Memory-level and Thread-level Parallelism Aware GPU Architecture Performance Analytical Model

Sunpyo Hong Hyesoon Kim
ECE School of Computer Science
Georgia Institute of Technology
{shong9, hyesoon}@cc.gatech.edu

Abstract

GPU architectures are increasingly important in the multi-core era due to their high number of parallel processors. Programming thousands of massively parallel threads is a big challenge for software engineers, but understanding the performance bottlenecks of those parallel programs on GPU architectures to improve application performance is even more difficult. Current approaches rely on programmers to tune their applications by exploiting the design space exhaustively without fully understanding the performance characteristics of their applications.

To provide insights into the performance bottlenecks of parallel applications on GPU architectures, we propose a simple analytical model that estimates the execution time of massively parallel programs. The key component of our model is estimating the number of parallel memory requests (we call this the memory warp parallelism*) by considering the number of running threads and memory bandwidth. Based on the degree of memory warp parallelism, the model estimates the cost of memory requests, thereby estimating the overall execution time of a program. Comparisons between the outcome of the model and the actual execution time in several GPUs show that the geometric mean of absolute error of our model on micro-benchmarks is 5.4% and on GPU computing applications is 13.3%. All the applications are written in the CUDA programming language.*

1. Introduction

The increasing computing power of GPUs gives them considerably higher peak computing power than CPUs. For example, NVIDIA's GTX280 GPUs [3] provide 933 Gflop/s with 240 cores, while Intel's Core2Quad processors [2] deliver only 100 Gflop/s. Intel's next generation of graphics processors will support more than 900 Gflop/s [35]. AMD/ATI's latest GPU (HD5870) provides 2.72 Tflop/s [1]. However, even though hardware is providing high performance computing, writing parallel programs to take full advantage of this high performance computing power is still a big challenge.

Recently, there have been new programming languages that aim to reduce programmers' burden in writing parallel applications for the GPUs such as Brook+ [5], CUDA [30], and OpenCL [21]. However, even with these newly developed programming languages, programmers still need to spend enormous time and effort to optimize their applications to achieve better performance [32]. Although the GPGPU community [15] provides general guidelines for optimizing applications using CUDA, *clearly* understanding

various features of the underlying architecture and the associated performance bottlenecks in their applications is still remaining homework for programmers. Therefore, programmers might need to vary all the combinations to find the best performing configurations [32].

To provide insight into performance bottlenecks in massively parallel architectures, especially GPU architectures, we propose a simple analytical model. The model can be used statically without executing an application. The basic intuition of our analytical model is that estimating the cost of memory operations is the key component of estimating the performance of parallel GPU applications. The execution time of an application is dominated by the latency of memory instructions, but the latency of each memory operation can be hidden by executing multiple memory requests concurrently. By using the number of concurrently running threads and the memory bandwidth consumption, we estimate how many memory requests can be executed concurrently, which we call *memory warp*[1] *parallelism (MWP)*.We also introduce *computation warp parallelism (CWP)*. CWP represents how much computation can be done by other warps while one warp is waiting for memory values. CWP is similar to a metric, arithmetic intensity[2][31] in the GPGPU community. Using both MWP and CWP, we estimate effective costs of memory requests, thereby estimating the overall execution time of a program.

We evaluate our analytical model based on the CUDA [28, 30] programming language, which is C with extensions for parallel threads. We compare the results of our analytical model with the actual execution time on several GPUs. Our results show that the geometric mean of absolute errors of our model on micro-benchmarks is 5.4% and on the Merge benchmarks [23][3] is 13.3%

The contributions of our work are as follows:

1. To the best of our knowledge, we propose the first analytical model for the GPU architecture. This can be easily extended to other multithreaded architectures as well.
2. We propose two new metrics, MWP and CWP, to represent the degree of warp level parallelism that provide key insights identifying performance bottlenecks.

[1] A warp is a batch of threads that are internally executed together by the hardware. Section 2 describes a warp.
[2] Arithmetic intensity is defined as math operations per memory operation.
[3] The Merge benchmarks consist of several media processing applications.

2. Background and Motivation

We provide a brief background on the GPU architecture and programming model that we modeled. Our analytical model is based on the CUDA programming model and the NVIDIA Tesla architecture [3, 10, 28] used in the GeForce 8-series GPUs.

2.1. Background on the CUDA Programming Model

The CUDA programming model is similar in style to a single-program multiple-data (SPMD) software model. The GPU is treated as a coprocessor that executes data-parallel kernel functions.

CUDA provides three key abstractions, a hierarchy of thread groups, shared memories, and barrier synchronization. Threads have a three level hierarchy. A grid is a set of thread blocks that execute a kernel function. Each grid consists of blocks of threads. Each block is composed of hundreds of threads. Threads within one block can share data using shared memory and can be synchronized at a barrier. All threads within a block are executed concurrently on a multithreaded architecture.

The programmer specifies the number of threads per block, and the number of blocks per grid. A thread in the CUDA programming language is much lighter weight than a thread in traditional operating systems. A thread in CUDA typically processes one data element at a time. The CUDA programming model has two shared read-write memory spaces, the shared memory space and the global memory space. The shared memory is local to a block and the global memory space is accessible by all blocks. CUDA also provides two read-only memory spaces, the constant space and the texture space, which reside in external DRAM, and are accessed via read-only caches.

2.2. Background on the GPU Architecture

Figure 1 shows an overview of the GPU architecture. The GPU architecture consists of a scalable number of *streaming multiprocessors* (SMs), each containing eight *streaming processor* (SP) cores, two special function units (SFUs), a multithreaded instruction fetch and issue unit, a read-only constant cache, and a 16KB read/write shared memory [10].

The SM executes a batch of 32 threads together called a *warp*. Executing a warp instruction applies the instruction to 32 threads, similar to executing a SIMD instruction like an SSE instruction [18] in X86. However, unlike SIMD instructions, the concept of warp is not exposed to the programmers, rather

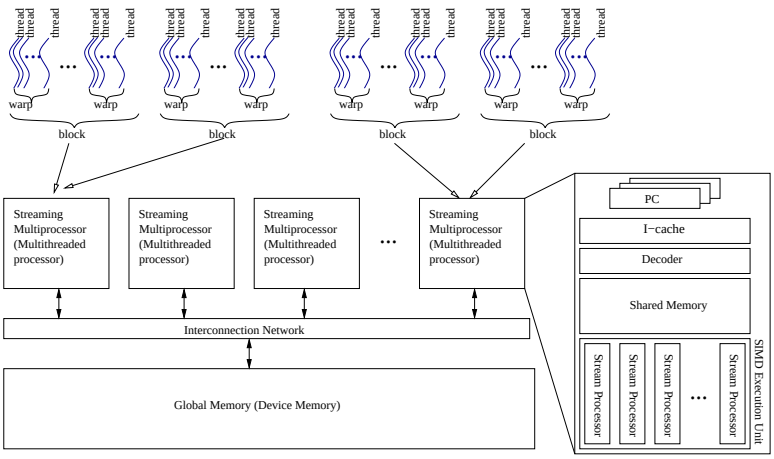

Figure 1. An overview of the GPU architecture

programmers write a program for one thread, and then specify the number of parallel threads in a block, and the number of blocks in a kernel grid. The Tesla architecture forms a warp using a batch of 32 threads [17, 11] and in the rest of the paper we also use a warp as a batch of 32 threads.

All the threads in one block are executed on one SM together. One SM can also have multiple concurrently running blocks. The number of blocks that are running on one SM is determined by the resource requirements of each block such as the number of registers and shared memory usage. The blocks that are running on one SM at a given time are called *active blocks* in this paper. Since one block typically has several warps (the number of warps is the same as the number of threads in a block divided by 32), the total number of active warps per SM is equal to the number of warps per block times the number of active blocks.

The shared memory is implemented within each SM multiprocessor as an SRAM and the global memory is part of the offchip DRAM. The shared memory has very low access latency (almost the same as that of register) and high bandwidth. However, since a warp of 32 threads access the shared memory together, when there is a bank conflict within a warp, accessing the shared memory takes multiple cycles.

2.3. Coalesced and Uncoalesced Memory Accesses

The SM processor executes one warp at one time, and schedules warps in a time-sharing fashion. The processor has enough functional units and register read/write ports to execute 32 threads (i.e. one warp) together. Since an SM has only 8 functional units, executing 32 threads takes 4 SM processor cycles for computation instructions.[4]

[4]In this paper, a computation instruction means a non-memory instruction.

When the SM processor executes a memory instruction, it generates memory requests and switches to another warp until all the memory values in the warp are ready. Ideally, all the memory accesses within a warp can be combined into one memory transaction. Unfortunately, that depends on the memory access pattern within a warp. If the memory addresses are sequential, all of the memory requests within a warp can be coalesced into a single memory transaction. Otherwise, each memory address will generate a different transaction. Figure 2 illustrates two cases. The CUDA manual [30] provides detailed algorithms to identify types of coalesced/uncoalesced memory accesses. If memory requests in a warp are uncoalesced, the warp cannot be executed until all memory transactions from the same warp are serviced, which takes significantly longer than waiting for only one memory request (coalesced case).

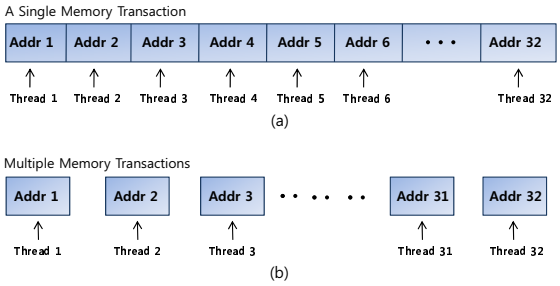

Figure 2. Memory requests from a single warp. (a) coalesced memory access (b) uncoalesced memory access

2.4. Motivating Example

To motivate the importance of a static performance analysis on the GPU architecture, we show an example of performance differences from three different versions of the same algorithm in Figure 3. The SVM benchmark is a kernel extracted from a face classification algorithm [38]. The performance of applications is measured on NVIDIA QuadroFX5600 [4]. There are three different optimized versions of the same SVM algorithm: *Naive*, *Constant*, and *Constant+Optimized*. *Naive* uses only the global memory, *Constant* uses the cached read-only constant memory[5], and *Constant+Optimized* also optimizes memory accesses[6] on top of using the constant memory. Figure 3 shows the execution time when the number of threads per block is varied. Even though the number of threads per block is varied, the number of blocks is adjusted to keep the total work the same. The performance improvement of *Constant+Optimized* and that of *Constant* over the *Naive* implementation are 24.36x and 1.79x respectively. Even though the

[5]The benefits of using the constant memory are (1) it has an on-chip cache per SM and (2) using the constant memory can reduce register usage, which might increase the number of running blocks in one SM.

[6]The programmer optimized the code to have coalesced memory accesses instead of uncoalesced memory accesses.

performance of each version might be affected by the number of threads, once the number of threads exceeds 64, the performance does not vary significantly.

