTX file, i.e. lines 4, 7, 9, 12, 15, 16, 18 and 20 in the PTX file for the mad.rz.f64.c.ne, do not have corresponding ELF instructions in the B part of the interpretation text file, i.e. no ELF instructions between the lines 6 and 7, 12 and 13, 15 and 16, 18 and 19, 19 and 20, 22 and 23, 25 and 26.

Table 5.1: Part of the Mad.rz.f64.c.ne PTX File

```
1
2  ..............................................................
3  DECLARATION OF THE REGISTER OF SUPPORT AND LOADING
4  .reg.u64 %ros ;                                  bar.sync 0 ;
5  ld.param.u64 %ros , [ aii ] ;                    bar.sync 0 ;
6  DECLARATION OF TWO AUXILIARY REGISTERS AND LOADINGS
7  .reg.s32 %reg_s32_au1 ;                          bar.sync 1 ;
8  ld.global.s32 %reg_s32_au1 , [ %ros +  8 ];      bar.sync 1 ;
9  .reg.s32 %reg_s32_au2 ;                          bar.sync 1 ;
10 ld.global.s32 %reg_s32_au2 , [ %ros + 12 ];      bar.sync 1 ;
11 DECLARATION OF A PREDICATE REGISTER AND ITS SETTING
12 .reg.pred %reg_p ;                               bar.sync 2 ;
13 setp.lt.s32 %reg_p , %reg_s32_au1 , %reg_s32_au2 ;  bar.sync 2 ;
14 DECLARATION OF THE REGISTERS OF THE PTX INSTRUCTION AND LOADINGS
15 .reg.f64 %reg_f64_0 ;                            bar.sync 3 ;
16 .reg.f64 %reg_f64_1 ;                            bar.sync 3 ;
17 ld.global.b64 %reg_f64_1 , [ %ros + 16 ] ;       bar.sync 3 ;
18 .reg.f64 %reg_f64_2 ;                            bar.sync 3 ;
19 ld.global.b64 %reg_f64_2 , [ %ros + 24 ] ;       bar.sync 3 ;
20 .reg.f64 %reg_f64_3 ;                            bar.sync 3 ;
21 ld.global.b64 %reg_f64_3 , [ %ros + 32 ] ;       bar.sync 3 ;
22 ..............................................................
```

Table 5.2: Part of the Interpretation Text File of the Mad.rz.f64.c.ne PTX File

```
1
2   ............................................................
3   /*0000*/    /*0x00005de428004404*/    MOV R1, c [0x1] [0x100];
4   /*0008*/    /*0xfc1fdc03207e0000*/    IMAD.U32.U32 RZ, R1, RZ, RZ;
5   /*0010*/    /*0xffffdc0450ee0000*/    BAR.RED.POPC RZ, RZ;
6   /*0018*/    /*0xffffdc0450ee0000*/    BAR.RED.POPC RZ, RZ;
7   /*0020*/    /*0xfc1fdc0450ee8000*/    BAR.RED.POPC RZ, 0x1;
8   /*0028*/    /*0x80001de428004000*/    MOV R0, c [0x0] [0x20];
9   /*0030*/    /*0x20029c034801c000*/    IADD R10.CC, R0, 0x8;
10  /*0038*/    /*0x93f2dc4348004000*/    IADD.X R11, RZ, c [0x0] [0x24];
11  /*0040*/    /*0x00a01c8584000000*/    LD.E R0, [R10];
12  /*0048*/    /*0xfc1fdc0450ee8000*/    BAR.RED.POPC RZ, 0x1;
13  /*0050*/    /*0xfc1fdc0450ee8000*/    BAR.RED.POPC RZ, 0x1;
14  /*0058*/    /*0x10a09c8584000000*/    LD.E R2, [R10+0x12];
15  /*0060*/    /*0xfc1fdc0450ee8000*/    BAR.RED.POPC RZ, 0x1;
16  /*0068*/    /*0xfc2fdc0450ee8000*/    BAR.RED.POPC RZ, 0x2;
17  /*0070*/    /*0x0801dc23188e0000*/    ISETP.LT.AND P0, pt, R0, R2, pt;
18  /*0078*/    /*0xfc2fdc0450ee8000*/    BAR.RED.POPC RZ, 0x2;
19  /*0080*/    /*0xfc3fdc0450ee8000*/    BAR.RED.POPC RZ, 0x3;
20  /*0088*/    /*0xfc3fdc0450ee8000*/    BAR.RED.POPC RZ, 0x3;
21  /*0090*/    /*0x20a11ca584000000*/    LD.E.64 R4, [R10+0x16];
22  /*0098*/    /*0xfc3fdc0450ee8000*/    BAR.RED.POPC RZ, 0x3;
23  /*00a0*/    /*0xfc3fdc0450ee8000*/    BAR.RED.POPC RZ, 0x3;
24  /*00a8*/    /*0x40a19ca584000000*/    LD.E.64 R6, [R10+0x24];
25  /*00b0*/    /*0xfc3fdc0450ee8000*/    BAR.RED.POPC RZ, 0x3;
26  /*00b8*/    /*0xfc3fdc0450ee8000*/    BAR.RED.POPC RZ, 0x3;
27  /*00c0*/    /*0x60a21ca584000000*/    LD.E.64 R8, [R10+0x32];
28  /*00c8*/    /*0xfc3fdc0450ee8000*/    BAR.RED.POPC RZ, 0x3;
29  ............................................................
```

This means that during the compiling phase, nvcc creates the ELF instructions used to declare the necessary ELF registers, but then inserts them in parts A and C of the generated fatbin file.

5.4 Generation Procedure: Fatbin Shells

Because a) we do not have ELF instructions to declare ELF registers, and because b) we do not know the instructions and the mechanism in the parts of a fatbin file used to assign

the hardware registers to the ELF registers, we need to develop a procedure to generate the PTX files that are compiled by nvcc to produce fatbin files with the number and type of ELF registers required to modify the B parts of the fatbin files.

If we are successful in this then we do not have any problem during the execution of the modified version of the fatbin file. This is because because we use the ELF registers that appear in the interpretation text file of the fatbin file to modify its B part. This happens because a) we only overwrite the B part of the fatbin file and because b) the B part has no jumps to parts A and C of the modified fatbin file, as was already the case for the original fatbin file.

For these reasons, the control has to be somehow passed to the beginning of the B part. This must occur independently of the mechanism used by the GPU architecture to execute a fatbin file. The B part will be executed in its entirety, without any jumps to parts A and C, as in the case of the original fatbin file. The overwritten B part will not therefore cause launch failures due to violations of the use of hardware resources (e.g. registers) that were not initially assigned to it.

An autonomic tool T_6 generates special PTX files. T_6 a priori knows a) the number and type of PTX instructions necessary to execute each of them, b) the number and type of ELF registers in the ELF instructions necessary to execute each PTX instruction, c) how many ELF registers, that appear in each ELF instruction, we want to reuse in the other ELF instructions that will execute the PTX instructions in the PTX file, and d) uses the editing guidelines described in the previous chapter.

Using these special PTX files, nvcc then produces fatbin files that have 1) the minimum number of ELF registers per type of ELF register that is necessary for the overwriting and modifications of the fatbin files, and that have 2) a greater number of ELF instructions in their B parts than the number of ELF instructions necessary to produce the desired ELF algorithmic implementations.

However, because the editing guidelines are based on some assumptions, even do such

assumptions are reasonable, the autonomic tool T_7 always checks each fatbin file generated to insure that 1) the number of ELF register types in the interpretation text file of the fatbin file is equal to the number of ELF register types that are necessary for the overwriting and modification of the fatbin file, and that 2) the B part of the fatbin file has more ELF instructions than the amount necessary for the modification of the fatbin file. If such checks for a fatbin file are not satisfied, then we can repeat the process, each time generating a PTX file with one more PTX register that corresponds to a remaining ELF register which is itself necessary to satisfy the checks.

5.5 Discovered Fatbin Hardware Constraints

Using these guidelines, we were always able to generate whatever type of fatbin file we wanted (and we generated millions of them). We also discovered the following: 1) the total number of ELF registers per fatbin file has to be smaller than 65; 2) the total number of ELF instructions that compose the B part of a fatbin file has to be smaller than 8193; c) the number of predicate ELF registers in a fatbin file has to be smaller than 9; and d) the ELF registers RZ, R0, R1, and pt have to be used in specific ELF instructions and in specific positions in these ELF instructions.

5.6 Summary

In this chapter we underlined some bugs and challenges we faced, and demonstrated that it is necessary to overcome them to be able to modify fatbin files and produce the desired ELF implementations. The most important points to remember from this chapter are the following:

- In the B part of an interpretation text file, there are no ELF instructions to assign ELF registers to a fatbin file. For this reason, the ELF registers assigned to a fatbin

file have to be assigned by nvcc during the compiling phase and later inserted by nvcc in part/s A and/or C 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, and so it must be in part/s A and/or C. This is because nvcc cannot know the launch configuration that a user will use for each execution of the fatbin file and so nvcc cannot know the number of GPU threads that will execute the fatbin file. The number of GPU threads could effectively be for each execution.

- To produce a desired ELF algorithmic implementation, T_6 a) generates a PTX file using the guidelines described in the previous chapter, b) gives the PTX file as input to nvcc to produce a fatbin file that, thanks to the use of the editing guidelines, has the number and type of ELF registers necessary for the modification of the B part of the fatbin file, and c) passes the fatbin file to T_7 to check for the correct number and type of ELF resources needed for modifying the fatbin file.

Chapter 6

ELF Kernels (B Parts of Fatbin Files)

To discover and quantify low level GPU architectural features and machine behaviors, it is important to stress specific aspects of the GPU architecture, and it is therefore necessary to produce thousands of different B parts of fatbin files (hereafter referred to as "ELF kernels").

Each ELF kernel has to be unique to the number, order and type of ELF instructions and registers in each ELF instruction. We experimentally verified that it is usually impossible to force nvcc to produce an ELF kernel with the specific number, order and type of ELF instructions and registers in each ELF instruction we want. For this reason, it is therefore necessary to first use nvcc to generate a fatbin file with the required hardware resources, i.e. the specific number and type of ELF registers, and next to overwrite its B part to produce the desired ELF kernel.

Furthermore, each ELF kernel has to be edited to provide an *a priori* guarantee that the byte transfers among different GPU memories will not slow down the ELF kernel's executions. This is important because, for the quantification of the architectural features, we need to have such a guarantee to ensure that the execution times of the ELF kernels are due only to the hardware limitations of the gigathread scheduler and the streaming multiprocessors (i.e. warp schedulers, instruction pipeline depths, waiting times due to write-read and read-read dependences among ELF registers, and the undisclosed hardware resources shared

among the possible subsets of functional units in each streaming multiprocessor).

6.1 *A Priori* Guarantees

Each ELF kernel cannot have more than 8192 ELF instructions, and therefore each ELF kernel must have a for loop. In this way we can further iterate on the for loop to make each GPU thread execute the desired number of ELF instructions.

The ELF instructions inside each for loop use some data. These data must be loaded before the beginning of the for loop. A written request to a specific location in the global memory and a memory synchronization barrier are both edited just before the beginning of the for loop. The GPU threads are released only after a) they have all met the synchronization barrier, and b) they all have satisfied all the previous memory requests, and therefore the writing request to the same global memory location. By doing this, we guarantee a) that all the necessary data, used inside the for loop, are in the ELF registers before the beginning of the for loop's execution, and b) that, during the for loop's execution, the execution of the ELF instructions cannot be slowed down by the byte transfers among GPU memories. Just after the synchronization barrier, each GPU thread reads the global GPU clock cycle and enters into the for loop. Later, just after the end of the for loop, each GPU thread reads the GPU clock cycle, executes a writing request to the same global memory location, and meets another memory synchronization barrier.

By structuring the ELF kernels in this way, it *a priori* guarantees that the execution times of the ELF kernels will not be influenced by the bandwidths and the latencies of the GPU memories. Therefore, the kernels are only influenced by the hardware limitations of the gigathread scheduler and the streaming multiprocessors.

Furthermore, we must consider a) that the worst case global memory latency is not greater than 800 functional unit clock cycles ([43, p. 87] and [46, p. 67] say that the worst case would be 800, while [42, p. 47] and [45, p. 57] argue for 600), and b) that the read and

write operations to the same global memory location are not atomic among GPU threads. As a result, this *a priori* guarantees that the memory synchronization barriers used in the ELF kernels will produce very little noise on the final calculation of the execution times of the for loops. This is because millions of ELF instructions are executed in each for loop.

6.2 ELF Kernels' Structures

All the ELF kernels have the same abstract structure: loading the data from the global memory into the hardware registers; overwriting a specific global memory location; synchronizing the threads; reading the machine global clock cycle contained in a hardware counter; running the for loop; reading the machine global clock cycle again; synchronizing the threads; and finally writing back the results to the global memory.

However, each ELF kernel has a different for loop because the order, number and type of ELF instructions and registers used in the ELF instructions are different for different PTX instructions.

A PTX case is a PTX instruction (e.g. sub.s32), that is executed in a particular way (e.g. conditional normal), and that considers a particular dependence type (e.g. read-read). A PTX sub-case is a PTX case with a particular dependence distance (e.g. 2) - see example below.

Table 6.1: PTX sub-case sub.s32.c.no.rr.2

```
The instruction considered in this case is the PTX instruction sub.s32,
in its conditional form (c), executed if the guard set is at true (no).
Its dependence type is read-read (rr) and its dependence distance is 2.

PTX File for the PTX sub-case sub.s32.c.no.rr.2:
.....................................................................
\\ loading data from the global memory into the hardware registers \\\\\\
\\ overwriting a specific global memory location \\\\\\\\\\\\\\\\\\\\\\
\\ synchronizing the threads \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\ reading the machine global clock cycle contained in a hardware counter
```

```
\\ BEGINNING FOR LOOP \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\ updating counter for loop iterations \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
@%guard_0 sub.s32 %res_0, %ope_0, %ope_0;  (instruction 1 - group 1)
@%guard_1 sub.s32 %res_1, %ope_1, %ope_1;  (instruction 2 - group 1)
@%guard_0 sub.s32 %res_0, %ope_0, %ope_0;  (instruction 1 - group 2)
@%guard_1 sub.s32 %res_1, %ope_1, %ope_1;  (instruction 2 - group 2)
@%guard_0 sub.s32 %res_0, %ope_0, %ope_0;  (instruction 1 - group 3)
@%guard_1 sub.s32 %res_1, %ope_1, %ope_1;  (instruction 2 - group 3)
.....................................................................
\\ updating guard for the repetition of the for loop \\\\\\\\\\\\\\\\\\
@%guard_0 sub.s32 %res_0, %ope_0, %ope_0;  (instruction 1 - group 1)
@%guard_1 sub.s32 %res_1, %ope_1, %ope_1;  (instruction 2 - group 1)
@%guard_0 sub.s32 %res_0, %ope_0, %ope_0;  (instruction 1 - group 2)
@%guard_1 sub.s32 %res_1, %ope_1, %ope_1;  (instruction 2 - group 2)
@%guard_0 sub.s32 %res_0, %ope_0, %ope_0;  (instruction 1 - group 3)
@%guard_1 sub.s32 %res_1, %ope_1, %ope_1;  (instruction 2 - group 3)
.....................................................................
\\ check predicate for loop repeating \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\ END FOR LOOP \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\ reading the machine global clock cycle \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\ synchronizing the threads \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\ writing back the results to the global memory \\\\\\\\\\\\\\\\\\\\\\\\
.....................................................................
```

The PTX code above shows groups of sub.s32 PTX instructions executed in a normal conditional mode - the guard has to be true. Each group has the same 2 sub.s32 PTX instructions - the name of the PTX registers determines whether 2 PTX instructions are equal. The dependence type between each pair of equal and subsequent sub.s32 PTX instructions is read-read, and the dependence distance is 2.

We need to minimize the noise given by a) the update 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. To do so, we a) place the update of the number of for cycles executed at the beginning of the for loop, b) place the setting of the guard in the middle of the for loop, and c) place the conditional jump at the end of the for loop.

For each PTX sub-case, we always produce 2 fatbin files. The first is produced using nvcc.

This first file is used, in chapter X, to compare the performance achievable programming using PTX, which exploits nvcc for the optimization and translation of PTX to ELF codes, with the performance achievable programming using ELF, which exploits the insight about the low level GPU architectural features and machine behaviors produced in chapter Y. This second file is produced by a set of autonomic tools that we implemented but do not describe here. This set of autonomic tools overwrites the B part of the first fatbin file and executes many checks and corrections, if necessary, to guarantee that the overwriting is successful. This second file is used in chapter Y to discover and quantify the low level GPU architectural features and machine behaviors.

Furthermore, because many single PTX instructions require the execution of a set of ELF instructions, the set of autonomic tools also produces an ELF kernel for each ELF instruction case. This allows us to discover and quantify low level GPU architectural features and machine behaviors tied to a single ELF instruction.

6.3 Autonomic ELF Kernel Generation

The ELF kernels must be of limited length (less than 8193 ELF instructions), the number of ELF registers per ELF kernel must be smaller than 65, and a limited number and type of ELF registers are necessary to execute the ELF instructions, which are then necessary to execute a PTX instruction. For these reasons, for each PTX case it is possible to automatically generate all the ELF kernels that are necessary to quantify the low level GPU architectural features and machine behaviors.

Table 6.2: Generation of ELF Kernels for the add.u64.n.wr PTX Case

```
PTX Instruction:     add.u64 %res, %op1, %op2
PTX Registers:       registers at 64 bits
Total PTX Registers: 3

ELF Instructions:    IADD R10.CC, R23, R34
```

```
                     IADD.X R11, R24, R35
ELF Registers:       registers at 32 bits
Total ELF Registers: 6

PTX Registers   ELF Registers
%res            R10, R11
%op1            R23, R24
%op2            R34, R35

PTX Instruction    ELF Instructions
add.u64            IADD (.CC type 1)
                   IADD (.X  type 2)
```

Each ELF Kernel has a maximum of 64 hardware registers (hardware limit per GPU thread). Let us suppose we need 13 normal ELF registers to execute PTX instructions, which are different from the add.u64 PTX instructions that compose the for loop of the ELF kernels, to generate the PTX sub-cases. Given the 3 normal reserved ELF registers RZ, R0 and R1, this means that we have a maximum of 64 - (13 + 3) = 64 - 16 = 48 normal ELF registers per thread to execute the add.u64 PTX instructions into the for loop.

```
PTX Case:            add.u64.n.wr
PTX Instruction:     add.u64 %res, %res, %res
PTX Registers:       registers at 64 bits
Total PTX Registers: 1

ELF Instructions:    IADD R2.CC, R2, R2
                     IADD.X R3, R3, R3
ELF Registers:       registers at 32 bits
Total ELF Registers: 2

PTX Registers   ELF Registers
%res            R2, R3
```

The execution of an add.u64 PTX instruction, when executed in normal mode with a write-read dependence, requires 2 32-bit ELF registers. This means that the autonomic tools will generate 48 / 2 = 24 different ELF kernels for the PTX case add.u64.n.wr, with one ELF kernel per dependence distance (see D below).

```
D.  PTX Instructions Per Group      Corresponding ELF Instructions

01  add.u64 %rao01,%rao01,%rao01;   IADD R2.CC, R2, R2; IADD.X R3, R3, R3;

02  add.u64 %rao01,%rao01,%rao01;   IADD R2.CC, R2, R2; IADD.X R3, R3, R3;
```

```
    add.u64 %rao02,%rao02,%rao02;   IADD R4.CC, R4, R4; IADD.X R5, R5, R5;

03  add.u64 %rao01,%rao01,%rao01;   IADD R2.CC, R2, R2; IADD.X R3, R3, R3;
    add.u64 %rao02,%rao02,%rao02;   IADD R4.CC, R4, R4; IADD.X R5, R5, R5;
    add.u64 %rao03,%rao03,%rao03;   IADD R6.CC, R6, R6; IADD.X R7, R7, R7;

..  ............................   ....................................
```

Let us suppose we dedicate a total of 6003 ELF instructions to the for loop of each ELF kernel. Excluding the 3 ELF instructions necessary to iterate on the for loops, this leaves 6000 instructions per for loop.

The first ELF kernel (i.e. the kernel generated for the dependence distance 01) will have a floor of (6000 / 2) = 3000 groups inside its for loop. Each group will be composed of 2 consecutive ELF instructions: IADD R2.CC, R2, R2 and IADD.X R3, R3, R3.

The second ELF kernel (i.e. the kernel generated for the dependence distance 02) will have a floor of (6000 / 4) = 1500 groups inside its for loop. Each group will be composed of 4 consecutive ELF instructions: IADD R2.CC, R2, R2, IADD.X R3, R3, R3, IADD R4.CC, R4, R4, and IADD.X R5, R5, R5.

Etc.

```
ELF Kernel 01                                              Group Numbers

....................................
\ BEGINNING FOR LOOP \\\\\\\\\\\\\\\\
\ updating counter for loop iterations
IADD R2.CC, R2, R2; IADD.X R3, R3, R3;                             1
IADD R4.CC, R4, R4; IADD.X R5, R5, R5;                             1
IADD R2.CC, R2, R2; IADD.X R3, R3, R3;                             2
IADD R4.CC, R4, R4; IADD.X R5, R5, R5;                             2
....................................                               .
IADD R2.CC, R2, R2; IADD.X R3, R3, R3;                          1499
IADD R4.CC, R4, R4; IADD.X R5, R5, R5;                          1499
IADD R2.CC, R2, R2; IADD.X R3, R3, R3;                          1500
IADD R4.CC, R4, R4; IADD.X R5, R5, R5;                          1500
\ setting predicate for loop repeating                           ...
IADD R2.CC, R2, R2; IADD.X R3, R3, R3;                          1501
IADD R4.CC, R4, R4; IADD.X R5, R5, R5;                          1501
IADD R2.CC, R2, R2; IADD.X R3, R3, R3;                          1502
IADD R4.CC, R4, R4; IADD.X R5, R5, R5;                          1502
....................................                             ...
IADD R2.CC, R2, R2; IADD.X R3, R3, R3;                          2999
```

```
IADD R4.CC, R4, R4; IADD.X R5, R5, R5;                                  2999
IADD R2.CC, R2, R2; IADD.X R3, R3, R3;                                  3000
IADD R4.CC, R4, R4; IADD.X R5, R5, R5;                                  3000
\ check predicate for loop repeating
\ END FOR LOOP \\\\\\\\\\\\\\\\
.....................................
```

6.4 Autonomic Launch Configuration Generation

Each ELF kernel requires a different number of hardware registers. The hardware limits of the GPU and the hardware registers of an ELF kernel determine the launch configurations that have to be used to execute that kernel, in order to discover and quantify the low level GPU architectural features and machine behaviors.

Table 6.3: Discovered or Disclosed GPU Hardware Limits

```
Maximum thread blocks per stream multiprocessor:                 8
Maximum number of warps per thread block:                        32
Maximum number of resident warps per stream multiprocessor:      48
Maximum number of hardware registers per resident thread:        64
Maximum number of predicate ELF registers per resident thread:   8
Number of 32 bits hardware registers per stream multiprocessor:  32756
```

Because there is a potential maximum number of resident warps, we can choose many different launch configurations. We therefore decided to choose the simplest launch configurations to stress the local hardware components of a stream multiprocessor. Furthermore, during the experiments, it became evident that for local stresses, a maximum of 32 warps per thread block was more than enough to correctly discover and quantify all the undisclosed low level GPU architectural features and machine behaviors.

Table 6.4: Maximum Number of Resident Warps per Stream Multiprocessor

```
PTX Case: add.u64.n.wr
```

Table for the ELF kernels:

Distance	Hardware Registers	Max Resident Warps	Maximum Chosen
01	13 + 3 + 2* 1 = 18	min(48 , 56) = 48	min(32 , 48) = 32
02	13 + 3 + 2* 2 = 20	min(48 , 51) = 48	min(32 , 48) = 32
03	13 + 3 + 2* 3 = 22	min(48 , 46) = 46	min(32 , 46) = 32
04	13 + 3 + 2* 4 = 24	min(48 , 42) = 42	min(32 , 42) = 32
05	13 + 3 + 2* 5 = 26	min(48 , 39) = 39	min(32 , 39) = 32
06	13 + 3 + 2* 6 = 28	min(48 , 36) = 36	min(32 , 36) = 32
07	13 + 3 + 2* 7 = 30	min(48 , 34) = 34	min(32 , 34) = 32
08	13 + 3 + 2* 8 = 32	min(48 , 32) = 32	min(32 , 32) = 32
09	13 + 3 + 2* 9 = 34	min(48 , 30) = 30	min(32 , 30) = 30
10	13 + 3 + 2*10 = 36	min(48 , 28) = 28	min(32 , 28) = 28
11	13 + 3 + 2*11 = 38	min(48 , 26) = 26	min(32 , 26) = 26
12	13 + 3 + 2*12 = 40	min(48 , 25) = 25	min(32 , 25) = 25
13	13 + 3 + 2*13 = 42	min(48 , 24) = 24	min(32 , 24) = 24
14	13 + 3 + 2*14 = 44	min(48 , 23) = 23	min(32 , 23) = 23
15	13 + 3 + 2*15 = 46	min(48 , 22) = 22	min(32 , 22) = 22
16	13 + 3 + 2*16 = 48	min(48 , 21) = 21	min(32 , 21) = 21
17	13 + 3 + 2*17 = 50	min(48 , 20) = 20	min(32 , 20) = 20
18	13 + 3 + 2*18 = 52	min(48 , 19) = 19	min(32 , 19) = 19
19	13 + 3 + 2*19 = 54	min(48 , 18) = 18	min(32 , 18) = 18
20	13 + 3 + 2*20 = 56	min(48 , 18) = 18	min(32 , 18) = 18
21	13 + 3 + 2*21 = 58	min(48 , 17) = 17	min(32 , 17) = 17
22	13 + 3 + 2*22 = 60	min(48 , 17) = 17	min(32 , 17) = 17
23	13 + 3 + 2*23 = 62	min(48 , 16) = 16	min(32 , 16) = 16
24	13 + 3 + 2*24 = 64	min(48 , 16) = 16	min(32 , 16) = 16

For the discovery and quantification of the local low level GPU architectural features and machine behaviors, we therefore selected only the launch configurations with 1 thread block, and a number of warps per thread block ranging from 1 to minimum(32 , maximum number of possible resident warps for the ELF kernel), for the use with each ELF kernel.

6.5 Summary

In this chapter we described the procedure used to generate the ELF kernels necessary for the discovery and quantification of the low level GPU architectural features and machine behaviors. The most important points to remember from this chapter are the following:

- The autonomic tools generate all the necessary ELF kernels and a priori guarantee that their executions will not be slowed down by the bandwidths or latencies of the GPU memories.

- The synchronization among threads introduces a small quantity of noise that will not have any meaningful effect on the quantification of the low level GPU architectural features and machine behaviors. This is because each single thread will execute millions of ELF instructions for each run.

Chapter 7

Undisclosed Features and Behaviors

Discovering, understanding, and quantifying the undisclosed GPU low level architectural features and machine behaviors is important to eliminate or reduce time consuming trials and errors in optimization process and to effectively drive code design.

These features and behaviors are: 1) the gigathread scheduler's thread block assignments to the streaming multiprocessors; 2) the warp schedulers' policies; 3) the maximum number of PTX or ELF instructions that a streaming multiprocessor can execute per functional unit clock cycle (this per type of PTX or ELF instruction); 4) the warp instructions' latencies, which determine how many functional unit clock cycles are necessary before we can re-use the registers used in a PTX or ELF instruction; and 5) the warps' latencies, which determine the number of functional unit clock cycles that are needed before rescheduling a warp.

7.1 Thread Block Assignments

There are no PTX or ELF instructions that allow users to assign a thread block from a launch configuration to a specific streaming multiprocessor. The assignment of the thread blocks is executed by the gigathread scheduler. Furthermore, the algorithm used by the gigathread scheduler is undisclosed. However, discovering the assignments and understanding the parameters that influence them is important to effectively optimize codes.

We know that the hardware registers assigned to each thread cannot be used to exchange data with other threads, and that thread blocks cannot migrate from a streaming multiprocessor to another streaming multiprocessor after the gigathread scheduler has distributed them. If, for example, a code that requires the exchange of data is designed assuming that 2 thread blocks will reside in the same streaming multiprocessor, but this is not the case during the code executions, then a greater quantity of data will be exchanged between the 2 L1 caches of the 2 streaming multiprocessors where the 2 thread blocks reside.

However, the data exchange among the L1 caches of the streaming multiprocessors and the L2 cache shared by all the streaming multiprocessors can easily nullify the other performance optimizations integrated in a code. This happens because: a) a maximum of 8 thread blocks can reside in a streaming multiprocessor; b) a maximum of 48 warps can reside in a streaming multiprocessor; and c) the minimum number of streaming multiprocessors in a Tesla C2070 is 14.

To effectively optimize codes (i.e. to better exploit data locality) it is therefore important to study the assignments and understand which parameters influence them. However, discovering and analyzing the thread block distributions is not only important to understand where each thread block will reside during the code execution, but also to verify if there are cases when the gigathread scheduler does not fairly assign thread blocks to the streaming multiprocessors (i.e. if it is possible that there is a greater number of thread blocks in some streaming multiprocessors).

Verifying the gigathread scheduler's fair distribution of thread blocks to the streaming multiprocessors is important, since a fair distribution is a necessary condition to make a code 100% efficient, and therefore to make its executions as short as possible. However, we found that the gigathread scheduler does not always fairly assign the thread blocks to the streaming multiprocessors. This was experimentally verified using thousands of runs for thousands of micro-benchmark kernels.

We know that each thread block of a launch configuration must have the same number

of threads and each thread has the same number and types of hardware registers. This is because each thread executes the entire fatbin file, and the number of hardware registers assigned to the fatbin file is determined by nvcc during the fatbin file generation.

Let us define the number of threads of a thread block n_t^{tb} and the number of hardware registers of each thread n_{hr}^t. A thread block can be assigned to a streaming multiprocessor if and only if the total number of hardware registers necessary for the thread block (n_{hr}^{tb}) is smaller than the total number of hardware registers of a streaming multiprocessor n_{hr}^{sm} (this number is equal to 32768).

We advise against the use of any launch configuration with a number of thread blocks n_{tb} greater than the number of streaming multiprocessors n^{sm}, and a total number of hardware registers per thread block greater than half the number of hardware registers of a streaming multiprocessor. This is because, for these cases, some thread blocks cannot be assigned at the beginning of the code execution, and therefore the compiler's optimization job becomes harder if different thread blocks need to share data during the executions (i.e. the gigathread scheduler should repeatedly switch and preempt thread blocks and the compiler could have to heavily modify the code because shared data would have to be temporarily saved somewhere else). Furthermore, for such configurations, we have experimentally verified that the assignment of the thread blocks is not deterministic, and so the thread blocks are usually assigned to different streaming multiprocessor at each code execution. This further increases the difficulty of efficiently exploiting data locality.

Better launch configurations are those with: 1) the number of thread blocks being an integer factor of the number of streaming multiprocessors (e.g. 42 thread blocks $= 3 \cdot 14$ streaming multiprocessors); and 2) the total number of hardware registers per group of thread blocks that we want to reside in a streaming multiprocessor (e.g. $3 \cdot n_{hr}^{tb}$) greater than half the number of hardware registers of a streaming multiprocessor $\frac{n_{hr}^{sm}}{2}$, and also smaller than the total number of hardware registers of a streaming multiprocessor n_{hr}^{sm}.

For these launch configurations the following is true: 1) the compiler's optimization job

is easier (i.e. it is not necessary to heavily modify the code or to synchronize the threads like before); and 2) the gigathread scheduler does not need to switch and preempt thread blocks during code executions. However, for these launch configurations, if the threads of a thread block need to exchange data, then the shared memory of each streaming multiprocessor needs to be partitioned among the number of thread blocks that reside in each streaming multiprocessor. With a maximum number of 8 thread blocks per streaming multiprocessor, and 6 warps per thread block, this would create very small 6 KB partitions for data exchanges.

The set of best launch configurations are those with: 1) the number of thread blocks equal to the number of streaming multiprocessors; and 2) the number of hardware registers per thread block greater than half the number of hardware registers of a streaming multiprocessor $\frac{n_{hr}^{sm}}{2}$, and also smaller than the total number of hardware registers of a streaming multiprocessor n_{hr}^{sm}. For these launch configurations, we experimentally determined that the assignments are deterministic and therefore at each run the same thread block will be assigned to the same streaming multiprocessor. Furthermore, for these launch configurations, all the thread blocks are assigned by the gigathread scheduler to the streaming multiprocessors at the beginning of the code executions.

The next set of best launch configurations are those with: 1) the number of thread blocks that is twice the number of streaming multiprocessors; and 2) the number of hardware registers necessary per group of 2 thread blocks that is greater than half the number of hardware registers of a streaming multiprocessor $\frac{n_{hr}^{sm}}{2}$, and also smaller than the total number of hardware registers of a streaming multiprocessor n_{hr}^{sm}. We experimentally found that for these launch configurations, as in the previous one, the assignments of the thread blocks are deterministic and therefore that at each run the same thread block will be assigned to the same streaming multiprocessor and all the thread blocks will be assigned to the streaming multiprocessors at the beginning of the code executions.

The launch configurations of this last set are useful because the maximum number of

warps per thread block must be 32, but the maximum number of warps that can reside in a streaming multiprocessor is 48. If the user uses between 33 and 48 warps per streaming multiprocessor then the user need to use one of the launch configurations of this set. This will provide: 1) the a priori guarantee that the gigathread scheduler will fairly distribute the thread blocks to the streaming multiprocessor; 2) the a priori guarantee that the assignments will be executed only at the beginning of the code executions; and 3) the location of each thread block.

Different fatbin files usually have a different number of hardware registers, and therefore the set of best launch configurations for a fatbin file is usually different from the set of best launch configurations from a different fatbin file. For example, let us consider a Tesla C2070 with 14 streaming multiprocessors and 2 fatbin files: ff_1 with 64 hardware registers, and ff_2 with 43 hardware registers. The best launch configurations for ff_1 are those that have 14 thread blocks and a number of warps per thread block ranging from 9 to 16 (these launch configurations are therefore a total of 8). In contrast, the best launch configurations for $ff2$ are those that have 14 thread blocks and a number of warps per thread block ranging from 17 to 32 (these launch configurations are therefore a total of 16).

Table 7.1: Legend for the thread block distribution tables.

```
-  LC: launch configuration        -  TB: number of thread blocks of the LC
-  LI: launch identifier           - WPB: number of warps per thread block
-  IC: instruction configuration   - RPT: hardware registers per thread
-  DD: dependence distance         - BXY: thread block identifier

The cells of the tables represent the identifier of the streaming multipro-
cessors where the blocks of threads are located. The Tesla C2070 has 14 st-
reaming multiprocessors, and therefore their identifiers range from 0 to 13
```

Table 7.2: Examples of not fair and deterministic thread block distributions

All of the following fatbin file-launch configuration couples have 14 thread blocks and require 4800 hardware registers per thread block (there are 14 streaming multiprocessors in a Tesla C2070, and 4800 is smaller than half the number of hardware registers of a streaming multiprocessor). The autonomic tools discovered that for launch configurations with a number of thread blocks that is equal to the number of streaming multiprocessors, and with a number of hardware registers per thread block that is smaller than half the number of hardware registers of a streaming multiprocessor, the gigathread scheduler does not fairly assign the thread blocks to the streaming multiprocessors. Furthermore, the not fair distribution is deterministic and it is independent of how many times we launch a fatbin file and of the type of ELF instructions that compose the fatbin file.

IC	DD	TB	WPB	RPT	B00	B01	B02	B03	B04	B05	B06	B07	B08	B09	B10	B11	B12	B13
add.u64.n.w	7	14	6	25	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
add.u64.n.w	1	14	10	15	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
min.s32.n.w	16	14	5	30	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
mul.lo.u32.n.w	3	14	10	15	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
or.b64.n.w	7	14	6	25	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
setp.gt.u32.n.w	39	14	3	50	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
sqrt.rn.ftz.f32.n.w	11	14	6	25	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
rem.u16.n.w	15	14	3	50	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
ex2.approx.f32.n.w	38	14	3	50	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
abs.s64.n.w	19	14	3	50	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
neg.s32.n.w	14	14	6	25	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
div.full.f32.n.w	1	14	10	15	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
rcp.rp.f32.n.w	8	14	6	25	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
max.u64.n.r	5	14	5	30	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
mul.wide.u32.n.r	5	14	6	25	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
neg.s64.c.no.w	10	14	3	50	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
or.b32.c.no.r	1	14	10	15	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
and.b32.c.ne.r	5	14	6	25	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
mul.wide.u32.c.ne.r	8	14	3	50	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
setp.gt.s64.c.ne.r	1	14	10	15	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
xor.b32.c.ne.r	5	14	6	25	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
abs.s16.c.ne.r	5	14	6	25	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**
not.b16.c.ne.r	5	14	6	25	**000**	**002**	003	004	006	007	008	009	010	011	012	003	**000**	**002**

Table 7.3: First example of fair but not deterministic thread block distributions

The ELF kernel for the PTX and.b32.n.w instruction configuration with a dependence distance of 14 is shown below. The kernel is run 10 times using 28 thread blocks and 32 warps per thread block. Each thread has 30 hardware registers; each thread block therefore requires 26880 hardware registers. Even do 26880 is greater than half of 32768 (the number of hardware registers of a streaming multiprocessor), the autonomic tools discovered that the distribution is not deterministic. This is because the launch configurations use a number of thread blocks that is twice the number of streaming multiprocessors. However, the autonomic tools discovered that the distributions are fair. This is because each thread block requires 26880 hardware registers and therefore no more than one thread block can reside in a streaming multiprocessor at each moment throughout the executions.

L	IC	DD	TB	WPB	RPT	B00	B01	B02	B03	B04	B05	B06	B07	B08	B09	B10	B11	B12	B13
1	and.b32.n.w	14	28	32	30	000	004	008	011	002	006	009	012	003	007	010	013	001	005
2	and.b32.n.w	14	28	32	30	000	004	008	011	002	006	009	012	003	007	010	013	001	005
3	and.b32.n.w	14	28	32	30	000	004	008	011	002	006	009	012	003	007	010	013	001	005
4	and.b32.n.w	14	28	32	30	000	004	008	011	002	006	009	012	003	007	010	013	001	005
5	and.b32.n.w	14	28	32	30	000	004	008	011	002	006	009	012	003	007	010	013	001	005
6	and.b32.n.w	14	28	32	30	000	004	008	011	002	006	009	012	003	007	010	013	001	005
7	and.b32.n.w	14	28	32	30	000	004	008	011	002	006	009	012	003	007	010	013	001	005
8	and.b32.n.w	14	28	32	30	000	004	008	011	002	006	009	012	003	007	010	013	001	005
9	and.b32.n.w	14	28	32	30	000	004	008	011	002	006	009	012	003	007	010	013	001	005
10	and.b32.n.w	14	28	32	30	000	004	008	011	002	006	009	012	003	007	010	013	001	005

L	IC	DD	TB	WPB	RPT	B14	B15	B16	B17	B18	B19	B20	B21	B22	B23	B24	B25	B26	B27
1	and.b32.n.w	14	28	32	30	001	005	010	013	007	006	009	012	003	000	004	008	011	002
2	and.b32.n.w	14	28	32	30	005	001	009	013	010	012	003	007	000	006	004	008	011	002
3	and.b32.n.w	14	28	32	30	005	001	013	010	003	007	012	008	002	009	004	011	006	000
4	and.b32.n.w	14	28	32	30	001	005	010	007	013	009	003	004	006	000	012	011	008	002
5	and.b32.n.w	14	28	32	30	001	005	013	010	009	007	003	002	012	004	008	011	006	000
6	and.b32.n.w	14	28	32	30	001	005	007	010	012	003	013	002	009	004	008	011	006	000
7	and.b32.n.w	14	28	32	30	001	005	007	010	013	003	002	006	009	012	000	004	008	011
8	and.b32.n.w	14	28	32	30	001	005	010	013	009	007	003	004	011	002	006	012	000	008
9	and.b32.n.w	14	28	32	30	001	005	009	007	013	010	002	006	004	003	000	008	011	012
10	and.b32.n.w	14	28	32	30	005	001	010	006	007	013	003	004	002	012	000	008	011	009

Table 7.4: Second example of fair but not deterministic thread block distributions

The ELF kernel for the PTX max.s64.c.no.w instruction configuration with a dependence distance of 27 is shown below. The kernel is run 10 times using 28 thread blocks and 10 warps per thread block. Each thread has 64 hardware registers; each thread block therefore requires 20480 hardware registers. Even do 20480 is greater than half of 32768 (the number of hardware registers of a streaming multiprocessor), the autonomic tools discovered that the distribution is not deterministic. This is because the launch configurations use a number of thread blocks that is twice the number of streaming multiprocessors. However, the autonomic tools discovered that the distributions are fair. This is because each thread block requires 20480 hardware registers and therefore no more than one thread block can reside in a streaming multiprocessor at each moment throughout the executions.

L	IC	DD	TB	WPB	RPT	B00	B01	B02	B03	B04	B05	B06	B07	B08	B09	B10	B11	B12	B13
1	max.s64.c.no.w	27	28	10	64	000	004	008	011	002	006	009	012	003	007	010	013	001	005
2	max.s64.c.no.w	27	28	10	64	000	004	008	011	002	006	009	012	003	007	010	013	001	005
3	max.s64.c.no.w	27	28	10	64	000	004	008	011	002	006	009	012	003	007	010	013	001	005
4	max.s64.c.no.w	27	28	10	64	000	004	008	011	002	006	009	012	003	007	010	013	001	005
5	max.s64.c.no.w	27	28	10	64	000	004	008	011	002	006	009	012	003	007	010	013	001	005
6	max.s64.c.no.w	27	28	10	64	000	004	008	011	002	006	009	012	003	007	010	013	001	005
7	max.s64.c.no.w	27	28	10	64	000	004	008	011	002	006	009	012	003	007	010	013	001	005
8	max.s64.c.no.w	27	28	10	64	000	004	008	011	002	006	009	012	003	007	010	013	001	005
9	max.s64.c.no.w	27	28	10	64	000	004	008	011	002	006	009	012	003	007	010	013	001	005
10	max.s64.c.no.w	27	28	10	64	000	004	008	011	002	006	009	012	003	007	010	013	001	005

L	IC	DD	TB	WPB	RPT	B14	B15	B16	B17	B18	B19	B20	B21	B22	B23	B24	B25	B26	B27
1	max.s64.c.no.w	27	28	10	64	005	001	003	007	013	010	002	000	004	008	011	009	006	012
2	max.s64.c.no.w	27	28	10	64	005	001	013	007	003	010	002	006	008	004	011	000	009	012
3	max.s64.c.no.w	27	28	10	64	001	005	003	007	010	006	013	011	002	004	008	009	012	000
4	max.s64.c.no.w	27	28	10	64	001	005	003	013	010	007	008	000	011	002	009	006	012	004
5	max.s64.c.no.w	27	28	10	64	001	005	013	003	010	007	002	004	011	009	008	006	012	000
6	max.s64.c.no.w	27	28	10	64	005	001	010	003	013	007	002	009	012	006	008	000	004	011
7	max.s64.c.no.w	27	28	10	64	005	001	003	007	010	013	009	004	008	000	012	011	002	006
8	max.s64.c.no.w	27	28	10	64	005	001	007	010	003	013	002	004	011	009	012	000	006	008
9	max.s64.c.no.w	27	28	10	64	005	001	003	013	010	007	009	008	002	012	004	011	006	000
10	max.s64.c.no.w	27	28	10	64	005	001	003	007	010	013	000	002	004	008	011	006	009	012
10	max.s64.c.no.w	27	28	10	64	000	004	008	011	002	006	009	012	003	007	010	013	001	005

Table 7.5: Examples of fair and deterministic thread block distributions

All of the following fatbin file-launch configuration couples have 14 thread blocks and require 22464 hardware registers per thread block (there are 14 streaming multiprocessors in a Tesla C2070 and 226464 is greater than half the number of hardware register of a streaming multiprocessor). The autonomic tools discovered that for launch configurations with a number of thread blocks that is equal to the number of streaming multiprocessors, and with a number of hardware registers per thread block that is greater than half the number of hardware registers of a streaming multiprocessor, the gigathread scheduler fairly assigns the thread blocks to the streaming multiprocessors. Furthermore, the distribution is deterministic and it is independent of how many times we launch a fatbin file and of the type of ELF instructions that compose the fatbin file.

IC	DD	TB	WPB	RPT	B00	B01	B02	B03	B04	B05	B06	B07	B08	B09	B10	B11	B12	B13
add.s32.n.w	12	14	27	26	000	002	003	004	006	007	008	001	009	010	011	005	012	013
add.s64.n.w	8	14	27	26	000	002	003	004	006	007	008	001	009	010	011	005	012	013
popc.b32.n.w	30	14	13	54	000	002	003	004	006	007	008	001	009	010	011	005	012	013
max.u32.n.w	13	14	27	26	000	002	003	004	006	007	008	001	009	010	011	005	012	013
mul.lo.s64.n.w	8	14	18	39	000	002	003	004	006	007	008	001	009	010	011	005	012	013
or.b32.n.w	12	14	26	27	000	002	003	004	006	007	008	001	009	010	011	005	012	013
setp.gt.u32.n.w	43	14	13	54	000	002	003	004	006	007	008	001	009	010	011	005	012	013
sqrt.rn.f32.n.w	25	14	18	39	000	002	003	004	006	007	008	001	009	010	011	005	012	013
fma.rm.f32.n.w	20	14	18	39	000	002	003	004	006	007	008	001	009	010	011	005	012	013
rem.u64.n.w	9	14	13	54	000	002	003	004	006	007	008	001	009	010	011	005	012	013
div.full.f32.n.w	7	14	26	27	000	002	003	004	006	007	008	001	009	010	011	005	012	013
ex2.approx.f32.n.w	27	14	18	39	000	002	003	004	006	007	008	001	009	010	011	005	012	013
mad.rz.f64.n.r	11	14	13	54	000	002	003	004	006	007	008	001	009	010	011	005	012	013
fma.rm.f64.n.r	4	14	27	26	000	002	003	004	006	007	008	001	009	010	011	005	012	013
rem.u32.n.r	9	14	18	39	000	002	003	004	006	007	008	001	009	010	011	005	012	013
neg.s64.n.r	4	14	27	26	000	002	003	004	006	007	008	001	009	010	011	005	012	013
lg2.approx.f32.n.r	4	14	27	26	000	002	003	004	006	007	008	001	009	010	011	005	012	013
min.u32.c.no.w	14	14	18	39	000	002	003	004	006	007	008	001	009	010	011	005	012	013
xor.b64.c.no.w	11	14	13	54	000	002	003	004	006	007	008	001	009	010	011	005	012	013
mul.lo.u32.c.no.r	5	14	27	26	000	002	003	004	006	007	008	001	009	010	011	005	012	013
and.b32.c.ne.w	14	14	18	39	000	002	003	004	006	007	008	001	009	010	011	005	012	013
add.rz.f64.c.ne.w	4	14	26	27	000	002	003	004	006	007	008	001	009	010	011	005	012	013
setp.lt.u32.c.ne.r	28	14	18	39	000	002	003	004	006	007	008	001	009	010	011	005	012	013

Table 7.6: Legend to determine the best launch configurations

```
Legend:                              SM's hardware constraints:
- TB: thread blocks                  - max 8 thread blocks
- SM: streaming multiprocessor       - max 48 warps
-  W: warps                          - max 64 registers per thread
- HR: hardware registers             - max 48 KB of shared memory
- SH: shared memory                  - max 32 warps per thread block
- LC: launch configuration           - max 32768 hardware registers
                                     - half max equal to 16384 (registers)
```

Table 7.7: Example to determine the best launch configurations

```
Example for the determination of the best launch configurations:

let's assume we have:
- 14 streaming multiprocessors and 1 SH per streaming multiprocessor
- the SH of each streaming multiprocessor set to 48 KB before the runs
- some LCs (number of thread blocks , number of warps per thread block)

let's consider 2 diff. fatbin files (FFs) with the following features:
- FF_1 = 45 hardware registers and 12 Kb of SH per thread block
- FF_2 = 54 hardware registers and 23 Kb of SH per thread block
- the FFs do not require the exchange data among different thread blocks
- the threads in a thread block can exchange data using the allocated SM
```

Table 7.8: Procedure to determine the best launch configurations

```
Procedure to determine the launch configurations of FF_1 and FF_2 that:
a) produce a fair distribution (equal number of thread blocks per SM)
b) force the thread block assignment at the beginning of the FF executions

-------------------------------------------------------------------------

FF_1 for the hardware registers:
launch configurations  TB per SM       W per SM      Hardware registers
( 14, from 12 to 22 )          1  from 12 to 22     from 17280 to 31680
( 28, from  6 to 11 )          2  from 12 to 22     from 17280 to 31680
( 42, from  4 to  7 )          3  from 12 to 21     from 17280 to 30240
```

```
( 56, from  3 to  5 )          4  from 12 to 20     from 17280 to 28800
( 70, from  3 to  4 )          5  from 15 to 20     from 21600 to 28800
( 84, from  2 to  3 )          6  from 12 to 18     from 17280 to 25920
( 98, from  2 to  3 )          7  from 14 to 21     from 20160 to 30240
(112, from  2 to  2 )          8  from 16 to 16     from 23040 to 23040

FF_1 for the shared memory: we cannot have more than 4 thread blocks
per SM because each thread block requires 12 KB of shared memory and
the shared memory is set at 48 KB.

Final FF_1 launch configurations (see below) that: a) produce a fair
distribution (equal number of thread blocks per SM); and b) force the
thread block assignment at the beginning of the FF executions.

launch configurations  TB per SM       W per SM   Hardware registers  SH
( 14, from 12 to 22 )          1  from 12 to 22  from 17280 to 31680  12
( 28, from  6 to 11 )          2  from 12 to 22  from 17280 to 31680  24
( 42, from  4 to  7 )          3  from 12 to 21  from 17280 to 30240  36
( 56, from  3 to  5 )          4  from 12 to 20  from 17280 to 28800  48

-------------------------------------------------------------------------

FF_2 for the hardware registers:
launch configurations  TB per SM       W per SM      Hardware registers
( 14, from 10 to 18 )          1  from 10 to 18     from 17280 to 31104
( 28, from  5 to  9 )          2  from 10 to 18     from 17280 to 31104
( 42, from  4 to  6 )          3  from 12 to 18     from 20736 to 31104
( 56, from  3 to  4 )          4  from 12 to 16     from 20736 to 27648
( 70, from  2 to  3 )          5  from 10 to 15     from 17280 to 25920
( 84, from  2 to  3 )          6  from 12 to 18     from 20736 to 31104
( 98, from  2 to  2 )          7  from 14 to 14     from 24192 to 24192
(112, from  2 to  2 )          8  from 14 to 14     from 24192 to 24192

FF_2 for the shared memory: we cannot have more than 2 thread blocks
per SM because each thread block requires 23 KB of shared memory and
the shared memory is set at 48 KB.

Final FF_2 launch configurations (see below) that: a) produce a fair
distribution (equal number of thread blocks per SM); and b) force the
thread block assignment at the beginning of the FF executions.

launch configurations  TB per SM       W per SM   Hardware registers  SH
( 14, from 10 to 18 )          1  from 10 to 18  from 17280 to 31104  23
( 28, from  5 to  9 )          2  from 10 to 18  from 17280 to 31104  46
```

7.2 The Warp Schedulers' Policy

A warp is a group of 32 threads and each thread block must have the same number of warps. After the gigathread scheduler has assigned a thread block to a streaming multiprocessor, the thread block cannot migrate. Therefore, the threads of a warp cannot migrate to a different streaming multiprocessor. Furthermore, the thread of a warp cannot migrate to a different warp. With 32 launch configurations per kernel, the number of warps that reside in a streaming multiprocessor ranges from 1 to 32. Each kernel is run thousands of times and each thread executes millions of instructions inside its for loop.

What we found it is that the starting and ending time differences within each pair of warps that reside in a streaming multiprocessor are very small. Such differences are almost constant and are independent of the kernel and of the launch configuration.

Table 7.9: ELF sub-case add.s32.n.wr.3 with launch configuration (1 , 4)

```
The instruction considered in this case is the ELF instruction ADD, with
operands that are 32-bit signed integers. The ELF instruction is executed
in normal mode ( n ). Its dependence type is write-read ( wr ) and its de-
pendence distance is 3. The kernel is launched using 1 thread block with
4 warps.

Legend:
- WX_st = Warp X's clock cycle starting time (warp enters in the for loop)
- WX_et = Warp X's clock cycle ending time (warp exits from the for loop)

Example:
W1_st = 103         W2_st = 143        W3_st = 125         W4_st = 132
W1_et = 2000008     W2_st = 2000139    W3_st = 2000121     W4_st = 2000133

Algorithm:
1) Calculation of the differences (Wi_st - Wy_st) and (Wi_et - Wy_et)
2) Comparison of the maximum starting and ending time differences

Starting time differences                          Ending time differences
abs(W1_st - W2_st) = 40                            abs(W1_st - W2_st) = 31
abs(W1_st - W3_st) = 22                            abs(W1_st - W3_st) = 13
abs(W1_st - W4_st) = 29                            abs(W1_st - W4_st) = 25
```

```
abs(W2_st - W3_st) = 18                    abs(W2_st - W3_st) = 18
abs(W2_st - W4_st) = 11                    abs(W2_st - W4_st) = 6
abs(W3_st - W4_st) = 7                     abs(W3_st - W4_st) = 12
```

In the above example, the starting and ending time differences are comparable for each warp pair. The maximum starting and ending time differences are comparable too because the maximum starting time difference is abs(W1_st - W2_st) = 40 and the the maximum ending time difference is abs(W1_st - W2_st) = 31. This is true for any kernel and any launch configuration. Furthermore, each thread, and therefore each warp, executes millions of instructions.

In addition, for a hardware designer, it is hard to determine what could be the best hardware policy to implement. This is because GPU architectures are complex, a wide variety of codes can run on GPUs, and a highly variable number of threads can be used for each code execution. Some policies could therefore give great advantages in some cases but not in others. However, it is difficult to make the right choice without a priori knowledge of what codes will run on the GPU.

It is therefore reasonable that the warp schedulers use a policy that is easy to implement in hardware and that provides good performance for regular codes that do not have divergences (these are the codes for which GPUs are usually developed). Such a policy also leads to the design of a smaller, easier, and more energy efficient hardware control logic.

In light of the previous presented results, these considerations lead us to the reasonable assumption that the warp schedulers' policy is a round robin one. However, in the future we will devise additional experiments that will exploit our ability to produce the desired ELF kernels to stress the warp schedulers in other different ways. This will be necessary to gather more evidence about the warp schedulers' policy or policies (we use policies here because the warp schedulers may be applying different policies to different streaming multiprocessor execution states).

7.3 The Warp Instructions' Features

NVIDIA does not disclose many PTX and ELF instructions' features (e.g. the warp instructions' latencies and the warps' latencies), and it is reasonable to assume that these features are probably different for different PTX and ELF instructions. The discovery and quantification of these features is important to understand how to avoid local pipeline stalls due to waiting for the production of the results of previous instructions and the re-reading of registers used as operands in previous instructions, and to understand the minimum number of local warps that have to reside in a streaming multiprocessor to hide a specific code's latencies.

The micro-benchmark kernels generate a table for each PTX or ELF instruction configuration, (e.g., add.s32 , n , write-read). The rows of these tables represent the number of warps that reside in a streaming multiprocessor (we therefore have one launch configuration per row). The columns represent the dependence distances for the instruction configuration (we therefore have one kernel per instruction configuration per column, because each kernel represents a different dependence distance). Each cell displays the average number of instruction configurations executed per functional unit clock cycle during the thousands of runs of each single couple (kernel , launch configuration). However, some launch configurations cannot be used with some kernels (see cells /////). This is because the necessary number of hardware registers per thread block would be greater than 32768 (the number of hardware registers of a streaming multiprocessor).

Table 7.10: Some PTX - ELF cases

```
Legend for the PTX instructions and their corresponding ELF instructions:

- (1) add. between signed integers (operands and results are 32-bits long)
- (2) predicate set to true or false (the operands are 32-bit integers)
      - if \%ope1 is greater than \%ope2 then set the predicate to true
      - if \%ope1 is = or <  than \%ope2 then set the predicate to false
- (3) the number of bits that are equal to 1 in a 32-bit register
- (4) max between signed integers (operands and results are 64-bits long)
- (5) mult. between signed integers (operands and results are 64-bits long)
```

Table 7.11: The PTX - ELF cases and their correspondences

Note how each PTX instruction is transformed into one or more ELF instructions, and the different number, type and dependences among ELF registers used into the ELF instructions. This generates completely different performance for the PTX instructions.

Instr.	PTX Instructions	PTX registers
(1)	add.s32 %res, %ope1, %ope2	3 normal
(2)	setp.gt.s32 %res, %ope1, %ope2	1 predicate and 2 normal
(3)	popc.b32 %res, %ope1	2 normal
(4)	%pred max.s64 %res, %ope1, %ope2	1 predicate and 3 normal
(5)	%pred mul.lo.s64 %res, %ope1, %ope2	1 predicate and 3 normal

Instr.	Corresponding ELF Instructions	ELF Registers
(1)	IADD R0, R3, R4	3 normal
(2)	ISETP.GT.AND P1, pt, R9, R7, pt	3 predicate and 2 normal
(3)	POPC R5, R6, R6	3 normal
(4)	IMNMX.XHI R9.CC, R3, R5, !pt	1 predicate and 3 normal
	IMNMX.U32.XL R8, R2, R4, !pt	1 predicate and 3 normal
	SEL R7, R9, R7, P0	1 predicate and 3 normal
	SEL R6, R8, R6, P0	1 predicate and 3 normal
(5)	IMUL.U32.U32 R9.CC, R2, R4	3 normal
	IMAD.U32.U32.HI.X R8.CC, R2, R4, RZ	4 normal
	SEL R6, R9, R6, !P0	1 predicate and 3 normal
	IMAD.U32.U32.X R8, R3, R4, R8	4 normal
	IMAD.U32.U32 R8, R2, R5, R8	4 normal
	SEL R7, R8, R7, !P0	1 predicate and 3 normal

Table 7.12: PTX - ELF cases: legend of the statistics

```
Legend:
- C01 (DPPNR): number of different PTX predicate not reserved registers
- C02 (DPPRR): number of different PTX predicate reserved registers
- C03 (DPNNR): number of different PTX normal not reserved registers
- C04 (DPNRR): number of different PTX normal reserved registers
- C05 (DEPNR): number of different ELF predicate not reserved registers
- C06 (DEPRR): number of different ELF predicate reserved registers
- C07 (DENNR): number of different ELF normal not reserved registers
- C08 (DENRR): number of different ELF normal reserved registers
- C09   (DPR): number of different PTX registers
- C10   (DER): number of different ELF registers
```

Table 7.13: Instruction and register statistics of the PTX - ELF cases

Instruction	C01	C02	C03	C04	C05	C06	C07	C08	C09	C10
(1)	0	0	3	0	0	0	3	0	3	3
(2)	1	0	2	0	1	1	2	0	3	4
(3)	0	0	2	0	0	0	2	0	2	2
(4)	1	0	3	0	0	1	3	0		
					0	1	3	0		
					1	0	2	0		
					1	0	2	0		
									4	10
(5)	1	0	3	0	0	0	3	0		
					0	0	3	1		
					1	0	2	0		
					0	0	3	0		
					0	0	3	0		
					1	0	2	0		
									4	10

Compared to the PTX add.s32, setp.gt.s32, and the pop.b32 instructions, which are each expanded into 1 ELF instruction, each PTX max.s64.c.no and mul.s64.lo.c.ne instruction is expanded in 4 and 6 ELF instructions, respectively.

Table 7.14: Legend for the quantification of the warp instructions' features

Symbol	Meaning	Associated colors
MT	maximum throughput	green and blue
IL	instruction's latency	green
WRL	write-read latency	green
RRL	read-read latency	magenta
WL	warp's latency	brown
LI	local instability	red
LS	local stability	blue

Table 7.15: Descriptions for the quantification of the warp instructions' features

Descriptions of the symbols:

1) Maximum Throughput (MT): maximum number of instructions executed by a streaming multiprocessor per functional unit clock cycle (e.g. 32 or 4).

2) Instruction's Latency (IL) or Write-Read Latency (WRL): number of functional unit clock cycles that is necessary to wait before being able to read, and therefore reuse, the register used to store the result of the instruction.

3) Read-Read Latency (RRL): number of functional unit clock cycles that is necessary to wait before being able to re-read a register used as operand in the instruction.

4) Warp's Latency (WL): number of functional unit clock cycles that is necessary to wait before a warp scheduler can reschedule the warp for the execution of the next instruction.

5) LI and LS: local stabilities and instabilities. We need to determine the cases when the local instabilities happen even if we know that the kernels' executions cannot be slowed down by the memories' bandwidths and latencies. This is because local stability is a necessary condition to reach the maximum throughput and therefore to produce code that is 100% efficient.

Table 7.16: Formulas for the quantification of the warp instructions' features

Note: there are 2 warp schedulers in each streaming multiprocessor and their clock frequencies are equal to half the clock frequency of the functional units that compose: a) the 2 groups of CUDA cores; b) the group of load and store units; and c) the group of 4 special functional units of each streaming multiprocessor.

Furthermore, each warp is scheduled as 2 groups of 16 threads if it requires the use of one of the 2 groups of 16 CUDA cores or of the group of 16 load and store units. If, instead, it requires the group of 4 special function units, then it is scheduled as 8 groups of 4 threads each.

Formulas:

1) Maximum Throughputs (MT): the cells in blue indicate the MTs. There is small statistical noise during the executions. The ceil of any blue number in the tables is the MT of the instruction.

2) Instruction's Latency (IL) or Write-Read Latency (WRL): the cells in green allow us to calculate the ILs and WRLs. Both are equal to the number of warps of the row of the green cells. This is because the tables consider instructions executed by the 2 group of CUDA cores. In fact, with 2 warp schedulers per streaming multiprocessor: a) 2 warps can be scheduled at each warp schedulers' clock cycle; 2) the warps are executed as 2 groups of 16 threads; and 3) the functional units work at a clock frequency that is 2 times faster than the warp schedulers' clock frequency.

3) Read-read Latency (RRL): the cells in magenta allow us to calculate the RRLs. The same reasoning applied to the calculation of the ILs and WRLs is also valid and applicable to the RRLs.

4) Warp's Latency (WL): the cells in brown allow us to calculate the WLs. The floor of (MT \ brown number) produces the number of functional unit clock cycles that it is necessary to wait for before a warp scheduler can reschedule again the warp.

5) Local Stabilities (LS) and Instabilities (LI): the cells in blue and red indicate the cases when we have local stability or instability in the single streaming multiprocessors. As the reader will see, even with kernels that cannot be slowed down by the memories' bandwidths and latencies, it is good to avoid launch configurations that imply the presence of an odd number of warps in a streaming multiprocessor.

