MASTER

Analysis and modeling of the timing behavior of GPU architectures

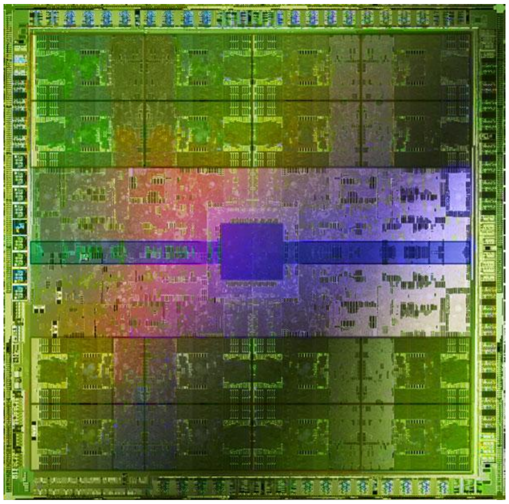
Voudouris, P.

Award date:
2014

Analysis and Modeling of the Timing Behavior of GPU Architectures

Master Thesis

Petros Voudouris



Electronic Systems group
Faculty of Electrical Engineering
Eindhoven University of Technology

Supervisors:	Henk Corporaal
	Gert-Jan van den Braak

November 2014

Abstract

Graphics processing units (GPUs) offer massive parallelism. Since a couple of years GPUs can also be used for more general purpose applications; a wide variety of applications can be accelerated efficiently with the use of the CUDA and OpenCL programming models.

Real-time systems frequently use many sensors that produce a big amount of data. GPUs can be used to process the data and remove the workload from the central process unit (CPU). To use the GPUs in real time systems it is required to have time predictable behavior. However, it is hard to give an estimation of the worst case execution time (WCET) of a GPU program, since only few timing details of the GPU architecture are given. In addition, inside the GPUs several arbiters are used, and their scheduling details are not always clearly described.

The Nvidia Fermi architecture is analyzed in order to identify the sources of time variation and unpredictability. Initially, all the main components of the architecture are discussed and investigated for sources of time variation and predictability.

From the analyzed components of the architecture we chose to continue with the warp scheduler, the scratchpad memory and the out-of-chip global memory. The warp scheduler determines the schedule of the warps and as a result influences significantly the total execution time. The scratchpad memory is widely used in GPU application to hide the memory latency. Finally, the global memory instructions were analyzed since they are used from all the applications. Micro-benchmarks implemented in assembly were used to quantify the sources of variation. We execute the same experiments multiple times in order to determine the variation of the execution time among the experiments.

A timing model for the warp scheduler based on results from the experiments was introduced. The model was tested for Convolution separable benchmark from CUDA SDK. The model was accurate for average case (96.5% of the experiments) with 2% and 3% error for Rows and Columns kernel respectively. For the best case the model had -15% and -54% and worst case 28% and 1% error for rows and columns kernel (3.5% of the experiment). In addition, it was shown that the model is scalable up to 6 warps without any modification.

Furthermore, the variable-rate warp scheduler was implemented in GPGPU-Sim that improves the time predictability of the GPU by assigning the warps in fixed slots. Initial results show that by assigning one warp with higher scheduling rate can improve the performance also by taking advantage of the inter-warp locality of the warps.

Finally, based on the findings of this report, the GPU can be used in real-time systems with soft deadlines. By using the suggested changes of the architecture, the time predictability of the architecture can be improved even more and support applications with more strict timing constrains.

Acknowledgments

I would like to express my sincere gratitude to my supervisors Professor Henk Corporaal and Gert-Jan van den Braak, who have supported me throughout this thesis with great patience and immense knowledge. Without their help the realization of this project would not be possible. Their suggestions and encouragement helped me to overcome difficulties throughout this project and especially during writing this report.

Furthermore, I would like to thank all the members of the PARsE group for their valuable feedback.

Last but not least, I would like thank to my family and my friends for their continuous support during my whole studies.

Table of Contents

1. Introduction

Initially, the motivation to use the graphic process units for general purpose computing in real-time systems is presented. Next the problems of using the GPUs in real time systems are discussed. Finally, the outline of the report is introduced.

1.1. Motivation

Computer systems are becoming more powerful while at the same time they become more complex. The performance improvement is essential to serve the requirements of modern systems. However, the state-of-the-art systems are so complex that it is not possible to analyze them in detail. In real-time systems the time predictability of a system is equally important with the correctness of the result. So, in the context of real-time systems the time predictability is essential for the system.

Real-time systems typically appear in cyber-physical systems (Figure 1), where sensors and actuators are present. The control part of the system processes the data from the sensors and gives the appropriate commands to the actuators. Cameras can be used as sensors for visual control of the system and all the sensors produce independent streams of data. Therefore, an architecture which can exploit data level parallelization can process the data produced from the sensors efficiently.

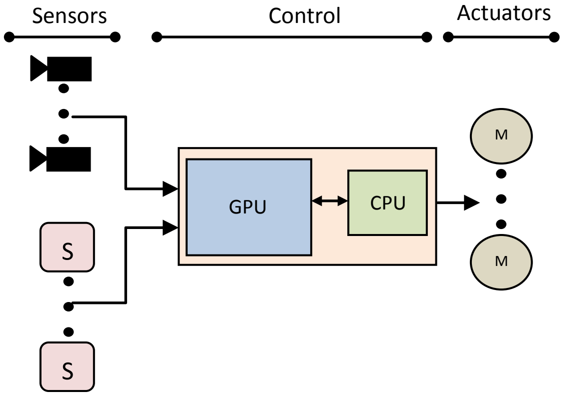


Figure 1 A real-time system with GPU that exploits the data level parallelism derived from multiple sensors.

For a system with many sensors, data level parallelization can be exploited by the graphics processing unit (GPU). It can process the data and remove the workload from the central process unit (CPU). GPUs provide massive parallelism and can be used for general purpose programs to improve the performance of compute intensive parts of the program. The Nvidia Fermi architecture provides mechanisms that improve the programmability of the GPU that are useful for general purpose GPU programming (GPGPU). Typically, GPUs hide memory latency by the concurrently executing threads. Fermi is the first GPU of Nvidia that has also data caches in order to hide memory latency. In addition, Fermi is a highly parallel architecture and shows sources of unpredictability that are related to parallel execution and shared resources.

1.2. Problem statement

Multiple executions of the same program with the same input on the same GPU take variable execution times. The source of variation has to be identified and quantified in order to be able to determine the range of the variation.

To use the GPU in a cyber-physical system with real-time characteristics, it is required to have time predictable behavior that provides guaranties for the worst case execution time. Therefore, it is required to determine variation in execution times. If the variation is bounded then the system has time predictable behavior. Otherwise, we cannot provide any guaranties for the worst case execution time and the architecture is time unpredictable.

Typically, timing analysis tools specialized for the worst case execution time are used to identify the timing behavior of the architecture. These tools are not available for GPUs thus a different approach is needed to be followed. In this report, a timing analysis of the Fermi GPU architecture is performed to identify the sources of time unpredictability. We focus on latency analysis since throughput analysis for Fermi has been performed extensively in literature. Microbenchmarks implemented in assembly for GPUs are used to identify and quantify the sources of time variation. To identify the variation of the execution time, the experiments performed multiple times. Based on the experiments a timing model is implemented and tested. Finally, a scheduling algorithm is proposed that improves the time predictability of the GPU.

1.3. Organization of the report

The rest of the report is organized as follows. First, in section 2 background information for the Fermi GPU architecture and the GPGPU-Sim cycle level simulator is given. Next, in section 3 the related work for time predictability is presented. In addition, related studies on performance analysis on GPU are discussed. Furthermore, section 4 introduces the concept of time predictability of hardware architectures and discusses the sources of timing variation in GPUs. Section 5 presents the experiments for the quantification of the sources of time variation for the warp scheduler, scratchpad memory and the global memory. In section 6 the timing model for the warp scheduling of the instructions is illustrated. The scalability of the model in terms of number of warps is discussed. In section 7, the model with the Convolution separable from CUDA SDK is tested and is compared with state-of-the-art performance model. Finally, section 8 describes the variable rate warp scheduler and section 9 concludes the report and presented the future work.

2. Background

Initially, the GPU architecture is described in section 2.1. Next, in section 2.2 background information is given for the software model for the GPU architecture. Finally, the GPGU-Sim cycle level simulator is presented in section 2.3.

2.1. GPU architecture

For this assignment, we are going to use the Nvidia Fermi GPU architecture. Fermi is not the latest architecture of Nvidia. The timeline in Figure 2 shows that the Kepler and Maxwell architectures have been introduced in 2012 and 2014 respectively while Fermi was introduced in 2009. The Fermi architecture was chosen because no assembler and simulator exist for Kepler and Maxwell. For Fermi, the asFermi assembler and the GPGPU-Sim simulator are available.

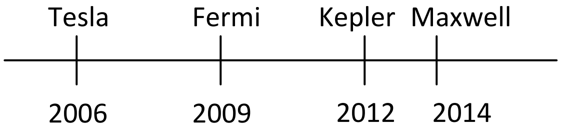


Figure 2 NVIDIA timeline of different architecture.

The GPU communicates with the CPU by peripheral component interconnect (PCI). The CPU sends the data through PCI to the GPU (Figure 3). The GPU process the data and sends back the results. The Fermi architecture has 16 SMs that are connected between each other with an interconnect network. In addition there is a level 2 cache memory which is shared between the cores.

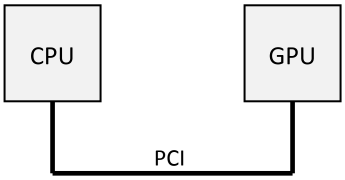


Figure 3 CPU and GPU communication

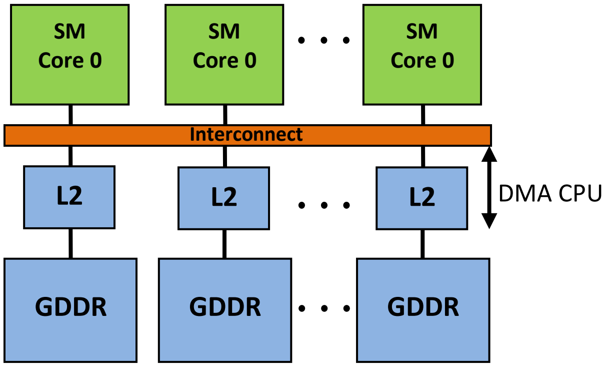


Figure 4 Overview of the Fermi GPU architecture

In the Fermi architecture, each SM (Figure 5) has two warp schedulers and two instruction dispatch units, so two warps can be executed concurrently by exploiting thread level parallelism (TLP). In addition, there is a register file of 128KB. In this way the different threads can have their own registers. Since every thread has its own register the cost of the context switching is eliminated. There are 2 sets of 16 cores, 16 load store (LD/ST) units and 4 special function units (SFU) that execute the instructions.

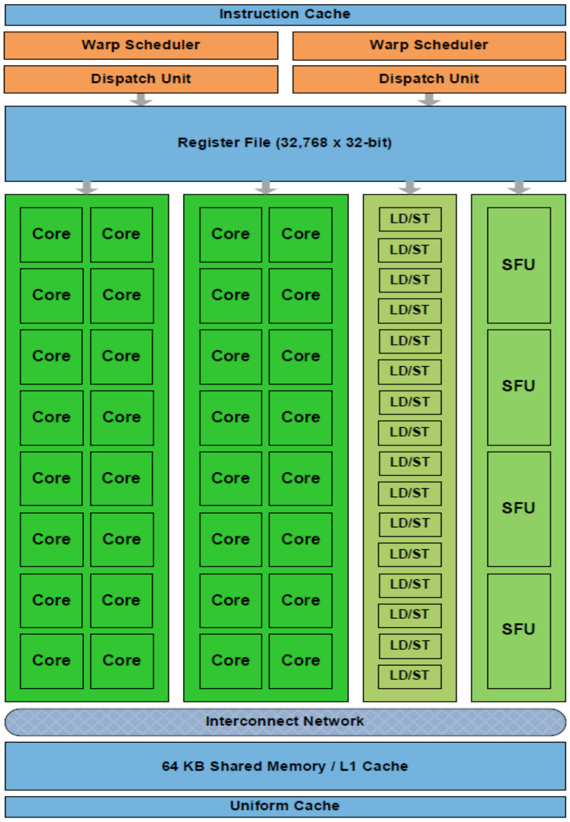


Figure 5 Block diagram of the multithreaded SIMD Processor of a Fermi GPU (SM) (Taken from [1]).

Furthermore, every SM has 64KB of on-chip memory that can be configured as 48 KB of shared memory and 16KB of L1 cache, or the other way (Figure 6). The Fermi architecture has a Level 2 cache memory which is shared between the SMs. The L2 cache memory is unified for instructions and data. It follows the traditional speed hierarchy but inverts the size hierarchy (Figure 7). Considering all the 16 SMs of the Fermi architecture the lowest level is bigger than level 1 and level 2 cache memories.

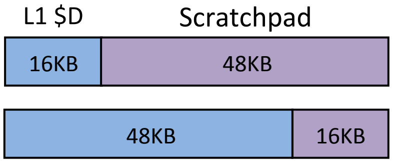


Figure 6 Two shared memory configurations

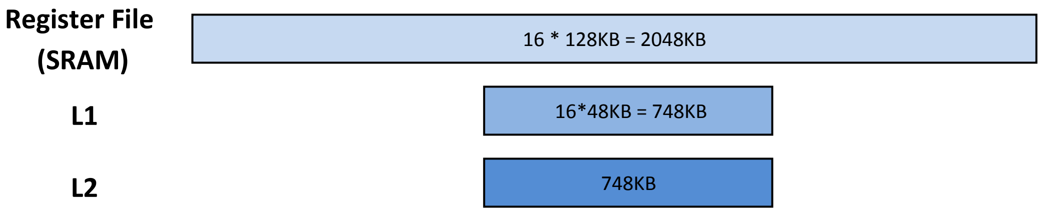


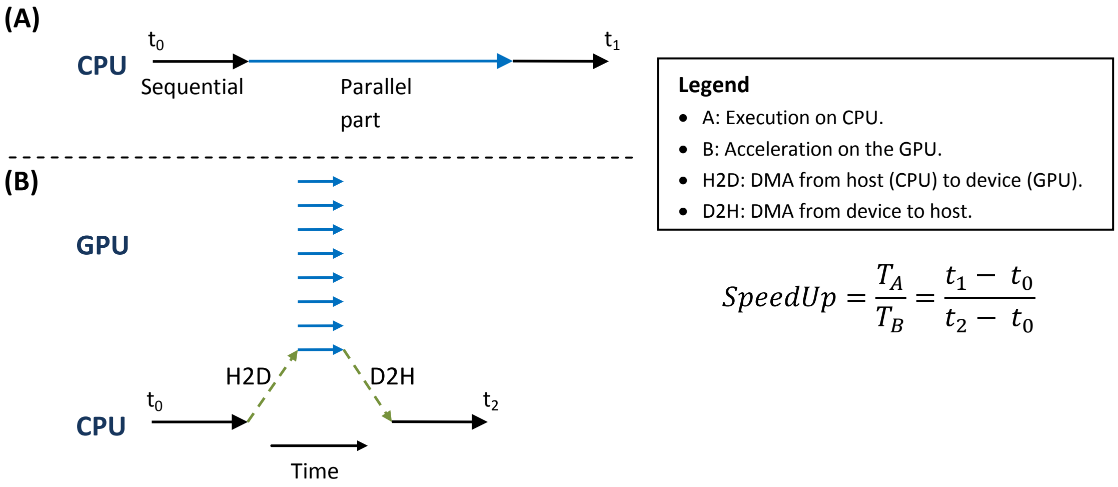
Figure 7 Cache memory hierarchy of Fermi GPU architecture which uses the second configuration of figure 6

2.2. CUDA programming model

Nowadays, GPUs are not used only for graphics. GPUs are also used to speed-up the compute intensive parts of general purpose programs. The massive parallelism of the GPUs can be used through programming models like compute unified device architecture (CUDA) and OpenCL for accelerating the programs. The programmer can choose which parts of the algorithm are better suited for mapping on the GPU in order to take advantage of the highly parallel hardware that the GPUs offer. CUDA and OpenCL provide abstraction to the programmer for scratchpad memory, thread ID and synchronization.

The central processing unit (CPU) and GPU have separate memories. Data should be transferred explicitly from one device to the other. The memory transfer hampers the speed-up if the computation on the GPU is small.

An example of executing a program at a GPU is presented in Figure 8. Initially, the normal execution of the program on the CPU is presented in (A). It consists of a part that can be parallelized and two sequential parts, at the beginning and at the end. The parallel part of the program can be executed on the GPU as is shown in (B). First, it is required to transfer the data from the CPU (host) to the GPU (device) through direct memory access (DMA) (host to device or H2D). After the parallel execution of the program on the GPU the data have to be transferred back again to the CPU (device to host or D2H). The potential speed-up, depends on cost of the transferring of the data between the CPU and the GPU and on the level of parallelism that exists in the program.


(A)
CPU
t0
t1
Sequential
Parallel part
(B)
GPU
H2D
D2H
CPU
t0
t2
Time
Legend
A: Execution on CPU.
B: Acceleration on the GPU.
H2D: DMA from host (CPU) to device (GPU).
D2H: DMA from device to host.

$$SpeedUp = \frac{T_A}{T_B} = \frac{t_1 - t_0}{t_2 - t_0}$$

Figure 8 Example of CPU execution (A) and GPU acceleration (B).

The Nvidia Fermi GPU is a scalable in-order architecture and consists of a set of multithreaded Streaming Multiprocessors (SM). A multiprocessor employs a unique architecture called SIMT (Single-Instruction, Multiple-Thread). The threads are independent elements and can be executed in parallel. In the SIMT architecture the multiprocessor manages, schedules, and executes threads in groups of 32, called warps. The warps are group in thread blocks. A thread blocks is mapped to an SM, so all the warps of a thread block are executed in one SM (Figure 9). The hierarchy of threads provides flexibility to the programmer and as a result improves the programmability of the GPU.

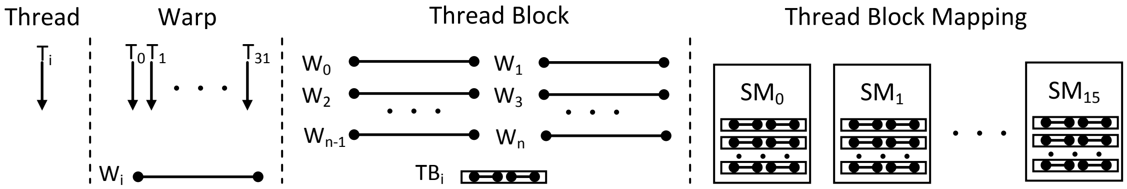


Figure 9 Hierarchy of the threads, warps and thread blocks in the Fermi GPU architecture and the mapping of the thread blocks to different SMs

To clarify more the how the GPU is programmed an example from Hennesy and Paterson book [38] is used. Initially, the code in C is presented; it calculates double precision aX plus Y. The code in C is going to be executed sequentially, so the iterations of the loop are executed one after the other.

```
0.  // Invoke (double-precision aX plus Y)DAXPY
1.  daxpy(n, 2.0, x, y);
2.
3.  // DAXPY in C
4.  void daxpy(int n, double a, double *x, double *y)
5.  {
6.       for (int i = 0; i < n; ++i)
7.            y[i] = a*x[i] + y[i];
8.  }
```

The CUDA code that has the same functionality with the code in C is presented below. Note that in this example it is not presented the transactions of the data to and from the GPU. To distinguish the code for the CPU (host) and the code for the GPU (device), the "__host__" and "__device__" keywords are used respectively. For the functions executed at the GPU the number of thread blocks and threads has to be determined. The syntax to call a function for the GPU is extended to " FunctionName <<<dimGrid, dimBlock>>> (parameters)", where dimGrid is the number of the blocks and the dimBlock is the number of threads per block.

CUDA provides identifiers for the thread blocks (blockIdx.x) and the threads (threadIdx.x). In addition, the size of the thread block is given by "blockDim.x". By combining the information of these 3 identifiers (line 11) the threads can be distinguished. For example, for two thread blocks and thread block size of 4 the value of variable "i" has the value as presented in Table 1. By using "i" as index to the arrays, all threads can process a different data element in parallel.

Table 1 Example of index calculations.

