MODELING PERFORMANCE AND POWER FOR ENERGY-EFFICIENT GPGPU COMPUTING

A Thesis
Presented to
The Academic Faculty

by

Sunpyo Hong

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
December 2012

MODELING PERFORMANCE AND POWER FOR ENERGY-EFFICIENT GPGPU COMPUTING

To my family.

ACKNOWLEDGEMENTS

I would like to take this opportunity to give thanks and a praise to God for giving me this opportunity, a guidance, and the strength to finish the study.

This dissertation would not have been possible without an encouragement from my wife, Juri Kim. It was her constant devotion that enabled me to move forward. Most importantly, I give my deepest appreciation and thanks to my parents, Soontae Hong and Cheolsook Choi. Without their unconditional support in everything, nothing would have been possible. I also give many thanks to Heejung Hong and Moohyun Song and to grandmother, Seongho Im, for their supports throughout the years.

I give my deep special thanks to my advisor, Prof. Hyesoon Kim. Without her guidance and passion in research, studying on this interesting topic and publishing in top conferences would not have been literally possible. Fortunately, I had a pleasure of traveling to different cities and even abroad, Saint-Malo in France, to present my work. I learned why I should always target the highest goal and focus on the most important problems. She has always guided me in the right direction, given detailed advices, and demonstrated how to be a good researcher and mentor without explicit words.

I give special thanks to Dr. Chi-Keung Luk for being a mentor ever since I started my study. He provided me an internship opportunity, and guided me on many technical challenges regarding the state-of-the-art tools, processor products, and research. I had a great time during the internship and had the opportunity to broaden my perspectives outside the research.

I would like to thank all the committee members. I thank Prof. Sudhakar Yalamanchili for the efficient communication for making the defense to happen as well as for the valuable advices. I particularly thank Prof. Milos Prvulovic for providing me a constructive high-level discussion regarding my work and suggestions for my future career. I thank Prof. Moinuddin Qureshi for the detailed feedback and sharp comments that I learned during the presentation. I also give special thanks to Prof. Richard Vuduc for arranging a full-scale separate presentation to prepare for the real defense and for those valuable comments.

I thank to all my lab members. I particularly thank Minjang for having to go through the same challenges, and for all the help related to job searching, interview, and coursework. I give thanks to Jaewoong and Jaekyu for their detailed feedback on the work, including the presentation and to Nagesh for helping on research and many Linux-related problems. I thank Joohwan for handling much of the on-going related work. Thanks to Hyojong for helping with Visual Studio as well as those fun table tennis games and Jieun for power-related discussions and experiments. Thanks to Pranith for the job-related discussions, Puyan for working on the experiments that proved useful for my research, Deukkyu for the study discussions and fun times, Jonghoon and Changhee for the various discussions, Jeewhan for helping me in setting up the Ph.D. direction during the early period.

I give special thanks to Sunghoon and Hoseon, whom I met during the undergraduate years at Georgia Tech. They constantly motivated me to study hard as well as being best friends. Also, many thanks go to friends and colleagues: Aniruddha, Chayoung, Daehyun Kim, Dongwon, Donghyuk, Honggab, Ioannis, Jaeyong, Jungju, Kaushik, Kihwan, Kwanyeob, Sangmin, Sangho, Sungbae, Sungkap, Yuejian. In addition to those whom I know personally, I would like to thank anonymous conference and journal reviewers who volunteered to read my papers and provided useful comments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Numerous problems that require a high computing power have existed since the emergence of the first computers. However, although computing power has consistently improved from increasing clock frequency, that improvement has mainly focused on the performance of single-threaded applications. As a result, complex problems still fell into the hands of people working in super-computing centers.

Given the particular time frame in which we are living, this is no longer the case. With the advent of graphics-processing unit (GPU), ordinary programmers, who used to have access to only multi-core processors, now have access to hundreds of execution units. Additionally, with the emergence of general-purpose GPU computing (GPGPU), the outcome is spectacular, a high number of programmers across the globe starting to convert their scientific and engineering applications to GPGPU computing. Many have reported speed-ups by a factor of more than 100 [83] on real-world applications. Thus, it appears on the surface as though computer development has reached the ultimate goal. However, it is important to assess whether this in reality has actually occurred.

The design of the GPU architecture has advanced at a fast rate in the market, and even a heterogeneous processor, which exhibits both CPU and GPU characteristics [9], began to emerge in the market. However, such a change is not fast enough for a programming model and programmers. Most programmers are still likely to keep optimizing their application performance blindly; that is, they keep optimizing the programs without first understanding the essential features of the architecture. This is a serious problem, and it may eventually lead to a higher cost for software development in terms of time and money. Furthermore, although profiling tools exist in addition to other parallelizing tools, they do have one limitation in common; they neither provide insightful knowledge of the architecture nor capture essential architectural features that affect performance. Thus, the relevant question then becomes rather than merely running the tools or simulations, is it possible to use only essential features and gain insight into performance?

To address this question, we proposed an analytical model that predicts performance for GPUs [33] in 2009. The model is based on a technical premise that memory-access cycles between multiple warps can be overlapped, and that amount of memory-level warp parallelism (MWP) depends on available bandwidth and internal dynamic random access memory (DRAM) cycles. Thus, by precisely knowing the effect of bandwidth and types of memory accesses, an effective memory latency can be calculated, thereby predicting an application's performance.

The benefit of this modeling approach is that, unlike other established profiling tools or hardware performance counters, the analytical model only uses key features of the GPU architecture to predict performance. This means that a person using this model is able to visualize performance-determining factors in a graph and point out the factors that limit performance. Therefore, a programmer not only finds performance-limiting factors, but also gains an in-depth understanding of the architecture. This work also provides several techniques for performance optimization by suggesting the generation of more threads, notifying that memory bandwidth is insufficient, and improving a memory transaction type to a coalesced memory-access.

Another important aspect of GPU-kernel optimization is power and energy savings. Most programmers so far have focused on performance optimizations because, traditionally optimizing for performance is better for energy savings, since the execution-time reduction often outweighs the benefits of other energy-saving techniques. However, in the GPU architecture, this is no longer true. In my previous work [34] in 2010, it was demonstrated that in some applications, there are different optimization points for saving energy. This observation is based on a detailed power model that I developed by using an empirical method and a previous performance model; I proposed the integrated power and performance prediction (IPP) framework to estimate an optimal thread and block configuration in compute-unified device architecture (CUDA) language. This work, published in ISCA 2010, is based on the insight that not all GPU cores need to be activated, depending on the type of application. Furthermore, this work can be used with other power-management techniques such as core gating and voltage scaling. In essence, the work demonstrates the benefit that when the power-gating technique is employed for a bandwidth-limited application, using fewer cores saves more than 10% of energy consumption.

In the future, we envision that a computer system will contain hundreds of cores. The challenge lies in the fact that some applications run more efficiently on a CPU than on a GPU. For this reason, most future processors will be heterogeneous in nature; a processor will have both a CPU and a GPU. Therefore, it is mandatory from both a performance and power perspective that work gets scheduled to a more efficient core.

To address that issue, I co-developed a dynamic-compiler system, Qilin [66], that reduces a programmers burden of mapping between a CPU and GPU. Based on the performance model, Qilin effectively predicts how much computation needs to be done on the CPU and on the GPU. Moreover, Qilin dynamically recompiles code and performs automatic partitioning of data depending on input and workload characteristics. The work shows that the best performance is achieved when the CPU and GPU are efficiently used together. While this dynamic approach finds a balanced workload distribution after a profiling phase, to reduce the overhead of profiling time and to provide insights into CPU and GPU analytically, I propose a static-time approach that requires instruction mixture and input-data size information. This work will leverage my previous work, which models performance and power for GPU architectures.

Finally, this research makes great strides in the analytical work domain. My work provides an easy to use, but thoroughly informative, insight into how to improve the performance and power efficiency of applications. It can also be extended to a compiler domain to guide performance optimizations or to architecture-simulation domains to provide fast and scalable simulation time. Because of this research, researchers and scientists will program their applications for GPGPU computing more easily and efficiently, thereby enhancing the discoveries of science and medicines.

1.1 Thesis Statement

An analytical model that predicts performance and power provides insights and can assist energy-efficient execution for a many-core architecture, which is not limited to a graphics processing unit.

1.2 Organization

The remainder of this document is presented as follows: Chapters II and III provide the research work for modeling GPU performance and power. Chapter IV presents the thermal analysis using customized thermospacer and thermocouples. Chapter V presents the throughput model that is a significant extension of the performance work. This work has modified the analytical model structure from bottom up, which enables a more thorough analysis in terms of bandwidth, achievable performance given instruction mixture, memory effect, etc. Chapter VI presents the related work to analytical modeling for performance and power, OpenCL, and energy-efficient execution in the heterogeneous system. Chapter VII presents the conclusion and future research directions. Appendix A introduces a preliminary work for investigating energy efficiency using the analytical model.

CHAPTER II

MODELING GPU PERFORMANCE

To provide insight into performance bottlenecks in massively-parallel GPU architectures, an analytical model is proposed. The model can be used statically without executing an application. The basic intuition of the 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 warps,[1] the amount of memory-bandwidth consumption, and the memory types, the number of overlapping memory accesses can be predicted, which is quantified as the memory warp parallelism (MWP) metric. Another metric that models computation is defined as 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 the arithmetic-intensity metric used in the GPGPU community [80], which is defined as the number of mathematical operations per memory operation. Then, by using both MWP and CWP values, the effective cost of memory requests can be estimated, thereby predicting an overall execution time of a GPU program.

2.1 Background

A brief background on the GPU architecture and the programming model is discussed.

[1]A warp is a batch of threads that are internally executed together by the hardware.

2.1.1 Background on the Programming Model

The compute-unified device architecture (CUDA) programming model is similar to a single program multiple data (SPMD) software model. In other words, each thread in a CUDA program executes the same code, but it accesses different memory locations and registers.

The programming model uses a hierarchy of thread groups: grid, block, and thread. A grid is a set of thread blocks that executes a kernel function, and each block is composed of hundreds of threads. There are three memory spaces: local, shared, and global. Local memory space is within a thread, shared memory space is within a block, and global memory space is within a grid. Hence, threads within a block can share data using shared memory, but across blocks, the data must be written back to the global memory. All threads within a block run concurrently using fine-grain multi-threading. A barrier can be enforced within a block. However, depending on the machine resources, blocks are not guaranteed to run concurrently. CUDA also provides two read-only memory spaces: constant space, and texture space.

2.1.2 Mapping the Programming Model to the Architecture

Figure 1 shows the high-level view of a GPU architecture. The version of a GPU in the figure is NVidia's G80 architecture, which is applicable to these series of GPUs that we actually used: 8800GTX, FX5600, 8800GT. This figure is also applicable to GT200 series such as GTX280 GPU that we used, since the architectural change for GT200 is at a more finer level than the figure shows. The architecture consists of a scalable number of streaming multi-processors (SM). Each SM contains eight streaming processors (SP), two special function units (SFU), a multi-threaded instruction-fetch unit, a read-only constant cache, and a 16KB shared memory [63].

Threads are dispatched to SM at the block granularity. The number of blocks that can be assigned to SM depends on the available resources (e.g., register file size, shared memory size, and thread-contexts size) and the amount of resources each block uses. If the maximum number of threads is assigned to SM, the occupancy metric[2] is one. During execution, SM forms a batch of 32 threads together, called a warp, which is the granularity of single instruction multiple data (SIMD) execution.

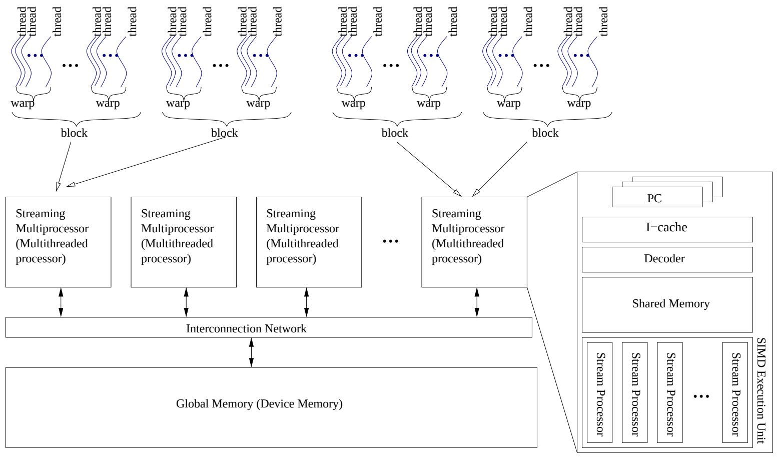


Figure 1: An overview of the GPU architecture.