Table 7.17: PTX instruction configuration (add.s32 , n , write-read)

A streaming multiprocessor is able to execute a maximum (MT) of 32 PTX add.s32.n instructions per functional unit clock cycle. This implies that the PTX add.s32.n instructions are executed by the 2 groups of 16 CUDA cores. The PTX add.s32.n instruction's latency (IL or write-read latency WRL) cannot be greater than 24 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	1.59	3.15	4.62	**5.07**	**5.09**	**5.07**	**5.09**	**5.07**	....
W02	3.18	6.29	9.24	10.14	10.17	10.14	10.18	10.14	....
W03	4.77	9.44	13.82	15.21	15.25	15.21	15.27	15.21	....
W04	6.36	12.58	18.43	20.28	20.34	20.28	20.36	20.28	....
W05	7.94	15.44	21.45	25.13	25.01	25.12	25.13	25.13	....
W06	9.53	18.54	25.67	**31.12**	**31.26**	**31.13**	**31.15**	**31.12**	....
W07	11.12	21.70	26.90	**26.87**	**26.87**	**26.88**	**26.89**	**26.88**	....
W08	12.70	24.66	**31.44**	**31.29**	**31.37**	**31.69**	**31.71**	**31.68**	....
W09	14.28	26.26	**27.91**	**27.88**	**27.94**	**27.94**	**27.97**	**27.89**	....
W10	15.86	28.14	**31.56**	**31.54**	**31.45**	**31.56**	**31.23**	**31.57**	....
W11	17.43	28.31	**28.45**	**28.38**	**28.41**	**28.43**	**28.41**	**28.38**	....
W12	19.01	**31.27**	**31.35**	**31.76**	**31.37**	**31.56**	**31.76**	**31.78**	....
W13	20.52	**28.75**	**28.79**	**28.79**	**28.81**	**28.80**	**28.83**	**28.80**	....
W14	22.07	**31.32**	**31.40**	**31.89**	**31.52**	**31.68**	**31.53**	**31.78**	....
W15	23.33	**29.02**	**29.03**	**29.06**	**29.07**	**29.05**	**29.06**	**29.05**	....
W16	24.83	**31.78**	**31.56**	**31.96**	**31.76**	**31.36**	**31.57**	**31.78**	....
W17	25.49	29.29	**29.33**	**29.28**	**29.31**	**29.31**	**29.32**	**29.29**	....
W18	26.99	**31.27**	**31.26**	**31.99**	**31.52**	**31.35**	**31.47**	**31.78**	....
W19	27.54	**29.42**	**29.42**	**29.43**	**29.50**	**29.47**	**29.48**	**29.45**	....
W20	28.94	**31.35**	**31.57**	**31.95**	**31.45**	**31.47**	**31.54**	**31.87**	....
W21	27.47	**29.55**	**29.58**	**29.64**	**29.65**	**29.65**	**29.67**	**29.64**	....
W22	28.80	**31.65**	**31.56**	**31.97**	**31.76**	**31.56**	**31.76**	**31.98**	....
W23	29.29	**29.69**	**29.70**	**29.79**	**29.78**	**29.79**	**29.79**	**29.79**	....
W24	**31.08**	**31.27**	**31.76**	**31.86**	**31.67**	**31.76**	**31.64**	**31.76**	....
W25	**27.98**	**29.80**	**29.83**	**29.89**	**29.87**	**29.88**	**29.87**	**29.87**	....
W26	**31.14**	**31.45**	**31.98**	**31.98**	**31.74**	**31.98**	**31.86**	**31.89**	....
W27	**29.08**	**29.87**	**29.92**	**29.92**	**29.95**	**29.95**	**29.97**	**29.96**	....
W28	**31.21**	**31.53**	**31.98**	**31.97**	**31.76**	**31.98**	**31.78**	**31.98**	....
W29	**28.48**	**29.93**	**30.01**	**30.03**	**30.02**	**30.04**	**30.04**	**30.02**	....
W30	**31.51**	**31.47**	**31.98**	**31.97**	**31.67**	**31.94**	**31.89**	**31.97**	....
W31	**30.19**	**30.04**	**30.03**	**30.03**	**30.07**	**30.06**	**30.07**	/////	....
W32	**31.75**	**31.65**	**31.98**	**31.93**	**31.76**	**31.99**	**31.93**	/////	....

Table 7.18: PTX instruction configuration (add.s32 , n , read-read)

The warp's latency (WL) for the PTX add.s32.n instruction is equal to 6 functional unit clock cycles (32 / 5.09). Re-reading any PTX operand register (read-read latency RRL) does not require more than 6 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	5.09	5.09	5.07	5.07	5.09	5.07	5.09	5.07	....
W02	10.18	10.18	10.14	10.14	10.17	10.14	10.18	10.14	....
W03	15.27	15.27	15.22	15.22	15.26	15.22	15.27	15.22	....
W04	20.37	20.37	20.29	20.29	20.34	20.29	20.37	20.29	....
W05	25.22	25.22	25.11	25.12	25.01	25.13	25.14	25.12	....
W06	31.25	31.26	31.13	31.14	31.01	31.13	31.16	31.13	....
W07	26.89	26.89	26.86	26.86	26.85	26.85	26.86	26.88	....
W08	31.71	31.71	31.71	31.69	31.67	31.69	31.71	31.70	....
W09	27.89	27.88	27.90	27.86	27.89	27.93	27.90	27.87	....
W10	31.94	31.94	31.94	31.93	31.98	31.92	31.01	31.96	....
W11	28.36	28.36	28.39	28.42	28.41	28.44	28.39	28.40	....
W12	31.92	31.93	31.91	31.92	31.99	31.95	31.99	31.92	....
W13	28.74	28.76	28.75	28.78	28.83	28.75	28.76	28.81	....
W14	31.90	31.91	31.88	31.91	31.91	31.88	31.92	31.91	....
W15	29.07	29.08	29.04	29.06	29.06	29.05	29.08	29.05	....
W16	31.99	31.97	31.97	31.99	31.98	31.98	31.97	31.99	....
W17	29.42	29.29	29.28	29.29	29.28	29.28	29.31	29.29	....
W18	31.99	31.78	31.86	31.94	31.98	31.89	31.92	31.94	....
W19	29.39	29.40	29.43	29.43	29.52	29.48	29.48	29.47	....
W20	31.94	31.92	31.89	31.97	31.93	31.97	31.92	31.91	....
W21	29.56	29.55	29.57	29.64	29.67	29.59	29.64	29.67	....
W22	31.98	31.97	31.93	31.94	31.99	31.97	31.93	31.92	....
W23	29.69	29.72	29.70	29.71	29.73	29.74	29.78	29.74	....
W24	31.98	31.91	31.93	31.92	31.97	31.93	31.94	31.92	....
W25	29.84	29.82	29.88	29.84	29.84	/////	/////	/////	....
W26	31.94	31.95	31.93	31.97	31.89	/////	/////	/////	....
W27	29.90	29.89	29.91	29.96	29.94	/////	/////	/////	....
W28	31.99	31.97	31.94	31.99	31.98	/////	/////	/////	....
W29	29.95	29.94	29.96	30.07	/////	/////	/////	/////	....
W30	31.99	31.97	31.98	31.96	/////	/////	/////	/////	....
W31	30.21	30.14	30.23	/////	/////	/////	/////	/////	....
W32	31.97	31.99	31.98	/////	/////	/////	/////	/////	....

Table 7.19: PTX instruction configurations (setp.gt.s32 , n , write-read and read-read)

A streaming multiprocessor is able to execute a maximum (MT) of 8 PTX setp.gt.s32.n instructions per functional unit clock cycle. This implies that the PTX setp.gt.s32.n instructions are executed by the 2 groups of 16 CUDA cores. The PTX setp.gt.s32.n instruction's latency (IL or write-read latency WRL) cannot be greater than 14 functional unit clocks cycles. The dependence distance (DD) does not affect the minimum number of warps (MNW) that are necessary to produce MT for the write-read dependence. The warp's latency (WL) is respectively equal to 14 and 6 functional unit clock cycles for the 2 PTX instruction configurations. Re-reading any PTX operand register (read-read latency RRL) does not require more than 28 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	setp.gt.s32.n.wr			setp.gt.s32.n.rr					
	DD01	DD02	DD03	DD01	DD02	DD03	DD04	DD05	DD06
W01	0.57	0.57	0.57	1.23	2.45	3.64	4.77	5.12	5.12
W02	1.13	1.13	1.13	2.45	4.89	7.28	9.53	10.23	10.23
W03	1.70	1.70	1.70	3.68	7.33	10.91	14.28	15.35	15.34
W04	2.27	2.27	2.27	4.91	9.78	14.55	19.07	20.46	20.45
W05	2.83	2.83	2.83	6.14	12.21	17.85	22.38	22.68	22.04
W06	3.40	3.40	3.40	7.36	14.65	21.40	26.89	31.01	31.02
W07	3.96	3.96	3.92	8.59	17.07	24.22	27.18	27.16	27.15
W08	4.49	4.52	4.52	9.81	19.50	27.59	31.03	31.04	31.03
W09	5.00	5.02	5.01	11.04	21.80	28.05	28.14	28.21	28.15
W10	5.54	5.52	5.49	12.26	24.17	31.02	31.31	31.16	31.24
W11	5.97	5.98	5.97	13.49	25.63	28.78	28.75	28.68	28.77
W12	6.48	6.49	6.48	14.71	28.10	31.43	31.28	31.26	31.25
W13	6.82	6.82	6.82	15.93	28.78	29.16	29.16	28.94	29.06
W14	7.72	7.71	7.70	17.15	31.02	31.22	31.24	31.16	31.28
W15	7.39	7.40	7.40	18.35	29.35	29.33	29.37	29.45	29.36
W16	7.85	7.86	7.85	19.58	31.22	31.32	31.33	31.30	31.31
W17	7.67	7.67	7.67	20.72	29.59	29.59	29.59	29.61	29.58
W18	7.92	7.91	7.91	21.87	31.32	31.31	31.33	31.30	31.31
W19	7.69	7.70	7.69	22.78	29.68	29.68	29.74	29.72	29.68
W20	7.93	7.94	7.93	23.89	31.33	31.31	31.28	31.30	31.31
W21	7.72	7.72	7.73	24.64	29.80	30.01	29.85	29.83	29.93
W22	7.93	7.94	7.94	25.90	31.33	31.32	31.30	31.31	31.32
W23	7.73	7.73	7.73	26.33	29.99	30.04	30.05	29.98	29.99
W24	7.94	7.94	7.94	27.42	31.33	31.31	31.31	31.39	31.31
W25	7.77	7.77	7.77	27.97	30.09	30.07	30.17	30.17	30.14
W26	7.96	7.96	7.96	29.15	31.31	31.30	31.33	31.34	31.31
W27	7.78	7.78	7.78	29.48	30.17	30.18	30.20	30.21	30.24
W28	7.96	7.96	7.96	31.02	31.32	31.29	31.28	31.27	31.28
W29	7.80	7.80	7.80	28.47	30.21	30.26	30.27	30.28	30.32
W30	7.96	7.96	7.96	31.12	31.31	31.29	31.29	31.36	31.29

Table 7.20: PTX instruction configuration (popc.b32 , n , write-read)

.
A streaming multiprocessor is able to execute a maximum (MT) of 16 PTX popc.b32.n instructions per functional unit clock cycle. This implies that the PTX popc.b32.n instructions are executed by the 2 groups of 16 CUDA cores. The PTX popc.b32 instruction's latency (IL or write-read latency WRL) cannot be greater than 12 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	1.59	3.16	4.65	5.12	5.13	5.12	5.13	5.11	....
W02	3.19	6.33	9.31	10.24	10.25	10.23	10.25	10.23	....
W03	4.78	9.49	13.93	15.35	15.35	15.34	15.38	15.34	....
W04	6.37	12.63	15.69	15.78	15.85	15.88	15.89	15.88	....
W05	7.96	15.55	15.45	15.80	15.83	15.81	15.82	15.84	....
W06	9.56	15.94	15.98	15.98	15.98	15.98	15.98	15.97	....
W07	11.14	15.39	15.90	15.93	15.93	15.92	15.93	15.93	....
W08	12.73	15.98	15.98	15.98	15.98	15.98	15.98	15.98	....
W09	14.27	15.84	15.95	15.95	15.94	15.95	15.94	15.94	....
W10	15.87	15.98	15.99	15.99	15.99	15.99	15.99	15.99	....
W11	15.92	15.89	15.96	15.95	15.96	15.96	15.96	15.95	....
W12	15.96	15.96	15.99	15.99	15.99	15.99	15.99	15.99	....
W13	15.47	15.91	15.96	15.96	15.96	15.95	15.96	15.95	....
W14	15.98	15.97	15.98	15.99	15.97	15.99	15.99	15.99	....
W15	15.70	15.95	15.97	15.97	15.95	15.96	15.96	15.96	....
W16	15.99	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W17	15.82	15.96	15.96	15.97	15.95	15.96	15.97	15.96	....
W18	15.98	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W19	15.88	15.96	15.96	15.95	15.96	15.96	15.95	15.96	....
W20	15.98	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W21	15.55	15.96	15.96	15.95	15.97	15.96	15.96	15.96	....
W22	15.95	15.99	15.99	15.99	15.98	15.99	15.99	15.99	....
W23	15.92	15.96	15.97	15.96	15.98	15.96	15.96	15.95	....
W24	15.97	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W25	15.72	15.96	15.96	15.97	15.96	15.97	15.98	15.95	....
W26	15.96	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W27	15.92	15.96	15.96	15.95	15.96	15.96	15.95	15.96	....
W28	15.98	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W29	15.81	15.96	15.95	15.96	15.97	15.95	15.97	15.96	....
W30	15.97	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W31	15.95	15.97	15.97	15.97	15.97	15.97	15.97	15.97	....
W32	15.98	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....

Table 7.21: PTX instruction configuration (popc.b32 , n , read-read)

The warp's latency (WL) for the PTX popc.b32.n instruction is equal to 6 functional unit clock cycles (16 / 5.12). Re-reading any PTX operand register (read-read latency RRL) does not require more than 6 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	5.12	5.12	5.13	5.13	5.12	5.12	5.13	5.12	....
W02	10.23	10.24	10.23	10.24	10.23	10.24	10.24	10.23	....
W03	15.35	15.38	15.37	15.38	15.37	15.36	15.38	15.37	....
W04	15.82	15.81	15.79	15.78	15.83	15.80	15.79	15.81	....
W05	15.87	15.86	15.85	15.84	15.83	15.88	15.85	15.88	....
W06	15.97	15.98	15.98	15.98	15.98	15.98	15.98	15.97	....
W07	15.93	15.92	15.91	15.93	15.92	15.89	15.93	15.92	....
W08	15.99	15.98	15.99	15.98	15.98	15.99	15.98	15.98	....
W09	15.95	15.95	15.94	15.95	15.94	15.95	15.95	15.94	....
W10	15.99	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W11	15.96	15.95	15.95	15.95	15.94	15.95	15.96	15.95	....
W12	15.99	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W13	15.96	15.96	15.96	15.95	15.96	15.95	15.96	15.95	....
W14	15.99	15.99	15.99	15.99	15.97	15.99	15.99	15.99	....
W15	15.96	15.95	15.96	15.96	15.95	15.96	15.96	15.95	....
W16	15.99	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W17	15.97	15.96	15.94	15.97	15.96	15.96	15.97	15.95	....
W18	15.99	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W19	15.96	15.96	15.96	15.95	15.95	15.96	15.96	15.96	....
W20	15.98	15.99	15.97	15.99	15.98	15.99	15.97	15.99	....
W21	15.96	15.95	15.96	15.94	15.97	15.96	15.95	15.96	....
W22	15.99	15.99	15.99	15.99	15.98	15.99	15.99	15.99	....
W23	15.92	15.93	15.97	15.96	15.95	15.96	15.94	15.92	....
W24	15.98	15.99	15.99	15.97	15.99	15.99	15.98	15.99	....
W25	15.92	15.96	15.94	15.97	15.98	/////	/////	/////	....
W26	15.99	15.99	15.99	15.97	15.90	/////	/////	/////	....
W27	15.96	15.93	15.94	15.95	15.94	/////	/////	/////	....
W28	15.99	15.99	15.97	15.99	15.99	/////	/////	/////	....
W29	15.92	15.96	15.93	/////	/////	/////	/////	/////	....
W30	15.98	15.99	15.99	/////	/////	/////	/////	/////	....
W31	15.94	15.97	15.96	/////	/////	/////	/////	/////	....
W32	15.98	15.99	15.99	/////	/////	/////	/////	/////	....

Table 7.22: PTX instruction configurations (max.s64 , c.no , write-read and read-read)

A streaming multiprocessor is able to execute a maximum (MT) of 4 PTX max.s64.c.no instructions per functional unit clock cycle. The PTX max.s64.c.no instruction's latency (IL or write-read latency WRL) cannot be greater than 8 functional unit clocks cycles. The dependence distance (DD) does not affect the minimum number of warps (MNW) that are necessary to produce MT. The warp's latency (WL) is equal to 8 functional unit clock cycles (4 / 0.49). Re-reading any PTX operand register (read-read latency RRL) does not require more than 8 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	max.s64.c.no.wr				max.s64.c.no.rr			
	DD01	DD02	DD03	....	DD01	DD02	DD03	....
W01	**0.49**	**0.48**	**0.49**	....	**0.49**	**0.49**	**0.49**	....
W02	1.22	1.22	1.22	....	1.22	1.22	1.22	....
W03	1.82	0.81	0.82	....	1.80	0.81	0.82	....
W04	2.42	2.43	2.42	....	2.43	2.42	2.43	....
W05	2.98	3.00	2.99	....	2.99	2.98	2.99	....
W06	3.51	3.45	3.40	....	3.51	3.46	3.55	....
W07	3.77	3.81	3.74	....	3.80	3.81	3.79	....
W08	**3.98**	**3.97**	**3.98**	....	**3.98**	**3.99**	**3.98**	....
W09	**3.84**	**3.85**	**3.85**	....	**3.84**	**3.85**	**3.84**	....
W10	**3.99**	**4.00**	**3.99**	....	**3.99**	**3.98**	**4.00**	....
W11	**3.92**	**3.93**	**3.94**	....	**3.91**	**3.95**	**3.94**	....
W12	**3.99**	**3.99**	**4.00**	....	**4.00**	**3.99**	**3.99**	....
W13	**3.96**	**3.97**	**3.95**	....	**3.96**	**3.97**	**3.96**	....
W14	**4.00**	**3.99**	**4.00**	....	**4.00**	**3.99**	**4.00**	....
W15	**3.97**	**3.96**	**3.97**	....	**3.96**	**3.97**	**3.98**	....
W16	**4.00**	**3.99**	**3.99**	....	**4.00**	**3.99**	**4.00**	....
W17	**3.97**	**3.98**	**3.99**	....	**3.98**	**3.99**	**3.98**	....
W18	**3.99**	**4.00**	**3.99**	....	**3.99**	**3.99**	**3.99**	....
W19	**3.98**	**3.99**	**3.98**	....	**3.98**	**3.99**	**3.98**	....
W20	**3.99**	**3.99**	**3.98**	....	**3.99**	**3.98**	**3.99**	....
W21	**3.99**	**3.99**	**3.99**	....	**3.99**	**3.99**	**3.99**	....
W22	**4.00**	**4.00**	**4.00**	....	**4.00**	**4.00**	**4.00**	....
W23	**3.98**	**3.99**	**3.98**	....	**3.99**	**3.98**	////	....
W24	**4.00**	**4.00**	**4.00**	....	**4.00**	**4.00**	////	....
W25	**3.98**	**3.99**	////	....	**3.98**	////	////	....
W26	**4.00**	**4.00**	////	....	**4.00**	////	////	....
W27	**3.99**	**3.98**	////	....	**3.99**	////	////	....
W28	**4.00**	**4.00**	////	....	**4.00**	////	////	....
W29	**3.99**	**3.99**	////	....	**3.99**	////	////	....
W30	**4.00**	**4.00**	////	....	**4.00**	////	////	....
W31	**3.91**	////	////	....	**3.91**	////	////	....
W32	**3.98**	////	////	....	**3.98**	////	////	....

Table 7.23: ELF instruction configurations (IMMMX.XHI , c.ne , wr-read and read-read)

A streaming multiprocessor is able to execute a maximum (MT) of 16 ELF IMMMX.XHI.X.c.ne instructions per functional unit clock cycle. The ELF IMMMX.XHI.X.c.ne instruction's latency (IL or write-read latency WRL) cannot be greater than 14 functional units clocks cycles. The dependence distance (DD) does not affect the minimum number of warps (MNW) that are necessary to produce MT. The warp's latency (WL) is equal to 13 functional unit clock cycles (16 / 1.22). Re-reading any ELF operand register (read-read latency RRL) does not require more than 14 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	IMMMX.XHI.c.ne.wr				IMMMX.XHI.c.ne.rr			
	DD01	DD02	DD03	....	DD01	DD02	DD03	....
W01	1.22	1.22	1.22	....	1.22	1.22	1.22	....
W02	2.44	2.44	2.44	....	2.44	2.44	2.44	....
W03	3.66	3.66	3.66	....	3.66	3.66	3.66	....
W04	4.88	4.88	4.88	....	4.88	4.88	4.88	....
W05	6.10	6.10	6.10	....	6.10	6.10	6.10	....
W06	7.32	7.32	7.32	....	7.32	7.32	7.32	....
W07	8.54	8.53	8.54	....	8.54	8.54	8.54	....
W08	9.76	9.76	9.76	....	9.76	9.76	9.76	....
W09	10.98	10.98	10.98	....	10.97	10.98	10.98	....
W10	12.19	12.20	12.21	....	12.18	12.17	12.19	....
W11	13.40	13.38	13.34	....	13.34	13.37	13.35	....
W12	14.54	14.56	14.49	....	14.47	14.50	14.43	....
W13	15.51	15.52	15.36	....	15.09	15.43	15.23	....
W14	15.87	15.86	15.87	....	15.87	15.89	15.87	....
W15	15.35	15.37	15.36	....	15.34	15.35	15.36	....
W16	15.95	15.96	15.95	....	15.95	15.95	15.96	....
W17	15.56	15.54	15.56	....	15.55	15.59	15.57	....
W18	15.97	15.98	15.97	....	15.97	15.97	15.96	....
W19	15.69	15.71	15.69	....	15.72	15.70	15.70	....
W20	15.97	15.98	15.97	....	15.97	15.99	15.97	....
W21	15.79	15.81	15.80	....	15.79	15.82	15.80	....
W22	15.98	15.99	15.98	....	15.98	15.98	15.98	....
W23	15.84	15.86	15.85	....	15.84	15.84	15.85	....
W24	15.97	15.98	15.97	....	15.97	15.98	15.97	....
W25	15.89	15.91	15.90	....	15.89	15.91	15.90	....
W26	15.97	15.98	15.97	....	15.97	15.96	15.97	....
W27	15.92	15.92	15.92	....	15.92	15.91	15.92	....
W28	15.97	15.98	15.97	....	15.97	15.99	15.97	....
W29	15.91	15.93	15.94	....	15.93	15.93	15.92	....
W30	15.97	15.98	15.97	....	15.97	15.96	15.97	....

Table 7.24: ELF instruction configuration (IMMMX.U32.XL , c.no , write-read)

.

A streaming multiprocessor is able to execute a maximum (MT) of 16 ELF IMMMX.U32.XL.c.no instructions per functional unit clock cycle. This implies that the ELF IMMMX.U32.XL.c.no instructions are executed by the 2 groups of 16 CUDA cores. The ELF IMMMX.U32.XL.c.no instruction's latency (IL or write-read latency WRL) cannot be greater than 12 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	1.59	3.15	4.62	5.07	5.09	5.07	5.09	5.08	....
W02	3.18	6.31	9.24	10.14	10.16	10.14	10.18	10.16	....
W03	4.77	9.45	13.82	15.22	15.26	15.23	15.28	15.22	....
W04	6.36	12.56	15.61	15.70	15.92	15.91	15.91	15.90	....
W05	7.95	15.27	15.42	15.75	15.74	15.75	15.75	15.75	....
W06	9.53	15.86	15.98	15.97	15.97	15.98	15.97	15.97	....
W07	11.12	15.36	15.86	15.89	15.89	15.90	15.90	15.90	....
W08	12.70	15.98	15.97	15.98	15.99	15.98	15.98	15.98	....
W09	14.23	15.81	15.93	15.93	15.93	15.92	15.93	15.93	....
W10	15.81	15.98	15.97	15.99	15.99	15.99	15.99	15.99	....
W11	15.89	15.86	15.94	15.94	15.94	15.92	15.94	15.94	....
W12	15.95	15.95	15.99	15.99	15.99	15.98	15.99	15.99	....
W13	15.46	15.89	15.95	15.95	15.94	15.95	15.94	15.95	....
W14	15.98	15.96	15.99	15.99	15.98	15.99	15.99	15.99	....
W15	15.69	15.94	15.95	15.95	15.95	15.94	15.95	15.95	....
W16	15.98	15.98	15.99	15.99	15.98	15.99	15.99	15.99	....
W17	15.81	15.95	15.96	15.96	15.94	15.96	15.95	15.96	....
W18	15.98	15.99	15.99	15.99	15.98	15.99	15.99	15.99	....
W19	15.88	15.95	15.95	15.95	15.94	15.95	15.95	15.95	....
W20	15.98	15.98	15.98	15.98	15.98	15.99	15.99	15.98	....
W21	15.59	15.95	15.95	15.95	15.95	15.95	15.94	15.95	....
W22	15.93	15.99	15.99	15.99	15.99	15.98	15.99	15.99	....
W23	15.89	15.95	15.95	15.96	15.95	15.95	15.96	15.96	....
W24	15.96	15.99	15.98	15.99	15.99	15.99	15.99	15.99	....
W25	15.74	15.96	15.97	15.96	15.95	15.96	15.96	15.96	....
W26	15.95	15.99	15.99	15.99	15.99	15.99	15.98	15.99	....
W27	15.90	15.96	15.95	15.96	15.96	15.96	15.96	15.96	....
W28	15.97	15.99	15.99	15.98	15.99	15.99	15.99	15.99	....
W29	15.82	15.96	15.95	15.96	15.95	15.96	15.96	15.96	....
W30	15.96	15.99	15.98	15.99	15.99	15.99	15.99	15.99	....
W31	15.94	15.96	15.95	15.96	15.96	15.96	15.96	/////	....
W32	15.98	15.99	15.98	15.99	15.99	15.99	15.99	/////	....

Table 7.25: ELF instruction configuration (IMMMX.U32.XL , c.no , read-read)

The warp's latency (WL) for the ELF IMMMX.U32.XL.c.no instruction is equal to 3 functional unit clock cycles (16 / 5.12). Re-reading any ELF operand register (read-read latency RRL) does not require more than 6 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	5.12	5.13	5.12	5.11	5.12	5.13	5.12	5.13	....
W02	10.18	10.17	10.14	10.14	10.17	10.15	10.17	10.14	....
W03	15.27	15.27	15.22	15.21	15.26	15.23	15.27	15.21	....
W04	15.72	15.67	15.67	15.69	15.71	15.70	15.71	15.71	....
W05	15.80	15.76	15.76	15.73	15.73	15.73	15.76	15.74	....
W06	15.98	15.97	15.97	15.99	15.97	15.97	15.98	15.97	....
W07	15.91	15.91	15.92	15.90	15.90	15.92	15.90	15.90	....
W08	15.99	15.98	15.98	15.98	15.98	15.97	15.98	15.98	....
W09	15.93	15.94	15.93	15.93	15.93	15.94	15.93	15.93	....
W10	15.99	15.98	15.99	15.99	15.99	15.98	15.99	15.99	....
W11	15.94	15.94	15.93	15.94	15.95	15.94	15.94	15.94	....
W12	15.99	15.99	15.98	15.99	15.99	15.99	15.99	15.99	....
W13	15.95	15.95	15.95	15.94	15.94	15.95	15.95	15.95	....
W14	15.99	15.99	15.99	15.99	15.99	15.97	15.99	15.99	....
W15	15.95	15.95	15.95	15.95	15.95	15.96	15.95	15.95	....
W16	15.99	15.99	15.99	15.98	15.99	15.99	15.99	15.99	....
W17	15.96	15.96	15.95	15.96	15.95	15.96	15.95	15.96	....
W18	15.99	15.99	15.98	15.99	15.99	15.99	15.99	15.97	....
W19	15.95	15.95	15.95	15.95	15.96	15.95	15.95	15.97	....
W20	15.98	15.98	15.98	15.98	15.99	15.98	15.98	15.98	....
W21	15.95	15.95	15.95	15.94	15.95	15.95	15.95	15.95	....
W22	15.99	15.99	15.98	15.99	15.99	15.99	15.99	15.99	....
W23	15.95	15.95	15.93	15.95	15.95	15.95	15.95	15.95	....
W24	15.99	15.99	15.99	15.98	15.99	15.98	15.99	15.99	....
W25	15.96	15.96	15.96	15.96	15.96	/////	/////	/////	....
W26	15.99	15.97	15.99	15.98	15.99	/////	/////	/////	....
W27	15.96	15.96	15.94	15.95	15.95	/////	/////	/////	....
W28	15.99	15.99	15.98	15.99	15.99	/////	/////	/////	....
W29	15.96	15.96	15.96	15.96	/////	/////	/////	/////	....
W30	15.99	15.99	15.98	15.99	/////	/////	/////	/////	....
W31	15.97	15.97	15.95	/////	/////	/////	/////	/////	....
W32	15.99	15.99	15.99	/////	/////	/////	/////	/////	....

Table 7.26: ELF instruction configuration (SEL , c.no , write-read)

A streaming multiprocessor is able to execute a maximum (MT) of 16 ELF SEL.c.no instructions per functional unit clock cycle. This implies that the ELF SEL.c.no instructions are executed by the 2 groups of 16 CUDA cores. The ELF SEL.c.no instruction's latency (IL or write-read latency WRL) cannot be greater than 12 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	DD01	DD02	DD03	DD04	DD05	DD06	....
W01	1.59	3.16	4.62	**5.07**	**5.09**	**5.07**	....
W02	3.18	6.31	9.24	10.14	10.19	10.14	....
W03	4.77	9.44	13.83	15.22	15.29	15.22	....
W04	6.36	12.57	**15.61**	**15.83**	**15.91**	**15.92**	....
W05	7.93	15.28	**15.42**	**15.73**	**15.71**	**15.75**	....
W06	9.53	**15.91**	**15.98**	**15.96**	**15.98**	**15.98**	....
W07	11.12	**15.37**	**15.86**	**15.87**	**15.90**	**15.89**	....
W08	12.70	**15.98**	**15.97**	**15.97**	**15.97**	**15.97**	....
W09	14.22	**15.81**	**15.93**	**15.93**	**15.91**	**15.92**	....
W10	15.81	**15.98**	**15.98**	**15.99**	**15.97**	**15.98**	....
W11	15.89	**15.85**	**15.94**	**15.91**	**15.92**	**15.94**	....
W12	**15.95**	**15.95**	**15.99**	**15.98**	**15.99**	**15.99**	....
W13	**15.46**	**15.91**	**15.95**	**15.93**	**15.94**	**15.95**	....
W14	**15.98**	**15.96**	**15.99**	**15.98**	**15.98**	**15.99**	....
W15	**15.69**	**15.93**	**15.95**	**15.94**	**15.95**	**15.95**	....
W16	**15.98**	**15.98**	**15.98**	**15.99**	**15.98**	**15.99**	....
W17	**15.81**	**15.94**	**15.95**	**15.96**	**15.93**	**15.95**	....
W18	**15.98**	**15.98**	**15.99**	**15.98**	**15.99**	**15.98**	....
W19	**15.87**	**15.94**	**15.95**	**15.94**	**15.95**	**15.95**	....
W20	**15.97**	**15.98**	**15.99**	**15.99**	**15.98**	**15.98**	....
W21	**15.59**	**15.95**	**15.94**	**15.94**	**15.95**	**15.95**	....
W22	**15.94**	**15.99**	**15.98**	**15.99**	**15.98**	**15.99**	....
W23	**15.90**	**15.95**	**15.94**	**15.96**	**15.94**	**15.96**	....
W24	**15.97**	**15.99**	**15.98**	**15.99**	**15.98**	**15.99**	....
W25	**15.74**	**15.95**	**15.96**	**15.95**	**15.95**	**15.96**	....
W26	**15.95**	**15.99**	**15.98**	**15.99**	**15.98**	**15.98**	....
W27	**15.90**	**15.96**	**15.95**	**15.96**	**15.93**	**15.96**	....
W28	**15.97**	**15.95**	**15.97**	**15.99**	**15.97**	**15.97**	....
W29	**15.81**	**15.95**	**15.96**	**15.95**	**15.96**	**15.96**	....
W30	**15.96**	**15.99**	**15.98**	**15.98**	**15.99**	**15.98**	....
W31	**15.94**	**15.97**	**15.97**	**15.96**	**15.96**	**15.95**	....
W32	**15.98**	**15.99**	**15.98**	**15.99**	**15.99**	**15.99**	....

Table 7.27: ELF instruction configuration (SEL , c.no , read-read)

The warp's latency (WL) for the ELF SEL.c.ne instruction is equal to 3 functional unit clock cycles (16 / 5.09). Re-reading any PTX operand register (read-read latency RRL) does not require more than 6 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	DD01	DD02	DD03	DD04	DD05	DD06	....
W01	5.09	5.09	5.07	5.09	5.09	5.07	....
W02	10.16	10.18	10.12	10.14	10.18	10.14	....
W03	15.27	15.29	15.21	15.23	15.25	15.21	....
W04	15.68	15.67	15.70	15.69	15.63	15.69	....
W05	15.77	15.77	15.76	15.76	15.73	15.75	....
W06	15.97	15.98	15.99	15.98	15.98	15.98	....
W07	15.91	15.91	15.90	15.90	15.88	15.91	....
W08	15.98	15.98	15.99	15.98	15.99	15.98	....
W09	15.94	15.95	15.94	15.94	15.93	15.94	....
W10	15.99	15.99	15.98	15.99	15.98	15.99	....
W11	15.95	15.95	15.94	15.93	15.94	15.94	....
W12	15.99	15.98	15.99	15.98	15.99	15.99	....
W13	15.95	15.95	15.94	15.95	15.96	15.95	....
W14	15.99	15.99	15.98	15.99	15.98	15.99	....
W15	15.95	15.95	15.94	15.95	15.94	15.95	....
W16	15.99	15.98	15.99	15.98	15.99	15.99	....
W17	15.95	15.96	15.95	15.94	15.95	15.96	....
W18	15.99	15.98	15.99	15.99	15.98	15.99	....
W19	15.95	15.94	15.96	15.94	15.95	15.96	....
W20	15.99	15.99	15.98	15.99	15.98	15.99	....
W21	15.95	15.96	15.95	15.96	15.94	15.96	....
W22	15.99	15.99	15.98	15.99	15.98	15.99	....
W23	15.96	15.94	15.95	15.94	15.96	15.95	....
W24	15.99	15.99	15.98	15.98	15.99	15.99	....
W25	15.96	15.95	15.95	15.94	15.96	/////	....
W26	15.99	15.98	15.99	15.98	15.99	/////	....
W27	15.95	15.96	15.96	15.94	15.96	/////	....
W28	15.99	15.98	15.99	15.98	15.99	/////	....
W29	15.96	15.94	15.95	15.95	/////	/////	....
W30	15.99	15.98	15.98	15.99	/////	/////	....
W31	15.96	15.97	15.95	/////	/////	/////	....
W32	15.97	15.98	15.99	/////	/////	/////	....

Table 7.28: PTX instruction configurations (mul.lo.s64 , c.ne , write-read and read-read)

A streaming multiprocessor is able to execute a maximum (MT) of 4 PTX mul.lo.s64.c.ne instructions per functional unit clock cycle. The PTX mul.lo.s64.c.ne instruction's latency (IL or write-read latency WRL) cannot be greater than 14 functional units clocks cycles. The dependence distance (DD) does not affect the minimum number of warps (MNW) that are necessary to produce MT. The warp's latency (WL) is equal to 13 functional unit clock cycles (4 / 0.30). Re-reading any PTX operand register (read-read latency RRL) does not require more than 14 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	mul.lo.s64.c.ne.wr				mul.lo.s64.c.ne.rr			
	DD01	DD02	DD03	....	DD01	DD02	DD03	....
W01	0.30	0.29	0.30	....	0.30	0.30	0.29	....
W02	0.67	0.66	0.66	....	0.67	0.66	0.66	....
W03	1.01	0.99	0.99	....	1.01	0.99	0.98	....
W04	1.35	1.31	1.31	....	1.35	1.31	1.30	....
W05	1.67	1.63	1.63	....	1.66	1.63	1.62	....
W06	1.98	1.92	1.92	....	1.99	1.93	1.93	....
W07	2.29	2.20	2.19	....	2.26	2.22	2.21	....
W08	2.50	2.50	2.49	....	2.58	2.51	2.51	....
W09	2.80	2.73	2.72	....	2.77	2.76	2.75	....
W10	3.09	3.00	3.01	....	3.06	3.04	3.04	....
W11	3.28	3.19	3.18	....	3.23	3.23	3.23	....
W12	3.54	3.46	3.45	....	3.49	3.50	3.50	....
W13	3.65	3.57	3.58	....	3.60	3.61	3.62	....
W14	3.86	3.79	3.81	....	3.81	3.82	3.83	....
W15	3.82	3.80	3.81	....	3.79	3.80	3.82	....
W16	3.95	3.94	3.95	....	3.94	3.94	3.95	....
W17	3.84	3.83	3.83	....	3.83	3.84	3.83	....
W18	3.97	3.96	3.97	....	3.97	3.97	3.97	....
W19	3.85	3.84	3.84	....	3.84	3.84	3.84	....
W20	3.97	3.97	3.96	....	3.96	3.97	3.97	....
W21	3.86	3.86	3.86	....	3.85	3.85	3.86	....
W22	3.97	3.97	3.97	....	3.97	3.97	3.97	....
W23	3.87	3.87	3.87	....	3.87	3.87	////	....
W24	3.98	3.97	3.97	....	3.98	3.97	////	....
W25	3.89	3.88	////	....	3.88	////	////	....
W26	3.98	3.98	////	....	3.98	////	////	....
W27	3.90	3.89	////	....	3.89	////	////	....
W28	3.98	3.98	////	....	3.98	////	////	....
W29	3.91	3.90	////	....	3.90	////	////	....
W30	3.98	3.98	////	....	3.98	////	////	....
W31	3.91	////	////	....	3.91	////	////	....
W32	3.98	////	////	....	3.98	////	////	....

Table 7.29: ELF instruction configuration (IMUL.U32.U32 , n , write-read)

.

A streaming multiprocessor is able to execute a maximum (MT) of 16 ELF IMUL.U32.U32.n instructions per functional unit clock cycle. This implies that the ELF IMUL.U32.U32.n instructions are executed by the 2 groups of 16 CUDA cores. The ELF IMUl.U32.U32.n instruction's latency (IL or write-read latency WRL) cannot be greater than 12 clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	1.59	3.15	4.62	5.07	5.09	5.07	5.08	5.08	....
W02	3.18	6.29	9.24	10.14	10.17	10.14	10.18	10.16	....
W03	4.77	9.44	13.82	15.22	15.26	15.22	15.28	15.22	....
W04	6.36	12.56	15.61	15.70	15.91	15.91	15.91	15.90	....
W05	7.95	15.27	15.42	15.74	15.74	15.75	15.75	15.75	....
W06	9.53	15.86	15.98	15.97	15.97	15.97	15.97	15.97	....
W07	11.12	15.36	15.86	15.90	15.89	15.90	15.90	15.90	....
W08	12.70	15.98	15.97	15.98	15.98	15.98	15.98	15.98	....
W09	14.23	15.81	15.93	15.93	15.93	15.93	15.93	15.93	....
W10	15.81	15.98	15.98	15.99	15.99	15.99	15.99	15.99	....
W11	15.89	15.86	15.94	15.94	15.94	15.93	15.94	15.94	....
W12	15.95	15.95	15.99	15.99	15.99	15.99	15.99	15.99	....
W13	15.46	15.89	15.95	15.95	15.94	15.95	15.95	15.95	....
W14	15.98	15.96	15.99	15.99	15.99	15.99	15.99	15.99	....
W15	15.69	15.94	15.95	15.95	15.95	15.95	15.95	15.95	....
W16	15.98	15.98	15.99	15.99	15.99	15.99	15.99	15.99	....
W17	15.81	15.95	15.96	15.96	15.95	15.96	15.95	15.96	....
W18	15.98	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W19	15.88	15.95	15.95	15.95	15.95	15.95	15.95	15.95	....
W20	15.98	15.98	15.98	15.99	15.98	15.99	15.99	15.98	....
W21	15.59	15.95	15.95	15.95	15.95	15.95	15.95	15.95	....
W22	15.93	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W23	15.89	15.95	15.95	15.96	15.95	15.96	15.96	15.96	....
W24	15.96	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W25	15.74	15.96	15.96	15.96	15.95	15.96	15.96	15.96	....
W26	15.95	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W27	15.90	15.96	15.96	15.96	15.96	15.96	15.96	15.96	....
W28	15.97	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W29	15.82	15.96	15.96	15.96	15.95	15.96	15.96	15.96	....
W30	15.96	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W31	15.94	15.96	15.96	15.96	15.96	15.96	15.96	/////	....
W32	15.98	15.99	15.99	15.99	15.99	15.99	15.99	/////	....

Table 7.30: ELF instruction configuration (IMUL.U32.U32 , n , read-read)

The warp's latency (WL) for the ELF IMUL.U32.U32.n instruction is equal to 3 functional unit clock cycles (16 / 5.12). Re-reading any ELF operand register (read-read latency RRL) does not require more than 6 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	**5.12**	**5.13**	**5.12**	**5.13**	**5.12**	**5.13**	**5.12**	**5.13**	....
W02	10.18	10.18	10.14	10.14	10.17	10.14	10.17	10.14	....
W03	15.27	15.27	15.21	15.21	15.26	15.21	15.27	15.21	....
W04	15.72	15.67	15.66	15.69	15.71	15.70	15.69	15.71	....
W05	15.80	15.76	15.75	15.73	15.73	15.73	15.76	15.73	....
W06	**15.97**	**15.97**	**15.97**	**15.97**	**15.97**	**15.97**	**15.97**	**15.97**	....
W07	**15.91**	**15.91**	**15.91**	**15.90**	**15.90**	**15.90**	**15.90**	**15.90**	....
W08	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	....
W09	**15.93**	**15.93**	**15.93**	**15.93**	**15.93**	**15.93**	**15.93**	**15.93**	....
W10	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	....
W11	**15.94**	**15.94**	**15.94**	**15.94**	**15.94**	**15.94**	**15.94**	**15.94**	....
W12	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	....
W13	**15.95**	**15.95**	**15.95**	**15.95**	**15.94**	**15.95**	**15.95**	**15.95**	....
W14	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	....
W15	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	....
W16	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	....
W17	**15.96**	**15.96**	**15.96**	**15.96**	**15.95**	**15.96**	**15.95**	**15.96**	....
W18	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	....
W19	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	....
W20	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	....
W21	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	....
W22	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	....
W23	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	....
W24	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	....
W25	**15.96**	**15.96**	**15.96**	**15.96**	**15.95**	/////	/////	/////	....
W26	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	/////	/////	/////	....
W27	**15.96**	**15.96**	**15.95**	**15.95**	**15.95**	/////	/////	/////	....
W28	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	/////	/////	/////	....
W29	**15.96**	**15.96**	**15.95**	**15.96**	/////	/////	/////	/////	....
W30	**15.99**	**15.99**	**15.99**	**15.99**	/////	/////	/////	/////	....
W31	**15.97**	**15.97**	**15.96**	/////	/////	/////	/////	/////	....
W32	**15.99**	**15.99**	**15.99**	/////	/////	/////	/////	/////	....

Table 7.31: ELF instruction configurations (IMAD.U32.U32.HI.X , n , wr-re and re-re)

A streaming multiprocessor is able to execute a maximum (MT) of 16 ELF IMAD.U32.U32.HI.X.n instructions per functional unit clock cycle. The ELF IMAD.U32 .U32.HI.X.n instruction's latency (IL or write-read latency WRL) cannot be greater than 14 functional unit clocks cycles. The dependence distance (DD) does not affect the minimum number of warps (MNW) that are necessary to produce MT. The warp's latency (WL) is equal to 13 functional unit clock cycles (16 / 1.22). Re-reading any PTX operand register (read-read latency RRL) does not require more than 14 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	IMAD.U32.U32.HI.X.n.wr				IMAD.U32.U32.HI.X.n.rr			
	DD01	DD02	DD03	....	DD01	DD02	DD03	....
W01	1.22	1.22	1.22	....	1.22	1.22	1.22	....
W02	2.44	2.44	2.44	....	2.44	2.44	2.44	....
W03	3.66	3.66	3.66	....	3.66	3.66	3.66	....
W04	4.88	4.88	4.88	....	4.88	4.88	4.88	....
W05	6.10	6.10	6.10	....	6.10	6.10	6.10	....
W06	7.32	7.32	7.32	....	7.32	7.32	7.32	....
W07	8.54	8.54	8.54	....	8.54	8.54	8.54	....
W08	9.76	9.76	9.76	....	9.76	9.76	9.76	....
W09	10.98	10.98	10.98	....	10.98	10.98	10.98	....
W10	12.19	12.20	12.19	....	12.18	12.19	12.19	....
W11	13.40	13.38	13.33	....	13.34	13.38	13.35	....
W12	14.54	14.57	14.49	....	14.46	14.50	14.43	....
W13	15.50	15.52	15.36	....	15.09	15.43	15.23	....
W14	15.87	15.87	15.87	....	15.87	15.87	15.87	....
W15	15.35	15.36	15.36	....	15.35	15.35	15.36	....
W16	15.95	15.95	15.95	....	15.95	15.95	15.95	....
W17	15.56	15.57	15.56	....	15.55	15.57	15.57	....
W18	15.97	15.97	15.97	....	15.97	15.97	15.97	....
W19	15.69	15.70	15.69	....	15.69	15.70	15.70	....
W20	15.97	15.97	15.97	....	15.97	15.97	15.97	....
W21	15.79	15.80	15.80	....	15.79	15.80	15.80	....
W22	15.98	15.98	15.98	....	15.98	15.98	15.98	....
W23	15.84	15.85	15.85	....	15.84	15.85	15.85	....
W24	15.97	15.97	15.97	....	15.97	15.97	15.97	....
W25	15.89	15.90	15.90	....	15.89	15.90	15.90	....
W26	15.97	15.97	15.97	....	15.97	15.97	15.97	....
W27	15.92	15.93	15.92	....	15.92	15.92	15.92	....
W28	15.97	15.97	15.97	....	15.97	15.97	15.97	....
W29	15.91	15.93	15.93	....	15.93	15.93	15.93	....
W30	15.97	15.97	15.97	....	15.97	15.97	15.97	....

Table 7.32: ELF instruction configuration (SEL , c.ne , write-read)

A streaming multiprocessor is able to execute a maximum (MT) of 16 ELF SEL.c.ne instructions per functional unit clock cycle. This implies that the ELF SEL.c.ne instructions are executed by the 2 groups of 16 CUDA cores. The ELF SEL.c.ne instruction's latency (IL or write-read latency WRL) cannot be greater than 12 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	DD01	DD02	DD03	DD04	DD05	DD06	....
W01	1.59	3.15	4.62	**5.07**	**5.09**	**5.07**	....
W02	3.18	6.29	9.24	10.14	10.17	10.14	....
W03	4.77	9.44	13.82	15.22	15.26	15.22	....
W04	6.36	12.56	**15.61**	**15.80**	**15.91**	**15.91**	....
W05	7.95	15.28	**15.44**	**15.73**	**15.73**	**15.75**	....
W06	9.53	**15.91**	**15.98**	**15.97**	**15.97**	**15.98**	....
W07	11.12	**15.37**	**15.86**	**15.90**	**15.90**	**15.90**	....
W08	12.70	**15.98**	**15.97**	**15.97**	**15.98**	**15.98**	....
W09	14.22	**15.81**	**15.93**	**15.93**	**15.93**	**15.93**	....
W10	15.81	**15.98**	**15.98**	**15.99**	**15.99**	**15.99**	....
W11	15.89	**15.85**	**15.94**	**15.94**	**15.94**	**15.94**	....
W12	**15.95**	**15.95**	**15.99**	**15.99**	**15.99**	**15.99**	....
W13	**15.46**	**15.89**	**15.95**	**15.95**	**15.94**	**15.95**	....
W14	**15.98**	**15.96**	**15.99**	**15.99**	**15.99**	**15.99**	....
W15	**15.69**	**15.94**	**15.95**	**15.95**	**15.95**	**15.95**	....
W16	**15.98**	**15.98**	**15.99**	**15.99**	**15.99**	**15.99**	....
W17	**15.81**	**15.95**	**15.95**	**15.96**	**15.95**	**15.95**	....
W18	**15.98**	**15.98**	**15.99**	**15.99**	**15.99**	**15.99**	....
W19	**15.87**	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	....
W20	**15.97**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	....
W21	**15.59**	**15.95**	**15.95**	**15.95**	**15.95**	**15.95**	....
W22	**15.94**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	....
W23	**15.90**	**15.95**	**15.95**	**15.96**	**15.95**	**15.96**	....
W24	**15.97**	**15.99**	**15.99**	**15.99**	**15.99**	**15.99**	....
W25	**15.74**	**15.96**	**15.96**	**15.96**	**15.95**	**15.96**	....
W26	**15.95**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	....
W27	**15.90**	**15.96**	**15.96**	**15.96**	**15.95**	**15.96**	....
W28	**15.97**	**15.98**	**15.97**	**15.97**	**15.97**	**15.97**	....
W29	**15.81**	**15.96**	**15.96**	**15.96**	**15.96**	**15.96**	....
W30	**15.96**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	....
W31	**15.94**	**15.97**	**15.96**	**15.96**	**15.96**	**15.97**	....
W32	**15.98**	**15.99**	**15.99**	**15.99**	**15.99**	**15.98**	....

Table 7.33: ELF instruction configuration (SEL , c.ne , read-read)

The warp's latency (WL) for the ELF SEL.c.ne instruction is equal to 3 functional unit clock cycles (16 / 5.09). Re-reading any PTX operand register (read-read latency RRL) does not require more than 6 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	DD01	DD02	DD03	DD04	DD05	DD06	....
W01	5.09	5.09	5.07	5.07	5.09	5.07	....
W02	10.18	10.18	10.14	10.14	10.17	10.14	....
W03	15.27	15.27	15.21	15.21	15.26	15.21	....
W04	15.68	15.69	15.73	15.69	15.67	15.69	....
W05	15.80	15.78	15.76	15.76	15.74	15.75	....
W06	15.97	15.98	15.97	15.98	15.97	15.98	....
W07	15.91	15.91	15.90	15.90	15.89	15.91	....
W08	15.98	15.98	15.98	15.98	15.98	15.98	....
W09	15.94	15.94	15.94	15.94	15.94	15.94	....
W10	15.99	15.99	15.99	15.99	15.99	15.99	....
W11	15.94	15.95	15.94	15.94	15.94	15.94	....
W12	15.99	15.99	15.99	15.99	15.99	15.99	....
W13	15.95	15.95	15.95	15.95	15.95	15.95	....
W14	15.99	15.99	15.99	15.99	15.99	15.99	....
W15	15.95	15.95	15.95	15.95	15.95	15.95	....
W16	15.99	15.99	15.99	15.99	15.99	15.99	....
W17	15.95	15.96	15.95	15.96	15.96	15.96	....
W18	15.99	15.99	15.99	15.99	15.99	15.99	....
W19	15.95	15.96	15.96	15.95	15.95	15.96	....
W20	15.99	15.99	15.99	15.99	15.99	15.99	....
W21	15.95	15.96	15.96	15.96	15.95	15.96	....
W22	15.99	15.99	15.99	15.99	15.99	15.99	....
W23	15.96	15.96	15.95	15.95	15.96	15.95	....
W24	15.99	15.99	15.99	15.99	15.99	15.99	....
W25	15.96	15.95	15.95	15.96	15.96	/////	....
W26	15.99	15.99	15.99	15.99	15.99	/////	....
W27	15.95	15.96	15.96	15.95	15.96	/////	....
W28	15.99	15.98	15.99	15.99	15.99	/////	....
W29	15.96	15.95	15.95	15.96	/////	/////	....
W30	15.99	15.98	15.98	15.99	/////	/////	....
W31	15.97	15.97	15.96	/////	/////	/////	....
W32	15.98	15.98	15.98	/////	/////	/////	....

Table 7.34: ELF instruction configuration (IMAD.U32.U32.X , n , write-read)

A streaming multiprocessor is able to execute a maximum (MT) of 16 ELF IMAD.U32.U32.X.n instructions per functional unit clock cycle. This implies that the ELF IMAD.U32.U32.X.n instructions are executed by the 2 groups of 16 CUDA cores. The ELF ELF IMAD.U32.U32.X.n instruction's latency (IL or write-read latency WRL) cannot be greater than 12 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	1.45	2.87	4.23	5.09	5.09	5.09	5.09	5.09	....
W02	2.89	5.74	8.47	10.18	10.17	10.18	10.18	10.18	....
W03	4.34	8.60	12.67	15.20	15.26	15.25	15.16	15.21	....
W04	5.78	11.44	15.54	15.66	15.88	15.86	15.87	15.84	....
W05	7.23	14.21	14.95	15.49	15.42	15.45	15.47	15.49	....
W06	8.67	15.93	15.97	15.97	15.97	15.97	15.97	15.97	....
W07	10.12	15.36	15.73	15.83	15.81	15.82	15.82	15.82	....
W08	11.56	15.97	15.96	15.98	15.98	15.97	15.98	15.98	....
W09	12.97	15.62	15.87	15.87	15.87	15.88	15.88	15.87	....
W10	14.39	15.98	15.98	15.98	15.98	15.98	15.98	15.98	....
W11	15.51	15.80	15.89	15.89	15.89	15.89	15.90	15.89	....
W12	15.89	15.94	15.98	15.99	15.99	15.99	15.99	15.99	....
W13	15.13	15.84	15.91	15.91	15.91	15.91	15.92	15.92	....
W14	15.96	15.95	15.98	15.98	15.98	15.98	15.98	15.98	....
W15	15.47	15.91	15.93	15.93	15.93	15.93	15.93	15.94	....
W16	15.98	15.97	15.98	15.98	15.98	15.98	15.98	15.98	....
W17	15.52	15.94	15.94	15.93	15.94	15.94	15.94	15.94	....
W18	15.98	15.98	15.98	15.98	15.98	15.98	15.98	15.98	....
W19	15.70	15.94	15.93	15.93	15.93	15.93	15.93	15.93	....
W20	15.98	15.98	15.98	15.98	15.98	15.98	15.98	15.98	....
W21	15.45	15.94	15.94	15.93	15.94	15.94	15.94	15.93	....
W22	15.91	15.98	15.98	15.98	15.98	15.98	15.98	15.98	....
W23	15.87	15.94	15.94	15.93	15.93	15.94	15.94	15.94	....
W24	15.96	15.99	15.98	15.98	15.98	15.98	15.98	15.98	....
W25	15.64	15.94	15.94	15.94	15.94	15.94	15.94	15.94	....
W26	15.94	15.98	15.98	15.98	15.98	15.98	15.98	15.98	....
W27	15.89	15.94	15.95	15.94	15.95	15.95	15.95	15.94	....
W28	15.97	15.98	15.98	15.98	15.98	15.98	15.98	15.98	....
W29	15.75	15.95	15.95	15.94	15.95	15.94	15.93	15.95	....
W30	15.95	15.99	15.99	15.99	15.99	15.99	15.99	15.99	....
W31	15.92	15.95	15.95	15.95	15.95	15.95	15.95	/////	....
W32	15.97	15.98	15.98	15.98	15.98	15.98	15.98	/////	....

Table 7.35: ELF instruction configuration (IMAD.U32.U32.X , n , read-read)

The warp's latency (WL) for the ELF IMAD.U32.U32.X.n instruction is equal to 3 functional unit clock cycles (16 / 5.09). Re-reading any ELF operand register (read-read latency RRL) does not require more than 6 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	5.09	5.09	5.09	5.09	5.09	5.09	5.09	5.09	....
W02	10.18	10.18	10.18	10.17	10.17	10.18	10.18	10.18	....
W03	15.18	15.18	15.17	15.21	15.26	15.26	15.26	15.26	....
W04	15.53	15.54	15.54	15.54	15.53	15.59	15.54	15.53	....
W05	15.72	15.51	15.43	15.50	15.45	15.45	15.46	15.48	....
W06	15.97	15.97	15.97	15.97	15.97	15.97	15.97	15.97	....
W07	15.90	15.87	15.82	15.82	15.82	15.81	15.82	15.83	....
W08	15.97	15.97	15.97	15.97	15.97	15.97	15.97	15.97	....
W09	15.88	15.86	15.87	15.86	15.89	15.87	15.89	15.88	....
W10	15.98	15.98	15.98	15.98	15.98	15.98	15.98	15.98	....
W11	15.92	15.89	15.89	15.89	15.89	15.90	15.89	15.90	....
W12	15.99	15.99	15.98	15.98	15.99	15.98	15.99	15.98	....
W13	15.92	15.91	15.92	15.91	15.92	15.92	15.92	15.92	....
W14	15.99	15.98	15.98	15.98	15.98	15.98	15.98	15.98	....
W15	15.95	15.94	15.93	15.93	15.93	15.94	15.94	15.93	....
W16	15.99	15.98	15.98	15.98	15.98	15.98	15.98	15.98	....
W17	15.92	15.93	15.93	15.93	15.93	15.92	15.94	15.93	....
W18	15.99	15.98	15.99	15.99	15.99	15.99	15.98	15.99	....
W19	15.92	15.93	15.92	15.94	15.93	15.92	15.91	15.93	....
W20	15.98	15.98	15.98	15.98	15.98	15.98	15.98	15.98	....
W21	15.94	15.94	15.94	15.94	15.94	15.93	15.94	15.95	....
W22	15.98	15.98	15.98	15.98	15.98	15.98	15.98	15.98	....
W23	15.95	15.94	15.95	15.94	15.94	15.94	15.95	15.94	....
W24	15.99	15.99	15.99	15.98	15.99	15.99	15.98	15.99	....
W25	15.94	15.94	15.94	15.94	15.94	/////	/////	/////	....
W26	15.98	15.98	15.98	15.98	15.98	/////	/////	/////	....
W27	15.94	15.95	15.95	15.94	15.95	/////	/////	/////	....
W28	15.99	15.99	15.98	15.99	15.98	/////	/////	/////	....
W29	15.95	15.95	15.95	15.94	/////	/////	/////	/////	....
W30	15.99	15.99	15.99	15.98	/////	/////	/////	/////	....
W31	15.95	15.95	15.95	/////	/////	/////	/////	/////	....
W32	15.98	15.98	15.98	/////	/////	/////	/////	/////	....

Table 7.36: ELF instruction configuration (IMAD.U32.U32 , n , write-read)

.

A streaming multiprocessor is able to execute a maximum (MT) of 16 ELF IMAD.U32.U32.n instructions per functional unit clock cycle. This implies that the ELF IMAD.U32.U32.n instructions are executed by the 2 groups of 16 CUDA cores. The ELF IMAD.U32.U32.n instruction's latency (IL or write-read latency WRL) cannot be greater than 6 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	1.45	2.87	4.23	**5.09**	**5.09**	**5.09**	**5.09**	**5.09**	....
W02	10.18	10.17	10.18	10.18	10.17	10.18	10.18	10.18	....
W03	15.17	15.19	15.18	15.21	15.26	15.26	15.25	15.21	....
W04	15.43	15.44	15.44	15.44	15.48	15.43	15.49	15.44	....
W05	15.72	15.51	15.42	15.50	15.46	15.45	15.49	15.49	....
W06	**15.97**	**15.97**	**15.97**	**15.97**	**15.97**	**15.97**	**15.97**	**15.97**	....
W07	**15.90**	**15.87**	**15.82**	**15.83**	**15.82**	**15.83**	**15.82**	**15.81**	....
W08	**15.97**	**15.97**	**15.97**	**15.98**	**15.98**	**15.97**	**15.98**	**15.98**	....
W09	**15.88**	**15.86**	**15.87**	**15.86**	**15.85**	**15.87**	**15.88**	**15.87**	....
W10	**15.98**	**15.99**	**15.98**	**15.98**	**15.98**	**15.99**	**15.98**	**15.98**	....
W11	**15.92**	**15.88**	**15.94**	**15.89**	**15.90**	**15.90**	**15.89**	**15.89**	....
W12	**15.99**	**15.98**	**15.99**	**15.98**	**15.99**	**15.99**	**15.99**	**15.99**	....
W13	**15.92**	**15.91**	**15.92**	**15.91**	**15.91**	**15.90**	**15.92**	**15.89**	....
W14	**15.99**	**15.98**	**15.99**	**15.99**	**15.98**	**15.99**	**15.98**	**15.99**	....
W15	**15.95**	**15.94**	**15.93**	**15.92**	**15.93**	**15.94**	**15.93**	**15.94**	....
W16	**15.98**	**15.97**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	....
W17	**15.52**	**15.94**	**15.94**	**15.93**	**15.94**	**15.94**	**15.94**	**15.94**	....
W18	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	....
W19	**15.70**	**15.94**	**15.93**	**15.93**	**15.93**	**15.93**	**15.93**	**15.93**	....
W20	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	....
W21	**15.45**	**15.94**	**15.94**	**15.93**	**15.94**	**15.94**	**15.94**	**15.93**	....
W22	**15.91**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	....
W23	**15.87**	**15.94**	**15.94**	**15.94**	**15.93**	**15.94**	**15.94**	**15.94**	....
W24	**15.96**	**15.99**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	....
W25	**15.64**	**15.94**	**15.94**	**15.93**	**15.94**	**15.94**	**15.94**	**15.94**	....
W26	**15.94**	**15.98**	**15.98**	**15.99**	**15.98**	**15.98**	**15.98**	**15.98**	....
W27	**15.89**	**15.94**	**15.94**	**15.94**	**15.95**	**15.95**	**15.95**	**15.94**	....
W28	**15.97**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	**15.98**	....
W29	**15.75**	**15.95**	**15.95**	**15.94**	**15.96**	**15.94**	**15.93**	**15.95**	....
W30	**15.95**	**15.99**	**15.99**	**15.98**	**15.99**	**15.99**	**15.99**	**15.99**	....
W31	**15.92**	**15.95**	**15.94**	**15.95**	**15.95**	**15.95**	**15.95**	/////	....
W32	**15.97**	**15.98**	**15.98**	**15.99**	**15.98**	**15.98**	**15.98**	/////	....