T_i	0	1	2	3	0	1	2	3
TB_i	0	0	0	0	1	1	1	1
i	0	1	2	3	4	5	6	7

```
// Invoke DAXPY with 256 threads per Thread Block
__host__  //CPU
int nblocks = (n+255) / 256;

daxpy<<<nblocks, 256>>>(n, 2.0, x, y);

// DAXPY in CUDA
__device__ //GPU
void daxpy(int n, double a, double *x, double *y)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```

2.3. GPGPU-Sim

The GPGPU-Sim is cycle level simulator designed to mimic the behavior of the Tesla and Fermi GPU architectures [3] [4]. The user can choose between these two architectures and select a range of parameters, such as warp scheduler, size of the caches etc. The implementation of the simulator is based on the documentation from Nvidia, patents and in some cases assumption, since Nvidia does not provide all the details of its architectures. A high level architecture of the GPGPU-Sim is presented in Figure 10. The architecture is split in two parts: first there is the SIMT front-end (green) and second the SIMD datapath (yellow). A more detailed view of this architecture is given in appendix C.

Initially, in the SIMT front-end the instructions are fetched from the instruction cache. The instructions are decoded and stored in the instruction buffer. Next at the scoreboard the instructions are checked for read after write (RaW) and write after write (WaW) dependencies and the scheduler decides which warp is going to be issued to the SIMD datapath. Furthermore, the SIMT-Stack is used to handle the branch divergence, by storing the diverge and converge points of each warp.

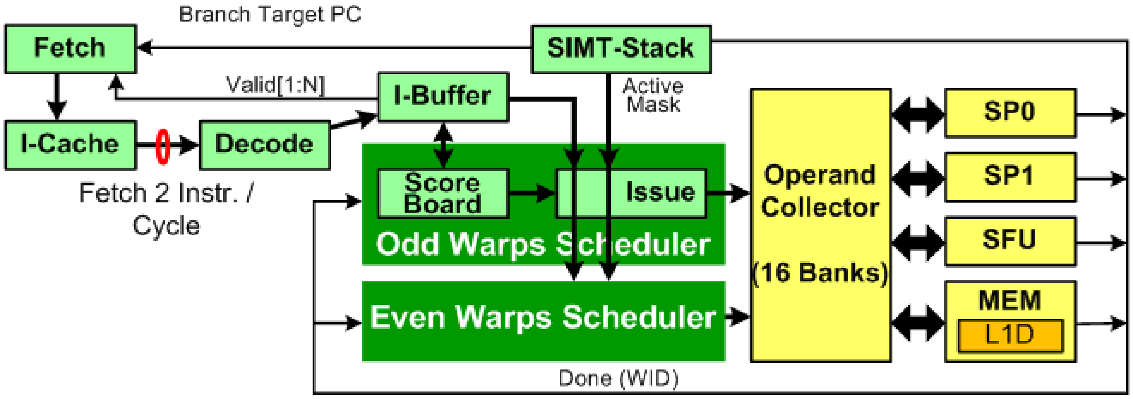


Figure 10 Architectural high level block diagram of GPGPU-Sim. (Taken from [4])

The SIMD datapath consists of the operand collector, two sets of 16 cores or streaming processors (SP) in the context of GPGPU-Sim, 4 special function units, the shared memory (scratchpad and level 1 data cache). The operand collector is a set of buffers and arbitration logic used to provide the appearance of a multi-ported register file using memory of multiple banks of single ported RAMs.

3. Related work

In this section the related work for the time predictability is presented. Furthermore, the related work for the timing analysis of Fermi GPU architecture is shown. For the timing analysis there are two types of related work, performance analysis for the WCET (Worst Case Execution Time) and performance analysis for the average case.

3.1. Time predictability

The problems that can occur in the WCET analysis of a system are described in [7]. More specifically, it describes how to decompose the problem in subtasks and it provides techniques that are used to define the upper bound of the execution time. It is shown that a given WCET analysis tool may not consider all these subtasks or may solve the same subtask in different ways. In addition it is shown that the processor behavior analysis significantly depends on the complexity of the processor architecture.

Furthermore, threats of predictability and proposes methods that can improve the time predictability are described in [9]. The sources of unpredictability are categorized in architectural level, software level, task level and distributed operations level. In addition sources of unpredictability that may occur through the interaction of these different levels are presented. For every category, the features which can be threats for time predictability are described.

The access time to off-chip memory is another source of time variability because it can vary from access to access. It depends on the dynamic memory traffic pattern and the arrivals of the memory request. The memory controller is responsible to serve the memory requests. The memory controller can be static or dynamic. Static memory-controllers serve the memory requests with static scheduling. Dynamic memory-controllers adapt to the different memory traffic patterns. Static memory-controllers are time predictable but not flexible to memory traffic changes and the opposite holds for the dynamic. A memory controller with static and dynamic characteristics that provides minimum throughput and maximum latency guaranties is proposed in [33].

Architectures with time predictable behavior are presented in [14] [15]. The main idea of these architectures is that they are composed from predictable subsystems. These architectures with the composition of predictable subsystem they have a predictable system. In addition, fine grain multi-threading is used in order to preserve the isolation of the execution. Time division multiple access (TDMA) scheme is used for the memory accesses. Also, the SPARC instruction set architecture (ISA) is extended with timing instructions that provide lower and upper bounds for the execution time [16]. Finally, time predictable architectures are accompanied with a set of WCET tools and a compiler that is specific for these architectures, in order to interpret the timing instructions appropriately (i.e. time blocks [17]).

On the contrary, Alan Burns et al argue that it is not necessary to have time predictable subsystems in order to have a time predictable system [18]. It proposes to have hardware components with independent random behavior (i.e. random replacement policy in caches). In this way the system can be analyzed more easily through probabilistic methods.

3.2. Fermi GPU timing analysis

To analyze a processor at a low level, it is required to have accurate documentation that describes all the design choices for the architecture. The Nvidia Fermi whitepaper does not provide details about the architecture [1]. More details regarding the control flow, cache and TLB (translation lookaside buffer) hierarchy for the Tesla architecture through detailed microbenchmarking are given in reference [2]. The results can be used as an indicator for the Fermi architecture. Furthermore, a cycle accurate simulator that mimics the behavior of the Fermi architecture is implemented in [3] [4]. A throughput analysis that is made with microbenchmarks implemented in assembly level is given in [5]. This paper also describes how a register file bank conflict may be produced for the Kepler architecture [6]. First we are going to describe the available literature that was performed for WCET analysis on GPUs. Afterwards, we are going to discuss some studies for average performance analysis execution time.

A hybrid analysis for GPU kernels is presented in paper [24]. The analysis was built as a tool on top of GPGPU-Sim. Hybrid analysis inserts time stamps at particular program points, called instrumentation points. The testing method was based on paper [25] and traces were produced for the kernel. The traces are a sequence of tuples of the shape (i_j,t_j) where i_j is an instruction point identifier and t_j is the time of execution. The traces are processed to calculate the WCET. This work does not analyze the hardware architecture. It is more general so it does not take into account any of the specific features of the GPU architecture.

Next, a static timing analysis for GPU kernels based on the method abstract CTA (cooperated thread array) simulation that they introduce is presented in [26]. The work is focused on one thread block only. Static divergence analysis was performed. The mini-SIMT machine based on the research work [27] was used. Although, no experimental verification was presented that uses this method.

Finally, an analytical model for the execution time of GPU programs is proposed in the work of Hong and Kim [28]. The components of the model are to estimate the memory warp parallelism (number of parallel memory request from different warps) and the computation warp parallelism, the number of warps that can be executed during a memory access. The model is general and does not make any assumptions for the architecture. The results have been compared with multiple architectures with 5.4% and 13.3% error for microbenchmarks and GPU computing application respectively.

4. Time predictability for GPUs

An introduction to time predictability is given in this section. The different interpretations of time predictability in literature and in this work are presented. Next, the hardware components of the Fermi GPU architecture are discussed for time variable behavior.

4.1. Introduction to time predictability

Real-time systems are computing systems that have to react within precise timing constrains. Thus, the correctness of a real-time system depends not only on the value of the computation but also on the time that the results are produced [8]. In the context of real-time systems a worst case execution time (WCET) analysis is performed in order to guaranty a stable and analyzable behavior of a system, while modern architectures are optimized for the common case (average case). To optimize the common case, complicated techniques are used like speculated methods (branch prediction, prefetching etc.), out-of-order execution etc. Even though, these methods improve the performance of the system significantly; they are very difficult to be analyzed through WCET analysis methods.

To clarify more, an example based on the examples given in [15] is presented in Figure 11. First, in this example the best case execution time (BCET), average case execution time (ACET) and WCET are consider to be the execution times measured. In addition, the observed execution time is measured through experiments for a specific data input on a specific processor. The lower and upper bound is used as the theoretical calculation of the best and worst case execution times. In order to provide a system that would always respect the timing restrictions, the upper bound should be used as worst case guaranty.

Initially, the processor A is a modern processor which is optimized for the common case. It uses mechanisms that improve the average case, like caches. Caches are difficult to be analyzed because they are hardware controlled and the prediction of a miss requires detailed knowledge of the current state of the cache. In addition, architecture A has an average case execution time (ACET) which is close to the best case execution time (BCET). Since it uses mechanisms that are difficult to be analyzed this means that the difference between the worst case execution time and the upper bound is significantly large since more pessimistic choices have to be made which will lead to a more conservative and less accurate model.

The architecture B in Figure 11 is a simple architecture that does not use mechanisms which are difficult to be analyzed. The average case is higher compared to processor A since it does not use mechanisms that improve the common case through mechanisms that are difficult to be analyzed (i.e. branch prediction, prefetching). The average case is more balanced between the BCET and WCET compared to processor A. Furthermore, since the architecture is simpler, it means that can be analyzed more precisely, so the difference of the WCET and the upper bound is smaller compared to processor A. From this example it is observed that, even processor A has a smaller WCET compared to processor B, the upper bound of processor A is higher compared to processor B. So although processor B is slower on average, it can provide better worst case guaranties.

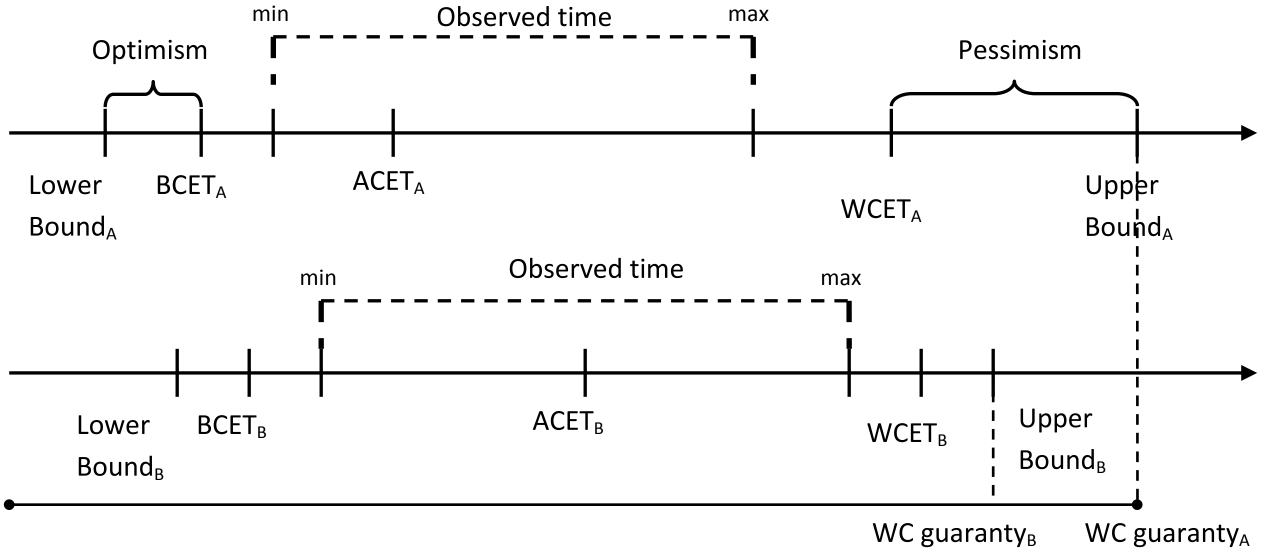


Figure 11 Time distribution for two types of processor. Processor A is modern processor optimized for the average case. Processor B is a simple processor that can be analyzed more precisely.

A time predictable architecture should have a limited number of states. In this way it can be analyzed for any possible scenario. On the contrary, WCET analysis for a complex architecture, like processor A, it might be infeasible. For WCET analysis all the possible states of the system have to be analyzed. For a complex system this leads to state space explosion and make the analysis infeasible [29].

To decompose the problem, the timing barrier that is proposed in [10] can be used. If it is possible to bring the system in a known state periodically by using the timing barrier then the state space is divided in smaller analyzable pieces.

For instance, a way to implement a mechanism like that on a GPU would be: periodically transfer parts of the code and data that are needed for a specific amount of time to the scratchpad memory. The scratchpad memory is software controlled and has time predictable access times. So, the system would work on code and data that are in a predictable memory, while at the background the data can be transferred from the global memory to the scratchpad memory.

Currently there is no stable formalization for time predictability in literature. Formalizations of the time predictability are compared and a new formalization is also introduced in paper [10]. A new formalization for time predictability is proposed in [11] where it differentiates the sources of unpredictability to software and hardware related. Furthermore, [12] introduces the deviation of cycles per instruction (CPI) as a metric for time predictability. All of these formalizations are general and they do not make any assumption regarding the type of the architecture, so they can be used also for GPUs. In addition, in [13] it is reported that time predictability depends on the hardware, the compiler and the WCET tools but no formalization is given.

Based on the observation until now, we distinguish the following cases for the different type of executions:

- **Constant timing behavior**: A program with the same input executed on the same architecture always takes the same execution time. In other words the WCET and BCET is the same and the execution time can be precisely statically predicted (WCET = BCET).

- **Variable tight bounded timing behavior**: A program with the same input executed on the same architecture can have a deterministic number of different execution times. The execution time for all the different cases can be precisely statically calculated.

- **Infeasibility of the timing analysis:** The complexity of the architecture is so high that the analysis is not feasible. In order to provide precise timing guaranties it is required to investigate all the possible states of the system. This type of analysis for a complex system leads to state space explosion and the analysis is not feasible.

- **Limited timing information of the architecture:** In this case we do not have the required timing information of the architecture in order to provide precise estimation of the timing behavior.

4.2. Analysis of hardware components for timing variation

The sources of time variations in Fermi GPU architecture are presented in this section. The main hardware components of the architecture are presented and analyzed for sources of

Warp scheduler and scoreboard

The Fermi architecture has two warp schedulers (odd and even) per SM and one instruction dispatch unit per scheduler. The scheduler chooses one warp of 32 threads based on the scheduling algorithm and issues it to one of the set of 16 SP, 16 load/store units or 4 special function units. The warps are executed in two cycle lock step fashion. Initially, the first 16 threads of the warp are executed and at next clock cycle the other 16 threads are executed in the same SPs.

Figure 12 shows an example of the two cycle lock step execution of two SP instructions. At the first clock cycle (CC1) the first 16 threads of the two warps are executed at the two SIMD lanes. At the second cycle (CC2) the next 16 threads are executed at the same SIMD lanes. A similar procedure is followed for the load/store units. So every two clock cycles, two warps from different schedulers can be executed. However since we have only 4 SFU it is required to perform 8 steps in this special case.

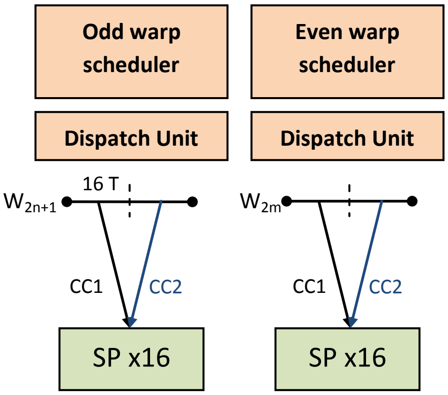


Figure 12 Example of two-lock-step execution for two warps.

The warp schedulers of the Fermi architecture use the loose round robin (LRR) algorithm. When a load from the off-chip memory is executed the kernel continues with the next instructions until the instruction that has read after write (RaW) dependency with the load instruction. If there are enough instructions to hide the memory latency the warp is scheduled in RR. If the data from the load are not ready, the warp stalls and another warp is scheduled. The memory latency can be hidden if there are enough warps that can be executed. The choices of the warp scheduler can change the scheduling order and as a result the overall execution time. In Fermi architecture uses the Loose Round Robin algorithm. When a warp stalls, another warp is scheduled.

To sum up, the warp scheduler, checks the scoreboard for data dependencies. Next it checks the available execution units in order to choose the instruction that can be scheduled. Furthermore, it checks the address of the instructions in order to preserve the order of the instruction, since the instructions of a warp have to be executed in-order. Finally, based on the scheduling algorithm chooses which warp to schedule (Figure 13).

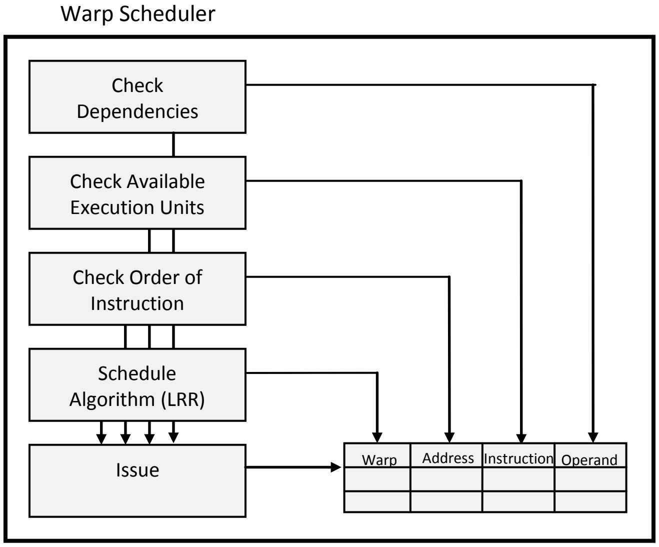


Figure 13 Overview of the warp scheduler.

The scoreboard is a buffer shared between the two warp schedulers and. In the scoreboard there can be many entries for every warp. An example, based on the scoreboard organization given in [38], of how the scoreboard is updated is given below. If the scoreboard algorithm updates the information in out-of-order then it can be different from execution to execution if the requests for the update arrive in a different way. If the update requests arrive in a different way from execution to execution then the warps will be ready in a different order from execution to execution. Consequently they will be scheduled in a different order from execution to execution.

```
0. LD.E  R0, [100]
1. ADD   R3, R2, R1
2. MAD   R7, R3, R0, R4
```

Let's assume that we have 3 warps per scheduler and a scoreboard with 6 positions.

Warp	Address	Instruction	Operands		
0	0	LD.E	-	-	1
2	0	LD.E	-	-	1
4	0	LD.E	-	-	1
0	1	ADD	-	1	1
2	1	ADD	-	1	1
4	1	ADD	-	1	1

Initially, the first 2 instructions of the 3 warps are in the scoreboard. Both instructions can be executed since there is no data dependency. The instructions are executed in order so the load to the global memory (LD.E) is going to be executed first. We assume that the warps are chosen with LRR algorithm. So W0 is going to be executed first, after 2 cycles W2 and after 4 cycles W4.

Warp	Address	Instruction	Operands		
0	0	ADD	-	1	1
2	0	ADD	-	1	1
4	0	ADD	-	1	1
0	1	MAD	0	0	1
2	1	MAD	0	0	1
4	1	MAD	0	0	1

Since LD.E has been executed, it is removed from the scoreboard and multiply-add (MAD) instruction is inserted. MAD has RaW dependency with LD.E and ADD so the operands are not ready (0).The ADD instruction can be executed since it does not have dependency with LD.E.

Warp	Address	Instruction	Operands		
0	1	MAD	0	1	1
2	1	MAD	0	1	1
4	1	MAD	0	1	1

Now that the ADD instructions is finished the register "R3" is ready so the field in scoreboard for this source register is updated to 1. We assume that LD.E has big latency so the source operand that has RaW dependency with LD.E is not ready yet. Therefore, the MAD instruction can be executed.

Warp	Address	Instruction	Operands		
0	1	MAD	1	1	1
2	1	MAD	1	1	1
4	1	MAD	1	1	1

Now we assume that the global memory respond with the data and the register "R0" is ready. The MAD instruction can now be executed.

Operand Collector

The register file of the Fermi GPU architecture is huge compare to the state-of-art CPUs. It has 32K of 32bit register which is 128KB. Since there are 16 SM in total there are 2MB are used for the register file. To save energy and area and to have the appearance of multiple ported register file is implemented as multiple banks of single ported RAM. An arbitration mechanism is used to minimize the register file bank conflicts. The arbitrator queues the warps and selects up to 4 register file requests that do not produce bank conflicts. The algorithm of the arbitration mechanism is not provided by the Nvidia. The arbitration mechanism can influences the time behavior of the GPU since it can determine the execution order of the warps. If the choices of the arbitration are not deterministic then the order of the warps can change from execution to execution.

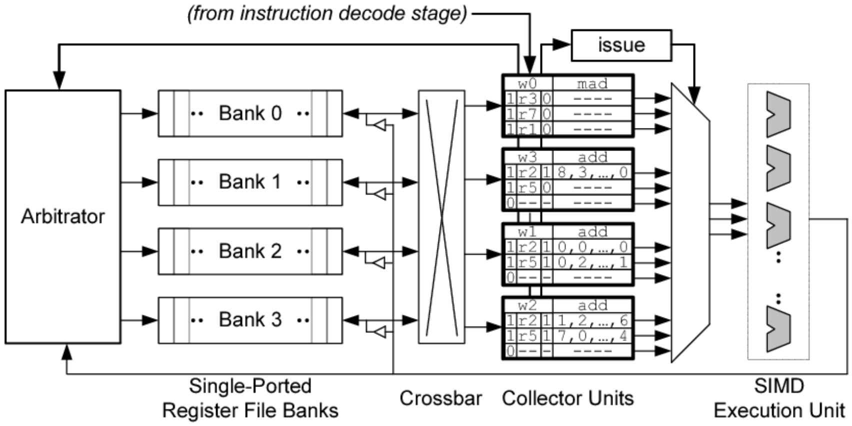


Figure 14 Operand collector taken from [4].

On-chip scratchpad memory

In the Fermi architecture, scratchpad memories are available. Scratchpad memories are software controlled and have time predictable behavior compared to the caches while they have similar performance. The programmer is responsible to allocate the data to the correct memory in order to have the desired performance while in caches the memory management is performed automatically by the hardware.

The scratchpad memory has 32 banks that can be used in parallel by 32 threads (one warp). When all the threads access a different bank, the access time is deterministic and is always the same for all the threads in a warp. A bank conflict occurs when multiple threads of a warp access a different address that belongs to the same bank. The execution time varies depending on the number of bank conflicts but it is still deterministic.

The address patterns determine the number of the bank conflicts. A modulo 32 operation on the address of the shared memory is performed in order to find which bank is going to be accessed. For example Table 2 Table 3 and Table 4 show the bank conflicts for stride 4, 16 and 32 of the first 8 threads of a warp. The tables are organized as follows; the thread ID of the threads in the warp is presented in the first column. Next, the memory address is shown in the second column and the last column show the bank number. For stride 4 there is a bank conflict between thread 0 and thread 8. Both threads of the same warp access bank 0. Similarly, for stride 16 threads even threads access bank 0 and odd threads access bank 16 so in total 16 bank conflicts for a warp of 32 threads. Finally, for stride 32 all threads access bank 0 so there are 32 bank conflicts.

Table 2 Bank conflicts for stride 4

Thread ID	Address	Bank
0	0	0
1	4	4
2	8	8
3	12	12
4	16	16
5	20	20
6	24	24
7	28	28
8	32	0

Table 3 Bank conflicts for stride 16

Thread ID	Address	Bank
0	0	0
1	16	16
2	32	0
3	48	16
4	64	0
5	80	16
6	96	0
7	112	16
8	128	0

Table 4 Bank conflicts for stride 32

Thread ID	Address	Bank
0	0	0
1	32	0
2	64	0
3	96	0
4	128	0
5	160	0
6	192	0
7	224	0
8	256	0

If the number of bank conflicts can be determined at compile time, then also the execution time can be determined. Techniques with deterministic behavior [30] that minimize the bank conflicts improve the execution time without increasing the complexity of the system.

The analysis to determine the number of bank conflicts increases the complexity of the whole analysis of the GPU. There are cases where the scratchpad memory addresses cannot be defined at compile time. For example, the index to the shared memory is given as input to the kernel. In this case it is required to assume the worst case and count all the unknown addresses as bank conflict. This approach leads to a pessimistic assumption for the worst case execution time.

The shared memory can also used for atomic operations. For an atomic operation, first it is required to lock a memory location in the shared memory. Next the operation is performed and finally the result is written back and the memory location is unlocked. If multiple threads are trying to perform an atomic operation to the same location, they have to be serialized. A detailed model for atomic operations can used [30] to precisely define the execution time of a kernel with atomic operations

Caches

Caches are one of the biggest sources of time variations for CPUs. Some of the works that analyze the problems of time predictability with caches are presented in [7, 9, 10, 15, 20, 25]. Currently, GPUs hide the memory latency by executing concurrently many threads. For GPUs the performance does not depend on caches like CPUs. Typically the scratchpad memory and high level of concurrent execution of threads are used to hide the memory latency of the global memory. So it is programmer's responsibility to allocate the data in the fast memory. L2 cache is used for communication. Since, L2 cache belongs to the global memory and it is common for all threads. The replacement policy that is used for all the caches in Fermi GPU architecture is not described in [1]. LRU replacement policy is assumed for Tesla architecture [2]. It has been shown that LRU is the replacement policy that is more suitable for predictability [20], while in [19] it is proposed that locally deterministic replacement policies can favor time predictability.

The Fermi architecture has two levels of instruction and data caches. There are separate level 1 instruction and data caches. Having the instructions and data in different memories improves the time predictability of the system since there is not interference of instructions and data. On the contrary level 2 data cache is unified for instructions and data. The analysis on unified caches is more complicated since we cannot separate the analysis for the instructions and data.In addition constant cache is used to keep the data that have values that are not changing. The separate memory for constants, favors time predictability since data that cannot change can be placed there and the memory accesses can have deterministic access time. Since a dedicated memory is used for this type of data it is expected to have smaller miss rate compared to a typical data cache.

GPUs are becoming more and more general purpose so it is expected to see caches to play a more important role in the future. Scheduling algorithms that take advantage of inter-warp and intra-warp locality have been proposed to use efficiently the cache, for example [22]. To extent our analysis in order to take into account the cache it is required to have a separate model for the caches like [37]. By using a cache model we can calculate the hit rate and the miss rate and as a result more accurate estimations for the best case and the worst case execution times can be given.

SIMT stack - Control flow mechanism

The control flow in the GPU architectures is different compared to a CPU. Predicate register and stacks are used in order to keep track of the divergence and convergence points of a thread block. The instruction in an IF-THEN-ELSE statement are executed by the SIMD Processor but only some of the SIMD lanes are enabled for the THEN instructions and some lanes for the ELSE instructions.

In other words, those lanes with the predicate set to true execute the instructions and store the result, and the other SIMD Lanes don’t perform an operation or store a result. For the ELSE statement, the instructions use the complement of the predicate (relative to the THEN statement), so the SIMD Lanes that were idle now perform the operation and store the result while their formerly active siblings don not. At the end of the ELSE statement, the instructions are unpredicated so the original computation can proceed. Thus, for equal length paths, an IF-THEN-ELSE operates at only 50% efficiency. A similar procedure is followed for the nested IF statements.

This control flow scheme is advantageous for the time predictability of the system since does not use any speculative method like branch prediction and the execution time of a branch can easily be predicted. In addition, the use of the predicate registers simplifies the control flow analysis and as a result the timing analysis of the GPU.

Off-chip global memory and interconnect

The global memory is GDRAM off-chip memory (Graphics Dynamic Random-Access Memory). It is a special case of DRAM optimized for high bandwidth. It is common for all the SMs. To calculate the latency of the global memory it is required to know the state of the memory, the contention of SMs interconnect and the interference between the accesses form the different SMs. Therefore, the access time to the global memory can be varied from execution to execution.

GPU has a separate off-chip memory from the CPU as mentioned, before. It is shared between all the SMs. The memory accesses can be coalesced which means that accesses to consecutive memory location can be combined and performed as one big memory access. The size of a coalescing memory access is 128 bytes. In GPGPU-Sim, the memory controller is configured as an Out-of-Order (OoO) First Ready First-Come First-Serve controller. We have quantified the variation of the execution for a subset of these sources of variation. More details are given in Section 5.3.

4.3. Conclusion

In this section we have made an introduction to time predictability. We categorize the different cases of time predictability based on the timing behavior of programs that execute on same architecture with same inputs. Next, the main hardware components of the Fermi GPU architecture were presented and discussed for time variations.

We chose to analyze the warp scheduler since the warp scheduler determines the starting time of the instructions and influences the total execution time. In addition, we chose to analyze also the scratchpad memory and the global memory instructions since they are frequently used in GPU applications.

Caches can influence significantly the variation in the execution time but we did not treat them in this report. Caches have high complexity and require a lot of time to analyze them in detail. L1 caches have been analyzed in [37]. The model for the caches can be intergraded with this work in the future. Furthermore, L2 cache is required also to be analyzed. L2 caches are even more complex to analyze them than L1 since they are shared between the SMs and unified for instructions and data. To avoid the complexity of L2 cache we would design the L2 cache to have random behavior. By having random behavior we can analyze them with probabilistic methods [18].

5. Quantification of time variation in Fermi GPU architecture

In this section we focused on the warp scheduler, shared memory and global memory and we quantify the time variation for the real hardware. The results of the experiments compared with the GPGPU-Sim cycle level simulator.

5.1. Experimental Methodology

To analyze the behavior of the GPU hardware architecture at a low-level, it is required to have detailed and accurate information. In some cases, the documentation is not available or precise. In order to describe the worst case scenario of the system it is required to be able to define all the possible states that the system can be in, therefore the limited documentation makes the analysis for WCET very difficult.

Since for Fermi architecture the timing details are not available, we have to analyze its behavior through experiments. The goal of the experiments in this chapter is to identify and quantify the sources of the unpredictability in a GPU. The experiments have been performed through microbenchmarking. The microbenchmarks are implemented in assembly language with the use of the "asfermi" [36] assembler.

The code below presents the measurement method of the execution time for the microbenchmarks. A counter is incremented at the half of the SM clock frequency and is stored in a 64-bit special register. From this register we read the 32 least significant bits with the "SR_ClockLo" (line 2) and we store this value to general purpose register "R0". The value of "R0" is shifted left by one (multiply by 2) and stored to "R4" to get the current value of the clock (line 3).

The same procedure is followed after the execution of the code (lines 7 and 8). Finally, the values of registers "R5" and "R4" are subtracted to find the execution time (line 9). We have measured that the calculation of the current value of the clock followed by a non-dependent operation takes 30 cycles.