Executing a warp instruction applies the same instruction to 32 threads, similar to executing a streaming SIMD extension (SSE) instruction in X86. However, unlike SSE instruction, the concept of warp is not exposed to the programmers, so they do not have to explicitly write a program that utilizes hardware resources. Instead, the architecture transparently forms a warp for execution.

The blocks that are running on one SM at a given time are called active blocks. Since one block typically has several warps, which is 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 multiplied by the number of active blocks.

The shared memory is implemented as a static random access memory (SRAM), whereas the global memory is implemented as a dynamic random access memory (DRAM). The shared memory has very low access latency, which is almost the same

[2]The metric is the ratio of assigned threads to maximum thread contexts per SM. Higher occupancy means more threads per SM, hence idle memory cycles are hidden more effectively with more available threads.

as that of accessing a register. However, when a warp accesses the shared memory simultaneously causing bank conflicts to occur within a warp, additional delay occurs.

2.1.3 Warp Execution and Types of Memory Accesses

SM executes one warp as single instruction multiple threads (SIMT) and schedules warps in a time-sharing fashion. The processor has enough functional units and register ports to execute 32 threads together. Since SM has eight functional units, as shown in Figure 2, issuing 32 threads takes four processor cycles for computation instructions.[3] When a memory instruction is executed, SM generates memory transactions and switches to another warp until all the memory values in the warp are ready. Ideally, all memory requests within the warp can be combined into one or more memory transactions. Unfortunately, that depends on the access patterns within the warp, the first memory address, the requested data size, and the hardware version (i.e., compute version). For the list of GPU products with different compute versions, the CUDA manual[77] should be referenced.

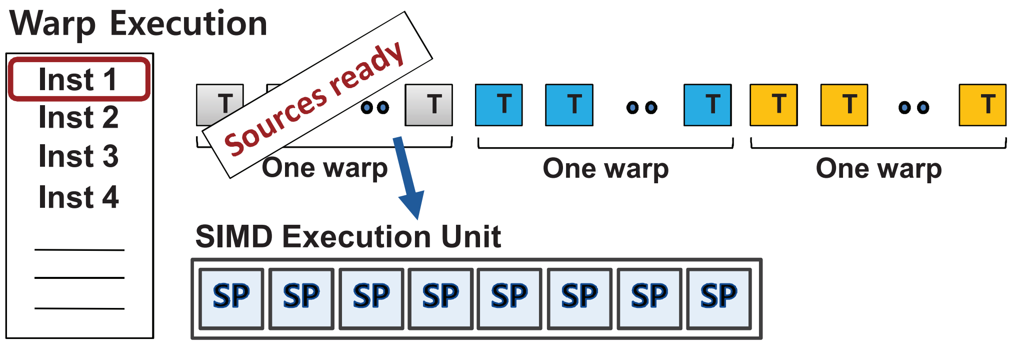


Figure 2: Warp execution.

To illustrate the high-level view of a memory transaction, Figure 3 illustrates the three cases. Regardless of the hardware version, when the first-requested address is aligned to 64 bytes, and the subsequent addresses are sequential, then only one memory transaction is generated, as shown in Case (a) of Figure 3. However, if one

[3] A computation instruction means a non-memory instruction.

of the conditions is not met, then either Case (b) or Case (c) can occur depending on the GPU compute version. Case (b) will occur for the earlier versions of 1.0 and 1.1, where multiple memory transactions are generated. This consumes memory bandwidth and degrades performance. Because SM is an in-order processor; the warp cannot continue execution until all the memory transactions are serviced.

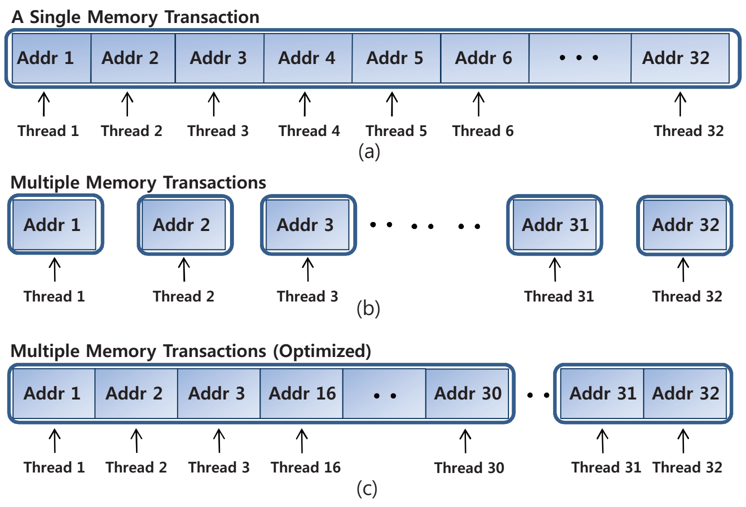


Figure 3: Memory transactions: (a) coalesced, (b) uncoalesced, (c) optimized.

For the versions of 1.2, 1.3, and higher, the number of memory transactions is optimized by hardware to reduce the memory traffic, as shown in Case (c) of Figure 3. Even if the first memory address is unaligned, the hardware will attempt to generate two large memory transactions for each memory segment rather than generating one transaction for each thread. To facilitate the discussion, a memory request is a coalesced access if only one memory transaction is generated and an uncoalesced access otherwise. A local-memory access is treated the same as a global-memory access.

In this work, shared-memory access is treated the same as register access. In other words, we assume that there will be no bank conflicts. The same approach is used for constant and texture caches.

2.2 Motivation

To motivate the importance of a performance analysis on the GPU architecture, the example of three different implementations of the same algorithm is shown in Figure 4. The SVM benchmark is a kernel extracted from the face-classification algorithm [101]. The performance of applications is measured on the QuadroFX5600 GPU. There are three different implementations: naive, constant, and constant+optimized. The naive version only uses the global memory. The constant version additionally uses the constant memory.[4] Unlike previous two versions, the constant+optimized version optimizes memory accesses to generate coalesced memory transactions.

Figure 4 shows the execution time when the number of threads per block is varied. Despite the changing number of threads, the number of blocks is adjusted to keep the total work the same. The performance improvement of the constant+optimized version and that of the constant version over the naive implementation is 24.36 and 1.79 times speed-up, respectively. Even though the 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.

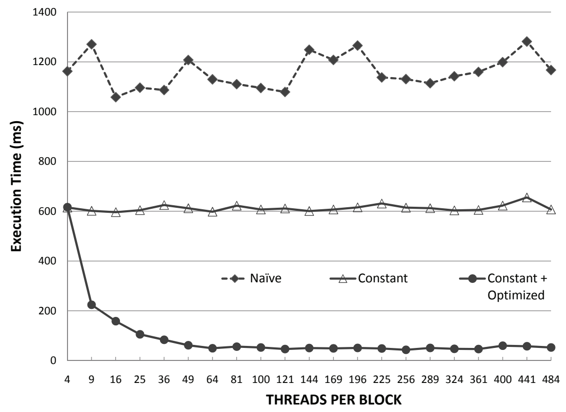


Figure 4: Optimization impacts on the SVM benchmark.

Figure 5 shows the occupancy [77] values for the three versions. As mentioned

[4]The benefit of using the constant memory is that the memory has an on-chip cache per SM, and register usage can be reduced, which can increase the number of running blocks in one SM.

in Section 2.1.2, an occupancy is a ratio of assigned threads to maximum number of thread contexts per SM. This means that the higher the occupancy, the more warps exist inside the SM for context-switching during a long-latency memory access. Traditionally, this metric has been used for optimizing an application performance [102], and programmers have attempted to reduce register usage and shared memory access to increase occupancy.

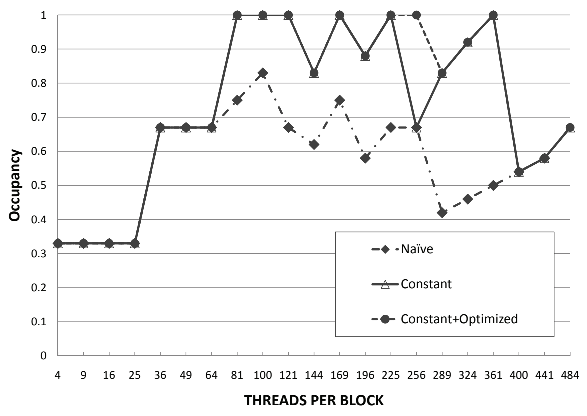


Figure 5: Occupancy values of the SVM benchmark.

Typically, a high occupancy is better for optimizing performance since many threads can hide the DRAM latency more effectively. However, Figure 5 shows that the occupancy metric does not sufficiently estimate the performance improvement. First, when the number of threads per block is less than 64, all the three cases show the same occupancy values even though the performance is different. Second, even though the occupancy value is increased, performance does not improve for some cases. For example, the performance of the constant version is not improved at all, even though the occupancy is increased from 0.35 to one. The reason is that when there are more warps than the available memory parallelism, the performance will not significantly improve with the increasing number of warps. Hence, we need other metrics to differentiate these three cases.

2.3 Analytical Model

2.3.1 Introduction of the Analytical Metrics

The GPU architecture is a multi-threaded 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 concurrently-executed warps can be hidden. The key purpose of the analytical model is to find 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, two metrics are introduced: memory warp parallelism (MWP), and computation warp parallelism (CWP). MWP represents the maximum number of warps per SM that can access the memory simultaneously during one memory-access period of one warp. The time period from right after one warp sends memory requests until all the memory requests from that same warp are serviced is called one memory-waiting period. CWP represents the number of warps that the SM can execute during one memory-waiting period plus one. A value of one is added to include the warp itself that is waiting for memory values. This addition means that CWP is always greater than or equal to one.

MWP is related to how much memory parallelism exists in the system. MWP is determined by the memory bandwidth, memory-bank parallelism, and the number of running warps per SM. MWP plays an important role in the analytical model. When the MWP value is higher than one, 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 the MWP is described in Section 2.3.2.

CWP is related to the program characteristic. It is similar to an arithmetic intensity [80], which represents the number of mathematical operations per memory access. However, unlike the arithmetic intensity, higher CWP means less computations per memory access. CWP also considers timing information, while an arithmetic intensity does not. CWP is mainly used to decide whether the total execution time is dominated by the computation cost or the memory access cost. When CWP is greater than MWP, the execution cost is dominated by the memory access cost. However, when MWP is greater than CWP, the execution cost is dominated by the computation cost.

2.3.2 The Cost of Executing Multiple Warps

To explain how executing multiple warps in each SM affects the total execution time, several scenarios are illustrated in Figures 6, 7, 8, and 9. A computation-period indicates the period when instructions from one warp are executed on the SM. A memory-waiting period indicates the period when memory requests are being serviced. The numbers inside the computation-period boxes and the memory-waiting period boxes in Figures 6, 7, 8, and 9 indicate a warp identification number.

Using Figure 6a as the baseline example, each warp has only one set of computations and memory accesses. In other words, no next set of dependent computations and memory accesses exists. Assume that the GPU memory system can service two memory warps simultaneously. Since one computation period is roughly one-third of one memory-waiting period, the SM can finish three warps' computation periods during one memory-waiting period. That means that MWP is two, and CWP is four. As a result, the six computation periods completely overlap with other memory-waiting periods. Hence, only two computations and four memory-waiting periods contribute to the total execution cycles.

Figure 6b shows a more realistic example; the next dependent sets of computations and memory accesses are added. The second computation period can start only after the first memory-waiting period of the same warp is finished. The MWP and CWP values are still the same as in Figure 6a. First, the SM 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. 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 idle cycles between the computation periods, the total execution cycles are the same as in Figure 6a. When CWP is higher than MWP, more computations from different warps can be finished during one memory-waiting period. Hence, the cost of the computation periods can almost always be hidden during a memory access.

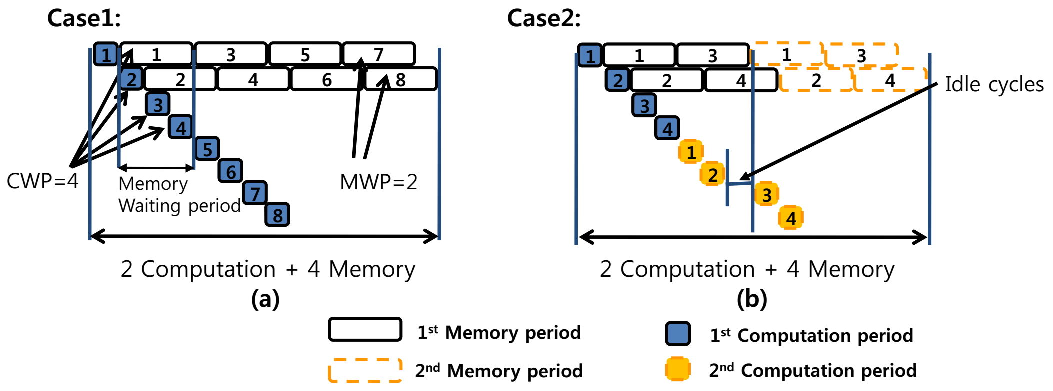


Figure 6: CWP is greater than MWP: (a) eight warps, (b) four warps.