Table 7.37: ELF instruction configuration (IMAD.U32.U32 , n , read-read)

The warp's latency (WL) for the ELF IMAD.U32.U32 instruction is equal to 3 functional unit clock cycles (16 / 5.09). Re-reading any PTX operand register (read-read latency RRL) does not require more than 6 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	5.09	5.09	5.09	5.09	5.09	5.09	5.09	5.09	....
W02	10.18	10.18	10.18	10.17	10.17	10.18	10.18	10.18	....
W03	15.18	15.18	15.17	15.21	15.26	15.26	15.26	15.26	....
W04	15.33	15.34	15.34	15.34	15.33	15.39	15.34	15.33	...
W05	15.72	15.51	15.43	15.50	15.45	15.45	15.46	15.48	....
W06	15.97	15.97	15.97	15.97	15.97	15.97	15.97	15.97	....
W07	15.90	15.87	15.82	15.82	15.82	15.81	15.82	15.83	....
W08	15.97	15.97	15.97	15.97	15.97	15.97	15.97	15.97	....
W09	15.88	15.86	15.87	15.86	15.89	15.87	15.89	15.88	....
W10	15.98	15.98	15.98	15.98	15.98	15.98	15.98	15.98	....
W11	15.92	15.89	15.89	15.89	15.89	15.90	15.89	15.90	....
W12	15.99	15.99	15.98	15.98	15.99	15.98	15.99	15.98	....
W13	15.92	15.91	15.92	15.91	15.92	15.92	15.92	15.92	....
W14	15.99	15.98	15.98	15.98	15.99	15.98	15.98	15.98	....
W15	15.95	15.94	15.93	15.93	15.93	15.92	15.94	15.93	....
W16	15.99	15.98	15.98	15.98	15.99	15.98	15.98	15.98	....
W17	15.92	15.93	15.93	15.92	15.93	15.92	15.94	15.93	....
W18	15.99	15.98	15.99	15.98	15.99	15.99	15.98	15.99	....
W19	15.92	15.93	15.92	15.93	15.93	15.92	15.91	15.93	....
W20	15.98	15.98	15.98	15.98	15.99	15.98	15.98	15.98	....
W21	15.94	15.94	15.94	15.94	15.92	15.93	15.94	15.95	....
W22	15.98	15.98	15.98	15.99	15.98	15.99	15.98	15.98	....
W23	15.95	15.93	15.95	15.94	15.93	15.94	15.95	15.94	....
W24	15.99	15.99	15.99	15.99	15.99	15.97	15.98	15.99	....
W25	15.94	15.94	15.93	15.92	15.94	/////	/////	/////	....
W26	15.98	15.98	15.99	15.98	15.98	/////	/////	/////	....
W27	15.94	15.95	15.93	15.94	15.95	/////	/////	/////	....
W28	15.99	15.99	15.98	15.98	15.98	/////	/////	/////	....
W29	15.95	15.95	15.95	15.94	/////	/////	/////	/////	....
W30	15.99	15.99	15.99	15.98	/////	/////	/////	/////	....
W31	15.96	15.95	15.95	/////	/////	/////	/////	/////	....
W32	15.98	15.98	15.98	/////	/////	/////	/////	/////	....

What the autonomic tools discovered is that different PTX and ELF instructions usually have different maximum throughputs, warp latencies, write-read latencies for their result registers, and read-read latencies for their operand registers. Furthermore, there are instructions, such as the PTX setp.gt.s32.n instruction configuration, which require only one ELF instruction for their execution, but have different maximum throughputs and warp latencies.

Table 7.38: Summary table for the PTX and ELF cases

Id is the case identifier (see table 7.11). PI is the PTX instruction, and CEI are the corresponding ELF instructions (the ELF instructions that are necessary to execute the PTX instruction, see table 7.11). MT is the maximum throughput (the maximum number of instructions executed per functional unit clock cycle by a streaming multiprocessor). WL is the warp latency (the number of functional unit clock cycles that are necessary to wait before a warp scheduler can reschedule the warp for the execution of the next instruction). WRL is the write-read latency (the number of functional units clock cycles that are necessary to wait before being able to read, and therefore reuse, the register used to store the result of an instruction). RL is the read-read latency (the number of functional unit clock cycles that are necessary to wait before being able to re-read a register used as an operand in an instruction). Table 7.16 describes the formulas used to determine MT, WL, WRL and RRL.

Id	PI	MT	WL	WRL	RRL	CEI	MT	WL	WRL	RRL
1	add (s32.n)	32	6	≤ 24	≤ 6	IADD (s32.n)	32	6	≤ 24	≤ 6
2	setp.gt (s32.n.wr)	8	14	≤ 14	//	ISETP.GT.AND (s32.n.wr)	8	13	≤ 14	//
	setp.gt (s32.n.rr)	32	6	//	≤ 28	ISETP.GT.AND (s32.n.wr)	32	6	//	≤ 28
3	popc (b32.n)	16	3	≤ 12	≤ 6	POPC (b32.n)	16	3	≤ 12	≤ 6
4	max (s64.c.no)	4	8	≤ 8	≤ 8	IMMMX.XHI (s32.c.ne)	16	13	≤ 14	≤ 14
						IMMMX.U32.XL (s32.c.ne)	16	3	≤ 12	≤ 6
						SEL (s32.c.no)	16	3	≤ 12	≤ 6
						SEL (s32.c.no)	16	3	≤ 12	≤ 6
5	mul.lo (s64.c.ne)	4	13	≤ 14	≤ 14	IMUL.U32.U32 (s32.n)	16	3	≤ 12	≤ 6
						IMAD.U32.U32.HI.X (s32.n)	16	13	≤ 14	≤ 14
						SEL (s32.c.ne)	16	3	≤ 12	≤ 6
						IMAD.U32.U32.X (s32.n)	16	3	≤ 12	≤ 6
						IMAD.U32.U32 (s32.n)	16	3	≤ 6	≤ 6
						SEL (s32.c.ne)	16	3	≤ 12	≤ 6

7.4 Summary

In this chapter we presented the results of the autonomic tools' discovery and quantification of the low level architectural features and machine behaviors. These results are important for understanding how to efficiently optimize codes. The most important points to remember from this chapter are the following:

- The gigathread scheduler does not always fairly distribute the thread blocks to the streaming multiprocessors. To produce a fair distribution, the number of hardware registers per thread block, when multiplied by the number of thread blocks that the user desires in each streaming multiprocessor, has to be greater than half the number of hardware registers of a streaming multiprocessor, and smaller than the total number of hardware registers of a streaming multiprocessor.

  Furthermore, no more than 8 thread blocks can reside at any time in a streaming multiprocessor, and the maximum total shared memory of each streaming multiprocessor is 48 KB.

  After choosing the number of hardware registers per thread (which is possible thanks to our reverse engineering), and remembering the previous constraints, we can then automatically generate all the launch configurations that will force the gigathread scheduler to fairly distribute the thread blocks to the streaming multiprocessors. However, these distributions are not always deterministic. This non-determinism can easily generate problems about enhancing data locality.

  What the autonomic tools discovered is that if users want to produce fair and deterministic distributions, then the number of thread blocks used to execute a GPU code has to be equal to the number of streaming multiprocessors, and all the previous constraints have to be satisfied.

- It is reasonable to infer that the warp schedulers' policy is a round robin one. It does not matter how many instructions the ELF kernels execute, how many times we launch

the kernels, and what launch configurations we use. However, all the warps essentially enter into and exit from the for loop together.

- Different PTX and ELF instructions have different maximum throughputs, warp latencies, write-read latencies for their result registers, and read-read latencies for their operand registers. The discovery and quantification of these features is important for determining the minimum number of warps that must reside in a streaming multiprocessor to achieve a highly efficient code.

- It is always better to avoid any launch configurations that imply an odd number of warps in one or more streaming multiprocessors. This is because, even if the ELF kernels cannot be slowed down by the memories' bandwidths and latencies, the autonomic tools discovered that an odd number of warps in a streaming multiprocessor creates local instabilities for the execution of any instruction in that multiprocessor. It is not clear why this happens, but it is notable that there are 2 warp schedulers in each streaming multiprocessor.

Chapter 8

Efficiency Losses and Low Efficacies

The autonomic tools reverse engineered the real ISAs and also discovered and quantified the GPU low level architectural features and machine behaviors. Now we can therefore quantify the performance loss incurred when a user programs using ELF, but does not know the quantification of the low level architectural features and machine behaviors. In addition, we can also quantify the performance loss incurred when a user programs using PTX, exploits the compiler for the transformation of even simple PTX codes into ELF codes, and does not know the quantification of the low level architectural features and machine behaviors.

8.1 (ELF + Insight) Vs ELF

Even the generation of very simple ELF kernels can easily produce large performance losses. This is because if the user does not know and quantify the undisclosed low level architectural features and machine behaviors, then it is difficult and very time consuming to effectively produce efficient codes. To demonstrate this, let us consider the following cases that are divided per type:

- $Cases_{t1}$: After the reverse engineering phase, a user can produce any desired ELF code but he/she does not know anything yet about the undisclosed low level architectural features and machine behaviors. In the worst case scenario of this situation, a user

could execute the ELF kernels using only one thread block with only one warp.

Let us assume that this is the scenario and, among the many possible examples, let us consider the microbenchmark ELF kernel produced for the PTX instruction configuration sub-case add.s32.n.wr.1. With only 1 warp in only 1 streaming multiprocessor, the average number of add.s32 instructions that the machine will execute per functional unit clock cycle will be equal to 1.59, see cell (W01 , DD1) in Table 8.3.

However, thanks to the discovery and quantification of the low level architectural features and machine behaviors, we know: 1) that each PTX add.s32 instruction is transformed into only one ELF IADD instruction; 2) that the maximum average number of PTX add.s32 instructions that a streaming multiprocessor can execute per functional unit clock cycle is equal to 32, and that therefore the PTX add.s32 instructions are executed by the 2 groups of 16 CUDA cores; 3) that there are 14 streaming multiprocessors in a Tesla C2070, 448 (i.e. $32 \cdot 14$) CUDA cores, and that each CUDA core can execute 1 PTX add.s32 instruction per functional unit clock cycle.

Therefore, the ELF code generated by a user with no knowledge of the low level architectural features and machine behaviors would have an efficiency equal to only 0.35%; see cell (W01 , DD1) in Table 8.4. Furthermore, such code would be 2 orders of magnitude slower than the same code when running using 14 thread blocks, with 28 warps per thread block, and a number of hardware registers per thread block that is greater than half the number of hardware registers of each streaming multiprocessor, but that is smaller than the total number of hardware registers of each streaming multiprocessor.

Next, let us launch the microbenchmark ELF kernel produced for the PTX instruction configuration sub-case add.s32.n.wr.1 using 1 thread block but more than 1 warp (this is the first column of the Table 8.3). If, for example, the user decides to use 3 warps, then the average number of PTX add.s32 instructions that the machine will execute

per functional unit clock cycle will be equal to 4.77, (see cell [W03 , DD1] in Table 8.3), and therefore the code efficiency will be equal to 1.06%, (see cell [W03 , DD1] in Table 8.4).

Furthermore, repeating the procedure with a number between 1 and 32 warps (32 is the maximum number of warps per thread block), and using all the ELF kernels generated for the PTX instruction configuration (add.s32 , n , write-read), the codes' efficiencies would be never greater than 7.14% (see Table 8.4).

- $Cases_{t2}$: Executing the launch procedure used for $Cases_{t1}$, but focusing on only 1 streaming multiprocessor, we can analyze the performance losses that a user would face at local streaming multiprocessor level with no knowledge of the low level architectural features and machine behaviors.

  More than 59% of the (ELF kernels , launch configuration) pairs produced for the PTX instruction configuration (add.s32 , n , write-read) lose at least 6% of the total performance, and some pairs have performance losses equal to 95% of the total performance (see the description at pages 111-112 for an explanation, Table 8.6 for the summary statistics, and Table 8.13 for the performance loss distribution).

- $Cases_{t3}$: any ELF kernel can be run using a number of thread blocks greater than 1. In this case, the number of launch configurations that can be used to launch an ELF kernel is simply staggering (it is greater than $2^{30} \cdot 1024$), and allows the generation of any performance loss. This is because, even when producing the desired ELF kernels, the user does not know anything about the gigathread scheduler's distribution policies, and therefore it is very easy to achieve load unbalancing in some streaming multiprocessors (some streaming multiprocessors can easily get a number of thread blocks that is greater than the number of fair thread blocks per streaming multiprocessor).

  However, when a thread block is assigned to a streaming multiprocessor, the thread block cannot migrate to another streaming multiprocessor. Furthermore, the codes

are regular (without divergences), and so each thread will execute the same number of ELF instructions.

Therefore, when some streaming multiprocessors are assigned a greater number of thread blocks than the number of fair thread blocks per streaming multiprocessor, some will execute a greater number of instructions compared to others, but the streaming multiprocessors themselves are equal, and so have equivalent computational power. This is true even if the local efficiencies of the streaming multiprocessors with the greatest number of thread blocks are equal to 100%.

For example, for all the subcases in Table 7.2, each ELF kernel is run using 14 thread blocks and the fair number of thread blocks per streaming multiprocessor is equal to 1. However, some streaming multiprocessors will get 2 thread blocks and others will get 0 thread blocks and therefore the performance loss per case can easily be greater than 50%, see Tables 8.16-8.17 for the PTX instruction configuration (or.b32 , n , write-read).

- $Cases_{t4}$: thanks to the discovery and quantification of the low level architectural features, we know that the gigathread scheduler will fairly distribute the thread blocks to the streaming multiprocessors if we use the launch configurations generated by procedure 7.8. This is because we can assign a greater number of hardware registers to each thread, this even if the thread will later use only some of the registers for code executions.

  Furthermore, we can force a fair distribution using any launch configuration with a number of thread blocks equal to the number of streaming multiprocessors, and a number of warps per thread block equal to or greater than 9. This is because the maximum number of hardware registers per thread is equal to 64, and therefore we need at least 9 warps per thread block to generate thread blocks that have the required number of hardware registers (i.e. more than half the number of hardware registers

of a streaming multiprocessor; see procedure 8.18). Performance losses against these cases are greater than those against cases of type 3; see Tables 8.29-8.31 for the PTX instruction configuration (and.b32 , n , write-read).

Table 8.1: Legend for the experiments

```
PIC: PTX instruction configuration (e.g. [ xor.b32 , n , read-read ] )
EIC: ELF instruction configuration (e.g. [ IADD , c.no , write-read ] )

OK: original kernels. The kernels are exactly those used for the disco-
    very and quantification of the low level architectural features and
    machine behaviors.
MK: modified kernels. We modified each kernel by assigning to each thread
    a greater number of hardware registers. This is necessary for the ca-
    ses of type four because we want to force the gigathread scheduler to
    fairly distribute the thread blocks to the streaming multiprocessors.

WX: number of warps per thread block ( WX ranges from 1 to 32 )
TB: number of thread blocks used for the launches ( usually 1 or 14 )
SM: streaming multiprocessors (1 or K x 14 with K integer and positive )
```

Table 8.2: First group of experiments

Exp	Type of Case	PTX Instruction Configuration	Tables	Pages
1	1	(add.s32 , n , wr)	8.3	109
			8.4	110
2	2	(add.s32 , n , wr)	descriptions	111 - 112
			8.7 - 8.12	113 - 118
			8.13	119
3	3	(or.b32 , n , wr)	8.14 - 8.15	120 - 121
			8.16 - 8.17	122 - 123
4	4	(and.b32 , n , wr)	descriptions	124 - 125
			8.20 - 8.22	126 - 128
			8.23 - 8.25	129 - 131
			8.26 - 8.28	132 - 134
			8.29 - 8.31	135 - 137

Table 8.3: Throughputs - PIC (add.s32 , n , write-read) - 1 SM - OK

A streaming multiprocessor is able to execute a maximum (MT) of 32 PTX add.s32.n instructions per functional unit clock cycle. This implies that the PTX add.s32.n instructions are executed by the 2 groups of 16 CUDA cores. The PTX add.s32.n instruction's latency (IL or write-read latency WRL) cannot be greater than 24 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red. WX is the number of warps that reside in the unique streaming multiprocessor used, while DDY is the dependence distance among instructions. For an explanation of the colors and symbols, see Tables 7.14, 7.15, and 7.16.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	1.59	3.15	4.62	5.07	5.09	5.07	5.09	5.07	....
W02	3.18	6.29	9.24	10.14	10.17	10.14	10.18	10.14	....
W03	4.77	9.44	13.82	15.21	15.25	15.21	15.27	15.21	....
W04	6.36	12.58	18.43	20.28	20.34	20.28	20.36	20.28	....
W05	7.94	15.44	21.45	25.13	25.01	25.12	25.13	25.13	....
W06	9.53	18.54	25.67	31.12	31.26	31.13	31.15	31.12	....
W07	11.12	21.70	26.90	26.87	26.87	26.88	26.89	26.88	....
W08	12.70	24.66	31.44	31.29	31.37	31.69	31.71	31.68	....
W09	14.28	26.26	27.91	27.88	27.94	27.94	27.97	27.89	....
W10	15.86	28.14	31.56	31.54	31.45	31.56	31.23	31.57	....
W11	17.43	28.31	28.45	28.38	28.41	28.43	28.41	28.38	....
W12	19.01	31.27	31.35	31.76	31.37	31.56	31.76	31.78	....
W13	20.52	28.75	28.79	28.79	28.81	28.80	28.83	28.80	....
W14	22.07	31.32	31.40	31.89	31.52	31.68	31.53	31.78	....
W15	23.33	29.02	29.03	29.06	29.07	29.05	29.06	29.05	....
W16	24.83	31.78	31.56	31.96	31.76	31.36	31.57	31.78	....
W17	25.49	29.29	29.33	29.28	29.31	29.31	29.32	29.29	....
W18	26.99	31.27	31.26	31.99	31.52	31.35	31.47	31.78	....
W19	27.54	29.42	29.42	29.43	29.50	29.47	29.48	29.45	....
W20	28.94	31.35	31.57	31.95	31.45	31.47	31.54	31.87	....
W21	27.47	29.55	29.58	29.64	29.65	29.65	29.67	29.64	....
W22	28.80	31.65	31.56	31.97	31.76	31.56	31.76	31.98	....
W23	29.29	29.69	29.70	29.79	29.78	29.79	29.79	29.79	....
W24	31.08	31.27	31.76	31.86	31.67	31.76	31.64	31.76	....
W25	27.98	29.80	29.83	29.89	29.87	29.88	29.87	29.87	....
W26	31.14	31.45	31.98	31.98	31.74	31.98	31.86	31.89	....
W27	29.08	29.87	29.92	29.92	29.95	29.95	29.97	29.96	....
W28	31.21	31.53	31.98	31.97	31.76	31.98	31.78	31.98	....
W29	28.48	29.93	30.01	30.03	30.02	30.04	30.04	30.02	....
W30	31.51	31.47	31.98	31.97	31.67	31.94	31.89	31.97	....
W31	30.19	30.04	30.03	30.03	30.07	30.06	30.07	/////	....
W32	31.75	31.65	31.98	31.93	31.76	31.99	31.93	/////	....

Table 8.4: Codes' efficiencies - PIC (add.s32 , n , write-read) - 14 SMs - OK

The code efficiency percentages for the ELF kernels generated for the PTX instruction configuration (add.s32 , n , write-read) when each ELF kernel is launched using only 1 thread block and only one streaming multiprocessor. For these tables, we supposed that a user could produce the desired ELF kernels thanks to the reverse engineering, but that he/she would not have any knowledge and quantification of the undisclosed low level architectural features (e.g. warp latency and write-read dependence latency) of the PTX instruction configuration (add.s32 , n , write-read). When using only one tread block the code efficiency was never greater than 7.14%, even if the ELF code is very simple and if the user has complete control over the ELF instructions executed by the GPU. Furthermore, in the worst case scenario (one thread block with one warp), the code efficiency was never greater than 0.35%.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	**0.35**	0.70	1.03	1.13	1.14	1.13	1.14	1.13	....
W02	0.70	1.40	2.06	2.26	2.27	2.26	2.27	2.26	....
W03	1.06	2.11	3.08	3.40	3.40	3.39	3.40	3.39	....
W04	1.42	2.81	4.11	4.53	4.54	4.53	4.54	4.52	....
W05	1.77	3.45	4.79	5.61	5.58	5.61	5.61	5.61	....
W06	2.13	4.14	5.73	6.95	6.98	6.95	6.95	6.95	....
W07	2.48	4.84	6.00	6.00	6.00	6.00	6.00	6.00	....
W08	2.83	5.50	7.02	6.98	7.00	7.01	7.01	7.01	....
W09	3.19	5.86	6.23	6.22	6.24	6.24	6.24	6.23	....
W10	3.54	6.28	7.04	7.04	7.02	7.04	6.63	7.05	....
W11	3.89	6.32	6.35	6.33	6.34	6.35	6.34	6.33	....
W12	4.23	6.98	7.00	7.01	7.00	7.04	7.01	7.01	....
W13	4.58	6.42	6.43	6.43	6.43	6.43	6.44	6.43	....
W14	4.93	6.99	7.00	7.12	7.04	7.01	7.04	7.10	....
W15	5.21	6.48	6.48	6.49	6.49	6.48	6.49	6.48	....
W16	5.54	7.01	7.04	7.13	7.09	7.00	7.05	7.10	....
W17	5.69	6.54	6.55	6.54	6.54	6.54	6.54	6.54	....
W18	6.02	6.98	6.98	**7.14**	7.04	7.00	7.03	7.10	....
W19	6.15	6.57	6.59	6.57	6.58	6.58	6.58	6.57	....
W20	6.46	7.00	7.05	7.13	7.13	7.02	7.04	7.11	....
W21	6.14	6.60	6.61	6.62	6.62	6.62	6.62	6.61	....
W22	6.43	7.07	7.05	**7.14**	7.09	7.05	7.09	**7.14**	....
W23	6.54	6.63	6.63	6.65	6.65	6.65	6.65	6.65	....
W24	6.94	6.98	7.09	7.11	7.07	7.09	7.06	7.09	....
W25	6.25	6.65	6.66	6.67	6.66	6.67	6.67	6.67	....
W26	6.95	7.02	**7.14**	**7.14**	7.09	7.18	7.11	7.12	....
W27	6.49	6.67	6.68	6.68	6.69	6.68	6.69	6.69	....
W28	6.97	7.03	**7.14**	**7.14**	7.09	7.14	7.09	**7.14**	....
W29	6.36	6.68	6.70	6.70	6.70	6.70	6.70	6.70	....
W30	6.81	6.80	6.92	6.91	6.85	7.13	7.12	**7.14**	....

Each ELF kernel generated for the PTX instruction configuration (add.s32 , n , wr) requires 21 hardware registers for the execution of instructions. These instructions are different from the add.s32.n.wr instructions that are inside the for loop. However, each thread can have, at maximum, 64 hardware registers. Therefore, 43 ELF kernels are generated for the PTX instruction configuration add.s32.n.wr. Next, the ELF kernel generated for the dependence distance 43 is discarded because spill load and spill store phenomena occur during its executions. This leaves 42 ELF kernels, one per dependence distance from 1 to 42. Therefore, Tables 8.7-8.12 together have a total of $32 \cdot 42 = 1344$ cells but some cases are impossible. This is because the number of hardware registers needed by the thread block is greater than the number of hardware registers of a streaming multiprocessor (see Tables 8.9-8.12). The number of possible cases is therefore equal to 1028.

Table 8.5: Legend for the sets of cells in the tables.

Sets	Colors	Cardinalities	Percentages
S1	Cyan	413	40.18
S2	Sepia	379	36.87
S3	WildStrawberry	26	2.53
S4	PineGreen	210	20.43

The cells (WX , DDY) can be classified into four different sets. See below for a description of each of the four sets and the tables of the codes' efficiencies for the PTX instruction configuration (add.s32 , n , write-read). The first minimum code efficiency of each set is framed with an oval box, while the first maximum code efficiency of each set is framed with a double box.

The cells of the set S_1 (color **cyan**) are the cells where WX is an even number, $WX \geq WL$ (WL is the warp latency; see Tables 7.14, 7.15, and 7.16), and $WX \cdot DDY \geq WRL$ (WRL is the write-read dependence latency; see Tables 7.14, 7.15, and 7.16)). The cells of the set S_2 (color **sepia**) are the cells where WX is an odd number, $WX \geq WL$, and

$WX \cdot DDY \geq WRL$. The cells of the set S_3 (color **wildstrawberry**) are the cells where $WX \cdot DDY < WRL$ and $WX \geq WL$. Finally the cells of the set S_4 (color **pinegreen**) are the cells where $WX < WL$.

Of course, every case (WX , DDY) has a different probability of happening but many of their efficiencies are low. This is disconcerting because: the ELF codes are very simple; we consider only 1 of the 14 streaming multiprocessors; and, thanks to the reverse engineering, we have complete control of the ELF for the generation of the ELF kernels. In spite of this, only 413 (40.18%) of the total cases reach a performance that is near 100%. Without the discovery and quantification of the low level architectural features and machine behaviors, a user should try all the cases to discover those with an high efficiency.

Furthermore, all the cases in the set S_2 (36.87%) have performance losses ranging from a minimum of 6.0% to a maximum of 16%. Things are not better for the cases in the set S_3 (2.53%), which have performance losses ranging from a minimum of 8.5% to a maximum of 70.2%. Finally, the cases in the set S_4 (20.43%) have performances ranging from a minimum of 21.5% to a maximum of 95%.

Table 8.6: PTX instr. config. (add.s32 , n , wr) - Statistics for the type 2 case.

Sets	Percentages	Minimum Perf. Loss	Maximum Perf. Loss
S1	40.18	0.0%	2.7%
S2	36.87	6.0%	16.0%
S3	2.53	8.5%	70.2%
S4	20.43	21.5%	95.0%

The number of cells (WX , DDY) with a performance that is comparable to the peak performance (maximum cell loss equal 2.7%) is smaller than 41%. 1.07% cells have performance losses between 6% and 4%, 12.13% cells have performance losses between 8% and 6%, 11.75% cells have performance losses between 10% and 8%, more than 35% cells have performance losses that are greater than 10%, and finally more than 21% cells have performance losses that are greater than 20%.

Table 8.7: Codes' efficiencies - PIC (add.s32 , n , write-read) - 1 SM - OK - Part 1

The code efficiency percentages for the ELF kernels generated for the PTX instruction configuration (add.s32 , n , write-read) when each ELF kernel is launched using only 1 thread block and only one streaming multiprocessor. For these tables, we supposed that a user could produce the desired ELF kernels thanks to the reverse engineering, but that he/she would not have any knowledge and quantification of the undisclosed low level architectural features (e.g. warp latency and write-read dependence latency) of the PTX instruction configuration (add.s32 , n , write-read).

	DD01	DD02	DD03	DD04	DD05	DD06	DD07
W01	5.0	9.8	14.4	15.8	15.9	15.8	15.9
W02	9.9	19.7	28.9	31.7	31.7	31.7	31.8
W03	14.9	29.5	43.2	47.5	47.7	47.5	47.7
W04	19.9	39.3	57.6	63.4	63.6	63.4	63.6
W05	24.8	48.3	67.0	78.5	78.2	78.5	78.5
W06	29.8	57.9	80.2	97.3	97.7	97.3	97.3
W07	34.8	67.8	84.0	84.0	84.0	84.0	84.0
W08	39.7	77.0	98.3	97.8	98.0	99.0	99.0
W09	44.6	82.1	87.2	87.1	87.3	87.3	87.4
W10	49.3	87.9	98.7	98.6	98.3	98.7	97.6
W11	54.5	88.5	88.9	88.7	88.8	88.8	88.8
W12	59.4	97.7	97.7	99.2	98.0	98.6	99.3
W13	64.1	89.8	90.0	90.0	90.0	90.0	90.0
W14	69.0	97.9	98.1	99.7	98.5	99.0	98.5
W15	72.9	90.7	90.7	90.8	90.8	90.8	90.8
W16	77.6	99.3	98.6	99.9	99.3	98.0	98.7
W17	79.7	91.5	91.7	91.5	91.6	91.6	91.7
W18	84.3	97.7	97.7	100	98.5	98.0	98.3
W19	86.1	91.9	91.9	92.0	92.2	92.1	92.1
W20	90.4	98.0	98.7	99.8	98.3	98.3	98.6
W21	85.8	92.3	92.4	92.6	92.7	92.7	92.7
W22	90.0	98.9	98.6	99.9	99.3	98.6	99.3
W23	91.5	92.8	92.8	93.1	93.1	93.1	93.1
W24	97.1	97.7	99.3	99.6	99.0	99.3	98.9
W25	87.4	93.1	93.2	93.4	93.3	93.4	93.3
W26	97.3	98.3	99.9	99.9	99.2	99.9	99.6
W27	90.9	93.3	93.5	93.5	93.6	93.6	93.7
W28	97.5	98.5	99.9	99.9	99.3	99.9	99.3
W29	89.0	93.5	93.8	93.8	93.8	93.9	93.9
W30	98.5	98.3	99.9	99.9	99.0	99.8	99.7
W31	94.3	93.9	93.8	93.8	94.0	94.0	94.0
W32	99.2	98.9	99.9	99.8	99.8	100	99.8

Table 8.8: Codes' efficiencies - PIC (add.s32 , n , write-read) - 1 SM - OK - Part 2

The code efficiency percentages for the ELF kernels generated for the PTX instruction configuration (add.s32 , n , write-read) when each ELF kernel is launched using only 1 thread block and only one streaming multiprocessor. For these tables, we supposed that a user could produce the desired ELF kernels thanks to the reverse engineering, but that he/she would not have any knowledge and quantification of the undisclosed low level architectural features (e.g. warp latency and write-read dependence latency) of the PTX instruction configuration (add.s32 , n , write-read).

	DD08	DD09	DD10	DD11	DD12	DD13	DD14
W01	15.9	15.8	15.9	15.9	15.8	15.9	15.8
W02	31.7	31.8	31.7	31.7	31.8	31.7	31.8
W03	47.6	47.7	47.8	47.5	47.6	47.7	47.8
W04	63.4	63.6	63.7	63.6	63.7	63.7	63.5
W05	78.6	78.5	78.5	78.6	78.5	78.5	78.5
W06	97.3	97.4	97.3	97.3	97.3	97.4	97.3
W07	84.0	84.0	84.1	84.0	84.1	84.0	84.1
W08	99.0	99.1	99.2	99.0	99.0	99.1	99.1
W09	87.4	87.3	87.4	87.4	87.3	87.4	87.4
W10	98.4	98.6	97.8	98.6	98.4	98.6	98.7
W11	88.8	88.7	88.8	88.9	88.8	88.9	88.8
W12	99.3	99.2	98.9	99.2	99.1	99.0	98.9
W13	90.0	90.1	90.0	90.1	90.1	90.0	90.0
W14	98.6	99.0	98.7	98.9	99.0	99.1	99.0
W15	90.8	90.8	90.9	90.8	90.9	90.8	90.9
W16	98.9	99.0	99.1	98.9	99.0	99.1	99.0
W17	91.7	91.6	91.7	91.8	91.8	91.7	91.7
W18	98.7	99.0	98.7	99.1	98.9	98.9	90.0
W19	92.1	92.0	92.2	92.3	92.0	92.1	92.2
W20	98.7	98.6	98.7	98.6	98.9	98.8	98.9
W21	92.7	92.9	92.8	92.9	92.8	92.7	92.8
W22	99.2	99.1	98.9	99.0	99.1	98.9	99.1
W23	93.2	93.1	93.2	93.2	93.2	93.2	93.3
W24	99.0	99.1	98.9	99.0	99.1	99.2	98.9
W25	93.4	93.3	93.4	93.5	93.4	93.4	93.3
W26	99.7	99.6	99.7	99.5	99.7	99.6	99.6
W27	93.6	93.5	93.7	93.5	93.6	93.5	93.6
W28	99.7	99.6	99.7	99.5	99.9	99.8	99.7
W29	93.8	93.9	93.8	93.8	93.8	93.8	93.9
W30	99.7	99.6	99.9	99.8	99.7	99.6	99.7
W31	94.0	94.1	94.2	94.0	94.1	94.1	94.0
W32	99.9	99.8	100	99.9	99.7	99.8	99.9

Table 8.9: Codes' efficiencies - PIC (add.s32 , n , write-read) - 1 SM - OK - Part 3

The code efficiency percentages for the ELF kernels generated for the PTX instruction configuration (add.s32 , n , write-read) when each ELF kernel is launched using only 1 thread block and only one streaming multiprocessor. For these tables, we supposed that a user could produce the desired ELF kernels thanks to the reverse engineering, but that he/she would not have any knowledge and quantification of the undisclosed low level architectural features (e.g. warp latency and write-read dependence latency) of the PTX instruction configuration (add.s32 , n , write-read). The ///// cases are impossible. This is because the number of hardware registers needed by the thread block would be greater than the number of hardware registers of each streaming multiprocessor.

	DD15	DD16	DD17	DD18	DD19	DD20	DD21
W01	**15.8**	**15.8**	**15.9**	**15.8**	**15.8**	**15.9**	**15.8**
W02	**31.7**	**31.7**	**31.7**	**31.7**	**31.7**	**31.8**	**31.8**
W03	**47.7**	**47.7**	**47.8**	**47.7**	**47.6**	**47.7**	**47.8**
W04	**63.6**	**63.6**	**63.7**	**63.6**	**63.6**	**63.7**	**63.6**
W05	**78.6**	**78.5**	**78.5**	**78.5**	**78.5**	**78.5**	**78.5**
W06	**97.4**	**97.4**	**97.3**	**97.5**	**97.4**	**97.4**	**97.4**
W07	**84.0**	**84.1**	**84.1**	**84.1**	**84.1**	**84.0**	**84.1**
W08	**99.1**	**99.1**	**99.2**	**99.1**	**99.2**	**99.1**	**99.1**
W09	**87.4**	**87.6**	**87.4**	**87.5**	**87.4**	**87.4**	**87.4**
W10	**98.5**	**98.6**	**97.8**	**98.5**	**98.4**	**98.5**	**98.7**
W11	**88.8**	**88.8**	**88.8**	**88.9**	**88.9**	**88.9**	**88.9**
W12	**99.3**	**99.4**	**98.9**	**99.2**	**99.4**	**99.2**	**98.8**
W13	**90.0**	**90.1**	**90.1**	**90.1**	**90.1**	**90.1**	**90.0**
W14	**98.6**	**99.5**	**98.7**	**98.8**	**99.0**	**99.2**	**99.3**
W15	**90.8**	**90.9**	**90.9**	**90.7**	**90.9**	**90.7**	**90.7**
W16	**98.9**	**99.2**	**99.3**	**99.0**	**99.0**	**99.1**	**99.2**
W17	**91.7**	**91.7**	**91.7**	**91.8**	**91.7**	**91.7**	**91.7**
W18	**98.9**	**99.1**	**98.7**	**99.2**	**98.9**	**99.1**	**90.1**
W19	**92.1**	**92.2**	**92.2**	**92.3**	**92.2**	**92.1**	**92.2**
W20	**98.8**	**98.6**	**98.9**	**98.6**	**99.1**	**98.9**	**99.1**
W21	**92.8**	**92.9**	**92.8**	**92.9**	**92.9**	**92.8**	**92.8**
W22	**99.2**	**99.1**	**99.1**	**99.2**	**99.1**	**99.2**	**99.1**
W23	**93.2**	**93.4**	**93.2**	**93.3**	**93.3**	**93.2**	**93.3**
W24	**99.1**	**99.1**	**99.2**	**99.1**	**99.1**	**99.4**	**99.1**
W25	**93.4**	**93.3**	**93.6**	**93.5**	**93.6**	/////	/////
W26	**99.7**	**99.7**	**99.7**	**99.5**	/////	/////	/////
W27	93.6	93.5	/////	/////	/////	/////	/////
W28	**99.8**	/////	/////	/////	/////	/////	/////
W29	/////	/////	/////	/////	/////	/////	/////
W30	/////	/////	/////	/////	/////	/////	/////
W31	/////	/////	/////	/////	/////	/////	/////
W32	/////	/////	/////	/////	/////	/////	/////

Table 8.10: Codes' efficiencies - PIC (add.s32 , n , write-read) - 1 SM - OK - Part 4

The code efficiency percentages for the ELF kernels generated for the PTX instruction configuration (add.s32 , n , write-read) when each ELF kernel is launched using only 1 thread block and only one streaming multiprocessor. For these tables, we supposed that a user could produce the desired ELF kernels thanks to the reverse engineering, but that he/she would not have any knowledge and quantification of the undisclosed low level architectural features (e.g. warp latency and write-read dependence latency) of the PTX instruction configuration (add.s32 , n , write-read). The ///// cases are impossible. This is because the number of hardware registers needed by the thread block would be greater than the number of hardware registers of each streaming multiprocessor.

	DD22	DD23	DD24	DD25	DD26	DD27	DD28
W01	**15.7**	**15.8**	**15.7**	**15.8**	**15.7**	**15.9**	**15.7**
W02	**31.6**	**31.7**	**31.7**	**31.6**	**31.7**	**31.7**	**31.8**
W03	**47.7**	**47.6**	**47.6**	**47.7**	**47.6**	**47.7**	**47.6**
W04	**63.7**	**63.6**	**63.7**	**63.6**	**63.7**	**63.7**	**63.7**
W05	**78.6**	**78.5**	**78.5**	**78.6**	**78.5**	**78.6**	**78.6**
W06	**97.5**	**97.4**	**97.5**	**97.5**	**97.3**	**97.4**	**97.3**
W07	**84.2**	**84.1**	**84.1**	**84.2**	**84.1**	**84.2**	**84.2**
W08	**99.3**	**99.2**	**99.2**	**99.1**	**99.2**	**99.1**	**99.2**
W09	**87.5**	**87.6**	**87.4**	**87.5**	**87.6**	**87.4**	**87.6**
W10	**98.6**	**98.6**	**97.8**	**98.5**	**98.5**	**98.5**	**98.6**
W11	**88.9**	**88.8**	**88.8**	**88.9**	**88.8**	**88.9**	**88.8**
W12	**99.4**	**99.4**	**98.9**	**99.2**	**99.4**	**99.4**	**99.2**
W13	**90.2**	**90.1**	**90.1**	**90.2**	**90.2**	**90.1**	**90.2**
W14	**98.7**	**99.5**	**98.7**	**98.8**	**99.1**	**99.3**	**99.3**
W15	**90.9**	**90.9**	**90.9**	**90.6**	**90.9**	**90.6**	**90.6**
W16	**98.8**	**99.3**	**99.3**	**99.1**	**99.0**	**99.2**	**99.0**
W17	**91.6**	**91.7**	**91.7**	**91.8**	**91.6**	**91.7**	**91.6**
W18	**99.1**	**99.2**	**99.0**	**99.2**	**98.9**	**99.3**	**90.0**
W19	**92.3**	**92.2**	**92.2**	**92.3**	**92.2**	**92.2**	**92.3**
W20	**98.9**	**98.6**	**98.8**	**98.6**	**99.0**	**98.8**	**99.2**
W21	**92.9**	**92.9**	**92.8**	**92.8**	**92.9**	**92.7**	/////
W22	**99.2**	**99.1**	**99.3**	**99.2**	**99.3**	**99.4**	/////
W23	**93.2**	**93.3**	/////	/////	/////	/////	/////
W24	/////	/////	/////	/////	/////	/////	/////
W25	/////	/////	/////	/////	/////	/////	/////
W26	/////	/////	/////	/////	/////	/////	/////
W27	/////	/////	/////	/////	/////	/////	/////
W28	/////	/////	/////	/////	/////	/////	/////
W29	/////	/////	/////	/////	/////	/////	/////
W30	/////	/////	/////	/////	/////	/////	/////
W31	/////	/////	/////	/////	/////	/////	/////
W32	/////	/////	/////	/////	/////	/////	/////

Table 8.11: Codes' efficiencies - PIC (add.s32 , n , write-read) - 1 SM - OK - Part 5

The code efficiency percentages for the ELF kernels generated for the PTX instruction configuration (add.s32 , n , write-read) when each ELF kernel is launched using only 1 thread block and only one streaming multiprocessor. For these tables, we supposed that a user could produce the desired ELF kernels thanks to the reverse engineering, but that he/she would not have any knowledge and quantification of the undisclosed low level architectural features (e.g. warp latency and write-read dependence latency) of the PTX instruction configuration (add.s32 , n , write-read). The ///// cases are impossible. This is because the number of hardware registers needed by the thread block would be greater than the number of hardware registers of each streaming multiprocessor.

	DD29	DD30	DD31	DD32	DD33	DD34	DD35
W01	**15.6**	**15.8**	**15.7**	**15.6**	**15.7**	**15.9**	**15.6**
W02	**31.6**	**31.7**	**31.7**	**31.8**	**31.7**	**31.8**	**31.8**
W03	**47.7**	**47.5**	**47.6**	**47.7**	**47.5**	**47.5**	**47.6**
W04	**63.7**	**63.6**	**63.7**	**63.6**	**63.5**	**63.7**	**63.5**
W05	**78.6**	**78.5**	**78.5**	**78.7**	**78.5**	**78.7**	**78.7**
W06	**97.3**	**97.4**	**97.5**	**97.4**	**97.3**	**97.4**	**97.5**
W07	**84.2**	**84.1**	**84.2**	**84.2**	**84.2**	**84.2**	**84.2**
W08	**99.3**	**99.4**	**99.2**	**99.4**	**99.2**	**99.1**	**99.4**
W09	**87.5**	**87.6**	**87.7**	**87.5**	**87.6**	**87.7**	**87.6**
W10	**98.6**	**98.5**	**97.8**	**98.5**	**98.7**	**98.5**	**98.5**
W11	**88.9**	**88.9**	**88.8**	**89.0**	**88.8**	**88.9**	**88.8**
W12	**99.3**	**99.4**	**98.9**	**99.2**	**99.3**	**99.4**	**99.3**
W13	**90.1**	**90.1**	**90.1**	**90.2**	**90.1**	**90.1**	**90.2**
W14	**98.7**	**99.5**	**98.6**	**98.8**	**99.6**	**99.3**	**99.3**
W15	**90.9**	**90.9**	**90.9**	**90.4**	**90.9**	**90.6**	**90.3**
W16	**98.4**	**99.3**	**99.3**	**99.1**	**99.4**	**99.2**	**99.3**
W17	**91.6**	**91.7**	**91.6**	**91.8**	**91.6**	**91.6**	**91.6**
W18	**99.1**	**99.2**	**99.2**	**99.2**	**98.9**	**99.5**	**90.5**
W19	**92.3**	**92.1**	**92.2**	**92.2**	/////	/////	/////
W20	**98.9**	**98.7**	/////	/////	/////	/////	/////
W21	/////	/////	/////	/////	/////	/////	/////
W22	/////	/////	/////	/////	/////	/////	/////
W23	/////	/////	/////	/////	/////	/////	/////
W24	/////	/////	/////	/////	/////	/////	/////
W25	/////	/////	/////	/////	/////	/////	/////
W26	/////	/////	/////	/////	/////	/////	/////
W27	/////	/////	/////	/////	/////	/////	/////
W28	/////	/////	/////	/////	/////	/////	/////
W29	/////	/////	/////	/////	/////	/////	/////
W30	/////	/////	/////	/////	/////	/////	/////
W31	/////	/////	/////	/////	/////	/////	/////
W32	/////	/////	/////	/////	/////	/////	/////

Table 8.12: Codes' efficiencies - PIC (add.s32 , n , write-read) - 1 SM - OK - Part 6

The code efficiency percentages for the ELF kernels generated for the PTX instruction configuration (add.s32 , n , write-read) when each ELF kernel is launched using only 1 thread block and only one streaming multiprocessor. For these tables, we supposed that a user could produce the desired ELF kernels thanks to the reverse engineering, but that he/she would not have any knowledge and quantification of the undisclosed low level architectural features (e.g. warp latency and write-read dependence latency) of the PTX instruction configuration (add.s32 , n , write-read). The ///// cases are impossible. This is because the number of hardware registers needed by the thread block would be greater than the number of hardware registers of each streaming multiprocessor.

	DD36	DD37	DD38	DD39	DD40	DD41	DD42
W01	**15.5**	**15.8**	**15.7**	**15.5**	**15.7**	**15.7**	**15.6**
W02	**31.6**	**31.8**	**31.7**	**31.7**	**31.6**	**31.7**	**31.5**
W03	**47.7**	**47.5**	**47.7**	**47.7**	**47.5**	**47.7**	**47.7**
W04	**63.7**	**63.6**	**63.7**	**63.6**	**63.5**	**63.4**	**63.3**
W05	**78.6**	**78.5**	**78.5**	**78.6**	**78.5**	**78.6**	**78.7**
W06	**97.6**	**97.4**	**97.5**	**97.5**	**97.5**	**97.4**	**97.5**
W07	**84.2**	**84.1**	**84.3**	**84.2**	**84.2**	**84.3**	**84.2**
W08	**99.4**	**99.4**	**99.2**	**99.4**	**99.3**	**99.2**	**99.4**
W09	**87.5**	**87.6**	**87.6**	**87.5**	**87.6**	**87.6**	**87.5**
W10	**98.7**	**98.5**	**97.8**	**98.5**	**98.6**	**98.5**	**98.6**
W11	**88.8**	**88.9**	**88.8**	**89.1**	**88.8**	**89.1**	**88.9**
W12	**99.5**	**99.4**	**98.9**	**99.3**	**99.3**	**99.4**	**99.4**
W13	**90.2**	**90.1**	**90.2**	**90.2**	**90.1**	**90.1**	**90.2**
W14	**98.7**	**99.5**	**98.7**	**98.8**	**99.6**	**99.4**	**99.3**
W15	**90.8**	**90.9**	**90.8**	**90.4**	**90.9**	**90.6**	**90.3**
W16	**98.5**	**99.3**	**99.3**	**99.2**	**99.4**	**99.3**	**99.3**
W17	**91.5**	**91.7**	**91.6**	**91.8**	/////	/////	/////
W18	/////	/////	/////	/////	/////	/////	/////
W19	/////	/////	/////	/////	/////	/////	/////
W20	/////	/////	/////	/////	/////	/////	/////
W21	/////	/////	/////	/////	/////	/////	/////
W22	/////	/////	/////	/////	/////	/////	/////
W23	/////	/////	/////	/////	/////	/////	/////
W24	/////	/////	/////	/////	/////	/////	/////
W25	/////	/////	/////	/////	/////	/////	/////
W26	/////	/////	/////	/////	/////	/////	/////
W27	/////	/////	/////	/////	/////	/////	/////
W28	/////	/////	/////	/////	/////	/////	/////
W29	/////	/////	/////	/////	/////	/////	/////
W30	/////	/////	/////	/////	/////	/////	/////
W31	/////	/////	/////	/////	/////	/////	/////
W32	/////	/////	/////	/////	/////	/////	/////

Table 8.13: Performance loss distribution - PIC (add.s32 , n ,write-read) - 1 SM - OK

For each of the 4 sets (see page 111), the table shows the number of cases per performance loss range and the percentage of total cases (1028). For each of the sets, $PL_{X-Y\%}$ represents a performance loss in the percentage range [X,Y) (X included and Y excluded).

	$PL_{98-100\%}$	$PL_{96-98\%}$	$PL_{94-96\%}$	$PL_{92-94\%}$	$PL_{90-92\%}$	$PL_{88-90\%}$	$PL_{86-88\%}$	$PL_{84-86\%}$	$PL_{82-84\%}$	$PL_{80-82\%}$
S_1	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
S_2	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
S_3	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
S_4	0 (0%)	0 (0%)	1 (0.10%)	0 (0%)	2 (0.19%)	0 (0%)	0 (0%)	41 (3.98%)	0 (0%)	2 (0.19%)

	$PL_{78-80\%}$	$PL_{76-78\%}$	$PL_{74-76\%}$	$PL_{72-74\%}$	$PL_{70-72\%}$	$PL_{68-70\%}$	$PL_{66-68\%}$	$PL_{64-66\%}$	$PL_{62-64\%}$	$PL_{60-62\%}$
S_1	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
S_2	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
S_3	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (0.10%)	0 (0%)	0 (0%)	1 (0.10%)	0 (0%)	1 (0.10%)
S_4	0 (0%)	0 (0%)	1 (0.10%)	0 (0%)	2 (0.19%)	39 (3.79%)	0 (0%)	0 (0%)	0 (0%)	1 (0.10%)

	$PL_{58-60\%}$	$PL_{56-58\%}$	$PL_{54-56\%}$	$PL_{52-54\%}$	$PL_{50-52\%}$	$PL_{48-50\%}$	$PL_{46-48\%}$	$PL_{44-46\%}$	$PL_{42-44\%}$	$PL_{40-42\%}$
S_1	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
S_2	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
S_3	0 (0%)	0 (0%)	1 (0.10%)	0 (0%)	1 (0.10%)	0 (0%)	0 (0%)	1 (0.10%)	1 (0.10%)	1 (0.10%)
S_4	0 (0%)	1 (0.10%)	0 (0%)	39 (3.79%)	1 (0.10%)	0 (0%)	0 (0%)	0 (0%)	1 (0.10%)	0 (0%)

	$PL_{38-40\%}$	$PL_{36-38\%}$	$PL_{34-36\%}$	$PL_{32-34\%}$	$PL_{30-32\%}$	$PL_{28-30\%}$	$PL_{26-28\%}$	$PL_{24-26\%}$	$PL_{22-24\%}$	$PL_{20-22\%}$
S_1	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
S_2	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
S_3	0 (0%)	0 (0%)	1 (0.10%)	1 (0.10%)	1 (0.10%)	0 (0%)	1 (0.10%)	0 (0%)	2 (0.19%)	1 (0.10%)
S_4	0 (0%)	39 (3.79%)	0 (0%)	1 (0.10%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	39 (3.79%)

	$PL_{18-20\%}$	$PL_{16-18\%}$	$PL_{14-16\%}$	$PL_{12-14\%}$	$PL_{10-12\%}$	$PL_{08-10\%}$	$PL_{06-08\%}$	$PL_{04-06\%}$	$PL_{02-04\%}$	$PL_{00-02\%}$
S_1	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	2 (0.19%)	0 (0%)	0 (0%)	51 (4.95%)	360 (34.95%)
S_2	0 (0%)	0 (0%)	39 (3.79%)	41 (3.98%)	41 (3.98%)	121 (11.75%)	126 (12.23%)	11 (1.07%)	0 (0%)	0 (0%)
S_3	1 (0.10%)	1 (0.10%)	3 (0.29%)	2 (0.19%)	1 (0.10%)	3 (0.29%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
S_4	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)

Table 8.14: Throughputs - PIC (or.b32 , n , write-read) - 14 SMs - OK - Part 1

Thanks to the discovery and quantification of the low level architectural features and machine behaviors, we know that a streaming multiprocessor is able to execute a maximum (MT) of 32 PTX or.b32.n instructions per functional unit clock cycle. This implies that the PTX or.b32.n instructions are executed by the 2 groups of 16 CUDA cores. The PTX or.b32.n instruction's latency (IL or write-read latency WRL) cannot be greater than 12 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. The rows of the table represent the number of warps of each thread block, while the columns of the table represent the dependence distances (there is one ELF kernel per dependence distance). However, note that we execute the ELF kernels generated for the PTX instruction configuration (or.b32 , n , write-read) using a number of thread blocks equal to the number of streaming multiprocessors (14), and that a Tesla C2070 can execute 448 $(14 \cdot 32)$ PTX or.b32.n instructions per functional unit clock cycle.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	DD09	DD10	DD11	DD12	DD13	DD14	DD15
W01	22.32	44.42	47.46	47.41	47.38	47.41	47.41	47.38	47.42	47.41	47.40	47.39	47.37	47.36	47.34
W02	44.65	88.84	94.48	94.66	94.60	94.59	94.63	94.57	94.48	94.55	94.57	94.54	94.55	94.53	94.37
W03	66.97	133.24	140.79	141.02	141.01	140.99	141.02	140.96	140.79	140.92	140.96	140.88	140.92	140.89	140.79
W04	89.29	176.25	185.31	185.58	185.56	185.55	185.58	185.52	185.31	185.46	185.52	185.42	185.47	185.45	185.43
W05	103.45	207.91	225.25	226.22	223.20	226.41	223.26	223.16	222.61	221.04	226.19	247.36	247.23	247.33	247.27
W06	122.14	222.60	222.79	222.95	222.95	222.91	222.95	222.91	222.90	222.90	222.92	222.86	222.85	222.83	222.79
W07	142.57	223.04	222.90	222.90	222.92	222.89	222.89	222.90	222.90	222.90	222.90	222.91	222.89	222.87	222.86
W08	172.06	223.12	223.03	223.04	223.04	223.02	223.04	223.03	223.04	223.04	223.04	223.03	223.04	223.01	223.04
W09	188.39	223.21	223.19	223.16	223.18	223.16	223.20	223.17	223.18	223.17	223.20	223.16	223.15	223.13	223.07
W10	207.79	223.26	223.19	223.15	223.17	223.18	223.14	223.16	223.19	223.16	223.14	223.17	223.16	223.13	223.09
W11	205.02	223.34	223.32	223.33	223.37	223.34	223.32	223.33	223.37	223.32	223.32	223.33	223.34	223.32	223.30
W12	223.43	223.29	223.29	223.29	223.29	223.28	223.27	223.28	223.29	223.28	223.28	223.29	223.28	223.27	223.29
W13	214.85	223.47	223.48	223.47	223.47	223.47	223.48	223.47	223.48	223.46	223.47	236.15	236.16	236.14	236.07
W14	226.92	223.39	223.39	223.38	223.07	223.38	223.38	223.39	223.38	223.38	223.38	416.63	416.58	416.57	416.34
W15	233.98	223.50	223.50	223.50	223.49	223.50	223.50	223.49	223.50	418.27	418.27	418.25	418.24	418.23	418.19
W16	239.21	223.47	223.47	223.49	223.48	223.49	223.48	223.48	418.34	416.76	417.89	418.87	417.98	417.97	417.76

Table 8.15: Throughputs - PIC (or.b32 , n , write-read) - 14 SMs - OK - Part 2

Thanks to the discovery and quantification of the low level architectural features and machine behaviors, we know that a streaming multiprocessor is able to execute a maximum (MT) of 32 PTX or.b32.n instructions per functional unit clock cycle. This implies that the PTX or.b32.n instructions are executed by the 2 groups of 16 CUDA cores. The PTX or.b32.n instruction's latency (IL or write-read latency WRL) cannot be greater than 12 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. The rows of the table represent the number of warps of each thread block, while the columns of the table represent the dependence distances (there is one ELF kernel per dependence distance). However, note that we execute the ELF kernels generated for the PTX instruction configuration (or.b32 , n , write-read) using a number of thread blocks equal to the number of streaming multiprocessors (14), and that a Tesla C2070 can execute 448 ($14 \cdot 32$) PTX or.b32.n instructions per functional unit clock cycle.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	DD09	DD10	DD11	DD12	DD13	DD14	DD15
W17	243.75	223.54	223.55	237.50	237.49	421.66	421.70	421.63	421.70	421.66	421.68	421.66	412.65	412.63	412.62
W18	227.29	223.53	223.55	230.17	230.18	422.78	423.65	424.14	423.78	423.89	424.07	424.18	424.32	424.31	424.27
W19	250.77	424.34	424.36	424.31	424.36	424.32	424.32	424.32	424.37	424.35	424.34	424.37	424.27	424.28	424.23
W20	419.76	417.87	416.89	415.36	421.89	420.64	419.87	420.78	421.66	420.53	419.22	418.89	419.27	418.46	417.89
W21	426.34	426.78	426.64	426.67	426.34	425.89	426.78	426.44	427.01	426.76	426.38	426.92	425.89	426.03	426.01
W22	426.72	426.34	425.89	426.72	426.26	426.73	426.59	426.88	426.41	426.76	426.31	426.72	426.69	426.87	426.79
W23	428.32	428.79	427.89	427.67	428.63	428.67	428.45	427.98	426.98	427.45	427.87	427.67	426.98	427.03	426.87
W24	430.87	429.71	429.34	430.35	430.78	430.72	429.23	430.07	430.72	429.78	429.68	430.07	429.98	430.05	430.02
W25	430.62	430.91	429.78	430.33	430.27	429.97	430.89	430.31	429.78	430.05	430.71	//////	//////	//////	//////
W26	428.78	429.34	429.64	428.79	429.06	428.34	428.94	429.13	428.17	428.52	427.89	//////	//////	//////	//////
W27	423.75	431.76	431.14	431.13	431.27	431.74	431.13	431.27	431.35	431.17	431.27	//////	//////	//////	//////
W28	439.76	446.77	443.76	442.96	444.63	440.62	443.92	442.76	444.07	443.79	443.27	//////	//////	//////	//////
W29	414.75	432.20	433.79	432.17	432.99	432.75	432.37	432.84	432.79	//////	//////	//////	//////	//////	//////
W30	424.24	442.87	441.27	444.36	443.92	441.72	443.67	441.57	440.78	//////	//////	//////	//////	//////	//////
W31	433.78	432.67	444.32	443.97	442.46	443.54	444.23	//////	//////	//////	//////	//////	//////	//////	//////
W32	446.32	444.76	445.32	442.64	443.23	444.72	443.98	//////	//////	//////	//////	//////	//////	//////	//////

Table 8.16: Codes' efficiencies - PIC (or.b32 , n , write-read) - 14 SMs - OK - Part 1

The code efficiencies of the ELF kernels, generated for the PTX instruction configuration (or.b32 , n , write-read), are shown below. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. The rows of the table represent the number of warps of each thread block, while the columns of the table represent the dependence distances (there is one ELF kernel per dependence distance). Note that we execute the ELF kernels using 14 thread blocks (equal to the number of streaming multiprocessors), and that a Tesla C2070 can execute 448 ($14 \cdot 32$) PTX or.b32.n instructions per functional unit clock cycle. The low code efficiencies are due to the gigathread scheduler, which does not fairly assign the thread blocks to the streaming multiprocessors.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	DD09	DD10	DD11	DD12	DD13	DD14	DD15
W01	5.0	9.9	10.6	10.6	10.6	10.6	10.6	10.6	10.6	10.6	10.6	10.6	10.6	10.6	10.6
W02	10.0	19.8	21.0	21.1	21.1	21.1	21.1	21.1	21.0	21.1	21.1	21.1	21.1	21.1	21.1
W03	15.0	29.7	31.4	31.5	31.5	31.5	31.5	31.4	31.4	31.5	31.5	31.4	31.5	31.5	31.4
W04	19.9	39.3	41.4	41.4	41.4	41.4	41.4	41.4	41.4	41.4	41.4	41.4	41.4	41.4	41.4
W05	23.1	46.4	50.3	50.5	49.8	50.5	49.8	49.8	49.7	49.3	50.5	55.2	55.2	55.2	55.2
W06	25.03	49.7	49.7	49.8	49.7	49.7	49.8	49.8	49.8	49.8	49.8	49.7	49.7	49.7	49.7
W07	31.8	49.8	49.7	49.7	49.8	49.8	49.8	49.8	49.8	49.7	49.7	49.8	49.7	49.7	49.8
W08	38.4	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	48.8	49.8
W09	42.1	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8
W10	46.4	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8
W11	45.8	49.8	49.8	49.8	49.9	49.8	49.8	49.8	49.9	49.9	49.8	49.8	49.8	49.9	49.8
W12	49.9	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8
W13	48.0	49.9	49.9	49.9	49.9	49.9	49.9	49.9	49.9	49.9	49.9	52.7	52.7	52.7	52.7
W14	50.7	49.9	49.9	49.9	49.8	49.9	49.9	49.9	49.9	49.9	49.9	93.0	93.0	93.0	92.9
W15	52.2	52.1	52.1	52.1	52.1	52.1	52.1	52.1	52.1	93.4	93.4	93.4	93.4	93.4	93.4
W16	53.4	49.9	49.9	49.9	49.9	49.9	49.9	49.9	93.4	93.0	93.3	93.5	100.0	100.0	99.9

Table 8.17: Codes' efficiencies - PIC (or.b32 , n , write-read) - 14 SMs - OK - Part 2

The code efficiencies of the ELF kernels, generated for the PTX instruction configuration (or.b32 , n , write-read), are shown below. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. The rows of the table represent the number of warps of each thread block, while the columns of the table represent the dependence distances (there is one ELF kernel per dependence distance). Note that we execute the ELF kernels using 14 thread blocks (equal to the number of streaming multiprocessors), and that a Tesla C2070 can execute 448 ($14 \cdot 32$) PTX or.b32.n instructions per functional unit clock cycle. The low code efficiencies are due to the gigathread scheduler, which does not fairly assign the thread blocks to the streaming multiprocessors.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	DD09	DD10	DD11	DD12	DD13	DD14	DD15
W17	54.4	49.9	49.9	53.0	53.0	94.1	94.1	94.1	94.1	94.1	94.1	94.1	92.1	92.1	92.1
W18	50.7	49.9	49.9	51.4	51.4	94.4	94.6	94.7	94.6	94.6	94.7	94.7	94.7	94.7	94.7
W19	56.0	94.7	94.7	94.7	94.7	94.7	94.7	94.7	94.7	94.7	94.7	94.7	94.7	94.7	94.7
W20	93.7	93.3	93.1	92.7	94.2	93.9	93.7	93.9	94.1	93.9	93.6	93.5	93.6	93.4	93.3
W21	95.2	95.3	95.2	95.2	95.2	95.1	95.3	95.2	95.3	95.3	95.2	95.3	95.1	95.1	95.1
W22	95.3	95.2	95.1	95.3	95.2	95.3	95.2	95.3	95.2	95.3	95.1	95.2	95.2	95.3	95.3
W23	95.6	95.7	95.5	95.5	95.7	95.7	95.6	95.5	95.3	95.4	95.5	95.5	95.3	95.3	95.3
W24	96.2	95.9	95.8	96.1	96.2	96.1	95.8	96.0	96.1	95.9	95.9	96.0	96.0	96.0	96.0
W25	96.1	96.2	95.9	96.1	96.0	96.0	96.2	96.1	95.9	96.0	96.1	////	////	////	////
W26	95.7	95.8	95.9	95.7	95.8	95.6	95.8	95.8	95.6	95.7	95.5	////	////	////	////
W27	94.6	96.4	96.2	96.2	96.3	96.8	96.2	96.3	96.3	96.2	96.3	////	////	////	////
W28	98.2	99.7	99.1	98.9	99.3	98.4	99.1	98.8	99.1	99.1	98.9	////	////	////	////
W29	92.6	96.5	96.8	96.5	96.7	96.6	96.5	96.6	96.6	////	////	////	////	////	////
W30	94.8	98.9	98.5	99.2	99.1	98.6	99.0	98.6	98.4	////	////	////	////	////	////
W31	96.8	96.6	99.6	99.1	98.7	99.0	99.2	////	////	////	////	////	////	////	////
W32	99.6	99.3	99.4	98.8	98.9	99.3	99.1	////	////	////	////	////	////	////	////

The Tables 8.16 and 8.17 illustrate the very low efficiencies that a user can easily achieve even if he/she produces the desired ELF kernels exploiting the reverse engineering phase but does not have any knowledge and quantification of the low level architectural features and machine behaviors (no knowledge and the use of all the streaming multiprocessors imply that we are considering the $Cases_{t3}$). In these conditions, a user will have to develop many different codes and run each code with many different launch configurations. This is a very time consuming and error-prone process.