```
1.   ...
2.   S2R R0, SR_ClockLo;
3.   SHL.W R4, R0, 0x1;
4.
5.   // code to be measured
6.
7.   S2R R1, SR_ClockLo;
8.   SHL.W R5, R1, 0x1;
9.
10.  IADD R6, R5, -R4;
11.  ...
```

To verify the behavior of the hardware we performed the same experiments at the GPGPU-Sim. The direct usage of assembly language in GPGPU-Sim is not supported. The microbenchmarks were implemented in the parallel thread execution (PTX) instruction set. PTX is an intermediate instruction set that is compatible with multiple GPU generations. The PTX code is optimized and translated to the target-architecture instructions. For our purpose the compiler optimizations do not affect the analysis due to the small number of instructions. Thus, the code in “asfermi” for the real hardware and the PTX code for the GPGPU-Sim are comparable.

The execution time was measured through inline assembly. The code is similar with the “asfermi” code and is presented bellow. Initially the registers are declared. Similarly with the “asfermi” code, we read the clock value before and after the code that we want to measure (time1 and time2). The values are subtracted and the result is stored in variable “%0”. The variable “%0” is reserved for the first output of the inline code. In this case “%0” is mapped to the variable “time_res”.

```
asm (
    "{\n\t"
      ".reg .u32 time1;\n\t"
      ".reg .u32 time2;\n\t"
      //Register declaration

      "mov.u32 time1, %clock;\n\t"
      "shl.b32 time1, time1, 1;\n\t"

      //code to be measured

      "mov.u32 time2, %clock;\n\t"
      "shl.b32 time2, time2, 1;\n\t"

      "sub.u32    %0, time2, time1;\n\t"
      "}"
    : "=r"(time_res)
  );
```

The tool chain to automate the procedure of the experiments is presented in Figure 15. Initially, the code is compiled and the executable is produced. With the use of “cuobjdump” tool the assembly code (.asm) is produced from the executable. The code is modified in order to be in shape that we can insert the “asfermi“code. A Python script produces and injects the assembly code that we want to measure. By using the “asfermi” assembler we produce the CUDA binary (.cubin). The code is compiled with “NVCC” to produce the executable.

A python script runs the executable and collects the timing measurements. The results from the timing measurements for the all the threads are collected. Another Python script formats the timing measurements in such a way that it is easy to plot them.

The procedure is automated and can be configured for the real hardware or the GPGPU-Sim. The user has to specify the code for measurement by using the functions implemented in Python. Finally, the thread block size, number of thread blocks and the number of repetitions of the experiments can be tuned to perform different types of experiments.

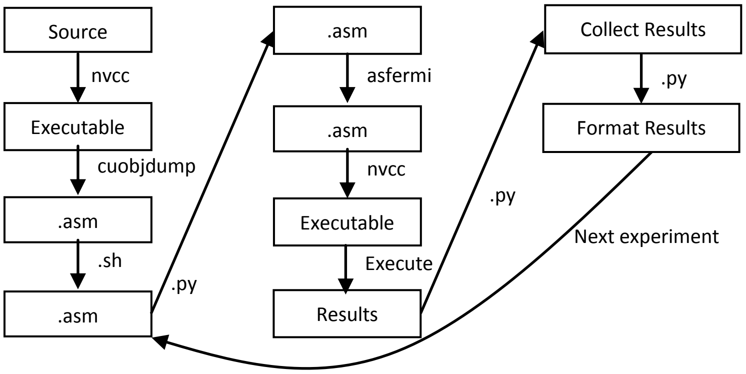


Figure 15 Tool chain to perform and automate the experiments.

5.2. Warp scheduler

First, we are using a thread block size of 32 threads, so there is only one warp. Two microbenchmarks with two “ADD” instructions were used. The first microbenchmark has two “ADD” instructions with no data dependency. The second microbenchmark uses two “ADD” instructions with a RaW dependency. The results are presented in Figure 16. The vertical axis shows the execution time in clock cycles and the horizontal axis shows the number of instructions. The dark blue line is the execution of the instructions without a data dependency and the light blue is the execution with a RaW dependency. From this graph we can see that in both cases the execution time is increased linearly with a linear increase of the number of the instructions. The execution with the RaW dependency has higher slope since the warp has to wait until the results from the instruction with the dependency are ready.

From the CUDA programming manual [34] we know that we can schedule a new warp every two cycles. Figure 16 shows that in the case that we have only one warp, a new instruction is scheduled after 6 cycles without dependency. From this we conclude that the HW can only schedule instructions from the same warp every 6 cycles. Other experiments show that multiple warps can be scheduled every 2 cycles. Furthermore, Figure 16 shows that when there is RaW dependency the latency increases to 18 clock cycles. From this we can conclude that the additional cost of a RaW dependency is 18-6 = 12 clock cycles.

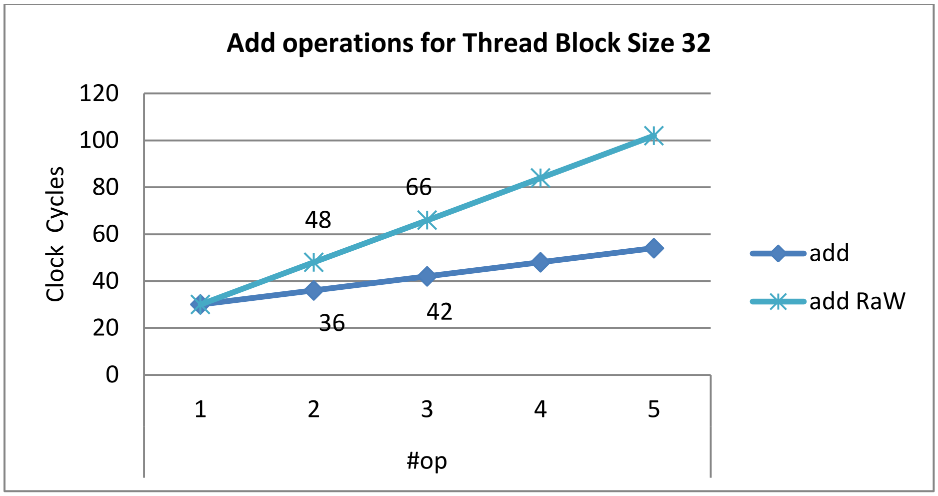


Figure 16 The execution time of two micro-benchmarks with two add instructions are presented. In the first micro-benchmark the instructions are independent while in the second one they have RaW dependency.

Instructions in a warp are executed in order. Since, the scheduler has to respect the order of the instructions, there is only one schedule. On the contrary for more than one warps per scheduler the situation is different. Warps are independent among each other, so they are scheduled dynamically without the need of any specific order. The scheduler dynamically chooses which warp to schedule based on the availability of the warps. Different scheduling decisions lead to different execution times. This behavior influences the time predictability of the system.

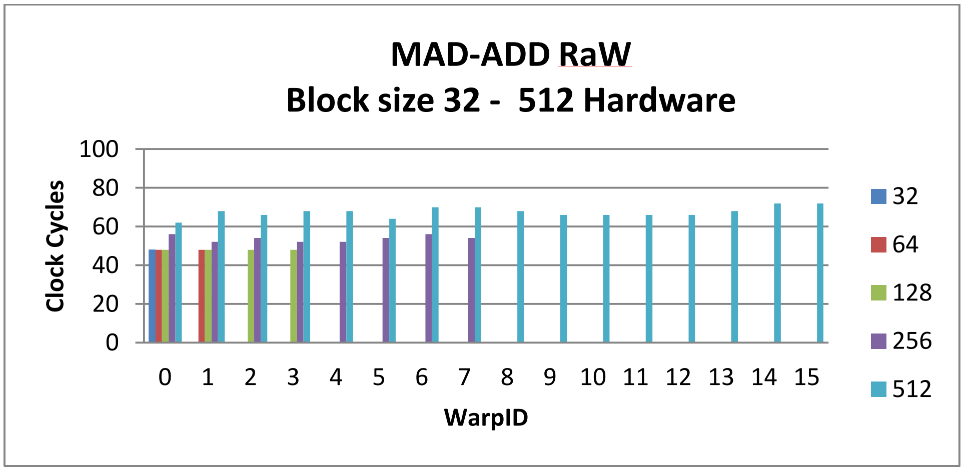


Figure 17 mad-add instructions with RaW dependency for different thread block sizes. Experiments on hardware

Figure 17 shows the execution time of two instructions for six different thread block sizes. For this microbenchmark, we use a multiply-add instruction ("MAD") and an "ADD" instruction that have read after write (RaW) dependency. The horizontal axis presents the warp ID of the warps. The vertical axis shows the execution time in clock cycles. We use thread block sizes of 32, 64, 128, 256 and 512 threads.

First, it is presented the execution time for thread block sizes 32, 64, 128 is the same for each warp. A warp can be scheduled every 2 cycles but an instruction from the same warp can be scheduled every 6 cycles (this behavior is explained detailed in section 6). Consequently, between two instructions of a warp “A”, 2 other warps can be scheduled on the same scheduler without influencing the execution time of “A”. For thread block sizes 256 and 512 that have more than 6 warps, the execution time of all warps increases and varies.

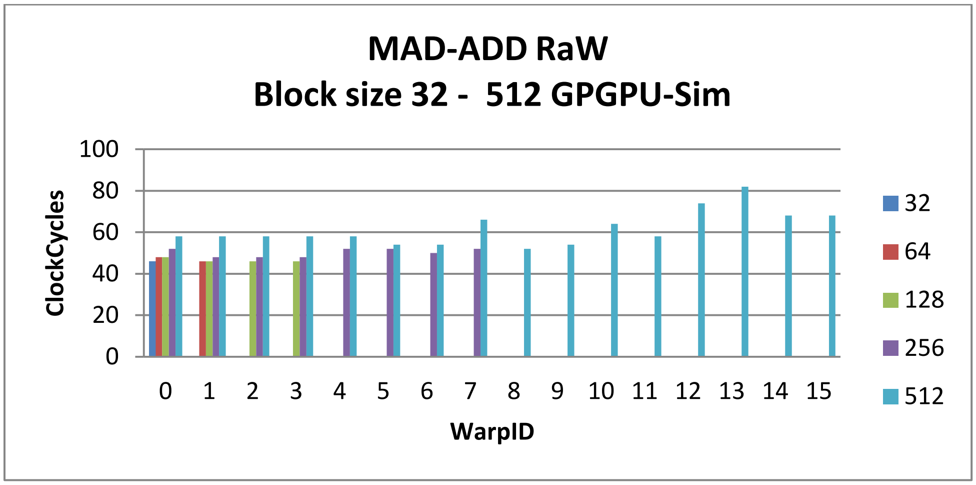


Figure 18 mad-add instructions with RaW dependency for different thread block sizes. Experiments at GPGPU-Sim

Figure 18 shows the same experiment for GPGPU-Sim. The horizontal axis shows the warp ID of warps for the different thread block sizes. The vertical axis shows the execution time in clock cycles. First the execution times for 64 and 128 thread block sizes are different compare to 32. This means that the scheduler modeled in GPGU-Sim schedules the warps every 2 cycles without taking into account the restriction of the 6 cycles.

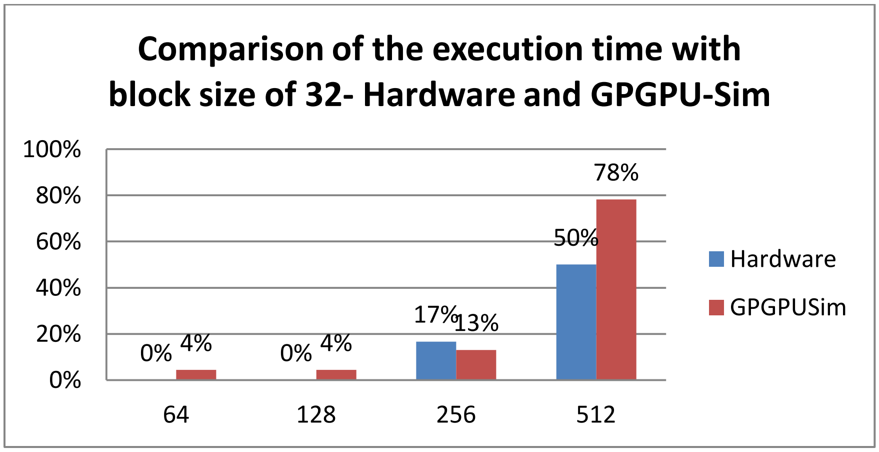


Figure 19 Difference of the execution time of thread block size 32 with the slowest warp for the different thread block sizes

Figure 19 presents the difference of the slowest warp and the warp of the 32 threads configuration for the real hardware and the GPGPU-Sim. The horizontal axis shows the thread block size and the vertical axis the percentage of the difference. First for thread block size of 64 and 128 threads the difference of the slowest warp and the configuration of 32 threads is 4% for the GPGPU-Sim while for hardware is 0. In addition, the behavior of the warp scheduler for 256 thread block size is similar to hardware; the biggest difference is 4%. For thread block size 512 the simulator seems to behave differently compare to the hardware. The difference of the execution time of the slowest warps with the execution of thread block size 32 is significant bigger. For real hardware the biggest difference is 50% while for the simulator is 78%. Therefore, the real hardware is more scalable in terms of number of warps compare to the GPGPU-Sim. In other words, the real hardware can maintain better performance for the increasing number of warps.

The execution time of the different warps is more balanced in real hardware compared to the GPGPU-Sim. The scheduler in hardware seems to be fairer between the different warps. All the warps experience approximately the same amount extra delay compared to the case with thread block size 32. On the contrary, in case of thread block size of 512 threads for GPGPU-Sim warp 12 and 13 are much slower compared to the other warps.

Now that we have a better insight of the warp scheduler we are going to examine it for timing variations. By performing the same experiments multiple times we notice that for a thread block size 512 threads the execution time can differs from experiment to experiment. More precisely, for thread block size 512 we repeat the experiment 100 times. The results are presented in Figure 20. The black line in the graph represents the common case. From the 100 experiments 94 of them had the same execution times. The rest of the experiments had different execution times and there is no common pattern.

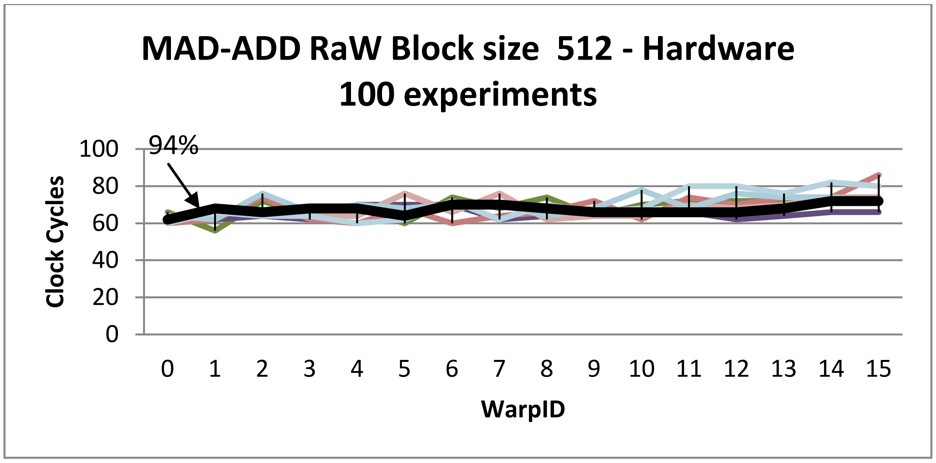


Figure 20 100 experiments for mad add instruction with RaW dependency executed by 512 threads (16 warps)

One of the experiments with different result was isolated to analyze its behavior in more detail. To visualize the results better, the starting times for the common case and the exception are presented in Table 6 and Table 5 respectively.

The first column presents the relative starting time of the warp. The “Odd” and “Even” columns contain the IDs of the warps that were scheduled by the odd and the even schedulers. The green boxes show the point in time that warps reach the first “S2R” instruction of the microbenchmark (line 2 of asfermi code).

Since in hardware we cannot see which instruction is executed in every cycle, the red boxes presents the cycles that we do not know exactly what is executed. The red box can be an instruction that we cannot identify at which cycle is executed or an idle cycle. Note that for every exception a different warp is scheduled first. Here we check only one case since we did not found any common pattern for the exceptions.

Since warps are scheduled based on in LRR fashion based on the warp IDs we expected that warp 0 would reach the “S2R” instruction first, however for the common case of even scheduler, warp 6 reaches it first. For the exception case the warps from the odd scheduler reach the “S2R” instruction 30 clock cycles later compared to the warps from the even scheduler. In addition, the warps are executed in a different order compared to the common case, which shows that an event changed the scheduling order.

In hardware it is not possible to identify what event changed the scheduling order. We use the GPGPU-Sim to trace all the instruction at cycle level. The simulator has a deterministic behavior and the results are going to be the same in every execution. The exception case cannot be replicated, so we worked on the common case only. For the common case we found all the sources of events that change the scheduling order. We assume that in the exception case there will be the same types of events but in a different quantity and at different point in time.

The simulator was modified to collect the information from the two schedulers. At every clock cycle we monitor which instructions were executed. We performed the same experiment in the simulator as on real hardware and we collect the traces. Some characteristic parts of the execution traces, are presented in appendix C. From the analysis of the traces we can find out what exactly is executed between the “S2R”.

Firstly, from the traces of GPGPU-Sim we first observe that the warps are executed in order based on their warp IDs. This behavior was different in hardware. Warp 6 reach the “S2R” instruction first in hardware while in GPGPU-Sim warp 0 does so. Secondly, from the information that we collect, the warp scheduler could be in these three states.

- **Active.**
  The GPU executes the instructions from the different warps. After checking for RaW dependencies at the scoreboard the scheduler can schedule a new warp every 2 clock cycles. In other words the warp scheduler find available warps that can be scheduled.

- **Idle because of RaW dependencies.**
  The scheduler cannot find any available warp to schedule therefore it has to wait until a RaW dependency is resolved. If all warps are scheduled fairly, so strict RR, then all warps have to wait the same amount of time for their RaW dependency.

- **Idle because the execution units are not available.**
  The scheduler before schedule a new instruction checks if the SP and SFU are available. More precisely, it checks the register between the “issue” and ”execute” stage. If the execution units are not ready then the scheduler is not able to issue a new warp and it has to wait. For instance

when two warps have to executed a load or store instruction, there is only 1 set of LD/ST units, therefore, one of the warps has to wait.

From these states we relate the exception cases with the third state. When a warp stalls, it loses its turn. If the stall is not constant then the warp waits different amounts of time so it can have a different schedule.

Another explanation for the exception cases can be the register file bank conflicts. The register file is in the operand collector and it is organized in banks. The arbitration mechanism that avoids the bank conflicts in the operand collector is not described in the documentation of Nvidia. The register file bank conflicts can introduce extra stall to the warps and change the scheduling order. However, in hardware we could not produce register bank conflicts by using the access patterns that was used to produce register bank conflicts for Kepler [5].

Table 5 Starting times of the warps for the two schedulers, common case

Clock Cycles	Odd	Even
0	🟥	6
2	🟥	🟥
4	🟥	🟥
6	1	🟥
8	🟥	8
10	3	0
12	🟥	2
14	🟥	🟥
16	🟥	10
18	5	4
20	9	🟥
22	11	🟥
24	13	12
26	7	14
28	15	🟥

Table 6 Starting times of the warps for the two schedulers, exception case

Clock Cycles	Odd	Even
0	🟥	2
2	🟥	🟥
4	🟥	🟥
6	🟥	🟥
8	🟥	6
10	🟥	🟥
12	🟥	🟥
14	🟥	🟥
16	🟥	🟥
18	🟥	🟥
20	🟥	0
22	🟥	🟥
24	🟥	🟥
26	🟥	🟥
28	🟥	🟥
30	1	🟥
32	11	4
34	3	🟥
36	🟥	🟥
38	5	🟥
40	🟥	8
42	7	10
44	9	12
46	13	🟥
48	15	14

5.3. On-chip scratchpad memory

In Figure 21 the latency of 10 loads (LDS) and stores (STS) is presented. The horizontal axis presents the number of operations (LDS and STS) and vertical axis presents the time in clock cycles. In both cases the execution time is increased linearly with linear increase of the number of operations. Furthermore, it is shown that the store has higher sloop compared to load. A new load and store can be performed to the shared memory after 26 and 30 clock cycles respectively.

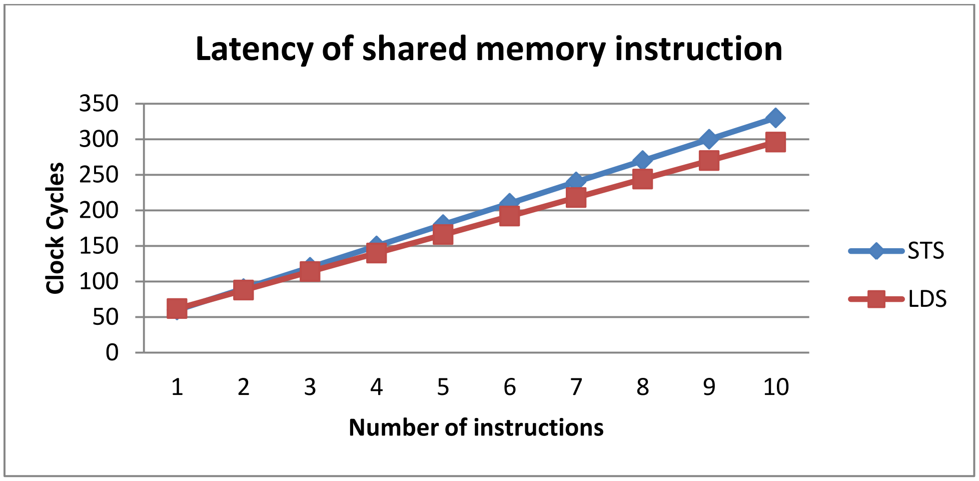


Figure 21 Latency for shared memory instructions

Figure 22 shows the execution time of memory access to the shared memory for one warp when we have bank conflicts. On the horizontal axis is the stride and on the vertical axis is the execution time in clock cycles. It can be seen that the execution time of load and store to the shared memory increases linearly with the number of bank conflicts. For stride 64 and 128 we have the same number of bank conflicts with stride 32 so the execution time remains constant.

In addition, when we performed a store and a load to the shared memory with RaW dependency the execution time is almost the same (24 cycles higher) compared to accumulating the load and store execution time. So the execution time for the bank conflicts remains the same when loads and stores are used interleaved.

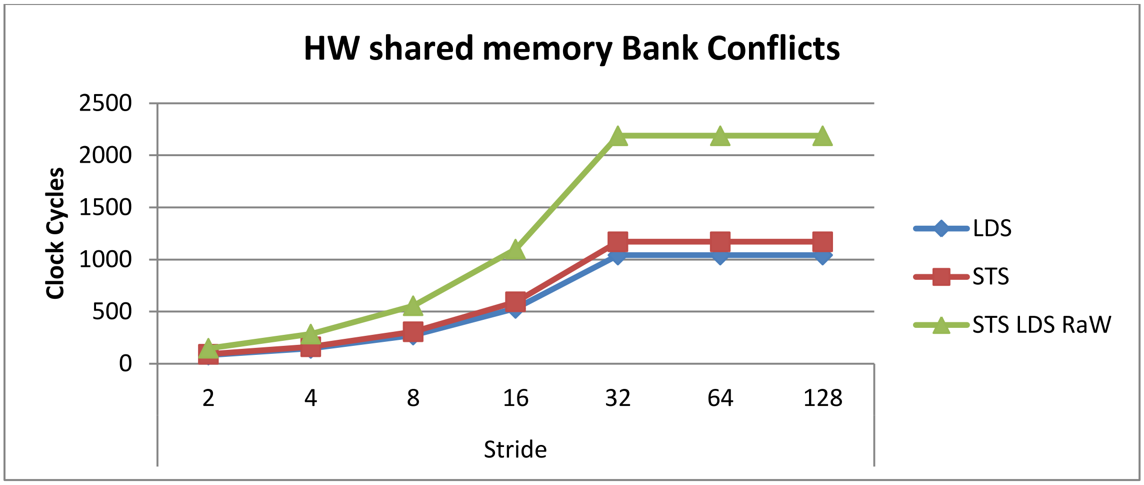


Figure 22 Execution time of bank conflicts to the shared memory.

5.4. Off-chip - Global memory

Similarly to the shared memory the latency to schedule loads and stores to the global memory is presented in Figure 23. First, it shows that the load takes much more time compare to the store because of the write buffer that we mention before. The slope of the lines defines the time that is needed to schedule a new instruction. So, for load and store we have 370 and 34 clock cycles respectively.

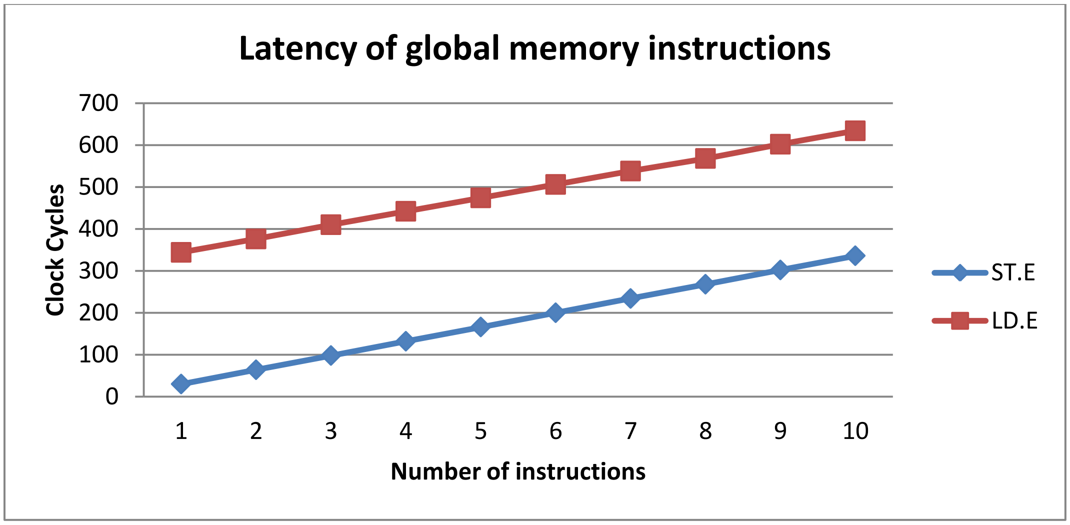


Figure 23 Latency for global memory instructions.

Figure 24 presents the variation of the execution time from experiment to experiment of the load instruction to global memory for thread block size of 512 threads. The threads access consecutive memory accesses. The horizontal axis show the warp ID of the warps and the vertical axis show the execution time in clock cycles. All the experiments follow the same trend however there is a small variation from execution to execution. This means that the warps of a thread block access the main memory in the same way in all the experiments but some other factors influence the execution time. The other factors can be the number of pending request, contention of the SMs interconnect, refresh rate of the global memory etc. From these experiments it is not possible to determine exactly this time variation. A detailed model of the memory is required in order to define precisely this variation.

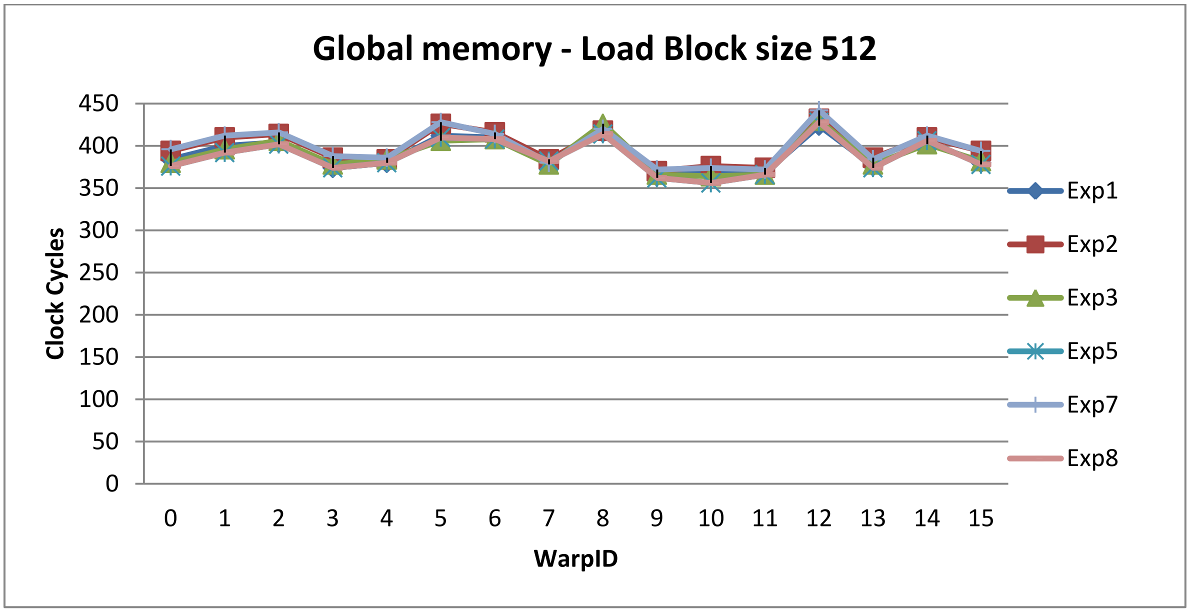


Figure 24 One load to the global memory – 8 experiments.

Figure 25 shows the execution time of different experiments for a store instruction to the global memory for thread block size 512 (16 warps). On the horizontal axis is the warp ID of the warps and on the vertical axis is the execution time in clock cycles. First it can be seen that for one execution (for instance the blue line) warps with warp ID 4 and 14 have different execution times compared to the other warps. The biggest difference is 100 cycles. We have checked for 32, 64, 128, 256 and 512 thread block sizes and this behavior was appeared only for the 512 which shows that the number of warps influence the timing behavior.

Due to the big difference in execution time and the fact that this problem is present for only for great number of warps we assume that there is a write buffer. The data are stored to the write buffer, the execution continues with the next instruction and the store to the global memory is performed in the background. To our knowledge for Fermi there is no documentation about a write buffer.

Secondly, by executing the same microbenchmark multiple times there is some small variation in execution times for most of the warps. In all of the experiments again some warps have significant higher execution time than other warps. It is not always the same warps that experience the different execution times. If we assume that there is a write buffer as we mentioned before, this means that the allocation of the buffer based on the arrival time.

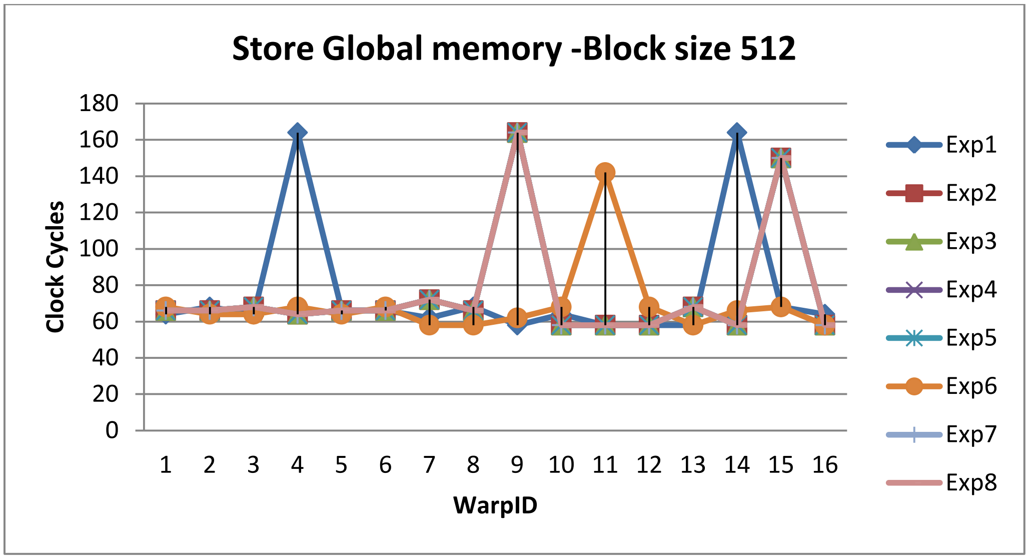


Figure 25 Multiple experiments for one store to the global memory instruction

5.5. Conclusion

Experiments implemented in “asfermi” were implemented to quantify the time variation for the warp scheduler, the shared memory and the global memory. From the experiments we measured the cycles per instruction for the arithmetic instruction with RaW dependency and without. In addition we measured the CPI for the shared memory and the global memory. The results are presented in Table 7.

Table 7 CPI of the different type of instructions.

Type	CPI
Reg	6
Reg RaW	18
LDS	26
STS	30
LD.E	350 - 500
ST.E	34

Furthermore, time variation of the warp scheduler for the microbenchmark with “MAD” and “ADD” instructions that have RaW dependency with thread block size of 512 threads was analyzed. We show that based on the analysis that we have done in GPGPU-Sim the source of this variation is related with a non constant stall of the warps that change the scheduling order. This stall is caused by the unavailability of the execution units or the operand collector.

Moreover, we show that the scratchpad memory has time predictable behavior when the bank conflicts can be determined. In contrast, the execution time to access the global memory varies for store and load instructions. For load instruction the execution time varies among different executions. From the experiments of store instruction we conclude that there is a write buffer. When the write buffer is full the execution time of the store instruction increases and varies from warp to warp.

6. Timing model

In this section the model for the scheduling of the instructions is presented. The timing model takes as input assembly instruction. Based on how the instructions are scheduled and the measured latency of the instructions provides an estimation for the execution time. Initially the timing model is presented and some examples for two simple microbenchmarks are given to explain the model in more detail. Next, the formulas that describe the model are given. Finally, the scalability of the model in terms of number of warps is discussed.

6.1. Instruction scheduling timing model for a single warp

From the observations in Section 5 we define two types of dependencies between instructions. First, there is an instruction dependency. An instruction from the same warp can be scheduled after 6 cycles. Secondly, there is a data dependency. For one warp, an instruction with a data dependency can be scheduled 6+12 = 18 cycles later.

The execution is in pipeline fashion so the warps can be scheduled every 2 cycles. So we assume that the pipeline stages are separated in two parts. We are going to follow the terminology of the GPGPU-Sim microarchitecture [3] (Figure 10).

First there is the “SIMT front end” (green part) in Figure 26 where the I-cache read, decoding of instructions and the operations on the scoreboard are performed. Next there is the SIMD Datapath (yellow part) where the operands are read and the actual execution of the instruction is performed. The next instruction from the same warp can be scheduled after 6 cycles, as the SIMT front 6 cycles latency (Figure 26).

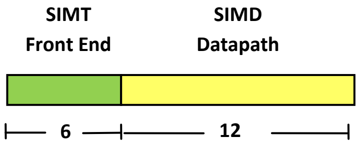


Figure 26 Clock cycles of the SIMT front end and the SIMD datapath

If there is an instruction dependency, the next instruction has to wait until the previous instruction finish the SIMT front end before it start (Figure 27 upper part). We assume that there is no mechanism, like bypassing, that can let the instruction to start before the finalization of the previous instruction. Similarly for the data dependency, the next instruction has to wait until the previous instruction has finished (Figure 27 lower part).

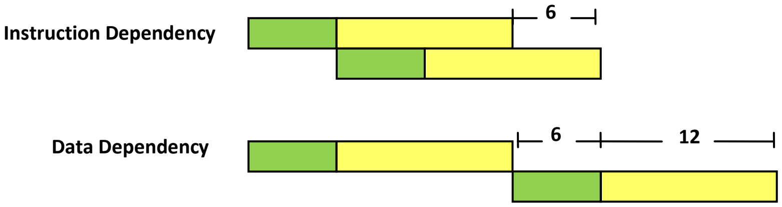


Figure 27 Instruction and data dependencies for one warp

When we have multiple warps the warps are scheduled by the two schedulers (odd and even). An example of the execution of multiple warps for the even scheduler is presented in Figure 28. Since they are instructions from different warps, there is no dependency among each other so they can be scheduled every two cycles according to [1].

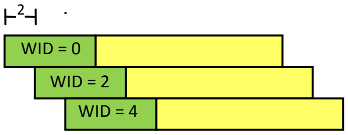
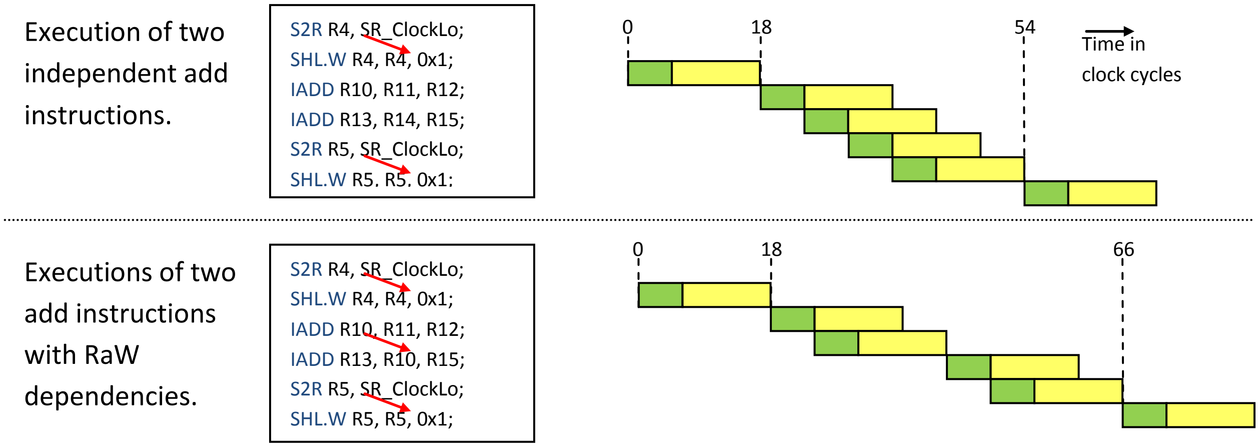


Figure 28 Execution for multiple warps of even scheduler. WID denotes the warp ID of every warp



Execution of two independent add instructions.