For both cases, the total execution cycles are the sum of two computation periods and four memory-waiting periods. Using MWP, the total execution cycles can be calculated by using Equation (1) and Equation (2) below. We divide *Comp_cycles* by *#Mem_insts* to get the number of cycles for one computation period. Note that the number of warps allocated per SM, N in Equation (1), is divided by MWP that represents memory parallelism.

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

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

MWP is greater than CWP for some cases. Figure 7a shows that the system can service eight memory warps concurrently. Since eight memory cycles are overlapped, the values of MWP and CWP are eight and four. Then, as soon as the first computation-period finishes, the processor can process the next set of memory and computation requests. The example in Figure 7a shows that the memory-waiting periods all overlap between each other except the last warp. Hence, the total execution cycles are the sum of eight computation periods and only one memory-waiting period.

Figure 7b shows an example with the next sets of dependent computations and memory accesses. Even with those next sets of instructions, since the memory access cycles are overlapped between the warps, the total execution cycles are dominated by the computation cycles, which are the sum of eight computation periods and only one memory-waiting period. Hence, when MWP is higher than CWP, the execution cycles can be calculated by Equation (3).

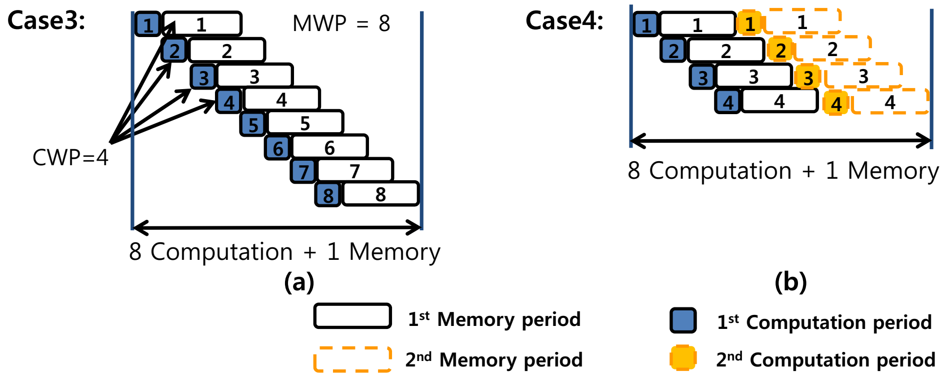


Figure 7: MWP is greater than CWP: (a) eight warps, (b) four warps.

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

Figure 8 shows an extreme case where not even one computation period can be finished while one memory-waiting period is completed. Even if MWP is eight, the application cannot take advantage of all the available memory parallelism. As a result, the total execution cycles are eight computation periods plus one memory-waiting period. This example shows that while MWP determines the available memory parallelism on the specific hardware, CWP also plays an important role in determining the execution behavior.

When an application does not have enough number of warps, the system cannot take advantage of all the available warp parallelism. By definition, MWP and CWP values cannot be greater than the number of active warps in SM, which is represented by N term.

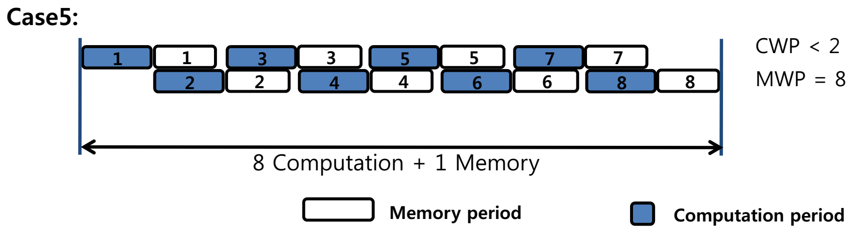


Figure 8: Computation cycles are greater than memory-waiting cycles.

Figure 9a shows the case when only one warp is running. Since there is no other warp that SM can switch to, all the executions are serialized. Hence, the total execution cycles are the sum of the computation and memory-waiting periods, where both CWP and MWP values are one in this case. Figure 9b shows two warps. Since MWP is limited by N, MWP is two. Even if one computation period is less than half of one memory-waiting period, because there are only two warps, CWP is two. Hence, the total execution time is roughly half the sum of all the computation periods and memory-waiting periods, as shown in Equation (4).

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

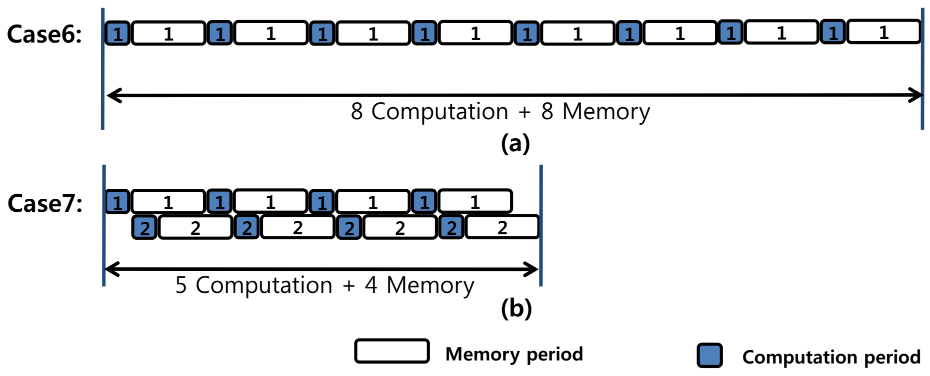


Figure 9: MWP is equal to N: (a) one warp, (b) two warps.

2.3.3 Calculating the Degree of MWP

MWP is slightly different from memory level parallelism (MLP) [27]. MLP represents how many memory requests can be serviced together. MWP represents the maximum number of warps in SM that can access the memory system simultaneously during one memory-waiting period. The main difference between MLP and MWP is that MWP counts all memory requests from a warp as one unit, while MLP counts all individual memory requests separately. As discussed in Section 2.1.3, one memory instruction in a warp can generate multiple memory transactions. This difference is important, because a warp cannot be executed until all values for a warp are ready.

MWP is tightly coupled with the DRAM system. In our analytical model, DRAM system is modeled as a simple queue; each active SM consumes an equal amount of memory bandwidth. Figure 10 shows the memory model and a time-line of memory warps.

MWP represents the number of memory warps per SM that can be handled during one memory-waiting period, represented by Mem_L in Figure 10. The latency of each memory transaction 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 DRAM, which includes access time, address translation, and data transfer time.

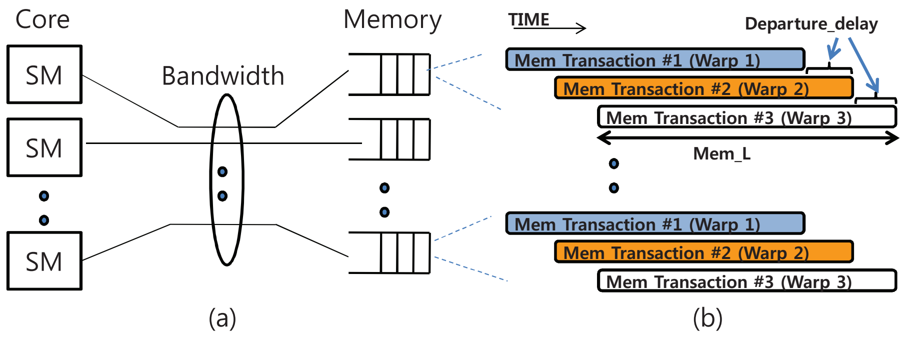

Core
SM
SM
SM
Bandwidth
Memory
(a)
TIME
Departure_delay
Mem Transaction #1 (Warp 1)
Mem Transaction #2 (Warp 2)
Mem Transaction #3 (Warp 3)
Mem_L
Mem Transaction #1 (Warp 1)
Mem Transaction #2 (Warp 2)
Mem Transaction #3 (Warp 3)
(b)

Figure 10: Memory system: (a) memory model, (b) time-line of memory warps.

The amount of memory-level parallelism depends on the following parameters: an available memory bandwidth, and a latency of the departure_delay with respect to Mem_L cycles. For example, even with an infinite memory bandwidth, if departure delay is significant, then fewer memory warps end up overlapping. As mentioned previously, MWP is also limited by how many active warps are allocated per SM. Therefore, MWP is modeled by Equation (5). MWP cannot be greater than the number of warps per SM that reach the peak memory bandwidth, represented by MWP_peak_BW, of the system. If fewer SMs are executing warps, then each SM can consume more bandwidth than when all SMs are executing warps.

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

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

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

When an application does not reach peak bandwidth, MWP is a function of memory-waiting period and departure delay. Figure 11 shows that the number of overlapped warps is obtained by dividing Mem_L by $Departure\_delay$. It also shows that both terms depend on memory-access types and GPU hardware versions. This is because, depending on the memory access type, a different number of memory transactions is generated, as shown in Figure 3. More transactions require additional processing cycles, thereby increasing the departure delay term (Equation (15)). For GPUs with the hardware versions of 1.3 and above, a memory request is optimized into as few memory transactions as possible.

Figure 12 shows that the latency of memory warps is dependent on the memory access type. For an uncoalesced memory request, since one warp requests multiple transactions, which is represented by $\#Uncoal\_per\_mw$, Mem_L includes departure delays for all the generated transactions. $Departure\_delay$ also includes $\#Uncoal\_per\_mw$ number of $Departure\_del\_uncoal$ cycles. Mem_LD is a round-trip latency to DRAM for each memory transaction. In this model, Mem_LD for uncoalesced and coalesced memory types is considered the same, even though a coalesced memory request might take a few more cycles because of a large data size. [5]

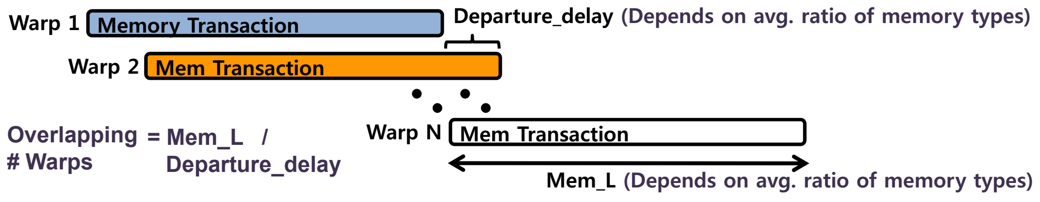

Warp 1
Memory Transaction
Departure_delay (Depends on avg. ratio of memory types)
Warp 2
Mem Transaction
Overlapping # Warps = Mem_L / Departure_delay
Warp N
Mem Transaction
Mem_L (Depends on avg. ratio of memory types)

Figure 11: Effects of memory types and hardware versions on MWP.

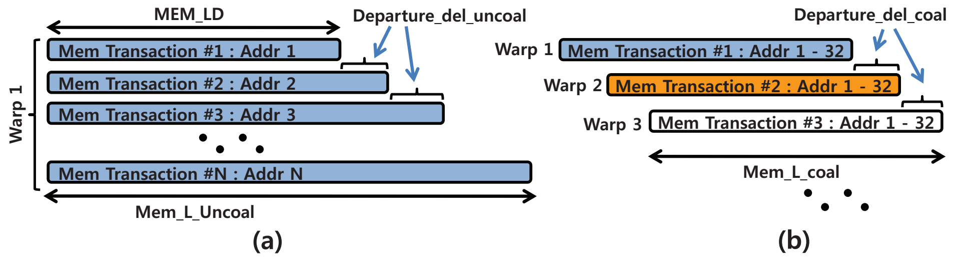

MEM_LD
Departure_del_uncoal
Warp 1
Mem Transaction #1 : Addr 1
Mem Transaction #2 : Addr 2
Mem Transaction #3 : Addr 3
Mem Transaction #N : Addr N
Mem_L_Uncoal
(a)
Departure_del_coal
Warp 1
Mem Transaction #1 : Addr 1 - 32
Warp 2
Mem Transaction #2 : Addr 1 - 32
Warp 3
Mem Transaction #3 : Addr 1 - 32
Mem_L_coal
(b)

Figure 12: Departure delays: (a) uncoalesced, (b) coalesced.