However, thanks to the insights we earned with the discovery and quantification of the low level architectural features we can modify kernels to force the gigathread scheduler to fairly distribute the thread blocks to the streaming multiprocessors ($Cases_{t4}$); see procedure 8.18. This is true even if we suppose that we do not have any knowledge and quantification of the low level architectural features.

Table 8.18: Procedure that generates fair launch configurations

Procedure used to generate launch configurations that force the gigathread scheduler to fairly distribute the thread blocks to the streaming multiprocessors. The below examples illustrate the procedure for launch configurations with a number of thread blocks that is equal to the number of streaming multiprocessor (14). As an example, let us take the PTX instruction configuration and.b32.n.wr. We need 21 hardware registers to execute the instructions that are different from the PTX and.b32 instructions that are inside the for loop of each ELF kernel. Furthermore, for each dependence distance DDX, we need X additional hardware registers for the execution of the PTX instruction configurations and.b32.n.wr that are inside each for loop. However, to force the gigathread scheduler to fairly distribute the thread blocks to the streaming multiprocessors, each thread block requires a number of hardware registers greater than half the number of hardware registers of each streaming multiprocessor, and equal to or smaller than the total number of hardware registers of each streaming multiprocessor. Therefore, each thread block must have at least 9 warps. This is because each thread cannot have more than 64 hardware registers. Therefore, 8 wa-

```
rps, each of which has 32 threads, are not enough per thread block. This
is because a thread block with only 8 warps and 64 hardware registers per
thread requires a number of hardware registers equal to and not greater
than half the number of hardware registers of a streaming multiprocessor
( 8 x 32 x 64 = 16384 = 32756 / 2 ).

The ELF kernel that is generated for the dependence distance X requires a
total of Y = ( 21 + X hardware registers). The number of warps per thread
block will next determine the possible numbers of additional hardware re-
gisters that we need to assign to each thread, in order to guarantee that
the gigathread scheduler will fairly distribute the thread blocks to the
streaming multiprocessors.

For launch configurations with 14 thread blocks, Tables X.X-Y.Y indicate
the possible ranges of the total numbers of hardware registers per thread
that are necessary to guarantee that the gigathread  scheduler will fair-
ly distribute the thread blocks to the streaming multiprocessors.
```

Table 8.19: PIC (and.b32 , n , write-read) - Hardware registers per thread

The ELF kernel generated for the PTX instruction configuration and.b32.n.wr for the dependence distance X requires Y = 21 + X hardware registers (see Table 8.18). However, each thread can have a maximum of 64 hardware registers. Therefore, 43 ELF kernels are generated, one for each dependence distance ranging from 1 to 43. However, the ELF kernel generated for the dependence distance 43 is discarded. This is because, during the compiling phase, we verified that spill load and spill store phenomena occur during the executions of the ELF kernel. Table 8.19 shows the number of hardware registers (HR) assigned to each thread for each of the remaining original 42 ELF kernels.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07
HR	21 + 01 = 22	21 + 02 = 23	21 + 03 = 24	21 + 04 = 25	21 + 05 = 26	21 + 06 = 27	28 + 07 = 28
	DD08	DD09	DD10	DD11	DD12	DD13	DD14
HR	21 + 08 = 29	21 + 09 = 30	21 + 10 = 31	21 + 11 = 32	21 + 12 = 33	21 + 13 = 34	21 + 14 = 35
	DD15	DD16	DD17	DD18	DD19	DD20	DD21
HR	21 + 15 = 36	21 + 16 = 37	21 + 17 = 38	21 + 18 = 39	21 + 19 = 40	21 + 20 = 41	21 + 21 = 42
	DD22	DD23	DD24	DD25	DD26	DD27	DD28
HR	21 + 22 = 43	21 + 23 = 44	21 + 24 = 45	21 + 25 = 46	21 + 26 = 47	21 + 27 = 48	21 + 28 = 49
	DD29	DD30	DD31	DD32	DD33	DD34	DD35
HR	21 + 29 = 50	21 + 30 = 51	21 + 31 = 52	21 + 32 = 53	21 + 33 = 54	21 + 34 = 55	21 + 35 = 56
	DD36	DD37	DD38	DD39	DD40	DD41	DD42
HR	21 + 36 = 57	21 + 37 = 58	21 + 38 = 59	21 + 39 = 60	21 + 40 = 61	21 + 41 = 62	21 + 42 = 63

Table 8.20: Additional HRs - PIC (and.b32 , n , write-read) - 14 SMs - MK - Part 1

The dependence distances (DD) shown range from 1 to 14. The ELF kernel generated for dependence distance X requires Y = 21 + X hardware registers (see Tables 8.18 and 8.19). However, if we use launch configurations with 14 thread blocks, then each thread block must have at least 9 warps (see Table 8.18) to force the gigathread scheduler to fairly distribute the thread blocks to the streaming multiprocessors. When the number of warps per thread block is equal to or greater than 9, then the ranges of additional hardware registers to be assigned to each thread are shown in each cell. However, note that in each case, each thread will use only a fraction of its hardware registers to execute the ELF kernel generated for dependence distance X. For example, if X is equal to 1, and we want to execute the ELF kernel using 14 thread blocks, each with 9 warps, then we need to produce a new ELF kernel with between 35 and 42 additional hardware registers. The // - // cases are impossible because the number of hardware registers necessary per thread block would be greater than the number of hardware registers of each streaming multiprocessor.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	DD09	DD10	DD11	DD12	DD13	DD14
W09	35 - 42	34 - 41	33 - 40	32 - 39	31 - 38	30 - 37	29 - 36	28 - 35	27 - 34	26 - 33	25 - 32	24 - 31	23 - 30	22 - 29
W10	30 - 42	29 - 41	28 - 40	27 - 39	26 - 38	25 - 37	24 - 36	23 - 35	22 - 34	21 - 33	20 - 32	19 - 31	18 - 30	17 - 29
W11	25 - 42	24 - 41	23 - 40	22 - 39	21 - 38	20 - 37	19 - 36	18 - 35	17 - 34	16 - 33	15 - 32	14 - 31	13 - 30	12 - 29
W12	21 - 42	20 - 41	19 - 40	18 - 39	17 - 38	16 - 37	15 - 36	14 - 35	13 - 34	12 - 33	11 - 32	10 - 31	09 - 30	08 - 29
W13	18 - 42	17 - 41	16 - 40	15 - 39	14 - 38	13 - 37	12 - 36	11 - 35	10 - 34	09 - 33	08 - 32	07 - 31	06 - 30	05 - 29
W14	15 - 42	14 - 41	13 - 40	12 - 39	11 - 38	10 - 37	09 - 36	08 - 35	07 - 34	06 - 33	05 - 32	04 - 31	03 - 30	02 - 29
W15	13 - 42	12 - 41	11 - 40	10 - 39	09 - 38	08 - 37	07 - 36	06 - 35	05 - 34	04 - 33	03 - 32	02 - 31	01 - 30	00 - 29
W16	10 - 42	09 - 41	08 - 40	07 - 39	06 - 38	05 - 37	04 - 36	03 - 35	02 - 34	01 - 33	00 - 32	00 - 31	00 - 30	00 - 29
W17	09 - 38	08 - 37	07 - 36	06 - 35	05 - 34	04 - 33	03 - 32	02 - 31	01 - 30	00 - 29	00 - 28	00 - 27	00 - 26	00 - 25
W18	07 - 34	06 - 33	05 - 32	04 - 31	03 - 30	02 - 29	01 - 28	00 - 27	00 - 26	00 - 25	00 - 24	00 - 23	00 - 22	00 - 21
W19	05 - 31	04 - 30	03 - 29	02 - 28	01 - 27	00 - 26	00 - 25	00 - 24	00 - 23	00 - 22	00 - 21	00 - 20	00 - 19	00 - 18
W20	04 - 29	03 - 28	02 - 27	01 - 26	00 - 25	00 - 24	00 - 23	00 - 22	00 - 21	00 - 20	00 - 19	00 - 18	00 - 17	00 - 16
W21	03 - 26	02 - 25	01 - 24	00 - 23	00 - 22	00 - 21	00 - 20	00 - 19	00 - 18	00 - 17	00 - 16	00 - 15	00 - 14	00 - 13
W22	02 - 24	01 - 23	00 - 22	00 - 21	00 - 20	00 - 19	00 - 18	00 - 17	00 - 16	00 - 15	00 - 14	00 - 13	00 - 12	00 - 11
W23	01 - 22	00 - 21	00 - 20	00 - 19	00 - 18	00 - 17	00 - 16	00 - 15	00 - 14	00 - 13	00 - 12	00 - 11	00 - 10	00 - 09
W24	00 - 20	00 - 19	00 - 18	00 - 17	00 - 16	00 - 15	00 - 14	00 - 13	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07
W25	00 - 18	00 - 17	00 - 16	00 - 15	00 - 14	00 - 13	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05
W26	00 - 17	00 - 16	00 - 15	00 - 14	00 - 13	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04
W27	00 - 15	00 - 14	00 - 13	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02
W28	00 - 14	00 - 13	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01
W29	00 - 13	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01	00 - 00
W30	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01	00 - 00	// - //
W31	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01	00 - 00	// - //	// - //
W32	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01	00 - 00	// - //	// - //	// - //

Table 8.21: Additional HRs - PIC (and.b32 , n , write-read) - 14 SMs - MK - Part 2

The dependence distances (DD) shown range from 1 to 14. The ELF kernel generated for dependence distance X requires Y = 21 + X hardware registers (see Tables 8.18 and 8.19). However, if we use launch configurations with 14 thread blocks, then each thread block must have at least 9 warps (see Table 8.18) to force the gigathread scheduler to fairly distribute the thread blocks to the streaming multiprocessors. When the number of warps per thread block is equal to or greater than 9, then the ranges of additional hardware registers to be assigned to each thread are shown in each cell. However, note that in each case, each thread will use only a fraction of its hardware registers to execute the ELF kernel generated for dependence distance X. For example, if X is equal to 1, and we want to execute the ELF kernel using 14 thread blocks, each with 9 warps, then we need to produce a new ELF kernel with between 35 and 42 additional hardware registers. The // - // cases are impossible because the number of hardware registers necessary per thread block would be greater than the number of hardware registers of each streaming multiprocessor.

	DD15	DD16	DD17	DD18	DD19	DD20	DD21	DD22	DD23	DD24	DD25	DD26	DD27	DD28
W09	21 - 28	20 - 27	19 - 26	18 - 25	17 - 24	16 - 23	15 - 22	14 - 21	13 - 20	12 - 19	11 - 18	10 - 17	09 - 16	08 - 15
W10	16 - 28	15 - 27	14 - 26	13 - 25	12 - 24	11 - 23	10 - 22	09 - 21	08 - 20	07 - 19	06 - 18	05 - 17	04 - 16	03 - 15
W11	11 - 28	10 - 27	09 - 26	08 - 25	07 - 24	06 - 23	05 - 22	04 - 21	03 - 20	02 - 19	01 - 18	00 - 17	00 - 16	00 - 15
W12	07 - 28	06 - 27	05 - 26	04 - 25	03 - 24	02 - 23	01 - 22	00 - 21	00 - 20	00 - 19	00 - 18	00 - 17	00 - 16	00 - 15
W13	04 - 28	03 - 27	02 - 26	01 - 25	00 - 24	00 - 23	00 - 22	00 - 21	00 - 20	00 - 19	00 - 18	00 - 17	00 - 16	00 - 15
W14	01 - 28	00 - 27	00 - 26	00 - 25	00 - 24	00 - 23	00 - 22	00 - 21	00 - 20	00 - 19	00 - 18	00 - 17	00 - 16	00 - 15
W15	00 - 28	00 - 27	00 - 26	00 - 25	00 - 24	00 - 23	00 - 22	00 - 21	00 - 20	00 - 19	00 - 18	00 - 17	00 - 16	00 - 15
W16	00 - 28	00 - 27	00 - 26	00 - 25	00 - 24	00 - 23	00 - 22	00 - 21	00 - 20	00 - 19	00 - 18	00 - 17	00 - 16	00 - 15
W17	00 - 24	00 - 23	00 - 22	00 - 21	00 - 20	00 - 19	00 - 18	00 - 17	00 - 16	00 - 15	00 - 14	00 - 13	00 - 12	00 - 11
W18	00 - 20	00 - 19	00 - 18	00 - 17	00 - 16	00 - 15	00 - 14	00 - 13	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07
W19	00 - 17	00 - 16	00 - 15	00 - 14	00 - 13	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04
W20	00 - 15	00 - 14	00 - 13	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02
W21	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01	00 - 00	// - //
W22	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01	00 - 00	00 - 00	00 - 00	// - //
W23	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01	00 - 00	// - //	// - //	// - //	// - //	// - //
W24	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01	00 - 00	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W25	00 - 04	00 - 03	00 - 02	00 - 01	00 - 00	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W26	00 - 03	00 - 02	00 - 01	00 - 00	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W27	00 - 01	00 - 00	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W28	00 - 00	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W29	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W30	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W31	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W32	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //

Table 8.22: Additional HRs - PIC (and.b32 , n , write-read) - 14 SMs - MK - Part 3

The dependence distances (DD) shown range from 1 to 14. The ELF kernel generated for dependence distance X requires Y = 21 + X hardware registers (see Tables 8.18 and 8.19). However, if we use launch configurations with 14 thread blocks, then each thread block must have at least 9 warps (see Table 8.18) to force the gigathread scheduler to fairly distribute the thread blocks to the streaming multiprocessors. When the number of warps per thread block is equal to or greater than 9, then the ranges of additional hardware registers to be assigned to each thread are shown in each cell. However, note that in each case, each thread will use only a fraction of its hardware registers to execute the ELF kernel generated for dependence distance X. For example, if X is equal to 1, and we want to execute the ELF kernel using 14 thread blocks, each with 9 warps, then we need to produce a new ELF kernel with between 35 and 42 additional hardware registers. The // - // cases are impossible because the number of hardware registers necessary per thread block would be greater than the number of hardware registers of each streaming multiprocessor.

	DD29	DD30	DD31	DD32	DD33	DD34	DD35	DD36	DD37	DD38	DD39	DD40	DD41	DD42
W09	07 - 14	06 - 13	05 - 12	04 - 11	03 - 10	02 - 09	01 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01
W10	02 - 14	01 - 13	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01
W11	00 - 14	00 - 13	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01
W12	00 - 14	00 - 13	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01
W13	00 - 14	00 - 13	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01
W14	00 - 14	00 - 13	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01
W15	00 - 14	00 - 13	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01
W16	00 - 14	00 - 13	00 - 12	00 - 11	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01
W17	00 - 10	00 - 09	00 - 08	00 - 07	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01	00 - 00	// - //	// - //	// - //
W18	00 - 06	00 - 05	00 - 04	00 - 03	00 - 02	00 - 01	00 - 00	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W19	00 - 03	00 - 02	00 - 01	00 - 00	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W20	00 - 01	00 - 00	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W21	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W22	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W23	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W24	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W25	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W26	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W27	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W28	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W29	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W30	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W31	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //
W32	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //	// - //

Table 8.23: Codes' efficiencies - PIC (and.b32 , n , write-read) - 14 SMs - MK - Part 1

The code efficiencies of the modified ELF kernels, generated for the PTX instruction configuration (and.b32 , n , write-read), are shown below. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. The rows of the table represent the number of warps of each thread block, while the columns of the table represent the dependence distances (there is one ELF kernel per dependence distance). Note that we execute the ELF kernels using 14 thread blocks (equal to the number of streaming multiprocessors), and that a Tesla C2070 can execute 448 ($14 \cdot 32$) PTX and.b32.n instructions per functional unit clock cycle. For each original ELF kernel (there is one kernel per dependence distance), we generate 24 new ELF kernels (one per number of warps per thread block). This is accomplished using Tables 8.20, 8.21, and 8.22. These new ELF kernels guarantee that, when they are executed, the gigathread scheduler will always fairly assign the thread blocks to the streaming multiprocessors.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	DD09	DD10	DD11	DD12	DD13	DD14
W09	44.9	82.9	87.9	88.2	88.0	88.0	87.8	89.1	87.9	87.8	88.0	87.9	87.9	88.1
W10	49.7	92.1	97.7	97.8	97.9	97.7	97.7	97.6	97.8	97.7	97.8	97.6	97.8	97.7
W11	54.6	89.2	89.9	89.9	89.7	90.0	89.7	89.7	89.7	89.7	89.7	89.7	89.5	89.6
W12	59.6	97.3	97.8	97.8	97.5	97.8	98.0	97.7	97.7	97.8	97.8	97.8	97.6	97.7
W13	64.4	90.5	90.8	90.9	91.0	91.2	91.0	90.7	90.7	90.8	90.8	90.8	90.7	91.0
W14	69.3	97.5	97.8	97.8	97.5	97.7	97.6	97.6	97.8	97.6	97.8	97.6	97.7	97.7
W15	73.1	91.8	91.8	91.8	91.7	91.9	91.7	91.8	91.7	91.8	91.7	91.7	91.8	91.7
W16	78.0	97.9	97.7	97.9	97.9	97.8	97.8	97.8	97.8	97.8	97.9	97.8	97.9	97.8
W17	79.8	92.5	92.4	92.5	92.4	92.4	92.4	92.4	92.5	92.4	92.6	92.5	92.4	92.4
W18	84.6	97.9	97.9	97.9	97.8	97.8	97.9	97.8	97.9	97.8	97.9	97.8	97.9	97.9
W19	86.7	92.8	92.7	92.8	93.1	92.8	93.4	92.7	92.9	93.0	92.8	92.9	92.7	93.0
W20	91.3	97.9	97.8	97.9	97.9	97.8	97.9	97.6	97.8	97.8	97.7	97.9	97.8	97.9
W21	86.2	93.2	93.3	93.3	93.5	93.5	93.6	93.6	93.6	93.6	93.4	93.6	93.6	93.3
W22	93.5	97.7	97.9	97.8	99.4	97.8	97.7	97.9	97.9	97.9	97.8	97.9	97.8	97.7
W23	92.7	93.7	93.6	93.8	93.8	94.0	93.8	93.7	93.9	93.8	93.8	93.8	93.8	93.8
W24	97.0	97.9	97.9	97.9	97.9	98.0	97.9	97.9	97.8	97.8	97.2	97.2	97.8	97.8
W25	87.7	97.9	97.9	97.9	97.9	98.0	97.9	97.9	97.8	97.8	97.2	97.2	97.8	97.8
W26	91.3	94.0	94.1	94.7	95.9	94.3	94.2	94.0	94.1	94.4	94.0	94.1	94.0	94.0
W27	91.5	94.2	94.4	94.7	94.4	94.7	94.7	94.5	94.5	94.3	94.5	94.4	94.5	94.4
W28	95.2	97.9	97.8	97.6	97.7	97.8	97.8	97.7	97.7	97.8	97.7	97.8	97.7	97.8
W29	89.4	94.5	94.7	94.6	94.3	94.8	94.8	94.6	94.7	94.6	94.7	94.7	94.6	94.7
W30	92.6	97.9	97.8	97.9	98.0	97.7	98.0	97.8	97.8	97.8	97.7	97.8	97.8	////
W31	95.0	94.6	94.8	95.0	94.6	95.0	95.0	94.9	95.0	94.7	94.8	94.9	////	////
W32	98.2	97.0	97.8	97.8	98.1	97.8	97.8	97.8	98.0	97.9	97.8	////	////	////

Table 8.24: Codes' efficiencies - PIC (and.b32 , n , write-read) - 14 SMs - MK - Part 2

The code efficiencies of the modified ELF kernels, generated for the PTX instruction configuration (and.b32 , n , write-read), are shown below. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. The rows of the table represent the number of warps of each thread block, while the columns of the table represent the dependence distances (there is one ELF kernel per dependence distance). Note that we execute the ELF kernels using 14 thread blocks (equal to the number of streaming multiprocessors), and that a Tesla C2070 can execute 448 ($14 \cdot 32$) PTX and.b32.n instructions per functional unit clock cycle. For each original ELF kernel (there is one kernel per dependence distance), we generate 24 new ELF kernels (one per number of warps per thread block). This is accomplished using Tables 8.20, 8.21, and 8.22. These new ELF kernels guarantee that, when they are executed, the gigathread scheduler will always fairly assign the thread blocks to the streaming multiprocessors.

	DD15	DD16	DD17	DD18	DD19	DD20	DD21	DD22	DD23	DD24	DD25	DD26	DD27	DD28
W09	88.0	87.9	88.1	87.9	88.0	88.0	87.9	87.9	88.1	88.0	88.1	88.0	87.9	87.9
W10	97.6	97.7	97.7	97.8	97.6	97.6	97.7	97.7	97.8	97.7	97.6	97.8	97.7	97.7
W11	89.5	89.6	89.5	89.6	89.6	89.5	89.4	89.6	89.5	89.6	89.5	89.4	89.6	89.5
W12	97.6	97.7	97.6	97.5	97.6	97.7	97.6	97.5	97.7	97.6	97.6	97.7	97.5	97.7
W13	90.8	90.9	90.8	91.0	90.8	90.9	90.7	90.9	91.0	90.9	90.8	90.9	91.0	90.9
W14	97.6	97.7	97.7	97.8	97.7	97.7	97.6	97.7	97.8	97.7	97.6	97.7	97.7	97.7
W15	91.7	91.6	91.7	91.6	91.7	91.7	91.7	91.6	91.7	91.7	91.6	91.7	91.6	91.7
W16	97.8	97.8	97.6	97.8	97.8	97.6	97.7	97.7	97.8	97.8	97.6	97.6	97.7	97.8
W17	92.3	92.4	92.4	92.3	92.4	92.4	92.3	92.4	92.3	92.3	92.4	92.4	92.3	92.3
W18	97.9	97.8	97.6	97.8	97.9	97.8	97.9	97.8	97.8	97.9	97.8	97.9	97.8	97.9
W19	92.8	92.9	93.0	92.9	92.8	93.0	92.9	92.9	92.9	93.0	92.9	93.0	92.9	92.9
W20	97.9	97.8	97.9	97.8	97.8	97.9	97.8	97.8	97.9	97.9	97.8	97.8	97.8	97.8
W21	93.3	93.2	93.3	93.3	93.2	93.2	93.3	93.2	93.3	93.2	93.3	93.2	93.2	////
W22	97.7	97.8	97.7	97.7	97.8	97.7	97.8	97.7	97.8	97.8	97.8	97.7	97.8	////
W23	93.8	93.7	93.8	93.7	93.8	93.8	93.7	93.8	93.8	////	////	////	////	////
W24	97.8	97.7	97.8	97.9	97.8	97.7	97.7	////	////	////	////	////	////	////
W25	94.2	94.4	94.3	94.4	94.3	////	////	////	////	////	////	////	////	////
W26	97.8	97.7	97.8	97.8	////	////	////	////	////	////	////	////	////	////
W27	94.4	94.3	////	////	////	////	////	////	////	////	////	////	////	////
W28	97.7	////	////	////	////	////	////	////	////	////	////	////	////	////
W29	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W30	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W31	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W32	////	////	////	////	////	////	////	////	////	////	////	////	////	////

Table 8.25: Codes' efficiencies - PIC (and.b32 , n , write-read) - 14 SMs - MK - Part 3

The code efficiencies of the modified ELF kernels, generated for the PTX instruction configuration (and.b32 , n , write-read), are shown below. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. The rows of the table represent the number of warps of each thread block, while the columns of the table represent the dependence distances (there is one ELF kernel per dependence distance). Note that we execute the ELF kernels using 14 thread blocks (equal to the number of streaming multiprocessors), and that a Tesla C2070 can execute 448 ($14 \cdot 32$) PTX and.b32.n instructions per functional unit clock cycle. For each original ELF kernel (there is one kernel per dependence distance), we generate 24 new ELF kernels (one per number of warps per thread block). This is accomplished using Tables 8.20, 8.21, and 8.22. These new ELF kernels guarantee that, when they are executed, the gigathread scheduler will always fairly assign the thread blocks to the streaming multiprocessors.

	DD29	DD30	DD31	DD32	DD33	DD34	DD35	DD36	DD37	DD38	DD39	DD40	DD41	DD42
W09	88.0	88.1	87.9	88.0	88.1	88.0	87.8	87.9	88.0	88.1	88.1	88.0	87.8	87.9
W10	97.7	97.6	97.8	97.7	97.7	97.8	97.6	97.7	97.8	97.7	97.8	97.8	97.7	97.8
W11	89.6	89.5	89.7	89.7	89.6	89.6	89.7	89.6	89.7	89.6	89.7	89.6	89.7	89.6
W12	97.7	97.6	97.7	97.6	97.7	97.6	97.7	97.7	97.7	97.6	97.7	97.7	97.6	97.7
W13	90.9	90.7	90.8	90.9	90.7	90.8	90.8	90.7	91.0	90.9	90.8	90.9	90.8	90.8
W14	97.7	97.6	97.8	97.6	97.6	97.7	97.6	97.6	97.8	97.8	97.7	97.6	97.6	97.7
W15	91.7	91.6	91.7	91.7	91.6	91.6	91.7	91.7	91.6	91.6	91.7	91.6	91.7	91.7
W16	97.8	97.7	97.7	97.6	97.7	97.7	97.8	97.8	97.7	97.7	97.8	97.6	97.7	97.8
W17	92.4	92.3	92.4	92.4	92.3	92.4	92.4	92.3	92.3	92.4	92.3	////	////	////
W18	97.8	97.7	97.9	97.8	97.8	97.7	97.6	////	////	////	////	////	////	////
W19	93.0	92.9	93.1	93.0	////	////	////	////	////	////	////	////	////	////
W20	97.9	97.8	////	////	////	////	////	////	////	////	////	////	////	////
W21	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W22	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W23	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W24	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W25	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W26	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W27	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W28	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W29	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W30	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W31	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W32	////	////	////	////	////	////	////	////	////	////	////	////	////	////

Table 8.26: Codes' efficiencies - PIC (and.b32 , n , write-read) - 14 SMs - OK - Part 1

The code efficiencies of the ELF kernels, generated for the PTX instruction configuration (and.b32 , n , write-read), are shown below. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. The rows of the table represent the number of warps of each thread block, while the columns of the table represent the dependence distances (there is one ELF kernel per dependence distance). Note that we execute the ELF kernels using 14 thread blocks (equal to the number of streaming multiprocessors), and that a Tesla C2070 can execute 448 ($14 \cdot 32$) PTX and.b32.n instructions per functional unit clock cycle. The low code efficiencies are due to the gigathread scheduler that does not fairly assign the thread blocks to the streaming multiprocessors.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	DD09	DD10	DD11	DD12	DD13	DD14
W09	42.0	49.8	49.8	49.9	49.9	49.8	49.8	49.8	49.9	49.9	49.9	49.8	49.9	49.8
W10	46.6	49.8	49.7	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8	49.8
W11	45.8	49.9	49.7	49.8	49.8	49.9	49.9	49.8	49.9	49.8	49.9	49.9	49.8	49.8
W12	49.9	49.9	49.8	49.9	49.9	49.9	49.8	49.9	49.9	49.9	49.8	49.9	49.9	49.8
W13	48.0	49.9	49.8	49.9	49.8	49.9	49.8	49.8	49.9	49.9	49.8	49.9	49.9	95.4
W14	50.5	49.9	49.8	49.9	49.8	49.9	49.9	49.8	49.9	49.9	49.8	49.8	49.9	95.4
W15	52.0	49.9	49.8	49.9	49.8	49.9	49.9	49.8	49.9	95.4	95.3	95.4	95.3	95.4
W16	53.5	49.9	49.8	49.9	49.8	49.8	49.9	49.8	49.9	95.4	95.3	95.4	95.4	95.3
W17	54.1	49.9	49.9	49.8	49.8	95.4	95.3	95.4	95.3	95.4	95.3	95.4	95.4	95.3
W18	50.7	49.9	49.9	49.8	49.8	95.3	95.4	95.4	95.3	95.4	95.4	95.4	95.3	95.4
W19	55.6	50.1	50.2	50.1	50.2	95.4	95.3	95.4	95.3	95.4	95.4	95.3	95.4	95.4
W20	91.4	95.5	95.4	95.5	95.4	95.6	95.5	95.3	95.4	95.3	95.5	95.5	95.5	95.3
W21	87.4	95.2	95.1	95.2	95.1	95.2	95.3	95.2	95.1	95.1	95.1	95.1	95.2	95.2
W22	91.5	95.1	95.1	95.2	95.1	95.2	95.1	95.2	95.2	95.1	95.2	95.2	95.1	95.2
W23	95.7	95.6	95.7	95.6	95.7	95.6	95.6	95.7	95.6	95.6	95.7	95.7	95.6	95.6
W24	91.1	95.1	95.2	95.1	95.2	95.2	95.1	95.2	95.1	95.2	95.1	95.2	95.2	95.1
W25	87.0	96.0	96.1	96.0	96.0	96.1	96.0	96.1	96.0	96.1	96.0	96.1	96.0	96.1
W26	89.4	99.7	99.8	99.7	99.8	99.7	99.8	99.7	99.8	99.7	99.8	99.6	99.7	99.8
W27	90.4	96.4	96.3	96.4	96.3	96.4	96.4	96.4	96.3	96.4	96.4	96.4	96.3	96.4
W28	99.1	95.7	95.8	95.7	95.6	95.8	95.7	95.6	95.7	95.8	95.7	95.7	95.8	95.9
W29	89.0	96.5	96.4	96.5	96.5	96.5	96.6	96.5	96.6	96.5	96.6	96.5	96.5	96.6
W30	92.8	99.7	99.6	99.7	99.6	99.7	99.6	99.7	99.6	99.8	99.7	99.6	99.7	99.8
W31	96.7	96.7	96.6	96.8	96.7	96.7	96.7	96.8	96.7	96.7	96.8	96.7	96.8	96.8
W32	99.2	96.9	97.2	98.3	97.9	98.7	99.1	98.5	98.7	99.1	96.7	97.8	98.3	99.0

Table 8.27: Codes' efficiencies - PIC (and.b32 , n , write-read) - 14 SMs - OK - Part 2

The code efficiencies of the ELF kernels, generated for the PTX instruction configuration (and.b32 , n , write-read), are shown below. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. The rows of the table represent the number of warps of each thread block, while the columns of the table represent the dependence distances (there is one ELF kernel per dependence distance). Note that we execute the ELF kernels using 14 thread blocks (equal to the number of streaming multiprocessors), and that a Tesla C2070 can execute 448 ($14 \cdot 32$) PTX and.b32.n instructions per functional unit clock cycle. The low code efficiencies are due to the gigathread scheduler that does not fairly assign the thread blocks to the streaming multiprocessors.

	DD15	DD16	DD17	DD18	DD19	DD20	DD21	DD22	DD23	DD24	DD25	DD26	DD27	DD28
W09	49.8	49.8	49.9	49.8	49.8	49.9	49.8	49.8	49.9	49.9	49.8	49.8	49.8	49.8
W10	49.8	49.9	49.8	49.9	49.9	49.8	49.8	49.8	49.9	49.9	49.9	49.9	49.8	49.8
W11	49.9	49.8	49.8	49.9	49.8	49.8	49.9	91.2	91.2	91.2	91.2	91.3	91.2	91.3
W12	49.9	49.8	49.8	49.9	49.8	49.9	49.8	91.2	91.3	91.3	91.2	91.3	91.3	91.2
W13	95.4	95.3	95.4	95.3	95.4	95.3	95.4	95.3	95.4	95.3	95.4	95.3	95.4	95.3
W14	95.4	95.3	95.4	95.3	95.4	95.3	95.3	95.4	95.4	95.3	95.4	95.4	95.4	95.3
W15	95.4	95.3	95.4	95.4	95.3	95.4	95.3	95.4	95.4	95.3	95.3	95.4	95.4	95.3
W16	95.4	95.3	95.4	95.3	95.3	95.4	95.3	95.4	95.3	95.4	95.4	95.3	95.4	95.3
W17	95.4	95.3	95.4	95.3	95.4	95.4	95.4	95.3	95.4	95.3	95.4	95.3	95.3	95.3
W18	95.4	95.4	95.5	95.5	95.4	95.4	95.4	95.4	95.4	95.4	95.4	95.5	95.4	95.4
W19	95.4	95.4	95.3	95.4	95.4	95.3	95.4	95.3	95.4	95.4	95.3	95.4	95.4	95.4
W20	95.4	95.4	95.3	95.4	95.4	95.3	95.4	95.3	95.4	95.4	95.3	95.4	95.4	95.4
W21	95.4	95.3	95.4	95.4	95.3	95.4	95.4	95.4	95.3	95.3	95.4	95.4	95.3	95.4
W22	95.2	95.1	95.2	95.1	95.2	95.2	95.1	95.2	95.1	95.1	95.2	95.2	95.1	95.2
W23	95.6	95.5	95.5	95.6	95.5	95.6	95.4	95.5	95.6	////	////	////	////	////
W24	95.4	95.3	95.4	95.3	95.3	95.4	95.3	////	////	////	////	////	////	////
W25	96.0	96.1	96.0	96.0	////	////	////	////	////	////	////	////	////	////
W26	99.7	99.6	99.7	99.6	////	////	////	////	////	////	////	////	////	////
W27	96.3	96.4	////	////	////	////	////	////	////	////	////	////	////	////
W28	95.8	////	////	////	////	////	////	////	////	////	////	////	////	////
W29	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W30	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W31	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W32	////	////	////	////	////	////	////	////	////	////	////	////	////	////

Table 8.28: Codes' efficiencies - PIC (and.b32 , n , write-read) - 14 SMs - OK - Part 3

The code efficiencies of the ELF kernels, generated for the PTX instruction configuration (and.b32 , n , write-read), are shown below. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. The rows of the table represent the number of warps of each thread block, while the columns of the table represent the dependence distances (there is one ELF kernel per dependence distance). Note that we execute the ELF kernels using 14 thread blocks (equal to the number of streaming multiprocessors), and that a Tesla C2070 can execute 448 ($14 \cdot 32$) PTX and.b32.n instructions per functional unit clock cycle. The low code efficiencies are due to the gigathread scheduler that does not fairly assign the thread blocks to the streaming multiprocessors.

	DD29	DD30	DD31	DD32	DD33	DD34	DD35	DD36	DD37	DD38	DD39	DD40	DD41	DD42
W09	49.9	49.9	49.8	89.8	89.7	89.9	89.8	89.8	89.9	89.7	89.8	89.8	89.8	89.7
W10	49.8	49.9	49.9	89.8	89.7	89.7	89.8	89.8	89.9	89.8	89.8	89.7	89.7	89.7
W11	91.2	91.3	91.2	91.2	91.2	91.3	91.1	91.2	91.3	91.2	91.2	91.2	91.2	91.3
W12	91.3	91.2	91.1	91.2	91.3	91.2	91.3	91.2	91.2	91.3	91.2	91.2	91.2	91.2
W13	95.4	95.4	95.4	95.4	95.3	95.3	95.4	95.4	95.3	95.3	95.4	95.4	95.4	95.3
W14	95.4	95.3	95.4	95.4	95.3	95.4	95.4	95.3	95.4	95.4	95.4	95.3	95.4	95.4
W15	95.4	95.4	95.3	95.4	95.3	95.4	95.4	95.4	95.3	95.3	95.3	95.3	95.4	95.4
W16	95.4	95.3	95.4	95.4	95.4	95.4	95.3	95.4	95.4	95.3	95.4	95.4	95.3	95.4
W17	95.4	95.4	95.4	95.3	95.4	95.4	95.3	95.4	95.3	95.4	95.3	////	////	////
W18	95.4	95.4	95.3	95.4	95.4	95.4	95.3	////	////	////	////	////	////	////
W19	95.4	95.3	95.4	95.5	////	////	////	////	////	////	////	////	////	////
W20	95.4	95.3	////	////	////	////	////	////	////	////	////	////	////	////
W21	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W22	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W23	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W24	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W25	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W26	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W27	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W28	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W29	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W30	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W31	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W32	////	////	////	////	////	////	////	////	////	////	////	////	////	////

Table 8.29: Efficiency differences - PIC (and.b32 , n , write-read)- 14 SMs - Part 1

The efficiency differences between the original and the modified ELF kernels generated for the PTX instruction configuration (and.b32 , n , write-read) are shown below. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. The rows of the table represent the number of warps of each thread block, while the columns of the table represent the dependence distances (there is one ELF kernel per dependence distance). Note that we execute the ELF kernels using 14 thread blocks (equal to the number of streaming multiprocessors), that a Tesla C2070 can execute 448 ($14 \cdot 32$) PTX and.b32.n instructions per functional unit clock cycle, and that the gigathread scheduler always fairly distributes the thread blocks to the streaming multiprocessors for the modified ELF kernels.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	DD09	DD10	DD11	DD12	DD13	DD14
W09	-2.9	-33.1	-38.1	-38.3	-38.1	-38.2	-38.0	-39.3	-38.0	-37.9	-38.1	-38.1	-38.0	-38.3
W10	-3.1	-42.3	-48.0	-48.0	-48.1	-47.9	-47.9	-47.8	-48.0	-47.9	-48.0	-47.8	-48.0	-47.9
W11	-8.8	-39.3	-40.2	-40.1	-39.9	-40.1	-39.8	-39.9	-38.8	-39.9	-39.8	-39.8	-39.7	-39.8
W12	-9.7	-47.4	-48.0	-47.9	-47.6	-47.9	-48.2	-47.8	-47.8	-47.9	-48.0	-47.9	-47.7	-47.9
W13	-16.4	-40.6	-41.0	-41.0	-41.7	-41.3	-41.2	-40.9	-40.8	-40.9	-41.0	-40.9	-40.8	+4.4
W14	-18.8	-47.6	-48.0	-47.9	-47.7	-47.8	-47.7	-47.8	-47.9	-47.9	-48.0	-47.8	-47.9	-2.3
W15	-21.1	-41.9	-42.0	-41.9	-41.9	-42.0	-41.8	-42.0	-41.8	+4.6	+4.6	+4.7	+4.5	+4.7
W16	-24.5	-48.0	-47.9	-48.0	-48.1	-48.0	-47.9	-48.0	-47.9	-2.4	-2.6	-2.4	-2.5	-2.5
W17	-25.7	-42.6	-42.5	-42.7	-42.6	+3.0	+2.9	+3.0	+2.8	+3.0	+2.7	+2.9	+3.0	+2.9
W18	-33.9	-48.0	-48.0	-48.1	-48.0	-2.5	-2.5	-2.4	-2.6	-2.4	-2.5	-2.4	-2.6	-2.5
W19	-21.1	-42.7	-42.5	-42.7	-42.9	+2.6	+1.9	+2.7	+2.4	+2.4	+2.6	+2.4	+2.7	+2.4
W20	+0.1	-2.5	-2.4	-2.4	-2.5	-2.2	-2.4	-2.4	-2.4	-2.5	-2.2	-2.4	-2.3	-2.6
W21	+1.2	+2.0	+1.8	+1.9	+1.6	+1.7	+1.7	+1.6	+1.5	+1.5	+1.7	+1.5	+1.6	+1.9
W22	-2.0	-2.6	-2.8	-2.6	-4.3	-2.6	-2.6	-2.7	-2.7	-2.8	-2.6	-2.7	-2.7	-2.5
W23	+3.0	+1.9	+2.1	+2.8	+2.9	+1.6	+1.8	+2.0	+1.7	+1.8	+1.9	+1.9	+1.8	+1.8
W24	-6.9	-2.8	-2.7	-2.8	-2.7	-2.8	-2.8	-2.7	-2.7	-2.6	-2.1	-2.0	-2.6	-2.7
W25	-0.7	-1.9	-1.8	-1.9	-1.9	-1.9	-1.9	-1.8	-1.8	-1.7	-1.2	-1.1	-1.8	-1.7
W26	-1.9	+5.7	+5.7	+5.0	+3.9	+5.4	+5.6	+5.7	+5.7	+5.3	+5.8	+5.5	+5.7	+5.8
W27	-1.1	+2.2	+1.9	+1.7	+1.9	+1.7	+1.7	+1.9	+1.8	+2.1	+1.9	+2.0	+1.8	+1.8
W28	+3.9	-2.2	-2.0	-1.9	-2.1	-2.0	-2.1	-2.1	-2.0	-2.0	-2.0	-2.1	-1.9	-1.9
W29	-0.4	+2.0	+1.7	+1.9	+2.2	+2.7	+1.8	+1.9	+1.9	+1.9	+1.8	+1.8	+1.9	+1.9
W30	+0.2	+1.8	+1.8	+1.8	+1.6	+2.0	+1.6	+1.9	+1.8	+2.0	+2.0	+1.8	+1.9	////
W31	+1.7	+2.1	+1.8	+1.8	+2.1	+1.7	+1.7	+1.9	+1.7	+2.0	+2.0	+1.8	////	////
W32	+1.0	-0.1	-0.6	+0.5	-0.2	+0.9	+1.3	+2.0	+0.7	-1.2	-1.2	////	////	////

Table 8.30: Efficiency differences - PIC (and.b32 , n , write-read)- 14 SMs - Part 2

The efficiency differences between the original and the modified ELF kernels generated for the PTX instruction configuration (and.b32 , n , write-read) are shown below. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. The rows of the table represent the number of warps of each thread block, while the columns of the table represent the dependence distances (there is one ELF kernel per dependence distance). Note that we execute the ELF kernels using 14 thread blocks (equal to the number of streaming multiprocessors), that a Tesla C2070 can execute 448 ($14 \cdot 32$) PTX and.b32.n instructions per functional unit clock cycle, and that the gigathread scheduler always fairly distributes the thread blocks to the streaming multiprocessors for the modified ELF kernels.

	DD15	DD16	DD17	DD18	DD19	DD20	DD21	DD22	DD23	DD24	DD25	DD26	DD27	DD28
W09	-38.2	-38.1	-38.2	-38.1	-38.2	-38.1	-38.1	-38.1	-38.2	-38.1	-38.3	-38.2	-38.1	-38.1
W10	-47.8	-47.8	-47.9	-47.9	-47.7	-47.8	-47.9	-47.9	-47.9	-47.8	-47.7	-47.9	-47.9	-47.9
W11	-39.6	-39.8	-39.7	-39.7	-39.8	-39.7	-39.5	+1.6	+1.7	+1.6	+1.7	+1.9	+1.6	+1.8
W12	-37.7	-37.9	-37.8	-37.6	-37.8	-37.8	-37.8	-6.3	-6.5	-6.3	-6.4	-6.4	-6.2	-6.5
W13	+5.4	+4.4	+4.6	+4.3	+4.6	+4.4	+4.7	+4.4	+4.4	+4.4	+4.6	+4.4	+4.4	+4.4
W14	-2.2	-2.4	-2.3	-2.5	-2.3	-2.4	-2.3	-2.3	-2.4	-2.4	-2.2	-2.3	-2.3	-2.4
W15	+3.7	+3.7	+3.7	+3.8	+3.6	+3.7	+3.6	+3.8	+3.7	+3.6	+3.7	+3.7	+3.8	+3.6
W16	-2.4	-2.5	-2.2	-2.5	-2.5	-2.2	-2.4	-2.3	-2.5	-2.4	-2.2	-2.3	-2.3	-2.5
W17	+3.1	+2.9	+3.0	+3.0	+3.0	+3.0	+3.1	+2.9	+3.1	+3.0	+3.0	+2.9	+3.0	+3.0
W18	-2.5	-2.4	-2.1	-2.3	-2.5	-2.4	-2.5	-2.4	-2.4	-2.5	-2.4	-2.4	-2.4	-2.5
W19	+2.6	+2.5	+2.3	+2.5	+2.6	+2.3	+2.5	+2.4	+2.5	+2.4	+2.4	+2.4	+2.5	+2.5
W20	-2.5	-2.4	-2.6	-2.4	-2.4	-2.6	-2.4	-2.5	-2.5	-2.5	-2.5	-2.4	-2.4	-2.4
W21	+2.1	+2.1	+2.1	+2.1	+2.1	+2.2	+2.1	+2.2	+2.0	+2.1	+2.1	+2.2	+2.1	////
W22	-2.5	-2.7	-2.5	-2.6	-2.6	-2.5	-2.7	-2.5	-2.7	-2.7	-2.6	-2.5	-2.7	////
W23	+1.8	+1.8	+1.7	+1.9	+1.7	+1.8	+1.7	+1.7	+1.8	////	////	////	////	////
W24	-2.4	-2.4	-2.4	-2.6	-2.5	-2.3	-2.4	////	////	////	////	////	////	////
W25	+1.8	+1.7	+1.7	+1.6	////	////	////	////	////	////	////	////	////	////
W26	+1.9	+1.9	+1.9	+1.8	////	////	////	////	////	////	////	////	////	////
W27	+1.9	+2.1	////	////	////	////	////	////	////	////	////	////	////	////
W28	-1.9	////	////	////	////	////	////	////	////	////	////	////	////	////
W29	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W30	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W31	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W32	////	////	////	////	////	////	////	////	////	////	////	////	////	////

Table 8.31: Efficiency differences - PIC (and.b32 , n , write-read)- 14 SMs - Part 3

The efficiency differences between the original and the modified ELF kernels generated for the PTX instruction configuration (and.b32 , n , write-read) are shown below. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. The rows of the table represent the number of warps of each thread block, while the columns of the table represent the dependence distances (there is one ELF kernel per dependence distance). Note that we execute the ELF kernels using 14 thread blocks (equal to the number of streaming multiprocessors), that a Tesla C2070 can execute 448 ($14 \cdot 32$) PTX and.b32.n instructions per functional unit clock cycle, and that the gigathread scheduler always fairly distributes the thread blocks to the streaming multiprocessors for the modified ELF kernels.

	DD29	DD30	DD31	DD32	DD33	DD34	DD35	DD36	DD37	DD38	DD39	DD40	DD41	DD42
W09	**-38.1**	**-38.2**	**-38.1**	**+1.8**	**+1.6**	**+1.9**	**-2.0**	**+1.9**	**+1.9**	**+1.6**	**+1.7**	**+1.8**	**+2.0**	**+1.8**
W10	**-47.9**	**-47.7**	**-47.9**	**-7.9**	**-8.0**	**-8.1**	**-7.8**	**-7.9**	**-7.9**	**-7.9**	**-8.0**	**-8.1**	**-8.0**	**-8.1**
W11	**+1.6**	**+1.8**	**+1.5**	**+1.5**	**+1.6**	**+1.7**	**+1.4**	**+1.6**	**+1.6**	**+1.6**	**+1.5**	**+1.6**	**+1.5**	**+1.7**
W12	**-6.4**	**-6.4**	**-6.6**	**-6.4**	**-6.4**	**-6.4**	**-6.4**	**-6.5**	**-6.5**	**-6.3**	**-6.5**	**-6.5**	**-6.4**	**-6.5**
W13	**+4.5**	**+4.7**	**+4.6**	**+4.5**	**+4.5**	**+4.5**	**+4.6**	**+4.7**	**+4.3**	**+4.4**	**+4.6**	**+4.5**	**+4.6**	**+4.5**
W14	**-2.3**	**-2.3**	**-2.4**	**-2.2**	**-2.3**	**-2.3**	**-2.2**	**-2.3**	**-2.4**	**-2.4**	**-2.3**	**-2.3**	**-2.2**	**-2.3**
W15	**+4.7**	**+4.8**	**+4.6**	**+4.7**	**+4.7**	**+4.8**	**+4.7**	**+4.7**	**+4.7**	**+4.7**	**+4.6**	**+4.7**	**+4.7**	**+4.7**
W16	**-2.4**	**-2.4**	**-2.3**	**-2.2**	**-2.3**	**-2.3**	**-2.5**	**-2.4**	**-2.3**	**-2.4**	**-2.4**	**-2.2**	**-2.4**	**-2.4**
W17	**+3.0**	**+3.1**	**+3.0**	**+2.9**	**+3.1**	**+3.0**	**+2.9**	**+3.1**	**+3.0**	**+3.0**	**+3.0**	////	////	////
W18	**-2.4**	**-2.3**	**-2.6**	**-2.4**	**-2.4**	**-2.3**	**-2.3**	////	////	////	////	////	////	////
W19	**+2.4**	**+2.4**	**+2.3**	**+2.5**	////	////	////	////	////	////	////	////	////	////
W20	**-2.5**	**-2.5**	////	////	////	////	////	////	////	////	////	////	////	////
W21	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W22	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W23	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W24	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W25	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W26	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W27	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W28	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W29	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W30	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W31	////	////	////	////	////	////	////	////	////	////	////	////	////	////
W32	////	////	////	////	////	////	////	////	////	////	////	////	////	////

8.2 (ELF + Insight) Vs (PTX + Compiler)

We were also curious to test nvcc compiler's efficacy. What we found it is that nvcc does not perform well even for very simple kernels, and that therefore its efficacy is low. Furthermore, we discovered that nvcc does not know that the gigathread scheduler does not always fairly distribute the thread blocks to the streaming multiprocessors. However, these two things (nvcc transformations and thread block distributions) can very easily generate losses greater than 40% of the total performance compared to when we produce and run ELF kernels using the insight we earned by the reverse engineering and the discovery and quantification of the low level architectural features and machine behaviors. To show this, let us consider the following types of cases:

- $Cases_{t5}$: let us take the same PTX kernels we generated for the microbenchmarks and instead translate them to ELF kernels using nvcc. During the translation, nvcc is free to apply all the optimizing transformations it thinks are useful to speedup the code executions and therefore to increase their efficiencies.

  To further simplify the experiments, let us use for the launches only one thread block with between 1 and 32 warps. In this case, we exploit only one streaming multiprocessor, and all the data reside in the hardware registers.

  However, even for the very simple microbenchmark kernels and these very simple executions, the ELF kernels produced by nvcc usually have smaller throughputs than the throughputs of the ELF kernels that we generated for the discovery and quantification of the low level architectural features and machine behaviors.

- $Cases_{t6}$: we can use any virtual number and type of PTX registers when programming is conducted in PTX. However, during the reverse engineering, we discovered that there are 2 types of ELF registers, predicate and normal, and that the maximum number of hardware registers per type that each thread can have is different (i.e. 8 predicate

registers, 1 of which is reserved; 64 normal registers, 3 of which are reserved; and a total of not more than 64 registers).

Therefore, to study nvcc's efficacy, we used nvcc for translating the PTX kernels that are analogous to the very simple PTX kernels that we used for the the discovery and quantification of the low level architectural features. However, these PTX kernels have a greater number and type of PTX registers than the maximum number of hardware registers, per type, that each thread can have.

After running the transformed PTX kernels with different launch configurations, we discovered that even for very simple launch configurations, the nvcc's efficacy was very low and that therefore the code efficiencies were also very low, which is caused by the spill load and spill store phenomena. We do not know why nvcc does not split the work into different time phases (since the nvcc code is closed). However, nvcc could split the work since the kernels are very simple.

What it is clear, it is that even for very simple kernels, users cannot use a random number of PTX registers. Furthermore, it is hard to determine the right number and type of virtual PTX resources to allocate to each thread without the insights earned from the reverse engineering phases, and from the discovery and quantification of the low level architectural features and machine behaviors.

- $Cases_{t7}$: finally, we wanted to verify if nvcc has any knowledge about the fact that the gigathread scheduler does not always fairly assign the thread blocks to the streaming multiprocessors. We therefore re-run the ELF kernels produced by nvcc for the PTX instruction configurations. For the ELF kernel executions, we used launch configurations with a number of thread blocks that is a factor of the number of streaming multiprocessors (14), and with a number of warps per thread block greater than 9. We did this because, with more than 9 warps per thread block, and using analogous ELF kernels produced by exploiting the insight earned from the reverse engineering phase,

we can use procedure 8.18 to force the gigathread scheduler to fairly distribute the thread blocks to the streaming multiprocessor.

For all the cases, the efficiencies of ELF kernels produced by nvcc were always smaller than the efficiencies of the same ELF kernels produced by exploiting the reverse engineering, the quantification of the low level architectural features and machine behaviors, and procedure 8.18.

Table 8.32: Legend for the low compiler's efficacy examples

```
PIC: PTX instruction configuration (e.g. [ xor.b32 , n , read-read ] )
EIC: ELF instruction configuration (e.g. [ IADD , c.no , write-read ] )

MK: modified kernels ( greater number of registers per thread )

WG: we generated the kernels ( used ELF or [ ELF + insight ] )
CG: the compiler generated the kernels ( compiler efficacy )

WX: number of warps per thread block ( WX ranges from 1 to 32 )
TB: number of thread blocks used for the launches ( usually 1 or 14 )
SM: streaming multiprocessors (1 or K x 14 with K integer and positive )
```

Table 8.33: Second group of experiments

Exp	Type of Case	PTX Instruction Configuration	Tables	Pages
5	5	(add.u32 , n , wr)	8.34 - 8.35	141 - 142
			8.36 - 8.37	143 - 144
			8.38 - 8.39	145 - 146
			8.40 - 8.41	147 - 148
			8.42 - 8.43	149 - 150
6	6	(fma.rm.f32 , n , wr)	8.44 - 8.49	151 - 156
7	7	(add.s32 , n , wr)	8.50	157
			8.51	158
			8.52	159

Table 8.34: Throughputs - PIC (add.u32 , n , write-read) - 1 SM - WG - Part 1

If we generate the ELF kernels necessary to execute the PTX kernels generated for the PTX instruction configuration (add.u32 , n , write-read) then a streaming multiprocessor is able to execute a maximum (MT) of 32 PTX add.u32.n instructions per functional unit clock cycle. This implies that the PTX add.u32.n instructions are executed by the 2 groups of 16 CUDA cores. The PTX add.s32.n instruction's latency (IL or write-read latency WRL) cannot be greater than 24 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red. WX is the number of warps that reside in the unique streaming multiprocessor used, while DDY is the dependence distance among instructions. For an explanation of the colors and symbols, see Tables 7.14, 7.15, and 7.16.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	1.59	3.15	4.62	5.07	5.09	5.07	5.09	5.07	....
W02	3.18	6.29	9.24	10.14	10.17	10.14	10.18	10.14	....
W03	4.77	9.44	13.82	15.21	15.25	15.21	15.27	15.21	....
W04	6.36	12.58	18.43	20.28	20.34	20.28	20.36	20.28	....
W05	7.94	15.44	21.45	25.13	25.01	25.12	25.13	25.13	....
W06	9.53	18.54	25.67	31.12	31.26	31.13	31.15	31.12	....
W07	11.12	21.70	26.90	26.87	26.87	26.88	26.89	26.88	....
W08	12.70	24.66	31.44	31.29	31.37	31.69	31.71	31.68	....
W09	14.28	26.26	27.91	27.88	27.94	27.94	27.97	27.89	....
W10	15.86	28.14	31.56	31.54	31.45	31.56	31.23	31.57	....
W11	17.43	28.31	28.45	28.38	28.41	28.43	28.41	28.38	....
W12	19.01	31.27	31.35	31.76	31.37	31.56	31.76	31.78	....
W13	20.52	28.75	28.79	28.79	28.81	28.80	28.83	28.80	....
W14	22.07	31.32	31.40	31.89	31.52	31.68	31.53	31.78	....
W15	23.33	29.02	29.03	29.06	29.07	29.05	29.06	29.05	....
W16	24.83	31.78	31.56	31.96	31.76	31.36	31.57	31.78	....
W17	25.49	29.29	29.33	29.28	29.31	29.31	29.32	29.29	....
W18	26.99	31.27	31.26	31.99	31.52	31.35	31.47	31.78	....
W19	27.54	29.42	29.42	29.43	29.50	29.47	29.48	29.45	....
W20	28.94	31.35	31.57	31.95	31.45	31.47	31.54	31.87	....
W21	27.47	29.55	29.58	29.64	29.65	29.65	29.67	29.64	....
W22	28.80	31.65	31.56	31.97	31.76	31.56	31.76	31.98	....
W23	29.29	29.69	29.70	29.79	29.78	29.79	29.79	29.79	....
W24	31.08	31.27	31.76	31.86	31.67	31.76	31.64	31.76	....
W25	27.98	29.80	29.83	29.89	29.87	29.88	29.87	29.87	....
W26	31.14	31.45	31.98	31.98	31.74	31.98	31.86	31.89	....
W27	29.08	29.87	29.92	29.92	29.95	29.95	29.97	29.96	....
W28	31.21	31.53	31.98	31.97	31.76	31.98	31.78	31.98	....
W29	28.48	29.93	30.01	30.03	30.02	30.04	30.04	30.02	....
W30	31.51	31.47	31.98	31.97	31.67	31.94	31.89	31.97	....

Table 8.35: Throughputs - PIC (add.u32 , n , write-read) - 1 SM - WG - Part 2

If we generate the ELF kernels necessary to execute the PTX kernels generated for the PTX instruction configuration (add.u32 , n , write-read), then a streaming multiprocessor is able to execute a maximum (MT) of 32 PTX add.s32.n instructions per functional unit clock cycle. This implies that the PTX add.s32.n instructions are executed by the 2 groups of 16 CUDA cores. The PTX add.u32.n instruction's latency (IL or write-read latency WRL) cannot be greater than 24 functional unit clock cycles. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. Note the presence of local instabilities in red. WX is the number of warps that reside in the unique streaming multiprocessor used, while DDY is the dependence distance among instructions. For an explanation of the colors and symbols, see Tables 7.14, 7.15, and 7.16.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	5.11	5.11	5.12	5.11	5.12	5.11	5.12	5.11	....
W02	10.23	10.22	10.25	10.23	10.25	10.22	10.24	10.23	....
W03	15.34	15.33	15.37	15.34	15.37	15.33	15.35	15.34	....
W04	20.46	20.44	20.50	20.46	20.50	20.44	20.47	20.46	....
W05	25.40	25.38	25.34	25.44	25.34	25.38	25.29	25.40	....
W06	31.47	31.44	31.40	31.47	31.40	31.47	31.40	31.44	....
W07	27.17	27.13	27.16	27.15	27.16	27.13	27.12	27.15	....
W08	31.02	30.99	31.03	31.02	31.01	30.99	30.98	31.01	....
W09	28.24	28.10	28.18	28.09	28.19	28.19	28.14	28.16	....
W10	31.32	31.26	31.27	31.27	31.24	31.29	31.25	31.29	....
W11	28.69	28.66	28.71	28.63	28.62	28.68	28.64	28.73	....
W12	31.20	31.26	31.27	31.23	31.27	31.26	31.21	31.25	....
W13	29.10	29.02	29.11	29.04	29.04	29.04	29.06	29.07	....
W14	31.24	31.26	31.29	31.26	31.24	31.26	31.23	31.28	....
W15	29.39	29.34	29.35	29.39	29.39	29.33	29.32	29.36	....
W16	31.31	31.27	31.30	31.31	31.30	31.29	31.27	31.31	....
W17	29.56	29.56	29.58	29.58	29.57	29.56	29.54	29.57	....
W18	31.31	31.29	31.30	31.31	31.30	31.29	31.27	31.31	....
W19	29.76	29.76	29.67	29.66	29.71	29.65	29.72	29.71	....
W20	31.27	31.29	31.32	31.31	31.31	31.31	31.26	31.22	....
W21	29.99	29.93	29.82	29.99	29.92	29.90	29.85	29.84	....
W22	31.30	31.29	31.30	31.29	31.31	31.29	31.23	31.35	....
W23	30.01	29.98	30.06	30.08	30.01	30.01	29.90	29.99	....
W24	31.31	31.35	31.31	31.31	31.36	31.29	31.27	31.32	....
W25	30.12	30.15	30.11	30.12	30.13	30.12	30.10	30.13	....
W26	31.31	31.27	31.30	31.29	31.30	31.31	31.29	31.30	....
W27	30.23	30.26	30.23	30.24	30.22	30.23	30.25	30.23	....
W28	31.28	31.24	31.25	31.27	31.24	31.26	31.25	////	....
W29	30.18	30.19	30.18	30.18	30.17	30.18	////	////	....
W30	31.32	31.31	31.33	31.32	31.34	31.32	////	////	....

Table 8.36: Codes' efficiencies - PIC (add.u32 , n , write-read) - 1 SM - WG - Part 1

The code efficiency percentages for the ELF kernels that we generated for the PTX instruction configuration (add.u32 , n , write-read) are shown below. Each ELF kernel was launched using only 1 thread block and only one streaming multiprocessor. We were able to generate the desired ELF kernels thanks to the insight earned during the reverse engineering phase, but we did not assume anything about the low level architectural features and machine behaviors.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	5.0	9.8	14.4	15.8	15.9	15.8	15.9	15.9	....
W02	9.9	19.7	28.9	31.7	31.7	31.7	31.8	31.7	....
W03	14.9	29.5	43.2	47.5	47.7	47.5	47.4	47.6	....
W04	19.9	39.3	57.6	63.4	63.6	63.4	63.6	63.4	....
W05	24.8	48.3	67.0	78.5	78.2	78.5	78.5	78.6	....
W06	29.8	57.9	80.2	97.3	97.7	97.3	97.3	97.3	....
W07	34.8	67.8	84.0	84.0	84.0	84.0	84.0	84.0	....
W08	39.7	77.0	98.3	97.8	98.0	99.0	99.0	99.0	....
W09	44.6	82.1	87.2	87.1	87.3	87.3	87.4	87.4	....
W10	49.3	87.9	98.7	98.6	98.3	98.7	97.6	98.4	....
W11	54.5	88.5	88.9	88.7	88.8	88.8	88.8	88.8	....
W12	59.4	97.7	97.7	99.2	98.0	98.6	99.3	99.3	....
W13	64.1	89.8	90.0	90.0	90.0	90.0	90.0	90.0	....
W14	69.0	97.9	98.1	99.7	98.5	99.0	98.5	98.6	....
W15	72.9	90.7	90.7	90.8	90.8	90.8	90.8	90.8	....
W16	77.6	99.3	98.6	99.9	99.3	98.0	98.7	98.9	....
W17	79.7	91.5	91.7	91.5	91.6	91.6	91.7	91.7	....
W18	84.3	97.7	97.7	100	98.5	98.0	98.3	98.7	....
W19	86.1	91.9	91.9	92.0	92.2	92.1	92.1	92.1	....
W20	90.4	98.0	98.7	99.8	98.3	98.3	98.6	98.7	....
W21	85.8	92.3	92.4	92.6	92.7	92.7	92.7	92.7	....
W22	90.0	98.9	98.6	99.9	99.3	98.6	99.3	99.2	....
W23	91.5	92.8	92.8	93.1	93.1	93.1	93.1	93.2	....
W24	97.1	97.7	99.3	99.6	99.0	99.3	98.9	99.0	....
W25	87.4	93.1	93.2	93.4	93.3	93.4	93.3	93.4	....
W26	97.3	98.3	99.9	99.9	99.2	99.9	99.6	99.7	....
W27	90.9	93.3	93.5	93.5	93.6	93.6	93.7	93.6	....
W28	97.5	98.5	99.9	99.9	99.3	99.9	99.3	99.7	....
W29	89.0	93.5	93.8	93.8	93.8	93.9	93.9	93.8	....
W30	98.5	98.3	99.9	99.9	99.0	99.8	99.7	99.7	....
W31	94.3	93.9	93.8	93.8	94.0	94.0	94.0	94.0	....
W32	99.2	98.9	99.9	99.8	99.8	100	99.8	99.9	....