```
S2R R4, SR_ClockLo;
SHL.W R4, R4, 0x1;
IADD R10, R11, R12;
IADD R13, R14, R15;
S2R R5, SR_ClockLo;
SHL.W R5, R5, 0x1;
```

Executions of two add instructions with RaW dependencies.

```
S2R R4, SR_ClockLo;
SHL.W R4, R4, 0x1;
IADD R10, R11, R12;
IADD R13, R10, R15;
S2R R5, SR_ClockLo;
SHL.W R5, R5, 0x1;
```

Figure 29 The code and the timeline for the two micro benchmarks. On the upper part the add instruction are independent. On the lower part the instructions have a RaW dependency.

At the upper part of Figure 29 the timeline for the microbenchmark with the independent instructions is presented. Initially the "S2R" instruction is executed. Since the "SHL" has a RaW dependency with "S2R" it can start at the 18th cycle. The "ADD" instruction has an instruction dependency. The next instruction can start at cycle 24. Similarly, the second "ADD" and the second "S2R" instruction can start at cycle 30 and 36. Again, the "SHL" has a RaW dependency so it has to wait until 54th cycle before it can start.

To compare the results from the experiments in Figure 16 and the timeline in Figure 29, it is required to subtract the values of clocks. We have to wait until the pipeline stage, where the value of the clock is ready. We assume that the correct value of the clock is ready when the "S2R" instruction is finished. So we can read the special register which has the value of the clock, 2 cycles after the completion of "S2R". This assumption is not strict, since we could assume that the value of the clock is ready before the completion of the instruction, but the result would be the same. It only matters to follow the same assumption for all the cases. Consequently, for the example with the execution of two "ADD" instructions without dependency is 54 – 18 = 36 cycles, which fits with the results from the experiments.

Similarly we have the execution for the case with the RaW dependency (Figure 29 lower part). Initially, the "S2R" is executed and the "SHL" has to wait until the 18th cycle because of the RaW dependency. The "ADD" instruction can start 6 cycles later. The second "ADD" has a RaW dependency so it can start when the first "ADD" is finished at cycle 42. "S2R" has no dependency with the "ADD" so it can start 6 cycles later. The "S2R" and the "SHL" have RaW dependency so they executed in the same way as before. Finally, for the example with the execution of the "ADD" instructions with RaW dependency is 66 – 18 = 48 cycles, which is the same with the results from the experiments in Figure 16.

6.2. Formulas of the timing model.

In this section the formulas that are used for the calculation of the execution time are described. We use the well known formula for the execution time $CC = CPI * N$. We assume that during the execution of the kernel the clock frequency remains constant. T is the execution time in clock cycles (CC); CPI is cycles per instruction and N the number of instruction. For this model we have used three types of instructions. Arithmetic, shared memory and global memory instructions, so the formula expands to:

$$CC = CC_{arithm} + CC_{shmem} + CC_{glomem} \text{ (1)}$$

The execution time of arithmetic instructions is described in equations 2 to 4. Since the execution time for arithmetic instructions with and without RaW dependency is different we have to distinguish them. The total execution time for the arithmetic instructions is the sum of the execution of independent and dependent instructions (equation 2).

$$CC_{arithm} = CC_{arithm_{ind}} + CC_{arithm_{dep}} \quad (2)$$
$$CC_{arithm_{ind}} = CPI_{arithm_{ind}} * N_{arithm_{ind}} \text{ (3)}$$
$$CC_{airthm_{dep}} = CPI_{arithm_{dep}} * N_{arithm_{dep}} \text{ (4)}$$

The execution time of the shared memory instructions is the sum of the execution time of loads and stores to the shared memory (equation 5). Similarly with the arithmetic instructions, the execution time of loads and stores to the shared memory are given by equations 6 and 7.

$$CC_{shmem} = CC_{shmem_{st}} + CC_{shmem_{ld}} \quad (5)$$
$$CC_{shmem_{st}} = CPI_{shmem_{st}} * N_{shmem_{st}} \quad (6)$$
$$CC_{shmem_{ld}} = CPI_{shmem_{ld}} * N_{shmem_{ld}} \quad (7)$$

For the global memory we describe the memory latency hiding. This behavior is described by equation 8. It is the summation of all instruction between the load request and the use of the data. This is the amount of time where next instructions can be executed in order to hide the memory latency. If the next instruction from load is the instruction that uses the data, (RaW dependency with the next instruction) no memory hiding can be achieved. Please note that, we refer to a different memory hiding that is achieved when different warps are executed. In this case, the warp continues the execution until the data are read.

$$L_{mem_{hide}} = \sum_{i=PC+1}^{PC=use\ of\ data} L_i \text{ , where } L_i \text{ is the latency in clock cycles of the instructions} \quad (8)$$

Similarly with the calculation for the previous types of instructions, the execution time of the global memory instructions is calculated. As mentioned before, the store to the global memory is performed to a write buffer. So the execution time for store is much smaller compared to the load to the global memory. Furthermore, the equation for load to the global memory is extended by subtracting the memory hiding.

$$CC_{glomem} = CC_{glomem_{st}} + CC_{glomem_{ld}} \quad (9)$$
$$CC_{glomem_{st}} = CPI_{glomem_{st}} * N_{glomem_{st}} \quad (10)$$
$$CC_{glomem_{ld}} = CPI_{glomem_{ld}} * N_{glomem_{ld}} - L_{mem_{hide}} \quad (11)$$

6.3. Scalability of the model

Until now the model that it is described works for one warp. The scalability of the model is required to be analyzed in more detail. Figure 30 shows that the execution time remains the same until 6 warps. This behavior derives from the restriction of the instruction dependency. For 6 warps, the warps are scheduled one after the other and they fit to the 6 cycles of the SIMT front end. So the scheduler until this point has only one choice.

For multiple warps the scheduler can choose which warp to schedule. This choice is determined by the scheduling algorithm. For instance in Figure 31, for one warp (upper part) the scheduling algorithm does not influence the execution time. On the contrary, for the case with 7 warps, the scheduler can choose between warp 0 and warp 6. Since the scheduling algorithm is round robin, then the warp 6 will be scheduled.

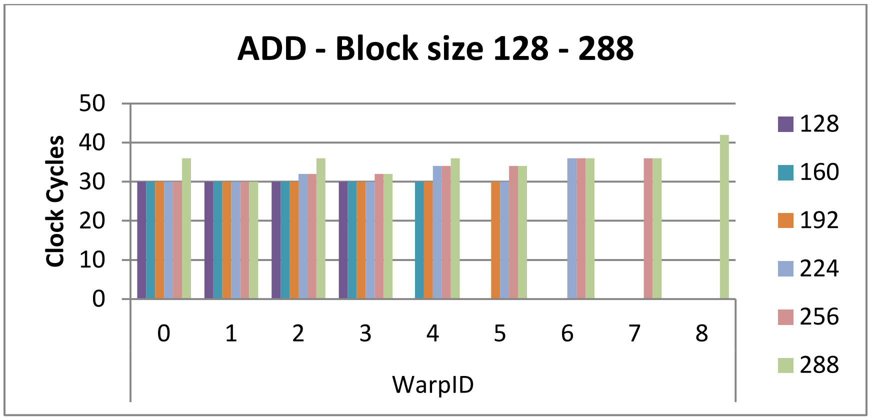


Figure 30 Execution time of one add instruction for different number of warps. The execution time remains the same until the size of the 6 warps

To test the scalability in terms of number of warps of the model, the model for multiple warps is tested. First 3 warps are used for the two microbenchmarks that we used before. The model is compared with the experiments.The model is tested for the case with 3 warps per scheduler. First, the microbenchmark with no dependency is used Figure 33. The code in assembly and the trace of the code is shown. The timeline of the execution is presented in similar way that it was presented before.

First warp 0, 2 and 4 are scheduled. Since there is a RaW dependency with the "SHL" instruction all the warps have to wait until the completion of their previous instruction. The pipeline is full until the last "SHL" where we have again RaW dependency. At this instruction all the warps have to wait until the end of the previous instructions. The execution times of all the warps are the same. However, since warp 2 and 4 start 2 and 4 clock cycles later respectively they finish also 2 and 4 clock cycles later. The timeline exactly fits with the measurement that we get from the experiments. In a similar way the timeline is constructed for the case with the RaW dependency. Again, the timeline fits with the experiments.

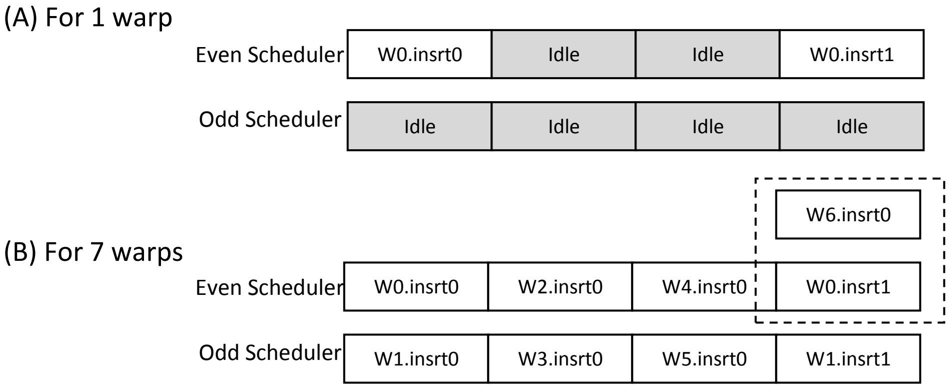
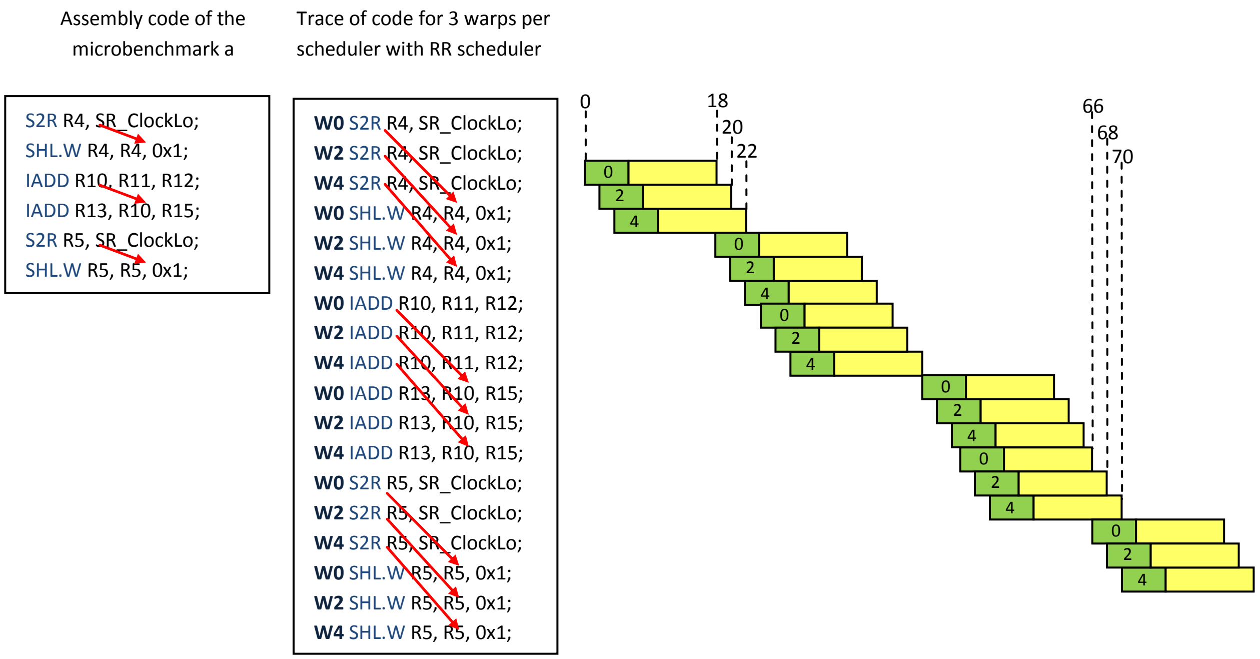


Figure 31 Execution for 1 and 7 warp. For 7 warps the scheduler can choose which warp to schedule.



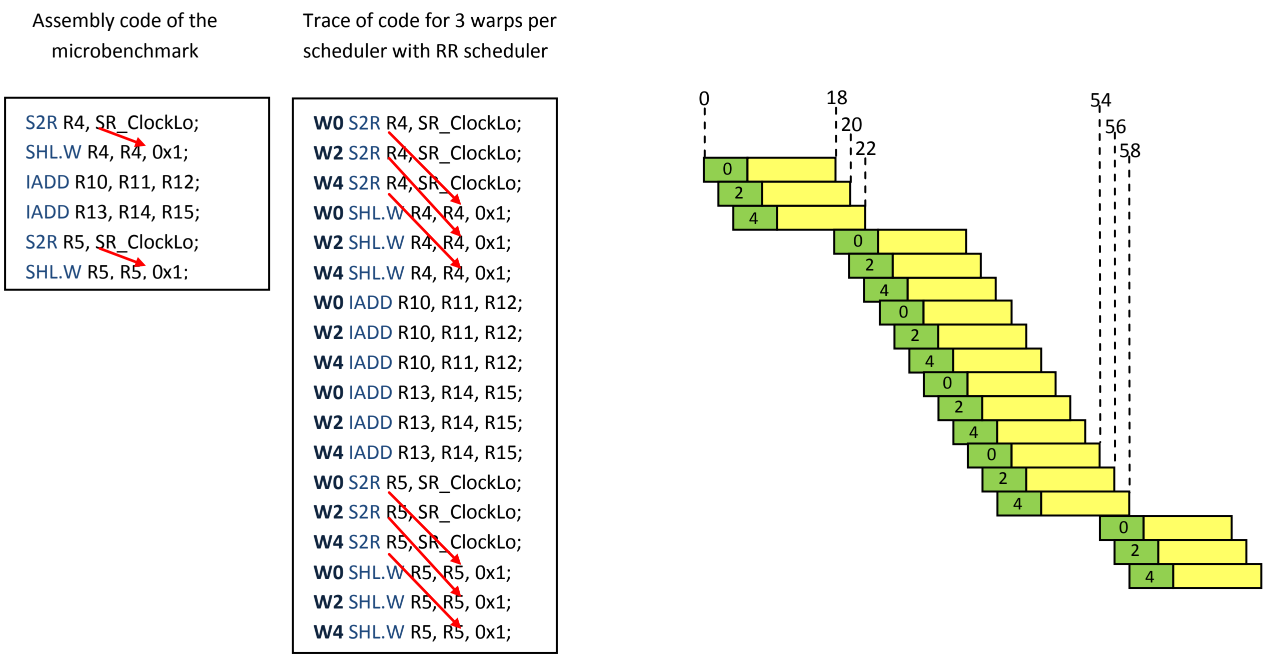
Assembly code of the microbenchmark a