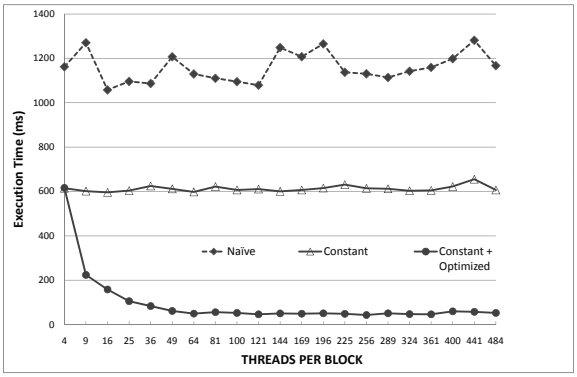

Figure 3. Optimization impacts on SVM

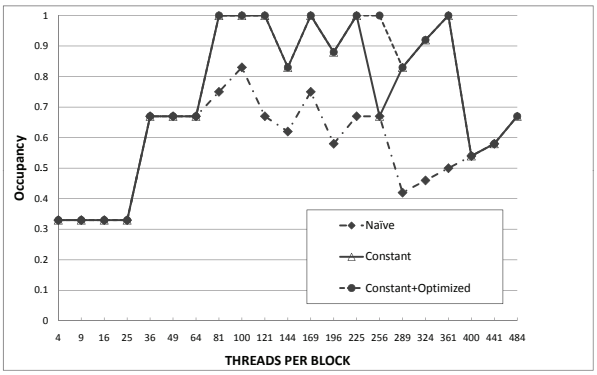

Figure 4. Occupancy values of SVM

Figure 4 shows SM processor occupancy [30] for the three cases. The SM processor occupancy indicates the resource utilization, which has been widely used to optimize GPU computing applications. It is calculated based on the resource requirements for a given program. Typically, high occupancy (the max value is 1) is better for performance since many actively running threads would more likely hide the DRAM memory access latency. However, SM processor occupancy does not *sufficiently* estimate the performance improvement as shown in Figure 4. First, when the number of threads per block is less than 64, all three cases show the same occupancy values even though the performances of 3 cases are different. Second, even though SM processor occupancy is improved, for some cases, there is no performance improvement. For example, the performance of *Constant* is not improved at all even though the SM processor occupancy is increased from 0.35 to 1. Hence, we need other metrics to differentiate the three cases and to understand what the critical component of performance is.

3. Analytical Model

3.1. Introduction to MWP and CWP

The GPU architecture is a multithreaded architecture. Each SM can execute multiple warps in a time-sharing fashion while one or more warps are waiting for memory values. As a result, the execution cost of warps that are executed concurrently can be hidden. The key component of our analytical model is finding out how many memory requests can be serviced and how many warps can be executed together while one warp is waiting for memory values.

To represent the degree of warp parallelism, we introduce two metrics, *MWP (Memory Warp Parallelism)* and *CWP (Computation Warp Parallelism).* MWP represents the maximum number of warps per SM that can access the memory simultaneously during the time period from right after the SM processor executes a memory instruction from one warp (therefore, memory requests are just sent to the memory system) until all the memory requests from the same warp are serviced (therefore, the processor can execute the next instruction from that warp). The warp that is waiting for memory values is called a *memory warp* in this paper. The time period from right after one warp sent memory requests until all the memory requests from the same warp are serviced is called one memory warp waiting period. CWP represents the number of warps that the SM processor can execute during one memory warp waiting period plus *one.* A value one is added to include the warp itself that is waiting for memory values. (This means that CWP is always greater than or equal to 1.)

MWP is related to how much memory parallelism in the system. MWP is determined by the memory bandwidth, memory bank parallelism and the number of running warps per SM. MWP plays a very important role in our analytical model. When MWP is higher than 1, the cost of memory access cycles from (MWP-1) number of warps is all hidden, since they are all accessing the memory system together. The detailed algorithm of calculating MWP will be described in Section 3.3.1.

CWP is related to the program characteristics. It is similar to an arithmetic intensity, but unlike arithmetic intensity, higher CWP means less computation per memory access. CWP also considers timing information but arithmetic intensity does not consider timing information. CWP is mainly used to decide whether the total execution time is dominated by computation cost or memory access cost. When CWP is greater than MWP, the execution cost is dominated by memory access cost. However, when MWP is greater than CWP, the execution cost is dominated by computation cost. How to calculate CWP will be described in Section 3.3.2.

3.2. The Cost of Executing Multiple Warps in the GPU architecture

To explain how executing multiple warps in each SM affects the total execution time, we will illustrate several scenarios in Figures 5, 6, 7 and 8. A computation period indicates the period when instructions from one warp are executed on the SM processor. A memory waiting period indicates the period when

memory requests are being serviced. The numbers inside the computation period boxes and memory waiting period boxes in Figures 5, 6, 7 and 8 indicate a warp identification number.

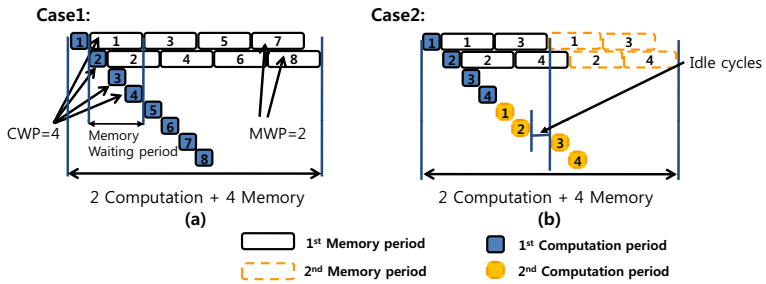

Figure 5. Total execution time when CWP is greater than MWP: (a) 8 warps (b) 4 warps

3.2.1. CWP is Greater than MWP For Case 1 in Figure 5a, we assume that all the computation periods and memory waiting periods are from different warps. The system can service two memory warps simultaneously. Since one computation period is roughly one third of one memory waiting warp period, the processor can finish 3 warps' computation periods during one memory waiting warp period. (i.e., MWP is 2 and CWP is 4 for this case.) As a result, the 6 computation periods are completely overlapped with other memory waiting periods. Hence, only 2 computations and 4 memory waiting periods contribute to the total execution cycles.

For Case 2 in Figure 5b, there are four warps and each warp has two computation periods and two memory waiting periods. The second computation period can start only after the first memory waiting period of the same warp is finished. MWP and CWP are the same as Case 1. First, the processor executes four of the first computation periods from each warp one by one. By the time the processor finishes the first computation periods from all warps, two memory waiting periods are already serviced. So the processor can execute the second computation periods for these two warps. After that, there are no ready warps. The first memory waiting periods for the remaining two warps are still not finished yet. As soon as these two memory requests are serviced, the processor starts to execute the second computation periods for the other warps. Surprisingly, even though there are some idle cycles between computation periods, the total execution cycles are the same as Case 1. When CWP is higher than MWP, there are enough warps that are waiting for the memory values, so the cost of computation periods can be almost always hidden by memory access periods.

For both cases, the total execution cycles are only the sum of 2 computation periods and 4 memory waiting periods. Using MWP, the total execution cycles can be calculated using the below two equations. We divide $Comp\_cycles$ by $\#Mem\_insts$ to get the number of cycles in one computation period.

$$Exec\_cycles = Mem\_cycles \times \frac{N}{MWP} + Comp\_p \times MWP \tag{1}$$

$$Comp\_p = Comp\_cycles / \#Mem\_insts \tag{2}$$

Mem_cycles: Memory waiting cycles per warp (see Equation (18))

$Comp\_cycles$: Computation cycles per warp (see Equation (19))

$Comp\_p$: Execution cycles of one computation period

$\#Mem\_insts$: Number of memory instructions per warp

N: Number of active running warps per SM

3.2.2. MWP is Greater than CWP In general, CWP is greater than MWP. However, for some cases, MWP is greater than CWP. Let's say that the system can service 8 memory warps concurrently. Again CWP is still 4 in this scenario. In this case, as soon as the first computation period finishes, the processor can send memory requests. Hence, a memory waiting period of a warp always immediately follows the previous computation period. If all warps are independent, the processor continuously executes another warp. Case 3 in Figure 6a shows the timing information. In this case, the memory waiting periods are all overlapped with other warps except the last warp. The total execution cycles are the sum of 8 computation periods and only one memory waiting period.

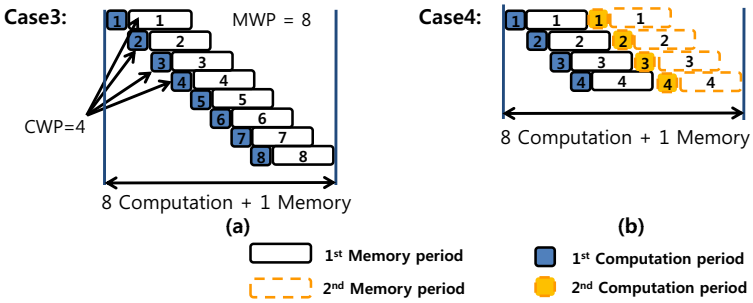

Figure 6. Total execution time when MWP is greater than CWP: (a) 8 warps (b) 4 warps

Even if not all warps are independent, when CWP is higher than MWP, many of memory waiting periods

are overlapped. Case 4 in Figure 6b shows an example. Each warp has two computation periods and two memory waiting periods. Since the computation time is dominant, the total execution cycles are again the sum of 8 computation periods and only one memory waiting period.

Using MWP and CWP, the total execution cycles can be calculated using the following equation:

$$Exec\_cycles = Mem\_p + Comp\_cycles \times N \tag{3}$$

Mem_p: One memory waiting period (= Mem_L in Equation (12))

Case 5 in Figure 7 shows an extreme case. In this case, not even one computation period can be finished while one memory waiting period is completed. Hence, CWP is less than 2. Note that CWP is always greater 1. Even if MWP is 8, the application cannot take advantage of any memory warp parallelism. Hence, the total execution cycles are 8 computation periods plus one memory waiting period. Note that even this extreme case, the total execution cycles of Case 5 are the same as that of Case 4. Case 5 happens when $Comp\_cycles$ are longer than Mem_cycles.

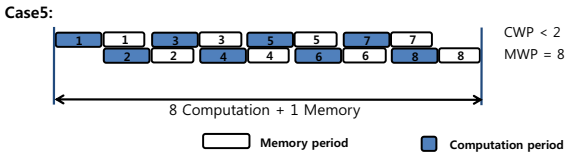

Figure 7. Total execution time when computation cycles are longer than memory waiting cycles. (8 warps)

3.2.3. Not Enough Warps Running The previous two sections described situations when there are enough number of warps running on one SM. Unfortunately, if an application does not have enough number of warps, the system cannot take advantage of all available warp parallelism. MWP and CWP cannot be greater than the number of active warps on one SM.