Table 8.37: Codes' efficiencies - PIC (add.u32 , n , write-read) - 1 SM - WG - Part 2

The code efficiency percentages for the ELF kernels that we generated for the PTX instruction configuration (add.u32 , n , write-read) are shown below. Each ELF kernel was launched using only 1 thread block and only one streaming multiprocessor. We were able to generate the desired ELF kernels thanks to the insight earned during the reverse engineering phase, but we did not assume anything about the low level architectural features and machine behaviors.

	DD09	DD10	DD11	DD12	DD13	DD14	DD15	DD16	....
W01	15.8	15.9	15.9	15.8	15.9	15.8	15.8	15.8	....
W02	31.8	31.7	31.7	31.8	31.7	31.8	31.7	31.7	....
W03	47.7	47.8	47.5	47.6	47.7	47.8	47.7	47.7	....
W04	63.6	63.7	63.6	63.7	63.7	63.5	63.6	63.6	....
W05	78.5	78.5	78.6	78.5	78.5	78.5	78.6	78.5	....
W06	97.4	97.3	97.3	97.3	97.4	97.3	97.4	97.4	....
W07	84.0	84.1	84.0	84.1	84.0	84.1	84.0	84.1	....
W08	99.1	99.2	99.0	99.0	99.1	99.1	99.1	99.1	....
W09	87.3	87.4	87.4	87.3	87.4	87.4	87.4	87.6	....
W10	98.6	97.8	98.6	98.4	98.6	98.7	98.5	98.6	....
W11	88.7	88.8	88.9	88.8	88.9	88.8	88.8	88.8	....
W12	99.2	98.9	99.2	99.1	99.0	98.9	99.3	99.4	....
W13	90.1	90.0	90.1	90.1	90.0	90.0	90.0	90.1	....
W14	99.0	98.7	98.9	99.0	99.1	99.0	98.6	99.5	....
W15	90.8	90.9	90.8	90.9	90.8	90.9	90.8	90.9	....
W16	99.0	99.1	98.9	99.0	99.1	99.0	98.9	99.2	....
W17	91.6	91.7	91.8	91.8	91.7	91.7	91.7	91.7	....
W18	99.0	98.7	99.1	98.9	98.9	99.0	98.9	99.1	....
W19	92.0	92.2	92.3	92.0	92.1	92.2	92.1	92.2	....
W20	98.6	98.7	98.6	98.9	98.8	98.9	98.8	98.6	....
W21	92.9	92.8	92.9	92.8	92.7	92.8	92.8	92.9	....
W22	99.1	98.9	99.0	99.1	98.9	99.1	99.2	99.1	....
W23	93.1	93.2	93.2	93.2	93.2	93.3	93.2	93.4	....
W24	99.1	98.9	99.0	99.1	99.2	98.9	99.1	99.1	....
W25	93.3	93.4	93.5	93.4	93.4	93.3	93.4	93.3	....
W26	99.6	99.7	99.5	99.7	99.6	99.6	99.7	99.7	....
W27	93.5	93.7	93.5	93.6	93.5	93.6	93.6	93.5	....
W28	99.6	99.7	99.5	99.9	99.8	99.7	99.8	/////	....
W29	93.9	93.8	93.8	93.8	93.8	93.9	/////	/////	....
W30	99.6	99.9	99.8	99.7	99.6	99.7	/////	/////	....
W31	94.1	94.2	94.0	94.1	94.1	94.0	/////	/////	....
W32	99.8	100	99.9	99.7	99.8	99.9	/////	/////	....

Table 8.38: Throughputs - PIC (add.u32 , n , write-read) - 1 SM - CG - Part 1

If nvcc generates the ELF kernels necessary to execute the PTX kernels generated for the PTX instruction configuration (add.u32 , n , write-read), then a streaming multiprocessor is able to execute a maximum (MT) of 16 PTX add.s32.n instructions per functional unit clock cycle.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	1.59	2.89	3.41	3.41	3.18	3.40	2.93	3.35	....
W02	3.19	5.77	6.84	6.84	6.37	6.83	5.88	6.71	....
W03	4.78	8.66	10.25	10.25	9.55	10.25	8.82	10.07	....
W04	6.38	11.54	13.67	13.67	12.73	13.66	11.76	13.42	....
W05	7.97	14.43	15.27	15.13	14.94	14.94	13.93	15.19	....
W06	9.56	15.88	15.89	15.89	15.78	15.90	14.53	15.90	....
W07	11.16	15.09	15.55	15.55	15.36	15.56	15.23	15.52	....
W08	12.75	15.92	15.92	15.92	15.93	15.93	15.39	15.94	....
W09	14.35	15.71	15.99	16.00	15.88	16.00	15.77	15.97	....
W10	15.94	15.94	15.94	15.94	15.94	15.95	15.96	15.97	....
W11	15.95	15.99	16.00	15.99	15.98	16.01	15.90	16.02	....
W12	15.96	15.96	15.95	15.96	15.96	15.96	15.97	15.98	....
W13	15.43	15.97	16.01	16.01	15.89	16.02	15.95	16.00	....
W14	15.97	15.98	15.96	15.97	16.02	15.97	15.98	16.02	....
W15	15.66	16.00	16.01	16.01	15.98	16.01	15.97	16.03	....
W16	15.97	15.97	15.97	15.97	15.98	15.98	15.99	15.99	....
W17	15.80	16.00	16.01	16.01	16.02	16.01	16.00	16.03	....
W18	15.98	15.97	15.98	15.98	15.98	15.98	16.00	16.00	....
W19	15.90	16.01	16.01	16.02	16.02	16.02	16.01	16.04	....
W20	15.98	15.98	15.98	15.98	15.99	15.99	16.00	16.00	....
W21	15.51	16.01	16.01	16.02	16.02	16.02	16.02	16.04	....
W22	16.00	15.98	15.98	15.99	15.99	15.99	16.04	16.00	....
W23	16.00	16.01	16.01	16.02	16.02	16.02	16.03	16.04	....
W24	15.98	15.99	15.99	15.99	15.99	15.99	16.00	16.01	....
W25	15.71	16.01	16.01	16.02	16.02	16.03	16.03	16.04	....
W26	16.00	15.99	15.99	15.99	16.00	16.00	16.04	16.01	....
W27	16.00	16.01	16.01	16.02	16.02	16.03	16.03	16.04	....
W28	15.99	15.99	15.99	15.99	16.00	16.00	16.01	16.01	....
W29	15.83	16.01	16.01	16.01	16.02	16.03	16.03	16.04	....
W30	16.00	15.99	15.99	15.99	16.00	16.00	16.04	16.01	....
W31	16.00	16.01	16.01	16.02	16.02	16.03	16.03	16.04	....
W32	15.99	15.99	15.99	16.00	16.00	16.00	16.01	16.01	....

Table 8.39: Throughputs - PIC (add.u32 , n , write-read) - 1 SM - CG - Part 2

If nvcc generates the ELF kernels necessary to execute the PTX kernels generated for the PTX instruction configuration (add.u32 , n , write-read), then a streaming multiprocessor is able to execute a maximum (MT) of 16 PTX add.s32.n instructions per functional unit clock cycle.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	3.41	3.31	3.38	3.42	3.35	3.39	3.42	3.32	....
W02	6.83	6.64	6.77	6.87	6.71	6.79	6.86	6.65	....
W03	10.25	9.96	10.16	10.30	10.06	10.19	10.30	9.97	....
W04	13.66	13.27	13.55	13.73	13.42	13.58	13.72	13.30	....
W05	15.16	15.07	15.09	15.15	15.05	15.10	15.10	15.14	....
W06	15.90	15.90	15.97	15.97	15.97	15.99	15.98	15.99	....
W07	15.57	15.51	15.56	15.61	15.54	15.58	15.62	15.55	....
W08	15.95	15.95	16.01	16.01	16.01	16.01	16.03	16.02	....
W09	16.02	15.96	16.00	16.04	15.99	16.02	16.05	16.00	....
W10	15.97	15.97	16.01	16.02	16.02	16.02	16.03	16.04	....
W11	16.03	16.03	16.04	16.05	16.04	16.06	16.06	16.05	....
W12	15.98	15.98	16.03	16.02	16.04	16.04	16.04	16.04	....
W13	16.03	16.00	16.02	16.05	16.02	16.05	16.06	16.04	....
W14	15.99	16.03	16.05	16.02	16.07	16.06	16.04	16.08	....
W15	16.03	16.04	16.05	16.05	16.06	16.06	16.07	16.08	....
W16	15.99	16.00	16.03	16.03	16.04	16.04	16.04	16.05	....
W17	16.04	16.04	16.05	16.05	16.06	16.06	16.07	16.08	....
W18	16.00	16.00	16.03	16.03	16.04	16.04	16.04	16.05	....
W19	16.04	16.04	16.05	16.06	16.06	16.07	16.07	16.08	....
W20	16.00	16.00	16.03	16.03	16.04	16.04	16.05	16.05	....
W21	16.04	16.05	16.05	16.06	16.06	16.07	16.07	16.08	....
W22	16.01	16.01	16.03	16.03	16.04	16.05	16.05	16.05	....
W23	16.04	16.05	16.05	16.06	16.06	16.07	16.07	16.08	....
W24	16.01	16.01	16.03	16.03	16.04	16.05	16.05	16.05	....
W25	16.04	16.05	16.05	16.06	16.06	16.07	16.07	16.08	....
W26	16.01	16.01	16.03	16.03	16.04	16.05	16.05	16.05	....
W27	16.04	16.05	16.05	16.06	16.07	16.07	16.08	16.08	....
W28	16.01	16.02	16.03	16.03	16.04	16.05	16.05	16.05	....
W29	16.04	16.05	16.05	16.06	16.06	16.07	16.08	16.08	....
W30	16.01	16.02	16.03	16.03	16.04	16.05	16.05	16.05	....
W31	16.04	16.05	16.06	16.06	16.07	16.07	16.08	16.08	....
W32	16.02	16.02	16.03	16.04	16.04	16.05	16.05	16.06	....

Table 8.40: Codes' efficiencies - PIC (add.u32 , n , write-read) - 1 SM - CG - Part 1

The code efficiency percentages for the ELF kernels that nvcc generated for the PTX instruction configuration (add.u32 , n , write-read) are shown below. Each ELF kernel was launched using only 1 thread block and only one streaming multiprocessor. Nvcc was free to apply any optimizing transformation it believed could be useful to speedup code executions.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	5.0	9.0	10.7	10.7	10.0	10.6	9.2	10.5	....
W02	10.0	18.0	21.4	21.4	20.0	21.3	18.4	21.0	....
W03	14.9	27.1	32.0	32.0	29.8	32.0	27.6	31.5	....
W04	19.9	36.1	42.7	42.7	39.8	42.7	36.8	41.9	....
W05	24.9	45.1	47.7	47.3	46.7	46.7	43.5	47.5	....
W06	29.9	49.6	49.7	49.7	49.3	49.7	45.4	49.7	....
W07	34.9	47.2	48.6	48.6	48.0	48.6	47.6	48.5	....
W08	39.8	49.8	49.8	49.8	49.8	49.8	48.1	49.8	....
W09	44.8	49.1	50.0	50.0	49.6	50.0	49.3	49.9	....
W10	49.8	49.8	49.8	49.8	49.8	49.8	49.9	49.9	....
W11	49.8	50.0	50.0	50.0	49.9	50.0	49.7	50.0	....
W12	49.9	49.9	49.8	49.9	49.9	49.9	49.9	49.9	....
W13	48.2	49.9	50.0	50.0	49.7	50.1	49.8	50.0	....
W14	49.9	49.9	49.9	49.9	50.1	49.9	49.9	50.1	....
W15	48.9	50.0	50.0	50.0	49.8	50.0	49.9	50.1	....
W16	49.9	49.9	49.9	49.9	49.9	49.9	50.0	50.0	....
W17	49.4	50.0	50.0	50.0	50.0	50.0	50.0	50.1	....
W18	49.9	49.9	49.9	49.9	49.9	49.9	50.0	50.0	....
W19	49.7	50.0	50.0	50.1	50.1	50.1	50.1	50.1	....
W20	49.9	49.9	49.9	49.9	50.0	50.0	50.0	50.0	....
W21	48.5	50.0	50.0	50.1	50.1	50.1	50.1	50.1	....
W22	50.0	49.9	49.9	50.0	50.0	50.0	50.1	50.0	....
W23	50.0	50.0	50.0	50.1	50.1	50.1	50.1	50.1	....
W24	49.9	50.0	50.0	50.0	50.0	50.0	50.0	50.0	....
W25	49.1	50.0	50.0	50.1	50.1	50.1	50.1	50.1	....
W26	50.0	50.0	50.0	50.0	50.0	50.1	50.1	50.0	....
W27	50.0	50.0	50.0	50.1	50.1	50.1	50.1	50.1	....
W28	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	....
W29	49.5	50.0	50.0	50.0	50.1	50.1	50.1	50.1	....
W30	50.0	50.0	50.0	50.0	50.0	50.0	50.1	50.1	....
W31	50.0	50.0	50.0	50.1	50.1	50.1	50.1	50.1	....
W32	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	....

Table 8.41: Codes' efficiencies - PIC (add.u32 , n , write-read) - 1 SM - CG - Part 2

The code efficiency percentages for the ELF kernels that nvcc generated for the PTX instruction configuration (add.u32 , n , write-read) are shown below. Each ELF kernel was launched using only 1 thread block and only one streaming multiprocessor. Nvcc was free to apply any optimizing transformation it believed could be useful to speedup code executions.

	DD09	DD10	DD11	DD12	DD13	DD14	DD15	DD16	....
W01	10.7	10.3	10.6	10.7	10.5	10.6	10.7	10.4	....
W02	21.3	20.8	21.2	21.5	21.0	21.2	21.4	20.8	....
W03	32.0	31.1	31.8	32.2	31.4	31.8	32.2	31.2	....
W04	42.7	41.5	42.3	42.9	41.9	42.4	42.9	41.2	....
W05	47.4	47.1	47.2	47.3	47.0	47.2	47.2	47.3	....
W06	49.7	49.7	49.9	49.9	49.9	50.0	50.0	50.0	....
W07	48.7	48.5	48.6	48.8	48.6	48.7	48.8	48.6	....
W08	49.8	49.8	50.0	50.0	50.0	50.0	50.1	50.1	....
W09	50.1	49.9	50.0	50.1	50.0	50.1	50.2	50.0	....
W10	49.9	49.9	50.0	50.1	50.1	50.1	50.1	50.1	....
W11	50.1	50.1	50.1	50.2	50.1	50.2	50.2	50.1	....
W12	50.0	50.0	50.1	50.1	50.1	50.1	50.1	50.1	....
W13	50.1	50.1	50.1	50.2	50.1	50.2	50.2	50.1	....
W14	50.0	50.1	50.2	50.1	50.2	50.1	50.1	50.3	....
W15	50.1	50.1	50.2	50.2	50.2	50.2	50.2	50.3	....
W16	50.0	50.0	50.1	50.1	50.1	50.1	50.1	50.2	....
W17	50.1	50.1	50.2	50.2	50.2	50.2	50.2	50.3	....
W18	50.1	50.1	50.1	50.1	50.1	50.1	50.1	50.2	....
W19	50.1	50.1	50.2	50.2	50.2	50.2	50.2	50.3	....
W20	50.0	50.0	50.1	50.1	50.1	50.1	50.2	50.2	....
W21	50.1	50.2	50.2	50.2	50.2	50.2	50.2	50.3	....
W22	50.0	50.0	50.1	50.1	50.1	50.2	50.2	50.2	....
W23	50.1	50.2	50.2	50.2	50.2	50.2	50.2	50.3	....
W24	50.0	50.0	50.1	50.1	50.1	50.2	50.2	50.2	....
W25	50.1	50.2	50.2	50.2	50.2	50.2	50.2	50.3	....
W26	50.0	50.0	50.1	50.1	50.1	50.2	50.2	50.2	....
W27	50.1	50.2	50.2	50.2	50.2	50.2	50.3	50.3	....
W28	50.0	50.1	50.1	50.1	50.1	50.2	50.2	50.2	....
W29	50.1	50.2	50.2	50.2	50.2	50.2	50.3	50.3	....
W30	50.0	50.1	50.1	50.1	50.1	50.2	50.2	50.2	....
W31	50.1	50.2	50.2	50.2	50.2	50.2	50.3	50.3	....
W32	50.1	50.1	50.1	50.1	50.1	50.2	50.2	50.2	....

Table 8.42: Efficiency differences - PIC (add.u32 , n , write-read) - 1 SM - Part 1

The efficiency difference percentages between the ELF kernels that nvcc generated and the ELF kernels that we generated for the PTX instruction configuration (add.u32 , n , write-read) are shown below. Each ELF kernel was launched using only 1 thread block and only 1 streaming multiprocessor.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	....
W01	0.0	-0.8	-3.7	-5.1	-5.9	-5.2	-6.7	-5.4	....
W02	+0.1	-1.7	-7.5	-10.3	-11.7	-10.4	-13.4	-10.7	....
W03	0.0	-2.4	-11.2	-15.5	-17.9	-15.5	-19.8	-16.1	....
W04	0.0	-3.2	-14.9	-20.7	-23.8	-20.7	-26.8	-21.5	....
W05	+0.1	-3.2	-19.3	-31.2	-31.5	-31.8	-35.0	-31.1	....
W06	+0.1	-8.3	-30.5	-47.6	-48.4	-47.6	-51.9	-47.6	....
W07	+0.1	-20.6	-35.4	-35.6	-36.0	-35.4	-36.4	-35.5	....
W08	+0.1	-27.2	-48.5	-48.0	-48.2	-49.2	-50.9	-49.2	....
W09	+0.2	-33.0	-37.2	-37.1	-37.5	-37.3	-38.1	-37.5	....
W10	+0.5	-38.1	-48.9	-48.8	-48.5	-38.9	-47.7	-48.5	....
W11	-4.7	-38.5	-38.9	-38.7	-38.9	-38.8	-39.1	-38.8	....
W12	-9.5	-47.8	-47.9	-49.3	-48.1	-48.7	-49.4	-49.4	....
W13	-15.9	-39.9	-40.0	-40.0	-40.3	-39.9	-40.2	-40.0	....
W14	-19.1	-48.0	-48.2	-49.8	-48.4	-49.1	-48.6	-48.5	....
W15	-24.0	-40.7	-40.7	-40.8	-41.0	-40.8	-40.9	-40.7	....
W16	-27.7	-49.4	-48.7	-50.0	-49.4	-48.1	-48.7	-48.9	....
W17	-20.3	-41.5	-41.7	-41.5	-41.6	-41.6	-41.7	-41.6	....
W18	-34.4	-47.8	-47.8	-50.1	-48.6	-48.1	-48.3	-48.7	....
W19	-36.4	-41.9	-41.9	-41.9	-42.1	-42.0	-42.0	-42.0	....
W20	-40.5	-48.1	-48.8	-49.9	-48.3	-48.3	-48.6	-48.7	....
W21	-37.3	-42.3	-42.4	-42.5	-42.6	-42.6	-42.8	-42.6	....
W22	-40.0	-49.0	-48.7	-49.9	-49.3	-48.6	-49.4	-49.2	....
W23	-41.5	-42.8	-42.8	-43.0	-43.0	-43.0	-43.2	-43.1	....
W24	-47.2	-47.7	-49.3	-49.6	-49.0	-49.3	-48.9	-49.0	....
W25	-38.3	-43.1	-43.2	-43.3	-43.2	-43.3	-43.4	-43.4	....
W26	-47.3	-48.3	-49.9	-49.9	-49.2	-49.8	-49.7	-49.7	....
W27	-40.9	-43.3	-43.5	-43.4	-43.5	-43.5	-43.6	-43.5	....
W28	-47.5	-48.5	-49.9	-49.9	-49.3	-49.9	-49.3	-49.7	....
W29	-39.5	-43.5	-43.8	-43.8	-43.7	-43.8	-44.0	-43.7	....
W30	-48.5	-48.3	-49.9	-49.9	-49.0	-49.8	-49.6	-49.6	....
W31	-44.3	-43.9	-43.8	-43.7	-43.9	-43.9	-43.9	-43.9	....
W32	-49.2	-48.9	-49.9	-49.8	-49.8	-50.0	-49.8	-49.9	....

Table 8.43: Efficiency differences - PIC (add.u32 , n , write-read) - 1 SM - Part 2

The efficiency difference percentages between the ELF kernels that nvcc generated and the ELF kernels that we generated for the PTX instruction configuration (add.u32 , n , write-read) are shown below. Each ELF kernel was launched using only 1 thread block and only 1 streaming multiprocessor.

	DD09	DD10	DD11	DD12	DD13	DD14	DD15	DD16	....
W01	**-5.1**	**-5.6**	**-5.3**	**-5.1**	**-5.4**	**-5.2**	**-5.1**	**-5.4**	....
W02	**-10.5**	**-10.9**	**-10.5**	**-10.3**	**-10.7**	**-10.6**	**-10.3**	**-10.9**	....
W03	**-15.7**	**-16.7**	**-15.7**	**-15.4**	**-16.3**	**-16.0**	**-15.5**	**-16.5**	....
W04	**-20.9**	**-22.2**	**-21.3**	**-20.8**	**-21.8**	**-21.1**	**-20.7**	**-22.4**	....
W05	**-31.1**	**-31.4**	**-31.4**	**-31.2**	**-31.5**	**-31.3**	**-31.4**	**-31.2**	....
W06	**-47.7**	**-47.6**	**-47.4**	**-47.4**	**-47.5**	**-47.3**	**-47.4**	**-47.4**	....
W07	**-35.3**	**-35.6**	**-35.6**	**-35.3**	**-35.4**	**-35.4**	**-35.2**	**-35.5**	....
W08	**-49.3**	**-49.4**	**-49.0**	**-49.0**	**-49.1**	**-49.1**	**-49.0**	**-49.0**	....
W09	**-37.2**	**-37.5**	**-37.4**	**-37.2**	**-37.4**	**-37.3**	**-37.2**	**-37.6**	....
W10	**-48.7**	**-47.9**	**-48.6**	**-48.3**	**-48.5**	**-48.6**	**-48.4**	**-48.5**	....
W11	**-38.6**	**-38.7**	**-38.8**	**-38.6**	**-38.8**	**-38.6**	**-38.6**	**-38.7**	....
W12	**-49.2**	**-48.9**	**-49.1**	**-49.0**	**-48.9**	**-48.8**	**-49.2**	**-49.3**	....
W13	**-40.0**	**-39.9**	**-49.0**	**-39.9**	**-39.9**	**-39.8**	**-39.8**	**-40.0**	....
W14	**-49.0**	**-48.6**	**-48.7**	**-48.9**	**-48.9**	**-48.9**	**-48.5**	**-49.2**	....
W15	**-40.7**	**-40.8**	**-40.6**	**-40.7**	**-40.6**	**-40.7**	**-40.6**	**-40.6**	....
W16	**-49.0**	**-49.1**	**-48.8**	**-48.9**	**-49.0**	**-48.9**	**-48.8**	**-49.0**	....
W17	**-41.5**	**-41.6**	**-41.6**	**-41.6**	**-41.5**	**-41.5**	**-41.5**	**-41.4**	....
W18	**-48.9**	**-48.6**	**-49.0**	**-48.8**	**-48.8**	**-48.9**	**-48.8**	**-48.9**	....
W19	**-41.9**	**-42.1**	**-42.1**	**-41.8**	**-41.9**	**-42.0**	**-41.9**	**-41.9**	....
W20	**-48.6**	**-48.7**	**-48.5**	**-48.8**	**-48.7**	**-48.8**	**-48.6**	**-48.4**	....
W21	**-42.8**	**-42.6**	**-42.7**	**-42.6**	**-42.5**	**-42.6**	**-42.6**	**-42.6**	....
W22	**-49.1**	**-48.9**	**-48.9**	**-49.0**	**-48.8**	**-48.9**	**-49.0**	**-48.9**	....
W23	**-43.0**	**-43.0**	**-43.0**	**-43.0**	**-43.0**	**-43.1**	**-43.0**	**-43.1**	....
W24	**-49.1**	**-48.9**	**-48.9**	**-49.0**	**-48.8**	**-48.9**	**-49.0**	**-48.9**	....
W25	**-43.2**	**-43.2**	**-43.2**	**-43.2**	**-43.2**	**-43.1**	**-43.2**	**-43.0**	....
W26	**-49.6**	**-49.7**	**-49.4**	**-49.6**	**-49.5**	**-49.4**	**-49.5**	**-49.5**	....
W27	**-43.4**	**-43.5**	**-43.3**	**-43.4**	**-43.3**	**-43.4**	**-43.3**	**-43.2**	....
W28	**-49.6**	**-49.6**	**-49.4**	**-49.8**	**-49.7**	**-49.5**	**-49.6**	/////	....
W29	**-43.8**	**-43.6**	**-43.6**	**-43.6**	**-43.6**	**-43.7**	/////	/////	....
W30	**-49.6**	**-49.8**	**-49.7**	**-49.6**	**-49.5**	**-49.5**	/////	/////	....
W31	**-44.0**	**-44.0**	**-43.8**	**-43.9**	**-43.9**	**-43.8**	/////	/////	....
W32	**-49.7**	**-49.9**	**-49.8**	**-49.6**	**-49.8**	**-49.7**	/////	/////	....

Table 8.44: Throughputs - PIC (fma.rm.f32 , n , write-read) - 1 SM - CG - Part 1

If nvcc generates the ELF kernels necessary to execute the PTX kernels generated for the PTX instruction configuration (fma.rm.f32 , n , write-read), but we use dependence distances greater than 46, then the ELF kernels incur spill load and spill store phenomena during their executions. This is because each thread can have no more than 64 hardware registers. Each kernel produced for the PTX instruction configuration (fma.rm.f32 , n , write-read) requires 18 registers for the executions of instructions that are different from the PTX instruction configurations (fma.rm.f32 , n , write-read) and that are inside the for loop. Therefore, only 64 - 18 = 46 registers can be used for the execution of the PTX instruction configurations (fma.rm.fs32 , n , write-read) that are inside the for loop (1 register per instruction configuration). Note the sharp decrease of the throughputs for dependence distances greater than 46 (e.g. dependence distances between 51 and 86).

	DD45	DD46	DD47	DD48	DD49	DD50	DD51	DD52	DD53	DD54	DD55	DD56	DD57	DD58
W01	3.42	3.41	3.41	3.42	3.41	3.42	3.41	3.41	3.40	3.40	3.39	3.39	3.38	3.37
W02	6.87	6.86	6.86	6.86	6.86	6.86	6.86	6.86	6.83	6.83	6.81	6.80	6.79	6.77
W03	10.30	10.28	10.29	10.29	10.29	10.29	10.28	10.29	10.25	10.25	10.22	10.20	10.18	10.15
W04	13.74	13.71	13.71	13.72	13.72	13.72	13.71	13.72	13.67	13.66	13.63	13.60	13.58	13.54
W05	17.13	17.08	17.09	17.09	17.09	17.10	17.13	17.09	17.02	17.02	16.98	16.94	16.91	16.86
W06	20.34	20.30	20.30	20.31	20.30	20.31	20.29	20.30	20.23	20.22	20.17	20.14	20.09	20.03
W07	23.64	23.59	23.60	23.61	23.60	23.62	23.59	23.60	23.52	23.51	23.45	23.41	23.36	23.29
W08	26.74	26.70	26.71	26.71	26.71	26.71	26.69	26.69	26.61	26.59	26.54	26.49	26.44	26.35
W09	28.53	28.73	28.74	28.66	28.64	28.72	28.55	28.73	28.67	28.66	28.62	28.56	28.54	28.46
W10	31.44	31.78	31.78	31.42	31.64	31.79	31.73	31.76	31.66	31.65	31.60	31.53	31.46	31.35
W11	28.93	28.93	28.93	28.91	28.94	29.05	28.87	28.80	28.91	28.85	28.85	29.00	28.83	28.67
W12	31.84	31.86	31.84	31.51	31.82	31.80	31.80	31.70	31.80	31.71	31.66	31.60	31.52	31.52
W13	29.47	29.55	29.52	29.44	29.48	29.38	29.37	29.43	29.41	29.43	29.29	29.31	29.30	29.14
W14	31.85	31.87	31.87	31.83	31.83	31.84	31.85	31.83	31.80	31.78	31.71	31.63	31.56	31.49
W15	29.90	29.89	29.89	29.58	29.85	29.78	29.91	29.79	29.82	29.80	29.76	29.70	29.64	29.54
W16	31.88	31.88	31.88	31.88	31.86	31.85	31.85	31.80	31.79	31.77	31.72	31.66	31.58	31.47

Table 8.45: Throughputs - PIC (fma.rm.f32 , n , write-read) - 1 SM - CG - Part 2

If nvcc generates the ELF kernels necessary to execute the PTX kernels generated for the PTX instruction configuration (fma.rm.f32 , n , write-read), but we use dependence distances greater than 46, then the ELF kernels incur spill load and spill store phenomena during their executions. This is because each thread can have no more than 64 hardware registers. Each kernel produced for the PTX instruction configuration (fma.rm.f32 , n , write-read) requires 18 registers for the executions of instructions that are different from the PTX instruction configurations (fma.rm.f32 , n , write-read) and that are inside the for loop. Therefore, only 64 - 18 = 46 registers can be used for the execution of the PTX instruction configurations (fma.rm.fs32 , n , write-read) that are inside the for loop (1 register per instruction configuration). Note the sharp decrease of the throughputs for dependence distances greater than 46 (e.g. dependence distances between 51 and 86).

	DD59	DD60	DD61	DD62	DD63	DD64	DD65	DD66	DD67	DD68	DD69	DD70	DD71	DD72
W01	3.37	3.34	3.29	3.26	3.02	3.00	2.86	2.75	2.74	2.64	2.52	2.48	2.41	2.28
W02	6.77	6.71	6.60	6.54	6.06	6.03	5.74	5.50	5.50	5.28	5.06	4.97	4.83	4.57
W03	10.15	10.06	9.91	9.81	9.09	9.01	8.61	8.18	8.25	7.93	7.53	7.44	6.89	6.47
W04	13.54	13.41	13.21	13.08	12.11	11.88	11.36	10.77	10.94	10.47	9.87	9.76	9.02	8.13
W05	16.87	16.71	16.45	16.29	15.10	14.88	14.15	13.45	13.59	12.94	11.23	10.33	8.98	8.11
W06	20.04	19.85	19.55	19.35	17.73	17.61	16.72	15.86	16.05	15.07	12.22	10.96	9.32	8.19
W07	23.30	23.08	22.73	22.50	20.57	20.34	19.35	18.33	16.55	14.40	12.00	10.80	9.21	8.14
W08	26.37	26.12	25.71	25.47	23.03	22.40	21.50	20.36	17.98	15.25	12.36	11.00	9.25	8.19
W09	28.35	28.10	27.55	27.23	23.01	22.10	20.56	19.46	17.19	14.83	12.17	10.80	9.20	8.16
W10	31.39	31.09	30.60	30.27	25.76	24.80	23.26	21.75	18.23	15.36	12.39	10.91	9.24	8.26
W11	28.53	28.25	27.51	26.49	23.02	22.05	21.11	20.15	17.45	15.03	12.27	10.86	9.29	8.38
W12	31.41	31.17	30.39	29.63	25.65	25.01	23.48	22.26	18.29	15.48	12.47	11.00	9.46	8.52
W13	29.06	28.69	27.75	26.76	23.49	23.05	21.71	20.75	17.58	15.22	12.40	11.01	9.54	8.67
W14	31.50	31.21	30.41	29.71	25.92	25.30	23.80	22.63	18.38	15.62	12.68	11.23	9.72	8.83
W15	29.50	29.23	28.60	27.52	24.18	23.63	22.19	21.28	17.77	15.41	12.67	11.31	9.85	9.02
W16	31.46	31.17	30.74	30.10	26.42	25.77	24.19	22.99	18.49	15.81	12.88	11.50	10.07	9.23

Table 8.46: Throughputs - PIC (fma.rm.f32 , n , write-read) - 1 SM - CG - Part 3

If nvcc generates the ELF kernels necessary to execute the PTX kernels generated for the PTX instruction configuration (fma.rm.f32 , n , write-read), but we use dependence distances greater than 46, then the ELF kernels incur spill load and spill store phenomena during their executions. This is because each thread can have no more than 64 hardware registers. Each kernel produced for the PTX instruction configuration (fma.rm.f32 , n , write-read) requires 18 registers for the executions of instructions that are different from the PTX instruction configurations (fma.rm.f32 , n , write-read) and that are inside the for loop. Therefore, only 64 - 18 = 46 registers can be used for the execution of the PTX instruction configurations (fma.rm.fs32 , n , write-read) that are inside the for loop (1 register per instruction configuration). Note the sharp decrease of the throughputs for dependence distances greater than 46 (e.g. dependence distances between 51 and 86).

	DD73	DD74	DD75	DD76	DD77	DD78	DD79	DD80	DD81	DD82	DD83	DD84	DD85	DD86
W01	2.06	1.95	1.84	1.74	1.65	1.58	1.53	1.49	1.42	1.38	1.34	1.30	1.24	1.22
W02	4.13	3.88	3.69	3.47	3.31	3.16	3.06	2.94	2.84	2.75	2.65	2.58	2.46	2.41
W03	5.80	5.28	5.05	4.77	4.35	4.11	3.98	3.88	3.65	3.50	3.33	3.21	3.00	2.93
W04	7.04	6.08	5.75	5.29	4.67	4.27	4.11	3.96	3.63	3.40	3.12	2.98	2.77	2.63
W05	7.18	6.13	5.65	5.17	4.50	4.08	3.89	3.68	3.36	3.14	2.87	2.73	2.53	2.39
W06	7.10	6.06	5.51	4.98	4.35	3.95	3.74	3.52	3.23	3.04	2.82	2.68	2.52	2.39
W07	6.99	6.02	5.45	4.91	4.33	3.96	3.74	3.53	3.27	3.11	2.92	2.78	2.63	2.50
W08	6.96	6.04	5.48	4.94	4.40	4.06	3.85	3.66	3.42	3.26	3.08	2.93	2.79	2.66
W09	6.96	6.11	5.57	5.03	4.54	4.21	4.01	3.83	3.59	3.43	3.25	3.11	2.96	2.83
W10	7.03	6.23	5.72	5.18	4.71	4.39	4.21	4.03	3.77	3.62	3.43	3.28	3.13	2.99
W11	7.15	6.40	5.88	5.35	4.89	4.58	4.40	4.23	3.95	3.80	3.61	3.45	3.30	3.15
W12	7.30	6.57	6.07	5.54	5.09	4.78	4.60	4.44	4.13	3.97	3.79	3.62	3.46	3.31
W13	7.46	6.75	6.26	5.71	5.27	4.97	4.75	4.59	4.30	4.14	3.96	3.78	3.63	3.48
W14	7.63	6.96	6.45	5.91	5.46	5.15	4.94	4.75	4.49	4.33	4.14	3.97	3.82	3.67
W15	7.81	7.15	6.65	6.10	5.66	5.34	5.13	4.94	4.68	4.53	4.33	4.16	4.02	3.85
W16	8.03	7.38	6.87	6.31	5.88	5.55	5.34	5.15	4.89	4.73	4.54	4.37	4.22	4.06

Table 8.47: Throughputs - PIC (fma.rm.f32 , n , write-read) - 1 SM - CG - Part 4

If nvcc generates the ELF kernels necessary to execute the PTX kernels generated for the PTX instruction configuration (fma.rm.f32 , n , write-read), but we use dependence distances greater than 46, then the ELF kernels incur spill load and spill store phenomena during their executions. This is because each thread can have no more than 64 hardware registers. Each kernel produced for the PTX instruction configuration (fma.rm.f32 , n , write-read) requires 18 registers for the executions of instructions that are different from the PTX instruction configurations (fma.rm.f32 , n , write-read) and that are inside the for loop. Therefore, only 64 - 18 = 46 registers can be used for the execution of the PTX instruction configurations (fma.rm.fs32 , n , write-read) that are inside the for loop (1 register per instruction configuration). Note the sharp decrease of the throughputs for dependence distances greater than 46 (e.g. dependence distances between 51 and 86).

	DD87	DD88	DD89	DD90	DD91	DD92	DD93	DD94	DD95	DD96	DD97	DD98	DD99	DD100
W01	1.19	1.16	1.13	1.10	1.07	1.05	1.03	1.00	0.98	0.96	0.94	0.93	0.92	0.90
W02	2.35	2.30	2.22	2.14	2.10	2.09	2.05	1.99	1.95	1.91	1.88	1.85	1.83	1.79
W03	2.97	2.69	2.52	2.39	2.28	2.75	2.50	2.00	1.69	1.49	1.34	1.25	1.17	1.09
W04	2.67	2.34	2.18	2.06	1.93	1.72	1.61	1.45	1.35	1.28	1.21	1.16	1.12	1.08
W05	2.33	2.15	2.01	1.93	1.84	1.75	1.68	1.53	1.45	1.38	1.31	1.27	1.23	1.20
W06	2.31	2.18	2.06	2.00	1.92	1.88	1.82	1.67	1.58	1.52	1.45	1.40	1.36	1.32
W07	2.42	2.31	2.19	2.14	2.06	2.04	1.97	1.82	1.73	1.66	1.60	1.56	1.53	1.49
W08	2.57	2.47	2.35	2.29	2.22	2.21	2.14	1.98	1.90	1.84	1.78	1.74	1.71	1.67
W09	2.73	2.63	2.51	2.45	2.37	2.37	2.29	2.14	2.07	2.01	1.96	1.92	1.88	1.85
W10	2.89	2.79	2.66	2.60	2.52	2.52	2.44	2.30	2.24	2.18	2.12	2.09	2.05	2.01
W11	3.04	2.94	2.81	2.74	2.66	2.65	2.58	2.45	2.38	2.34	2.28	2.23	2.20	2.16
W12	3.20	3.10	2.97	2.91	2.82	2.81	2.74	2.62	2.56	2.51	2.45	2.41	2.36	2.32
W13	3.37	3.27	3.14	3.08	2.99	2.98	2.91	2.79	2.73	2.68	2.62	2.57	2.53	2.49
W14	3.56	3.46	3.33	3.27	3.17	3.17	3.10	2.98	2.92	2.86	2.80	2.75	2.71	2.66
W15	3.75	3.65	3.53	3.46	3.35	3.36	3.29	3.17	3.10	3.04	2.97	2.93	2.88	2.83
W16	3.95	3.84	3.72	3.65	3.53	3.56	3.50	3.36	3.29	3.23	3.16	3.11	3.06	3.01

Table 8.48: Throughputs - PIC (fma.rm.f32 , n , write-read) - 1 SM - CG - Part 5

If nvcc generates the ELF kernels necessary to execute the PTX kernels generated for the PTX instruction configuration (fma.rm.f32 , n , write-read), but we use dependence distances greater than 46, then the ELF kernels incur spill load and spill store phenomena during their executions. This is because each thread can have no more than 64 hardware registers. Each kernel produced for the PTX instruction configuration (fma.rm.f32 , n , write-read) requires 18 registers for the executions of instructions that are different from the PTX instruction configurations (fma.rm.f32 , n , write-read) and that are inside the for loop. Therefore, only 64 - 18 = 46 registers can be used for the execution of the PTX instruction configurations (fma.rm.fs32 , n , write-read) that are inside the for loop (1 register per instruction configuration). Note the sharp decrease of the throughputs for dependence distances greater than 46 (e.g. dependence distances between 51 and 86).

	DD101	DD102	DD103	DD104	DD105	DD106	DD107	DD108	DD109	DD110	DD111	DD112
W01	0.88	0.87	0.86	0.85	0.84	0.83	0.82	0.80	0.80	0.79	0.78	0.77
W02	1.76	1.73	1.72	1.70	1.68	1.62	1.59	1.52	1.50	1.48	1.46	1.41
W03	1.03	0.98	0.94	0.90	0.88	0.85	0.83	0.81	0.79	0.78	0.77	0.77
W04	1.05	1.02	0.99	0.98	0.96	0.94	0.93	0.91	0.90	0.89	0.87	0.86
W05	1.16	1.13	1.11	1.09	1.08	1.06	1.05	1.03	1.02	1.01	0.99	0.98
W06	1.29	1.26	1.23	1.22	1.21	1.19	1.18	1.16	1.15	1.14	1.13	1.12
W07	1.45	1.43	1.40	1.39	1.38	1.36	1.35	1.33	1.32	1.30	1.28	1.27
W08	1.64	1.61	1.59	1.57	1.55	1.53	1.52	1.50	1.48	1.46	1.44	1.43
W09	1.82	1.79	1.76	1.74	1.72	1.69	1.68	1.65	1.63	1.61	1.59	1.57
W10	1.97	1.94	1.91	1.88	1.87	1.83	1.82	1.78	1.76	1.75	1.72	1.70
W11	2.12	2.08	2.06	2.03	2.00	1.97	1.95	1.92	1.90	1.88	1.86	1.84
W12	2.28	2.25	2.22	2.19	2.16	2.13	2.11	2.07	2.05	2.03	2.01	1.98
W13	2.44	2.41	2.38	2.34	2.32	2.27	2.26	2.22	2.19	2.18	2.15	2.13
W14	2.61	2.58	2.54	2.50	2.48	2.43	2.41	2.37	2.34	2.32	2.30	2.27
W15	2.78	2.74	2.70	2.66	2.64	2.59	2.56	2.52	2.49	2.47	2.45	2.42
W16	2.96	2.91	2.88	2.83	2.80	2.75	2.72	2.68	2.64	2.62	2.60	2.56

Table 8.49: Throughputs - PIC (fma.rm.f32 , n , write-read) - 1 SM - CG - Part 6

If nvcc generates the ELF kernels necessary to execute the PTX kernels generated for the PTX instruction configuration (fma.rm.f32 , n , write-read), but we use dependence distances greater than 46, then the ELF kernels incur spill load and spill store phenomena during their executions. This is because each thread can have no more than 64 hardware registers. Each kernel produced for the PTX instruction configuration (fma.rm.f32 , n , write-read) requires 18 registers for the executions of instructions that are different from the PTX instruction configurations (fma.rm.f32 , n , write-read) and that are inside the for loop. Therefore, only 64 - 18 = 46 registers can be used for the execution of the PTX instruction configurations (fma.rm.fs32 , n , write-read) that are inside the for loop (1 register per instruction configuration). Note the sharp decrease of the throughputs for dependence distances greater than 46 (e.g. dependence distances between 51 and 86).

	DD113	DD114	DD115	DD116	DD117	DD118	DD119	DD120	DD121	DD122	DD123	DD124
W01	0.77	0.76	0.75	0.75	0.74	0.73	0.73	0.72	0.72	0.71	0.71	0.70
W02	1.33	1.32	1.29	1.29	1.27	1.27	1.26	1.26	1.24	1.03	0.78	0.62
W03	0.78	0.76	0.77	0.77	0.77	0.78	0.78	0.78	0.78	0.74	0.68	0.67
W04	0.85	0.84	0.84	0.83	0.83	0.84	0.85	0.86	0.87	0.84	0.78	0.79
W05	0.97	0.96	0.95	0.94	0.94	0.94	0.95	0.96	0.97	0.96	0.91	0.90
W06	1.10	1.09	1.08	1.07	1.07	1.06	1.06	1.08	1.08	1.09	1.04	1.03
W07	1.25	1.24	1.23	1.22	1.21	1.20	1.20	1.20	1.21	1.22	1.18	1.17
W08	1.41	1.40	1.39	1.38	1.37	1.36	1.34	1.34	1.34	1.35	1.33	1.31
W09	1.56	1.54	1.53	1.52	1.51	1.50	1.48	1.47	1.46	1.46	1.44	1.43
W10	1.68	1.66	1.66	1.65	1.64	1.62	1.60	1.59	1.57	1.56	1.55	1.54
W11	1.82	1.80	1.79	1.77	1.76	1.74	1.72	1.71	1.70	1.68	1.67	1.66
W12	1.96	1.95	1.93	1.91	1.90	1.88	1.86	1.85	1.83	1.81	1.80	1.80
W13	2.10	2.08	2.07	2.05	2.04	2.02	2.00	1.98	1.96	1.94	1.94	1.93
W14	2.25	2.22	2.21	2.20	2.18	2.16	2.13	2.12	2.10	2.08	2.07	2.07
W15	2.39	2.37	2.35	2.33	2.32	2.29	2.27	2.25	2.23	2.21	2.20	2.20
W16	2.53	2.51	2.50	2.47	2.46	2.44	2.41	2.39	2.36	2.34	2.34	2.34

Table 8.50: Codes' efficiencies - PIC (add.s32 , n , write-read) - 14 SMs - WG - MK

The code efficiencies of the modified ELF kernels that we generated for the PTX instruction configuration (add.s32 , n , write-read) are shown below. The kernels' executions cannot be slowed down by the memories' bandwidths and latencies. The rows of the table represent the number of warps of each thread block, while the columns of the table represent the dependence distances (there is one ELF kernel per dependence distance). Note that we execute the ELF kernels using 14 thread blocks (equal to the number of streaming multiprocessors), and that a Tesla C2070 can execute 448 ($14 \cdot 32$) PTX add.s32.n instructions per functional unit clock cycle. For each original ELF kernel (there is one kernel per dependence distance) we generate 24 new ELF kernels (one per number of warps per thread block) using procedure 8.18. These new ELF kernels guarantee that, when they are executed, the gigathread scheduler will always fairly assign the thread blocks to the streaming multiprocessors.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	DD09	DD10	DD11	DD12	DD13	DD14	DD15
W09	44.6	82.1	87.2	87.1	87.3	87.3	87.4	87.4	87.3	87.4	87.4	87.3	87.4	87.4	....
W10	49.3	87.9	98.7	98.6	98.3	98.7	97.6	98.4	98.6	97.8	98.6	98.4	98.6	98.7	....
W11	54.5	88.5	88.9	88.7	88.8	88.8	88.8	88.8	88.7	88.8	88.9	88.8	88.9	88.8	....
W12	59.4	97.7	97.7	99.2	98.0	98.6	99.3	99.3	99.2	98.9	99.2	99.1	99.0	98.9	....
W13	64.1	89.8	90.0	90.0	90.0	90.0	90.0	90.0	90.1	90.0	90.1	90.1	90.0	90.0	....
W14	69.0	97.9	98.1	99.7	98.5	99.0	98.5	98.6	99.0	98.7	98.9	99.0	99.1	99.0	....
W15	72.9	90.7	90.7	90.8	90.8	90.8	90.8	90.8	90.8	90.9	90.8	90.9	90.8	90.9	....
W16	77.6	99.3	98.6	99.9	99.3	98.0	98.7	98.9	99.0	99.1	98.9	99.0	99.1	99.0	....
W17	79.7	91.5	91.7	91.5	91.6	91.6	91.7	91.7	91.6	91.7	91.8	91.8	91.7	91.7	....
W18	84.3	97.7	97.7	100	98.5	98.0	98.3	98.7	99.0	98.7	99.1	98.9	98.9	99.0	....
W19	86.1	91.9	91.9	92.0	92.2	92.1	92.1	92.1	92.0	92.2	92.3	92.0	92.1	92.2	....
W20	90.4	98.0	98.7	99.8	98.3	98.3	98.6	98.7	98.6	98.7	98.6	98.9	98.8	98.9	....
W21	85.8	92.3	92.4	92.6	92.7	92.7	92.7	92.7	92.9	92.8	92.9	92.8	92.7	92.8	....
W22	90.0	98.9	98.6	99.9	99.3	98.6	99.3	99.2	99.1	98.9	99.0	99.1	98.9	99.1	....
W23	91.5	92.8	92.8	93.1	93.1	93.1	93.1	93.2	93.1	93.2	93.2	93.2	93.2	93.3	....
W24	97.1	97.7	99.3	99.6	99.0	99.3	98.9	99.0	99.1	98.9	99.0	99.1	99.2	98.9	....
W25	87.4	93.1	93.2	93.4	93.3	93.4	93.3	93.4	93.3	93.4	93.5	93.4	93.4	93.3	....
W26	97.3	98.3	99.9	99.9	99.2	99.9	99.6	99.7	99.6	99.7	99.5	99.7	99.6	99.6	....
W27	90.9	93.3	93.5	93.5	93.6	93.6	93.7	93.6	93.5	93.7	93.5	93.6	93.5	93.6	....
W28	97.5	98.5	99.9	99.9	99.3	99.9	99.3	99.7	99.6	99.7	99.5	99.9	99.8	99.7	....
W29	89.0	93.5	93.8	93.8	93.8	93.9	93.9	93.8	93.9	93.8	93.8	93.8	93.8	93.9	....
W30	98.5	98.3	99.9	99.9	99.0	99.8	99.7	99.7	99.6	99.9	99.8	99.7	99.6	99.7	....
W31	94.3	93.9	93.8	93.8	94.0	94.0	94.0	94.0	94.1	94.2	94.0	94.1	94.1	94.0	....
W32	99.2	98.9	99.9	99.8	99.8	100	99.8	99.9	99.8	100	99.9	99.7	99.8	99.9	....

Table 8.51: Codes' efficiencies - PIC (add.s32 , n , write-read) - 14 SMs - CG

The code efficiencies of the ELF kernels generated by nvcc for the PTX instruction configuration (add.s32 , n , write-read) are shown below. The rows of the table represent the number of warps of each thread block, while the columns of the table represent the dependence distances (there is one ELF kernel per dependence distance). Note that we execute the ELF kernels using 14 thread blocks (equal to the number of streaming multiprocessors) and that a Tesla C2070 can execute 448 ($14 \cdot 32$) PTX add.s32.n instructions per functional unit clock cycle. The low code efficiencies are due to the gigathread scheduler, which does not fairly assign the thread blocks to the streaming multiprocessors, and to the transformations that nvcc applied to the structures of the codes (i.e. number of hardware registers used, dependence distances among hardware registers, type and order of the ELF instructions generated to execute the PTX kernels).

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	DD09	DD10	DD11	DD12	DD13	DD14	DD15
W09	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.1	25.1	25.1	25.1	....
W10	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.1	25.1	....
W11	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.1	25.1	25.1	....
W12	30.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.1	25.1	25.1	25.1	....
W13	32.5	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.1	25.1	25.1	25.1	....
W14	35.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.1	25.1	25.1	25.1	25.1	....
W15	37.5	25.0	25.0	25.0	25.0	25.0	25.1	25.0	25.0	25.0	25.1	25.1	25.1	25.1	....
W16	40.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.1	25.1	25.1	25.1	....
W17	42.5	25.0	25.0	25.0	25.0	25.0	25.1	25.0	25.0	25.0	25.1	25.1	25.1	25.1	....
W18	45.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0	25.1	25.1	25.1	25.1	....
W19	47.5	26.4	26.4	26.4	26.4	26.4	25.5	26.4	26.5	26.5	26.5	26.5	26.5	50.2	....
W20	49.9	27.8	27.8	27.8	27.8	27.8	26.2	27.8	27.8	28.0	27.9	50.0	50.1	50.1	....
W21	45.0	29.2	29.2	29.2	29.2	29.2	27.1	29.2	29.2	29.2	29.2	50.2	50.2	50.2	....
W22	47.2	30.5	30.6	30.6	30.6	30.6	28.3	30.6	30.6	50.0	50.0	50.1	50.1	50.1	....
W23	47.9	31.9	31.9	32.0	32.0	29.4	32.0	32.0	50.2	50.2	50.2	50.2	50.2	50.2	....
W24	50.0	33.3	33.3	33.3	33.3	33.3	30.7	33.4	50.0	50.0	50.1	50.1	50.1	50.1	....
W25	49.1	50.0	50.0	50.0	50.0	50.0	50.1	50.1	50.1	50.1	50.2	50.2	50.2	50.2	....
W26	50.0	50.0	55.5	50.0	50.0	50.0	50.1	50.0	50.0	50.0	50.1	50.1	50.1	50.1	....
W27	50.0	50.0	50.0	50.0	50.0	50.1	50.1	50.1	50.1	50.2	50.2	50.2	50.2	50.2	....
W28	50.0	50.0	50.0	55.5	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.1	50.1	50.2	....
W29	49.5	50.0	50.0	50.0	50.0	50.1	50.1	50.1	50.1	50.2	50.2	50.2	50.2	50.2	....
W30	50.0	50.0	50.0	50.0	50.0	50.0	50.1	62.6	50.0	50.1	50.1	50.1	50.2	50.2	....
W31	50.0	50.0	50.0	50.0	50.0	50.1	50.1	50.1	50.1	50.2	50.2	50.2	50.2	50.2	....
W32	50.0	50.2	50.2	50.2	50.0	50.0	50.0	50.1	50.1	50.2	50.1	50.1	50.2	50.2	....

Table 8.52: Efficiency differences - PIC (add.s32 , n , write-read) - 14 SMs

The efficiency differences between the ELF kernels generated by nvcc and the modified ELF kernels that we generated for the PTX instruction configuration (add.s32 , n , write-read) are shown below. The rows of the table represent the number of warps of each thread block, while the columns of the table represent the dependence distances (there is one ELF kernel per dependence distance). Note that we execute the ELF kernels using 14 thread blocks (equal to the number of streaming multiprocessors), that a Tesla C2070 can execute 448 ($14 \cdot 32$) PTX add.s32.n instructions per functional unit clock cycle, and that the gigathread scheduler always fairly distributes the thread blocks to the streaming multiprocessors for the modified ELF kernels.

	DD01	DD02	DD03	DD04	DD05	DD06	DD07	DD08	DD09	DD10	DD11	DD12	DD13	DD14	DD15
W09	**-19.6**	**-57.1**	**-62.2**	**-62.1**	**-62.3**	**-62.3**	**-62.4**	**-62.4**	**-62.3**	**-62.4**	**-62.3**	**-62.1**	**-62.3**	**-62.3**	.....
W10	**-24.3**	**-62.9**	**-73.7**	**-73.6**	**-73.3**	**-73.7**	**-72.6**	**-73.4**	**-73.6**	**-72.8**	**-73.6**	**-73.4**	**-73.5**	**-73.6**	.....
W11	**-29.5**	**-63.5**	**-63.9**	**-63.7**	**-63.8**	**-63.8**	**-63.7**	**-63.8**	**-63.7**	**-63.8**	**-63.9**	**-63.7**	**-63.8**	**-63.7**	.....
W12	**-29.4**	**-72.7**	**-72.7**	**-74.2**	**-73.0**	**-73.6**	**-74.3**	**-74.3**	**-74.2**	**-73.9**	**-74.1**	**-74.0**	**-73.9**	**-73.9**	.....
W13	**-31.6**	**-64.8**	**-65.0**	**-65.0**	**-65.0**	**-65.0**	**-64.9**	**-65.0**	**-65.1**	**-65.0**	**-65.0**	**-65.0**	**-64.9**	**-64.9**	.....
W14	**-34.0**	**-72.9**	**-73.1**	**-74.7**	**-73.5**	**-74.0**	**-73.5**	**-73.6**	**-75.0**	**-73.6**	**-73.8**	**-73.9**	**-74.0**	**-73.9**	.....
W15	**-35.4**	**-65.7**	**-65.7**	**-65.8**	**-65.8**	**-65.8**	**-65.7**	**-65.8**	**-65.8**	**-65.9**	**-65.7**	**-65.8**	**-65.7**	**-65.8**	.....
W16	**-37.6**	**-74.3**	**-73.6**	**-74.9**	**-74.3**	**-73.0**	**-73.7**	**-73.9**	**-74.0**	**-74.1**	**-73.8**	**-73.9**	**-74.0**	**-73.9**	.....
W17	**-37.2**	**-66.5**	**-66.2**	**-66.5**	**-66.6**	**-66.6**	**-66.6**	**-66.7**	**-66.6**	**-66.6**	**-66.7**	**-66.6**	**-66.7**	**-66.6**	.....
W18	**-39.3**	**-72.7**	**-72.7**	**-75.0**	**-73.5**	**-73.0**	**-73.3**	**-73.7**	**-74.0**	**-73.7**	**-74.0**	**-73.8**	**-73.8**	**-73.9**	.....
W19	**-38.6**	**-65.5**	**-65.5**	**-65.6**	**-65.8**	**-67.7**	**-65.7**	**-65.6**	**-65.5**	**-65.7**	**-65.8**	**-65.5**	**-65.7**	**-40.0**	.....
W20	**-40.5**	**-70.2**	**-71.9**	**-72.0**	**-70.5**	**-70.5**	**-72.4**	**-71.9**	**-70.8**	**-70.7**	**-70.7**	**-48.9**	**-48.7**	**-48.8**	.....
W21	**-40.8**	**-63.1**	**-63.2**	**-63.4**	**-63.5**	**-63.5**	**-63.5**	**-65.6**	**-63.5**	**-63.7**	**-63.6**	**-63.7**	**-63.6**	**-63.8**	.....
W22	**-42.8**	**-68.4**	**-68.0**	**-69.3**	**-68.7**	**-68.0**	**-71.0**	**-68.6**	**-68.5**	**-48.9**	**-49.0**	**-48.1**	**-48.8**	**-49.0**	.....
W23	**-43.6**	**-60.9**	**-60.9**	**-61.1**	**-61.1**	**-63.7**	**-61.1**	**-61.2**	**-43.0**	**-42.9**	**-43.0**	**-43.0**	**-43.0**	**-43.1**	.....
W24	**-47.1**	**-64.4**	**-66.0**	**-66.3**	**-65.7**	**-66.0**	**-68.2**	**-65.6**	**-49.1**	**-49.1**	**-48.9**	**-49.0**	**-49.1**	**-48.8**	.....
W25	**-38.3**	**-43.1**	**-43.2**	**-43.4**	**-43.3**	**-43.4**	**-43.2**	**-43.3**	**-43.2**	**-43.2**	**-43.3**	**-43.2**	**-43.2**	**-43.1**	.....
W26	**-47.3**	**-48.3**	**-44.4**	**-49.0**	**-49.2**	**-49.9**	**-49.5**	**-49.7**	**-49.6**	**-49.7**	**-49.5**	**-49.6**	**-49.5**	**-49.4**	.....
W27	**-40.9**	**-43.3**	**-43.5**	**-43.5**	**-43.6**	**-43.5**	**-43.6**	**-43.5**	**-43.4**	**-43.5**	**-43.3**	**-43.4**	**-43.3**	**-43.4**	.....
W28	**-47.5**	**-48.5**	**-49.9**	**-44.4**	**-49.3**	**-49.9**	**-49.3**	**-49.7**	**-49.5**	**-49.7**	**-49.4**	**-49.8**	**-49.7**	**-49.5**	.....
W29	**-39.5**	**-43.5**	**-43.8**	**-43.8**	**-43.8**	**-43.8**	**-43.8**	**-43.8**	**-43.7**	**-43.8**	**-43.6**	**-43.6**	**-43.6**	**-43.7**	.....
W30	**-48.5**	**-48.3**	**-49.9**	**-49.9**	**-49.0**	**-49.8**	**-49.6**	**-37.1**	**-49.6**	**-44.8**	**-49.7**	**-49.6**	**-49.4**	**-49.5**	.....
W31	**-44.3**	**-43.9**	**-43.8**	**-43.8**	**-44.0**	**-43.9**	**-43.9**	**-43.9**	**-43.9**	**-44.0**	**-43.8**	**-43.9**	**-43.9**	**-43.9**	.....
W32	**-49.2**	**-48.7**	**-49.7**	**-49.6**	**-49.8**	**-50.0**	**-49.8**	**-49.8**	**-49.7**	**-49.9**	**-49.8**	**-49.6**	**-49.6**	**-49.7**	.....

8.3 Summary

In this chapter we have presented the results regarding the code efficiencies and the low compiler's efficacy. The most important points to remember are the following:

- Even with the ability to produce ELF kernels, with no knowledge and quantification of the low level architectural features and machine behaviors, it is usually easy to generate (ELF kernel , launch configuration) pairs that lose more than 90% of the total performance (see $Cases_{t1}$ at page 104).

- Even with the ability to produce ELF kernels, with no knowledge and quantification of the low level architectural features and machine behaviors, with a number of warps per thread block greater than 1, and with the use of only one streaming multiprocessor (see $Cases_{t2}$ at page 106), usually more than 59% of the (ELF kernel , launch configuration) pairs generated for an instruction configuration lose more than 6% of the total performance and some have performance losses that are greater than 90% of the total performance.

  This is because with no knowledge and quantification of the low level architectural features and machine behaviors, even with the ability to produce ELF kernels, it is extremely difficult to correctly guess the write-read, the read-read, and the warp latencies of the different instructions. However, these latencies are necessary to drive the design and development of highly efficient codes.

  Furthermore, with no knowledge and quantification of the low level architectural features and machine behaviors, the optimization process becomes very time consuming. This is because the user has to generate many different ELF kernels, and run each of the kernels using many different launch configurations, to discover the more efficient (ELF kernel , launch configuration) pairs.

- If a user can produce ELF kernels, has no knowledge of the low level architectural

features and machine behaviors, and uses a number of thread blocks greater than 1, then it is very easy to lose more than 50% of the total performance (see $Cases_{t3}$ at page 106).

This is because the user should guess or discover the more efficient launch configurations. However, an exhaustive search to discover the more performing launch configurations for a code (the number of possible launch configurations is greater than $2^{30} \cdot 1024$), as happens for $Cases_{t2}$, would be not possible.

Furthermore, during the discovery and quantification of the low level architectural features and machine behaviors, we discovered that the gigathread scheduler does not always fairly assign the thread blocks to the streaming multiprocessors. However, when a thread block is assigned to a streaming multiprocessor, it cannot migrate to another streaming multiprocessor. Therefore, if each thread has to execute an almost equal number of instructions (e.g. the code has no divergences), then a not fair distribution consistently generates performance losses greater than 50% of the total performance.

- If a user can produce ELF kernels, has the quantification of the low level architectural features and machine behaviors, and uses a number of thread blocks greater than 1, then he/she can assign a greater number of hardware registers to each thread to produce the guarantee that the gigathread scheduler will fairly assign the thread blocks to the streaming multiprocessors (see $Cases_{t4}$ at page 107).

  Furthermore, with the quantification of the low level architectural features and machine behaviors, a user understands how to scientifically design and develop ELF kernels to produce a priori guarantees about their performances, and therefore the previous and very time consuming optimization process is eliminated.

- Nvcc does not produce efficient ELF codes. This happens even for the very simple PTX kernels used for the discovery and the quantification of the low level architectural features and machine behaviors.