```
S2R R4, SR_ClockLo;
SHL.W R4, R4, 0x1;
IADD R10, R11, R12;
IADD R13, R10, R15;
S2R R5, SR_ClockLo;
SHL.W R5, R5, 0x1;
```

Trace of code for 3 warps per scheduler with RR scheduler

```
W0 S2R R4, SR_ClockLo;
W2 S2R R4, SR_ClockLo;
W4 S2R R4, SR_ClockLo;
W0 SHL.W R4, R4, 0x1;
W2 SHL.W R4, R4, 0x1;
W4 SHL.W R4, R4, 0x1;
W0 IADD R10, R11, R12;
W2 IADD R10, R11, R12;
W4 IADD R10, R11, R12;
W0 IADD R13, R10, R15;
W2 IADD R13, R10, R15;
W4 IADD R13, R10, R15;
W0 S2R R5, SR_ClockLo;
W2 S2R R5, SR_ClockLo;
W4 S2R R5, SR_ClockLo;
W0 SHL.W R5, R5, 0x1;
W2 SHL.W R5, R5, 0x1;
W4 SHL.W R5, R5, 0x1;
```

Figure 32 Execution of 3 warps, instructions with RaW dependency



Assembly code of the microbenchmark

```
S2R R4, SR_ClockLo;
SHL.W R4, R4, 0x1;
IADD R10, R11, R12;
IADD R13, R14, R15;
S2R R5, SR_ClockLo;
SHL.W R5, R5, 0x1;
```

Trace of code for 3 warps per scheduler with RR scheduler