In an application, some memory requests are coalesced, and some are not. Since

[5] Mem_LD refers to the average cycles for a single memory transaction. On the other hand, Mem_L considers multiple memory transactions generated from a memory request of a warp.

multiple warps are running concurrently, the analytical model uses an weighted average of the memory latency of coalesced and uncoalesced memory types. A weight is determined by the number of coalesced and uncoalesced memory requests, as shown in Equation (10) and Equation (11).

2.3.4 Calculating the Degree of CWP

Once memory latency for each warp is calculated, obtaining CWP is straightforward. CWP_full is used when there are enough number of warps. When CWP_full is greater than N, which is the number of active warps per SM, CWP is N. Otherwise, CWP_full becomes CWP.

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

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

2.3.5 Total Number of Executed Blocks Per SM

SM executes hundreds of threads concurrently. Depending on the application, the total number of blocks assigned by the programmer varies. Some applications could contain hundreds of blocks, while others could even contain thousands of blocks. As mentioned in Section 2.1.1, threads are assigned to SM at block granularity, and we assume that these blocks will uniformly spread out on the available number of SMs. For example, if there are 30 SMs and 3000 blocks, each SM will fetch 100 blocks on average. However, these 100 blocks cannot be assigned to SM at one time as current GPU specifications only allow up to eight blocks maximum to be assigned at once due to resource constraints. Hence, the number of blocks that can be assigned at once depends on the amount of resources each block requires such as registers and shared-memory usage per block. This occupancy information and the number of blocks assigned to SM can be calculated at static time [77]. If only five blocks are assigned to SM, then each SM is expected to execute 100 blocks on average. Thus, $\#Rep$ is introduced to represent how many times each SM is expected to repeat the execution of certain number of blocks.

2.3.6 Total Execution Cycles

Several stages of calculations are necessary to obtain the total execution cycles. First, a weight of uncoalesced and coalesced memory accesses is calculated, as shown in Equation (10) and Equation (11).

$$Weight\_uncoal = \frac{\#Uncoal\_Mem\_insts}{(\#Uncoal\_Mem\_insts + \#Coal\_Mem\_insts)} \tag{10}$$

$$Weight\_coal = \frac{\#Coal\_Mem\_insts}{(\#Coal\_Mem\_insts + \#Uncoal\_Mem\_insts)} \tag{11}$$

The term, Mem_L, represents an average latency of a memory access. Since two types of memory accesses exist on GPU architecture, an effective latency is calculated by considering the number of memory accesses for each type, as shown in Equation (14).

$$Mem\_L\_Uncoal = Mem\_LD + (\#Uncoal\_per\_mw - 1) \times Departure\_del\_uncoal \tag{12}$$

$$Mem\_L\_Coal = Mem\_LD + Departure\_del\_coal \tag{13}$$

$$Mem\_L = (Mem\_L\_Uncoal \times Weight\_uncoal) + (Mem\_L\_Coal \times Weight\_coal) \tag{14}$$

In DRAM system, depending on the type of a memory access, the cycles necessary to process consecutive memory accesses are represented by $Departure\_delay$ term, as shown in Equation (15).

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

Then, MWP, the memory-level parallelism metric, is calculated by finding the minimum of $MWP\_Without\_BW$ and MWP_peak_BW as shown in Equation (16) and Equation (17).

$$MWP\_Without\_BW\_full = Mem\_L / Departure\_delay \quad (16)$$
$$MWP\_Without\_BW = MIN(MWP\_Without\_BW\_full, N) \quad (17)$$

Assuming no memory-level parallelism and computation-level parallelism exist, a serialized cycles for memory and computation are calculated in Equation (18) and Equation (19). For computation cycles, the term, M_Factor models different throughputs of computation instructions.

$$Comp\_cycles = \#Issue\_cycles \times M\_Factor \times \#total\_insts \quad (18)$$
$$Mem\_cycles = Mem\_L\_Uncoal \times \#Uncoal\_Mem\_insts \quad (19)$$
$$+ Mem\_L\_Coal \times \#Coal\_Mem\_insts$$

Since all the necessary work can not be computed in one round, Equation (20) shows how many rounds of computations need to be executed.

$$\#Rep = \frac{\#Blocks}{\#Active\_blocks\_per\_SM \times \#Active\_SMs} \quad (20)$$

Table 1: Summary of the model parameters.

Model Parameter	**Definition**	**Obtained**
#Threads_per_block	Number of threads per block	Programmer
#Blocks	Total number of blocks in a program	Programmer
#Active_blocks_per_SM	Number of concurrently running blocks on one SM	Based on machine resources [77]
#Active_SMs	Number of active SMs	Based on machine resources
N	Concurrently running warps on one SM	Active_blocks x Warps_per_block
#Comp_insts	Total dynamic number of computation instructions in one thread	Source code analysis
#Mem_insts	Total dynamic number of memory instructions in one thread	Source code analysis
#Coal_Mem_insts	Number of coalesced memory type instructions in one thread	Source code analysis
#Uncoal_Mem_insts	Number of uncoalesced memory type instructions in one thread	Source code analysis
#Synch_insts	Total dynamic number of synchronization insts in one thread	Source code analysis
#Total_insts	Total dynamic number of instructions in one thread	#Comp_insts + #Mem_insts
Mem_LD	DRAM access latency (Machine configuration)	Table 5
Departure_del_coal	Delay between two coalesced memory transactions	Table 5
Departure_del_uncoal	Delay between two uncoalesced memory transactions	Table 5
#Coal_per_mw	Number of memory transactions per warp (coalesced access)	1
#Uncoal_per_mw	Number of memory transactions per warp (uncoalesced access)	Source code analysis
Mem_L	Warp memory access latency, depends on memory type and HW	Equation (14)
Issue_cycles	Number of cycles to issue one instruction (pipelined)	4 cycles [39]
Freq	Clock frequency of the SM	Table 2
#Threads_per_warp	Number of threads per warp	32 [77]
Mem_Bandwidth	Bandwidth between DRAM and GPU cores	Table 2

Finally, depending on the MWP and CWP values, the total execution cycles for an entire application are calculated by Equation (21), Equation (22), and Equation (23).

Case1: If (MWP is N warps per SM) and (CWP is N warps per SM)

$$(Mem\_cycles + Comp\_cycles + \frac{Comp\_cycles}{\#Mem\_insts} \times (MWP - 1)) \times \#Repw \tag{21}$$

Case2: If (CWP >= MWP) or (Comp_cycles > Mem_cycles)

$$(Mem\_cycles \times \frac{N}{MWP} + \frac{Comp\_cycles}{\#Mem\_insts} \times (MWP - 1)) \times \#Repw \tag{22}$$

Case3: If (MWP > CWP)

$$(Mem\_L + Comp\_cycles \times N) \times \#Repw \tag{23}$$

2.3.7 Cycles Per Instruction (CPI)

Cycles per Instruction (CPI) is commonly used to represent the cost of each instruction. Using total execution cycles, CPI is calculated by using Equation (24). Note that CPI is the cost when an instruction is executed by all threads in one warp.

$$CPI = \frac{Exec\_cycles\_app}{\#Total\_insts \times \frac{\#Threads\_per\_block}{\#Threads\_per\_warp} \times \frac{\#Blocks}{\#Active\_SMs}} \tag{24}$$

2.3.8 Coalesced and Uncoalesced Memory Accesses

A latency of memory instruction is heavily dependent on the memory-access type. Whether memory requests inside a warp can be coalesced or not depends on the memory-system design and memory-access patterns in a warp. The evaluated GPUs have two coalesced and uncoalesced polices. Earlier versions have differences compared with more recent versions of 1.3 and higher. The difference is that stricter rules are applied for a warp to be coalesced. For recent versions, the rules are relaxed; all memory requests are coalesced into as few memory transactions as possible.

2.3.9 Synchronization Effects

The programming model supports a barrier synchronization. Since blocks are asynchronously assigned to different SMs for execution, synchronization is only supported between the threads inside the block. When SM executes this barrier instruction, additional delay occurs, because the next instruction cannot be executed until all the threads inside the block reach this barrier point.

Figure 13 illustrates the additional delay effect. Surprisingly, the delay is less than one memory-waiting period. NpWB, introduced in Equation (25), is the number of parallel warps per block. This term is used instead of MWP since only the warps inside the block are synchronized.

$$\begin{aligned} Synch\_cost = {} & Departure\_delay \times (NpWB - 1) \times \#synch\_insts \\ & \times \#Active\_blocks\_per\_SM \times \#Rep \end{aligned} \tag{25}$$

$$NpWB = MIN(MWP,\ \#Active\_warps\_per\_block) \tag{26}$$

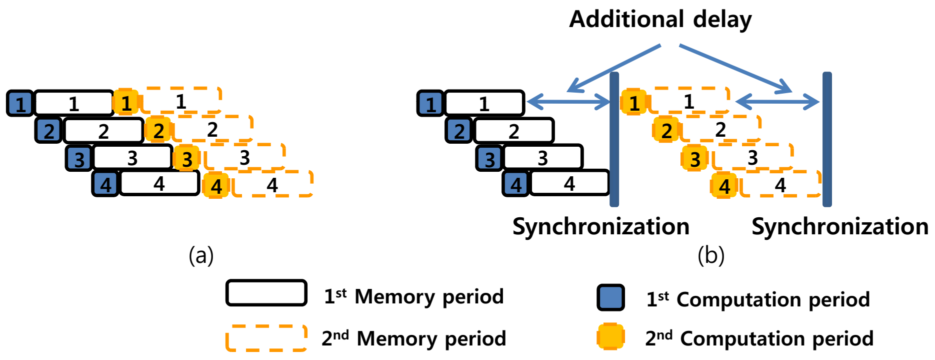


Figure 13: Delays: (a) no synchronization, (b) synchronization.

$$Exec\_cycles\_with\_synch = Exec\_cycles\_app + Synch\_cost \tag{27}$$

The additional delay-per-synchronization instruction is the multiple of *Departure_delay*, (NpWB-1), and the number of blocks. The final execution cycles of an application with synchronization are calculated by Equation (27).

2.4 Experimental Methodology

2.4.1 List of GPU Architectures Used for Evaluation

Table 2 shows the list of GPUs used for the experiment. GTX280 supports 64-bit floating-point operations, and it has a hardware version of 1.3, which improves uncoalesced memory accesses. To measure the GPU kernel execution time, cudaEventRecord function is used, which uses GPU shader-clock cycles. All the measured execution time is the average of ten runs. The benchmarks are compiled with NVCC [77] compiler.

Table 2: The specifications of GPUs used in the experiment.

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	[33]
#Coal_per_mw	1	1	1	1

2.4.2 Designing Micro Benchmark

To test the analytical model and find memory model parameters, a set of Micro benchmarks is designed. The number of load instructions and computation instructions per loop is varied. Each benchmark has two memory-access patterns: coalesced and uncoalesced. Table 3 summarizes the list of micro-benchmarks and shows the number of memory and computation instructions per warp. The numbers inside the parentheses show the number of floating-point instructions. For example, Mb1 has no memory instructions, while Mb7 has six memory instructions.

Table 3: 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)

2.4.3 Merge Benchmark

To test how the analytical model can predict typical GPGPU applications, six benchmarks in the Merge work [62] are used. Table 4 describes each benchmark and summarizes the characteristics. 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 found in PTX code.

Table 4: Characteristics of merge benchmarks.

Benchmark	**Description**	**Input size**	**Comp Insts**	**Reg**	**Shared Memory**	**Arith. intensity**
Sepia [62]	Filter for artificially aging images	7000 x 7000	71	7	52B	11.8
Linear [62]	Image filter for computing 9-pixels avg.	10000 x 10000	111	15	60B	3.7
SVM [62]	Kernel from a SVM-based algorithm	736 x 992	10871	9	44B	13.3
Mat. (naive)	Naive version of matrix multiplication	2000 x 2000	12043	10	88B	3
Mat. (tiled) [77]	Tiled version of matrix multiplication	2000 x 2000	9780 - 24580	18	3960B	48.7
Blackscholes [77]	European option pricing	9000000	137	11	36B	19

2.5 Results

2.5.1 Calculating Memory Model Parameters

Micro benchmarks are used to measure the parameters that are required to model the memory system. *Mem_LD*, *Departure_del_uncoal*, and *Departure_del_coal* parameters are varied to find the best-fitting values.

Table 5 summarizes the results. FX5600, 8800GTX, and 8800GT use the same parameters. *Departure_del_coal* is related to the memory-access time to a single memory transaction. *Departure_del_uncoal* for GTX280 is larger than that of FX5600. This is because there are more processing cycles associated with minimizing the number of memory transactions. The performance benefit of having fewer memory transactions is greater than having a higher departure delay.