On a case by case basis, for the same launch configuration, the ELF code produced by nvcc for a PTX kernel is usually 30-50% less efficient than the ELF code produced by direct translation (see $Cases_{t5}$ at 138). This is because nvcc modifies the code structure of the PTX kernel, saves some registers, and applies other transformations.

The results clearly show that either nvcc is not aware of the specific values of the write-read, read-read, and warp latencies of the different instructions, or that nvcc is not able to effectively exploit this knowledge even for the transformation of very simple PTX kernels.

- PTX is only a virtual assembly. With no knowledge and quantification of the low level architectural features and machine behaviors, it is easy to use the wrong number and type of virtual resources for a PTX kernel. If this happens, even for very simple PTX kernels, then nvcc will generate ELF kernels that incur spill load and spill store phenomena during their executions. These phenomena ruin the code efficiencies and can easily generate performance losses greater than 90% of the total performance (see $Cases_{t6}$ at page 138).

- We discovered that nvcc does not know that the gigathread scheduler does not always fairly assign the thread blocks to the streaming multiprocessors. Therefore, it generates ELF kernels that, when run using launch configurations with a number of thread blocks greater than one, can easily be 30-75% less efficient (see $Cases_{t7}$ at 139) than analogous kernels generated by exploiting the insight earned from the reverse engineering, procedure 8.18, and the insight earned from the discovery and quantification of the low level architectural features and machine behaviors.

Chapter 9

Related Work

In this chapter, we divide the research on NVIDIA GPUs into three main branches: 1) research executed on pre-2009 GPU architectures; 2) results that are applicable to any GPU architecture; and 3) research executed on post-2009 GPU architectures. This classification is important because the GF100 architecture, introduced in 2009, and later architectures are very different from previous architectures. Furthermore, all the architectures introduced after 2009 are simply an evolution of the GF100 architecture and therefore maintain and improve on the hardware components without revolutionizing its hardware layout. Finally, we present the importance of our work and explain why it is different from this other researcher.

9.1 The Pre-2009 Era

Before 2009, compiler modifications were possible and Yang et. al [64, 63] implemented some modules at the top of the contemporaneous compilers. The modules checked both coalesced and not-coalesced memory accesses to then modify, if possible, the data layout to make all the accesses coalesced. The compiler also automatically discovered data sharing patterns, executed thread and thread block merges by enhancing data locality, and pre-fetched data for the next iterations.

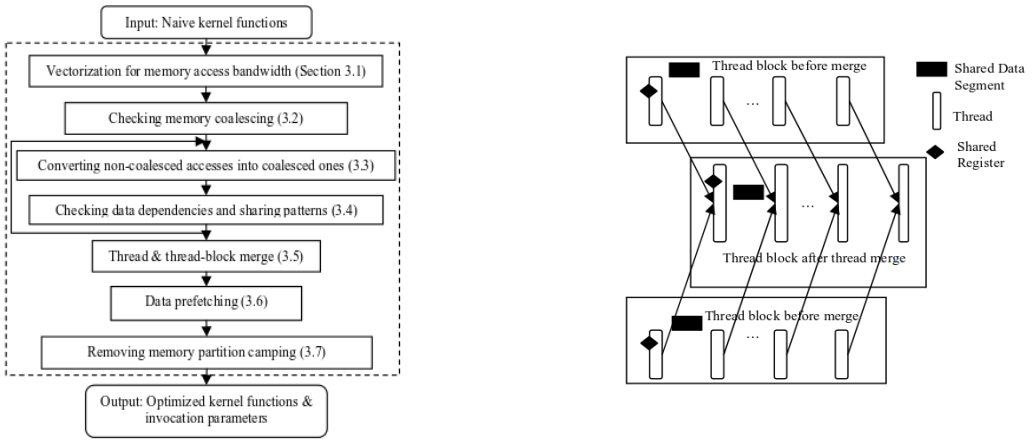

Figure 9.1: The proposed compiler framework is on the left and the improved memory reuse achieved by merging threads is on the right (courtesy of [64, 63]).

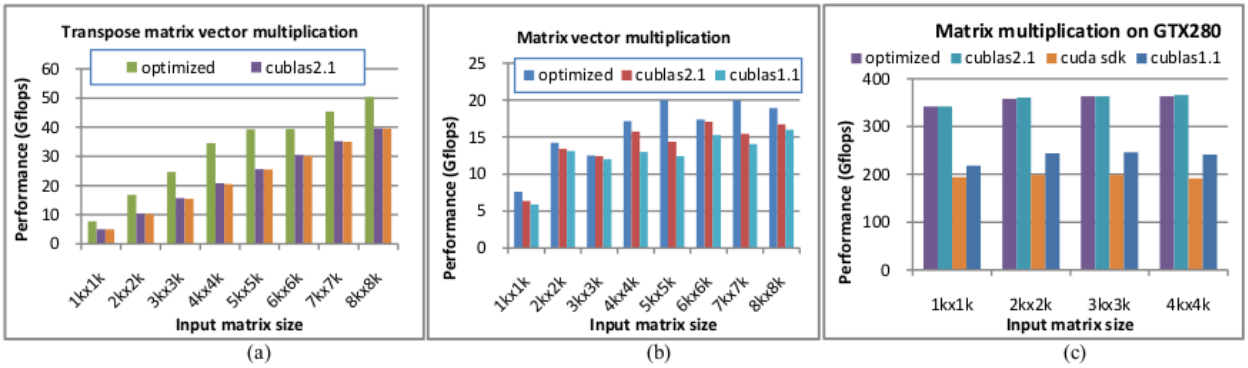

Figure 9.2: Performance comparison for the optimizing compiler (courtesy of [64, 63]).

An interesting GPU power and performance model was developed by Hong and Kim [51]. Their model predicted execution times to determine the GPU power expenditure, and therefore does not require measures of execution times, hardware performance counters, or architectural simulations. The authors in [51] modeled the power consumption coming form the streaming multiprocessors and from the different memories. Furthermore, they parameterized the model, created a temperature model, and modeled the increases in static power consumption. Their model is not accurate for control flow intensive applications, asymmetric applications, and texture cache intensive applications, but produced very accurate predictions for the regular benchmarked applications.

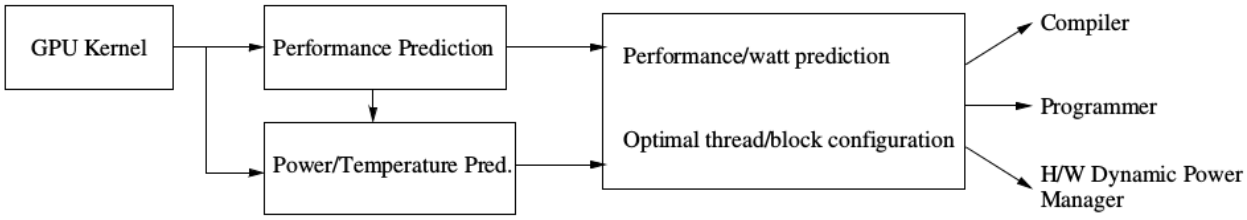

Figure 9.3: Overview of the power performance model (courtesy of [51]).

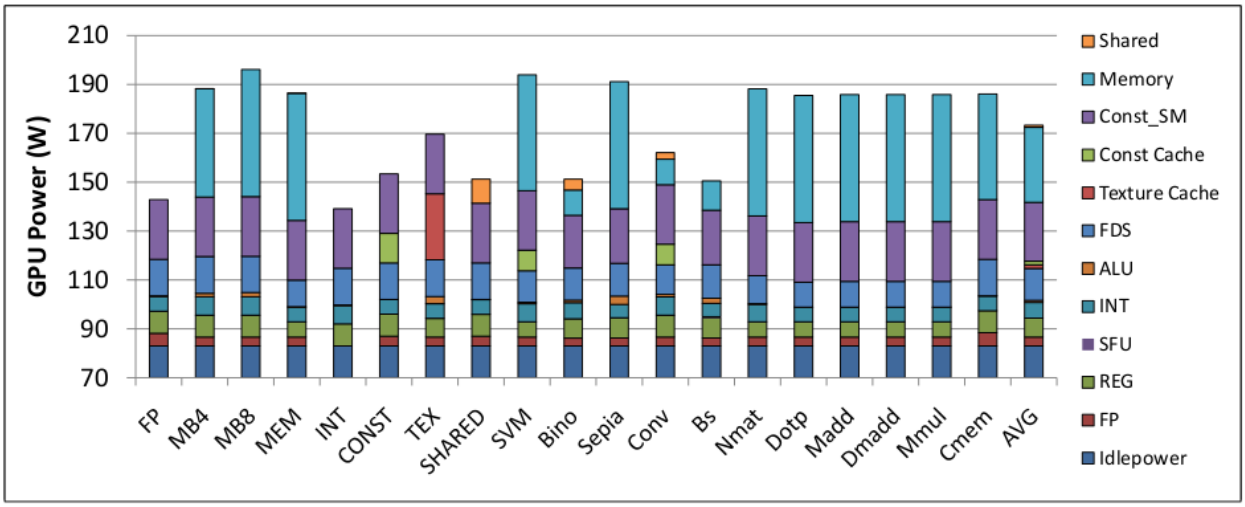

Figure 9.4: Power breakdown graph for all the evaluated benchmarks (courtesy of [51]).

NVIDIA did not disclose many details of its GPUs. Wong et al. [62] therefore demystify some pre-2009 architectural details. In their paper, they measure and quantify various undisclosed features of the processing elements and memory hierarchies of the NVIDIA GPU GTX280. As they correctly stated, the quantification of such undisclosed features is useful to understand how to optimize codes. They studied and quantified a) cache characteristics from latency plots; b) clock overheads and their characteristics; c) the features of the arithmetic pipelines; d) three control flow features (branch divergence, branch reconvergence, and the effects of serialization due to SIMT behaviors); e) different barrier synchronization behaviors and times; f) hardware register features (e.g. size and number of registers); g) shared, texture, and global memory features; h) memory translation features (global and texture); i) constant memory features; and finally l) instruction supply. The authors' job was greatly simplified by decuda (*https://github.com/laanwj/decuda/wiki*). Decuda is a disassembler for fatbin

files or cubin files that provides insight into the instructions that the compiler generates for the pre-2009 G8x and G9x architectures. The authors used decuda to analyze the native instruction sequences generated by nvcc and to analyze the code generated to handle branch divergence and reconvergence.

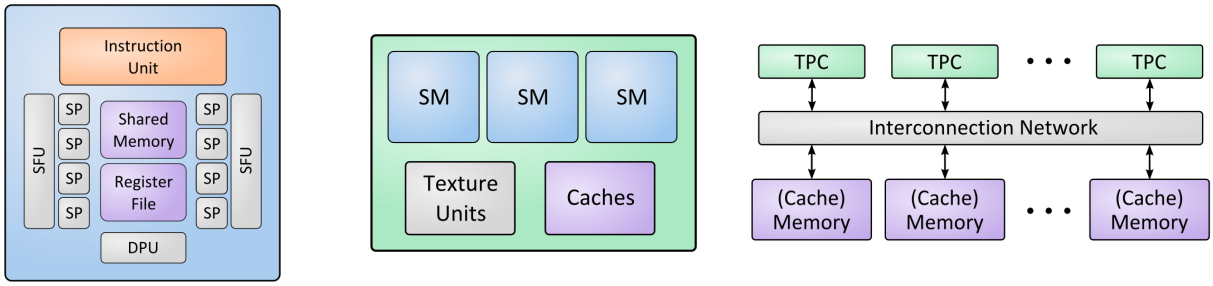

Figure 9.5: The streaming multiprocessor architecture is on the left, the thread processing cluster architecture is in the middle, and the configuration of the memory banks that are interconnected with the thread processing clusters is on the right. These are all hardware components of the NVIDIA GTX 280 architecture (courtesy of [62]).

A good quantitative performance analysis model for pre-2009 GPU architectures was created by Zhang and Owens [66]. Their model identified code bottlenecks and quantitatively analyzed performance. The authors of [66] built their model by implementing and running specific micro-benchmarks for three major GPU components: the instruction pipeline, shared memory access, and global memory access. In the test cases, the model was able to predict code performances with a 5-15% error margin. Furthermore, their model suggested architectural improvements on hardware resource allocations, bank conflicts, block scheduling, and memory transaction granularity.

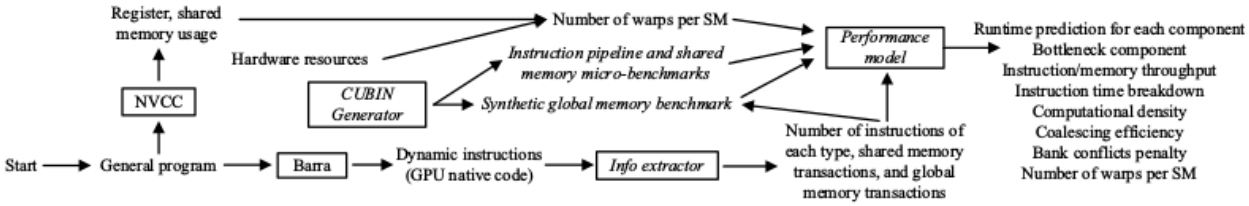

Figure 9.6: Performance modeling workflow. Tools in italics are those developed by Zhang and Owens. While the CUBIN generator produces benchmarks based on GPU native code, the Barra simulator [11] produces the dynamic instruction count for the info extractor component. Note that nvcc is used to discover the number of hardware registers assigned to each thread and to gain information on the shared memory usage (courtesy of [66]).

An interesting aspect of the work executed in [66] is that the model its based on the GPUs native instruction set, instead of the intermediate PTX virtual ISA or a high-level language. In fact, PTX cannot be directly executed by the GPU hardware, but rather has to be transformed by the compiler into a set of native machine instructions. However, the compiler usually introduces significant code changes during the translation and optimization processes.

Compared to Zhang and Owens, Baghsorkhi et al. [2] instead created an adaptive performance modeling tool. First, they created a work flow graph that is an abstract representation of a GPU kernel. Next, using these work flow graphs, they identified the performance bottlenecks and estimate the execution times of each GPU kernel. The performance model was validated using CUDA (Compute Unified Device Architecture), and contrary to previous models, each factor was first measured in isolation, and only later the factors were combined to model the machine. Baghsorkhi et al. therefore claim that "the interactive effects between different performance factors are modeled correctly" (page 2 of [2]).

Finally, for pre-2009 GPU architecture, Hong and Kim [31] developed an analytical model with memory-level and thread-level parallelism awareness. Their model is simple, and its key component estimates "the number of parallel memory requests by considering the number of running threads and memory bandwidth" (page 1 of [31]). Apparently, this is the first analytical model for GPU architectures, and Hong and Kim are the first to propose two metrics for capturing the degree of warp level parallelism that they then effectively used to identify performance bottlenecks.

9.2 General Results

A set of papers develop arguments and study phenomena that are important and valid for any GPU architecture. These papers usually present the performance improvements that could be achieved if the insights that they discovered were to be integrated in future GPU hardware architectures.

Branch divergence can significantly degrade performance. Fung et al. [26] therefore explored mechanisms for more efficient SIMD branch executions. Their results show that regrouping threads on the fly, after the occurrence of one or more divergences, can increase performance "by an average of 20.7% for an estimated area increase of 4.7%" (p. 1 of [26]).

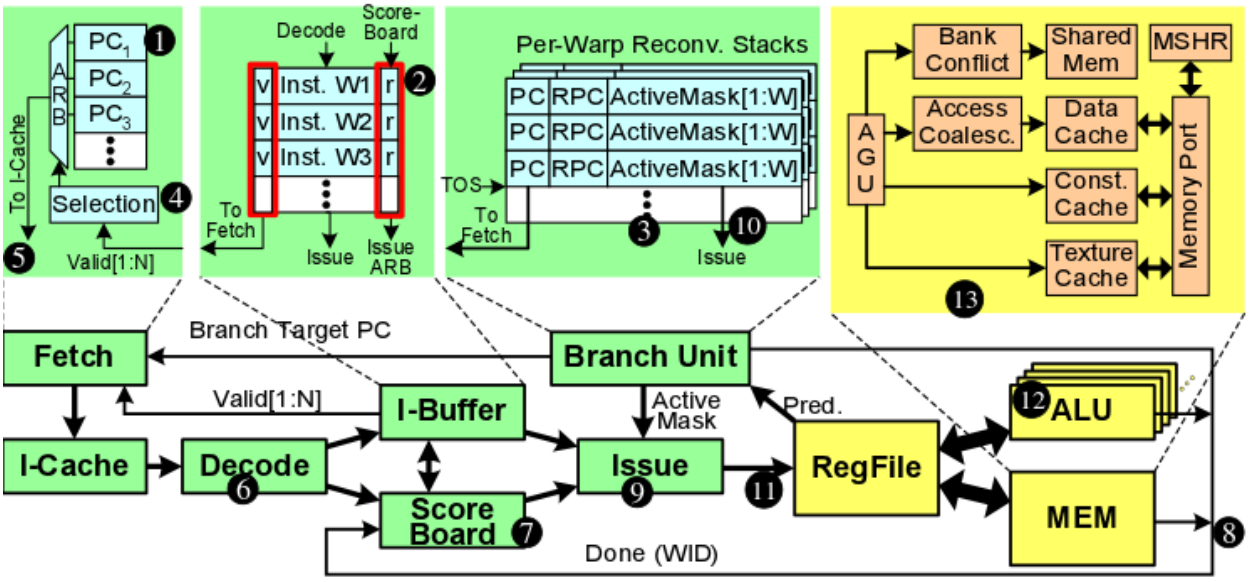

Figure 9.7: SIMT core microarchitecture of a pre-2009 GPU architecture. N equals the number of warps per core and W equals the number of threads per warp (courtesy of [25]).

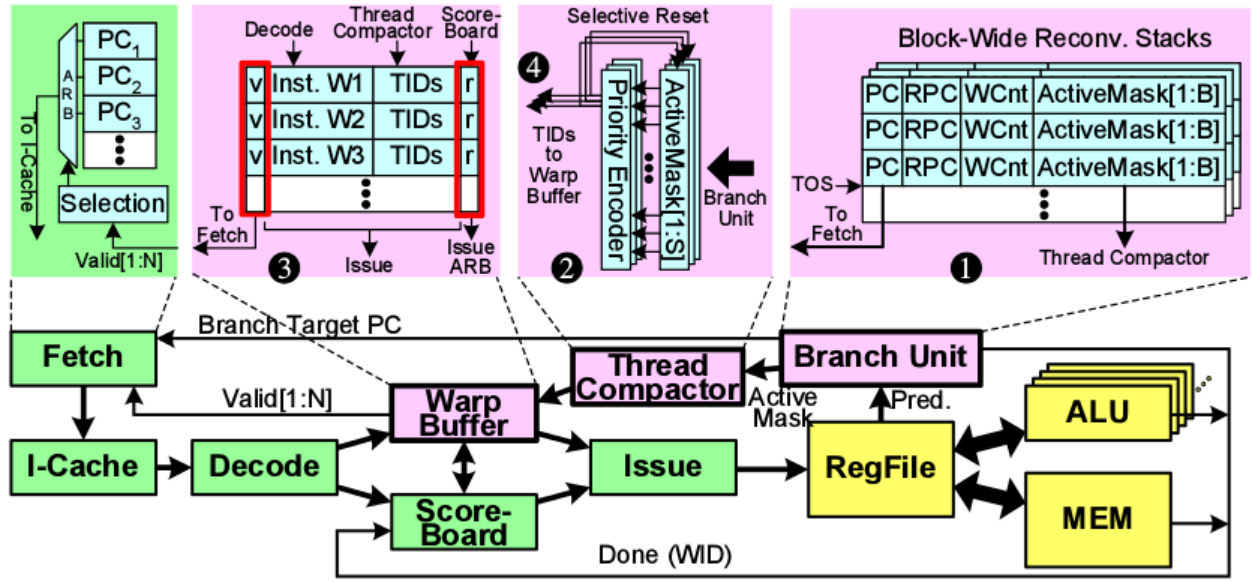

Figure 9.8: Proposed modifications to the SIMT core microarchitecture for the implementation of thread block compaction. N equals the number of warps, B equals the maximum number of threads per warp, W equals the number of threads in a warp, and S equals B divided by W (courtesy of [25]).

Two years later, Fung and Aamodt also demonstrated the importance of thread block compaction for efficient SIMT control flow in [25]. When simulated, Fung and Aamodt reported that their compaction algorithm provided "an average speedup of 22% over a baseline per-warp, stack-based reconvergence mechanism, and 17% versus dynamic warp formation on a set of CUDA applications that suffer significantly from control flow divergence" (page 1 of [25]).

CUDA is the high level tool most commonly used to program NVIDIA GPUs. It is a parallel computing platform and programming model invented by NVIDIA specifically for its GPUs. CUDA works as an extension of the C language, hides from programmers many low level GPU architectural details, and increases code portability. Users can find several books on CUDA. Sanders and Kandrot [50] introduce CUDA using several examples that also use atomic instructions and computational streams. Kirk and Hwu [35] describe several techniques for constructing parallel efficient programs (all their techniques have a computational view of the underlying architecture). Cook [12] describes the usual core concepts (e.g. threads, blocks, grids, and memories), but also illustrates specific CUDA issues on different hardware platforms (i.e. Mac, Linux and Windows with several NVIDIA chipsets). Witt [61] provides a comprehensive guide to GPU programming that continues the work described in [50]. His discussion of hardware and software is more detailed than [50], and describes CUDA 5.0 and the Kepler architectures. Finally, Cheng et al. [9] describe some of the challenges users face when trying to efficiently utilize massive scalar processors at peak performance, and provide instructions on how to profile and tune applications on NVIDIA GPUs.

The works above survey the concept of CUDA transformation recipes. However, Rudy et al. [49] went one step further and develop a programming language interface (CUDA-CHiLL) that allows users to describe and optimize transformations and code generations. The interface also allows users to rapidly prototype compiler algorithms and easily create of different sequences of optimizing transformations. Furthermore, the interface allows users

to separately describe how the compiler has to map codes to architectural features, and therefore it facilitates code porting and code maintenance.

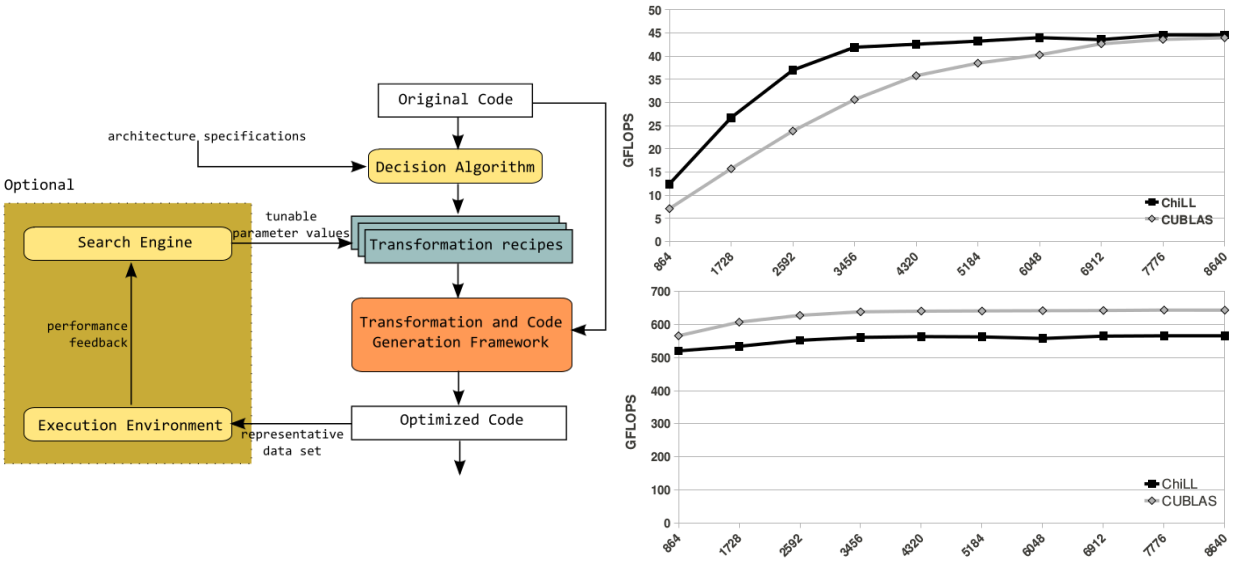

Figure 9.9: The compiler developer workflow is on the left. On the right is the performance comparison between the automatically-generated CUDA-CHiLL code for the single precision matrix-vector (top), and the matrix-matrix multiplication kernels (bottom), both executed on a Tesla C2050 (courtesy of [49]).

Unified Parallel C (UPC) is usually used to achieve high parallel performance computing on large-scale clusters composed of parallel machines. Chen et al. [8] extended UPC to take advantage of the GPUs' computational power. Beyond introducing hierarchical data distribution, they also implemented the necessary compiling system and unified the data management procedure to optimize data transfers and memory layout for CPUs and GPUs. Chen et al. claimed that their UPC extension has better programmability than the mixed MPI/CUDA approach, and that it is effective for achieving good performance on GPU clusters.

Wang and Ranganathan [59] addressed the power consumption problem and developed an instruction-level prediction mechanism to estimate the energy consumption of a program. This estimate was based on the number of active streaming multiprocessors and the PTX code that nvcc produces. Their results show that it is possible to minimize energy consumption without losing much performance. For their benchmarks, Wang and Ranganathan were

able to predict the optimal number of active cores, which therefore produced energy savings ranging from 7.31% to 11.76% and losing only 4.92% of total performance.

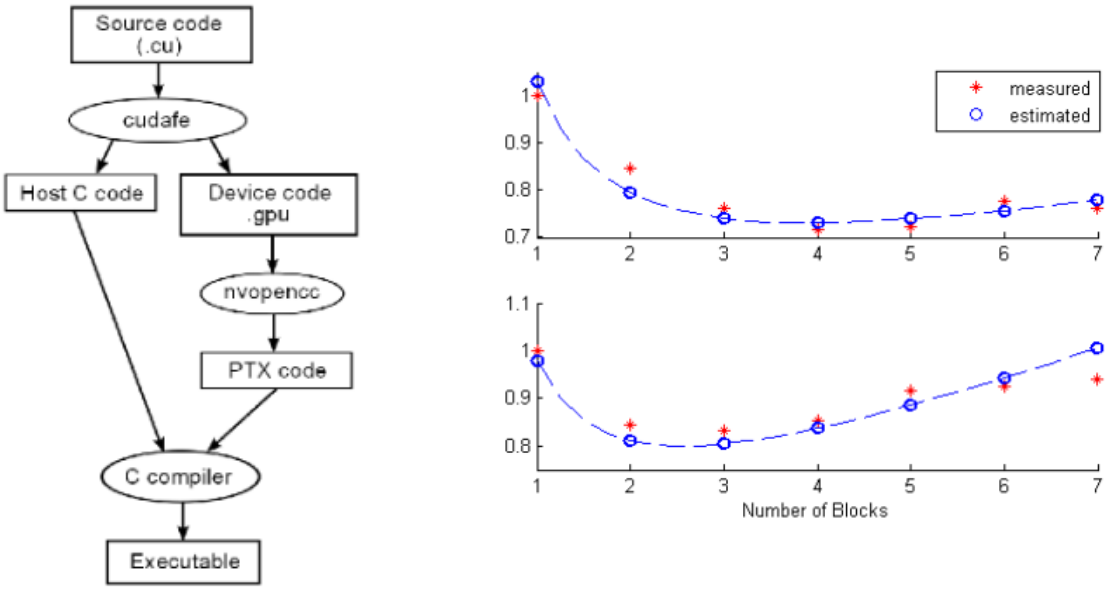

Figure 9.10: The standard CUDA compilation workflow is on the left. The normalized energy consumption for matrix multiplication for two different inputs is on the right (courtesy of [59]).

Finally, Lee and Vetter [37] performed an initial evaluation of directive-based GPU programming models for productive exascale computing, and identified issues "in the functionality, scalability, tunability, and debuggability of the existing models" (page 1 of [37]), which have different features and provide different levels of abstraction for optimizing codes.

Features		PGI	OpenACC	HMPP	OpenMPC	hiCUDA	R-Stream
Code regions to be offloaded		loops	structured blocks	loops	structured blocks	structured blocks	loops
Loop mapping		parallel vector	parallel vector	parallel	parallel	parallel	parallel
Data management	GPU memory allocation and free	explicit implicit	explicit implicit	explicit implicit	explicit implicit	explicit	implicit
	Data movement between CPU and GPU	explicit implicit	explicit implicit	explicit implicit	explicit implicit	explicit	implicit
Compiler optimizations	Loop transformations	implicit	imp-dep	explicit	explicit		implicit
	Data management optimizations	explicit implicit	imp-dep	explicit implicit	explicit implicit	implicit	implicit
GPU-specific features	Thread batching	indirect implicit	indirect implicit	explicit implicit	explicit implicit	explicit	explicit implicit
	Utilization of special memories	indirect implicit	indirect imp-dep	explicit	explicit implicit	explicit	implicit

Figure 9.11: This shows "the type of information that GPU directives can provide. In the table, explicit means that directives exist to control the feature explicitly, implicit indicates that the compiler implicitly handles the feature, indirect shows that users can indirectly control the compiler for the feature, and imp-dep means that the feature is implementation-dependent" (courtesy of [37]).

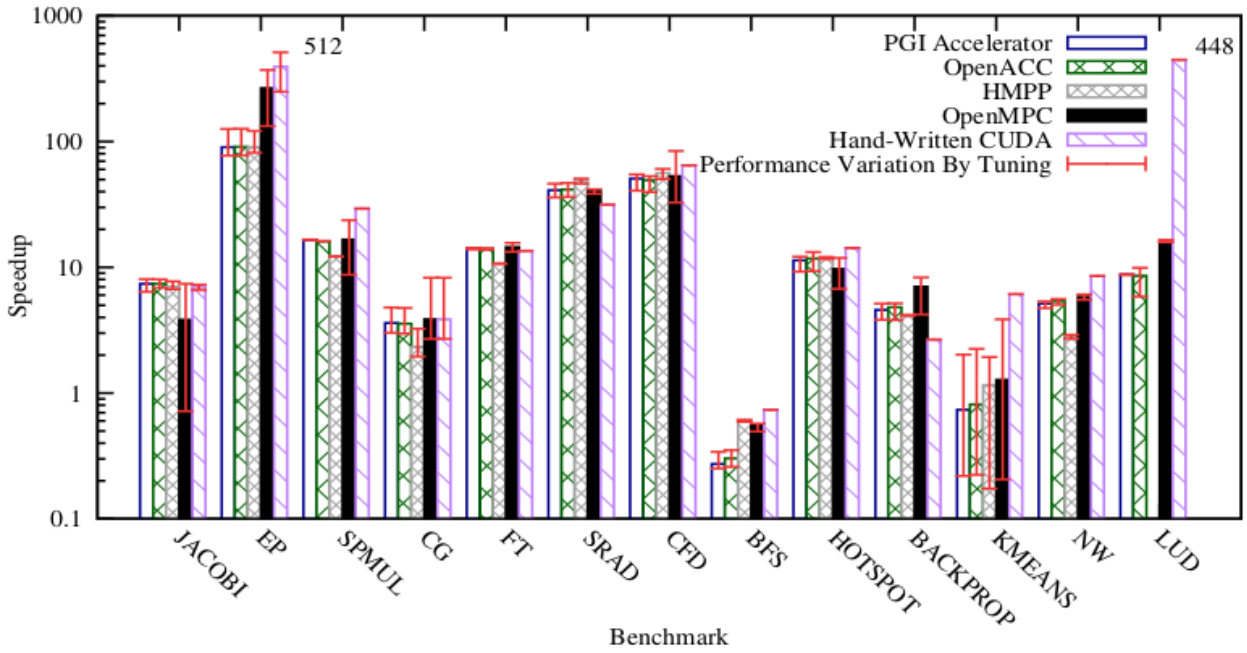

Figure 9.12: Performance of GPU programs when they are translated using directive-based GPU compilers. The speedups in the table are over the serial ported CPU codes and use the largest available input data were used (courtesy of [37]).

9.3 The Post-2009 Era

For NVIDIA architectures post 2009, and therefore for the GF100 and later architectures, we know that the compiler code is closed and that the real ISAs are undisclosed. After 2009, people started to optimize code for the complex NVIDIA GPU architectures by working at source level. However, even do people could modify the compiler and work at assembly level before 2009, people almost always preferred to work at the source level. This was due to the difficulty of working at the assembly level and of identifying and quantifying many NVIDIA GPU low level architectural features and machine behaviors.

Brodtkorb et al. [6] correctly pointed out that optimizing codes so they can fully utilize GPU hardware can easily take months or years. They therefore provided an overview of possible programming strategies and explained possible future trends. They divided the programming strategies into the following categories: 1) guidelines for latency hiding and

thread performance; 2) memory guidelines to avoid bandwidth bottlenecks; and 3) guidelines that consider CPUs and GPUs as different processors that operate asynchronously, but also need to communicate with each other.

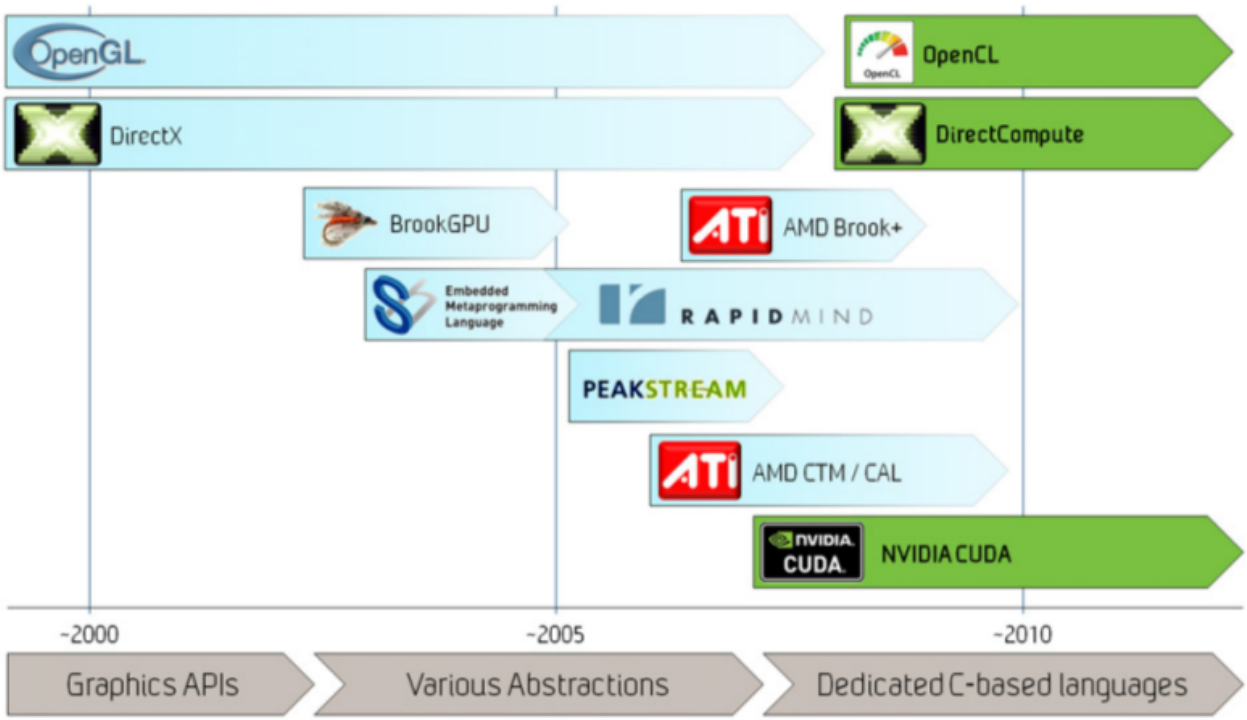

Figure 9.13: History of programming languages for GPU computing (courtesy of [6]).

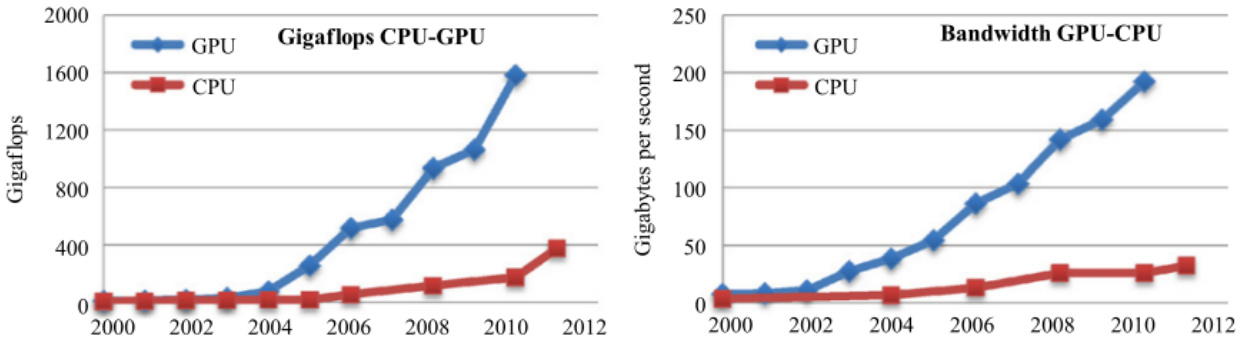

Figure 9.14: Theoretical peak performances and bandwidths (courtesy of [6]).

Baskaran et al. [4] tackled the challenge of transforming C codes into CUDA codes. This is because C codes do not run on NVIDIA GPUs. In their work, sequential C codes were automatically transformed into parallel codes for NVIDIA GPUs. The tools that Baskaran

et al. implemented are important because they relieve users of the burden of managing the GPU memory hierarchy and the parallel interactions among GPU threads, both of which are important for increasing the speed of code development. To accomplish this increase in speed, some transformations satisfy specific tiling conditions while others execute affine transformations attempt to make the parallel loops synchronization-free. Furthermore, Baskaran et al. claimed that the performances of the codes that they generated using the developed tools were close to the performance of hand-optimized CUDA codes.

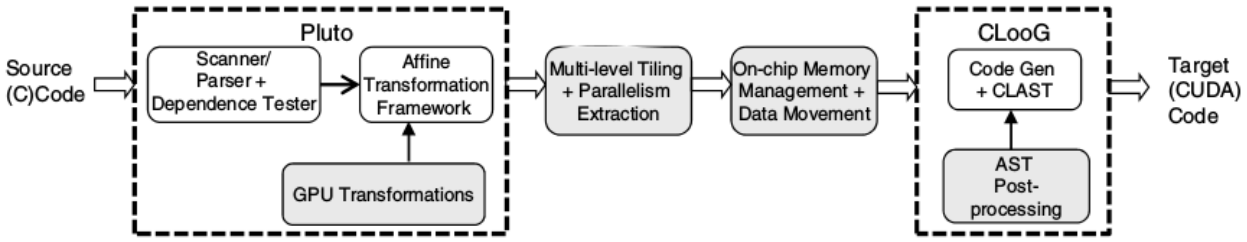

Figure 9.15: The C-to-CUDA code generation framework (courtesy of [4]).

Davidson and Owens [1] introduced several techniques for auto-tuning data-parallel algorithms on GPUs. Their techniques tried to tune algorithms independently of the underlying hardware architecture. These technique: a) identified the tunable parameters of algorithms; b) discovered the relationships between the input and parameter space; and c) modeled and exploited these relationships to facilitate the tuning process along two axes (workload specific and machine specific). While the authors' techniques produced speedups for all the benchmarked cases, the authors also found their dependency strengths.

Table 9.1: Dependency strengths of different algorithms (courtesy of [1]).

Algorithm Name	Device Dependency	Workload Dependency
Reduction	Strong	Weak
Scalar Product	Medium	Strong
N-Body	Weak	Strong
SGEMM	Strong	Medium

A lot of auto-tuning tools have been developed explicitly to accelerate matrix multiplication codes, which are important in many physics application, ranging from dense matrix multiplication with cache exploitation [20], to model-driven auto-tuning tools for sparse matrix-vector multiplication [10]. Some tools auto-tune only CUDA parameters for parametric sparse vector multiplications [30], while others generate Basic Linear Algebra Subprograms (BLAS) for specific GPU architectures [19]. Other tools optimize General Matrix Multiplication (GEMM) kernels and a few specifically optimize GEMM for Fermi GPUs, see [36]. Finally, there are auto-tuning tools that either accelerate only dense vector-vector operations [53], or use ELLR-T approaches to accelerate the sparse matrix-vector product [58].

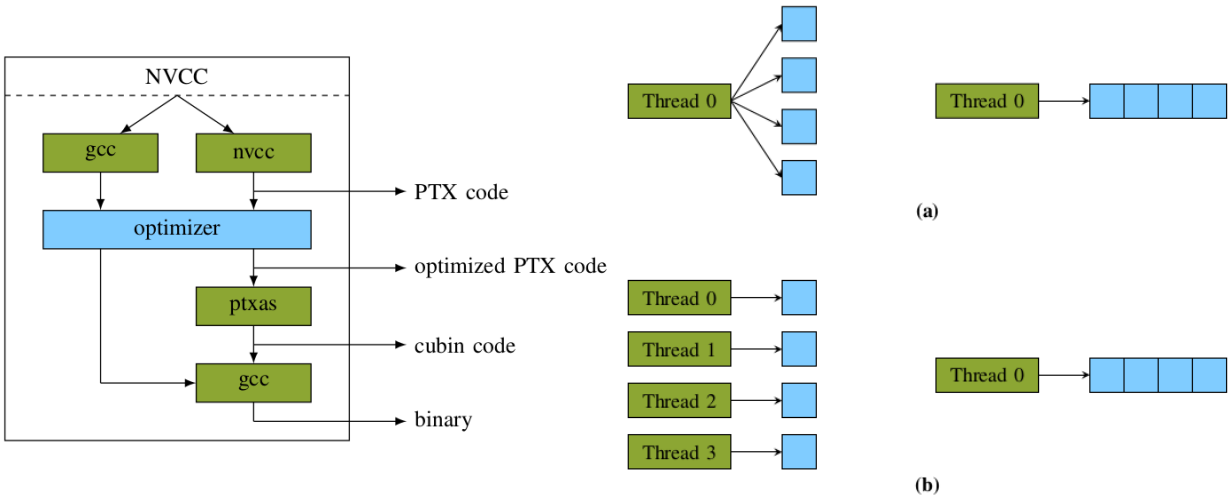

Figure 9.16: The old CUDA compilation trajectory and the optimization tool are on the left. The intra- and inter-thread memory access grouping mechanism are on the right (courtesy of [5]).

Some auto-tuning tools are, in reality, new programming notations. Layout modification usually provides good speedups and so is one of the possible targets of the new programming notations. These notations are used to optimize the compile-time coalescing and the compile-time memory access grouping. The compile time coalescing considers the intra-loop and inter-loop variable swapping. The compile-time memory access grouping consider the intra-thread and inter-thread memory access grouping. By instrumenting the codes using these

programming notations, the compiler can better optimize the codes to produce speedups of up to two orders of magnitude, but this can only happen for very specific codes [5].

Other auto-tuning tools include new programming languages. Some of these languages, like [22], relieve users of the burden of explicitly managing numerous low-level architectural details about the CPU-GPU communications, the synchronizations among GPU threads, and the management of different GPU memories. The authors' goal in [22] was to make GPUs more accessible and to make it easier to exploit their computational power. By introducing Lime and the compiler they developed, the authors enforced isolation and immutability invariants that facilitated the compiler's optimization job. Some of the optimizations that were applied by their compiler include: a) mapping of non-scalar data to different memory structures to better exploit data locality; b) discovering data access patterns; c) if possible, mapping private thread data to fast private memories; d) smart mapping of data into the local shared memories; e) smart access to data that reside in the local shared memories by avoiding bank conflicts that would slow down code executions; and f) possible exploitation of texture memories.

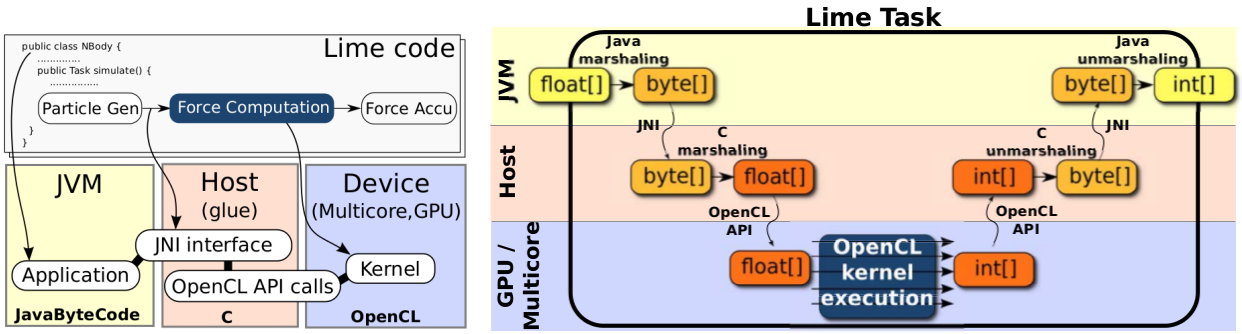

Figure 9.17: On the left, the developed compiler starts from a Lime program and produces three things: 1) the application code that runs in the JVM; 2) the C code that handles data exchanges and the calls to the OpenCL API; and 3) the OpenCL kernel code. On the right is an example of the transformation of an array of floats into an array of integers (courtesy of [22]).

Gray et al. [29] develop several types of optimizations for HMPP. HMPP is a high-level directive-based language and source-to-source compiler that can generate CUDA / OpenCL

code. Gray et al. executed the auto-tuning procedures on the codes on the Poly Bench suite. The auto-tuning procedures mainly focused on loop permutation, loop unrolling, and tiling. The auto-tuned codes are usually significantly faster than the default implementations, and are advantageous even when compared to manually optimized codes.

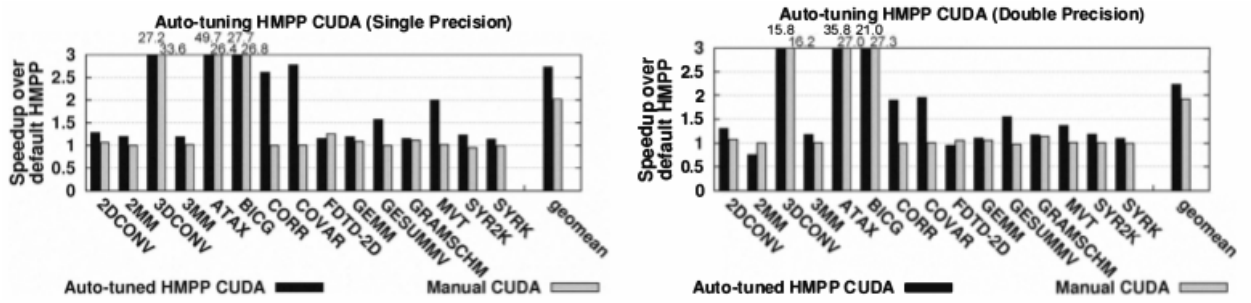

Figure 9.18: Performance comparison of auto-tuned HMPP CUDA (courtesy of [29]).

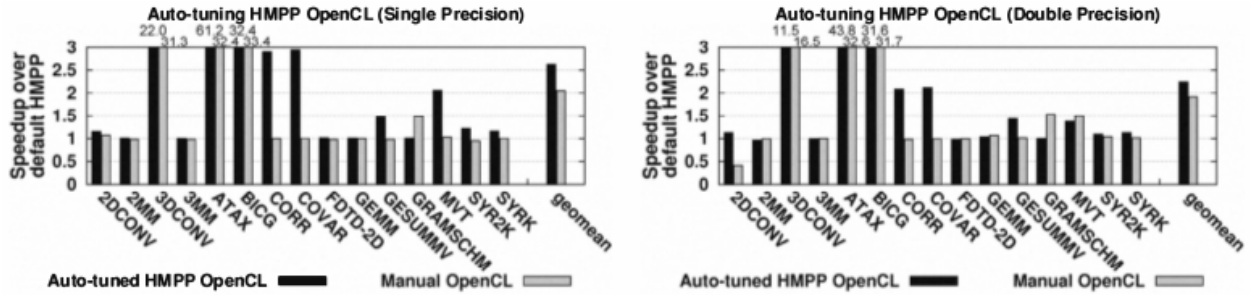

Figure 9.19: Performance comparison of auto-tuned HMPP OpenCL (courtesy of [29]).

Communications between CPUs and GPUs are one of the main culprits behind low performance executions. Some tools, like [33], do not depend on the strength of static compile-time analyses or user-supplied annotations, but are rather a set of compiler transformations and run-time libraries that manage and optimize all CPU-GPU communications. The main contribution of [33] is the CPU-GPU Communication Manager (CGCM). Before the CGCM, users could only use several semi-automatic techniques, but had to insert annotations into codes to manage communications. CGCM provides the first fully automatic CPU-GPU communication management system and is the first fully automatic CPU-GPU communication optimization system.

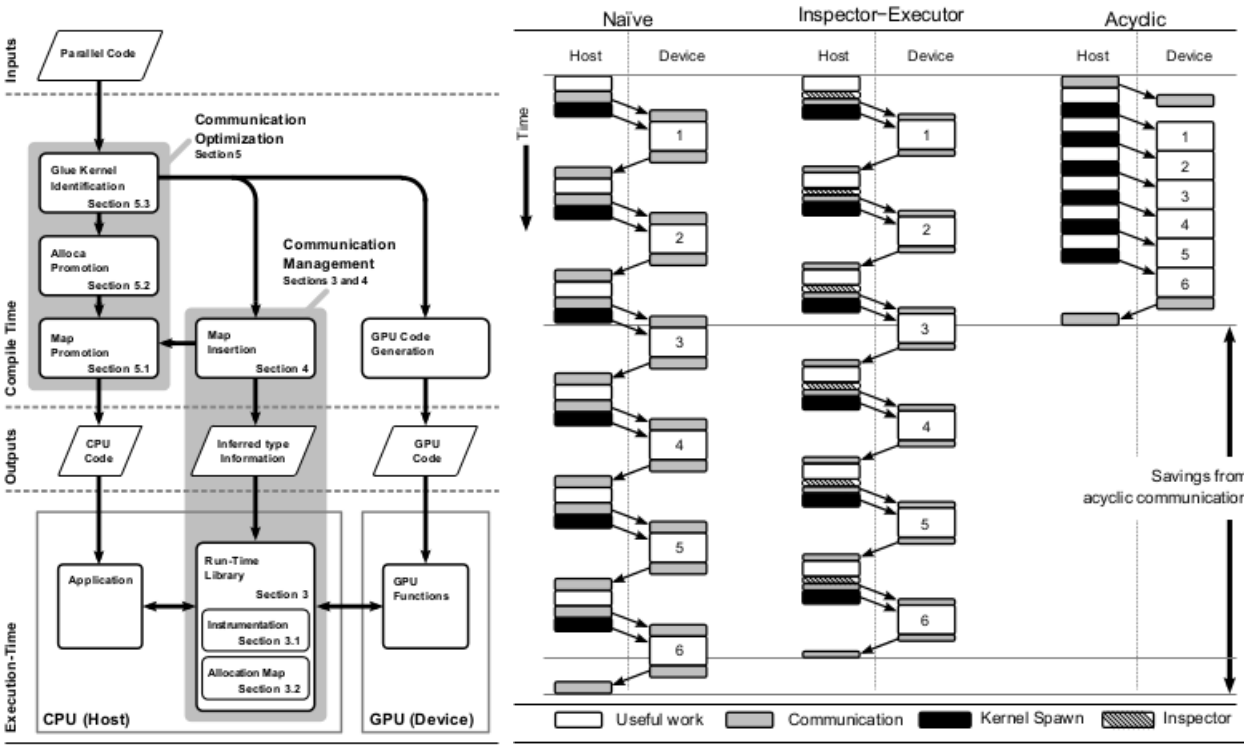

Figure 9.20: On the left is a high level overview of the CPU-GPU Communication Manager. On the right are the execution schedules for several communication patterns (courtesy of [33]).

To further alleviate the productivity bottlenecks in GPU programming, Gray et al. [38] studied the ability of GPU programs to adapt to different data inputs. The G-ADAPT framework was implemented for the study. Given an input, the framework reduces the large optimization space and explores each individual optimization. To improve the framework's efficiency, Gray et al. developed cross-input predictive models that automatically predicted the (near-)optimal configurations for arbitrary inputs to GPU programs. The framework first executed some heuristic-based empirical searches, next executed some pattern recognition procedures, and finally produced the optimized codes.

BENCHMARKS

Benchmark	Description	Num of Inputs	Prediction acc	Training iterations	Training time (s)
convolution	convolution filter of a 2D signal	10	100%	200	2825
matrixMul	dense matrix multiplication	9	100%	196	2539
mvMul	dense matrix-vector multiplication	15	93.3%	124	124
reduction	sum of array	15	80%	75	29
scalarProd	scalar products of vector pairs	7	100%	93	237
transpose	matrix transpose	18	100%	54	1639
transpose-co	matrix transpose with coalescing memory references	18	100%	54	631

Figure 9.21: The G-Adapt framework's accuracy (courtesy of [38]).

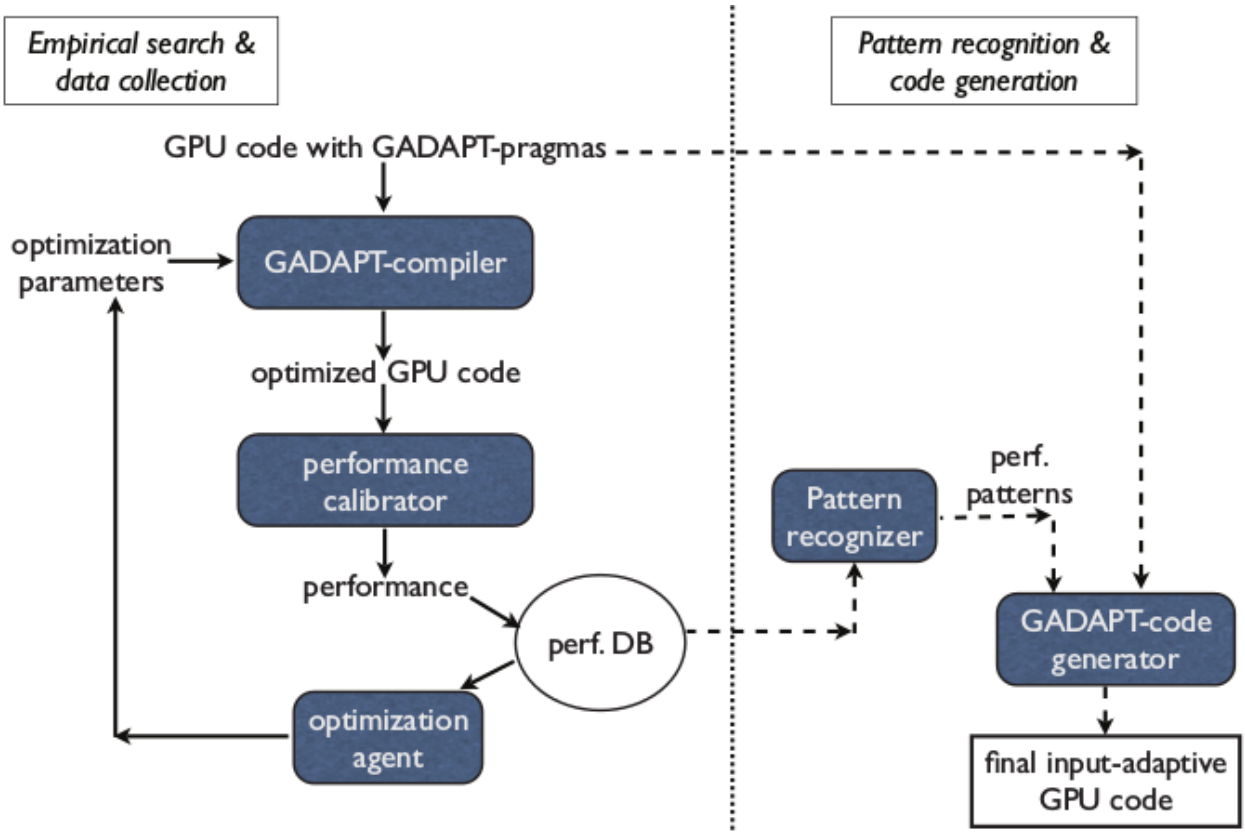

Figure 9.22: The G-Adapt framework's components and work flow (courtesy of [38]).

Control flow in GPU applications is one of the most important optimization techniques. If more GPU threads in a warp follow one or more distinct branches, then the whole GPU application will slow down considerably. Ocelot characterizes and transforms unstructured control flows in GPU applications. More recently, Ocelot has also become a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems [21].

In [7], Burtscher et al. studied the input sensitivity of irregular GPU applications. In these applications, the control flow and memory access patterns are data-dependent and statically unpredictable. To study the input sensitivity of such applications, Burtscher et al. therefore a) defined two measures (the control flow irregularity and the memory access irregularity), and b) used 13 benchmark kernels (Breadth-First Search, Barnes Hut, Data Compression, Delaunay Mesh Refinement, Points-to Analysis, Survey Propagation, Single-Source Shortest Paths, Traveling Salesman Problem, Black-Scholes, Histogram, Monte Carlo,

Matrix Multiplication and N-Body). Burtscher et al. found that the control-flow irregularity and memory-access irregularity were largely independent of each other and that highly tuned implementations exhibited little irregularity.

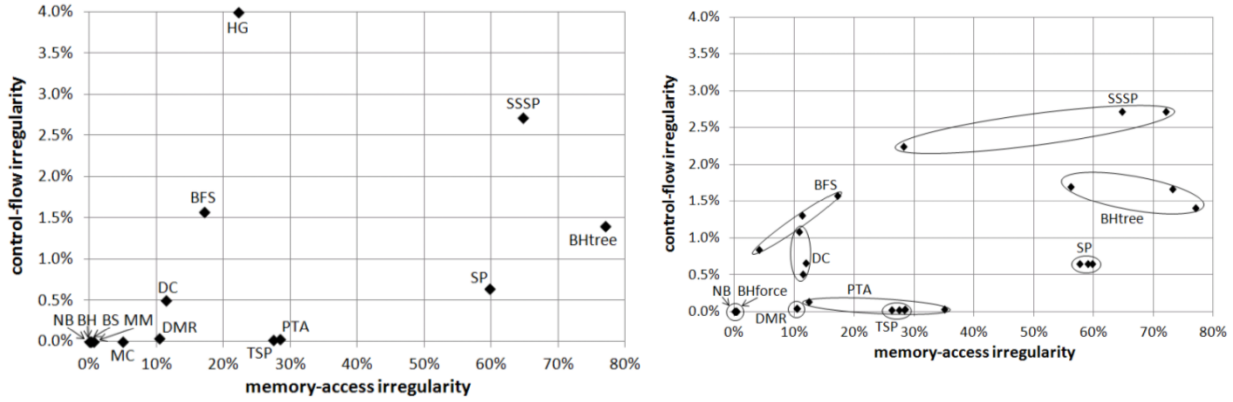

Figure 9.23: On the left is the placement of the kernels in the irregularity space. On the right is the input sensitivity of the various kernels used for the benchmarks (courtesy of [7]).

Pure software solutions like [65] have been developed by Zhang et al. to eliminate dynamic irregularities on the fly. The advantages of these solutions are that they do not require hardware extensions or off-line profiling. The optimization overhead is minimal and does not jeopardize the efficiencies of the base codes.

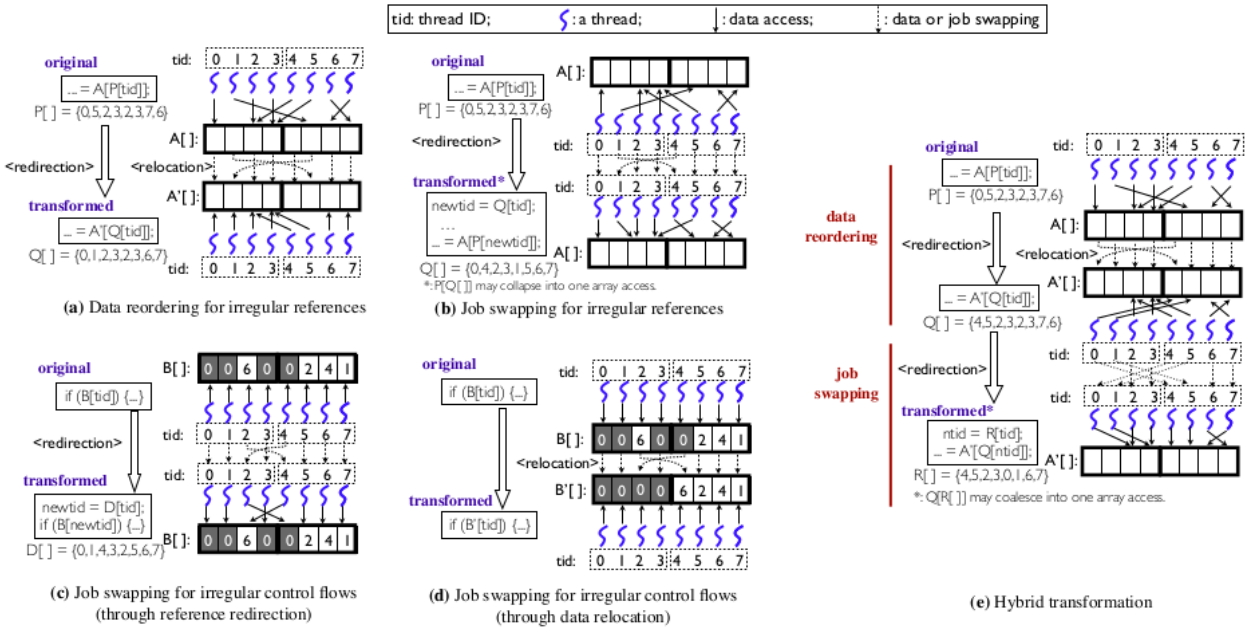

Figure 9.24: Examples that illustrate the data reordering and data swapping procedures, which are necessary to eliminate code memory access irregularities (courtesy of [65]).

Furthermore, the importance of layout modification for producing coalesced accesses, and the importance of workload balancing for code executions, has also been demonstrated for more recent architectures, [54]. These findings remains true even if new architectures are intrinsically different in their number, type and size of different GPU memories.

Some frameworks, such as GROPHECY [41] translate codes. These frameworks usually receive CPU code skeletons as input. Beyond translation, these frameworks can also predict GPU performance. While the effort is noteworthy, different GPUs have different architectures, which are often completely different from each other, and therefore such frameworks are useful only when the relative benefits of translation and optimization make sense. Meng et al. validated GROPHECY using kernel benchmarks and data-parallel codes in legacy scientific applications. They claimed that "the measured performance of manually tuned codes deviates from the projected performance by 17% in geometric mean" (page 1 of [41]).