```
W0 S2R R4, SR_ClockLo;
W2 S2R R4, SR_ClockLo;
W4 S2R R4, SR_ClockLo;
W0 SHL.W R4, R4, 0x1;
W2 SHL.W R4, R4, 0x1;
W4 SHL.W R4, R4, 0x1;
W0 IADD R10, R11, R12;
W2 IADD R10, R11, R12;
W4 IADD R10, R11, R12;
W0 IADD R13, R14, R15;
W2 IADD R13, R14, R15;
W4 IADD R13, R14, R15;
W0 S2R R5, SR_ClockLo;
W2 S2R R5, SR_ClockLo;
W4 S2R R5, SR_ClockLo;
W0 SHL.W R5, R5, 0x1;
W2 SHL.W R5, R5, 0x1;
W4 SHL.W R5, R5, 0x1;
```

Figure 33 Execution of 3 warps, independent add instructions

6.4. Conclusion

A timing model for the warp scheduler was presented. It is based on the observation of the experiments that we performed at Section 5. Initially the model for a single warp was proposed. The instruction dependency was introduced where the instructions from the same warps can be scheduled every 6 cycles. The formulas for the execution time were given based on the well known equation for the execution time in clock cycles $CC = CPI * N$.

Finally, the model was tested for the scalability in terms of number of warps and was shown that the model can be extended for 6 warps (3 warps per scheduler) without any modification. For higher number of warps further study of the warp scheduler is required.

7. Benchmark: Convolution Separable

We use the measurements for the CPI Section 5 and the formalization of Section 6 to test the model for the convolution separable benchmark from the CUDA SDK. Afterwards, the model is compared with the model proposed by Hong and Kim [28].

7.1. Application of the model

The convolution separable application from the CUDA SDK was used to test the accuracy of the model. The 2D convolution application is split in two 1D convolutions, one for the rows and one for the columns. The two parts of the algorithm are implemented in two different kernels that are executed one after the other. The kernels are using 32 threads, so only one warp is running on the GPU. We use only one warp because from the experiments we have seen that the execution with one warp has time predictable behavior. Based on the analysis that was performed at Section 6.3 the model can be tested also for 6 warps.The distributions of the different types of instruction that are used from the two kernels are presented in Figure 34. Both kernels efficiently use the shared memory so most of the instructions are arithmetic instruction and shared memory instructions. For the rows kernel, only 1% of the instructions have RaW dependencies while for columns kernel this is significantly higher, 22%.

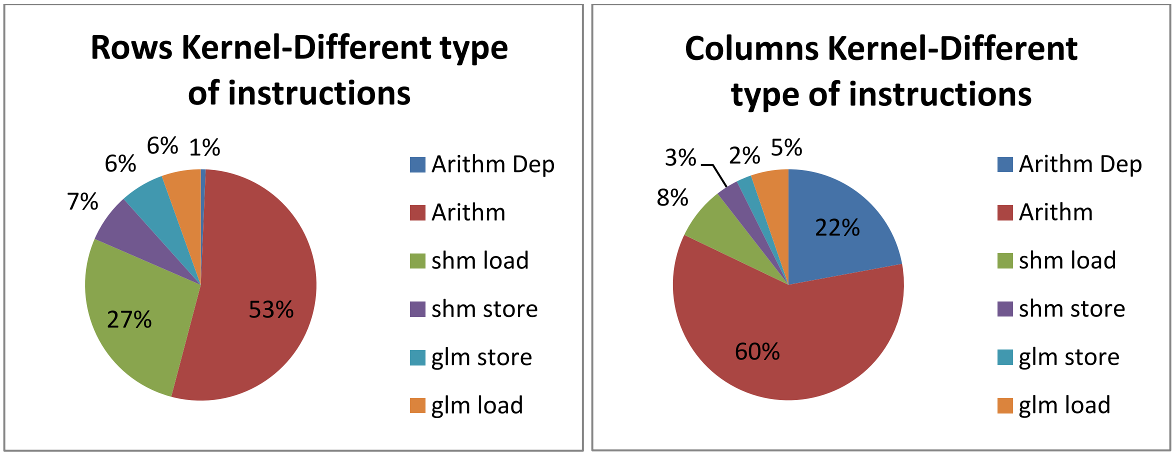


Figure 34 Percentage of Instruction types for the rows and Columns kernel

Table 8 Instruction types for the rows kernel

Arithm Dep	Arithm	shm load	shm store	glm store	glm load	Total
1	78	40	9	9	8	145

Table 9 Instruction types for the columns kernel

Arithm Dep	Arithm	shm load	shm store	glm store	glm load	Total
21	57	7	3	2	5	95

For the comparison of the experiments and the model we use the best, average and worst case execution time (BC, AC and WC). In order to determine the BC, AC and WC, WCET analysis tools are needed. To our knowledge this type of tools are not available for GPUs and specifically for the Fermi architecture. So, the BC, AC and WC are calculated by executing the same experiments multiple times.

For the BC of the model, the load to the global memory takes the minimum amount of time (64 cycles which is the same as the store to the write buffer). For the worst case of the model the load takes the maximum value because we assume that no memory hiding can be applied: $L_{mem_{hide}} = 0$.

Table 10 and Table 11 show the error of the model and the experiments of the rows and the columns kernel. The best, average and worst case execution time for the experiments and the model are compared. First for rows, it can be seen that the average case has an error of 2% although for best case is underestimated by -15% and for worst case 38% is overestimated. The results show that, the average case can be predicted precisely although the best case and the average case are too optimistic and pessimistic respectively. The reason is that we do not perform any detail analysis for the global memory accesses. We assume for the best case that all the memory accesses can hide the memory latency. Similarly for the worst case we assume that none of the memory accesses can hide the memory latency.

Table 10 Error of the model and the experiments for the rows kernel

	BC	AC	WC
Experiment	3365	3410	4004
Model	2856	3510	5544
Error	-15%	2%	38%

Table 11 Error of the model and the experiments for the columns kernel

	BC	AC	WC
Experiment	3106	3127	3426
Model	1428	3228	3458
Error	-54%	3%	1%

The error for the columns kernel is similar for the average case but the best and the worst case are significantly different. The best case is considerably underestimated while the worst case is predicted more precisely. This behavior can be explained by the fact that the best, average and worst case are close together. In this case the compiler does not find enough instructions to put between the load to the global memory instruction and the use of the data. So the memory latency cannot be hidden efficiently.

The range of the prediction is graphically presented in Figure 35 to understand the results intuitively. From this figure it can be seen that the prediction range for both cases is approximately the same. In addition, it can be noted that the different distribution of the execution times for the two kernels. Furthermore, for the columns kernel due to the limited distribution the BC, AC and WC are more close together compared to the rows kernel.

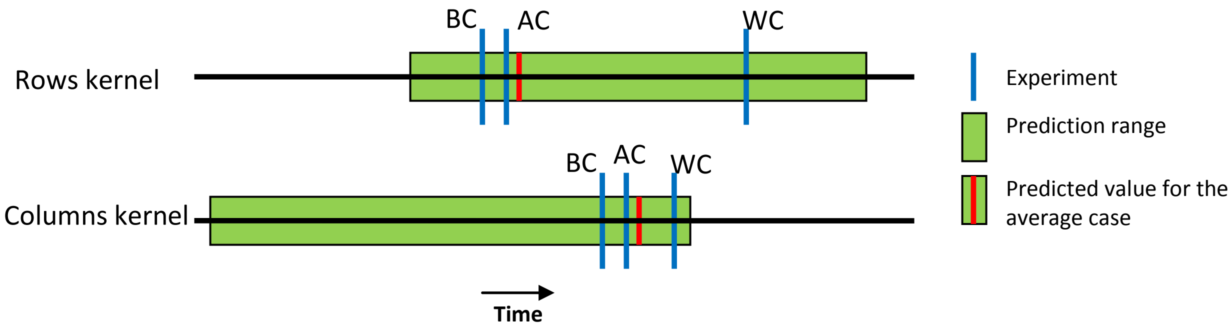


Figure 35 It is presented the best, average and worst case of the experiments and prediction range of the model. The prediction of the average case is shown with a presented line.

7.2. Comparison

Furthermore, the model proposed in [28] is compared with this model. For simplicity we are going to call it "MWP model" (memory warp parallelization), derived from the new metric that have proposed. In this paper the Tesla architecture is modeled. Although, the model is based on warp level parallelism for the memory accesses and the computation, it does not make significant assumption about the architecture. The model was implemented in Python and the parameters were tuned to match the Fermi architecture. The parameters that were used for the two kernels are presented in Table 12. The results are presented in Table 13 for the Rows and Table 14 for the Columns kernel. Initially, it can be seen that for the case of the rows kernel the results are very similar. For columns kernel the difference between the MWP model and the execution time is significant (18%). As we mentioned before this kernel has many RaW dependencies (Figure 34). The MWP model does not take into account the extra cost of the dependencies which has as a result the underestimation of the computation part of the kernel.

Table 12 Values for the parameters of the MWP model

Parameters	Rows Kernel	Columns Kernel
#Threads_per_warp	32	32
#Issue_cycles	2	2
Clock Frequency (GHz)	607	607
Mem_Bandwidth	133.9 GB/s	133.9 GB/s
Mem_LD	400	470
Departure_del_uncoal	10	10
Departure_del_coal	4	4
#Threads_per_block	32	32
#Blocks	1	1
#Active_SMs	1	1
#Active_blocks_per_SM	1	1
#Active_warps_per_SM	1	1
#Total_insts	146	95
#Comp_insts	138	90
#Mem_insts	8	5
#Uncoal_Mem_insts	0	0
#Coal_Mem_insts	8	5
#Synch_insts	1	1
#Coal_per_mw	1	1
#Uncoal_per_mw	32	32
Load_bytes_per_warp (Bytes)	4*32	4*32

Table 13 Rows comparisons of the models

Rows Kernel	AC
Experiments	3410
Model	3510
MWP model	3492
Model Error	2%
MWP error	2%

Table 14 Columns Comparison of the models

Columns kernel	AC
Experiments	3127
Model	3228
MWP Model	2540
Model Error	3%
MWP error	-18%

7.3. Effect of the memory latency on the models

Figure 36 and Figure 37 show how the latency of load to the global memory influences the two models. For both cases with linear increase of the memory latency the execution time increases linearly. This is expected since in the formulas the global memory latency is part of the linear combination for the calculation of the execution time. Note that, this is the average value of the global memory latency since it can change from access to access. In addition, MWP model seems to be more sensitive to changes of the latency since it has higher slop for both kernels.

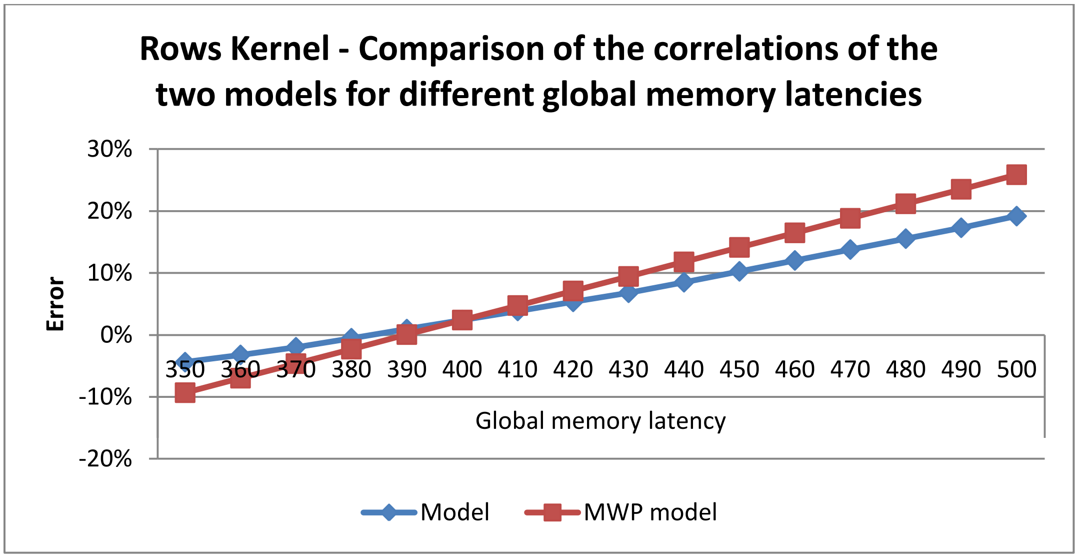


Figure 36 Rows kernel – Error of the two models for different values of the global memory latency.

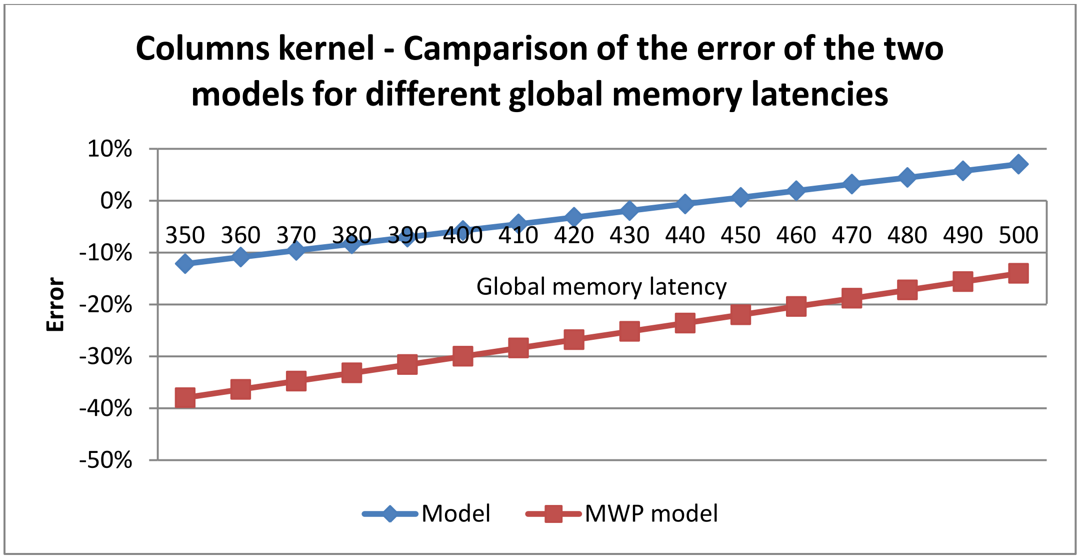


Figure 37 Columns kernel - Error of the two models for different values of the global memory latency.

7.4. Analysis of the benchmark for time variation

Until now, the average case of the execution time was discussed. Now we analyze in more detail the distribution of the different execution times. Figure 38 shows the distribution of 200 experiments for the execution times for the Rows kernel. The horizontal axis show the different execution times from experiment to experiment and on the vertical axis is shown the number of experiments. The bars show the number of experiments that have the same execution time and the experiment accumulation is presented with the red line. The dash lines present the average case the execution time calculated with our model. We notice that the calculation from the model can give an upper bound for 96.5% of the experiments with 2% error from the AC. The rest 3.5% of experiments is the one that determines the WCET so the kernel has to be analyzed in more detail.

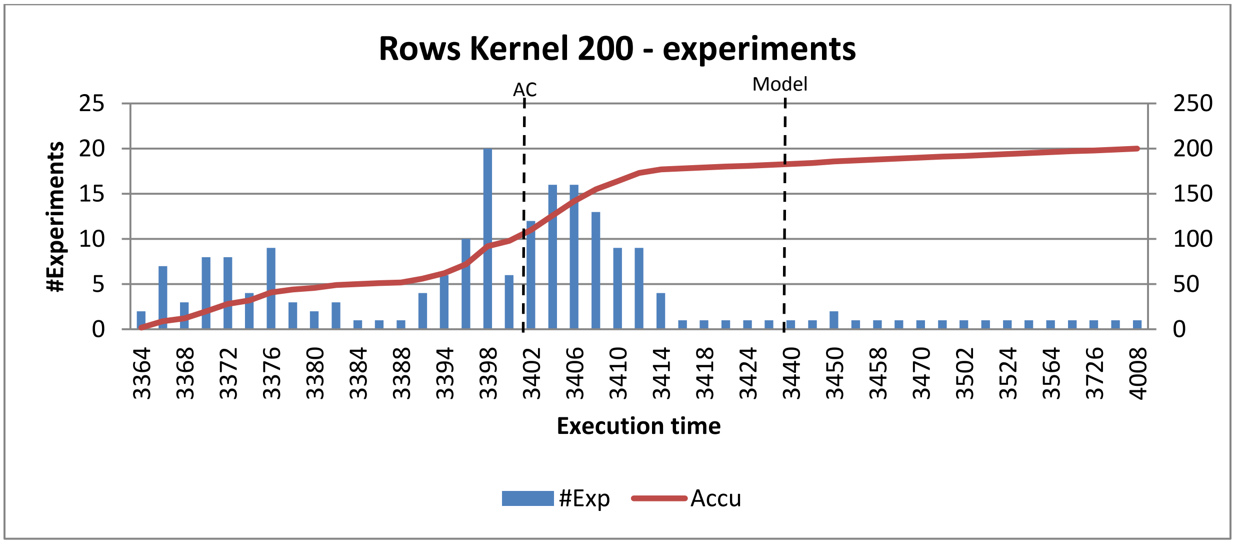


Figure 38 Distribution of execution time part of Rows kernel.

We split the kernel in three parts: (1) load from the global memory, (2) computation and (3) store to the global memory. show the distribution of the instructions of the three parts and in Table 15, Table 16 and Table 17 the instructions in absolute values can be found.

If we sum all the instructions from the different parts (Table 15, Table 16 and Table 17) it will not be the same with total number instruction of the kernel (Table 8) due to overhead of the instructions. More precisely there are 4 (2 for starting time and 2 for finishing time) extra instructions of Load part since the instruction are placed all together from the compiler. In addition 4 instructions multiply by 8 measurements for the computation part (32). Similarly with the computation part we have 32 instructions. Furthermore, the values for the measurements have to be store to the memory, so we have to calculate also the store instructions. However, the compiler reorders the instructions and it can be inside or outside the measured part.

The load part loads the data that initial are in the global memory to the shared memory to minimize the memory access cost of the future memory access. There is also a store to the global memory instruction (ST.E) in this part. This instruction stores the starting time of the measurement. The compiler performs instruction reordering to minimize the RaW dependencies, so it was not possible to exclude these instruction from this part. Next, in the computation part the data are read from the shared memory and the actual computation is performed. Finally, in the store part the data are stored to global memory.

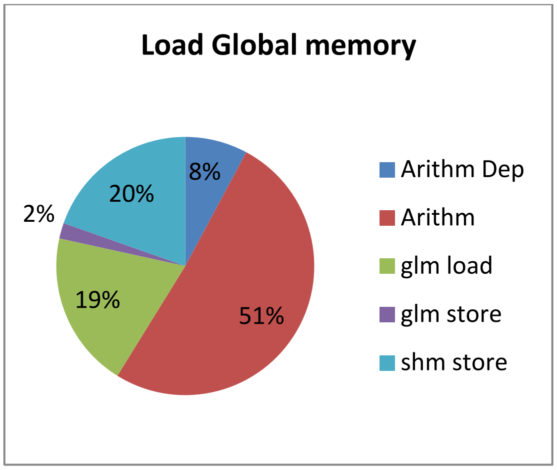


Figure 39 Percentage of instructions for load part

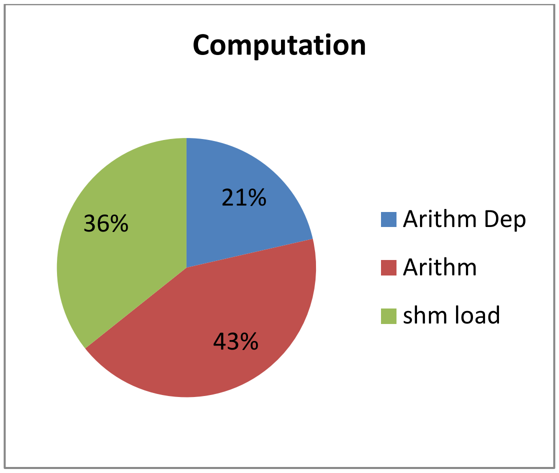


Figure 40 Percentage of instructions for percentage part

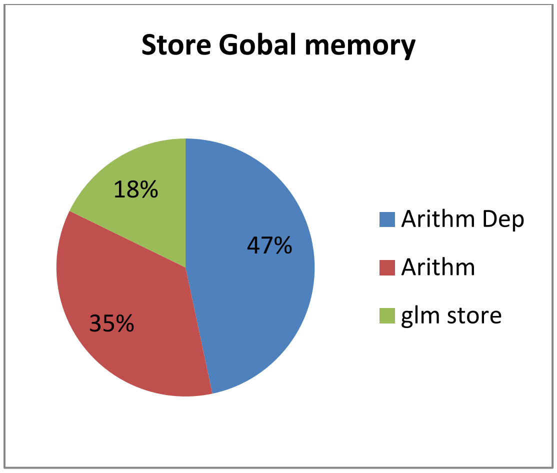


Figure 41 Percentage of instructions for store part

Table 15 Instructions for load part.

Arithm Dep	Arithm	glm load	glm store	shm store	Total
4	26	10	1	10	51

Table 16 Instructions for computation part.

Arithm Dep	Arithm	shm store	Total
24	48	40	112

Table 17 Instructions for store part.

Arithm Dep	Arithm	glm store	Total
21	16	8	45

We execute different parts of the kernel 500 times in order to define the variation of the execution times. The results are presented in Figure 42, Figure 43 and Figure 44 for the load, computation and store part respectively. All parts have variation in the execution times from execution to execution.

First we present the results for the load part. There are 5 main categories of execution times. The biggest difference of the main categories is 10 clock cycles. We have shown in section 5 (Figure 24) that the execution time of load instructions to the global memory has variation from execution to execution, so this difference is expected. Furthermore, we see that the difference of the best case and the worst case is 324 clock cycles.

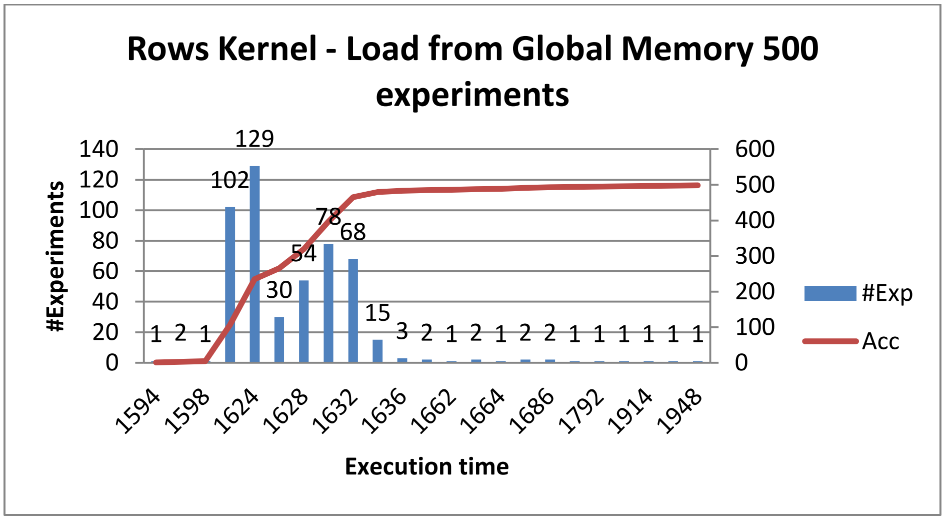


Figure 42 Distribution of execution time for the load part of rows kernel

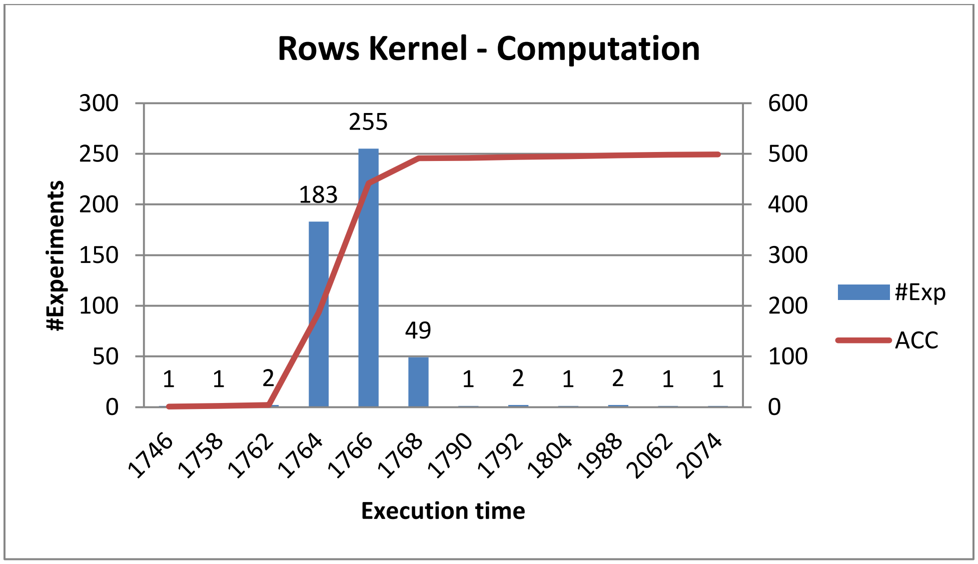


Figure 43 Distribution of execution time for the computation part of rows kernel

The results for the computation part are presented in Figure 43. We see that the execution times are distributed in 3 main categories. The largest difference in these categories is 4 clock cycles. However, we note that the difference between the best case and worst case is 328 clock cycles. We relate this difference with the variation that we found for the execution of mad-add instruction with RaW dependency (Figure 20). In this case we have difference 23% compared to the average execution of the different warps for 6% of the experiments. The main difference with the problem of Figure 20 is that we use 16 warps while here we use only one.

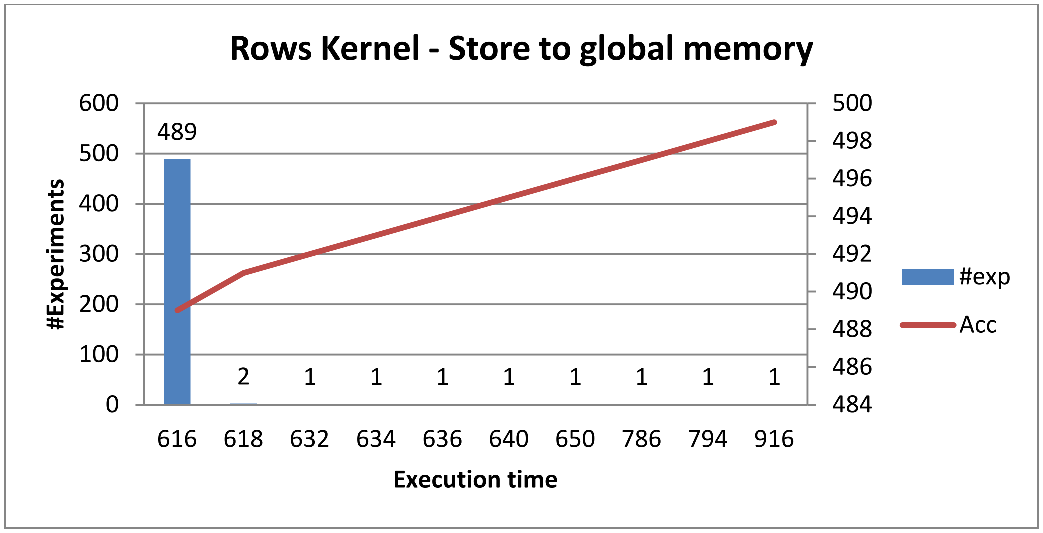


Figure 44 distribution of execution time for the store part of rows kernel

The results for the store part are presented in Figure 44 that 98% of the experiments have the same execution time, 616 clock cycles. We note also that the average case is the same with the best case. This behavior shows that the write buffer is used efficiently. The worst case is 300 clock cycles slower compare to the average case (48%) which is significant big compared to the load and the computation part. This difference is consistent with the experiment that we had in section 5 for the store instruction to the global memory, where again for store instruction we had the biggest difference. For one store instruction the variation was 100 clock cycles. So if 3 more stores out of 8 can use the write buffer efficiently would lead to the 300 clock cycles different that we have for this case.

7.5. Model scalability in terms of warps

The model is analyzed for its scalability in terms of the number of warps it can support. We maintain the same model as in section 6.3. Our hypothesis is that the execution time would remain the same until it reaches a thread block size of 6 warps and afterwards execution time will increase. The time behavior for a thread block size larger than 6 warps requires further research.

Figure 45 presents the results of the experiments for a different number of warps and the model for the execution time. The horizontal axis shows the number of warps used by the kernel. The vertical axis shows the execution time in clock cycles. Initially, for a thread block size of up to 2 warps the execution time remains almost the same (1% increase for 2 warps) for the BC, AC and WC. Next, for 2 to 4 warps the execution time decreases by 5% for the BC and WC. This behavior was unexpected, however the execution time of WC remains almost the same (1.5% increase). It is assumed that the difference between the WC and BC and AC is due to the variation in global memory access. A more detailed analysis is required to determine this problem. Finally, for more than 6 warps the execution time increases as expected.

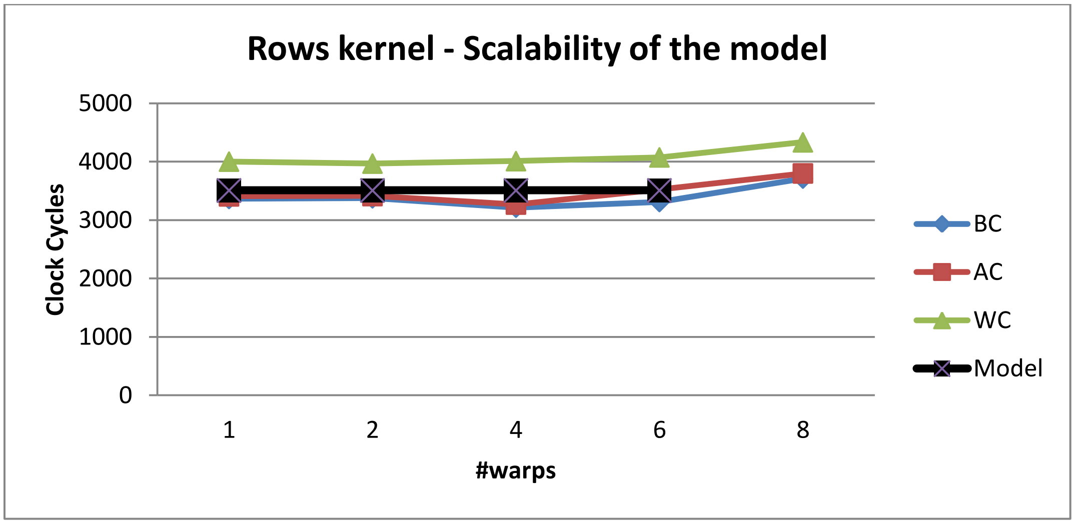


Figure 45 Scalability of the model in terms of number of warps for Rows kernel.

7.6. Conclusion

The timing model was tested with the convolution separable from CUDA SDK. The experiments show that for the configuration with a single warp, the execution can be predicted with 2% and 3% error for rows and kernels for the average case. For the BC and WC the error is -15% and 38% for rows kernel and -54% and 1% for the columns kernel. In other words the model can provide an upper bound for 96.5% of the experiments with 2% and 3% error.

In addition, the model was compared with Hong and Kim model [28]. The two models match for the rows kernel but for the Columns kernel the MWP model has -18% error. This difference is caused from the fact that their model does not take into account the extra cost of RaW dependencies. The two models are compared for different global memory latency values. For both models the execution time increased linearly with the linear increase of the memory latency. Finally the scalability of the model was investigated and the model was tested for 1, 2, 4, 6 and 8 warps. The results show that the model is scalable up to 6 warps (192 threads) as was expected. For more than 6 warps further study is required.

8. Variable rate warp scheduling

As described previously, the Fermi warp scheduler uses the Loose Round Robin algorithm. Warps are scheduled, based to their warp ID. A warp with a smaller ID will be scheduled before a warp with a higher warp ID. When there is a load to global memory, the scheduler dynamically will find the next available warp to schedule. The scheduler is fair for all the warps so the warps reach long instruction at close points in time. To improve the time predictability of the system it is required to schedule the warps in a more controlled way. It is easier to analyze its behavior of the GPU if the warps have constant scheduling frequency.

Initially, it is described the motivation to use the variable-rate warp scheduler and the problems that the algorithm solves are presented. Finally, experimental evidence is given to verify the correctness of the algorithm.

8.1. Motivation for the variable rate warp scheduler

Without any restriction to the warp scheduler we cannot always determine which warp is going to be scheduled next. If we set a fixed scheduling rate for all the warps then it is easier to determine the execution time [16].

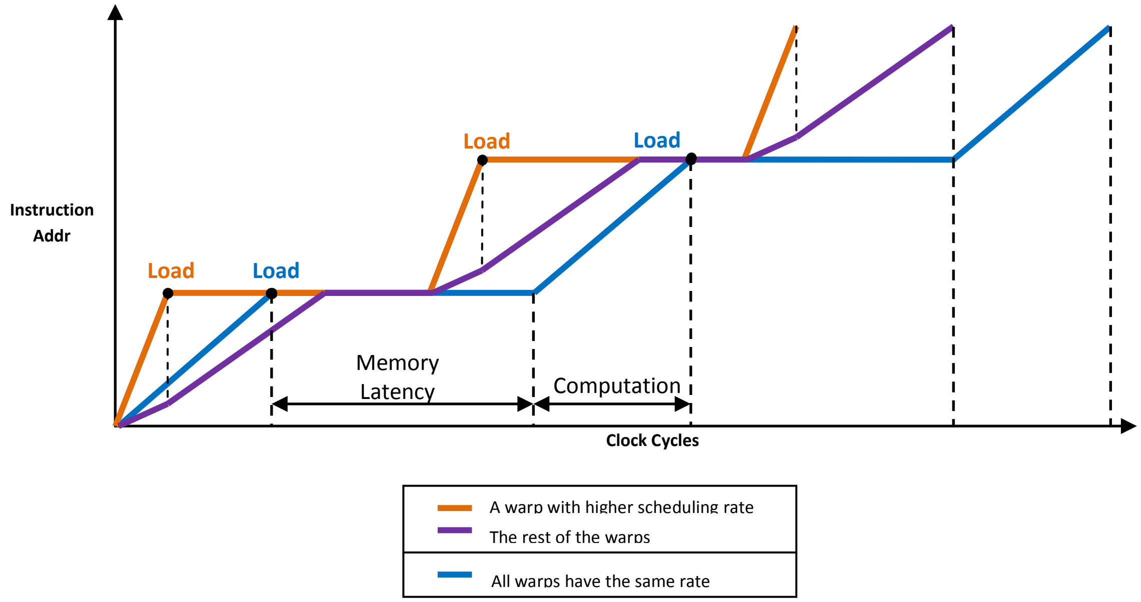


Figure 46 Typical execution in GPU of the warps is presented with the blue line. All the warps are scheduled with the same rate (RR) so the execution times of all warps are presented the blue line. We compare the typical execution with the where the same number of warps is used but one warp is scheduler faster (orange line). The rest of the warps are scheduled in RR fashion.