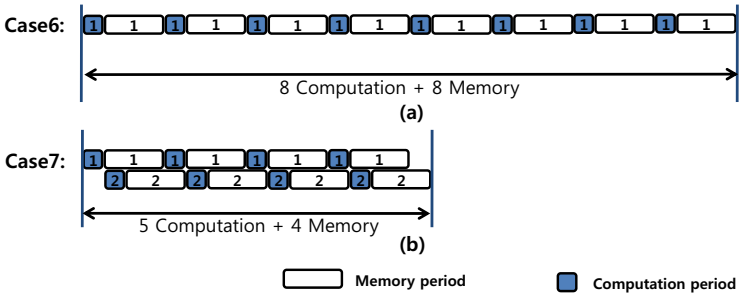

Figure 8. Total execution time when MWP is equal to N: (a) 1 warp (b) 2 warps

Case 6 in Figure 8a shows when only one warp is running. All the executions are serialized. Hence, the total execution cycles are the sum of the computation and memory waiting periods. Both CWP and MWP are 1 in this case. Case 7 in Figure 8b shows there are two running warps. Let's assume that MWP is two. Even if one computation period is less than the half of one memory waiting period, because there are only two warps, CWP is still two. Because of MWP, the total execution time is roughly the half of the sum of all the computation periods and memory waiting periods.

Using MWP, the total execution cycles of the above two cases can be calculated using the following equation:

$$\begin{aligned} Exec\_cycles =& Mem\_cycles \times N/MWP + Comp\_cycles \times \\ & N/MWP + Comp\_p(MWP - 1) \\ =& Mem\_cycles + Comp\_cycles + Comp\_p(MWP - 1) \end{aligned} \tag{4}$$

Note that for both cases, MWP and CWP are equal to N, the number of active warps per SM.

3.3. Calculating the Degree of Warp Parallelism

3.3.1. Memory Warp Parallelism (MWP) MWP is slightly different from MLP [13]. MLP represents how many memory requests can be serviced together. MWP represents the maximum number of *warps* in each SM that can access the memory simultaneously during one memory warp waiting period. The main difference between MLP and MWP is that MWP is counting all memory requests from a warp as one unit, while MLP counts all individual memory requests separately. As we discussed in Section 2.3, one memory instruction in a warp can generate multiple memory transactions. This difference is very important because a warp cannot be executed until all values are ready.

MWP is tightly coupled with the DRAM memory system. In our analytical model, we model the DRAM system as a simple queue and each SM has its own queue. Each active SM consumes an equal amount of memory bandwidth. Figure 9 shows the memory model and a timeline of memory warps.

The latency of each memory warp is at least Mem_L cycles. $Departure\_delay$ is the minimum departure distance between two consecutive memory warps. Mem_L is a round trip time to the DRAM, which includes the DRAM access time and the address and data transfer time.

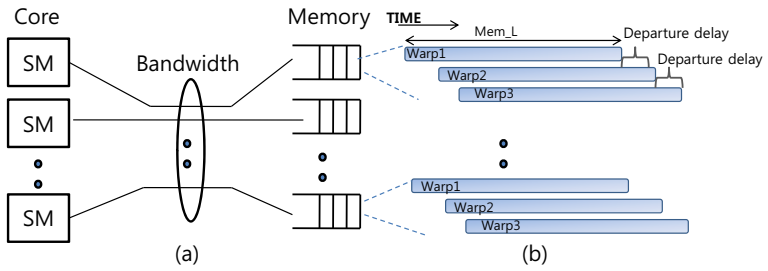

Figure 9. Memory system model: (a) memory model (b) timeline of memory warps

MWP represents the number of memory warps per SM that can be handled during Mem_L cycles. MWP cannot be greater than the number of warps per SM that reach the peak memory bandwidth (MWP_peak_BW) of the system as shown in Equation (5). If fewer SMs are executing warps, each SM can consume more bandwidth than when all SMs are executing warps. Equation (6) represents MWP_peak_BW. If an application does not reach the peak bandwidth, MWP is a function of Mem_L and $departure\_delay$. $MWP\_Without\_BW$ is calculated using Equations (10) – (17). MWP cannot be also greater than the number of active warps as shown in Equation (5). If the number of active warps is less than $MWP\_Without\_BW\_full$, the processor does not have enough number of warps to utilize memory level parallelism.

$$MWP = MIN(MWP\_Without\_BW, MWP\_peak\_BW, N) \tag{5}$$

$$MWP\_peak\_BW = \frac{Mem\_Bandwidth}{BW\_per\_warp \times \#ActiveSM} \tag{6}$$

$$BW\_per\_warp = \frac{Freq \times Load\_bytes\_per\_warp}{Mem\_L} \tag{7}$$

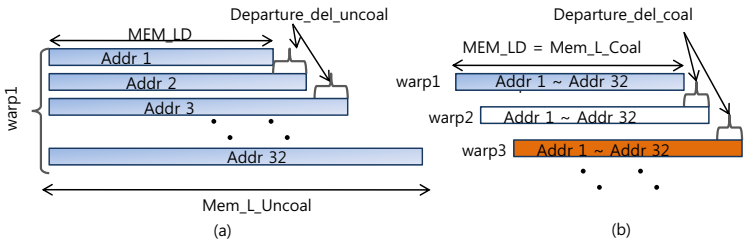

Figure 10. Illustrations of departure delays for uncoalesced and coalesced memory warps: (a) uncoalesced case (b) coalesced case

The latency of memory warps is dependent on memory access pattern (coalesced/uncoalesced) as shown in Figure 10. For uncoalesced memory warps, since one warp requests multiple number of transactions ($\#Uncoal\_per\_mw$), Mem_L includes departure delays for all $\#Uncoal\_per\_mw$ number of transactions. $Departure\_delay$ also includes $\#Uncoal\_per\_mw$ number of $Departure\_del\_uncoal$ cycles. Mem_LD is a round-trip latency to the DRAM for each memory transaction. In this model, Mem_LD for uncoalesced and coalesced are considered as the same, even though a coalesced memory request might take a few more cycles because

of large data size.

In an application, some memory requests would be coalesced and some would be not. Since multiple warps are running concurrently, the analytical model simply uses the weighted average of memory latency of coalesced and uncoalesced latency for the memory latency (Mem_L). A weight is determined by the number of coalesced and uncoalesced memory requests as shown in Equations (13) and (14). MWP is calculated using Equations (10) – (17). The parameters used in these equations are summarized in Table 1. Mem_LD, $Departure\_del\_coal$ and $Departure\_del\_uncoal$ are measured with micro-benchmarks as we will show in Section 5.1.

3.3.2. Computation Warp Parallelism (CWP) Once we calculate the memory latency for each warp, calculating CWP is straightforward. CWP_full is when there are enough number of warps. When CWP_full is greater than N (the number of active warps in one SM) CWP is N, otherwise, CWP_full becomes CWP.

$$CWP\_full = \frac{Mem\_cycles + Comp\_cycles}{Comp\_cycles} \tag{8}$$

$$CWP = MIN(CWP\_full, N) \tag{9}$$

3.4. Putting It All Together in CUDA

So far, we have explained our analytical model without strongly being coupled with the CUDA programming model to simplify the model. In this section, we extend the analytical model to consider the CUDA programming model.

3.4.1. Number of Warps per SM The modeled GPU architecture executes 100s of threads concurrently. Nonetheless, not all threads in an application can be executed at the same time. The processor fetches a few blocks at one time and then it fetches additional blocks as soon as one block retires. $\#Rep$ represents how many times a single SM executes multiple active number of blocks. For example, when there are 40 blocks in an application and 4 SMs. If each SM can execute 2 blocks concurrently, $\#Rep$ is 5. Hence, the total number of warps per SM is $\#Active\_warps\_per\_SM$ (N) times $\#Rep$. N is determined by machine resources.

3.4.2. Total Execution Cycles Depending on MWP and CWP values, total execution cycles for an entire application ($Exec\_cycles\_app$) are calculated using Equations (22),(23), and (24). Mem_L is calculated in

Equation (12). Execution cycles that consider synchronization effects will be described in Section 3.4.6.

$$Mem\_L\_Uncoal = Mem\_LD + (\#Uncoal\_per\_mw - 1) \times Departure\_del\_uncoal \quad (10)$$

$$Mem\_L\_Coal = Mem\_LD \quad (11)$$

$$Mem\_L = Mem\_L\_Uncoal \times Weight\_uncoal + Mem\_L\_Coal \times Weight\_coal \quad (12)$$

$$Weight\_uncoal = \frac{\#Uncoal\_Mem\_insts}{(\#Uncoal\_Mem\_insts + \#Coal\_Mem\_insts)} \quad (13)$$

$$Weight\_coal = \frac{\#Coal\_Mem\_insts}{(\#Coal\_Mem\_insts + \#Uncoal\_Mem\_insts)} \quad (14)$$

$$Departure\_delay = (Departure\_del\_uncoal \times \#Uncoal\_per\_mw) \times Weight\_uncoal + Departure\_del\_coal \times Weight\_coal \quad (15)$$

$$MWP\_Without\_BW\_full = Mem\_L / Departure\_delay \quad (16)$$

$$MWP\_Without\_BW = MIN(MWP\_Without\_BW\_full, \#Active\_warps\_per\_SM) \quad (17)$$

$$Mem\_cycles = Mem\_L\_Uncoal \times \#Uncoal\_Mem\_insts + Mem\_L\_Coal \times \#Coal\_Mem\_insts \quad (18)$$

$$Comp\_cycles = \#Issue\_cycles \times (\#total\_insts) \quad (19)$$

$$N = \#Active\_warps\_per\_SM \quad (20)$$

$$\#Rep = \frac{\#Blocks}{\#Active\_blocks\_per\_SM \times \#Active\_SMs} \quad (21)$$

If (MWP is N warps per SM) and (CWP is N warps per SM)

$$Exec\_cycles\_app = (Mem\_cycles + Comp\_cycles + \frac{Comp\_cycles}{\#Mem\_insts} \times (MWP - 1)) \times \#Rep \quad (22)$$

Else if (CWP $>=$ MWP) or (Comp_cycles $>$ Mem_cycles)

$$Exec\_cycles\_app = (Mem\_cycles \times \frac{N}{MWP} + \frac{Comp\_cycles}{\#Mem\_insts} \times (MWP - 1)) \times \#Rep \quad (23)$$

Else

$$Exec\_cycles\_app = (Mem\_L + Comp\_cycles \times N) \times \#Rep \quad (24)$$

*All the parameters are summarized in Table 1.

3.4.3. Dynamic Number of Instructions Total execution cycles are calculated using the number of dynamic instructions. The compiler generates intermediate assembler-level instruction, the NVIDIA PTX instruction set [30]. PTX instructions translate nearly one to one with native binary microinstructions later.[7] We use the number of PTX instructions for the dynamic number of instructions.

The total number of instructions is proportional to the number of data elements. Programmers must decide the number of threads and blocks for each input data. The number of total instructions per thread is related to how many data elements are computed in one thread, programmers must know this information.

[7]Since some PTX instructions expand to multiple binary instructions, using PTX instruction count could be one of the error sources in the analytical model.