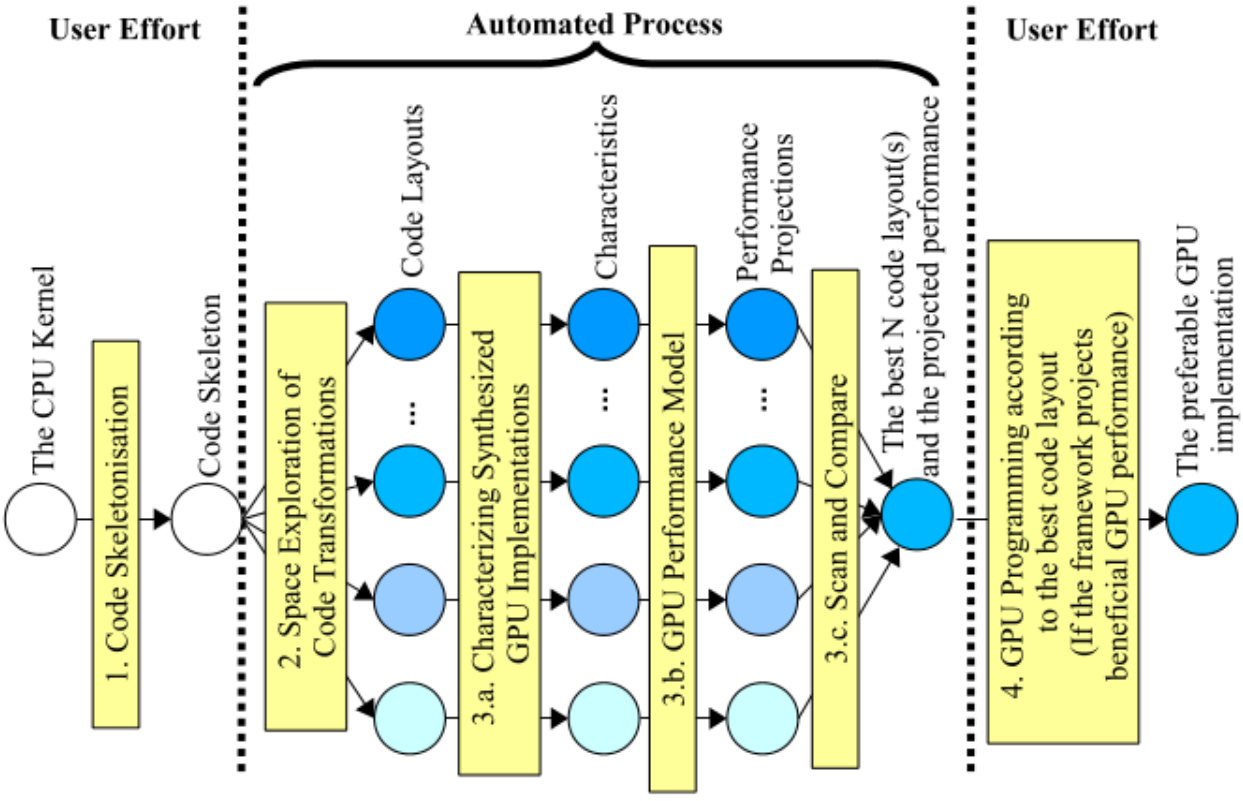

Figure 9.25: An overview of GROPHECY (courtesy of [41]).

Frameworks can also be used for PTX code analysis; see [23]. By instrumenting PTX code, these frameworks simplify such cumbersome and often error-prone user jobs, allow to better characterize different types of workloads, and to discover load unbalancing information that is usually impossible to discover working at the CUDA level.

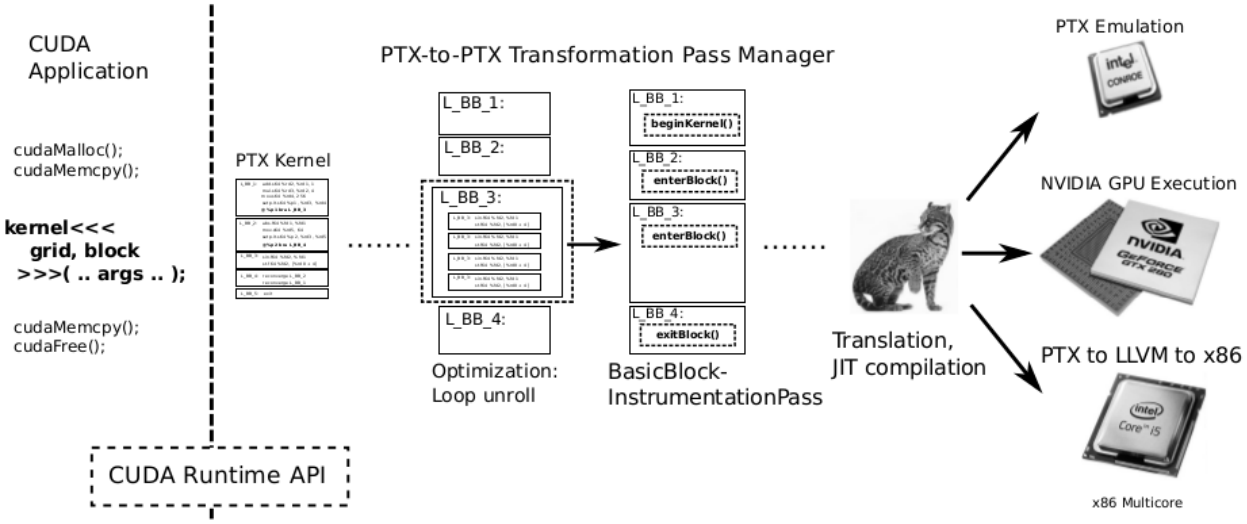

Figure 9.26: The GPU Ocelot dynamic compilation infrastructure (courtesy of [23]).

Some framework / auto-tuning tool hybrids like CUDA-Lite [57] decide the correct memories to use, deal with the memory hierarchy, and code the data transfers among the different memories.

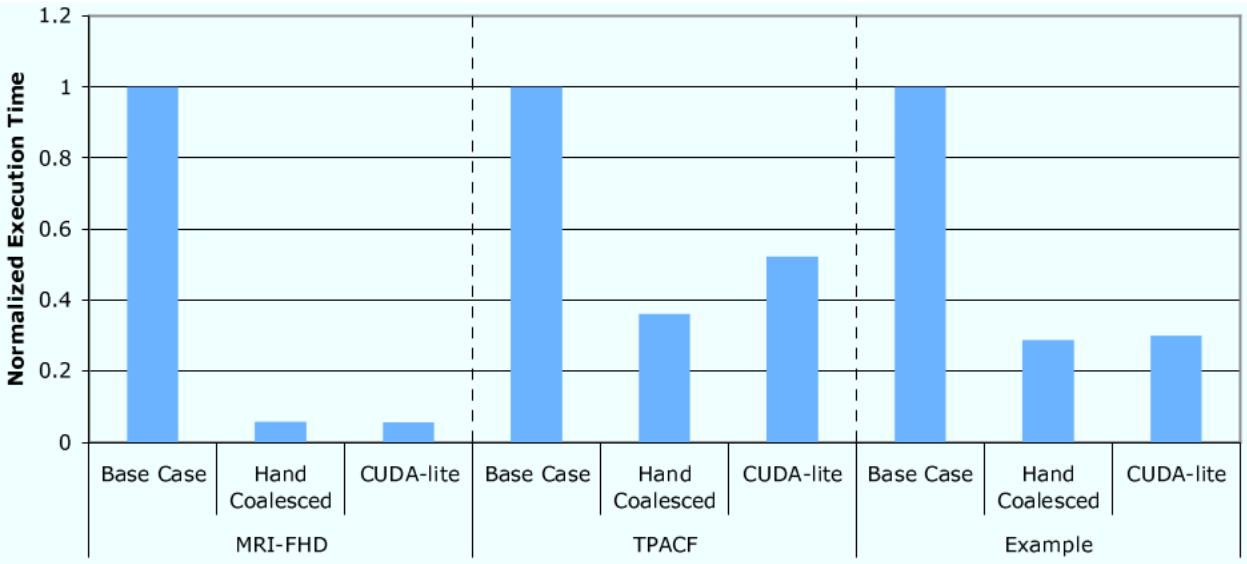

Figure 9.27: Results for the three CUDA Lite benchmarks (courtesy of [57]).

However, Baghsorkhi et al. [3] go further and efficiently evaluate the performance memory hierarchies for highly multithreaded GPUs. To accomplish this, Baghsorkhi et al. exploit a trace-based memory hierarchy model coupled with a Monte Carlo experimental methodology. Their statistical approach overcomes the problem of disturbed execution timing. Using their model, Baghsorkhi et al. optimized the memory accesses in sparse matrix vector multiply kernels and achieved speedups of at least a factor of 2.4 on NVIDIA Tesla C2050 GPUs.

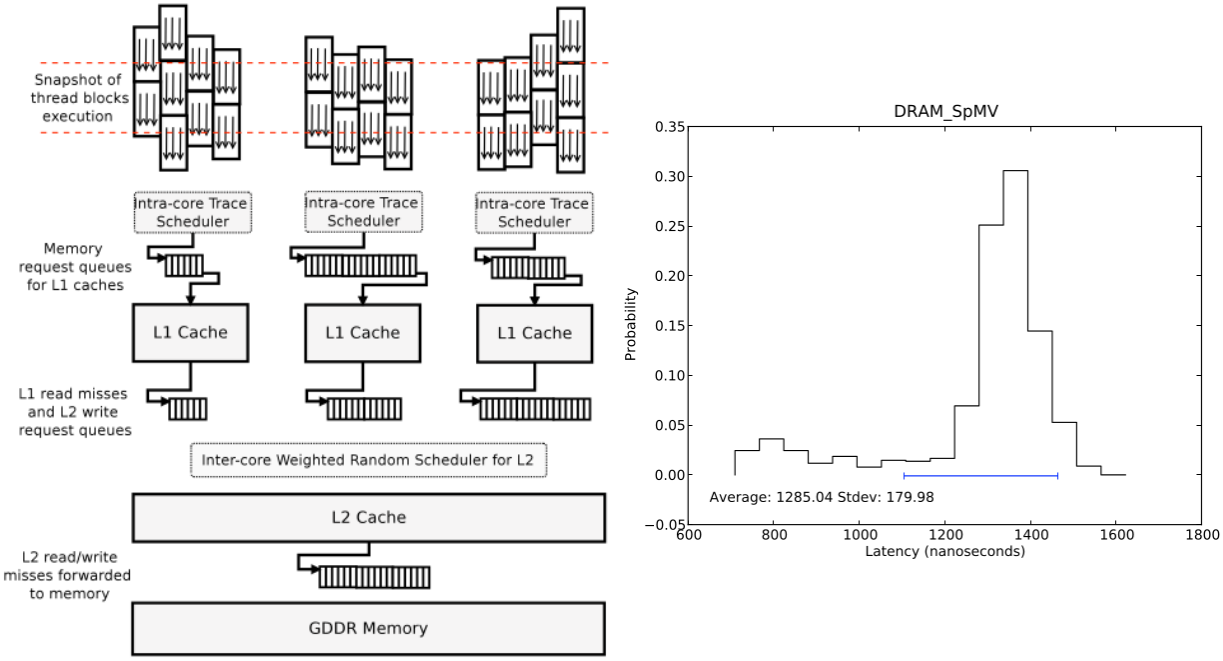

Figure 9.28: The used stochastic memory model is on the left, and on the right is a probability distribution used by Baghsorkhi et at. for the main memory latency (courtesy of [3]).

Torres et al. [55] study the relations among size and shape of thread blocks used in the launch configurations, stream multiprocessor occupancy, and global memory access patterns. These relations are important dimensional combinations for code optimization. Torres et al. study their influence on code optimization for Fermi architectures.

Writing efficient GPU code is often difficult and requires the exercise of specialized architectural features. Good performance can usually be achieved only after an intensive manual tuning phase. A programmer usually needs to test combinations of multi code versions, architectural parameters, and data inputs. N. Farooqui et al. therefore developed Leo [24].

Leo automates much of the effort needed to instrument codes. Using Leo, a user simply insert some abstraction commands in codes. By analyzing the abstractions, Leo explores the optimization space and accordingly modified the code. Using Leo, Farooqui et al. report achieving "from 1.12 to 27x speedup in kernel runtimes, which translates to 7-40% improvement for end-to-end performance" (page 1 of [24]).

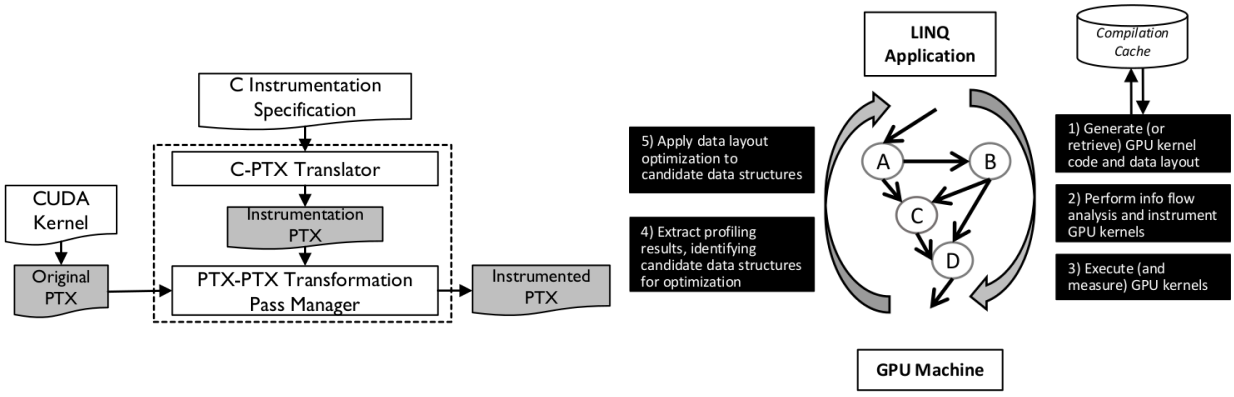

Figure 9.29: GPU instrumentation engine and high-level overview of Leo (courtesy of [24]).

Finally, Magni et al. [39] point out that saturating the GPU hardware resources can be used to reduce the tuning overhead by one order of magnitude. This is important because producing high performance code on GPUs is often very time consuming. Furthermore, the whole tuning process has to be repeated for each target code and platform.

9.4 The Position and Importance of Our Work

There has been only one very weak attempt to reverse engineer the real ISA used by Fermi architectures (*https://code.google.com/p/asfermi/*). However, that attempt quickly failed because of the big challenges encountered in the reverse engineering. Furthermore, we experimentally verified that this previous effort, even for the few PTX instructions that were reverse engineered, was compiler dependent and worked only with very old compiler versions. In addition, there are known issues in the asFermi project code, and there is no clear explanation of the validation procedures that were used for the few PTX instructions that were reverse engineered. We are therefore the first and only researchers to have accurately

reverse engineered almost every PTX instruction (virtual ISA) and ELF instruction (real ISA) for Fermi/Tesla architectures that use the GF100 architecture, and to scientifically validate what we discovered.

Furthermore, we are the first to discover: 1) that the real NVIDIA ISAs do no have a fixed format; 2) how many and what type of ELF instructions are needed to execute each PTX instruction; 3) the correspondences between PTX and ELF registers; 4) how to exploit the compiler to map ELF registers to hardware registers; 5) how the gigathread scheduler distributes thread blocks to the steaming multiprocessors; 6) the probable warp schedulers' policy; 7) the maximum throughput of each PTX and ELF instruction; 8) the warp latency of each instruction; 9) the write-read dependence latency for the re-use of the result register of each instruction; 10) the read-read dependence latency for the re-use of the operand register of each instruction; 11) the conditions that create local instabilities in a streaming multiprocessor, even if all the data are local and always available into the hardware registers.

In addition, for any NVIDIA GPU architecture post-2009, we are also the first to have: 1) devised a reverse engineering methodology to discover their real ISAs; 2) implemented this methodology into a set of autonomic tools; 3) devised another methodology to produce any desired real ISA kernel and made the procedure independent of the compiler version; 4) implemented this methodology into a set of semi-autonomic tools; 5) devised another methodology to create specific real ISA kernels for the discovery and quantification of the GPU undisclosed low level architectural features and machine behaviors; 6) implemented the methodology into a set of autonomic tools that produce these real ISA kernels, and discovered and quantified the GPU undisclosed low level architectural features and machine behaviors; 7) discovered that, even for very simple kernels like the microbenchmark kernels, you lose more than 40% of the total performance if you use PTX and the compiler; and, finally, 8) speeded up code performance by using the real ISAs and by exploiting the insights learned during the discovery and quantification of the GPU low level architectural features and machine behaviors for code development and code design.

Chapter 10

Conclusion

NVIDIA GPUs are very powerful machines and are able to execute TeraFlops of instructions per second. However, accelerating codes on GPUs is difficult. This is partially due to the fact that their architectures are different than CPU architectures. In addition to these differences, we also do not have control of the assembly code executed by NVIDIA GPUs, cannot modify the compiler code, and do not know how to identify and exploit the GPU low level architectural features and machine behaviors. This is the case because, to protect its market leadership position, NVIDIA does not disclose its real assembly (ELF) Instruction Set Architectures (ISAs), keeps the compiler (nvcc) code closed, and does not disclose many of its GPU low level architectural features and machine behaviors.

This lack of control and information creates some challenges. Regarding assembly codes, we depend on the compiler for code translation, and there is no guarantee that different compiler versions or different compilers will not produce codes that will have completely different efficiencies. Regarding compiler efficiency, we cannot improve the compiler because we cannot directly implement any optimization in it. Regarding GPU features and behaviors, it is difficult to develop scientific design methodologies that produce a priori code efficiency guarantees and/or to eliminate the very time consuming trials and errors that are now necessary to optimize codes.

However, thanks to this thesis, it is now possible to reverse engineer the ELF ISA of any NVIDIA GPU manufactured after 2009. This is because we developed a set of autonomic tools that uses cuobjdump, which is supported in all NVIDIA releases.

In using these tools, we discovered that many single PTX instructions are transformed into subsets of ELF instructions, and that the number of ELF registers of the ELF instructions that are necessary to execute a single PTX instruction is usually greater than the number of PTX registers that appear in the PTX instruction. The selection of the right PTX instructions is therefore important to avoid code slowdowns, and is only possible after the reverse engineering is completed.

In addition, the tools also discovered that the ELF ISAs do not use a fixed format, and that there are no ELF instructions to declare ELF registers. This means that nvcc injects ELF instructions, which assign subsets of ELF registers to each GPU thread and map each ELF register to a different hardware register.

By exploiting nvcc, we therefore developed a procedure that generates fatbin files. These fatbin files have the necessary ELF assignment/mapping instructions, and also have the hardware resources that are necessary for their partial overwriting. If we use this procedure and the reverse engineered ISAs, then we can produce any desired ELF kernel. The procedure is applicable to any NVIDIA GPU after 2009.

We also developed another set of autonomic tools that produce ELF kernels, and discover and quantify the many undisclosed GPU low level architectural features and machine behaviors. The discovery and quantification of these features and behaviors is important for exploring future hardware possibilities, for better understanding the NVIDIA GPU architectures, and for making the optimization process more scientific.

This second set of tools also discovered and quantified the relationships among the following features and behaviors: a) the number of thread blocks used to execute a GPU code; b) the number of threads per thread block; c) the number of hardware registers assigned to each thread; d) the assignment policies used by the gigathread scheduler to distribute the

thread blocks to the streaming multiprocessors; e) the maximum throughputs of each PTX or ELF instruction (the throughput is expressed as the number of instructions executed per functional unit clock cycle per streaming multiprocessor); f) the minimum number of warps that need to reside in a streaming multiprocessor to reach the maximum throughput of each single PTX or ELF instruction; g) the warps' latencies per PTX or ELF instruction; h) the situations that generate local instabilities in the streaming multiprocessors; and i) the write-read and read-read latencies of the registers used in each PTX or ELF instruction (the latencies are expressed as the number of functional unit clock cycles).

We also found that it is impossible to accurately and correctly discover and quantify the GPU low level architectural features and machine behaviors without bypassing the compiler transformations. However, accuracy and correctness are important for understanding how to efficiently optimize codes.

Furthermore, exploiting the insight gained during the reverse engineering phase, and the direct translation of PTX codes to ELF codes, together produced ELF codes that are easily 1 or 2 orders of magnitude faster than the ELF codes produced by nvcc. This happens even for very simple PTX codes like those used for the microbenchmarks. The compiler versions would therefore benefit from the integration of this knowledge.

In addition, a person currently has to consider a staggering number of low level architectural features and machine behaviors when he/she designs and optimizes high performance computing codes using modern architectures. This problem will be further compounded by increasing performance scale. Autonomic tools like those we have developed hide from users the staggering number of details, and therefore simplify their jobs. Furthermore, the tools can drive code design and present to users the best choices for code performance and efficiency.

Our control of the ELF code executed by the GPU, and the discovery and quantification of the GPU low level architectural features and machine behaviors, also allow us to create an accurate and fully parametric model of the machine. The integration of this model in a

simulator will allow us to study and explore architectural possibilities that do not yet exist, as well as assessing the impact of future architectural choices on existing or hypothetical codes.

The discovery and quantification of the GPU low level architectural features and machine behaviors also allow for the development of proofs that produce a priori code efficiency guarantees. These guarantees are especially important for corporations that design and manufacture supercomputers (e.g. NVIDIA, IBM and Intel) and national laboratories (e.g. LLNL and Oak Ridge). This is because the ability to provide a priori code efficiency guarantees on machines that need to be built and delivered in the next 2-5 years would be beneficial for both the manufactures (i.e. higher probability to win contracts) and the users (i.e. guarantee that the codes will be efficient and therefore that they will be as fast as possible).

Finally, while we move to exascale performance, it is impossible to expect users to be able to effectively manage the greater and greater number of low level details already present in current architectures. If we do not create a middle layer that hides from users the staggering architectural complexities of future machines, then creating efficient codes will become increasingly difficult.

As with this thesis, the goal of this layer should be to discover and quantify the low level architectural features and machine behaviors, and to generate parametric formulas and inequalities that capture the relationships among these features and behaviors. These formulas and inequalities would create very big systems of equations and inequalities like those used in operational research. Methods applied in operational research would therefore be applicable to these systems as well. This layer cannot solve NP-Complete problems, but it can interact with users during the design and optimization of codes. This should be done by hiding the underlying machine features and behaviors, and by forcing the users to select the right design / optimization choices after the reception of some users' inputs.

Furthermore, it is probable that the number of hardware defects and bugs in manufac-

turing processes will increase. Users can run the layer only one time or only at periodic intervals. At each run, the layer can: a) be used to discover hardware design bugs or manufacturing defects; b) be used to discover how the features and behaviors of the underlying aging hardware changes; c) incorporate insights from the previous processes into equations and inequalities; and d) advise the user of the changes that will lower the efficiencies of the running codes.

Bibliography

[1] A.Davidson and J.D.Owens. Toward Techniques for Auto-Tuning GPU Algorithms. *Applied Parallel and Scientific Computing, Lecture Notes in Computer Science*, 7134:110–119, 2012.

[2] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, 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 (PPoPP)*, pages 105–114, 2010.

[3] S. S. Baghsorkhi, I. Gelado, M. Delahaye, and W. W. Hwu. Efficient Performance Evaluation of Memory hierarchy for Highly Multithreaded Graphics Processors. *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 23–24, 2012.

[4] M. M. Baskaran, J. Ramanujam, 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.

[5] G. V. D. Braak, B. Mesman, and H. Corporaal. Compile-Time GPU Memory Access Optimizations. *International Conference on Embedded Computer Systems*, pages 200–207, 2010.

[6] A. R. Brodtkorb, T. R. Hagen, and M. L. SaTra. Graphics Processing Unit (GPU) Programming Strategies and Trends in GPU Computing. *J. of Parallel and Distributed Computing*, 73(1):4–13, 2013.

[7] M. Burtscher, R. Nasre, and K. Pingali. A Quantitative Study of Irregular Programs on GPUs. *IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151, 2012.

[8] L. Chen, L. Liu, S. Tang, L. Huang, Z. Jing, S. Xu, D. Zhang, and B. Shou. Unified Parallel C for GPU Clusters: Language Extensions and Compiler Implementation. *Lecture Notes in Computer Science on Languages and Compilers for Parallel Computing*, 6548:151–165, 2011.

[9] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. Wrox, 2014.

[10] J. W. Choi, A. Singh, 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.

[11] S. Collange, D. Defour, and D. Parello. Barra, a Parallel Functional GPGPU Simulator. *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 351–360, 2010.

[12] S. Cook. *CUDA Programming: a Developer's Guide to Parallel Computing with GPUs (Applications of GPU Computing)*. Morgan Kaufmann, 2012.

[13] N. Corporation. NVIDIA GeForce GTX 750 Ti. `http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf`. [Online; accessed 22-Decemberr-2014].

[14] N. Corporation. NVIDIAs Next Generation CUDA Compute Architecture: Fermi. `http://www.nvidia.com/content/pdf/fermi_white_papers/nvidiafermicomputearchitecturewhitepaper.pdf`. [Online; accessed 24-Decemberr-2014].

[15] N. Corporation. NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf`. [Online; accessed 29-Decemberr-2014].

[16] I. Corporbation. 2nd Generation Intel Core vPro Processor Family. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/core-vpro-2nd-generation-core-vpro-processor-family-paper.pdf`. [Online; accessed 30-Decemberr-2014].

[17] I. Corporbation. A Quantum Leap in Enterprise Computing. `http://www.intel.com/Assets/en_US/PDF/prodbrief/323499.pdf`. [Online; accessed 26-Decemberr-2014].

[18] I. Corporbation. Intel Core i7-3960X Processor Extreme Ed. `http://ark.intel.com/products/63696`. [Online; accessed 27-Decemberr-2014].

[19] H. Cui, L. Wang, J. Xue, Y. Yang, and X. Feng. Automatic Library Generation for BLAS3 on GPUs. *Parallel and Distributed Processing Symposium*, 2011.

[20] X. Cui, Y. Chen, C. Zhang, and H. Mei. Auto-Tuning Dense Matrix Multiplication for GPGPU with Cache. *International Conference on Parallel and Distributed Systems*, 2010.

[21] G. F. Diamos, A. R. Kerr, S. Yalamanchili, 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.

[22] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a High-Level Language for GPUs (via Language Support for Architectures and Compilers). *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 2012.

[23] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan. A Framework for Dynamically Instrumenting GPU Compute Applications within GPU Ocelot. *4th Workshop on General-Purpose Computation on Graphics Procesing Units*, pages 1–9, 2011.

[24] N. Farooqui, C. Rossbach, Y. Yu, and K. Schwan. Leo: A Profile-Driven Dynamic Optimization Framework for GPU Applications. *Conference on Timely Results in Operative Systems*, pages 1–14, 2014.

[25] W. W. L. Fung and T. M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture*, pages 25–36, 2011.

[26] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(2):1–12, 2009.

[27] J. Ghorpade, J. Parande, M. Kulkarni, and A. Bawaska. GPGPU Processing in CUDA Architecture. *Advanced Computing: An International Journ.*, 3(1):1–16, 2012.

[28] P. N. Glaskowsky. NVIDIAs Fermi: The First Complete GPU Computing Architecture . `http://sbel.wisc.edu/Courses/ME964/Literature/whitePaperFermiGlaskowsky.pdf`. [Online; accessed 19-Decemberr-2014].

[29] S. G. Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazo. Auto-Tuning a High-Level Language Targeted to GPU Codes. *Innovative Parallel Computing (InPar)*, pages 1–10, 2012.

[30] 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.

[31] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 152–163, 2009.

[32] S. Huang, S. Xiao, and W. Feng. On the Energy Efficiency of Graphics Processing Units for Scientific Computing. *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2009.

[33] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU Communication Management and Optimization. *Conference on Programming Language Design and Implementation*, pages 142–151, 2011.

[34] Y. Jiao, H. Lin, P. Balaji, and W. Feng. Power and Performance Characterization of Computational Kernels on the GPU. *2010 IEEE/ACM International Conference on Physical and Social Computing (CPSCom)*, pages 221–228, 2010.

[35] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors, a Hands-on Approach*. Morgan Kaufmann, 2012.

[36] J. Kurzak, S. Tomov, and J. Dongarra. Autotuning GEMM Kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2045–2057, 2012.

[37] S. Lee and J. S. Vetter. Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2012.

[38] Y. Liu, E. Z. Zhang, 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.

[39] A. Magni, C. Dubach, and M. F. P. O. Boyle. Exploiting GPU Hardware Saturation for Fast Compiler Optimization. *Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–8, 2014.

[40] C. McClanahan. History and Evolution of GPU Architecture. `http://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf`, 2011. [Online; accessed 20-Decemberr-2014].

[41] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, 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*, pages 1–11, 2011.

[42] Nvidia. CUDA C Best Practices Guide Version 3.1. `http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation`, 2010. [Online; accessed 04-July-2011].

[43] Nvidia. CUDA C Programming Guide Version 3.1. `http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation`, 2010. [Online; accessed 11-January-2011].

[44] Nvidia. PTX (Parallel Thread Execution) ISA Version 2.2. `http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation`, 2010. [Online; accessed 27-June-2011].

[45] Nvidia. CUDA C Best Practices Guide Version 4.1. `http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation`, 2012. [Online; accessed 07-September-2012].

[46] Nvidia. CUDA C Programming Guide Version 4.1. `http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation`, 2012. [Online; accessed 13-September-2012].

[47] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Graphics Processing Units - Powerful, Programmable, and Highly Parallel - are Increasingly Targeting General-Purpose Computing Applications. *Proceedings of the IEEE*, 96(5):879–889, 2008. [Online; accessed 22-Decemberr-2014].

[48] P. Ricoux, J. Y. Berthou, and T. Bidot. European Exascale Software Initiative (EESI2): Towards Exascale Roadmap Implementations. `http://www.eesi-project.eu/pages/menu/homepage.php`, 2014. [Online; accessed 19-Decemberr-2014].

[49] G. Rudy, M. M. Khan, M. Hall, C. Chen, and J. Chame. A Programming Language Interface to Describe Transformations and Code Generation. *Lecture Notes in Computer Science on Languages and Compilers for Parallel Computing*, 6548:136–150, 2011.

[50] J. Sanders and E. Kandrot. *CUDA by Example: an Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[51] S.Hong and H. Kim. An Integrated GPU Power and Performance Model. *Proceedings of the 37th Annual International Symposium on Computer Architectures*, pages 280–289, 2010.

[52] G. Sissons and B. McMillan. Improving the Efficiency of GPU Clusters. `http://web.stanford.edu/group/hpc/cgi-bin/hpcday2010/wp-content/uploads/2010/08/platform_gpu_whitepaper_071410.pdf`. [Online; accessed 28-Decemberr-2014].

[53] 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.

[54] J. A. Stratton, N. Anssari, C. Rodrigues, S. I., N. Obeid, C. Liwen, G. Liu, and W. Hwu. Optimization and Architecture Effects on GPU Computing Workload Performance. *Innovative Parallel Computing*, pages 1–10, 2012.

[55] Y. Torres, A. G. Escribano, and D. R. Llanos. Understanding the Impact of CUDA Tuning Techniques for Fermi. *High Performance Computing Symposium*, pages 631–639, 2011.

[56] D. Tristram and K. Bradshaw. Determining the Difficulty of Accelerating Problems on a GPUU. *South African Computer Journal*, 53:1–15, 2014.

[57] S. Ueng, M. Lathara, S. S. Baghsorkhi, and W. W. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. *Languages and Compilers for Parallel Computing*, pages 1–15, 2008.

[58] F. Vazqueza, J. Fernandeza, and E. Garzon. Automatic Tuning of the Sparse Matrix Vector Product on GPUs Based on the ELLR-T Approach. *Parallel Computing*, 38(8):408–420, 2012.

[59] Y. Wang and N. Ranganathan. An Instruction-Level Energy Estimation and Optimization Methodology for GPU. *IEEE 11th International Conference on Computer and Information Technology (CIT)*, pages 621–628, 2011.

[60] B. Wilkinson. Emergence of GPU systems and clusters for general purpose High Performance Computing. `http://www.google.com/url?sa=tandrct=jandq=andesrc=sandsource=webandcd=3andved=0CDUQFjACandurl=http%3A%2F%2Fcoitweb.uncc.edu%2F~abw%2FITCS6010S11%2FCUDAHistory.pptandei=xJ2IVJ71LZeYoQSJx4DIBgandusg=AFQjCNEL7SLrbUmMHa_5m7eqT0Lz1uEL-Aandsig2=IdesJbwi7VhpMCI4hG1oswandbvm=bv.81456516,d.cGU`, 2011. [Online; accessed 21-Decemberr-2014].

[61] N. Witt. *The CUDA Handbook: a Comprehensive Guide to GPU Programming*. Addison-Wesley Professional, 2013.

[62] H. Wong, M. M. Papadopoulou, M. S. Alvandi, and A. Moshovos. Demystifying GPU Microarchitecture Through Microbenchmarking. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 235–246, 2010.

[63] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for Memory Optimization and Parallelism Management. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 86–97, 2010.

[64] Y. Yang, P. Xiang, J. Kong, 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.

[65] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, 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.

[66] Y. Zhang and J. Owens. A Quantitative Performance Analysis Model for GPU Architectures. *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pages 382–393, 2011.

Appendices

AI Some Reverse Engineered ELF Instructions

We report here some of the reverse engineered ELF instructions, the bit values of their opcodes, and the bit positions associated with each register or constant that appears in the instructions.

Table 1: Algorithm To Create An Abstract Human Readable Text Form Representation

```
Human Readable Text Form Representation: SEL R2, R10, R2, !P0;

Registers   Our Identifiers   Order per Identifier   Asbtract Forms
       R2               NT2                      0           NT2ER0
      R10               NT2                      1           NT2ER1
       R2               NT2                      2           NT2ER2
       P0               NT1                      0           NT1ER0

Example -> NT2ER1
N       -> sub-type of the ELF register: not reserved
T2      -> type of the ELF register: normal ELF register
ER1     -> from left to right, second normal not reserved ELF register

Example -> ST1ER4
S       -> sub-type of the ELF register: reserved
T1      -> type of the ELF register: predicate ELF register
ER4     -> from left to right, fourth predicate reserved ELF register

Abstract Forms   Frequencies   Features
           ST2             0          0
           ST1             0        0_0
           NT2             3      0_0_3
           NT1             1    0_0_3_1

Position Result Register: 0 -> first register appearing in the instruction
                          (this is not always true for other instructions)

Abstract Form Result Register: NT2ER0 (possibly anything for other instr.)

Human Readable .....      Abstract Human Readable Text Form Representation
SEL R2, R10, R2, !P0  SEL_NT2ER0,_NT2ER1,_NT2ER2,_!NT1ER0_0_0_3_1_0_NT2ER0

If there is any constant then the result register is transformed to XXXX
to represent the fact that the instruction uses some constants as opera-
nds.
```

Table 2: The Features Of An Abstract Human Readable Text Form Representation

```
Example of Abstract Human Readable Text Form Representation

SEL_NT2ER0,_NT2ER1,_NT2ER2,_!NT1ER0_0_0_3_1_0_NT2ER0

The SEL ELF instruction does not use any reserved ELF register. It instead
uses 3 normal ELF registers and 1 normal predicate (negated) ELF register.

-------------------------------------------------------------------------

This SEL ELF instruction does not require the use of reserved ELF registe-
rs. Furthermore, reserved ELF registers are not considered for the calcu-
lation of the order of the identifiers. The bits of the reserved registers
are in fact considered fixed, as are the bits of the opcode of the instru-
ction.

-------------------------------------------------------------------------

Definition of Sensible Field: (Any not reserved ELF register or constant)

Legend:
 -: bit equal to 0
 +: bit equal to 1
 0: bit positions associated to the first  sensible field from left
 1: bit positions associated to the second sensible field from left
 2: bit positions associated to the third  sensible field from left
 3: bit positions associated to the forth  sensible field from left

-------------------------------------------------------------------------

Bit Flipped   Bit Value    Field  ELF Register Type  Order Per Identifier

         63           -   opcode                  \                     \
         62           -   opcode                  \                     \
         61           +   opcode                  \                     \
         60           -   opcode                  \                     \
         59           -   opcode                  \                     \
         58           -   opcode                  \                     \
         57           -   opcode                  \                     \
         56           -   opcode                  \                     \
         55           -   opcode                  \                     \
         54           -  opcode                  \                     \
         53           -   opcode                  \                     \
```

```
          52          +   opcode                  \                   \
          51        -\+   NT1ER0           predicate                   3
          50        -\+   NT1ER0           predicate                   3
          49        -\+   NT1ER0           predicate                   3
          48          -   opcode                  \                   \
          47          -   opcode                  \                   \
          46          -   opcode                  \                   \
          45          -   opcode                  \                   \
          44          -   opcode                  \                   \
          43          -   opcode                  \                   \
          42          -   opcode                  \                   \
          41          -   opcode                  \                   \
          40          -   opcode                  \                   \
          39          -   opcode                  \                   \
          38          -   opcode                  \                   \
          37          -   opcode                  \                   \
          36          -   opcode                  \                   \
          35          -   opcode                  \
          34          -   opcode                  \                   \
          33          -   opcode                  \                   \
          32          -   opcode                  \                   \
          31        -\+   NT2ER2             normal                   2
          30        -\+   NT2ER2             normal                   2
          29        -\+   NT2ER2             normal                   2
          28        -\+   NT2ER2             normal                   2
          27        -\+   NT2ER2             normal                   2
          26        -\+   NT2ER2             normal                   2
          25        -\+   NT2ER1             normal                   1
          24        -\+   NT2ER1             normal                   1
          23        -\+   NT2ER1             normal                   1
          22        -\+   NT2ER1             normal                   1
          21        -\+   NT2ER1             normal                   1
          20        -\+   NT2ER1             normal                   1
          19        -\+   NT2ER0             normal                   0
          18        -\+   NT2ER0             normal                   0
          17        -\+   NT2ER0             normal                   0
          16        -\+   NT2ER0             normal                   0
          15        -\+   NT2ER0             normal                   0
          14        -\+   NT2ER0             normal                   0
          13          -   opcode                  \                   \
          12          +   opcode                  \                   \
          11          +   opcode                  \                   \
           .          .   opcode                  \                   \
           0          -   opcode                  \                   \
```

Table 3: Identifiers and abstract forms of few discovered ELF instructions (Part 1).

ID	ABSTRACT TEXT FORM REPRESENTATIONS
001	BAR.RED.POPC_ST2ER0,_ST2ER1_2_0_0_0_0_XXXX
002	BAR.RED.POPC_ST2ER0,_XxX0_1_0_0_0_1_XXXX
003	BFE.U32_NT2ER0,_NT2ER1,_XxX0_0_0_2_0_1_XXXX
004	BPT.DRAIN_XxX0_0_0_0_0_1_XXXX
005	CAL_XxX0_0_0_0_0_1_XXXX
006	DADD.ST2ER0_NT2ER0,_NT2ER1,_NT2ER2_1_0_3_0_0_XXXX
007	DFMA_NT2ER0,_NT2ER1,_-NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
008	DFMA_NT2ER0,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
009	DFMA.RM_NT2ER0,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
010	DFMA.RP_NT2ER0,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
011	DFMA.ST2ER0_NT2ER0,_NT2ER1,_NT2ER2,_NT2ER3_1_0_4_0_0_XXXX
012	DMUL_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
013	DMUL_NT2ER0,_NT2ER1,_XxX0_0_0_2_0_1_XXXX
014	DSETP.EQ.AND_NT1ER0,_ST1ER0,_NT2ER0,_ST2ER0,_ST1ER1_1_2_1_1_0_NT1ER0
015	DSETP.EQ.AND_NT1ER0,_ST1ER0,__NT2ER0_,_XxX0,_ST1ER1_0_2_1_1_1_XXXX
016	DSETP.LE.AND_NT1ER0,_ST1ER0,__NT2ER0_,_XxX0,_ST1ER1_0_2_1_1_1_XXXX
017	DSETP.NE.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0
018	EXIT_0_0_0_0_0_XXXX
019	F2F.FTZ.PASS_NT2ER0,__NT2ER1__0_0_2_0_0_NT2ER0
020	F2F.FTZ.PASS_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
021	F2I.FTZ.U32.F32.TRUNC_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
022	F2I.U32.F32.TRUNC_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
023	F2I.U64.F32.TRUNC_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
024	FADD.FTZ_NT2ER0,_NT2ER1,_XxX0_0_0_2_0_1_XXXX
025	FADD_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
026	FFMA_NT2ER0,_NT2ER1,_-NT2ER2,_c_[XxX0]_[XxX1]_0_0_3_0_2_XXXX
027	FFMA_NT2ER0,_NT2ER1,_NT2ER2,_c_[XxX0]_[XxX1]_0_0_3_0_2_XXXX
028	FFMA_NT2ER0,_NT2ER1,_-NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
029	FFMA_NT2ER0,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
030	FFMA_NT2ER0,_NT2ER1,_XxX0,_ST2ER0_1_0_2_0_1_XXXX
031	FFMA.RM_NT2ER0,_NT2ER1,_-NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
032	FFMA.RM_NT2ER0,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
033	FFMA.RP_NT2ER0,_NT2ER1,_NT2ER2,_c_[XxX0]_[XxX1]_0_0_3_0_2_XXXX
034	FFMA.RP_NT2ER0,_NT2ER1,_-NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
035	FLO.U32_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
036	FMUL.FTZ_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
037	FMUL.FTZ_NT2ER0,_NT2ER1,_XxX0_0_0_2_0_1_XXXX
038	FMUL_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
039	FMUL_NT2ER0,_NT2ER1,_XxX0_0_0_2_0_1_XXXX
040	FMUL.ST2ER0_NT2ER0,_NT2ER1,_NT2ER2_1_0_3_0_0_XXXX
041	FSETP.EQ.FTZ.AND_NT1ER0,_ST1ER0,_NT2ER0,_ST2ER0,_ST1ER1_1_2_1_1_0_NT1ER0
042	FSETP.EQ.FTZ.AND_NT1ER0,_ST1ER0,_NT2ER0,_XxX0,_ST1ER1_0_2_1_1_1_XXXX
043	FSETP.GT.AND_NT1ER0,_ST1ER0,__NT2ER0_,_XxX0,_ST1ER1_0_2_1_1_1_XXXX
044	FSETP.LE.FTZ.AND_NT1ER0,_ST1ER0,__NT2ER0_,_XxX0,_ST1ER1_0_2_1_1_1_XXXX
045	FSETP.LT.AND_NT1ER0,_ST1ER0,__NT2ER0_,_XxX0,_ST1ER1_0_2_1_1_1_XXXX
046	FSETP.LT.AND_NT1ER0,_ST1ER0,_NT2ER0,_XxX0,_ST1ER1_0_2_1_1_1_XXXX
047	FSETP.LT.FTZ.AND_NT1ER0,_ST1ER0,_NT2ER0,_ST2ER0,_ST1ER1_1_2_1_1_0_NT1ER0
048	FSETP.NEU.FTZ.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0
049	I2F.F32.U32_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0

Table 4: Identifiers and abstract forms of few discovered ELF innstructions (Part 2).

ID	ABSTRACT TEXT FORM REPRESENTATIONS
050	I2F.F32.U32.RP_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
051	I2F.F32.U32.ST2ER0_NT2ER0,_NT2ER1_1_0_2_0_0_XXXX
052	I2F.F32.U64.RP_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
053	I2I.S32.S16_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
054	I2I.S32.S32_NT2ER0,__NT2ER1__0_0_2_0_0_NT2ER0
055	I2I.S32.S32_NT2ER0,-NT2ER1_0_0_2_0_0_NT2ER0
056	I2I.U16.U32_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
057	I2I.U32.U16_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
058	I2I.U32.U8_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
058	IADD32I_NT2ER0,_NT2ER1,-XxX0_0_0_2_0_1_XXXX
059	IADD32I_NT2ER0,_NT2ER1,_XxX0_0_0_2_0_1_XXXX
060	IADD_NT2ER0.CC,_NT2ER1,_c_[XxX0]_[XxX1]_0_0_2_0_2_XXXX
061	IADD_NT2ER0.CC,_NT2ER1,-NT2ER2_0_0_3_0_0_NT2ER0
062	IADD_NT2ER0.CC,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
063	IADD_NT2ER0.CC,-NT2ER1,_ST2ER0_1_0_2_0_0_NT2ER0
064	IADD_NT2ER0.CC,_NT2ER1,_XxX0_0_0_2_0_1_XXXX
065	IADD_NT2ER0,_NT2ER1,-NT2ER2_0_0_3_0_0_NT2ER0
066	IADD_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
067	IADD_NT2ER0,-NT2ER1,_XxX0_0_0_2_0_1_XXXX
068	IADD_NT2ER0,_NT2ER1,_XxX0_0_0_2_0_1_XXXX
069	IADD_NT2ER0,_ST2ER0,-NT2ER1_1_0_2_0_0_NT2ER0
070	IADD_ST2ER0.CC,_NT2ER0,-NT2ER1_1_0_2_0_0_XXXX
071	IADD_ST2ER0.CC,_NT2ER0,-ST2ER1_2_0_1_0_0_XXXX
072	IADD.X_NT2ER0,_NT2ER1,_c_[XxX0]_[XxX1]_0_0_2_0_2_XXXX
073	IADD.X_NT2ER0,_NT2ER1,-NT2ER2_0_0_3_0_0_NT2ER0
074	IADD.X_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
075	IADD.X_NT2ER0,-NT2ER1,_ST2ER0_1_0_2_0_0_NT2ER0
077	IADD.X_NT2ER0,_NT2ER1,_ST2ER0_1_0_2_0_0_NT2ER0
078	IADD.X_NT2ER0,_NT2ER1,_XxX0_0_0_2_0_1_XXXX
079	IADD.X_NT2ER0,_ST2ER0,_c_[XxX0]_[XxX1]_1_0_1_0_2_XXXX
080	IADD.X_NT2ER0,_ST2ER0,_ST2ER1_2_0_1_0_0_NT2ER0
081	ICMP.EQ_NT2ER0,_NT2ER1,_c_[XxX0]_[XxX1],_NT2ER2_0_0_3_0_2_XXXX
082	ICMP.EQ.U32_NT2ER0,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
083	ICMP.LT_NT2ER0,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
084	IMAD.U32.U32.HI_NT2ER0,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
085	IMAD.U32.U32.HI.X_NT2ER0.CC,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
086	IMAD.U32.U32.HI.X_NT2ER0.CC,_NT2ER1,_NT2ER2,_ST2ER0_1_0_3_0_0_NT2ER0
087	IMAD.U32.U32.HI.X_NT2ER0,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
088	IMAD.U32.U32.HI.X_NT2ER0,_NT2ER1,_NT2ER2,_ST2ER0_1_0_3_0_0_NT2ER0
089	IMAD.U32.U32_NT2ER0.CC,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
090	IMAD.U32.U32_NT2ER0,-NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
091	IMAD.U32.U32_NT2ER0,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
092	IMAD.U32.U32_ST2ER0.CC,_NT2ER0,_NT2ER1,_NT2ER2_1_0_3_0_0_XXXX
093	IMAD.U32.U32_ST2ER0,_NT2ER0,_NT2ER1,_ST2ER1_2_0_2_0_0_XXXX
094	IMAD.U32.U32_ST2ER0,_NT2ER0,_ST2ER1,_ST2ER2_3_0_1_0_0_XXXX
095	IMAD.U32.U32.X_NT2ER0.CC,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
096	IMAD.U32.U32.X_NT2ER0,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0
097	IMNMX_NT2ER0,_NT2ER1,_NT2ER2,_!ST1ER0_0_1_3_0_0_NT2ER0
098	IMNMX_NT2ER0,_NT2ER1,_NT2ER2,_ST1ER0_0_1_3_0_0_NT2ER0

Table 5: Identifiers and abstract forms of few discovered ELF innstructions (Part 3).

ID	ABSTRACT TEXT FORM REPRESENTATIONS
099	IMNMX.U32_NT2ER0,_NT2ER1,_NT2ER2,_!ST1ER0_0_1_3_0_0_NT2ER0
100	IMNMX.U32_NT2ER0,_NT2ER1,_NT2ER2,_ST1ER0_0_1_3_0_0_NT2ER0
101	IMNMX.U32.XHI_NT2ER0.CC,_NT2ER1,_NT2ER2,_!ST1ER0_0_1_3_0_0_NT2ER0
102	IMNMX.U32.XHI_NT2ER0.CC,_NT2ER1,_NT2ER2,_ST1ER0_0_1_3_0_0_NT2ER0
103	IMNMX.U32.XLO_NT2ER0,_NT2ER1,_NT2ER2,_!ST1ER0_0_1_3_0_0_NT2ER0
104	IMNMX.U32.XLO_NT2ER0,_NT2ER1,_NT2ER2,_ST1ER0_0_1_3_0_0_NT2ER0
105	IMNMX.XHI_NT2ER0.CC,_NT2ER1,_NT2ER2,_!ST1ER0_0_1_3_0_0_NT2ER0
106	IMNMX.XHI_NT2ER0.CC,_NT2ER1,_NT2ER2,_ST1ER0_0_1_3_0_0_NT2ER0
107	IMUL.HI_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
108	IMUL_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
109	IMUL.U32.U32.HI_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
110	IMUL.U32.U32_NT2ER0.CC,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
111	IMUL.U32.U32_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
112	IMUL.U32.U32_ST2ER0.CC,_NT2ER0,_NT2ER1_1_0_2_0_0_XXXX
113	ISCADD_NT2ER0,_-NT2ER1,_NT2ER2,_XxX0_0_0_3_0_1_XXXX
114	ISCADD_NT2ER0,_NT2ER1,_NT2ER2,_XxX0_0_0_3_0_1_XXXX
115	ISETP.EQ.AND_NT1ER0,_ST1ER0,_NT2ER0,_ST2ER0,_ST1ER1_1_2_1_1_0_NT1ER0
116	ISETP.EQ.U32.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0
117	ISETP.EQ.U32.AND_NT1ER0,_ST1ER0,_NT2ER0,_ST2ER0,_NT1ER1_1_1_1_2_0_NT1ER0
118	ISETP.EQ.U32.AND_NT1ER0,_ST1ER0,_NT2ER0,_ST2ER0,_ST1ER1_1_2_1_1_0_NT1ER0
119	ISETP.GE.AND_NT1ER0,_ST1ER0,_NT2ER0,_ST2ER0,_ST1ER1_1_2_1_1_0_NT1ER0
120	ISETP.GE.AND_NT1ER0,_ST1ER0,_NT2ER0,_XxX0,_ST1ER1_0_2_1_1_1_XXXX
121	ISETP.GE.U32.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0
122	ISETP.GE.U32.X.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0
123	ISETP.GT.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0
124	ISETP.GT.U32.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0
125	ISETP.GT.U32.AND_NT1ER0,_ST1ER0,_NT2ER0,_XxX0,_ST1ER1_0_2_1_1_1_XXXX
126	ISETP.GT.U32.OR_NT1ER0,_ST1ER0,_NT2ER0,_XxX0,_NT1ER1_0_1_1_2_1_XXXX
127	ISETP.GT.U32.X.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0
128	ISETP.GT.X.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0
129	ISETP.LE.AND_NT1ER0,_ST1ER0,_NT2ER0,_XxX0,_ST1ER1_0_2_1_1_1_XXXX
130	ISETP.LE.U32.AND_NT1ER0,_ST1ER0,_NT2ER0,_c_[XxX0]_[XxX1],_ST1ER1_0_2_1_1_2_XXXX
131	ISETP.LE.U32.AND_NT1ER0,_ST1ER0,_NT2ER0,_XxX0,_ST1ER1_0_2_1_1_1_XXXX
132	ISETP.LT.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0
133	ISETP.LT.U32.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0
134	ISETP.LT.U32.X.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0
135	ISETP.LT.X.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0
136	ISETP.NE.AND_NT1ER0,_ST1ER0,_NT2ER0,_ST2ER0,_ST1ER1_1_2_1_1_0_NT1ER0
137	ISETP.NE.U32.AND_NT1ER0,_ST1ER0,_NT2ER0,_ST2ER0,_NT1ER1_1_1_1_2_0_NT1ER0
138	ISETP.NE.U32.AND_NT1ER0,_ST1ER0,_NT2ER0,_ST2ER0,_ST1ER1_1_2_1_1_0_NT1ER0
139	ISETP.NE.U32.X.AND_NT1ER0,_ST1ER0,_NT2ER0,_ST2ER0,_ST1ER1_1_2_1_1_0_NT1ER0
140	ISETP.NE.X.AND_NT1ER0,_ST1ER0,_NT2ER0,_ST2ER0,_ST1ER1_1_2_1_1_0_NT1ER0
141	LDC_NT2ER0,_c_[XxX0]_[XxX1]_0_0_1_0_2_XXXX
142	LD.E.64_NT2ER0,_[NT2ER1]_0_0_2_0_0_XXXX
143	LD.E.64_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_0_1_XXXX
144	LD.E.CG.64_NT2ER0,_[NT2ER1]_0_0_2_0_0_XXXX
145	LD.E.CG.64_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_0_1_XXXX
146	LD.E.CG_NT2ER0,_[NT2ER1]_0_0_2_0_0_XXXX
147	LD.E.CG_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_0_1_XXXX

Table 6: Identifiers and abstract forms of few discovered ELF innstructions (Part 4).

ID	ABSTRACT TEXT FORM REPRESENTATIONS
148	LD.E.CG.U16_NT2ER0,_[NT2ER1]_0_0_2_0_0_XXXX
149	LD.E.CG.U16_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_0_1_XXXX
150	LD.E.CS.64_NT2ER0,_[NT2ER1]_0_0_2_0_0_XXXX
151	LD.E.CS.64_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_0_1_XXXX
152	LD.E.CS_NT2ER0,_[NT2ER1]_0_0_2_0_0_XXXX
153	LD.E.CS_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_0_1_XXXX
154	LD.E.CS.U16_NT2ER0,_[NT2ER1]_0_0_2_0_0_XXXX
155	LD.E.CS.U16_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_0_1_XXXX
156	LD.E.CV.64_NT2ER0,_[NT2ER1]_0_0_2_0_0_XXXX
157	LD.E.CV.64_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_0_1_XXXX
158	LD.E.CV_NT2ER0,_[NT2ER1]_0_0_2_0_0_XXXX
159	LD.E.CV_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_0_1_XXXX
160	LD.E.CV.U16_NT2ER0,_[NT2ER1]_0_0_2_0_0_XXXX
161	LD.E.CV.U16_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_0_1_XXXX
162	LD.E_NT2ER0,_[NT2ER1]_0_0_2_0_0_XXXX
163	LD.E_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_0_1_XXXX
164	LD.E.U16_NT2ER0,_[NT2ER1]_0_0_2_0_0_XXXX
165	LD.E.U16_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_0_1_XXXX
166	LONT1ER0I.AND_NT2ER0,_NT2ER1,_-XxX0_0_0_2_1_1_XXXX
167	LONT1ER0I.AND_NT2ER0,_NT2ER1,_XxX0_0_0_2_1_1_XXXX
168	LONT1ER0I.OR_NT2ER0,_NT2ER1,_XxX0_0_0_2_1_1_XXXX
169	LOP.AND_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
170	LOP.OR_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
171	LOP.PASS_B_NT2ER0,_ST2ER0,_ NT2ER1_1_0_2_0_0_NT2ER0
172	LOP.XOR_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
173	MEMBAR.CTA_0_0_0_0_0_XXXX
174	MEMBAR.GL_0_0_0_0_0_XXXX
175	MEMBAR.SYS_0_0_0_0_0_XXXX
176	MOV32I_NT2ER0,_XxX0_0_0_1_0_1_XXXX
177	MOV32I_NT2ER0,_-XxX0_0_0_1_0_1_XXXX
178	MOV_NT2ER0,_c_[XxX0]_[XxX1]_0_0_1_0_2_XXXX
179	MOV_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
180	MOV_NT2ER0,_ST2ER0_1_0_1_0_0_NT2ER0
181	MUFU.EX2_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
182	MUFU.LG2_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
183	MUFU.RCNT1ER0H_NT2ER0,_NT2ER1_0_0_2_1_0_NT1ER0
184	MUFU.RCP_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
185	MUFU.RSQ_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
186	MUFU.SIN_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
187	@!NT1ER0_DMUL_NT2ER0,_NT2ER1,_XxX0_0_0_2_1_1_XXXX
188	@NT1ER0_FADD_NT2ER0,_NT2ER1,_XxX0_0_0_2_1_1_XXXX
189	@NT1ER0_FMUL.FTZ_NT2ER0,_NT2ER1,_XxX0_0_0_2_1_1_XXXX
190	@NT1ER0_FMUL_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_1_0_NT2ER0
191	@NT1ER0_FMUL_NT2ER0,_NT2ER1,_XxX0_0_0_2_1_1_XXXX
192	@NT1ER0_IADD_NT2ER0.CC,_NT2ER1,_-NT2ER2_0_0_3_1_0_NT2ER0
193	@NT1ER0_IADD_NT2ER0.CC,_NT2ER1,_XxX0_0_0_2_1_1_XXXX
194	@NT1ER0_IADD_NT2ER0,_NT2ER1,_-NT2ER2_0_0_3_1_0_NT2ER0
195	@!NT1ER0_IADD_NT2ER0,_NT2ER1,_XxX0_0_0_2_1_1_XXXX
196	@NT1ER0_IADD_NT2ER0,_NT2ER1,_XxX0_0_0_2_1_1_XXXX

Table 7: Identifiers and abstract forms of few discovered ELF innstructions (Part 5).

ID	ABSTRACT TEXT FORM REPRESENTATIONS
197	@NT1ER0_IADD.X_NT2ER0,_NT2ER1,_-NT2ER2_0_0_3_1_0_NT2ER0
198	@NT1ER0_IADD.X_NT2ER0,_NT2ER1,_ST2ER0_1_0_2_1_0_NT2ER0
199	@!NT1ER0_LD.E.64_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
200	@NT1ER0_LD.E.64_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
201	@!NT1ER0_LD.E.64_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
202	@NT1ER0_LD.E.64_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
203	@!NT1ER0_LD.E.CG.64_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
204	@NT1ER0_LD.E.CG.64_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
205	@!NT1ER0_LD.E.CG.64_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
206	@NT1ER0_LD.E.CG.64_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
207	@!NT1ER0_LD.E.CG_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
208	@NT1ER0_LD.E.CG_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
209	@!NT1ER0_LD.E.CG_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
210	@NT1ER0_LD.E.CG_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
211	@!NT1ER0_LD.E.CG.U16_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
212	@NT1ER0_LD.E.CG.U16_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
213	@!NT1ER0_LD.E.CG.U16_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
214	@NT1ER0_LD.E.CG.U16_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
215	@!NT1ER0_LD.E.CS.64_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
216	@NT1ER0_LD.E.CS.64_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
217	@!NT1ER0_LD.E.CS.64_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
218	@NT1ER0_LD.E.CS.64_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
219	@!NT1ER0_LD.E.CS_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
220	LD.E.CS_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
221	@!NT1ER0_LD.E.CS_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
222	@NT1ER0_LD.E.CS_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
223	@!NT1ER0_LD.E.CS.U16_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
224	@NT1ER0_LD.E.CS.U16_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
225	@!NT1ER0_LD.E.CS.U16_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
226	@NT1ER0_LD.E.CS.U16_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
227	@!NT1ER0_LD.E.CV.64_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
228	@NT1ER0_LD.E.CV.64_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
229	@!NT1ER0_LD.E.CV.64_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
230	@NT1ER0_LD.E.CV.64_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
231	@!NT1ER0_LD.E.CV_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
232	@NT1ER0_LD.E.CV_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
233	@!NT1ER0_LD.E.CV_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
234	@NT1ER0_LD.E.CV_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
235	@!NT1ER0_LD.E.CV.U16_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
236	@NT1ER0_LD.E.CV.U16_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
237	@!NT1ER0_LD.E.CV.U16_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
238	@NT1ER0_LD.E.CV.U16_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
239	@!NT1ER0_LD.E_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
240	@NT1ER0_LD.E_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
241	@!NT1ER0_LD.E_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
242	@NT1ER0_LD.E_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
243	@!NT1ER0_LD.E.U16_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
244	@NT1ER0_LD.E.U16_NT2ER0,_[NT2ER1]_0_0_2_1_0_XXXX
245	@!NT1ER0_LD.E.U16_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX

Table 8: Identifiers and abstract forms of few discovered ELF innstructions (Part 6).

ID	ABSTRACT TEXT FORM REPRESENTATIONS
246	@NT1ER0_LD.E.U16_NT2ER0,_[NT2ER1 +XxX0]_0_0_2_1_1_XXXX
247	@!NT1ER0_LOP.PASS_B_NT2ER0,_ST2ER0,_ NT2ER1_1_0_2_1_0_NT2ER0
248	@!NT1ER0_MOV32I_NT2ER0,_-XxX0_0_0_1_1_1_XXXX
249	@!NT1ER0_MOV_NT2ER0,_NT2ER1_0_0_2_1_0_NT2ER0
250	@NT1ER0_MOV_NT2ER0,_ST2ER0_1_0_1_1_0_NT2ER0
251	@NT1ER0_MUFU.RCP_NT2ER0,_NT2ER1_0_0_2_1_0_NT2ER0
252	@!NT1ER0_SHL.W_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_1_0_NT2ER0
253	@!NT1ER0_SHR.W_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_1_0_NT2ER0
254	@!NT1ER0_ST.E.64_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
255	@NT1ER0_ST.E.64_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
256	@!NT1ER0_ST.E.64_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
257	@NT1ER0_ST.E.64_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
258	@!NT1ER0_ST.E.CG.64_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
259	@NT1ER0_ST.E.CG.64_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
260	@!NT1ER0_ST.E.CG.64_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
261	@NT1ER0_ST.E.CG.64_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
262	@!NT1ER0_ST.E.CG_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
263	@NT1ER0_ST.E.CG_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
264	@!NT1ER0_ST.E.CG_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
265	@NT1ER0_ST.E.CG_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
266	@!NT1ER0_ST.E.CG.U16_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
267	@NT1ER0_ST.E.CG.U16_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
268	@!NT1ER0_ST.E.CG.U16_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
269	@NT1ER0_ST.E.CG.U16_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
270	@!NT1ER0_ST.E.CS.64_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
271	@NT1ER0_ST.E.CS.64_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
272	@!NT1ER0_ST.E.CS.64_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
273	@NT1ER0_ST.E.CS.64_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
274	@!NT1ER0_ST.E.CS_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
275	@NT1ER0_ST.E.CS_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
276	@!NT1ER0_ST.E.CS_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
277	@NT1ER0_ST.E.CS_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
278	@!NT1ER0_ST.E.CS.U16_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
279	@NT1ER0_ST.E.CS.U16_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
280	@!NT1ER0_ST.E.CS.U16_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
281	@NT1ER0_ST.E.CS.U16_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
282	@!NT1ER0_ST.E_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
283	@NT1ER0_ST.E_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
284	@NT1ER0_ST.E_[NT2ER0 +XxX0],_NT2ER1_0_0_2_1_1_XXXX
285	@!NT1ER0_ST.E_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
286	@NT1ER0_ST.E_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
287	@!NT1ER0_ST.E.U16_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
288	@NT1ER0_ST.E.U16_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
289	@!NT1ER0_ST.E.U16_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
290	@NT1ER0_ST.E.U16_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
291	@NT1ER0_ST.E.U8_[NT2ER0 +XxX0],_NT2ER1_0_0_2_1_1_XXXX
292	@!NT1ER0_ST.E.WT.64_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
293	@NT1ER0_ST.E.WT.64_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
294	@!NT1ER0_ST.E.WT.64_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX

Table 9: Identifiers and abstract forms of few discovered ELF innstructions (Part 7).