Table 5: Results of the memory-model parameters.

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

Using the parameters in Table 5, CPI values for the micro-benchmarks are calculated. Figure 14 shows the average CPI for both the measured values and the predicted values. The results show that the average geometric mean of the error is 5.4%. As the number of load instructions increases, the CPI increases. For the coalesced cases of Mb1_C to Mb7_C, the cost of load instructions is almost hidden because of the high MWP. However, for the uncoalesced cases of Mb1_UC to Mb7_UC, the cost of load instructions linearly increases with the number of load instructions.

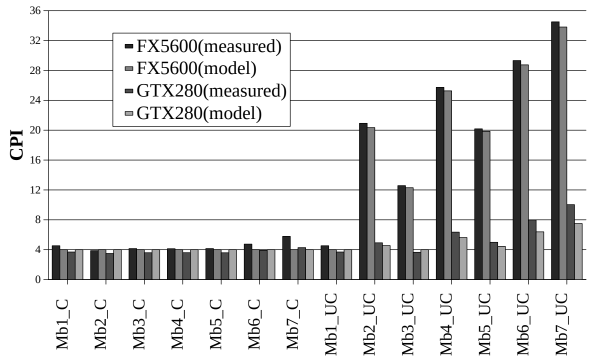


Figure 14: CPI of the micro benchmark.

2.5.2 Evaluation of Merge Benchmark

Figure 15 and 16 show the measured and estimated execution times of the Merge benchmark on the FX5600 and GTX280. The number of threads per block is changed from four to 512. The number 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 number of data elements. In other words, if the number of threads in a block is increased, the total number of blocks is reduced to make the total amount of work the same. Hence, execution times are mostly the same.

Figure 17 shows the measured and estimated CPI values across four GPUs. CPI shows more information than execution time since CPI is also a performance metric. For example, a CPI value that is close to four means that application is reaching the peak performance, because four cycles are needed to issue one instruction to a warp.

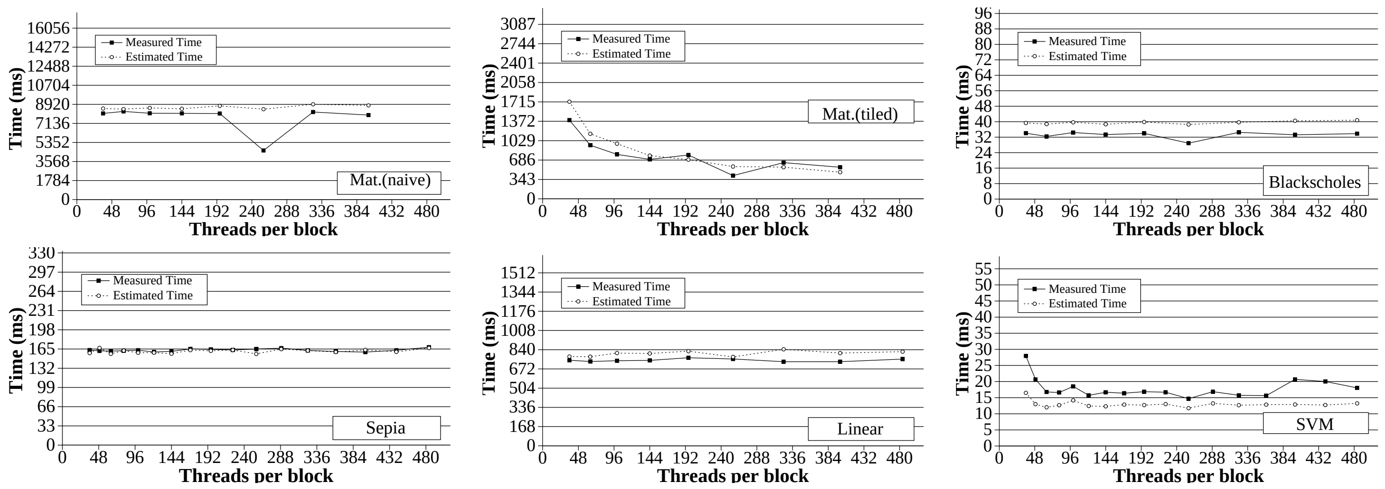


Figure 15: Execution time of the merge benchmarks on FX5600.

The average values of CWP and MWP per SM are shown in Figures 18 and 19, respectively. Compared to other GPUs, 8800GT has the least amount of bandwidth, resulting in the highest CPI in contrast to GTX280. Generally, higher arithmetic intensity means lower CPI (i.e., 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 the highest MWP and the lowest CPI values, because only SVM has fully coalesced-memory accesses.

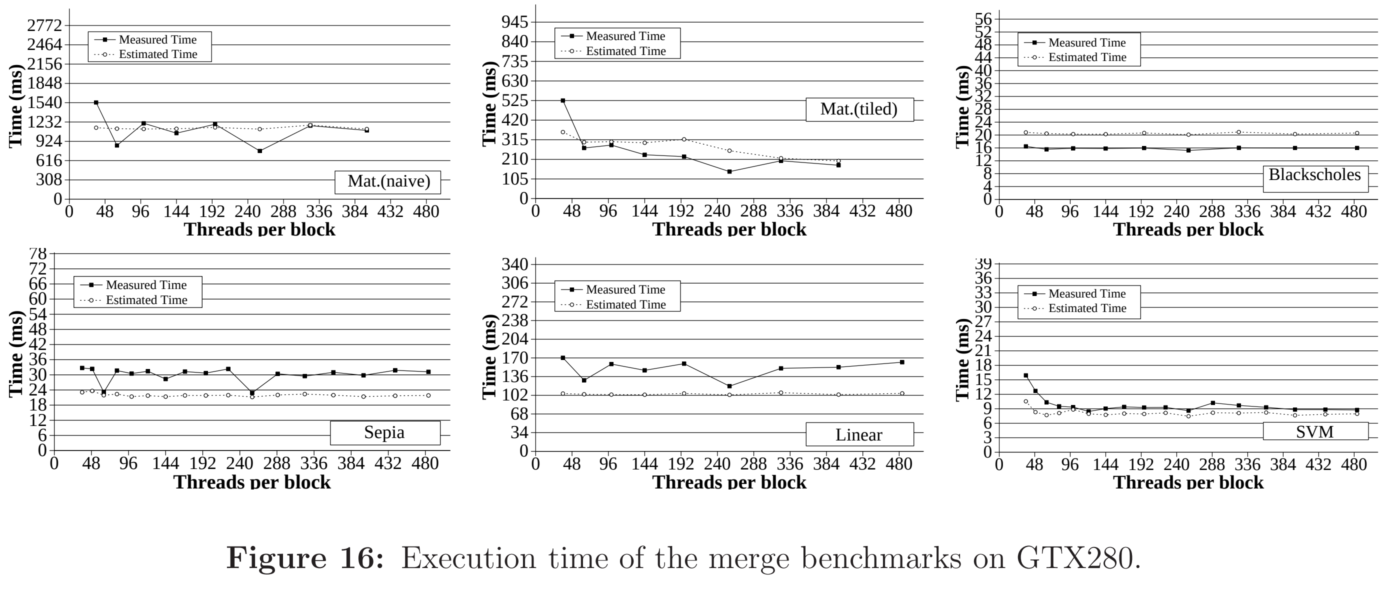


Figure 16: Execution time of the merge benchmarks on GTX280.

The MWP values in GTX280 are higher than the other GPUs, because even though the most memory requests are not fully coalesced, the number of memory transactions is optimized, which results in higher MWP. All other benchmarks are limited by *departure_delay*, which is also the reason why these applications never reach the peak bandwidth.

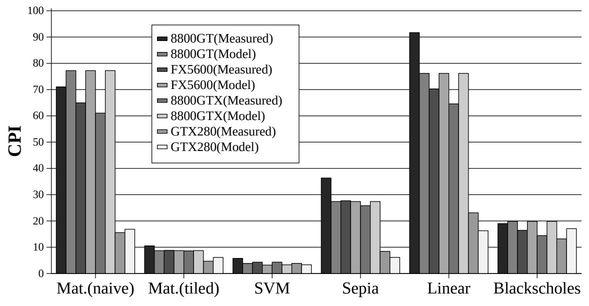


Figure 17: CPI of the merge benchmarks.

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

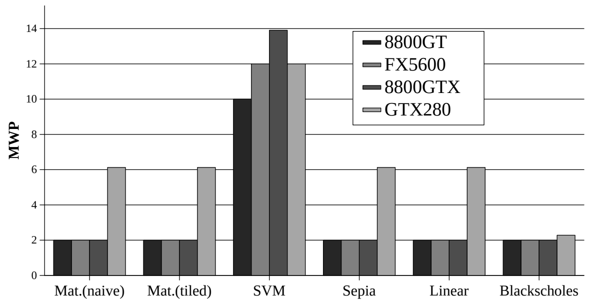


Figure 18: MWP per SM of the merge benchmarks.

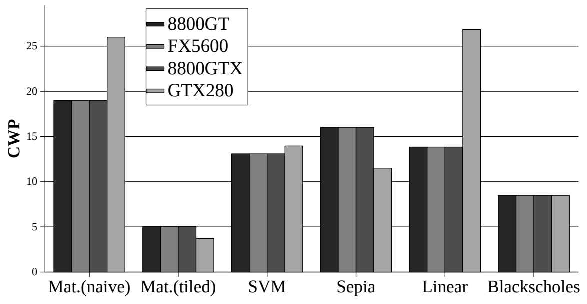


Figure 19: CWP per SM of the merge benchmarks.

The geometric mean of the estimated CPI error on the Merge benchmark is 13.3%, as shown in Figure 17. Generally, the error is higher for GTX280 than the rest of evaluated GPUs, because the number of memory transactions is difficult to predict due to optimizations by hardware.

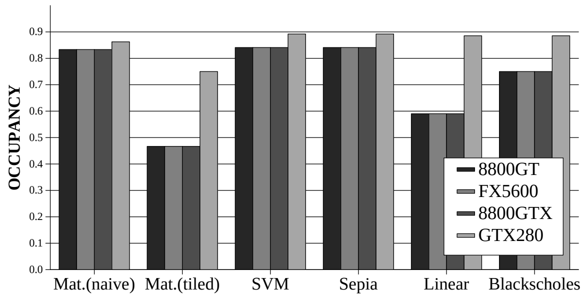


Figure 20: Occupancy of the merge benchmarks.

2.6 More Validations and Insights

In this section, to further validate the analytical model, we implemented the instruction analyzer used by the GPU emulator (Ocelot [17]) for finding the dynamic number of instructions. Also, to give further insights, we discuss in more detail topics such as the effects of independent and dependent memory accesses, long-latency computations, divergent warp execution, and synchronization effects.

2.6.1 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 the three components, we vary the number of warps and plot the corresponding MWP values in Figure 21 and 22.

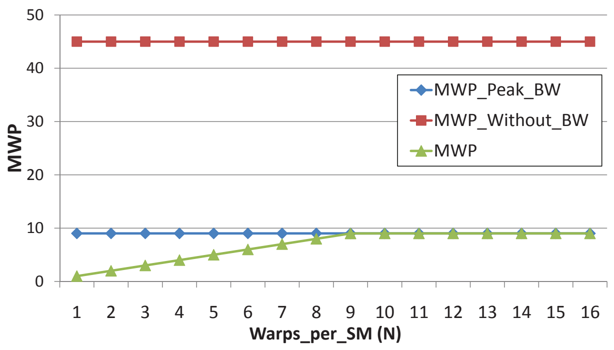


Figure 21: MWP analysis: coalesced memory accesses.

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 up to 9 for coalesced case but only up to 5 for the uncoalesced case.

Now, to provide insights into the analytical model, we revisit the example in Section 2.2. Figures 23 and 24 show N, MWP_without_BW, MWP_peak_BW, MWP,

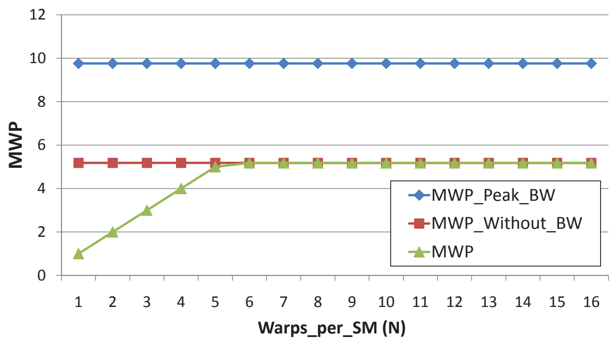


Figure 22: Visualization of MWP (Top: coalesced case, Bottom: uncoalesced case).

and CWP for the `Constant+Optimized` case and `Naive` case from Figure 4, respectively. Here, we explain the performance behavior with MWP_peak_BW and MWP_Without_BW instead of MWP because the final MWP is the minimum of those two terms and the number of running warps (N), as shown in Equation (5). The limiting term for Figure 23 is 12 (MWP_peak_BW), and it is 2 (MWP_Without_BW) for Figure 24.