Table 1. Summary of Model Parameters

	Model Parameter	**Definition**	**Obtained**
1	#Threads_per_warp	Number of threads per warp	32 [30]
2	Issue_cycles	Number of cycles to execute one instruction	4 cycles [17]
3	Freq	Clock frequency of the SM processor	Table 3
4	Mem_Bandwidth	Bandwidth between the DRAM and GPU cores	Table 3
5	Mem_LD	DRAM access latency (machine configuration)	Table 6
6	Departure_del_uncoal	Delay between two uncoalesced memory transactions	Table 6
7	Departure_del_coal	Delay between two coalesced memory transactions	Table 6
8	#Threads_per_block	Number of threads per block	Programmer specifies inside a program
9	#Blocks	Total number of blocks in a program	Programmer specifies inside a program
10	#Active_SMs	Number of active SMs	Calculated based on machine resources
11	#Active_blocks_per_SM	Number of concurrently running blocks on one SM	Calculated based on machine resources [30]
12	#Active_warps_per_SM (N)	Number of concurrently running warps on one SM	Active_blocks_per_SM x Number of warps per block
13	#Total_insts	(#Comp_insts + #Mem_insts)	
14	#Comp_insts	Total dynamic number of computation instructions in one thread	Source code analysis
15	#Mem_insts	Total dynamic number of memory instructions in one thread	Source code analysis
16	#Uncoal_Mem_insts	Number of uncoalesced memory type instructions in one thread	Source code analysis
17	#Coal_Mem_insts	Number of coalesced memory type instructions in one thread	Source code analysis
18	#Synch_insts	Total dynamic number of synchronization instructions in one thread	Source code analysis
19	#Coal_per_mw	Number of memory transactions per warp (coalesced access)	1
20	#Uncoal_per_mw	Number of memory transactions per warp (uncoalesced access)	Source code analysis[16](Table 3)
21	Load_bytes_per_warp	Number of bytes for each warp	Data size (typically 4B) x #Threads_per_warp

If we know the number of elements per thread, counting the number of total instructions per thread is simply counting the number of computation instructions and the number of memory instructions per data element. The detailed algorithm to count the number of instructions from PTX code is provided in an extended version of this paper [16].

3.4.4. Cycles Per Instruction (CPI) Cycles per Instruction (CPI) is commonly used to represent the cost of each instruction. Using total execution cycles, we can calculate Cycles Per Instruction using Equation (25). Note that, CPI is the cost when an instruction is executed by all threads in one warp.

$$CPI = \frac{Exec\_cycles\_app}{\#Total\_insts \times \frac{\#Threads\_per\_block}{\#Threads\_per\_warp} \times \frac{\#Blocks}{\#Active\_SMs}} \tag{25}$$

3.4.5. Coalesced/Uncoalesced Memory Accesses As Equations (15) and (12) show, the latency of memory instruction is heavily dependent on memory access type. Whether memory requests inside a warp can be coalesced or not is dependent on the memory system design and memory access pattern in a warp. The GPUs that we evaluated have two coalesced/uncoalesced polices, specified by the Compute capability version. The CUDA manual [30] describes when memory requests in a warp can be coalesced or not in more detail. Earlier compute capability versions have two differences compared with the later version(1.3): (1) stricter rules are applied to be coalesced, (2) when memory requests are uncoalesced, one warp generates

32 memory transactions. In the latest version (1.3), the rules are more relaxed and all memory requests are coalesced into as few memory transactions as possible.[8]

The detailed algorithms to detect coalesced/uncoalesced memory accesses and to count the number of memory transactions per each warp at static time are provided in an extended version of this paper [16].

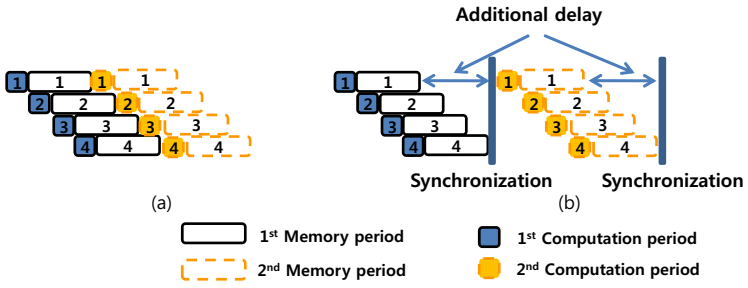

Figure 11. Additional delay effects of thread synchronization: (a) no synchronization (b) thread synchronization after each memory access period

3.4.6. Synchronization Effects The CUDA programming model supports thread synchronization through the `__syncthreads()` function. Typically, all the threads are executed asynchronously whenever all the source operands in a warp are ready. However, if there is a barrier, the processor cannot execute the instructions after the barrier until all the threads reach the barrier. Hence, there will be additional delays due to a thread synchronization. Figure 11 illustrates the additional delay effect. Surprisingly, the additional delay is less than one waiting period. Actually, the additional delay per synchronization instruction is the multiple of $Departure\_delay$, the number of blocks and (NpWB-1). NpWB, which is newly introduced in this equation, is the number of parallel warps per block. NpWB is used instead of MWP since warps inside a block are synchronized. The final execution cycles of an application with synchronization delay effect

[8] In the CUDA manual, compute capability 1.3 says all requests are coalesced because all memory requests within each warp are always combined into as few transactions as possible. However, in our analytical model, we use the coalesced memory access model only if all memory requests are combined into one memory transaction.

can be calculated by Equation (28).

$$Synch\_cost = Departure\_delay \times (NpWB - 1) \times \#synch\_insts$$

$$\times \#Active\_blocks\_per\_SM \times \#Rep \tag{26}$$

$$NpWB = MIN(MWP, \#Active\_warps\_per\_block) \tag{27}$$

$$Exec\_cycles\_with\_synch = Exec\_cycles\_app + Synch\_cost \tag{28}$$

3.5. Code Example

To provide a concrete example, we apply the analytical model for a tiled matrix multiplication example in Figure 12 to a system that has 80GB/s memory bandwidth, 1GHz frequency and 16 SM processors. Let's assume that the programmer specified 128 threads per block (4 warps per block), and 80 blocks for execution. And 5 blocks are actively assigned to each SM ($Active\_blocks\_per\_SM$) instead of 8 maximum blocks[9] due to high resource usage.

```
1: MatrixMulKernel<<<80, 128>>> (M, N, P);
2: ....
3: MatrixMulKernel(Matrix M, Matrix N, Matrix P)
4: {
5:   // init code ...
6:
7:  for (int a=starta, b=startb, iter=0; a<=enda;
8:           a+=stepa, b+=stepb, iter++)
9:  {
10:    __shared__ float Msub[BLOCKSIZE][BLOCKSIZE];
11:    __shared__ float Nsub[BLOCKSIZE][BLOCKSIZE];
12:
13:    Msub[ty][tx] = M.elements[a + wM * ty + tx];
14:    Nsub[ty][tx] = N.elements[b + wN * ty + tx];
15:
16:    __syncthreads();
17:
18:    for (int k=0; k < BLOCKSIZE; ++k)
19:      subsum += Msub[ty][k] * Nsub[k][tx];
20:
21:    __syncthreads();
22:  }
23:
24:  int index = wN * BLOCKSIZE * by + BLOCKSIZE
25:  P.elements[index + wN * ty + tx] = subsum;
26:}
```

Figure 12. CUDA code of tiled matrix multiplication

We assume that the inner loop is iterated only once and the outer loop is iterated 3 times to simplify the example. Hence, $\#Comp\_insts$ is 27, which is 9 computation (Figure 13 lines 5, 7, 8, 9, 10, 11, 13, 14, and 15) instructions times 3. Note that `ld.shared` instructions in Figure 13 lines 9 and 10 are also counted

[9]Each SM can have up to 8 blocks at a given time.

```
1: ...                              // Init Code
2:
3:  $OUTERLOOP:
4:  ld.global.f32  %f2, [%rd23+0];  //
5:  st.shared.f32  [%rd14+0], %f2;  //
6:  ld.global.f32  %f3, [%rd19+0];  //
7:  st.shared.f32  [%rd15+0], %f3;  //
8:  bar.sync   0;                   // Synchronization
9:  ld.shared.f32  %f4, [%rd8+0];   // Innerloop unrolling
10: ld.shared.f32  %f5, [%rd6+0];   //
11: mad.f32   %f1, %f4, %f5, %f1;   //
12: // the code of unrolled loop is omitted
13: bar.sync   0;                   // synchronization
14: setp.le.s32  %p2, %r21, %r24;   //
15: @%p2 bra  $OUTERLOOP;           // Branch
16: ...                             // Index calculation
17: st.global.f32 [%rd27+0], %f1;   // Store in P.elements
```

Figure 13. PTX code of tiled matrix multiplication

as a computation instruction since the latency of accessing the shared memory is almost as fast as that of the register file. Lines 13 and 14 in Figure 12 show global memory accesses in the CUDA code. Memory indexes (`a+wM*ty+tx`) and (`b+wN*ty+tx`) determine memory access coalescing within a warp. Since `a` and `b` are more likely not a multiple of 32, we treat that all the global loads are uncoalesced [16]. So $\#Uncoal\_Mem\_insts$ is 6, and $\#Coal\_Mem\_insts$ is 0.

Table 2 shows the necessary model parameters and intermediate calculation processes to calculate the total execution cycles of the program. Since CWP is greater than MWP, we use Equation (23) to calculate $Exec\_cycles\_app$. Note that in this example, the execution cost of synchronization instructions is a significant part of the total execution cost. This is because we simplified the example. In most real applications, the number of dynamic synchronization instructions is much less than other instructions, so the synchronization cost is not that significant.

4. Experimental Methodology

4.1. The GPU Characteristics

Table 3 shows the list of GPUs used in this study. GTX280 supports 64-bit floating point operations and also has a later computing version (1.3) that improves uncoalesced memory accesses. To measure the GPU kernel execution time, `cudaEventRecord` API that uses GPU Shader clock cycles is used. All the measured execution time is the average of 10 runs.

Table 2. Applying the Model to Figure 12

Model Parameter	Obtained	Value
Mem_LD	Machine conf.	420
Departure_del_uncoal	Machine conf.	10
#Threads_per_block	Figure 12 Line 1	128
#Blocks	Figure 12 Line 1	80
#Active_blocks_per_SM	Occupancy [30]	5
#Active_SMs	Occupancy [30]	16
#Active_warps_per_SM	$128/32(Table\ 1) \times 5$	20
#Comp_insts	Figure 13	27
#Uncoal_Mem_insts	Figure 12 Lines 13, 14	6
#Coal_Mem_insts	Figure 12 Lines 13, 14	0
#Synch_insts	Figure 12 Lines 16, 21	6 = 2×3
#Coal_per_mw	see Sec. 3.4.5	1
#Uncoal_per_mw	see Sec. 3.4.5	32
Load_bytes_per_warp	Figure 13 Lines 4, 6	128B = $4B \times 32$
Departure_delay	Equation (15)	320=32×10
Mem_L	Equations (10), (12)	$730=420 + (32-1) \times 10$
MWP_without_BW_full	Equation (16)	2.28 =$730/320$
BW_per_warp	Equation (7)	0.175GB/S = $\frac{1G \times 128B}{730}$
MWP_peak_BW	Equation (6)	28.57=$\frac{80GB/s}{0.175GB \times 16}$
MWP	Equation (5)	2.28=MIN(2.28, 28.57, 20)
Comp_cycles	Equation (19)	132 cycles= $4 \times (27+6)$
Mem_cycles	Equation (18)	$4380 = (730 \times 6)$
CWP_full	Equation (8)	34.18=$(4380 + 132)/132$
CWP	Equation (9)	20 = MIN(34.18, 20)
#Rep	Equation (21)	1 = $80/(16 \times 5)$
Exec_cycles_app	Equation (23)	$38450 = 4380 \times \frac{20}{2.28} + \frac{132}{6} \times (2.28-1)$
Synch_cost	Equation (26)	12288= $320 \times (2.28-1) \times 6 \times 5$
Final Time	Equation (28)	50738 =$38450 + 12288$

4.2. Micro-benchmarks

All the benchmarks are compiled with NVCC [30]. To test the analytical model and also to find memory model parameters, we design a set of micro-benchmarks that simply repeat a loop for 1000 times. We vary the number of load instructions and computation instructions per loop. Each micro-benchmark has two memory access patterns: coalesced and uncoalesced memory accesses.