A typical execution of a program on a GPU is presented with a blue line in Figure 46. On the horizontal axis the time in clock cycles is presented and on the vertical axis the instruction address. The figure above shows how the program advances during its execution. If all the warps are scheduled with the same rate, then the scheduling algorithm is RR and all the warps have same execution time. So in this diagram with a round robin algorithm all the warps would be on top of each other so it reduces to one line.

From the diagram it can be seen that there are two types of execution. First we have the memory part. The program has to wait for the memory to get the data; since the program is not advancing the line is horizontal. Afterwards, it starts the computation part. For simplicity, it is assumed that the instruction address increases linearly with time, which means that all the instructions have the same execution time.

Since with this algorithm we restrict the options of the scheduler we expect to have degradation in performance. By using the variable rate algorithm the performance of the system can be improved by setting one warp to be scheduled more frequently compared to the other warps. In this way the long instruction and for this case the load from the global memory is reached in different points in time. The warp that it is scheduled more frequently reaches the memory instruction earlier. Since warps with close warp IDs have spatial locality, the fast warp will bring also the data for the other warps by taking advantage of the spatial locality of the data. In this way the latency can be hidden more efficiently. Graphically, this behavior is presented in Figure 46 . We have same number of warps as we have in the case with the blue line but the orange line is one warp that it is scheduled more frequently and the purple line is the rest of warps that are scheduled with the same frequency.

The warp that is scheduled more frequently it is executed faster. Thus the computation part has a bigger slop compared to initial execution (blue line). The memory part remains the same since the memory latency does not depend on the scheduling frequency.

The rest of the warps (purple line) are scheduled less frequently compared to the initial case (blue line). The warps are scheduled only in the slots where the fast warp is not scheduled. Consequently, the slope of the purple line is smaller compared to the initial case. However, when the fast warp reaches the load instruction has to wait for the results. So it is not scheduled and the rest of the warps can use all the slots. The slope of the purple line increases and it becomes bigger compared to the blue one since fewer warps have to be scheduled.

With this scheduling scheme the performance of the GPU can be improved since the fast warp can reach the load to global memory instruction earlier, while the other warps are still working on previous instructions.

8.2. Implementation and Results

We have made an implementation in GPGPU-Sim. The implementation has been performed in the same way as the other schedulers in GPGU-Sim are implemented. The scheduler algorithm can be chosen from the configuration file.

Figure 47 shows an example execution for the even scheduler. The behavior of the odd scheduler is identical. For this example we configure the GPGU-Sim with perfect memories in order to make more clear the execution of the different warps. The time is presented on the horizontal axis. The vertical axis shows the instruction address. We execute only arithmetic instructions in this example. W0 is scheduled once rate = 2 (e.g. W0 W2 W0 W4…). The higher rate of scheduling for W0 and W1 has as a result to finish faster the execution of the kernel. The rest of the warps are executed in round robin fashion.

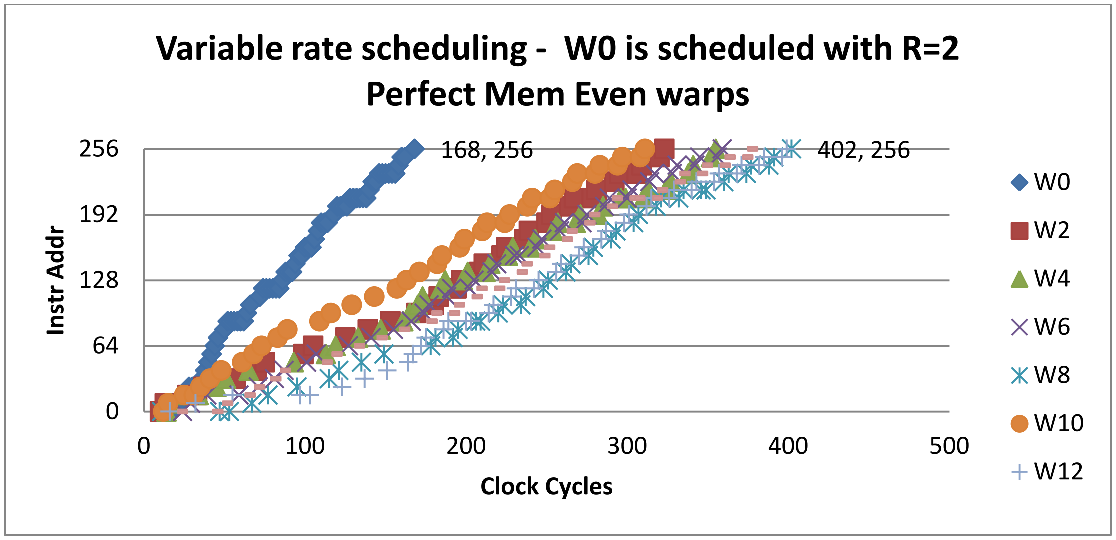


Figure 47 Example execution with perfect memories for variable rate scheduler.

The execution was tested with a microbenchmark that is using only arithmetic instructions and we enable the memories in contrast with the previous example. The results are presented in Figure 48 and Figure 49 for the even and the odd scheduler. From the two graphs we see that there is a memory latency every 16 instruction. Since we do not perform any accesses to the global memory, instruction cache misses introduce the memory latency. In real hardware we did not notice so many cache misses.

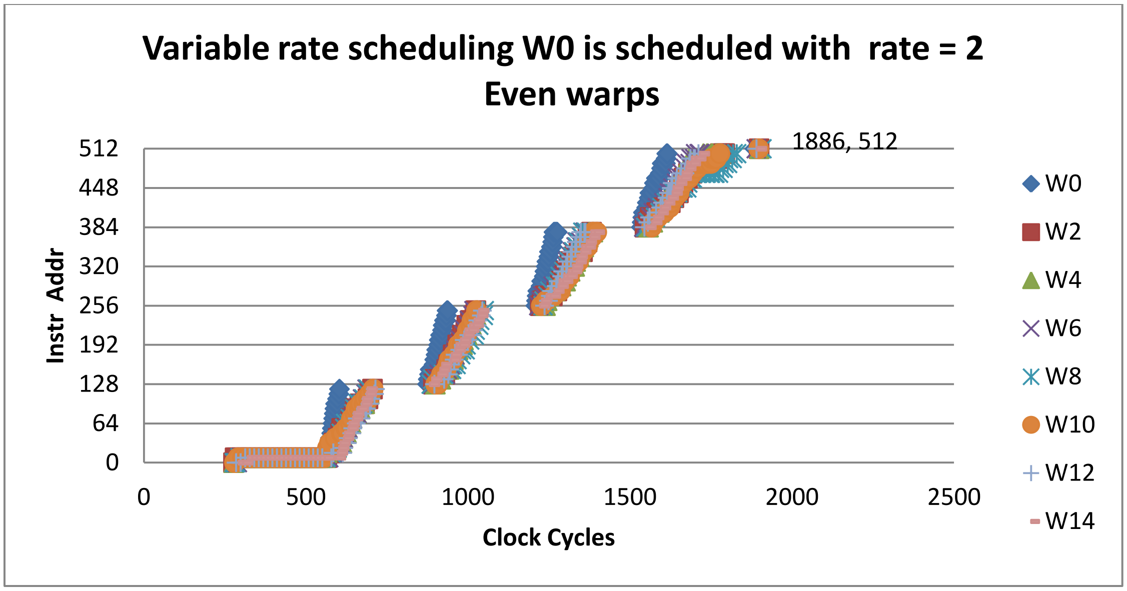


Figure 48 Even scheduler, the warps are executed with variable rate scheduling.

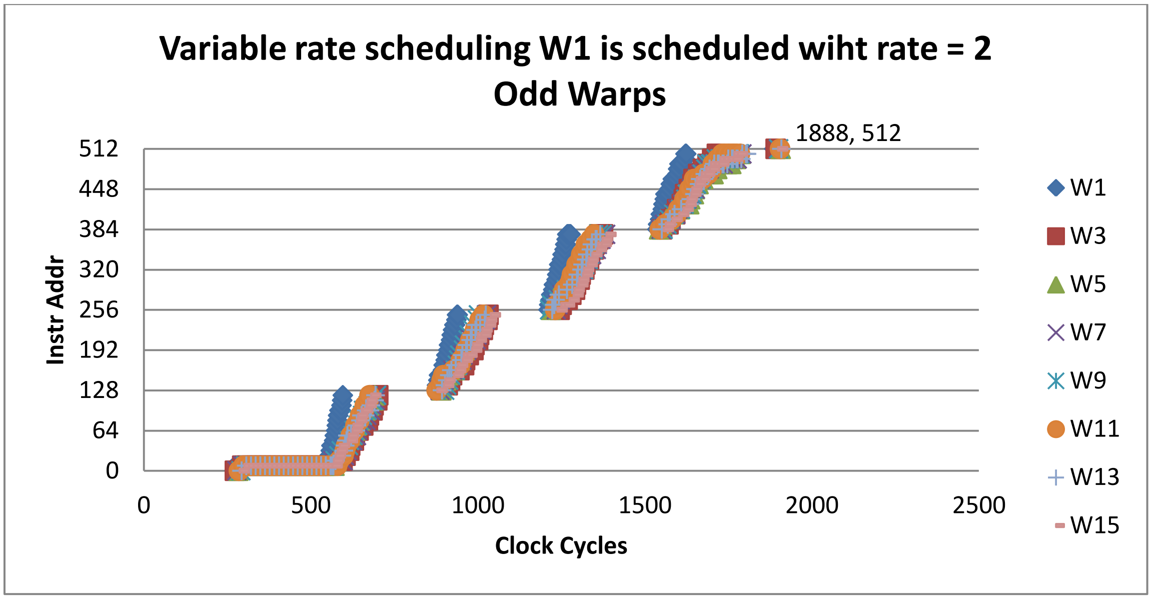


Figure 49 Even scheduler, the warps are executed with variable rate scheduling.

The results are compared with the available schedulers in GPGPU-Sim (Table 18). The round robin (RR scheduler), the greedy then oldest (GTO) scheduler where the warps executed in round robin fashion but when a warp blocks (i.e. global memory access) it continues with the oldest warp, and the “two level” scheduler [21]. The two-level scheduler is a two level hierarchical round robin scheduling.

Table 18 Comparison of the variable rate scheduler to the different schedulers from GPGU-Sim.

Scheduler	IPC
LRR	14.6
Two Level	13.4
GTO	15.7
Variable rate	15.5

8.3. Conclusion

Dynamic scheduling of the warps can influence the time predictability of the GPU. Different scheduling decisions at run time can lead to different execution times. We show that the variable-rate warp scheduler can be used to improve time predictability, by assigning the warps to fixed slots. In addition, initial results show that when a warp is assigned to a higher scheduling rate than the other warps, the performance can be improved due to the inter-warp spatial locality of the data.

9. Conclusions and future work

First a summary of the findings is presented in subsection 9.1. Section 9.2 presents the future work and some options about how to use the timing model and the variable rate scheduler. Finally, section 9.3 concludes the report.

9.1. Summary

Nowadays GPUs gain popularity among the general purpose parallel architectures. Programming models like CUDA and OpenCL are used for GPGPU programming in order to accelerate general purpose applications. The timing behavior of the Fermi GPU architecture was examined through latency analysis. Microbenchmarks implemented in assembly that expose the timing information of the architecture were used. The experiments are compared with the GPGPU-Sim cycle level simulator. The main contributions of this project are:

- We found that the warp scheduler has different time behavior for 4% of the experiments for microbenchmark with one “MAD” and one “ADD” instruction with RaW dependency. We show that possible reasons for this behavior are the unavailability of the execution units or the register file bank conflicts.

- The scratchpad memory was analyzed and we approved that it has time predictable behavior when the number of bank conflicts is known. In addition, the variation of the execution time of load to the global memory was measured.

- An Instruction dependency with cost of 6 cycles between the instructions of the same warp was observed. The cost of the RaW (18 clock cycles) dependency was quantified. Consequently, we found the number of the pipeline stages for arithmetic instructions. Furthermore, the latency of the shared and global memory instructions was measured. For the global memory we take into account the memory hiding for one warp.

- A timing model for the warp scheduler was introduced. The model was tested for Convolution separable benchmark from CUDA SDK. The model was accurate for average case with 2% and 3% error for rows and columns kernel respectively. For the best case the model had -15% and -54% and worst case 28% and 1% error for rows and columns kernel. Finally, we show that the model is strait forward scalable up to 6 warps.

- Finally, the variable warp scheduler algorithm was proposed and implemented in GPGPU-Sim based on the thread scheduler of FlexPret architecture [16]. The algorithm improves the time predictability of the system by scheduling the warps in fixed slots. In addition, initial results show that for the configuration with one warp scheduled with higher rate than the others can improve also performance.

9.2. Future work

There is plenty of space to improve the Fermi GPU architecture in order to improve the time predictable behavior. Next we present some indications for the future work:

- Implementation of a tool that uses the model. It will take as input assembly code and produce estimation for the execution time.

- The size of the write buffer has to be determined. To achieve that we would use microbenchmarks for the store instruction to the global memory. By gradually increasing the number of warps, the size of the buffer could be shown. In this way it would be possible to estimate when the write buffer would be full and as a result to predict more precisely the WCET. Consequently we would have a better estimation for the WCET by replacing the equation 10 with the equation 13.

  $$T_{glomem_{st}} = f(\#warps) * N_{glomem_{st}} \qquad (13)$$

  Furthermore, the model can be modified to include also the time variation of the load instruction to the global memory. More precisely, the equation 11 can be replaced by equation 14. The function “f” represents the variation of the execution times from execution to execution.

  $$T_{glomem_{ld}} = f(variation) * N_{glomem_{ld}} - L_{mem_{hide}} \qquad (14)$$

- The scalability of the model can be extended. First it is necessary to define the behavior of the warp scheduler for more than 6 warps. Based on the behavior of the warp scheduler the model should be adapted accordingly.

- The model can be extended by using the level 1 cache model proposed in [37]. By using this model we could predict more precisely the execution time of the memory accesses. In addition, in order to include the L2 cache memory in the analysis we would design the L2 cache in a different way. It is very complex to analyze it, since it is shared between the SMs and is unified for data and instructions. Some initial design decisions would be:

    - The separation of the instructions and data L2 cache memories. It simplifies the analysis, since the data and instructions can be analyzed independently.
    - Random replacement policy. With random replacement policy we can analyze the behavior of the cache with probabilistic methods.

- The model can be extended in order to take into account the bank conflicts to the shared memory

$$T_{shmem_{ld}} = CPI_{shmem_{ld}} * N_{shmem_{ld}\,NBC} + \sum_{\mathrm{i}}^{\mathrm{N}_{\mathrm{shmem}_{\mathrm{ld}}}\,BC} \mathrm{n}_{\mathrm{i}}\, \mathrm{CPI}_{\mathrm{shmem}_{\mathrm{ld}}} \quad (14)$$

$$\text{where } \mathrm{n}_{\mathrm{i}} \text{ is the number of bank conflicts of instruction i}$$

- Design space exploration can be performed for the different scheduling rates of the warps to find the optimal tradeoff between performance and time predictability. Finally, the variable-rate warp scheduler can be modified in GPGPU-Sim in order to be able to change the scheduling rates of the warps at run time. In this way the warp scheduler can have a more adaptive behavior. Therefore, the scheduler would be able to schedule warps with mixed criticality.

9.3. Conclusion

The Fermi GPU architecture was analyzed to get a better insight of its time behavior. The characteristics of the GPU architecture are advantageous for the time predictability.

The SIMT model provides hierarchy of the threads that are executed in parallel. Thus, the resources of the system are assigned in a hierarchical way and can be analyzed easier. In addition, Fermi uses configurable scratchpad - cache memories. The scratchpad memory is controlled by software and so, it can be used for time predictable memory accesses. Furthermore, the control flow mechanism does not use speculation mechanisms (e.g. branch prediction) which would make the analysis much more complicated. Due to the instruction dependency, the warp scheduler has time predictable behavior for up to 6 warps. The main source of variation was caused by the out-of-chip global memory. There are known solutions that improve the time predictability of the out-of-chip memory accesses. Finally, nowadays GPGPUs seem to be a hot research topic, and so changes and improvements in hardware architecture that improve time predictability can be adopted more easily.

In conclusion, the Fermi GPU architecture is suitable for real-time applications with soft deadlines. The initial results show that with few changes in the hardware architecture, it can have time predictable behavior and so it can be suitable for real time applications with more strict timing constrains.

References

[1] Leischner, Nikolaj, Vitaly Osipov, and Peter Sanders. "Fermi architecture white paper." (2009).

[2] Wong, Henry, M-M. Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. "Demystifying GPU microarchitecture through microbenchmarking." In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pp. 235-246. IEEE, 2010.

[3] Bakhoda, Ali, George L. Yuan, Wilson WL Fung, Henry Wong, and Tor M. Aamodt. "Analyzing CUDA workloads using a detailed GPU simulator." In*Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 163-174. IEEE, 2009.

[4] http://www.gpgpu-sim.org/

[5] Lai, Junjie, and André Seznec. "Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus." In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pp. 1-10. IEEE, 2013.

[6] NVIDIA, Nvidias next generation cuda compute architecture: Kepler gk110, Whitepaper, NVIDIA Corp., 2012.

[7] Wilhelm, Reinhard, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat et al. "The worst-case execution-time problem—overview of methods and survey of tools." *ACM Transactions on Embedded Computing Systems (TECS)* 7, no. 3 (2008): 36

[8] Butazzo, Giorgio C. "Hard real-time computing systems." *Predictable scheduling algorithms and applications* (3rd edition).

[9] Thiele, Lothar, and Reinhard Wilhelm. "Design for timing predictability." *Real-Time Systems* 28, no. 2-3 (2004): 157-177

[10] Kirner, Raimund, and Peter Puschner. "Time-predictable computing." In*Software Technologies for Embedded and Ubiquitous Systems*, pp. 23-34. Springer Berlin Heidelberg, 2011.

[11] Grund, Daniel, Jan Reineke, and Reinhard Wilhelm. "A Template for Predictability Definitions with Supporting Evidence." In *PPES*, pp. 22-31. 2011.

[12] Zhang, Wei, and Yiqiang Ding. "Standard deviation of CPI: A new metric to evaluate architectural time predictability." In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, pp. 111-112. IEEE, 2013.

[13] Schoeberl, Martin. "Is time predictability quantifiable?." In *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pp. 333-338. IEEE, 2012.

[14] Lickly, Ben, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. "Predictable programming on a precision timed architecture." In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pp. 137-146. ACM, 2008.

[15] Schoeberl, Martin. "Time-predictable computer architecture." *EURASIP Journal on Embedded Systems* 2009 (2009): 2.

[16] Zimmer, Michael, David Broman, Christopher Shaver, and Edward A. Lee. "FlexPRET: A Processor Platform for Mixed-Criticality Systems." In *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS). IEEE*. 2014.

[17] Prakash, Aayush, and Hiren D. Patel. "An instruction scratchpad memory allocation for the precision timed architecture." In *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 659-664. EDA Consortium, 2012.

[18] Burns, Alan, and David Griffin. "Predictability as an emergent behaviour." In*Robert I. Davis and Linh TX Phan, editeurs, 4th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, pp. 27-29. 2011.

[19] Heckmann, Reinhold, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. "The influence of processor architecture on the design and the results of WCET tools." *Proceedings of the IEEE* 91, no. 7 (2003): 1038-1054.

[20] Reineke, Jan, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. "Timing predictability of cache replacement policies." *Real-Time Systems* 37, no. 2 (2007): 99-122.

[21] Narasiman, Veynu, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. "Improving GPU performance via large warps and two-level warp scheduling." In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 308-317. ACM, 2011.

[22] Jog, Adwait, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. "Orchestrated scheduling and prefetching for gpgpus." In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 332-343. ACM, 2013.

[23] Schoeberl, Martin. *Jop: A java optimized processor for embedded real-time systems*. VDM Publishing, 2008.

[24] Betts, Adam, and Alastair Donaldson. "Estimating the WCET of GPU-accelerated applications using hybrid analysis." *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*. IEEE, 2013.

[25] Wegener, Joachim, and Frank Mueller. "A comparison of static analysis and evolutionary testing for the verification of timing constraints." *Real-Time Systems* 21.3 (2001): 241-268.

[26] Hirvisalo, Vesa. "On Static Timing Analysis of GPU Kernels." *14th International Workshop on Worst-Case Execution Time Analysis}*. Ed. Heiko Falk. Vol. 39. Schloss Dagstuhl--Leibniz-Zentrum fuer Informatik}, 2014.

[27] Chattopadhyay, Sudipta, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. "A unified WCET analysis framework for multicore platforms." *ACM Transactions on Embedded Computing Systems (TECS)* 13, no. 4s (2014): 124.

[28] Hong, Sunpyo, and Hyesoon Kim. "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness." *ACM SIGARCH Computer Architecture News*. Vol. 37. No. 3. ACM, 2009.

[29] Mushtaq, Hamid, Zaid Al-Ars, and Koen Bertels. "Accurate and efficient identification of worst-case execution time for multicore processors: A survey."*IDT*. 2013.

[30] van den Braak, Gert-Jan, Juan Gómez-Luna, Henk Corporaal, Jose Maria Gonzalez-Linares, and Nicolas Guil. "Simulation and architecture improvements of atomic operations on GPU scratchpad memory." In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pp. 357-362. IEEE, 2013.

[31] Lundqvist, Thomas, and Per Stenstrom. "Timing anomalies in dynamically scheduled microprocessors." *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*. IEEE, 1999.

[32] Wilhelm, Reinhard, et al. "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems." *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 28.7 (2009): 966-978.

[33] Akesson, Benny, Kees Goossens, and Markus Ringhofer. "Predator: a predictable SDRAM memory controller." *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, 2007.

[34] http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3FIq0pVOb

[35] Patterson, David. "The top 10 innovations in the new NVIDIA Fermi architecture, and the top 3 next challenges." *NVIDIA Whitepaper* (2009).