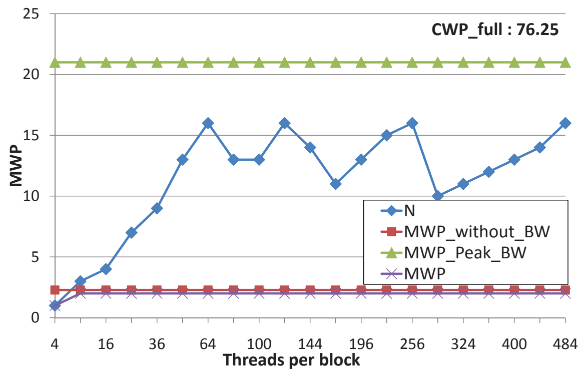


Figure 23: MWP, CWP analysis on the optimized SVM.

The main reason for 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.

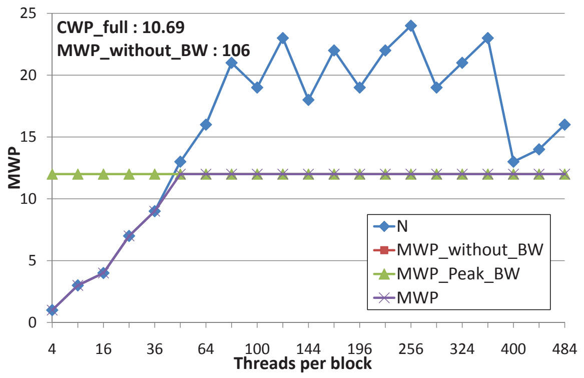


Figure 24: MWP, CWP analysis on the naive SVM.

2.6.2 Instruction Analyzer

To validate and facilitate the process of obtaining model inputs such as the number of dynamic instructions, the instruction analyzer for Ocelot is implemented. When Ocelot emulates a CUDA code on a CPU, our tool analyzes the execution, and produces outputs (Instructions, Threads/Blocks, Occupancy, etc) that can be passed to the analytical model.

Figure 25 shows the output when the tool is used on the Merge benchmark suite. The y-axis shows the error ratio between the hand analysis and the Ocelot execution. The number beside the bar shows the absolute difference in number of instructions.

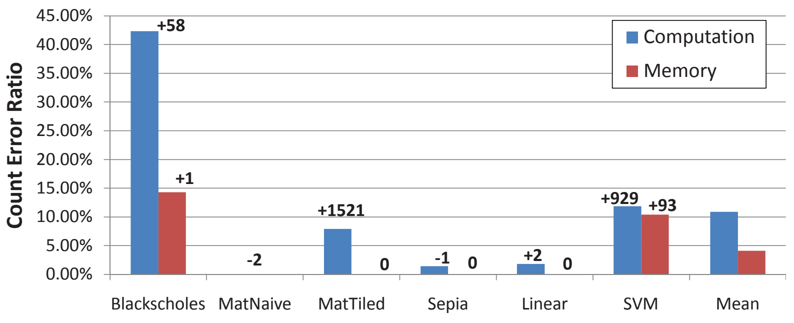


Figure 25: Instruction count comparison.

The figure shows that the average error is 10.9% for computation instructions and 4.12% for memory instructions. Note that memory accesses have much more impact on performance than the number of computations.

2.6.3 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 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 26 illustrates both cases (dependent and independent instructions).

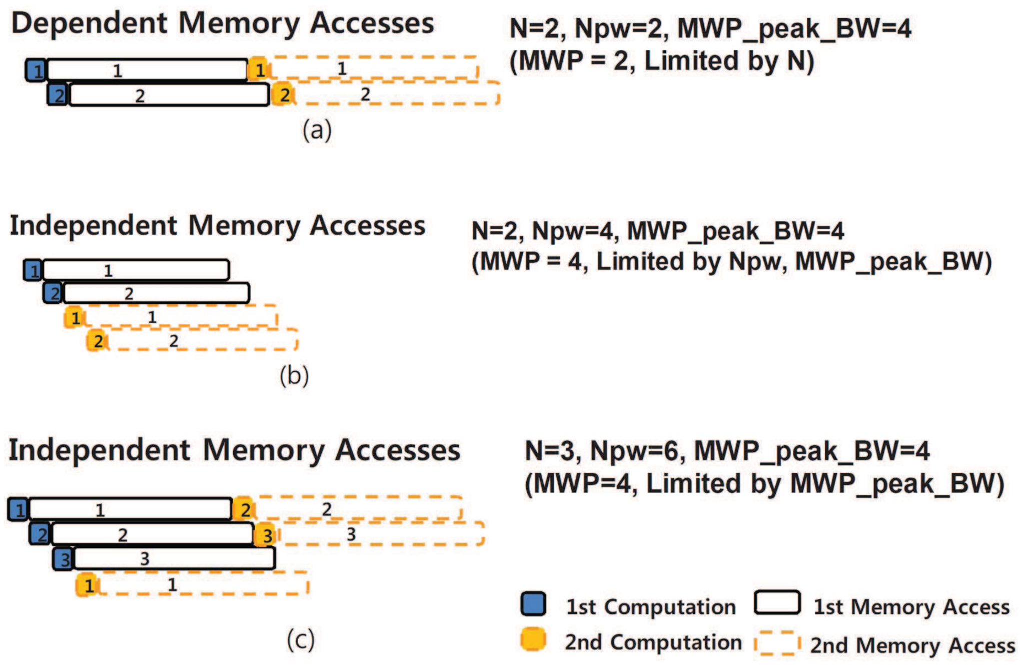

Dependent Memory Accesses
N=2, Npw=2, MWP_peak_BW=4
(MWP = 2, Limited by N)
(a)
Independent Memory Accesses
N=2, Npw=4, MWP_peak_BW=4
(MWP = 4, Limited by Npw, MWP_peak_BW)
(b)
Independent Memory Accesses
N=3, Npw=6, MWP_peak_BW=4
(MWP=4, Limited by MWP_peak_BW)
(c)
1st Computation
2nd Computation
1st Memory Access
2nd Memory Access

Figure 26: Illustration of dependent and independent memory accesses.

The numbers inside the computation and memory periods indicate the warp identification numbers, representing two to three warps per SM. In the dependent case, two series of memory operations from the same warp are serialized (Part (a)). But in the independent case, since memory operations from the same warp can be serviced concurrently, they are all parallelized as if there were four warps per SM rather than two (Part (b)). Part (c) shows that there is one more warp than in Part (b) and memory requests are independent. However, since the GPU system has limited bandwidth, not all memory requests overlap. More detail is discussed with Figure 27 and 28 below.

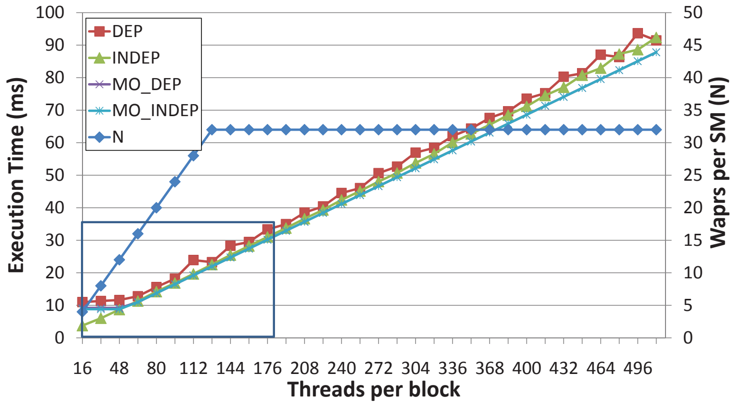


Figure 27: Effects of dependent and independent memory accesses.

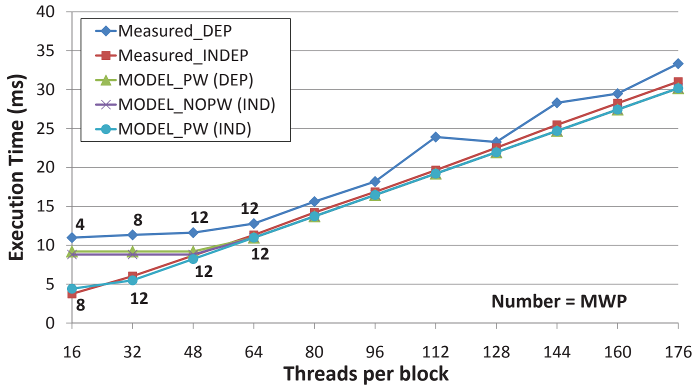


Figure 28: Model prediction of dependent and independent memory accesses.

To evaluate the effects of dependent/independent memory accesses on actual performance, we design micro-benchmarks, where one benchmark is dependent on the previous value of the memory load (DEP), while the other is not (INDEP). Both cases have the same number of instructions and instruction mixtures. Figure 27 shows the execution time of two cases as we increase the number of warps (i.e., all the threads execute the same code, so the total amount of work is also increased). When the number of warps 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 have a similar execution time. The main reason is that when there are fewer warps per SM and the following memory requests are independent, those requests can be processed together, as shown in Figure 26b, thereby increasing effective memory-level parallelism. Note that although only four warps are allocated per SM in Figure 28(b), for independent memory accesses, the effective number of warps (N_{pw}) is eight. But as more warps are allocated beyond MWP, SM can find available warps regardless of memory type; the performance between the two is similar, which is represented by Figure 26C.

In our analytical model, we assume that all instructions within a warp are dependent on the previous instructions. However, for memory requests, that will result in a serialization of all memory requests in one warp. Therefore, the N_{pw} term in Equation 28, which represents the effective parallel number of warps, is used. Typically, the number of parallel warps is the same as the total number of warps per SM (N). However, when there are independent memory requests and few warps, more warps (i.e., more memory requests) can be executed in parallel. Therefore, the N term should be replaced by N_{pw} for a more thorough analysis when the number of warps is less than MWP. We calculate this effective number of parallel warps by finding the number of memory-independent requests.

$$N_{pw} = N \times \frac{\#Ind\_mem\_req}{(\#Mem\_req - \#Ind\_mem\_req)} \tag{28}$$

$$MWP = MIN(MWP\_Without\_BW, MWP\_peak\_BW, N_{pw}) \tag{29}$$

The N_{pw} term is only used for calculating effective MWP. As shown in Equation (29), the term, 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 27. This is the same case where there are not enough running warps. We believe that this example shows the interrelated effects among the number of allocated warps per SM, types of memory accesses, and MWP clearly.

Figure 28 shows the outcome of three 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). Figure 28 is an enhanced version of the boxed area in Figure 27. 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 that 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 keeping total memory operations the same. The results show that the predicted execution time using Figure 28(MODEL_PW) estimates the execution time precisely for these two cases but not with the old model (MODEL_NOPW).

2.6.4 Long-Latency Computation Instructions

In our analytical model, we apply different instruction latencies based on the instruction types. Table 6 summarizes the throughput of instructions based on the CUDA manual and our experimental measurement. A throughput of one means that each functional unit can finish one operation per cycle, which results in eight operations (ops)/cycle since there are eight processing units per SM. [6] M_Factor is modeled as one when the throughput is eight ops/cycle and it is proportionately increased (more cycles) as the instruction throughput is decreased.

Table 6: Instruction throughput.