Table 3. The specifications of GPUs used in this study

Model	8800GTX	Quadro FX5600	8800GT	GTX280
#SM	16	16	14	30
(SP) Processor Cores	128	128	112	240
Graphics Clock	575 MHz	600 MHz	600 MHz	602 MHz
Processor Clock	1.35 GHz	1.35GHz	1.5 GHz	1.3 GHz
Memory Size	768 MB	1.5 GB	512 MB	1 GB
Memory Bandwidth	86.4 GB/s	76.8 GB/s	57.6 GB/s	141.7 GB/s
Peak Gflop/s	345.6	384	336	933
Computing Version	1.0	1.0	1.1	1.3
#Uncoal_per_mw	32	32	32	[16]
#Coal_per_mw	1	1	1	1

Table 4. The characteristics of micro-benchmarks

# inst. per loop	Mb1	Mb2	Mb3	Mb4	Mb5	Mb6	Mb7
Memory	0	1	1	2	2	4	6
Comp. (FP)	23 (20)	17 (8)	29 (20)	27(12)	35(20)	47(20)	59(20)

Table 5. Characteristics of the Merge Benchmarks (Arith. intensity means arithmetic intensity.)

Benchmark	Description	Input size	Comp insts	Mem insts	Arith. intensity	Registers	Shared memory
Sepia [23]	Filter for artificially aging images	7000 x 7000	71	6 (uncoal)	11.8	7	52B
Linear [23]	Image filter for computing 9-pixels avg.	10000 x 10000	111	30 (uncoal)	3.7	15	60B
SVM [23]	Kernel from a SVM-based algorithm	736 x 992	10871	819 (coal)	13.3	9	44B
Mat. (naive)	Naive version of matrix multiplication	2000 x 2000	12043	4001(uncoal)	3	10	88B
Mat. (tiled) [30]	Tiled version of matrix multiplication	2000 x 2000	9780 - 24580	201 - 1001(uncoal)	48.7	18	3960B
Blackscholes [30]	European option pricing	9000000	137	7 (uncoal)	19	11	36B

4.3. Merge Benchmarks

To test how our analytical model can predict typical GPGPU applications, we use 6 different benchmarks that are mostly used in the Merge work [23]. Table 5 explains the description of each benchmark and summarizes the characteristics of each benchmark. The number of registers used per thread and shared memory usage per block are statically obtained by compiling the code with *-cubin* flag. The rest of the characteristics are statically determined and can be found in PTX code.

5. Results

5.1. Micro-benchmarks

The micro-benchmarks are used to measure the constant variables that are required to model the memory system. We vary three parameters (Mem_LD, $Departure\_del\_uncoal$, and $Departure\_del\_coal$) for each GPU to find the best fitting values. FX5600, 8800GTX and 8800GT use the same model parameters. Table 6 summarizes the results. $Departure\_del\_coal$ is related to the memory access time to a single memory block. $Departure\_del\_uncoal$ is longer than $Departure\_del\_coal$, due to the overhead of 32 small memory access requests. $Departure\_del\_uncoal$ for GTX280 is much longer than that of FX5600. GTX280 coalesces 32 thread memory requests per warp into the minimum number of memory access requests, and the overhead per access request is higher, with fewer accesses.

Using the parameters in Table 6, we calculate CPI for the micro-benchmarks. Figure 14 shows the

Table 6. Results of the Memory Model Parameters

Model	FX5600	GTX280
Mem_LD	420	450
Departure_del_uncoal	10	40
Departure_del_coal	4	4

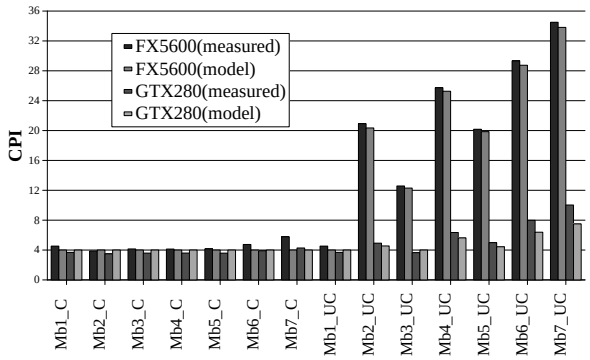

Figure 14. CPI on the micro-benchmarks

average CPI of the micro-benchmarks for both measured value and estimated value using the analytical model. The results show that the average geometric mean of the error is 5.4%. As we can predict, as the benchmark has more number of load instructions, the CPI increases. For the coalesced load cases (Mb1_C – Mb7_C), the cost of load instructions is almost hidden because of high MWP but for uncoalesced load cases (Mb1_UC – Mb7_UC), the cost of load instructions linearly increases as the number of load instructions increases.

5.2. Merge Benchmarks

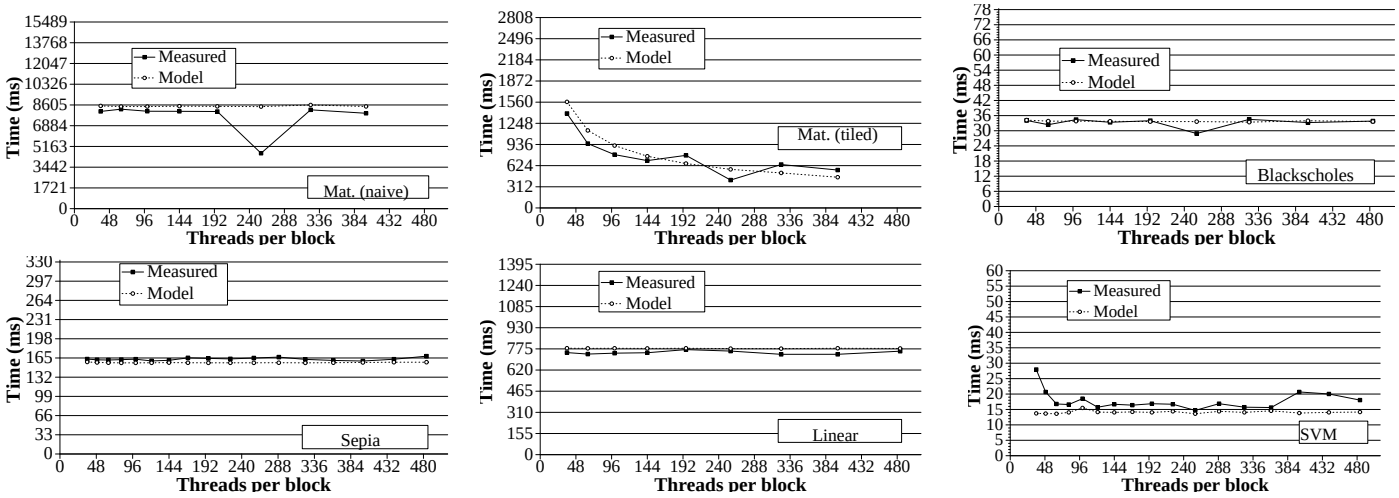

Figure 15. The total execution time of the Merge benchmarks on FX5600

Figure 15 and Figure 16 show the measured and estimated execution time of the Merge benchmarks on FX5600 and GTX280. The number of threads per block is varied from 4 to 512, (512 is the maximum value that one block can have in the evaluated CUDA programs.) Even though the number of threads is varied, the programs calculate the same amount of data elements. In other words, if we increase the number of threads in a block, the total number of blocks is also reduced to process the same amount of data in one

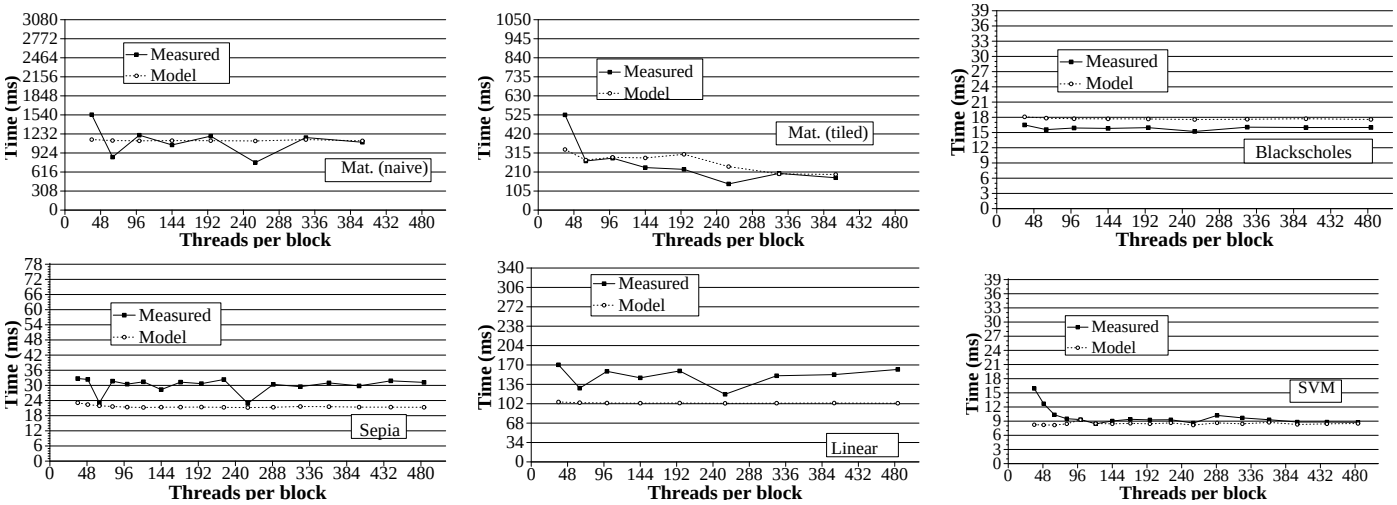

Figure 16. The total execution time of the Merge benchmarks on GTX280

application. That is why the execution times are mostly the same. For the Mat.(tiled) benchmark, as we increase the number of threads the execution time reduces, because the number of active warps per SM increases.