[36] https://code.google.com/p/asfermi/

[37] Nugteren, Cedric, Gert-Jan van den Braak, Henk Corporaal, and Henri Bal. "A Detailed GPU Cache Model Based on Reuse Distance Theory." In *HPCA-20: International Symposium on High Performance Computer Architecture. IEEE*. 2014.

[38] Hennessy, John L., and David A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

Appendix A: Detailed experiments

For completeness the experiments that were performed to identify and quantify the sources of timing variation are presented. The experiments presented in this section have the expected behavior. The experiments that expose the time variation are presented in section 5. First, the experiments for the arithmetic instructions are shown. Next the experiments for the shared memory are shown. Afterwards, the global memory is analyzed. Finally, the experiments for the different warp schedulers in GPGPU-Sim are shown.

A.1. Arithmetic instruction

For the arithmetic instructions the operations: addition (ADD), multiplication (MUL), cosine (COS), multiplication-addition (mad) were used. In Table 19 the execution time of the micro-benchmarks is presented for different number of operations with no dependencies. Table 19 shows that all the instructions consume the same amount of time. This means that the latency of the instruction is modified to fit with the slowest one. This behavior favors throughput and makes the pipeline easier to implement.
In addition, for the first instruction the execution time is measured as 30 clock cycles because we have to take into account also the cost of the measurement (one shift operation in this case). However, a 6 clock cycles delay is required for decoding and the scheduling steps that are necessary for every instruction. So, every 6 cycles a new instruction start.

Table 19 Execution time of arithmetic Instructions in clock cycles - No dependencies

#OP	ADD	MUL	COS	MAD
1	30	30	30	30
2	36	36	36	36
3	42	42	42	42
4	48	48	48	48
5	54	54	54	54

Similarly, in Table 20 are the results from the experiment where all the instructions have a RaW dependency. The first line is the same as in the previous experiments since it is only one instruction. For the next instructions, it can be noticed that a delay of 18 clock cycles is required for “ADD”,”MUL” and “MAD”. So the cost of waiting for the dependency is $18 - 6 = 12\, clock\ cycles$ for these instructions. The delay for the data dependency for the “COS” instruction is larger ($22 - 6 = 16\, clock\ cycles$). At the special function unit (SFU) the “COS” instruction is implemented.

The SFU is a more complex functional unit for special operations like cosine and sine. The implementation details of this functional unit are not available. But, we can assume that the SFU has a deeper pipeline than the SPs, so a stall may influence more stages in the pipeline. In addition, there are only 4 SFU so it is required to perform more steps compare to the instruction that they are implemented in SP.

Table 20 Execution time of arithmetic Instructions in clock cycles - RaW dependency.

#OP	ADD	MUL	COS	MAD
1	30	30	30	30
2	48	48	52	48
3	66	66	74	66
4	84	84	96	84
5	102	102	118	102

In Table 21 the execution time of mixed type arithmetic instructions is presented; namely “MUL” with “ADD”, “MAD” with “ADD”, and “COS” with “ADD”. With this experiment we want to check if the instructions are composable. In this experiment there is not dependency between the instructions. This experiment was performed to identify if the interaction of different functional units can influence the execution time. We can observe that the results in the table are the same with the experiment results where the same instructions were used. The difference between the two instructions set is 12 clock cycles (2*6).

Table 21 Execution time of mixed type arithmetic Instructions in clock cycles – No dependencies.

#OP	MUL ADD	MAD ADD	COS ADD
2	36	36	36
4	48	48	48
6	60	60	60
8	72	72	72

Similarly, Table 22 shows the results from the experiment where all the instructions have a RaW dependency. For example, in case that we have 4 operations, there are 3 RaW dependencies between the operations. As previously, the results from the experiments are the same with the ones with the instructions of the same type. In this work we focus on the instructions that are commonly used. We are not going to examine in more detail the instructions that are implemented in SFU.

Table 22 Execution time of mixed arithmetic Instructions in clock cycles – RaW dependencies.

#OP	MUL ADD	MAD ADD	COS ADD
2	48	48	52
4	84	86	92
6	120	122	132
8	156	160	172

As it is mentioned before, the experiments that were presented until now were using thread block size of 32. The following graphs in this subsection present the behavior of the GPU for different size of thread block. The experiments present the execution time of instructions for thread block sizes of 32, 64, 128, 256 and 512. On the vertical axis is the execution time in clock cycles. On the horizontal axis is presented the warp ID of warps in the different block sizes. A group of 32 threads form a warp which executes the same instruction if no divergence occurs.

Figure 50 shows the execution time of one add instruction for thread block of sizes of 32, 64, 128, 256 and 512. It can be seen that until 128 of thread block size the execution time is the same for all warps. However the execution time for the block sizes of 256 and 512 is increased. By increasing the number of warps, the execution time of every warp increases also since more warps have to be scheduled. The instructions of the warps are executed based on the LRR algorithm. So by increasing the number of warps, the distance between two instructions from the same warp will increase also and as a consequence the execution time of the warp will increase also.

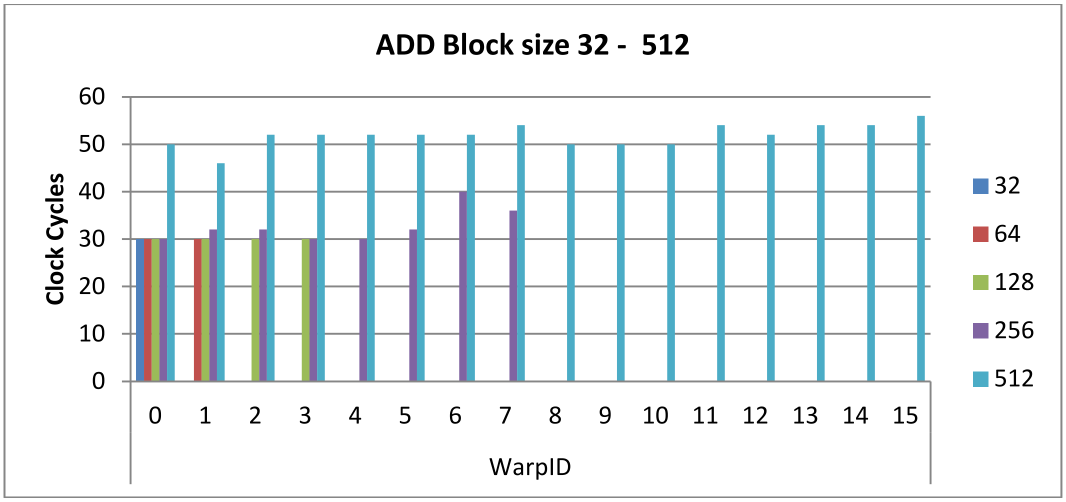


Figure 50 one add instruction for different thread block sizes.

We execute the experiment 8 times for a thread block size of 512 threads in order to identify if the procedure is time predictable. The results are presented in Figure 51 and it can be noticed that for all the experiments the execution time of the different warps is the same.

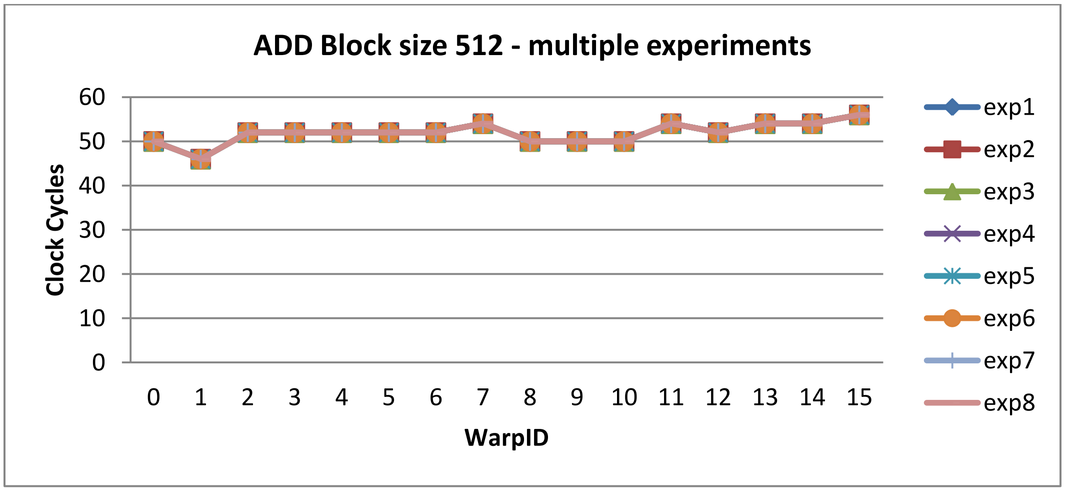


Figure 51 multiple experiments for one add instruction executed by 512 threads (16 warps).

Similar experiments were performed for the mad instruction. The results are presented in Figure 52 and Figure 53 and follow the same pattern with the add instruction. However, for thread block size of 256 and 512 it can be seen that the execution times of warps have different values compared to the execution times of the add instruction. For instance, in case of block size of 512 for add warp 1 is the fastest one, while for mad warp 7 is the fastest one. This level of detail was not possible to be analyzed without the appropriate documentation, so we were not able to identify the reason for the different behavior of "MAD" and "ADD" for thread block sizes of 256 and 512.

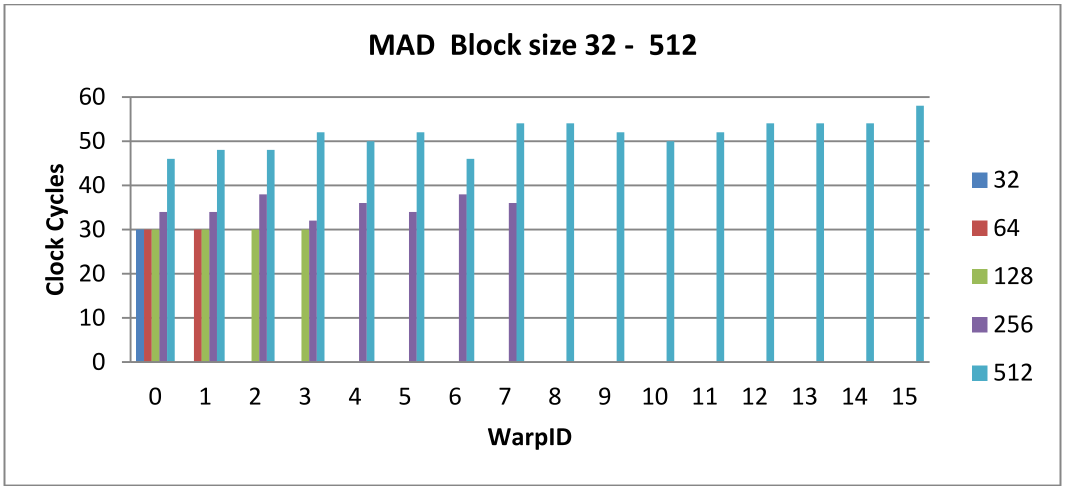


Figure 52 One mad instruction for different thread block sizes.

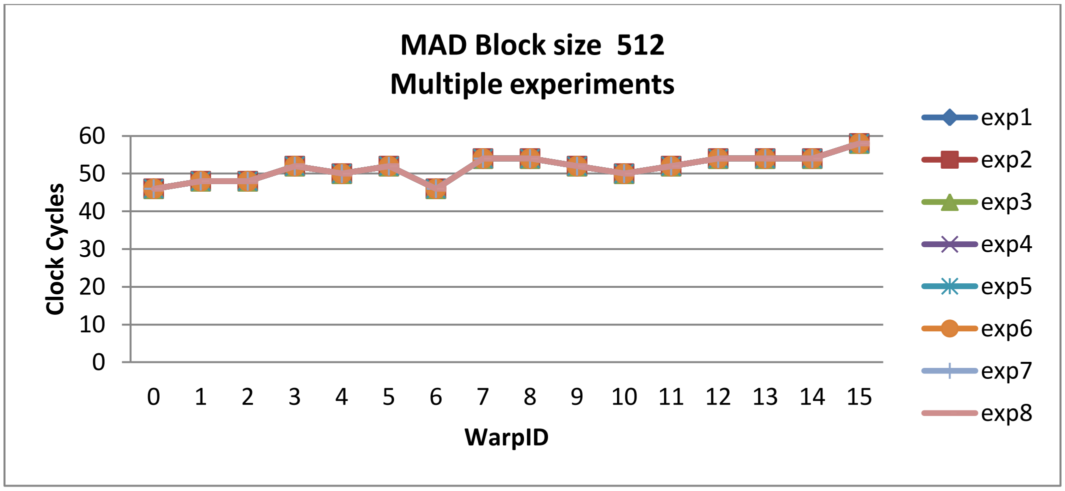


Figure 53 multiple experiments for one mad instruction executed by 512 threads (16 warps).

The next experiments are using a mad and add instruction for thread block sizes as used before. Figure 54 shows the results for the experiment where no data dependencies are used. The results are similar to the previous experiments, however it can be noticed that the execution time for thread block size of 256 has a bigger variation from warp to warp compared to the separate execution of mad and add.

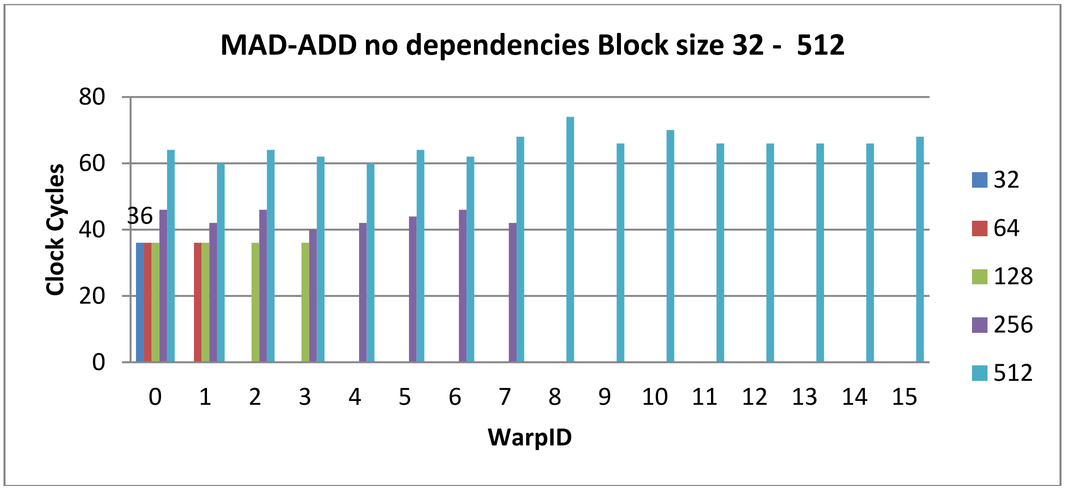


Figure 54 mad-add instructions with no dependency for different thread block sizes

Similar with the previous cases, the experiments performed multiple times to identify if the execution times of the two instructions differ from experiment to experiment. In Figure 55 it can be seen that all the experiments have the same execution time, so no variation in the execution time was observed.

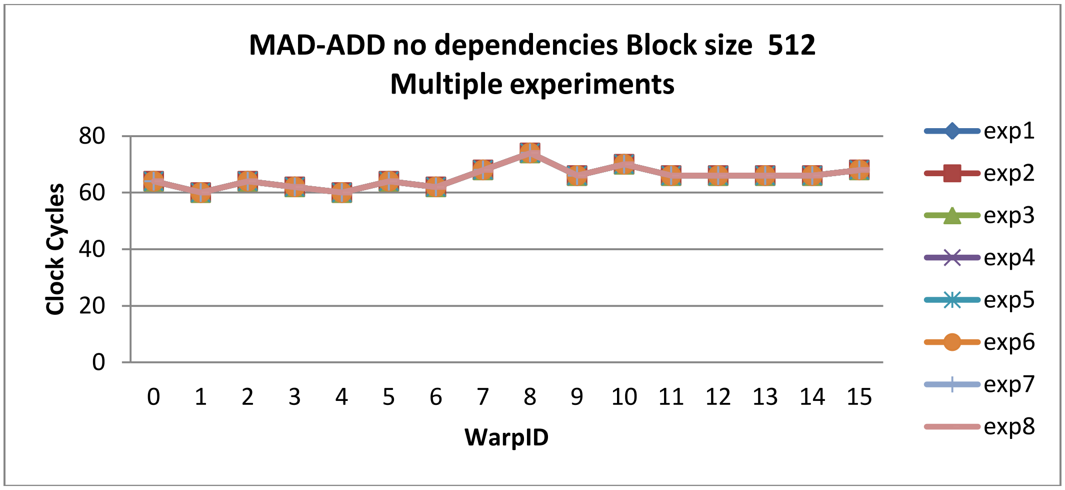


Figure 55 Multiple experiments for mad add instruction with no dependency executed by 512 threads (16 warps).

A.2.Shared memory

In this section the instruction for the scratchpad shared memory are presented. The behavior is similar to the arithmetic instructions. Again, as before, first we are going to perform experiments with few instructions. Afterward the experiments are executed multiple times to check if the results are the same from execution to execution.

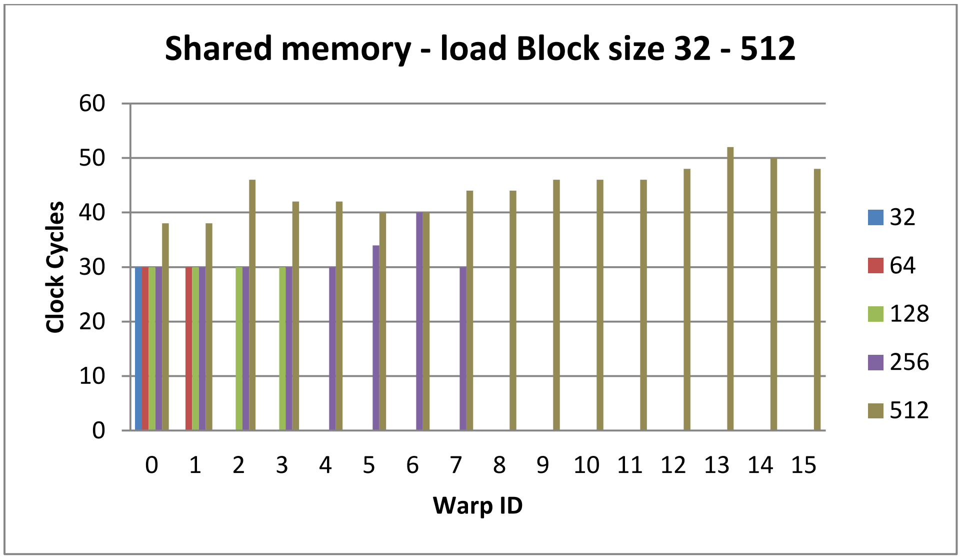


Figure 56 one load to the shared memory instruction for different thread block sizes

Figure 56 shows the execution time of one load instruction to the shared memory. The behavior is the same as with the arithmetic instructions. The horizontal axis show the warp ID for the different warps and the vertical axis shows the execution time in clock cycles. For thread block sizes 32, 64 and 128 the execution time remains the same due to the restrictions of the scheduler that we explained in sections 5 and 6. Similarly with the arithmetic instructions the execution time for thread block size 256 and 512 is increased since more than 6 warps have to be scheduled.

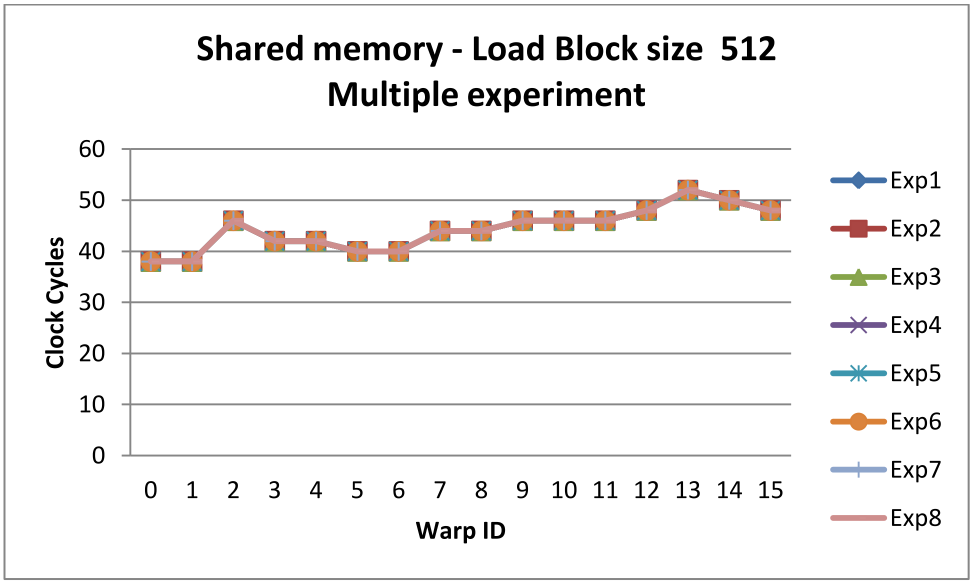


Figure 57 multiple experiments for one load to shared memory instruction executed by 512 threads (16 warps)

We performed the same experiment multiple times. For all 8 experiments the execution times of the different warps were the same. This behavior is expected since the scratchpad memory is a software controlled, on-chip memory and there is no interference with any other component.

In the same way the experiments for the store instruction to the shared memory were performed. In Figure 58 the execution time of the one store instruction is presented for block sizes of 32, 64, 128, 256 and 512. Again, the execution time of thread block sizes 32 to 128 is the same as executing only one warp. The execution time for one store to the shared memory for thread block sizes 256 and 512 the execution time increases for the same reasons that we mentioned before.

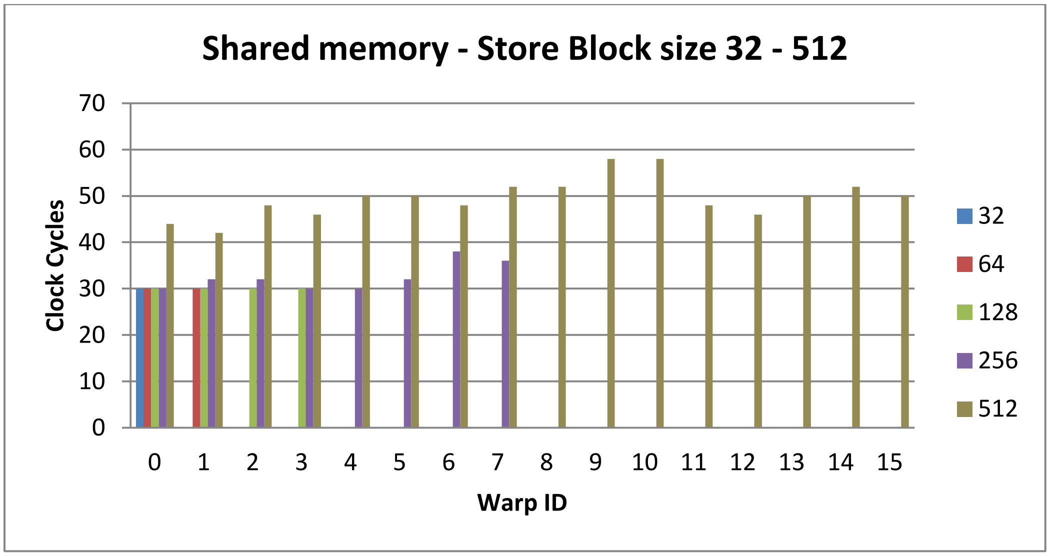


Figure 58 one load to the shared memory instruction for different thread block sizes

The experiment for one store instruction to the on-chip scratchpad shared memory was performed multiple times in order to identify different execution times from experiment to experiment.

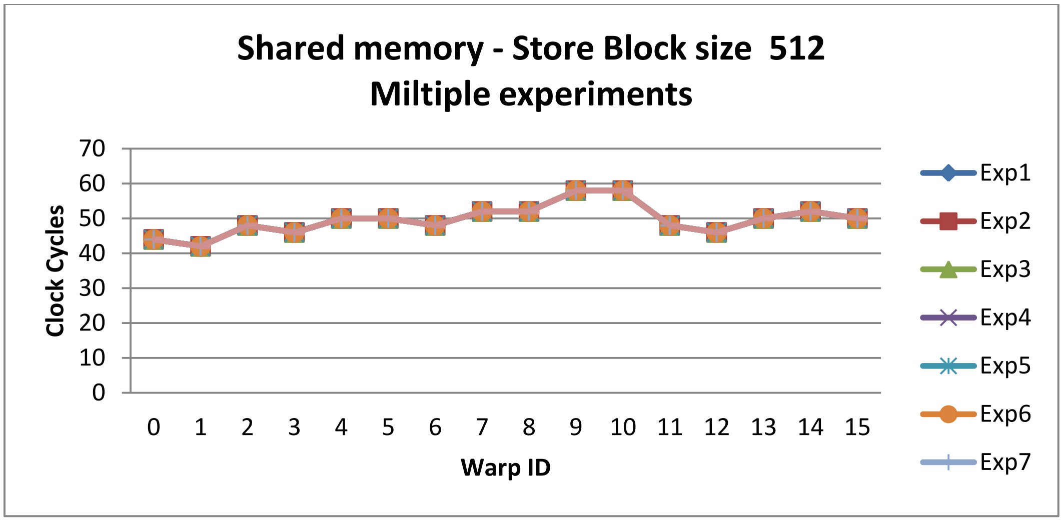


Figure 59 multiple experiments for one store to shared memory instruction executed by 512 threads (16 warps)

From the experiments we can see that they same execution time so the scratchpad memory has time predictable behavior also for store instruction.

The shared memory is analyzed for multiple instructions. For the same thread block sizes that we have used before we check how the execution time changes for the increasing number of loads and stores to the shared memory. Figure 60 and Figure 61 show the latency of load and store to the shared memory for the increasing number of instructions. The horizontal axis presents the number of operations and the vertical axis the execution time in clock cycles. We present only the slowest warp since the slowest warp determines the overall execution time.

From the two diagrams it can be seen that the latency for store and load to the shared memory increases linearly with the linear increase of the instructions for all thread block sizes. Moreover, for thread block sizes 32, 64, 128 and 256 the execution times of the slowest warp are the same. The lines are on top of each other. For the case of 512 threads the execution time increases since.

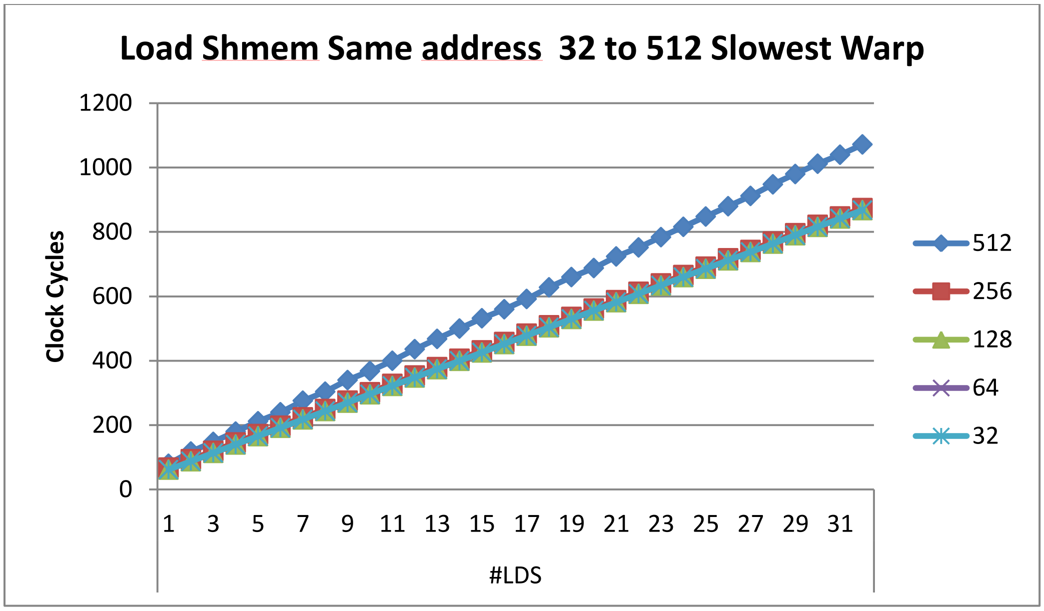


Figure 60 Latency of the load to shared memory instruction for increasing number of instructions

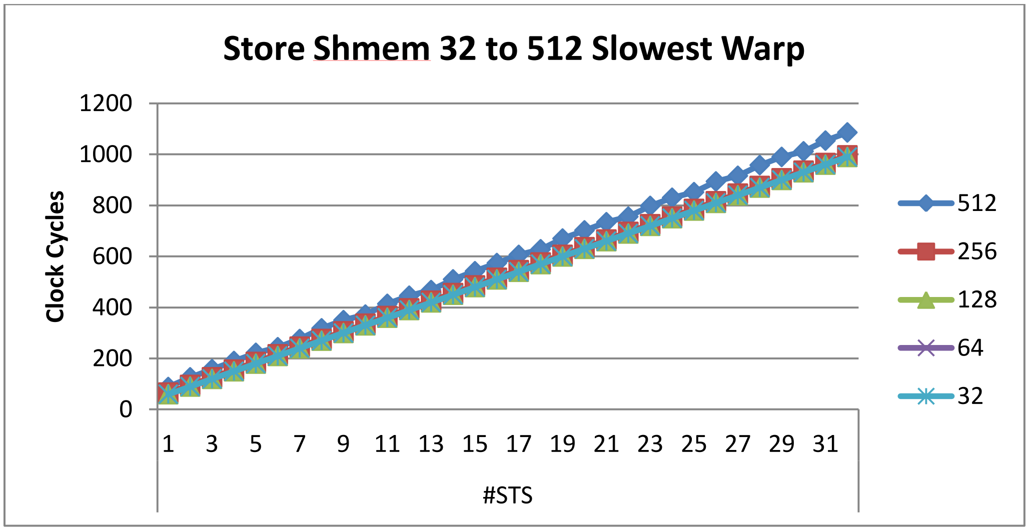


Figure 61 Latency of the store to shared memory instruction for increasing number of instructions

A.3. Global memory

In this section the execution time of load and store to the global memory is discussed. The experiment performed in a similar way as before. The horizontal axis show the number of operations and the vertical axis show the execution time in clock cycles. Figure 62 shows the latency of 32 load instruction to the global memory of the slowest warp for thread block size of 32, 64, 128, 256 and 512. For all the case the latency of load instruction increases linearly with a linear increase of the number of the instructions. Furthermore, the slop for all the thread block sizes is the same but they start from different initial points.

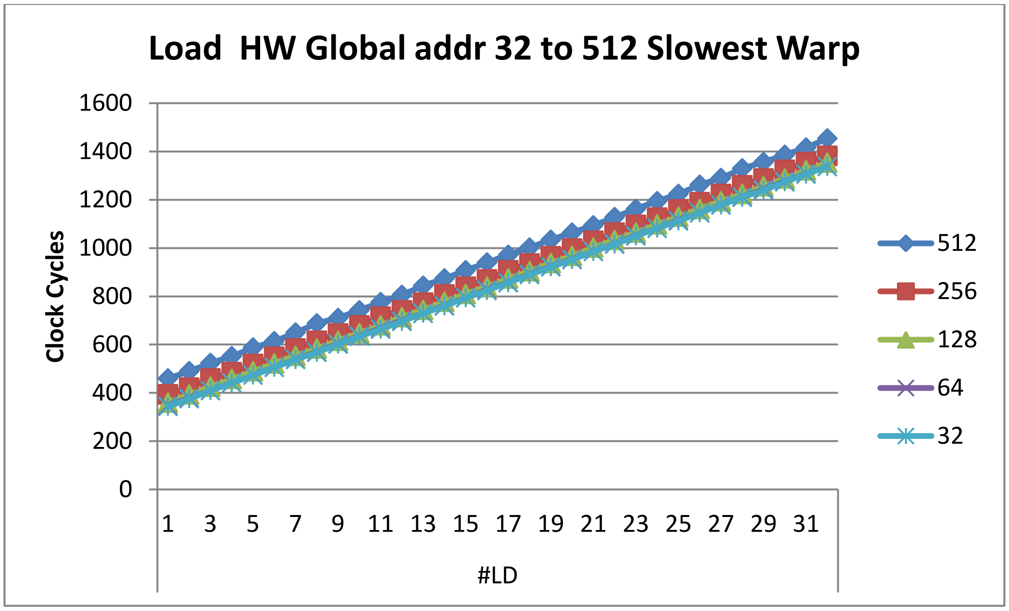


Figure 62 Thirty two load instructions to the global memory for thread block sizes of 32, 64, 128, 256 and 512.

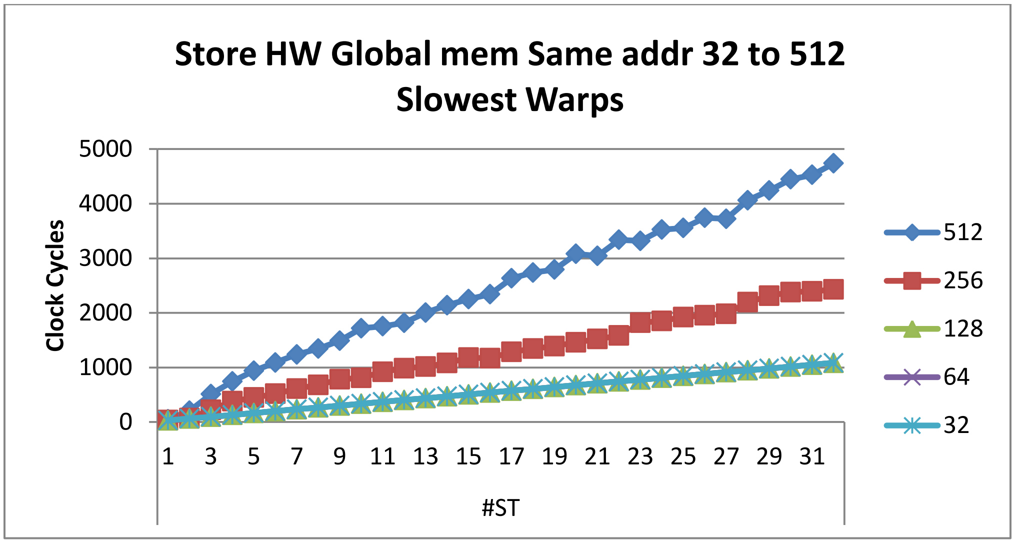


Figure 63 Thirty two store instructions to the global memory for thread block sizes of 32, 64, 128, 256 and 512.

Figure 63 shows 32 stores to the global memory for the slowest warp of thread block sizes of 32, 64, 128, 256 and 512. For thread block sizes of 32, 64 and 128 the execution time is the same. For these three cases the execution time increases linearly with the linear increase to the number of instructions. For the cases of 256 and 512 thread block sizes Compared to load the execution time of thread block sizes is smaller but for thread block sizes of 256 and 512 is m.

Appendix B: Detailed GPGPU-Sim diagram

In this section the architecture of the GPGU-Sim is analyzed in more detail.
Figure 64 shows a detail architecture block diagram of the GPGPU-Sim [3]. It was made based on Figure 10. The components of Figure 10 were extracted. The detail functionality can be found in [ref].

The diagram is separated in two parts. First there is the SIMT front. From the instruction cache the instructions are fetched. The instructions are decoded and stored in the instruction buffer. The RaW and WaW dependencies are checked in the scoreboard. From the implementation of GPGU-Sim we found that there is only one scoreboard for the two schedulers in comparison with Figure 10 that suggest that there is one for every scheduler. The scheduler chooses which warp is going to be issued from the dispatch unit. Furthermore, the SIMT-Stack is used to handle the branch divergence. It stores the diverge (PR) and converge points (RPC) of each warp.

The SIMD datapath consists of the operand collector, the shared memory (scratchpad and level 1 data cache) and the SIMD execution units (SP and SFU). In practice the register file is a small SRAM memory with 4 banks and arbitration logic that is responsible to avoid the bank conflicts.

The blue parts show the memories (instruction and data). The green parts are the buffer of the system; scoreboard, PC for the all the threads, instruction buffer, SIMT stack and collector units. The orange parts are the arbiters of the architecture. The brown color shows the register file with the four banks. The purple color shows the SIMD execution units and load store units.

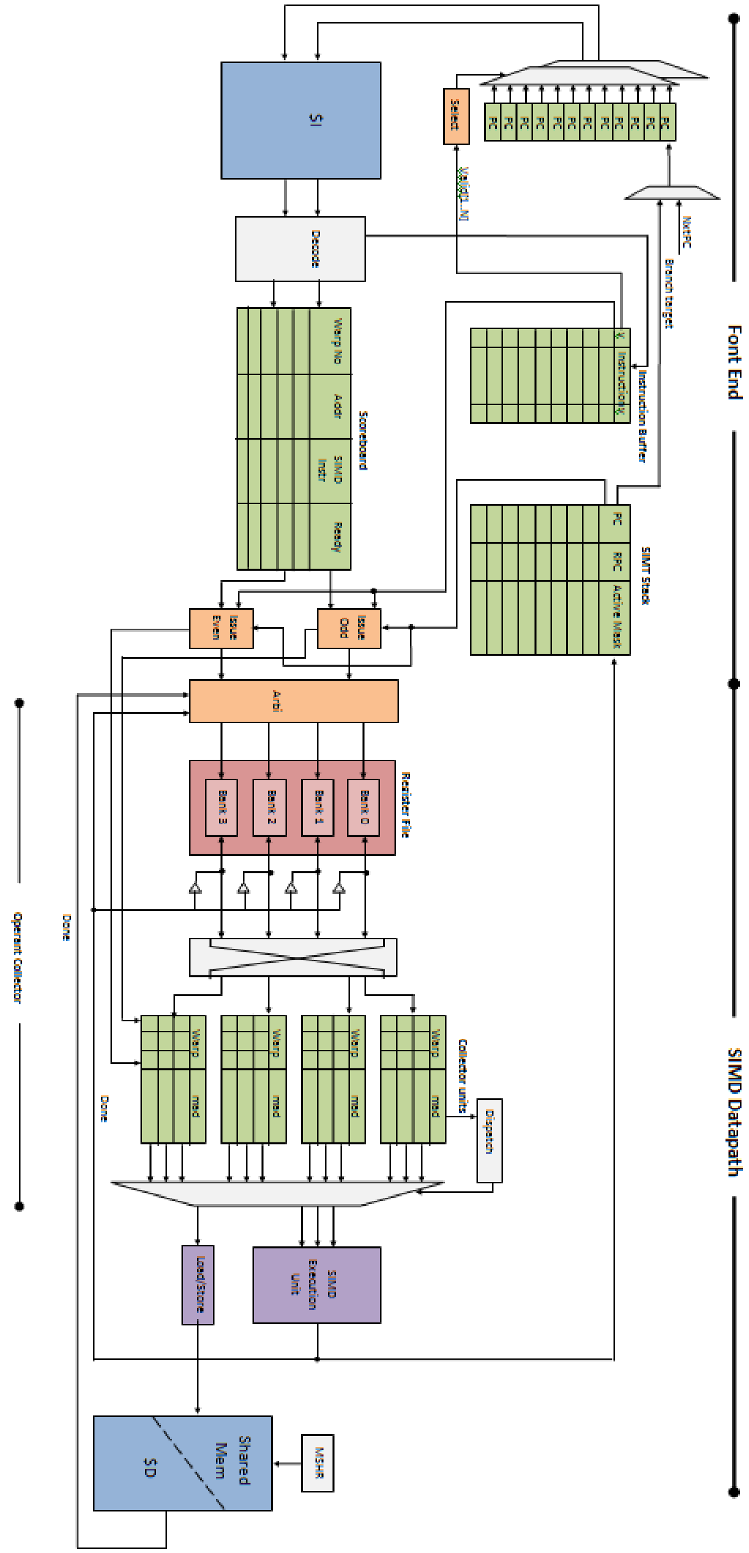


Figure 64 detailed GPGPU-SIM architecture diagram

Appendix C: Program trace from GPGPU-Sim

The tables show the three different states that the warp can be. The first column present the starting time in clock cycles. The other two tables show the warps. In the first column is the WID and the second column the instruction that is executed. Color meaning

- Green: The warp is executed.
- Red: The warp is stalled.
- Yellow: The warp is stalled in scorebord, so it has to wait for the dependencies to be resolved.
- Purple: The execution units are not available.

Active

	Odd Scheduler		Even Scheduler	
Starting time	WID	Instruction	WID	Instruction
85	9	add.u32 t16, t16, 10	0	add.u32 t15, t15, 10
86	11	add.u32 t16, t16, 10	2	mov.u32 time1, %clock
87	13	add.u32 t16, t16, 10	4	mov.u32 time1, %clock
88	15	add.u32 t16, t16, 10	6	mov.u32 time1, %clock

Idle because of RaW dependencies.

Starting time	WID	Instruction	WID	Instruction
250	15	add.u64 %rd4, %rd3, %rd2	4,10,12	Scoreboard delay
251	7,9,11	Scoreboard delay	6	st.global.u32 [%rd6+0], %r9
252	9,11,15	Scoreboard delay	8	ld.param.u64 %rd5
253	11,15	Scoreboard delay	10,12,4	Scoreboard delay
254	9,11,15	Scoreboard delay	14	ld.param.u64 %rd5
255	9,11,15	Scoreboard delay	0	ld.param.u64 %rd5

Idle because the execution units are not available.

Starting time	WID	Instruction	WID	Instruction
108	1	mov.u32 time1, %clock	6	add.u32 t16, t15, t14
109	3,5,7,9,13	N/A SP&SFU	8	add.u32 t13, t12, t11
110	1,3,5,7,9,11	N/A SP&SFU	10	add.u32 t13, t12, t11
111	1,3,5,7,9,11	N/A SP&SFU	12	add.u32 t13, t12, t11
112	3	add.u32 t13, t12, t11	14	shl.b32 time1, time1, 1
113	5	shl.b32 time1, time1, 1	0	shl.b32 time1, time1, 1
114	7	add.u32 t13, t12, t11	2	mov.u32 time2, %clock
115	9	add.u32 t13, t12, t11	4	add.u32 t16, t15, t14
116	11	add.u32 t13, t12, t11	6	mov.u32 time2, %clock
117	1,3,7,9,11,5	N/A SP&SFU	8	add.u32 t16, t15, t14
118	1,3,5,7,11,5	N/A SP&SFU	0,4,10,12,14	N/A SP&SFU

Appendix D: Acronyms and Abbreviations

	Meaning		Meaning
ACET	Average Case Execution Time	NVCC	Nvidia CUDA Compiler
ADD	Addition	OoO	Out of Order
BCET	Best Case Execution Time	OpenCL	Open Computing Language
CC	Clock Cycles	PCI	Peripheral Component Interconnect
CPI	Cycles Per Instruction	PTX	Parallel Thread Execution
CPU	Central Process Unit	RAM	Random Access Memory
CTA	Cooperative Thread Array	RaW	Read after Write
CUBIN	CUDA Binary	S2R	Store special register to general-purpose Register.
CUDA	Compute Unified Device Architecture	SDK	software development kit
D2H	Device to Host	SFU	Special Function Unit
DMA	Direct Memory Access	SHL	Shift Left
GDDR	graphics double data rate	SIMD	Single Instruction Multiple Data
GDRAM	Graphic Dynamic Random Access Memory	SIMT	Single Instruction Multiple Threads
GPGPU	General Purpose Graphics Process Unit	SM	Streaming Multiprocessor
GPGPU-Sim	General Purpose Graphics Process Unit - Simulator	SP	Streaming processor
GPU	Graphics Process Unit	SPARC	Scalable Processor ARChitecture
GTO	Greedy Then Oldest	ST.E	Store External
H2D	Host to Device	STS	Store shared
ISA	Instruction Set Architecture	TBi	Thread Block with ID i
KB	Kilo Byte	Ti	Thread with ID i
L1 $D	Level 1 Data cache	TL	Two Level
L1 $I	Level 1 Instruction cache	TLP	Thread Level Parallelism
LB	Lower Bound	UB	Upper Bound
LD.E	Load External	WaW	Write after Write
LD/ST	Load/Store	WCET	Worst Case Execution Time
LDS	Load shared	Wi	Warp with ID i
LRR	Loose Round Robin	WID	warp ID
LRU	Least Recently Used		
MAD	multiply addition		
MWP	Memory Warp Parallelism		