Instructions	**Ops/cycle (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	Costly	30, 35

Floating-point (FP) operations have the maximum throughput compared to INT operations in the evaluated GPU architectures. For FP instructions, the output is generated for every cycle (pipelined). However, the throughtput for INT operations is lower (less than 8 ops/s). We believe that INT operations are translated to multiple binary instructions at run-time (detailed information is not publicly available). Therefore, effective ops/cycle is less than 8. Hence, we model this effective throughput degradation by using the M_Factor term. The throughput for FP operations such as addition, multiplication, and multiply-addition are 8 ops/cycle. However, instructions such as modulo and integer multiplication take much longer latency, reducing the throughput by the factors of 4.3 and 35, respectively.

$$Comp\_cycles = (\#Issue\_cycles \times M\_Factor) \times \#total\_insts \tag{30}$$

Equation (30) shows the improved calculation for computation cycles over the previous Equation (18)

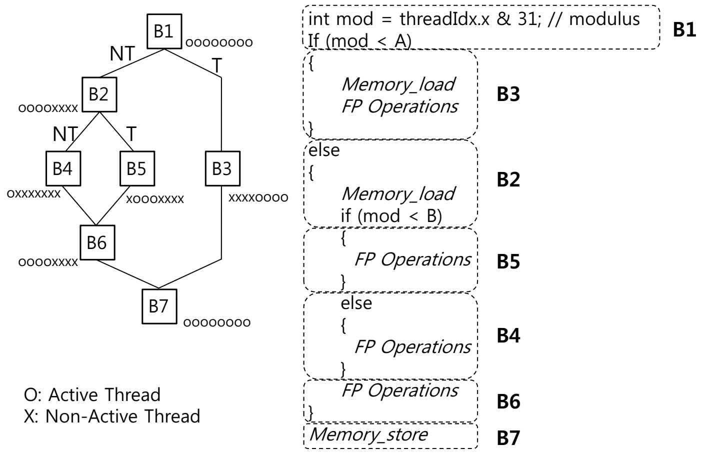
2.6.5 Divergent branches

When a warp diverges (i.e., diverges within 32 threads), the execution of diverged warps is serialized [77]. [7] This means that while one path is executed, the threads on

[6]Ops/cycle is used in the CUDA manual. M_Factor is introduced to to proportionately model long-latency instructions.

[7]Several recent studies have focused on reducing unnecessary idle cycles during divergent execution [25, 95].

the other path are idle. Figure 29 shows an example. The branch at basic block 1 in the figure diverges. An active bitmap mask shows that the 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.



```
int mod = threadIdx.x & 31; // modulus
If (mod < A)
{
    Memory_load
    FP Operations
}
else
{
    Memory_load
    if (mod < B)
    {
      FP Operations
    }
    else
    {
      FP Operations
    }
    FP Operations
}
Memory_store
```

Figure 29: Illustration of a divergent execution.

Figure 30 shows the model predictions and the measured execution time. If the model only takes the 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 [25]. The *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.

2.6.6 Effects of Synchronization

The cost of synchronization is modeled in Section 2.3.9 using Equation (25) and Equation (27). To evaluate the synchronization cost in more detail, we compare the performance delta between two programs in Figure 31, where the only difference is the barrier instruction (bar.sync).

Figure 32 shows an experiment where only one SM is active (i.e., one block is

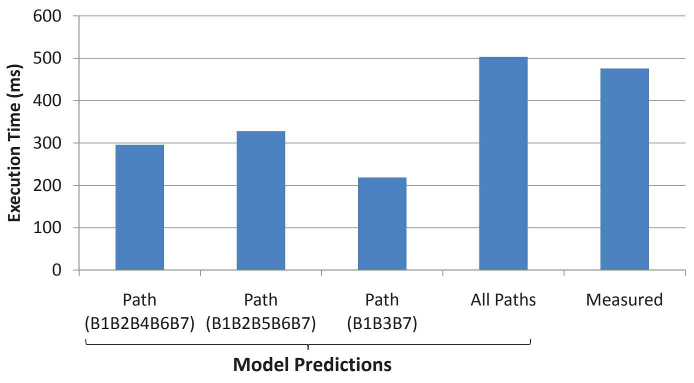


Figure 30: Effects of divergent branches on the execution time.

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

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

Figure 31: PTX code for synchronization analysis.

used). When there is only one warp, there should be 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 one warp. As 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 for observing the cost.

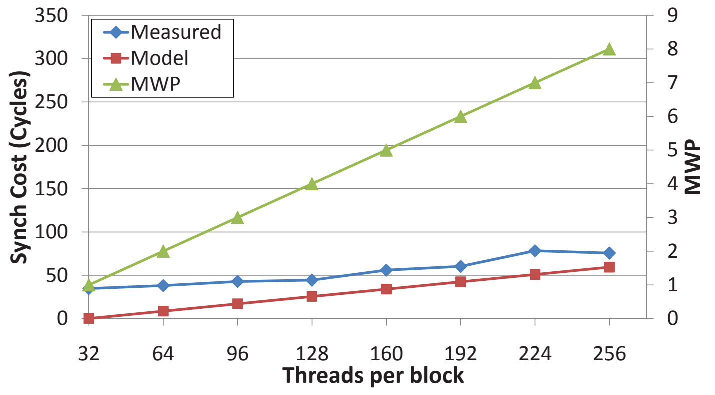


Figure 32: Synch delay: one block (one SM active).

Figures 33 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 the 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 2.3.9, with respect to overall performance, synchronization delay cycles are not significant.

2.6.7 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 a 100% cache hit ratio in most GPGPU applications. The current G80 architecture does not have a hardware cache for the global memory. Typical stream applications running on 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

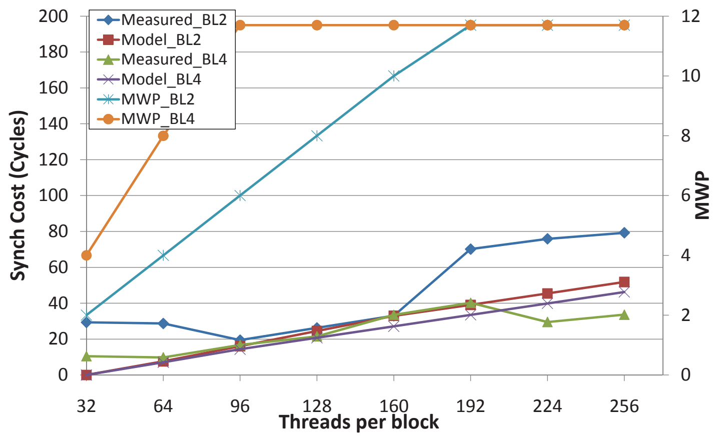


Figure 33: Synch delay: two vs. four blocks allocated per SM.

should include a cache model. In future work, we will include cache models.

2.7 Summary

This chapter 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 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 or not they should perform certain optimizations. 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 programming.

CHAPTER III

MODELING GPU PERFORMANCE AND POWER

The number of cores inside the chip, especially in GPUs, is increasing dramatically. For example, GTX280 has 30 streaming multi-processors (SM) with 240 cores, and the next generation GPU will have 512 cores [76]. Even though GPU applications are highly throughput-oriented, not all applications require all available cores to achieve the best performance. This study focuses on an important issue of saving energy on many-core architecture. The issue is whether or not all available cores need to activated every time an application is executed.

Figure 34 shows performance, power consumption, and energy efficiency as the number of active cores is changed,[1] where energy efficiency is obtained by dividing performance by power. The power consumption increases as the number of cores is increased. Depending on the circuit design, the gradient of an increase in power consumption also changes.

Two different types of applications are shown in Figure 34. In Type 1, the performance increases linearly, because applications can utilize all the computing powers in the cores. However, in Type 2, the performance is saturated after a certain number of active cores as a result of bandwidth limitations [93, 103]. Once the number of memory requests from cores exceeds the peak memory bandwidth, increasing the number of cores does not lead to a better performance. For this work, the number of cores that shows the highest performance per watt is called the optimal number of cores.

In Type 2, since the performance does not increase linearly, using all the cores

[1]Active cores mean the cores that are executing a program.

consumes more energy than using the optimal number of cores. However, for application Type 1, utilizing all the cores consumes the least amount of energy because of a reduction in execution time. The optimal number of cores for Type 1 is the maximum number of available cores, but that of Type 2 is less than the maximum value. Hence, if optimal number of cores can be predicted at static time, then either the compiler or the programmer can configure the number of threads and blocks to utilize fewer cores, or a dynamic thread manager can achieve the same task.

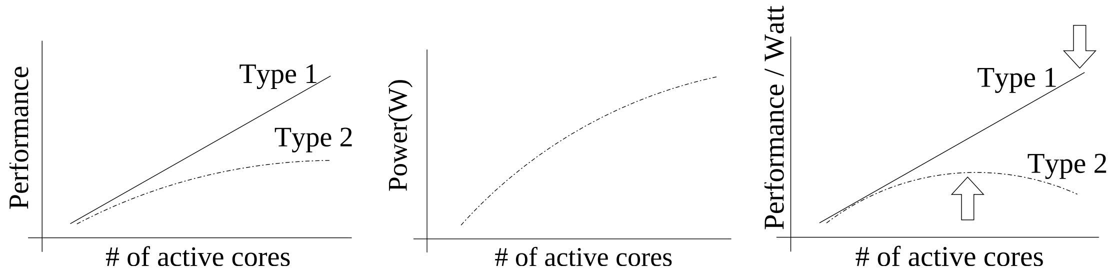


Figure 34: Performance, power, and performance/watt vs. active cores.

To achieve this goal, an integrated power and performance (IPP) system is proposed. Figure 35 shows an overview of the IPP system. It takes a GPU kernel as an input and predicts both power consumption and performance together, whereas previous models predict only the execution time or power consumption separately. Moreover, IPP does not require architectural timing simulations or hardware performance counters; instead IPP uses the outcomes of a timing model.

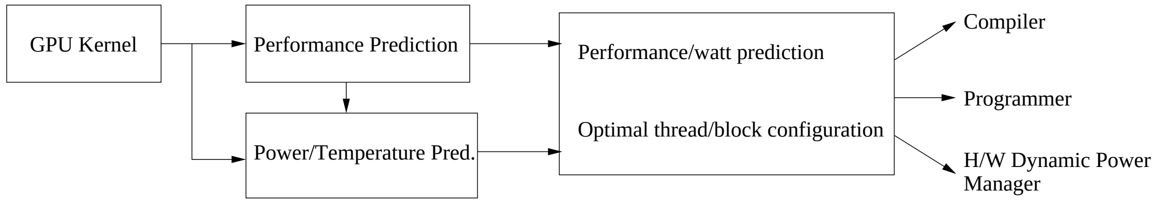

GPU Kernel
Performance Prediction
Power/Temperature Pred.
Performance/watt prediction
Optimal thread/block configuration
Compiler
Programmer
H/W Dynamic Power Manager

Figure 35: Overview of the IPP system.

Using the power and performance outcomes, IPP predicts the optimal number of cores that results in the highest performance per watt. Unlike previous works, IPP demonstrates energy savings in a real GPU system. The results show that by using fewer cores based on the IPP prediction, we can save up to 22.09% and on average 10.99% of run-time energy consumption for the five memory bandwidth-limited benchmarks. Furthermore, the amount of energy savings for GPUs that employ a power-gating technique is predicted. The evaluations show that with power-gating, IPP can save 25.85% of the total energy consumption for the five bandwidth-limited benchmarks.

3.1 Background on Power

Power consumption can be divided into two parts: dynamic power and static power, as shown in Equation (31).

$$Power = Dynamic\_power + Static\_power \tag{31}$$

Dynamic power is the switching overhead in transistors, so it is determined by run-time events. Static power is mainly determined by circuit technology, chip layout, and operating temperature.

3.1.1 Building a Power Model Using an Empirical Method

Isci and Martonosi [44] proposed an empirical method to building a power model. They measured and modeled the Intel Pentium 4 processor. Equation (32) shows the basic power model discussed in [44]. It consists of the idle power plus the dynamic power for each hardware component, where the $MaxPower$ and $ArchitecturalScaling$ terms are heuristically determined. For example, $MaxPower$ is empirically determined by running several training benchmarks that stress fewer architectural components. Access rates are obtained from performance counters. They indicate how often an architectural unit is accessed per unit of time, where one is the maximum value.

$$Power = \sum_{i=0}^{n} (AccessRate(C_i) \times ArchitecturalScaling(C_i) \times MaxPower(C_i) \tag{32}$$
$$+ NonGatedClockPower(C_i)) + IdlePower$$

3.1.2 Static Power

As the transistor technology is scaled down, static power consumption is increased [8]. To understand static power consumption and temperature effects, static power models are explained. Butts and Sohi [12] presented the following simplified leakage-power model for an architecture-level study, as shown in Equation (33).

$$P_{static} = V_{cc} \cdot N \cdot K_{design} \cdot \hat{I}_{leak} \tag{33}$$

V_{cc} is the supply voltage, N is the number of transistors in the design, and K_{design} is a constant factor that represents technology characteristics. $\hat{I}_{leak}$ is a normalized leakage current for a single transistor that depends on V_{th}, which is the threshold voltage. Later, Zhang et al. [104] improved this static-power model to consider temperature effects and operating voltages in HotLeakage software tool. In their model, K_{design} is no longer a constant. It depends on temperature, where $\hat{I}_{leak}$ is a function of temperature and supply voltage. The leakage current can be expressed, as shown in Equation (34).

$$\hat{I}_{leak} = \mu_0 \cdot C_{OX} \cdot \frac{W}{L} \cdot e^{b(V_{dd}-V_{dd0})} \cdot v_t^2 \cdot (1 - e^{\frac{-V_{dd}}{v_t}}) \cdot e^{\frac{-|V_{th}|-V_{off}}{n \cdot v_t}} \tag{34}$$

v_t is the thermal voltage that is represented by kT/q, and it depends on temperature. The threshold voltage, V_{th}, is also a function of temperature. Since v_t^2 is the dominant temperature-dependent factor in Equation (34), the leakage power quadratically increases with temperature. However, in a normal operating temperature range, the leakage power can be simplified as a linear model [92].