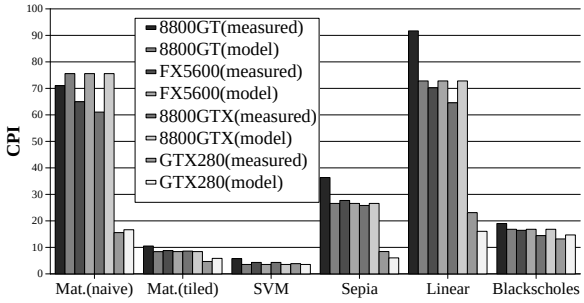

Figure 17. CPI on the Merge benchmarks

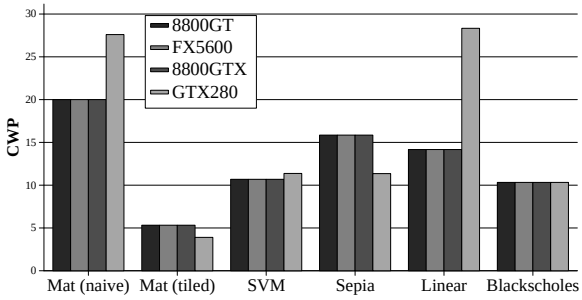

Figure 18. CWP per SM on the Merge benchmarks

Figure 17 shows the average of the measured and estimated CPIs in Figures 15 and 16 configurations. The average values of CWP and MWP per SM are also shown in Figures 18 and 19 respectively. 8800GT has the least amount of bandwidth compared to other GPUs, resulting in the highest CPI in contrast to GTX280. Generally, higher arithmetic intensity means lower CPI (lower CPI is higher performance). However, even though the Mat.(tiled) benchmark has the highest arithmetic intensity, SVM has the lowest CPI value. SVM has higher MWP and CWP than those of Mat.(tiled) as shown in Figures 18 and 19. SVM has the highest MWP and the lowest CPI because only SVM has fully coalesced memory accesses. MWP in GTX280 is higher than the rest of GPUs because even though most memory requests are not fully coalesced, they are still combined into as few requests as possible, which results in higher MWP. All other benchmarks are limited by $departure\_delay$, which makes all other applications never reach the peak

memory bandwidth.

Figure 20 shows the average occupancy of the Merge benchmarks. Except Mat.(tiled) and Linear, all other benchmarks have higher occupancy than 70%. The results show that occupancy is less correlated to the performance of applications.

The final geometric mean of the estimated CPI error on the Merge benchmarks in Figure 17 over all four different types of GPUs is 13.3%. Generally the error is higher for GTX 280 than others, because we have to estimate the number of memory requests that are generated by partially coalesced loads per warp in GTX280, unlike other GPUs which have the fixed value 32. On average, the model estimates the execution cycles of FX5600 better than others. This is because we set the machine parameters using FX5600.

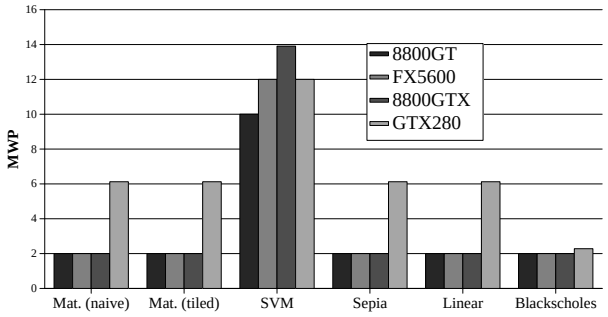

Figure 19. MWP per SM on the Merge benchmarks

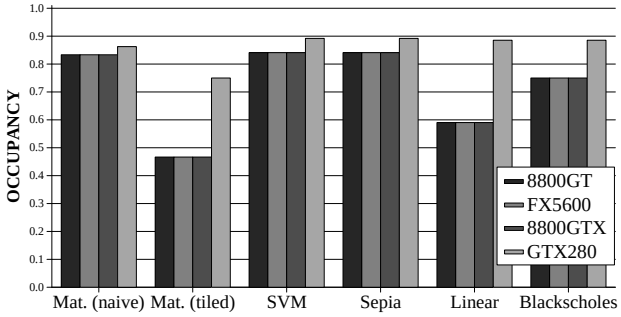

Figure 20. Occupancy on the Merge benchmarks

There are several error sources in our model: (1) We used a very simple memory model and we assume that the characteristics of the memory behavior are similar across all the benchmarks. We found out that the outcome of the model is very sensitive to MWP values. (2) We assume that the DRAM memory scheduler schedules memory requests equally for all warps. (3) We do not consider the bank conflict latency in the shared memory. (4) All computation instructions have the same latency even though some special functional unit instructions have longer latency than others. (5) For some applications, the number of threads per block is not always a multiple of 32. (6) The SM retires warps as a block granularity. Even though there are free cycles, the SM cannot start to fetch new blocks, but the model assumes on average active warps.

5.3. Insights Into The Model

The MWP value is limited by three factors: memory level parallelism inside an application, DRAM throughput, and bandwidth between SMs and GPU DRAM. The throughput is dependent on DRAM configuration and the ratio of memory access types (between coalesced and uncoalesced accesses). To visualize how MWP is affected by three components, we vary the number of warps and plot the corresponding MWP values in Figure 21.

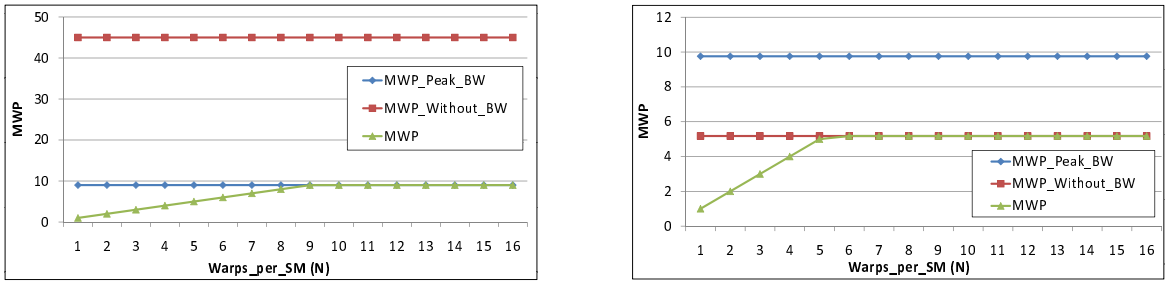

Figure 21. Visualization of MWP for GTX280 (Left:coalesced case, Right:uncoalesced case)

The results show that, uncoalesced memory accesses can never saturate available memory bandwidth. Increasing the number of warps (through different parallelization techniques or changing the occupancy) increases MWP until 9 for coalesced case but only 5 for uncoalesced case.

Now, to provide insights into the analytical model, we revisit the example in Section 2.4. Figures 22 and 23 show N, MWP_without_BW, MWP_peak_BW, MWP, and CWP for `Constant+Optimized` case and `Naive` case from Figure 3 respectively. Here, we explain the performance behavior with MWP_peak_BW and MWP_Without_BW instead of MWP, because final MWP is the minimum of those two terms and the number of running warps (N) as shown in Equation (5). Limiting MWP term for Figure 22 is 12 (MWP_peak_BW), and it is 2 (MWP_Without_BW) for Figure 23. The main reason of this difference is that `Constant+Optimized` has coalesced memory accesses, but `Naive` has uncoalesced memory accesses. Until N reaches MWP_peak_BW, which is 40, increasing N reduces execution time for `Constant+Optimized` since more warps can increase memory level parallelism. However, in `Naive`, N is always greater than MWP_without_BW, so increasing N does not improve performance since maximum memory level parallelism is already reached.

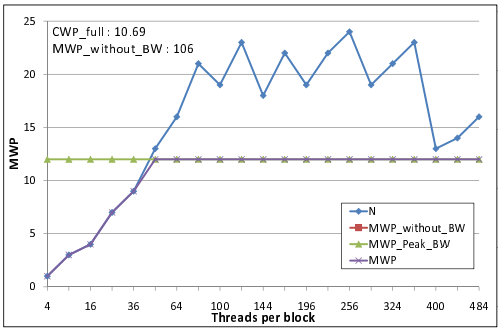

Figure 22. MWP, CWP Analysis on the Optimized SVM

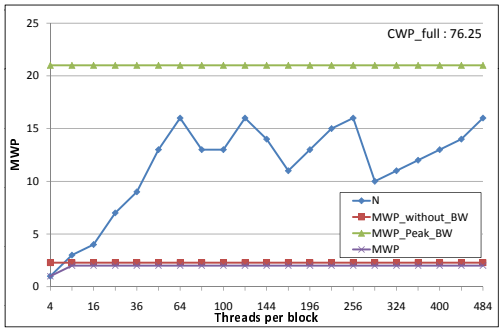

Figure 23. MWP, CWP Analysis on the Naive SVM

6. Improving the Model and More validations

In this section, we improve the analytical model by considering more complex cases: independent memory operations and long-latency computations. We also validate divergent warps and synchronization effects in more detail.

6.1. Effects of Dependent/Independent Memory Accesses

The Tesla architecture is an in-order processor within a warp. It stops issuing an instruction from a warp if not all source operands are ready and switches to another ready warp. When a warp generates a global memory request, if the subsequent instructions do not source the outcome of the global load (i.e., the subsequent instructions are not dependent on the previous memory-requesting instruction), the instructions can be still issued as long as all the source operands are ready. Hence, global memory memory requests from the same warp could be serviced together if they (and including all the instructions between two global load instructions) are not dependent on the first load instruction. Figure 24 illustrates both cases (dependent instructions and independent instructions). The numbers inside the computation and memory periods indicate warp identification numbers. In the dependent-instruction case, all memory operations from the same warp is serialized, but in the independent-instruction case, memory operations from the same warp can be still serviced concurrently, thereby increasing effective memory level parallelism (MWP).

To evaluate the effect of dependent/independent memory accesses in actual performance, we design micro benchmarks, where one benchmark is dependent on the previous value of the memory load (DEP), but the other is not (INDEP). Both cases have the exact same number of instructions and instruction mixtures. Figure 26 shows the execution time of two cases as we increase the number of warps (i.e., all threads execute the same code, so the total amount of work is also increased.) When the number of warps

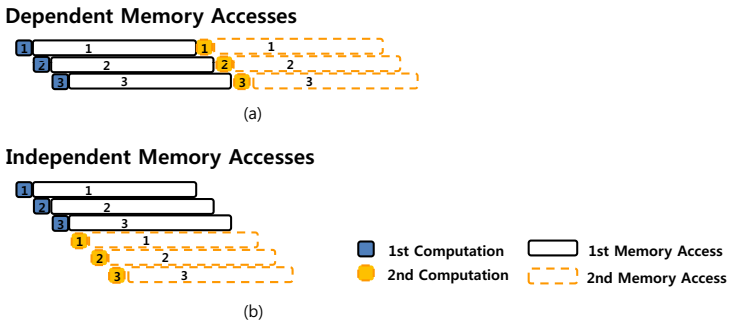

Figure 24. Illustration of dependent/independent memory accesses

per SM is less than MWP, the execution time of INDEP is much shorter than that of DEP. However, once N is greater than MWP, both benchmarks take similar execution time. The main reason is that when there are enough running warps than available memory-level parallelism, the processor can always find other ready warps for execution.