ID	ABSTRACT TEXT FORM REPRESENTATIONS
295	@NT1ER0_ST.E.WT.64_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
296	@!NT1ER0_ST.E.WT_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
297	@NT1ER0_ST.E.WT_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
298	@!NT1ER0_ST.E.WT_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
299	@NT1ER0_ST.E.WT_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
300	@!NT1ER0_ST.E.WT.U16_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
301	@NT1ER0_ST.E.WT.U16_[NT2ER0],_ST2ER0_1_0_1_1_0_XXXX
302	@!NT1ER0_ST.E.WT.U16_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
303	@NT1ER0_ST.E.WT.U16_[NT2ER0 +XxX0],_ST2ER0_1_0_1_1_1_XXXX
304	POPC_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
305	PSET.AND.AND_NT2ER0,_ST1ER0,_NT1ER0,_NT1ER1_0_1_1_2_0_NT2ER0
306	PSETP.AND.AND_NT1ER0,_ST1ER0,_ST1ER1,_!NT1ER1,_!NT1ER2_0_2_0_3_0_NT1ER0
307	PSETP.AND.AND_NT1ER0,_ST1ER0,_ST1ER1,_!NT1ER1,_NT1ER2_0_2_0_3_0_NT1ER0
308	PSETP.AND.AND_NT1ER0,_ST1ER0,_ST1ER1,_NT1ER1,_!NT1ER2_0_2_0_3_0_NT1ER0
309	PSETP.AND.AND_NT1ER0,_ST1ER0,_ST1ER1,_NT1ER1,_NT1ER2_0_2_0_3_0_NT1ER0
310	PSETP.AND.OR_NT1ER0,_ST1ER0,_NT1ER1,_NT1ER2,_!NT1ER3_0_1_0_4_0_NT1ER0
311	PSETP.AND.OR_NT1ER0,_ST1ER0,_NT1ER1,_NT1ER2,_NT1ER3_0_1_0_4_0_NT1ER0
312	PSETP.OR.AND_NT1ER0,_ST1ER0,_NT1ER1,_NT1ER2,_NT1ER3_0_1_0_4_0_NT1ER0
313	PSETP.OR.AND_NT1ER0,_ST1ER0,_ST1ER1,_NT1ER1,_NT1ER2_0_2_0_3_0_NT1ER0
314	R2P_PR,_NT2ER0,_XxX0_0_0_1_0_1_XXXX
315	RET_0_0_0_0_0_XXXX
316	RRO.EX2_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
317	RRO.SINCOS_NT2ER0,_NT2ER1_0_0_2_0_0_NT2ER0
318	S2R_NT2ER0,_SR_ClockHi_0_0_1_0_0_NT2ER0
319	S2R_NT2ER0,_SR_ClockLo_0_0_1_0_0_NT2ER0
320	S2R_NT2ER0,_SR_CTAid_X_0_0_1_0_0_NT2ER0
321	S2R_NT2ER0,_SR_CTAid_Y_0_0_1_0_0_NT2ER0
322	S2R_NT2ER0,_SR_CTAid_Z_0_0_1_0_0_NT2ER0
323	S2R_NT2ER0,_SR_EqMask_0_0_1_0_0_NT2ER0
324	S2R_NT2ER0,_SR_GeMask_0_0_1_0_0_NT2ER0
325	S2R_NT2ER0,_SR_GridParam_0_0_1_0_0_NT2ER0
326	S2R_NT2ER0,_SR_GtMask_0_0_1_0_0_NT2ER0
327	S2R_NT2ER0,_SR_LaneId_0_0_1_0_0_NT2ER0
328	S2R_NT2ER0,_SR_LeMask_0_0_1_0_0_NT2ER0
329	S2R_NT2ER0,_SR_LtMask_0_0_1_0_0_NT2ER0
330	S2R_NT2ER0,_SR_Tid_X_0_0_1_0_0_NT2ER0
331	S2R_NT2ER0,_SR_Tid_Y_0_0_1_0_0_NT2ER0
332	S2R_NT2ER0,_SR_Tid_Z_0_0_1_0_0_NT2ER0
333	S2R_NT2ER0,_SR_VirtCfg_0_0_1_0_0_NT2ER0
334	S2R_NT2ER0,_SR_VirtId_0_0_1_0_0_NT2ER0
335	SEL_NT2ER0,_NT2ER1,_c_[XxX0]_[XxX1],_!NT1ER0_0_0_2_1_2_XXXX
336	SEL_NT2ER0,_NT2ER1,_c_[XxX0]_[XxX1],_NT1ER0_0_0_2_1_2_XXXX
337	SEL_NT2ER0,_NT2ER1,_NT2ER2,_!NT1ER0_0_0_3_1_0_NT2ER0
338	SEL_NT2ER0,_NT2ER1,_NT2ER2,_NT1ER0_0_0_3_1_0_NT2ER0
339	SEL_NT2ER0,_NT2ER1,_XxX0,_!NT1ER0_0_0_2_1_1_XXXX
340	SEL_NT2ER0,_NT2ER1,_XxX0,_NT1ER0_0_0_2_1_1_XXXX
341	SEL_NT2ER0,_ST2ER0,_NT2ER1,_!NT1ER0_1_0_2_1_0_NT2ER0
342	SEL_NT2ER0,_ST2ER0,_NT2ER1,_NT1ER0_1_0_2_1_0_NT2ER0
343	SEL_NT2ER0,_ST2ER0,_ST2ER1,_NT1ER0_2_0_1_1_0_NT2ER0

Table 10: Identifiers and abstract forms of few discovered ELF innstructions (Part 8).

ID	ABSTRACT TEXT FORM REPRESENTATIONS
344	SEL_NT2ER0,_ST2ER0,_XxX0,_!NT1ER0_1_0_1_1_1_XXXX
345	SEL_NT2ER0,_ST2ER0,_XxX0,_NT1ER0_1_0_1_1_1_XXXX
346	SHL_NT2ER0,_NT2ER1,_XxX0_0_0_2_0_1_XXXX
347	SHL.W_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
348	SHL.W_NT2ER0,_NT2ER1,_XxX0_0_0_2_0_1_XXXX
349	SHR.U32.W_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
350	SHR.U32.W_NT2ER0,_NT2ER1,_XxX0_0_0_2_0_1_XXXX
351	SHR.W_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0
352	ST.E.64_[NT2ER0],_NT2ER1_0_0_2_0_0_XXXX
353	ST.E.64_[NT2ER0],_ST2ER0_1_0_1_0_0_XXXX
354	ST.E.64_[NT2ER0 +XxX0],_NT2ER1_0_0_2_0_1_XXXX
355	ST.E.64_[NT2ER0 +XxX0],_ST2ER0_1_0_1_0_1_XXXX
356	ST.E.CG.64_[NT2ER0],_ST2ER0_1_0_1_0_0_XXXX
357	ST.E.CG.64_[NT2ER0 +XxX0],_ST2ER0_1_0_1_0_1_XXXX
358	ST.E.CG_[NT2ER0],_ST2ER0_1_0_1_0_0_XXXX
359	ST.E.CG_[NT2ER0 +XxX0],_ST2ER0_1_0_1_0_1_XXXX
360	ST.E.CG.U16_[NT2ER0],_ST2ER0_1_0_1_0_0_XXXX
361	ST.E.CG.U16_[NT2ER0 +XxX0],_ST2ER0_1_0_1_0_1_XXXX
362	ST.E.CS.64_[NT2ER0],_ST2ER0_1_0_1_0_0_XXXX
363	ST.E.CS.64_[NT2ER0 +XxX0],_ST2ER0_1_0_1_0_1_XXXX
364	ST.E.CS_[NT2ER0],_ST2ER0_1_0_1_0_0_XXXX
365	ST.E.CS_[NT2ER0 +XxX0],_ST2ER0_1_0_1_0_1_XXXX
366	ST.E.CS.U16_[NT2ER0],_ST2ER0_1_0_1_0_0_XXXX
367	ST.E.CS.U16_[NT2ER0 +XxX0],_ST2ER0_1_0_1_0_1_XXXX
368	ST.E_[NT2ER0],_NT2ER1_0_0_2_0_0_XXXX
369	ST.E_[NT2ER0],_ST2ER0_1_0_1_0_0_XXXX
370	ST.E_[NT2ER0 +XxX0],_NT2ER1_0_0_2_0_1_XXXX
371	ST.E_[NT2ER0 +XxX0],_ST2ER0_1_0_1_0_1_XXXX
372	ST.E.U16_[NT2ER0],_ST2ER0_1_0_1_0_0_XXXX
373	ST.E.U16_[NT2ER0 +XxX0],_NT2ER1_0_0_2_0_1_XXXX
374	ST.E.U16_[NT2ER0 +XxX0],_ST2ER0_1_0_1_0_1_XXXX
375	ST.E.U8_[NT2ER0],_NT2ER1_0_0_2_0_0_XXXX
376	ST.E.WT.64_[NT2ER0],_ST2ER0_1_0_1_0_0_XXXX
377	ST.E.WT.64_[NT2ER0 +XxX0],_ST2ER0_1_0_1_0_1_XXXX
378	ST.E.WT_[NT2ER0],_ST2ER0_1_0_1_0_0_XXXX
379	ST.E.WT_[NT2ER0 +XxX0],_ST2ER0_1_0_1_0_1_XXXX
380	ST.E.WT.U16_[NT2ER0],_ST2ER0_1_0_1_0_0_XXXX
381	ST.E.WT.U16_[NT2ER0 +XxX0],_ST2ER0_1_0_1_0_1_XXXX
382	VADD.ACC_NT2ER0,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0

Table 11: Instruction and bit identifiers of few discovered ELF innstructions (Part 1 - Bits 63-32).

ID	Bits																														
	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
001	-	+	-	+	-	-	-	-	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
002	+	-	+	-	-	-	-	+	+	+	-	+	+	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+
003	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	+	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2
004	+	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0
005	-	+	-	+	-	0	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
006	-	+	-	-	+	-	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
007	-	-	+	-	-	-	-	-	-	3	3	3	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
008	-	-	+	-	-	-	-	-	-	3	3	3	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
009	-	-	+	-	-	-	-	-	+	3	3	3	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
010	-	-	+	-	-	-	-	+	-	3	3	3	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
011	-	-	+	-	-	-	-	+	+	4	4	4	4	4	4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
012	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
013	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	+	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2
014	-	-	-	+	+	-	-	+	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
015	-	-	-	+	+	-	-	+	-	-	-	-	+	+	+	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
016	-	-	-	+	+	-	-	+	+	-	-	-	+	+	+	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
017	-	-	-	+	+	-	+	-	+	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
018	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
019	-	-	-	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
020	-	-	-	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
021	-	-	-	+	-	+	-	-	+	-	-	-	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
022	-	-	-	+	-	+	-	-	-	-	-	-	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
023	-	-	-	+	-	+	-	-	-	-	-	-	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
024	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	+	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2
025	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
026	-	-	+	+	-	-	-	-	-	2	2	2	2	2	2	-	+	-	3	3	3	3	4	4	4	4	4	4	4	4	4	4
027	-	-	+	+	-	-	-	-	-	2	2	2	2	2	2	-	+	-	3	3	3	3	4	4	4	4	4	4	4	4	4	4
028	-	-	+	+	-	-	-	-	-	3	3	3	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
029	-	-	+	+	-	-	-	-	-	3	3	3	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
030	-	-	+	+	-	-	-	-	-	+	+	+	+	+	+	-	+	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2
031	-	-	+	+	-	-	-	-	+	3	3	3	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
032	-	-	+	+	-	-	-	-	+	3	3	3	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
033	-	-	+	+	-	-	-	+	-	2	2	2	2	2	2	-	+	-	3	3	3	3	4	4	4	4	4	4	4	4	4	4
034	-	-	+	+	-	-	-	+	-	3	3	3	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
035	-	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
036	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
037	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	+	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2
038	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
039	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	+	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2
040	-	+	-	+	+	-	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
041	-	-	+	-	+	-	-	+	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
042	-	-	+	-	+	-	-	+	-	-	-	-	+	+	+	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
043	-	-	+	-	-	-	+	-	-	-	-	-	+	+	+	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
044	-	-	+	-	+	-	-	+	+	-	-	-	+	+	+	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
045	-	-	+	-	-	-	-	-	+	-	-	-	+	+	+	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
046	-	-	+	-	-	-	-	-	+	-	-	-	+	+	+	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
047	-	-	+	-	+	-	-	-	+	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
048	-	-	+	-	+	+	+	-	+	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
049	-	-	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 12: Instruction and bit identifiers of few discovered ELF innstructions (Part 1 - Bits 31-00).

ID	Bits																														
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
001	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	-	+	+	+	-	-	-	-	-	-	-	+	-	-
002	+	+	+	+	+	1	1	1	1	1	1	+	+	+	+	+	+	-	+	+	+	-	-	-	-	-	-	-	+	-	-	-
003	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
004	0	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+
005	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+
006	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	+	+	+	-	-	-	-	-	-	-	-	-	+
007	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	-	-	-	-	-	-	-	-	+
008	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	+
009	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	+
010	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	+
011	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	+	+	+	-	-	-	-	-	-	-	-	-	+
012	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	+
013	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	+
014	+	+	+	+	+	+	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	-	+
015	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	+	-	-	-	-	-	-	+
016	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	+	-	-	-	-	-	-	+
017	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	-	+
018	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	-	+	+	+	+	-	-	+	+	+
019	1	1	1	1	1	1	-	+	-	-	+	-	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	+	-	-
020	1	1	1	1	1	1	-	+	-	-	+	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
021	1	1	1	1	1	1	-	+	-	-	+	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
022	1	1	1	1	1	1	-	+	-	-	+	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
023	1	1	1	1	1	1	-	+	-	-	+	+	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
024	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	+	-	-	-	-	-
025	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	-
026	4	4	4	4	-	+	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	-	-	-	-	-	-	-	-	-
027	4	4	4	4	-	+	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	-
028	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	-	-	-	-	-	-	-	-	-
029	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	-
030	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	-
031	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	-	-	-	-	-	-	-	-	-
032	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	-
033	4	4	4	4	-	+	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	-
034	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	-	-	-	-	-	-	-	-	-
035	1	1	1	1	1	1	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
036	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	-	-	-
037	2	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	-	-
038	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	-
039	2	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-
040	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	+	+	+	-	-	-	-	-	-	-	-	-	-
041	+	+	+	+	+	+	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	-	-
042	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-
043	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	+	-	-	-	-	-	-	-
044	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	+	-	-	-	-	-	-	-
045	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	+	-	-	-	-	-	-	-
046	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	-	-
047	+	+	+	+	+	+	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	-	-
048	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	-	-
049	1	1	1	1	1	1	-	+	-	-	+	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-

Table 13: Instruction and bit identifiers of few discovered ELF innstructions (Part 2 - Bits 63-32).

ID	Bits																															
	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
050	-	-	-	+	+	-	-	-	-	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
051	-	-	-	+	+	-	-	-	-	-	-	-	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
052	-	-	-	+	+	-	-	-	-	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
053	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
054	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
055	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
056	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
057	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
058	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
059	-	-	-	-	+	-	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
060	-	-	-	-	+	-	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
061	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	+	-	+	2	2	2	2	3	3	3	3	3	3	3	3	3	3
062	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
063	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
064	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
065	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	+	+	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2
066	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
067	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
068	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	+	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2
069	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	+	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2
070	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
071	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
072	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
073	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	+	2	2	2	2	3	3	3	3	3	3	3	3	3	3
074	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
075	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
076	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
077	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
078	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	+	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2
079	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	+	2	2	2	2	3	3	3	3	3	3	3	3	3	3
080	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
081	-	-	+	+	-	-	-	+	-	4	4	4	4	4	4	-	-	+	2	2	2	2	3	3	3	3	3	3	3	3	3	3
082	-	-	+	+	-	-	-	+	-	3	3	3	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
083	-	-	+	+	-	-	-	-	+	3	3	3	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
084	-	-	+	-	-	-	-	-	-	3	3	3	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
085	-	-	+	-	-	-	-	-	+	3	3	3	3	3	3	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
086	-	-	+	-	-	-	-	-	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
087	-	-	+	-	-	-	-	-	+	3	3	3	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
088	-	-	+	-	-	-	-	-	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
089	-	-	+	-	-	-	-	-	-	3	3	3	3	3	3	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
090	-	-	+	-	-	-	-	-	-	3	3	3	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
091	-	-	+	-	-	-	-	-	-	3	3	3	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
092	-	-	+	-	-	-	-	-	-	3	3	3	3	3	3	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
093	-	-	+	-	-	-	-	-	-	+	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
094	-	-	+	-	-	-	-	-	-	+	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
095	-	-	+	-	-	-	-	-	+	3	3	3	3	3	3	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
096	-	-	+	-	-	-	-	-	+	3	3	3	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
097	-	-	-	-	+	-	-	-	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
098	-	-	-	-	+	-	-	-	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 14: Instruction and bit identifiers of few discovered ELF innstructions (Part 2 - Bits 31-00).

ID	Bits																														
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
050	1	1	1	1	1	1	-	+	-	-	+	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
051	2	2	2	2	2	2	-	+	-	-	+	-	1	1	1	1	1	1	-	+	+	+	-	-	-	-	-	-	-	+	-	-
052	1	1	1	1	1	1	-	+	+	-	+	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
053	1	1	1	1	1	1	-	-	+	-	+	-	0	0	0	0	0	0	-	+	+	+	+	-	+	-	-	-	-	+	-	-
054	1	1	1	1	1	1	-	+	-	-	+	-	0	0	0	0	0	0	-	+	+	+	+	-	+	+	-	-	-	+	-	-
055	1	1	1	1	1	1	-	+	-	-	+	-	0	0	0	0	0	0	-	+	+	+	+	+	+	-	-	-	-	+	-	-
056	1	1	1	1	1	1	-	+	-	-	-	+	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
057	1	1	1	1	1	1	-	-	+	-	+	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
058	1	1	1	1	1	1	-	-	-	-	+	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
059	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	-
060	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	-
061	3	3	3	3	-	-	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
062	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	+	-	-	-	-	-	-	+	+
063	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
064	+	+	+	+	+	+	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	-	-	-	-	-	-	-	+	+
065	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
066	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	+	-	-	-	-	-	-	+	+
067	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
068	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	-	-	-	-	-	-	-	+	+
069	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
070	2	2	2	2	2	2	+	+	+	+	+	+	0	0	0	0	0	0	-	+	+	+	-	+	-	-	-	-	-	-	+	+
071	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	-	+	+	+	-	+	-	-	-	-	-	-	+	+
072	+	+	+	+	+	+	1	1	1	1	1	1	+	+	+	+	+	+	-	+	+	+	-	+	-	-	-	-	-	-	+	+
073	3	3	3	3	3	-	-	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	-	+
074	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	+	-	+	-	-	-	-	+	+
075	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	-	+	+
076	+	+	+	+	+	+	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	-	-	+	-	-	-	-	+	+
077	+	+	+	+	+	+	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	-	+	+
078	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	-	+	+
079	3	3	3	3	3	-	-	+	+	+	+	+	+	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	-	+
080	+	+	+	+	+	+	+	+	+	+	+	+	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	-	+	+
081	3	3	3	3	3	-	+	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	+	-	-	-	+
082	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
083	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	+	-	-	-	+	+
084	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	-	+	+
085	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	-	+	+
086	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	-	+	+
087	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	-	+	+
088	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	-	+	+
089	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
090	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	-	-	-	-	-	-	-	+	+
091	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
092	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	+	+
093	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	+	+
094	+	+	+	+	+	+	1	1	1	1	1	1	+	+	+	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	+	+
095	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
096	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
097	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	+	-	-	-	+	+
098	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	+	-	-	-	+	+

Table 15: Instruction and bit identifiers of few discovered ELF innstructions (Part 3 - Bits 63-32).

ID	Bits																														
	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
099	-	-	-	-	+	-	-	-	-	-	-	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
100	-	-	-	-	+	-	-	-	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
101	-	-	-	-	+	-	-	-	-	-	-	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
102	-	-	-	-	+	-	-	-	-	-	-	-	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
103	-	-	-	-	+	-	-	-	-	-	-	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
104	-	-	-	-	+	-	-	-	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
105	-	-	-	-	+	-	-	-	-	-	-	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
106	-	-	-	-	+	-	-	-	-	-	-	-	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
107	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
108	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
109	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
110	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
111	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
112	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
113	-	+	-	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
114	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
115	-	-	-	+	+	-	-	+	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
116	-	-	-	+	+	-	-	+	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
117	-	-	-	+	+	-	-	+	-	-	-	-	4	4	4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
118	-	-	-	+	+	-	-	+	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
119	-	-	-	+	+	-	+	+	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
120	-	-	-	+	+	-	+	+	-	-	-	-	+	+	+	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
121	-	-	-	+	+	-	+	+	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
122	-	-	-	+	+	-	+	+	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
123	-	-	-	+	+	-	+	-	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
124	-	-	-	+	+	-	+	-	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
125	-	-	-	+	+	-	+	-	-	-	-	-	+	+	+	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
126	-	-	-	+	+	-	+	-	-	-	+	-	4	4	4	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
127	-	-	-	+	+	-	+	-	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
128	-	-	-	+	+	-	+	-	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
129	-	-	-	+	+	-	-	+	+	-	-	-	+	+	+	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
130	-	-	-	+	+	-	-	+	+	-	-	-	+	+	+	-	-	+	3	3	3	3	4	4	4	4	4	4	4	4	4	4
131	-	-	-	+	+	-	-	+	+	-	-	-	+	+	+	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
132	-	-	-	+	+	-	-	-	+	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
133	-	-	-	+	+	-	-	-	+	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
134	-	-	-	+	+	-	-	-	+	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
135	-	-	-	+	+	-	-	-	+	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
136	-	-	-	+	+	-	+	-	+	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
137	-	-	-	+	+	-	+	-	+	-	-	-	4	4	4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
138	-	-	-	+	+	-	+	-	+	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
139	-	-	-	+	+	-	+	-	+	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
140	-	-	-	+	+	-	+	-	+	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
141	-	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2
142	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
143	+	-	-	-	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
144	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
145	+	-	-	-	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
146	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
147	+	-	-	-	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

Table 16: Instruction and bit identifiers of few discovered ELF innstructions (Part 3 - Bits 31-00).

ID	Bits																														
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
099	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
100	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
101	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	+	+	-	-	-	-	+	+
102	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	+	+	-	-	-	-	+	+
103	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	-	+	+
104	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	-	+	+
105	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	+	+	+	-	-	-	+	+
106	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	+	+	+	-	-	-	+	+
107	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	+	+	+	-	-	-	+	+
108	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	+	-	+	-	-	-	+	+
109	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	-	+	+
110	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
111	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
112	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	+	+
113	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	3	3	3	3	3	-	-	-	+	+
114	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	3	3	3	3	3	-	-	-	+	+
115	+	+	+	+	+	+	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	+	-	-	-	+	+
116	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	+	+
117	+	+	+	+	+	+	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	+	+
118	+	+	+	+	+	+	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	+	+
119	+	+	+	+	+	+	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	+	-	-	-	+	+
120	3	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	+	-	-	-	+
121	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	+	+
122	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	+	-	-	-	-	+	+
123	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	+	-	-	-	+	+
124	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	+	+
125	3	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	+
126	3	3	3	3	3	3	3	2	2	2	+	+	+	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	+
127	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	+	-	-	-	-	+	+
128	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	+	+	-	-	-	+	+
129	3	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	+	-	-	-	+
130	4	4	4	4	-	+	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	+	+
131	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	+	+
132	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	+	-	-	-	+	+
133	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	+	+
134	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	+	-	-	-	-	+	+
135	3	3	3	3	3	3	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	+	+	-	-	-	+	+
136	+	+	+	+	+	+	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	+	-	-	-	+	+
137	+	+	+	+	+	+	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	+	+
138	+	+	+	+	+	+	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	+	+
139	+	+	+	+	+	+	2	2	2	2	2	2	0	0	0	+	+	+	-	+	+	+	-	-	-	+	-	-	-	-	+	+
140	+	+	+	+	+	+	2	2	2	2	2	2	-	0	0	0	+	+	-	+	+	+	-	-	-	+	+	-	-	-	+	+
141	2	2	2	2	2	2	+	+	+	+	+	+	0	0	0	0	0	0	-	+	+	+	-	-	+	-	-	-	-	+	+	-
142	-	-	-	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	+	-	+	-	-	-	+	-
143	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	+	-	+	-	-	+	-	+
144	-	-	-	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	+	+	-	+	-	-	+	-	+
145	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	+	+	-	+	-	-	+	-	+
146	-	-	-	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	+	+	-	-	-	-	+	-	+
147	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	+	+	-	-	-	-	+	-	+

Table 17: Instruction and bit identifiers of few discovered ELF innstructions (Part 4 - Bits 63-32).

ID	Bits																																
	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
148	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
149	+	-	-	-	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
150	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
151	+	-	-	-	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
152	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
153	+	-	-	-	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
154	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
155	+	-	-	-	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
156	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
157	+	-	-	-	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
158	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
159	+	-	-	-	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
160	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
161	+	-	-	-	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
162	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
163	+	-	-	-	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
164	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
165	+	-	-	-	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
166	-	-	+	+	+	-	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
167	-	-	+	+	+	-	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
168	-	-	+	+	+	-	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
169	-	+	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
170	-	+	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
171	-	+	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
172	-	+	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
173	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
174	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
175	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
176	-	-	-	+	+	-	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
177	-	-	-	+	+	-	+	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
178	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	+	1	1	1	1	2	2	2	2	2	2	2	2	2	2
179	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
180	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
181	+	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
182	+	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
183	+	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
184	+	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
186	+	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
187	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
188	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
189	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
190	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
191	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
192	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
193	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	+	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
194	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
195	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3
196	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	+	+	3	3	3	3	3	3	3	3	3	3	3	3	3	3

Table 18: Instruction and bit identifiers of few discovered ELF innstructions (Part 4 - Bits 31-00).

ID	Bits																														
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
148	-	-	-	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	+	-	+	-	-	-	+	-	+
149	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	+	-	+	-	-	-	+	-	+
150	-	-	-	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	-	+	-	+	-	-	+	-	+
151	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	-	+	-	+	-	-	+	-	+
152	-	-	-	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	-	+	-	-	-	-	+	-	+
153	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	-	+	-	-	-	-	+	-	+
154	-	-	-	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	-	-	+	-	-	-	+	-	+
155	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	-	-	+	-	-	-	+	-	+
156	-	-	-	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	+	+	-	+	-	-	+	-	+
157	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	+	+	-	+	-	-	+	-	+
158	-	-	-	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	+	+	-	-	-	-	+	-	+
159	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	+	+	-	-	-	-	+	-	+
160	-	-	-	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	+	-	+	-	-	-	+	-	+
161	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	+	-	+	-	-	-	+	-	+
162	-	-	-	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	+	-	-	-	-	+	-	+
163	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	+	-	-	-	-	+	-	+
164	-	-	-	-	-	-	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	+	-	+
165	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	+	-	+
166	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	+	+	+	-	-	-	-	-	-	-	-	+	-
167	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	+	+	+	-	-	-	-	-	-	-	-	+	-
168	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	+	+	+	-	-	-	+	-	-	-	-	+	-
169	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
170	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	-	-	-	-	+	+
171	2	2	2	2	2	2	+	+	+	+	+	+	0	0	0	0	0	0	-	+	+	+	-	+	+	+	-	-	-	-	+	+
172	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	+	-	-	-	-	-	+	+
173	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	-	-	-	-	-	-	-	+	-	+
174	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	-	-	-	-	+	-	-	+	-	+
175	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	-	-	-	+	-	-	-	+	-	+
176	1	1	1	1	1	1	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	+	+	+	+	-	-	-	+	-
177	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	+	+	+	+	-	-	-	+	-
178	2	2	2	2	-	-	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	+	+	+	+	-	-	+	-	-
179	1	1	1	1	1	1	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	+	+	+	+	-	-	+	-	-
180	+	+	+	+	+	+	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	+	+	+	+	-	-	+	-	-
181	-	-	-	-	+	-	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	-
182	-	-	-	-	+	+	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	-
183	-	-	-	+	+	-	2	2	2	2	2	2	1	1	1	1	1	1	-	+	+	+	-	-	-	-	-	-	-	-	-	-
184	-	-	-	+	-	-	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	-
185	-	-	-	+	-	+	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	-
186	-	-	-	-	-	+	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	-
187	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	-	-	-	-	-	-	-	-	-	+
188	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	-	-	-	-	-	-	-	-	-
189	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	-	-	+	-	-	-	-	-	-
190	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	-	-	-	-	-	-	-	-	-
191	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	-	-	-	-	-	-	-	-	-
192	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	+	-	-	-	-	-	-	+	+
193	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	-	-	-	-	-	-	-	+	+
194	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	+	-	-	-	-	-	-	+	+
195	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	-	-	-	-	-	-	-	-	+	+
196	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	-	-	-	-	-	-	-	+	+

Table 19: Instruction and bit identifiers of few discovered ELF innstructions (Part 5 - Bits 63-32).

ID	Bits																														
	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
197	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
198	-	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
199	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
200	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
201	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
202	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
203	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
204	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
205	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
206	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
207	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
208	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
209	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
210	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
211	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
212	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
213	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
214	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
215	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
216	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
217	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
218	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
219	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
220	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
221	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
222	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
223	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
224	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
225	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
226	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
227	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
228	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
229	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
230	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
231	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
232	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
233	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
234	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
235	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
236	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
237	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
238	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
239	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
240	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
241	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
242	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
243	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
244	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
245	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

Table 20: Instruction and bit identifiers of few discovered ELF innstructions (Part 5 - Bits 31-00).

ID	Bits																														
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
197	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	+	-	+	-	-	-	-	+	+
198	+	+	+	+	+	+	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	-	-	+	-	-	-	-	+	+
199	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	-	-	+	-	+	-	-	+	-	+
200	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	-	+	-	+	-	-	+	-	+
201	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	-	-	+	-	+	-	-	+	-	+
202	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	-	+	-	+	-	-	+	-	+
203	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	-	+	+	-	+	-	-	+	-	+
204	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	+	+	-	+	-	-	+	-	+
205	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	-	+	+	-	+	-	-	+	-	+
206	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	+	+	-	+	-	-	+	-	+
207	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	-	+	+	-	-	-	-	+	-	+
208	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	+	+	-	-	-	-	+	-	+
209	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	-	+	+	-	-	-	-	+	-	+
210	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	+	+	-	-	-	-	+	-	+
211	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	-	+	-	+	-	-	-	+	-	+
212	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	+	-	+	-	-	-	+	-	+
213	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	-	+	-	+	-	-	-	+	-	+
214	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	+	-	+	-	-	-	+	-	+
215	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	+	-	+	-	+	-	-	+	-	+
216	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	+	-	+	-	+	-	-	+	-	+
217	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	+	-	+	-	+	-	-	+	-	+
218	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	+	-	+	-	+	-	-	+	-	+
219	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	+	-	+	-	-	-	-	+	-	+
220	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	+	-	+	-	-	-	-	+	-	+
221	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	+	-	+	-	-	-	-	+	-	+
222	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	+	-	+	-	-	-	-	+	-	+
223	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	+	-	-	+	-	-	-	+	-	+
224	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	+	-	-	+	-	-	-	+	-	+
225	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	+	-	-	+	-	-	-	+	-	+
226	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	+	-	-	+	-	-	-	+	-	+
227	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	+	+	+	-	+	-	-	+	-	+
228	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	+	+	+	-	+	-	-	+	-	+
229	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	+	+	+	-	+	-	-	+	-	+
230	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	+	+	+	-	+	-	-	+	-	+
231	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	+	+	+	-	-	-	-	+	-	+
232	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	+	+	+	-	-	-	-	+	-	+
233	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	+	+	+	-	-	-	-	+	-	+
234	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	+	+	+	-	-	-	-	+	-	+
235	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	+	+	-	+	-	-	-	+	-	+
236	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	+	+	-	+	-	-	-	+	-	+
237	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	+	+	-	+	-	-	-	+	-	+
238	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	+	+	-	+	-	-	-	+	-	+
239	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	-	-	+	-	-	-	-	+	-	+
240	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	-	+	-	-	-	-	+	-	+
241	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	-	-	+	-	-	-	-	+	-	+
242	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	-	+	-	-	-	-	+	-	+
243	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	-	-	-	+	-	-	-	+	-	+
244	-	-	-	-	-	-	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	-	-	+	-	-	-	+	-	+
245	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	-	-	-	+	-	-	-	+	-	+

Table 21: Instruction and bit identifiers of few discovered ELF innstructions (Part 6 - Bits 63-32).

ID	Bits																															
	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
246	+	-	-	-	-	+	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
247	-	+	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
248	-	-	-	+	+	-	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
249	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
250	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
251	+	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
252	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
253	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
254	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
255	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
256	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
257	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
258	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
259	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
260	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
261	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
262	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
263	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
264	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
265	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
266	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
267	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
268	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
269	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
270	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
271	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
272	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
273	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
274	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
275	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
276	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
277	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
278	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
279	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
280	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
281	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
282	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
283	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
284	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
285	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
286	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
287	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
288	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
289	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
290	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
291	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
292	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
293	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
294	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

Table 22: Instruction and bit identifiers of few discovered ELF innstructions (Part 6 - Bits 31-00).

ID	Bits																														
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
246	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	-	-	+	-	-	-	+	-	+
247	3	3	3	3	3	3	+	+	+	+	+	+	1	1	1	1	1	1	+	0	0	0	-	+	+	+	-	-	-	-	+	+
248	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	-	+	+	+	+	-	-	-	+	-
249	2	2	2	2	2	2	-	-	-	-	-	-	1	1	1	1	1	1	+	0	0	0	-	+	+	+	+	-	-	+	-	-
250	+	+	+	+	+	+	-	-	-	-	-	-	1	1	1	1	1	1	-	0	0	0	-	+	+	+	+	-	-	+	-	-
251	-	-	-	+	-	-	2	2	2	2	2	2	1	1	1	1	1	1	-	0	0	0	-	-	-	-	-	-	-	-	-	-
252	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	+	0	0	0	-	-	-	-	-	-	-	-	+	+
253	3	3	3	3	3	3	2	2	2	+	+	+	1	1	1	1	1	1	+	0	0	0	-	-	-	-	+	-	-	-	+	+
254	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	-	-	+	-	+	-	-	+	-	+
255	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	-	-	+	-	+	-	-	+	-	+
256	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	-	-	+	-	+	-	-	+	-	+
257	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	-	-	+	-	+	-	-	+	-	+
258	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	-	+	+	-	+	-	-	+	-	+
259	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	-	+	+	-	+	-	-	+	-	+
260	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	-	+	+	-	+	-	-	+	-	+
261	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	-	+	+	-	+	-	-	+	-	+
262	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	-	+	+	-	-	-	-	+	-	+
263	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	-	+	+	-	-	-	-	+	-	+
264	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	-	+	+	-	-	-	-	+	-	+
265	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	-	+	+	-	-	-	-	+	-	+
266	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	-	+	-	+	-	-	-	+	-	+
267	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	-	+	-	+	-	-	-	+	-	+
268	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	-	+	-	+	-	-	-	+	-	+
269	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	-	+	-	+	-	-	-	+	-	+
270	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	+	-	+	-	+	-	-	+	-	+
271	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	+	-	+	-	+	-	-	+	-	+
272	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	+	-	+	-	+	-	-	+	-	+
273	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	+	-	+	-	+	-	-	+	-	+
274	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	+	-	+	-	-	-	-	+	-	+
275	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	+	-	+	-	-	-	-	+	-	+
276	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	+	-	+	-	-	-	-	+	-	+
277	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	+	-	+	-	-	-	-	+	-	+
278	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	+	-	-	+	-	-	-	+	-	+
279	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	+	-	-	+	-	-	-	+	-	+
280	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	+	-	-	+	-	-	-	+	-	+
281	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	+	-	-	+	-	-	-	+	-	+
282	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	-	-	-	+	-	-	-	+	-	+
283	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	-	-	-	+	-	-	-	+	-	+
284	2	2	2	2	2	2	1	1	1	1	1	1	3	3	3	3	3	3	-	0	0	0	-	-	-	+	-	-	-	+	-	+
285	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	-	-	-	+	-	-	-	+	-	+
286	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	-	-	-	+	-	-	-	+	-	+
287	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	-	-	-	-	+	-	-	+	-	+
288	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	-	-	-	-	+	-	-	+	-	+
289	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	-	-	-	-	+	-	-	+	-	+
290	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	-	-	-	-	+	-	-	+	-	+
291	2	2	2	2	2	2	1	1	1	1	1	1	3	3	3	3	3	3	-	0	0	0	-	-	-	-	-	-	-	+	-	+
292	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	+	+	+	-	+	-	-	+	-	+
293	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	+	+	+	-	+	-	-	+	-	+
294	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	+	+	+	-	+	-	-	+	-	+

Table 23: Instruction and bit identifiers of few discovered ELF innstructions (Part 7- Bits 63-32).

ID	Bits																														
	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
295	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
296	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
297	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
298	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
299	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
300	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
301	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
302	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
303	+	-	-	+	-	+	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
304	-	+	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
305	-	-	-	-	+	-	-	-	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
306	-	-	-	-	+	+	-	-	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
307	-	-	-	-	+	+	-	-	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
308	-	-	-	-	+	+	-	-	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
309	-	-	-	-	+	+	-	-	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
310	-	-	-	-	+	+	-	-	-	-	+	-	-	2	2	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
311	-	-	-	-	+	+	-	-	-	-	+	-	-	2	2	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
312	-	-	-	-	+	+	-	-	-	-	-	-	2	2	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
313	-	-	-	-	+	+	-	-	-	-	-	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
314	-	-	+	+	-	+	-	-	-	-	-	-	-	-	-	-	+	+	1	1	1	1	1	1	1	1	1	1	1	1	1	1
315	+	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
316	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
317	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
318	-	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+
319	-	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+
320	-	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
321	-	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
322	-	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
323	-	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
324	-	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
325	-	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
326	-	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
327	-	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
328	-	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
329	-	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
330	-	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
331	-	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
332	-	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
333	-	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
334	-	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
335	-	-	+	-	-	-	-	-	-	-	-	+	4	4	4	-	-	+	2	2	2	2	3	3	3	3	3	3	3	3	3	3
336	-	-	+	-	-	-	-	-	-	-	-	-	4	4	4	-	-	+	2	2	2	2	3	3	3	3	3	3	3	3	3	3
337	-	-	+	-	-	-	-	-	-	-	-	+	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
338	-	-	+	-	-	-	-	-	-	-	-	-	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
339	-	-	+	-	-	-	-	-	-	-	-	+	3	3	3	-	+	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2
340	-	-	+	-	-	-	-	-	-	-	-	-	3	3	3	-	+	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2
341	-	-	+	-	-	-	-	-	-	-	-	+	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
342	-	-	+	-	-	-	-	-	-	-	-	-	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
343	-	-	+	-	-	-	-	-	-	-	-	-	3	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 24: Instruction and bit identifiers of few discovered ELF innstructions (Part 7 - Bits 31-00).

ID	Bits																														
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
295	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	+	+	+	-	+	-	-	+	-	+
296	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	+	+	+	-	-	-	-	+	-	+
297	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	+	+	+	-	-	-	-	+	-	+
298	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	+	+	+	-	-	-	-	+	-	+
299	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	+	+	+	-	-	-	-	+	-	+
300	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	+	+	-	+	-	-	-	+	-	+
301	-	-	-	-	-	-	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	+	+	-	+	-	-	-	+	-	+
302	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	+	0	0	0	+	+	-	+	-	-	-	+	-	+
303	2	2	2	2	2	2	1	1	1	1	1	1	+	+	+	+	+	+	-	0	0	0	+	+	-	+	-	-	-	+	-	+
304	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
305	-	-	-	2	2	2	-	-	-	3	3	3	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
306	-	-	+	3	3	3	-	-	+	4	4	4	-	0	0	0	+	+	-	+	+	+	-	-	-	-	-	-	-	+	-	-
307	-	-	+	3	3	3	-	-	-	4	4	4	-	0	0	0	+	+	-	+	+	+	-	-	-	-	-	-	-	+	-	-
308	-	-	-	3	3	3	-	-	+	4	4	4	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	+	-	-
309	-	-	-	-	3	3	3	-	-	4	4	4	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	+	-	-
310	-	-	-	3	3	3	-	-	+	4	4	4	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	+	-	-
311	-	-	-	3	3	3	-	-	-	4	4	4	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	+	-	-
312	-	+	-	-	3	3	3	-	-	4	4	4	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	+	-	-
313	-	+	-	-	3	3	3	-	-	4	4	4	0	0	0	+	+	+	-	+	+	+	-	-	-	-	-	-	-	+	-	-
314	1	1	1	1	1	1	0	0	0	0	0	0	-	-	-	-	-	-	-	+	+	+	-	-	-	-	-	-	-	+	-	-
315	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	-	+	+	+	+	-	-	+	+	+
316	1	1	1	1	1	1	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	+	-	-	-	-	-
317	1	1	1	1	1	1	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	-	-
318	-	+	-	-	-	+	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
319	-	+	-	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
320	+	-	-	+	-	+	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
321	+	-	-	+	+	-	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
322	+	-	-	+	+	+	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
323	+	+	+	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
324	+	+	+	+	-	-	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
325	+	-	+	+	-	-	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
326	+	+	+	-	+	+	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
327	-	-	-	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
328	+	+	+	-	+	-	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
329	+	+	+	-	-	+	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
330	+	-	-	-	-	+	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
331	+	-	-	-	+	-	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
332	+	-	-	-	+	+	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
333	-	-	-	-	+	-	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
334	-	-	-	-	+	+	-	-	-	-	-	-	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
335	3	3	3	3	3	-	-	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	-
336	3	3	3	3	3	-	+	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-
337	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
338	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
339	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
340	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
341	2	2	2	2	2	2	+	+	+	+	+	+	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
342	2	2	2	2	2	2	+	+	+	+	+	+	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
343	+	+	+	+	+	+	+	+	+	+	+	+	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-

Table 25: Instruction and bit identifiers of few discovered ELF innstructions (Part 8 - Bits 63-32).

ID	Bits																															
	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
344	-	-	+	-	-	-	-	-	-	-	-	+	3	3	3	-	+	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2
345	-	-	+	-	-	-	-	-	-	-	-	-	3	3	3	-	+	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2
346	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2
347	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
348	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2
349	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
350	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	+	+	2	2	2	2	2	2	2	2	2	2	2	2	2	2
351	-	+	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
352	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
353	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
354	+	-	-	+	-	+	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
355	+	-	-	+	-	+	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
356	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
357	+	-	-	+	-	+	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
358	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
359	+	-	-	+	-	+	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
360	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
361	+	-	-	+	-	+	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
362	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
363	+	-	-	+	-	+	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
364	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
365	+	-	-	+	-	+	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
366	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
367	+	-	-	+	-	+	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
368	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
369	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
370	+	-	-	+	-	+	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
371	+	-	-	+	-	+	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
372	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
373	+	-	-	+	-	+	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
374	+	-	-	+	-	+	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
375	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
376	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
377	+	-	-	+	-	+	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
378	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
379	+	-	-	+	-	+	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
380	+	-	-	+	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
381	+	-	-	+	-	+	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
382	+	+	-	-	-	-	+	-	-	3	3	3	3	3	3	-	+	+	+	-	-	+	-	-	-	-	-	-	-	+	+	-

Table 26: Instruction and bit identifiers of few discovered ELF innstructions (Part 8 - Bits 31-00).

ID	Bits																														
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
344	2	2	2	2	2	2	+	+	+	+	+	+	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
345	2	2	2	2	2	2	+	+	+	+	+	+	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	+	-	-
346	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	+	-	-	-	-	-	-	-	+	+
347	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
348	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
349	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
350	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	-	-	-	-	-	+	+
351	2	2	2	2	2	2	1	1	1	+	+	+	0	0	0	0	0	0	-	+	+	+	-	-	-	-	+	-	-	-	+	+
352	-	-	-	-	-	-	0	0	0	0	0	0	1	1	1	1	1	1	-	+	+	+	-	-	+	-	+	-	-	+	-	+
353	-	-	-	-	-	-	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	-	-	+	-	+	-	-	+	-	+
354	1	1	1	1	1	1	0	0	0	0	0	0	2	2	2	2	2	2	-	+	+	+	-	-	+	-	+	-	-	+	-	+
355	1	1	1	1	1	1	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	-	-	+	-	+	-	-	+	-	+
356	-	-	-	-	-	-	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	-	+	+	-	+	-	-	+	-	+
357	1	1	1	1	1	1	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	-	+	+	-	+	-	-	+	-	+
358	-	-	-	-	-	-	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	-	+	+	-	-	-	-	+	-	+
359	1	1	1	1	1	1	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	-	+	+	-	-	-	-	+	-	+
360	-	-	-	-	-	-	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	-	+	-	+	-	-	-	+	-	+
361	1	1	1	1	1	1	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	-	+	-	+	-	-	-	+	-	+
362	-	-	-	-	-	-	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	+	-	+	-	+	-	-	+	-	+
363	1	1	1	1	1	1	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	+	-	+	-	+	-	-	+	-	+
364	-	-	-	-	-	-	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	+	-	+	-	-	-	-	+	-	+
365	1	1	1	1	1	1	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	+	-	+	-	-	-	-	+	-	+
366	-	-	-	-	-	-	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	+	-	-	+	-	-	-	+	-	+
367	1	1	1	1	1	1	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	+	-	-	+	-	-	-	+	-	+
368	-	-	-	-	-	-	0	0	0	0	0	0	1	1	1	1	1	1	-	+	+	+	-	-	+	-	-	-	-	+	-	+
369	-	-	-	-	-	-	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	-	-	+	-	-	-	-	+	-	+
370	1	1	1	1	1	1	0	0	0	0	0	0	2	2	2	2	2	2	-	+	+	+	-	-	+	-	-	-	-	+	-	+
371	1	1	1	1	1	1	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	-	-	+	-	-	-	-	+	-	+
372	-	-	-	-	-	-	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	-	-	-	+	-	-	-	+	-	+
373	1	1	1	1	1	1	0	0	0	0	0	0	2	2	2	2	2	2	-	+	+	+	-	-	-	+	-	-	-	+	-	+
374	1	1	1	1	1	1	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	-	-	-	+	-	-	-	+	-	+
375	-	-	-	-	-	-	0	0	0	0	0	0	1	1	1	1	1	1	-	+	+	+	-	-	-	-	-	-	-	+	-	+
376	-	-	-	-	-	-	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	+	+	+	-	+	-	-	+	-	+
377	1	1	1	1	1	1	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	+	+	+	-	+	-	-	+	-	+
378	-	-	-	-	-	-	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	+	+	+	-	-	-	-	+	-	+
379	1	1	1	1	1	1	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	+	+	+	-	-	-	-	+	-	+
380	-	-	-	-	-	-	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	+	+	-	+	-	-	-	+	-	+
381	1	1	1	1	1	1	0	0	0	0	0	0	+	+	+	+	+	+	-	+	+	+	+	+	-	+	-	-	-	+	-	+
382	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	-	+	+	+	-	-	-	+	+	-	-	+	-	-

AII Architectural Features of Some Warp Instructions

We report the architectural features of some PTX and ELF warp instructions. In this appendix, we do not report the local instability phenomena (see section 7.3). For some detailed examples of the procedures and the formulas used to determine and quantify the architectural features of some PTX and ELF warp instructions, see sections 7.3 and 7.16.

Table 27: Legend for the quantification of the warp instructions' features

Symbol	Meaning	Associated colors
MT	maximum throughput	green and blue
IL	instruction's latency	green
WRL	write-read latency	green
RRL	read-read latency	magenta
WL	warp's latency	brown

Table 28: Descriptions for the quantification of the warp instructions' features

Descriptions of the symbols:

1) Maximum Throughput (MT): maximum number of instructions executed by a streaming multiprocessor per functional unit clock cycle (e.g. 32 or 4).

2) Instruction's Latency (IL) or Write-Read Latency (WRL): number of functional unit clock cycles that are necessary to wait before being able to read, and therefore reuse, the register used to store the result of the instruction.

3) Read-Read Latency (RRL): number of functional unit clock cycles that are necessary to wait before being able to re-read a register used as an operand in the instruction.

4) Warp's Latency (WL): number of functional unit clock cycles that are necessary to wait before a warp scheduler can reschedule the warp for the execution of the next instruction.

Table 29: Architectural features of some PTX Instruction Configurations

PTX Instr. Config.	MT	WL	WRL	RRL	PTX Instr. Config.	MT	WL	WRL	RRL
add (s32.c.ne)	16	16	≤ 22	≤ 16	add (s32.c.no)	16	16	≤ 22	≤ 16
add (s32.n)	32	6	≤ 24	≤ 6	add (s64.n)	16	20	≤ 20	≤ 20
add (s64.c.ne)	8	18	≤ 18	≤ 18	add (s64.c.no)	8	18	≤ 18	≤ 18
add (u32.c.ne)	16	16	≤ 22	≤ 16	add (u32.c.no)	16	16	≤ 22	≤ 16
add (u32.n)	32	6	≤ 24	≤ 6	add (u64.n)	16	20	≤ 20	≤ 20
add (u64.c.ne)	8	18	≤ 18	≤ 18	add (u64.c.no)	8	18	≤ 18	≤ 18
and (b32.c.ne)	16	16	≤ 22	≤ 16	and (b32.c.no)	16	16	≤ 22	≤ 16
and (b32.n)	32	6	≤ 24	≤ 6	and (b64.n)	16	6	≤ 12	≤ 6
and (b64.c.ne)	8	10	≤ 10	≤ 10	and (b64.c.no)	8	10	≤ 10	≤ 10
max (s32.c.ne)	8	6	≤ 10	≤ 6	max (s32.c.no)	8	6	≤ 10	≤ 6
max (s32.n)	16	4	≤ 10	≤ 4	max (s64.n)	8	8	≤ 8	≤ 8
max (s64.c.ne)	4	8	≤ 8	≤ 8	max (s64.c.no)	4	8	≤ 8	≤ 8
max (u32.c.ne)	8	6	≤ 10	≤ 6	max (u32.c.no)	8	6	≤ 10	≤ 6
max (u32.n)	16	4	≤ 10	≤ 4	max (u64.n)	8	8	≤ 8	≤ 8
max (u64.c.ne)	4	8	≤ 8	≤ 8	max (u64.c.no)	4	8	≤ 8	≤ 8
min (s32.c.ne)	8	6	≤ 10	≤ 6	min (s32.c.no)	8	6	≤ 10	≤ 6
min (s32.n)	16	4	≤ 10	≤ 4	min (s64.n)	8	8	≤ 8	≤ 8
min (s64.c.ne)	4	8	≤ 8	≤ 8	min (s64.c.no)	4	8	≤ 8	≤ 8
min (u32.c.ne)	8	6	≤ 10	≤ 6	min (u32.c.no)	8	6	≤ 10	≤ 6
min (u32.n)	16	4	≤ 10	≤ 4	min (u64.n)	8	8	≤ 8	≤ 8
min (u64.c.ne)	4	8	≤ 8	≤ 8	min (u64.c.no)	4	8	≤ 8	≤ 8
mul.lo (s32.c.ne)	16	16	≤ 20	≤ 16	mul.lo (s32.c.no)	16	16	≤ 20	≤ 16
mul.lo (s32.n)	16	4	≤ 10	≤ 4	mul.lo (s64.n)	4	10	≤ 10	≤ 10
mul.lo (s64.c.ne)	16	14	≤ 14	≤ 14	mul.lo (s64.c.no)	16	14	≤ 14	≤ 14
mul.lo (u32.c.ne)	16	16	≤ 20	≤ 16	mul.lo (u32.c.no)	16	16	≤ 20	≤ 16
mul.lo (u32.n)	16	4	≤ 10	≤ 4	mul.lo (u64.n)	4	10	≤ 10	≤ 10
mul.lo (u64.c.ne)	16	14	≤ 14	≤ 14	mul.lo (u64.c.no)	16	14	≤ 14	≤ 14
or (b32.c.ne)	16	16	≤ 22	≤ 16	or (b32.c.no)	16	16	≤ 22	≤ 16
or (b32.n)	32	6	≤ 24	≤ 6	or (b64.n)	16	6	≤ 12	≤ 6
or (b64.c.ne)	8	12	≤ 12	≤ 12	or (b64.c.no)	8	12	≤ 12	≤ 12
pop (b32.c.ne)	16	16	≤ 20	≤ 16	pop (b32.c.no)	16	16	≤ 20	≤ 16
pop (b32.n)	16	4	≤ 10	≤ 4	pop (b64.n)	8	10	≤ 16	≤ 10
pop (b64.c.ne)	8	16	≤ 16	≤ 16	pop (b64.c.no)	8	16	≤ 16	≤ 16
set.gt (s32.c.ne)	16	16	≤ 16	≤ 16	set.gt (s32.c.no)	16	16	≤ 16	≤ 16
set.gt (s32.n.rr)	32	8	//	≤ 28	set.gt (s32.n.wr)	8	16	≤ 16	//
set.gt (s64.c.ne)	8	24	≤ 24	≤ 24	set.gt (s64.c.no)	8	24	≤ 24	≤ 24
set.gt (s64.n.rr)	16	20	//	≤ 20	set.gt (s64.n.wr)	8	24	≤ 24	//

Table 30: Architectural features of some PTX Instruction Configurations

PTX Instr. Config.	MT	WL	WRL	RRL	PTX Instr. Config.	MT	WL	WRL	RRL
set.gt (u32.c.ne)	16	16	≤ 16	≤ 16	set.gt (u32.c.no)	16	16	≤ 16	≤ 16
set.gt (u32.n.rr)	32	8	//	≤ 28	set.gt (u32.n.wr)	8	16	≤ 16	//
set.gt (u64.c.ne)	8	24	≤ 24	≤ 24	set.gt (u64.c.no)	8	24	≤ 24	≤ 24
set.gt (u64.n.rr)	16	20	//	≤ 20	set.gt (u64.n.wr)	8	24	≤ 24	//
set.lt (s32.c.ne)	16	16	≤ 16	≤ 16	set.lt (s32.c.no)	16	16	≤ 16	≤ 16
set.lt (s32.n.rr)	32	8	//	≤ 28	set.lt (s32.n.wr)	8	16	≤ 16	//
set.lt (s64.c.ne)	8	24	≤ 24	≤ 24	set.lt (s64.c.no)	8	24	≤ 24	≤ 24
set.lt (s64.n.rr)	16	20	//	≤ 20	set.lt (s64.n.wr)	8	24	≤ 24	//
set.lt (u32.c.ne)	16	16	≤ 16	≤ 16	set.lt (u32.c.no)	16	16	≤ 16	≤ 16
set.lt (u32.n.rr)	32	8	//	≤ 28	set.lt (u32.n.wr)	8	16	≤ 16	//
set.lt (u64.c.ne)	8	24	≤ 24	≤ 24	set.lt (u64.c.no)	8	24	≤ 24	≤ 24
set.lt (u64.n.rr)	16	20	//	≤ 20	set.lt (u64.n.wr)	8	24	≤ 24	//
xor (b32.c.ne)	16	16	≤ 22	≤ 16	xor (b32.c.no)	16	16	≤ 22	≤ 16
xor (b32.n)	32	6	≤ 24	≤ 6	xor (b64.n)	16	6	≤ 12	≤ 6
xor (b64.c.ne)	8	10	≤ 10	≤ 10	xor (s64.c.no)	8	10	≤ 10	≤ 10

Table 31: Architectural features of some ELF Instruction Configurations
For the ELF Instruction Configuration Identifiers see Table 1 at page 198.

ELF Instr. Config.	MT	WL	WRL	RRL
IADD.X_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0	32	6	≤ 24	≤ 6
IADD_NT2ER0.CC,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0	32	28	≤ 28	≤ 28
IADD_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0	32	6	≤ 24	≤ 6
ICMP.LT_NT2ER0,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0	16	4	≤ 12	≤ 4
IMAD.U32.U32.HI.X_NT2ER0.CC,_NT2ER1,_NT2ER2,_ST2ER0_1_0_3_0_0_NT2ER0	16	14	≤ 14	≤ 14
IMAD.U32.U32.X_NT2ER0,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0	16	4	≤ 12	≤ 4
IMAD.U32.U32_NT2ER0,_NT2ER1,_NT2ER2,_NT2ER3_0_0_4_0_0_NT2ER0	16	4	≤ 12	≤ 4
IMNMX.U32.XHI_NT2ER0.CC,_NT2ER1,_NT2ER2,_ST1ER0_0_1_3_0_0_NT2ER0	16	14	≤ 14	≤ 14
IMNMX.U32.XHI_NT2ER0.CC,_NT2ER1,_NT2ER2,ST1ER0_0_1_3_0_0_NT2ER0	16	14	≤ 14	≤ 14
IMNMX.U32.XLO_NT2ER0,_NT2ER1,_NT2ER2,_ST1ER0_0_1_3_0_0_NT2ER0	16	4	≤ 10	≤ 4
IMNMX.U32.XLO_NT2ER0,_NT2ER1,_NT2ER2,ST1ER0_0_1_3_0_0_NT2ER0	16	4	≤ 10	≤ 4
IMNMX.U32_NT2ER0,_NT2ER1,_NT2ER2,_ST1ER0_0_1_3_0_0_NT2ER0	16	4	≤ 10	≤ 4
IMNMX.U32_NT2ER0,_NT2ER1,_NT2ER2,ST1ER0_0_1_3_0_0_NT2ER0	16	4	≤ 10	≤ 4
IMNMX.XHI_NT2ER0.CC,_NT2ER1,_NT2ER2,_ST1ER0_0_1_3_0_0_NT2ER0	16	14	≤ 14	≤ 14
IMNMX.XHI_NT2ER0.CC,_NT2ER1,_NT2ER2,ST1ER0_0_1_3_0_0_NT2ER0	16	14	≤ 14	≤ 14
IMNMX_NT2ER0,_NT2ER1,_NT2ER2,_ST1ER0_0_1_3_0_0_NT2ER0	16	4	≤ 10	≤ 4
IMNMX_NT2ER0,_NT2ER1,_NT2ER2,ST1ER0_0_1_3_0_0_NT2ER0	16	4	≤ 10	≤ 4

Table 32: Architectural features of some ELF Instruction Configurations
For the ELF Instruction Configuration Identifiers see Table 1 at page 198.

ELF Instr. Config.	MT	WL	WRL	RRL
IMUL.HI_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0	16	4	≤ 10	≤ 4
IMUL.U32.U32.HI_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0	16	4	≤ 10	≤ 4
IMUL.U32.U32_NT2ER0.CC,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0	16	14	≤ 14	≤ 14
IMUL.U32.U32_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0	16	4	≤ 10	≤ 4
IMUL_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0	16	4	≤ 10	≤ 4
ISETP.EQ.U32.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (rr)	32	8	//	≤ 28
ISETP.EQ.U32.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (wr)	8	16	≤ 16	//
ISETP.GT.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (rr)	32	8	//	≤ 28
ISETP.GT.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (wr)	8	16	≤ 16	//
ISETP.GT.U32.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (rr)	32	8	//	≤ 28
ISETP.GT.U32.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (wr)	8	16	≤ 16	//
ISETP.GT.U32.X.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (rr)	32	8	//	≤ 28
ISETP.GT.U32.X.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (wr)	8	16	≤ 16	//
ISETP.GT.X.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (rr)	32	8	//	≤ 28
ISETP.GT.X.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (wr)	8	16	≤ 16	//
ISETP.LT.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (rr)	32	8	//	≤ 28
ISETP.LT.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (wr)	8	16	≤ 16	//
ISETP.LT.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (rr)	32	8	//	≤ 28
ISETP.LT.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (wr)	8	16	≤ 16	//
ISETP.LT.U32.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (rr)	32	8	//	≤ 28
ISETP.LT.U32.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (wr)	8	16	≤ 16	//
ISETP.LT.U32.X.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (rr)	32	8	//	≤ 28
ISETP.LT.U32.X.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (wr)	8	16	≤ 16	//
ISETP.LT.X.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (rr)	32	8	//	≤ 28
ISETP.LT.X.AND_NT1ER0,_ST1ER0,_NT2ER0,_NT2ER1,_ST1ER1_0_2_2_1_0_NT1ER0 (wr)	8	16	≤ 16	//
LOP.AND_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0	32	6	≤ 24	≤ 6
LOP.OR_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0	32	6	≤ 24	≤ 6
LOP.XOR_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0	32	6	≤ 24	≤ 6
POPC_NT2ER0,_NT2ER1,_NT2ER2_0_0_3_0_0_NT2ER0	32	4	≤ 10	≤ 4
PSETP.AND.AND_NT1ER0,_ST1ER0,_ST1ER1,_NT1ER1,_NT1ER2_0_2_0_3_0_NT1ER0	16	4	≤ 12	≤ 12
PSETP.AND.AND_NT1ER0,_ST1ER0,_ST1ER1,_NT1ER1,NT1ER2_0_2_0_3_0_NT1ER0	16	4	≤ 12	≤ 12
PSETP.AND.OR_NT1ER0,_ST1ER0,_NT1ER1,_NT1ER2,_NT1ER3_0_1_0_4_0_NT1ER0	16	4	≤ 12	≤ 12
PSETP.AND.OR_NT1ER0,_ST1ER0,_NT1ER1,_NT1ER2,NT1ER3_0_1_0_4_0_NT1ER0	16	4	≤ 12	≤ 12
SEL_NT2ER0,_NT2ER1,_NT2ER2,_NT1ER0_0_0_3_1_0_NT2ER0	16	4	≤ 10	≤ 4
SEL_NT2ER0,_NT2ER1,_NT2ER2,NT1ER0_0_0_3_1_0_NT2ER0	16	4	≤ 10	≤ 4
SEL_NT2ER0,_ST2ER0,_NT2ER1,_NT1ER0_1_0_2_1_0_NT2ER0	16	4	≤ 10	≤ 4
SEL_NT2ER0,_ST2ER0,_NT2ER1,NT1ER0_1_0_2_1_0_NT2ER0	16	4	≤ 10	≤ 4
SEL_NT2ER0,_ST2ER0,_NT2ER1,_!NT1ER0_1_0_2_1_0_NT2ER0	16	4	≤ 10	≤ 4