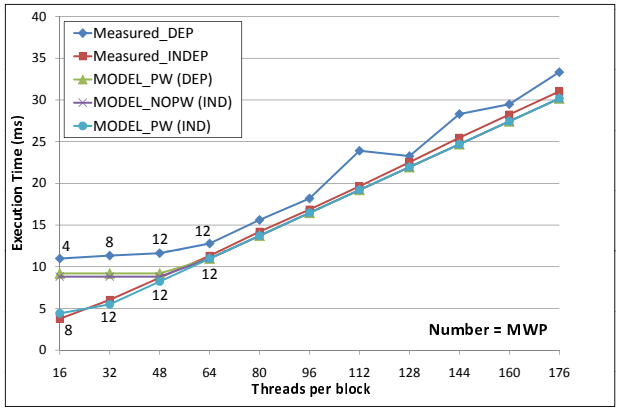

Figure 25. Model prediction of dependent/independent memory accesses

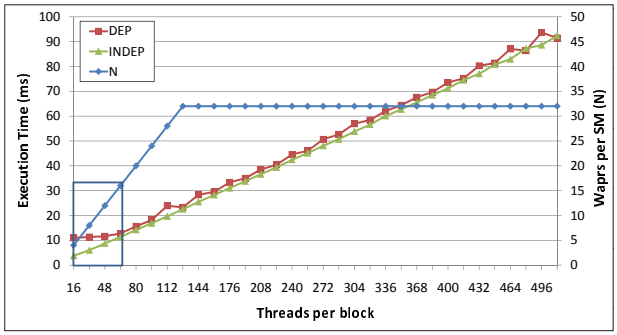

Figure 26. Effects of dependent/independent memory accesses

In our analytical model, we have assumed that all instructions within a warp is dependent on the previous instructions, which results in a serialization of all memory requests from one warp. We improve the analytical model by introducing N_{pw}, which is the number of parallel warps. Typically, the number of parallel warps is the same as the total number of warps (N). However, when there are independent memory requests, more number of warps (i.e., more number of memory requests) can be executed in parallel. We calculate this effective number of parallel warps by calculating the number of memory independent requests.

$$N_{pw} = N \times \frac{\#ind\_mem\_req}{(\#mem\_req - \#ind\_mem\_req)} \tag{29}$$

$\#ind\_mem\_req$: number of independent memory requests

$\#mem\_req$: number of total memory requests

$$MWP = MIN(MWP\_Without\_BW, MWP\_peak\_BW, N_{pw}) \tag{30}$$

This N_{pw} is only used for calculating MWP. As shown in Equation (30), N_{pw} can affect MWP, only when N is less than either $MWP\_Without\_BW$ or MWP_peak_BW, which explains the behavior in Figure 26. This is the same case when there are not enough running warps (Section 3.2.3 case). Hence, this scenario rarely occurs because multiple conditions need to be satisfied (not enough running warps, and independent memory requests). Most memory requests are dependent on the outcome of the previous memory request inside the same warp, otherwise, programmers could simply increase the number of warps (actually they should have increased) in application from the beginning especially when there are not enough running warps.

Figure 25 shows the outcome of two models and actual measured value for two different memory access cases. (1) independent memory accesses with the original model: MODEL_NOPW(IND), (2) independent memory accesses with the new model: MODEL_PW (IND), and (3) dependent memory accesses with new model: MODEL_PW (DEP).[10] Figure 25 zooms the boxed area in Figure 26. The experiment demonstrates two important behaviors when the number of threads is less than 48. First, for dependent memory accesses, the execution time is not increased linearly (almost the same). Second, the execution time of dependent memory accesses is much longer than that of the independent memory accesses. The reason the flat area exists is when the number of warps is too small, even if we increase the total work, the work takes almost the same amount of time because the execution time is dominated by memory operations. The additional memory requests due to additional warps are all serviced concurrently thereby total memory operations remain the same. The results show that the predicted execution time using (25)(MODEL_PW) estimates the execution time precisely for these two cases but not with the old model (MODEL_NOPW).

[10] dependent memory accesses with old model is the same as new model

6.2. Long Latency Computation Instructions

In our analytical model, we apply different instruction latencies based on the instruction types.

Table 7 summarizes the throughput of instructions based on the CUDA manual and our experimental measurement. Throughput of one means that each functional unit can finish one instruction at one time, which results in 8 Ops/s. Ops/s means the operations per second per SM. [11] M_Factor term is equal to one when the throughput is 8 Ops/s, and it is proportionately increased as the throughput is decreased.

Surprisingly, FP (floating-point) operations are faster than INT operations in GPU architecture. Throughput for FP operations such as addition, multiplication and multiply-addition is equal to the number of functional units (i.e., 8 SP processors in the Tesla architecture, 8 FP operations per cycle). But instructions such as modulo and integer multiplication take much longer latency, reducing the throughput by the factors of 4.3 and 35 respectively.

Table 7. Instruction Throughput

Instructions	Ops/s (M_Factor)[M]	M_Factor [Experiment]
FPadd FPmul FPmad	8 (1)	1
Intadd	8 (1)	1
FPdiv	2 (4)	4.2
Intmul	2 (4)	4.3
Intdiv, Modulo	Very Costly	30, 35

$$Comp\_cycles = (\#Issue\_cycles \times M\_Factor) \times \#total\_insts \tag{31}$$

Equation (31) shows the improved calculation for computation cycles over the previous Equation (19) by considering variable instruction latency.

6.3. Divergent branches

When a warp diverges (i.e., diverges within 32 threads), the execution of diverged warps is serialized [30]. [12] This means that while one path is executed, the threads on the other path are idle. Figure 27 shows an example. The branch at basic block 1 in the figure diverges. Active bitmap mask shows that first four threads take the taken path while the rest takes the not-taken path. Basic block 2 also has a divergent branch. Hence, there are three paths (B1B2B4B6B7, B1B2B5B6B7, B1B3B7) in this example.

[11] Ops/s is used in the CUDA manual. M_Factor is the term used to model longer-latency instructions.
[12] Several recent studies have focused on reducing unnecessary idle cycles during divergent execution [12, 37].

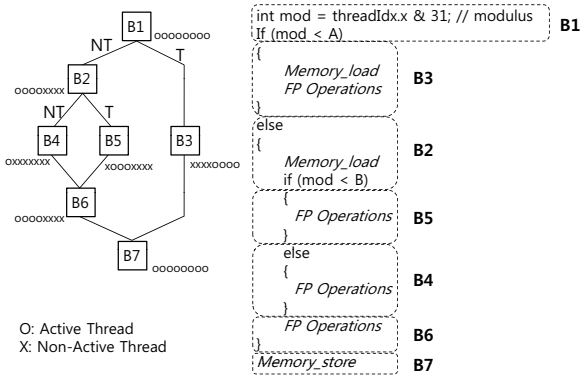

Figure 27. Illustration of a divergent execution

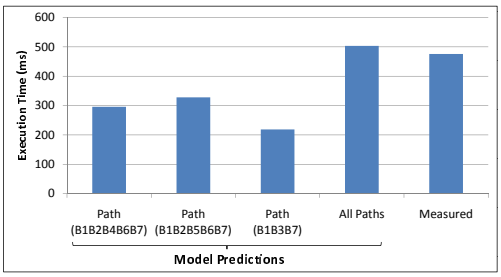

Figure 28. Effects of divergent branches on the execution time

Figure 28 shows the model predictions and the measured execution time. If the model only takes execution time of each individual path into account (the first three bars in the figure), the execution time is much shorter than the actual execution time. In the current GPU architecture, all the divergent paths are serially executed [12]. *All paths* bar in the figure is the sum of all the paths in the divergent branch, which shows only 6% delta with the actual measured time.

6.4. Effects of Synchronization

The cost of synchronization is modeled in Section 3.4.6 using equations (26) and (28). To evaluate the synchronization cost in more detail, we compare the performance delta between two programs in Figure 29 where the only difference is the barrier instruction (bar.sync).

```
Program A (Synchronization)
...
9:  ld.global.f32   %f1, [%r8+0];
10: mov.f32      %f2, 0f41200000;
11: mul.f32      %f3, %f1, %f2;
12: bar.sync      0;                 //Synchronization
13: st.global.f32   [%r8+0], %f3;

Program B (No Synchronization)
...
9:  ld.global.f32   %f1, [%r8+0];
10: mov.f32      %f2, 0f41200000;
11: mul.f32      %f3, %f1, %f2;
12: st.global.f32   [%r8+0], %f3;
```

Figure 29. PTX code for synchronization analysis

Figure 30 shows an experiment where only one SM is active (i.e., one block is used). When there is only one warp, there should no performance penalty due to synchronization. However, in the measured data, we still observe some minor penalties from the `bar.sync` instruction. We estimate that this overhead is coming from the fetch unit or other schedulers. Please note that using bar.synch just for one warp is

not a typical case which might cause unexpected overhead. Programmers should not use bar.sync just for only one warp. Having predicted, as we increase the number of threads (warps) in the core, the cost of synchronization increases. The model predicts the increasing cost accurately but with the absolute delta due to the initial cost difference. In this experiment, we intentionally use only one SM to observe the cost.

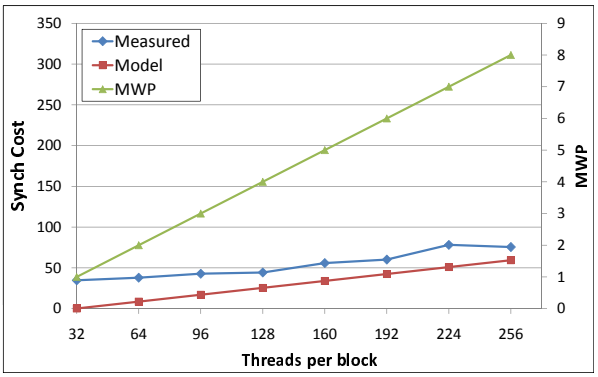

Figure 30. synchronization delay: 1 block (1 SM active)

Figures 31 shows the performance delta when all SMs are actively running multiple blocks. Resource usage for each GPU kernel is manually controlled to allocate two blocks per SM for *BL2* and four blocks per SM for *BL4*. In this experiment, we observe both the effect of number of blocks and MWP. Increasing the number of blocks also increases the cost of synchronization, because memory requests are delayed by intervention with warps in other blocks. Since the number of warps is still less than $MpWB$, the synchronization cost is increased continuously. The model predictions show that a high-level trend for synchronization is modeled. The geometric error for BL2 is 19.65% and 11.42% for BL4. As previously mentioned in Section 3.4.6, with respect to overall performance, synchronization delay cycles are not significant.

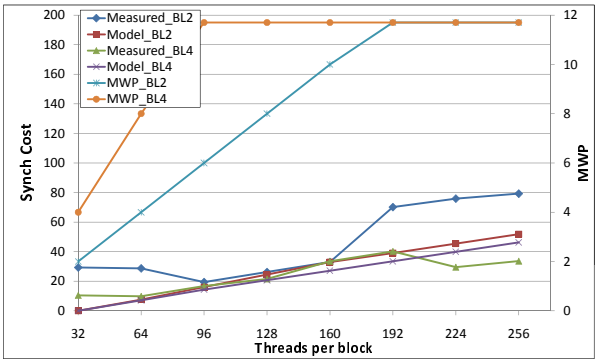

Figure 31. Synchronization delay: 2 vs 4 blocks allocated per SM

6.5. Limitations of the Analytical Model

Our analytical model does not consider the cost of cache misses such as I-cache, texture cache, or constant cache. The cost of cache misses is negligible due to almost 100% cache hit ratio in most of GPGPU applications. The current G80 architecture does not have a hardware cache for the global memory. Typical stream applications running on the GPUs do not have strong temporal locality. However, if an application has a temporal locality and a future architecture provides a hardware cache, the model should include a model of cache. In future work, we will include cache models.

7. Related Work

We discuss research related to our analytical model in the areas of performance analytical modeling, and GPU performance estimation.

7.1. Analytical Modeling

There have been many existing analytical models proposed for superscalar processors [29, 27, 26]. Most work did not consider memory level parallelism or even cache misses. Karkhanis and Smith [19] proposed a first-order superscalar processor model to analyze the performance of processors. They modeled long latency cache misses and other major performance bottleneck events using a first-order model. They used different penalties for dependent loads. Recently, Chen and Aamodit [8] improved the first-order superscalar processor model by considering the cost of pending hits, data prefetching and MSHRs(Miss Status/Information Holding Registers). They showed that not modeling prefetching and MSHRs can increase errors significantly in the first-order processor model. However, they only showed memory instructions' CPI results comparing with the results of a cycle accurate simulator.

There is a rich body of work that predicts parallel program performance prediction using stochastic modeling or task graph analysis, which is beyond the scope of our work. Saavedra-Barrera and Culler [33] proposed a simple analytical model for multithreaded machines using stochastic modeling. Their model uses memory latency, switching overhead, the number of threads that can be interleaved and the interval between thread switches. Their work provided insights into the performance estimation on multithreaded architectures. However, they have not considered synchronization effects. Furthermore, the application characteristics are represented with statistical modeling, which cannot provide detailed performance esti-

mation for each application. Their model also provided insights into a saturation point and an efficiency metric that could be useful for reducing the optimization spaces even though they did not discuss that benefit in their work.

Sorin et al. [36] developed an analytical model to calculate throughput of processors in the shared memory system. They developed a model to estimate processor stall times due to cache misses or resource constrains. They also discussed coalesced memory effects inside the MSHR. The majority of their analytical model is also based on statistical modeling.

7.2. GPU Performance Modeling

Our work is strongly related with other GPU optimization techniques. The GPGPU community provides insights into how to optimize GPGPU code to increase memory level parallelism and thread level parallelism [15]. However, all the heuristics are qualitatively discussed without using any analytical models. The most relevant metric is an occupancy metric that provides only general guidelines as we showed in the Section 2.4. Ryoo et al. [32] proposed two metrics to reduce optimization space for programmers by calculating utilization and efficiency of applications. However, their work focused on non-memory intensive workloads. We thoroughly analyzed both memory intensive and non-intensive workloads to estimate the performance of applications. Furthermore, their work just provided optimization spaces to reduce program tuning time. In contrast, we predict the actual program execution time. Bakhoda et al. [7] implemented a GPU simulator and analyzed the performance of CUDA applications using the simulation output.

Recently, Baghsorkhi et al. [6] proposed a model using a workflow graph as an abstract interpretation of a GPU kernel. PDG (program dependence graph) which contains control and data dependence information is used to predict performance. Kothapalli et al. [22] used a combination of known models (BSP, PRAM, QRQW) for predicting GPU performance. Predicting multiple GPU performance using a single GPU performance is proposed by Schaa et al. [34].

Luk et al. [24] empirically modeled the performance of GPGPU applications as a linear model using run-time information for a dynamic compilation system. Williams et al. proposed a roofline model to visualize the performance of multicore architectures [40]. The roofline model sets an upper bound on the

performance of a kernel that depends on memory intensity and computation intensity metrics.

Recently, several application programmers have developed a performance model for specific applications. Choi et al. [9] proposed a GPU Kernel performance model of sparse matrix-vector multiply (SpMV) kernel for autotuning. The proposed model guides the autotuning process that is input-matrix dependent. Meng et al. [25] presented a model for optimizing iterative stencil loops used for image processing, data mining and physical simulations. Govindaraju et al. [14] presented a memory model to improve the performance of applications (SGEMM, FFT) by improving texture cache usage. The work by Liu et al [39] modeled the performance of bio-sequence alignment applications written in GLSL (OpenGL Shading Language) [20]. All these models are simplified for specific applications where our model is generic to all GPGPU applications.

8. Conclusions

This paper proposed and evaluated a memory parallelism aware analytical model to estimate execution cycles for the GPU architecture. The key idea of the analytical model is to find the maximum number of memory warps that can execute in parallel, a metric which we called MWP, to estimate the effective memory instruction cost. The model calculates the estimated CPI (cycles per instruction), which could provide a simple performance estimation metric for programmers and compilers to decide whether they should perform certain optimizations or not. Our evaluation shows that the geometric mean of absolute error of our analytical model on micro-benchmarks is 5.4% and on GPU computing applications is 13.3%. We believe that this analytical model can provide insights into how programmers should improve their applications, which will reduce the burden of parallel programmers.

9. References

[1] ATI Mobility RadeonTM HD5870 Graphics-Overview. http://www.amd.com/us/press-releases/Pages/amd-press-release-2009sep22.aspx.

[2] Intel Core2 Quad Processors. http://www.intel.com/products/processor/core2quad.

[3] NVIDIA GeForce series GTX280, 8800GTX, 8800GT. http://www.nvidia.com/geforce.

[4] NVIDIA Quadro FX5600. http://www.nvidia.com/quadro.

[5] Advanced Micro Devices, Inc. AMD Brook+. http://ati.amd.com/technology/streamcomputing/AMD-Brookplus.pdf.

[6] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. W. Hwu. An adaptive performance modeling tool for gpu architectures. In *PPoPP*, 2010.

[7] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed GPU simulator. In *IEEE ISPASS*, April 2009.

[8] X. E. Chen and T. M. Aamodt. A first-order fine-grained multithreaded throughput model. In *HPCA*, 2009.

[9] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *PPoPP*, 2010.

[10] E. Lindholm, J. Nickolls, S.Oberman and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March-April 2008.

[11] M. Fatica, P. LeGresley, I. Buck, J. Stone, J. Phillips, S. Morton, and P. Micikevicius. High Performance Computing with CUDA, SC, 2008.

[12] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *MICRO*, 2007.

[13] A. Glew. MLP yes! ILP no! In *ASPLOS Wild and Crazy Idea Session '98*, Oct. 1998.

[14] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *SC*, 2006.

[15] GPGPU. General-Purpose Computation Using Graphics Hardware. http://www.gpgpu.org/.

[16] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. Technical Report TR-2009-003, Atlanta, GA, USA, 2009.

[17] W. W. Hwu and D. Kirk. Ece 498 al: Applied parallel programming, spring 2010. http://courses.ece.uiuc.edu/ece498/al/.

[18] Intel SSE / MMX2 / KNI documentation. http://www.intel80386.com/simd/mmx2-doc.html.

[19] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ISCA*, 2004.

[20] J. Kessenich, D. Baldwin, and R. Rost. The OpenGL shading language. http://www.opengl.org/documentation.

[21] Khronos. Opencl - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

[22] K. Kothapalli, R. Mukherjee, S. Rehman, S. Patidar, P. J. Narayanan, and K. Srinathan. A performance prediction model for the cuda gpgpu platform. In *HiPC*, 2009.

[23] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS XIII*, 2008.

[24] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Micro*, 2009.

[25] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, 2009.

[26] P. Michaud and A. Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *HPCA*, 2001.

[27] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *PACT*, 1999.

[28] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, March-April 2008.

[29] D. B. Noonburg and J. P. Shen. Theoretical modeling of superscalar processor performance. In *MICRO-27*, 1994.

[30] NVIDIA Corporation. *CUDA Programming Guide, V3.0.*

[31] M. Pharr and R. Fernando. *GPU Gems 2*. Addison-Wesley Professional, 2005.

[32] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S. Ueng, J. Stratton, and W. Hwu. Program optimization space pruning for a multithreaded gpu. In *CGO*, 2008.

[33] R. H. Saavedra-Barrera and D. E. Culler. An analytical solution for a markov chain modeling multithreaded. Technical report, Berkeley, CA, USA, 1991.

[34] D. Schaa and D. Kaeli. Exploring the multiple-gpu design space. In *IPDPS*, 2009.

[35] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman,

R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 2008.

[36] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic evaluation of shared-memory systems with ILP processors. In *ISCA*, 1998.

[37] D. Tarjan, J. Meng, and K. Skadron. Increasing memory miss tolerance for simd cores. In *SC*, 2009.

[38] C. A. Waring and X. Liu. Face detection using spectral histograms and SVMs. *Systems, Man, and Cybernetics, Part B, IEEE Transactions on*, 35(3):467–476, June 2005.

[39] S. B. Weiguo Liu, Muller-Wittig. Performance predictions for general-purpose computation on gpus. 2007.

[40] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

Algorithms 1 and 2 describe how to to determine the type of memory accesses and the number of memory instructions for each type. Depending on GPU computing version, the number of memory transactions for uncoalesced case is different. Algorithm 1 is for versions less than 1.3 and Algorithm 2 is for versions higher than or equal to 1.3.

Algorithm 1 Algorithm to detect coalesced/uncoalesced memory requests

INPUT
A source code which accesses a global memory is obtained.
(e.g., float localvar = dm_data[Index] where dm_data is a pointer to a global memory)
(Note: Index is a function of induction variables (if any), thread and block identifiers)
STEP1
If sizeof(localvar) is 4, 8, or 16 bytes, then STEP1 successful
STEP2
If d/d(tx) Index == 1, then STEP2 successful // Incremental value of Index with respect to tx
STEP3
Let T be a subset of Index in which beginning thread indices for an half-warp are used
(e.g., T = Index|tx=16t, t=0,1,2,...,n such that tx <= txmax)
If all index values in T are divisible by 16, then STEP3 successful
OUTPUT
If STEP1, STEP2, STEP3 are all successful, then the memory access is coalesced
Else the memory access is uncoalesced

Algorithm 2 Algorithm to detect the size and the number of memory transactions (Computing version 1.3 or above)

INPUT
A source code which accesses a global memory is obtained.
(e.g., float localvar = dm_data[Index] where dm_data is a pointer to a global memory)
(Note: Index is a function of induction variables (if any), thread and block identifiers)
STEP1
N = 0
STEP2
Let T be a subset of Index in which beginning thread indices for an half-warp are used
(e.g., T = Index|tx=16t, t=0,1,2,...,n such that tx <= txmax)
If any index value in T is not divisible by 16, then N=N+1
STEP3
Result_increment = d/d(tx) Index // Incremental value of Memory_index with respect to tx
Result_size = 32 bytes if sizeof(localvar) is 1 64 bytes if sizeof(localvar) is 2 128 bytes if sizeof(localvar) is 4, 8, 16
STEP4
N = N + ceiling[(Result_increment * 16 * sizeof(localvar)) / Result_size]
OUTPUT
Transactional size = Result_size
Number of memory transactions = N