3.2 Power and Temperature Models

3.2.1 Overall Model

GPU power consumption is modeled in Equation (32) [44]. GPU_power term consists of $Runtime\_power$ and $IdlePower$ terms, as shown in Equation (35). The $NonGatedClockPower$ term is not used in this model, because the evaluated GPUs do not employ clock-gating technique. $IdlePower$ is the power consumption when a GPU is turned on, but no application is running. $Runtime\_power$ is the additional power consumption, which is the sum of run-time powers from all active SMs and GPU DRAM, required to execute an application. The additional power from SMs is modeled by RP_SMs, and DRAM is modeled by RP_Memory term, as shown in Equation (36).

$$GPU\_power = Runtime\_power + IdlePower \tag{35}$$

$$\begin{aligned} Runtime\_power &= \sum_{i=0}^{n} RP\_Component_i \\ &= RP\_SMs + RP\_Memory \end{aligned} \tag{36}$$

3.2.2 Modeling Power for Streaming Multi-processors

To model the run-time power of SM, SM is decomposed into several physical components, as shown in Equation (37) and Table 7. The texture and constant caches are included in the $SM\_Component$ term, because they are shared between multiple SMs in the evaluated GPU system. One texture cache is shared by three SMs, and each SM has its own constant cache. RP_Const_SM is a constant-runtime power component for each SM. It models power consumption from several units, including I-cache and the frame buffer, which always consume relatively a constant amount of power when a core is active. A cumulative power from multiple SMs is obtained by

Table 7: List of instructions that access each architectural unit.

PTX Instruction	**Architectural Unit**	**Variable Name**
add_int sub_int addc_int subc_int sad_int div_int rem_int abs_int mul_int mad_int mul24_int mad24_int min_int neg_int	Int. arithmetic unit	RP_Int
add_fp sub_fp mul_fp fma_fp neg_fp min_fp lg2_fp ex2_fp mad_fp div_fp abs_fp	Floating point unit	RP_Fp
sin_fp cos_fp rcp_fp sqrt_fp rsqrt_fp	SFU	RP_Sfu
xor cnot shl shr mov cvt set setp selp slct and or	ALU	RP_Alu
st_global ld.global	Global memory	RP_GlobalMem
st_local ld.local	Local memory	RP_LocalMem
tex	Texture cache	RP_Texture_Cache
ld_const	Constant cache	RP_Const_Cache
ld_shared st_shared	Shared memory	RP_Shared
setp selp slct and or xor shr mov cvt st_global ld_global ld_const add mad24 sad div rem abs neg shl min sin cos rcp sqrt rsqrt set mul24 sub addc subc mul mad cnot ld_shared st_local ld_local tex	Register file	RP_Reg
All instructions	FDS (Fetch/Decode/Schedule)	RP_FDS

Equation (38) as a simple summation.

$$\sum_{i=0}^{n} SM\_Component_i = RP\_Int + RP\_Fp + RP\_Sfu + RP\_Alu + RP\_Texture\_Cache + RP\_Const\_Cache + RP\_Shared + RP\_Reg + RP\_FDS + RP\_Const\_SM \tag{37}$$

$$RP\_SMs = Num\_SMs \times \sum_{i=0}^{n} SM\_Component_i \tag{38}$$

Table 7 summarizes the modeled architectural components used by each instruction type and the corresponding variable names in Equation (37). All instructions access the fetch decode schedule (FDS) unit. For the register unit, all instructions accessing the register file are assumed to have the same number of register operands per instruction. The exact number of register accesses per instruction depends on the instruction type and the number of operands, but we found that the power consumption with respect to number of register operands is negligible.

As Equation (32) shows, dynamic-power consumption is dependent on the access rate of each hardware component. Isci and Martonosi used a combination of hardware performance counters to measure access rates [44]. Since GPUs do not have any speculative execution, access rates based on the dynamic number of instructions and execution times can be estimated without using the hardware performance counters.

Equation (39) shows how to calculate RP_{comp}, the run-time power for each architectural component such as RP_Reg for register unit. RP_{comp} is the multiplication of $AccessRate_{comp}$ and $MaxPower_{comp}$. $MaxPower_{comp}$ is described in Table 8 and discussed in Section 3.2.4. Note that RP_Const_SM is not dependent on $AccessRate_{comp}$.

Equation (40) shows how to calculate an access rate for each component. The dynamic number of instructions per component, $DAC\_per\_th_{comp}$, is the sum of instructions that access an architectural component, as shown in Equation (41). Equation (42) shows the term, $Warps\_per\_SM$, which indicates how many warps are executed in one SM. The execution cycles are divided by four, because one instruction is fetched, scheduled, and executed every four cycles. This normalization also makes the maximum value of the $AccessRate_{comp}$ term to be one.

$$RP_{comp} = MaxPower_{comp} \times AccessRate_{comp} \tag{39}$$

$$AccessRate_{comp} = \frac{DAC\_per\_th_{comp} \times Warps\_per\_SM}{Exec\_cycles/4} \tag{40}$$

$$DAC\_per\_th_{comp} = \sum_{i=0}^{n} Number\_Inst\_per\_warps_i(comp) \tag{41}$$

$$Warps\_per\_SM = \left(\frac{\#Threads\_per\_block}{\#Threads\_per\_warp} \times \frac{\#Blocks}{\#Active\_SMs} \right) \tag{42}$$

3.2.3 Modeling Power for Memory System

The evaluated GPU system has five different memory spaces: global, shared, local, texture, and constant. The shared memory space uses a software-managed cache that is inside the SM. The texture and constant memories are located in the graphics double data rate (GDDR) memory. The global and local memories share the same physical graphics memory, hence RP_Memory considers both memory spaces, as shown in Equation (43). Shared, constant, and texture memory spaces are modeled separately as SM components.

$$\begin{aligned} RP\_Memory &= \sum_{i=0}^{n} Memory\_component_i \\ &= RP\_GlobalMem + RP\_LocalMem \end{aligned} \tag{43}$$

3.2.4 Power Model Parameters

To obtain the power model parameters, a set of synthetic Micro benchmarks that stresses different architectural components in the GPU is designed. For example, the benchmark that stresses FP units contains a high ratio of FP instructions in a loop.

The optimum set of $MaxPower_{comp}$ values in Equation (39) that minimize the error between the measured power and the outcome of the equation is searched. To avoid searching through a large space of values, the initial value for each architecture unit is estimated based on the relative physical die sizes of the unit [44]. Table 8 shows the parameters used for $MaxPower_{comp}$. Eight units require a special linear approach [44]; an initial increase from idle to relatively low-access rate causes a large granularity of increase in power consumption while a further increase causes a smaller increase. The Spec.Linear column in Table 8 indicates whether the $AccessRate_{comp}$ term in Equation (39) needs to be replaced with the function of $0.1365 * ln(AccessRate_{comp}) + 1.001375$.

Figure 36 shows how the overall power is distributed among the individual architectural components for all evaluated benchmarks. Section 3.4 presents the detailed benchmark descriptions and the evaluation methodology. On average, memory, idle-power, and RP_Const_SM consume more than 60% of the total GPU power. Register file and FDS units also consume higher power than other components, because almost

Table 8: Empirical power parameters for architectural units.

Units	**MaxPower**	**OnChip**	**Spec.Linear**
FP	0.2	Yes	Yes
REG	0.3	Yes	Yes
ALU	0.2	Yes	No
SFU	0.5	Yes	No
INT	0.25	Yes	Yes
FDS (Fetch/Dec/Sch)	0.5	Yes	Yes
Shared memory	1	Yes	No
Texture cache	0.9	Yes	Yes
Constant cache	0.4	Yes	Yes
Const_SM	0.813	Yes	No
Global memory	52	No	Yes
Local memory	52	No	Yes

all instructions access these units.

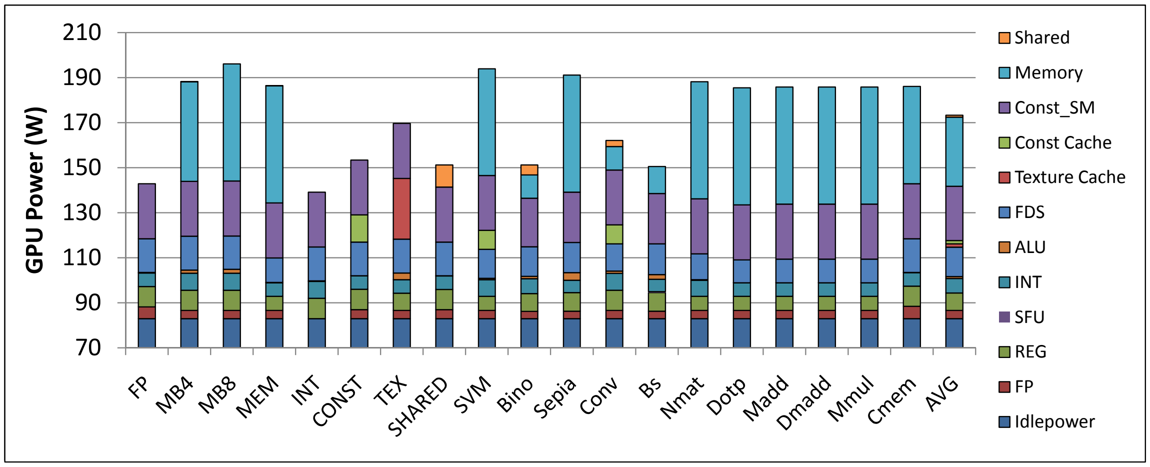


Figure 36: Power-breakdown graph for all the evaluated benchmarks.

3.2.5 Active SMs vs. Power Consumption

To measure the power consumption of each SM, another set of Micro benchmarks to control the number of active SMs is designed. The benchmarks are designed such that only one block can be executed in each SM. Thus, as the number of blocks is varied, the number of active SMs is changed, too. Even though the evaluated GPU does not employ power gating, idle SMs do not consume as much power as active SMs do as a result of low-activity factors [82] (i.e., idle SMs do not change values in circuits as often as active SMs). Hence, there are significant differences in the total power consumption depending on the number of active SMs.

Figure 37 shows an increase in power consumption as the number of active SMs is increased. The maximum delta between using only one SM versus all SMs is 37 watts. Since there is no power-gating technique employed, the power consumption does not increase linearly as the number of SMs is increased. Hence, a log-based model instead of a linear curve is used, as shown in Equation (44). The memory-power consumption is also modeled by following the log-based trend. Finally, run-time power can be modeled by taking the number of active SMs as shown in Equation (48).

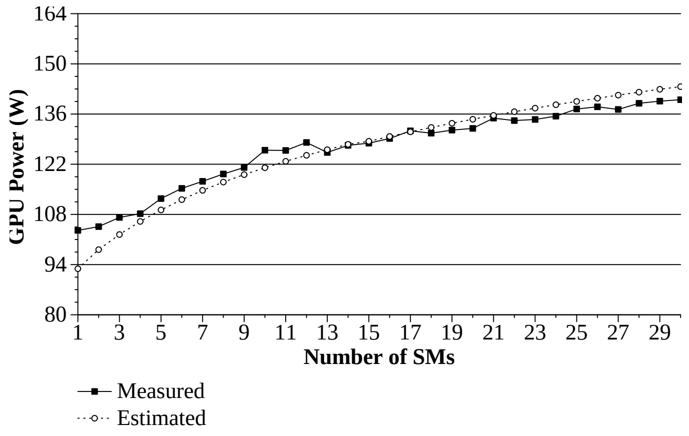


Figure 37: Power consumption vs. Active SMs.

$$RP\_SMs = Max\_SM \times log_{10}(\alpha \times Active\_SMs + \beta) \tag{44}$$

$$Max\_SM = (Num\_SMs \times \sum_{i=0}^{n} SM\_Component_i) \tag{45}$$

$$\alpha = (10 - \beta)/Num\_SMs \tag{46}$$

$$\beta = 1.1 \tag{47}$$

$$Runtime\_power = (Max\_SM + RP\_Memory) \times log_{10}(\alpha \times Active\_SMs + \beta) \tag{48}$$

3.2.6 Temperature Model

Traditionally,temperature models for CPUs have been represented by an RC model [90], where the parameters are empirically found by using a step function experiment. In this work, Equation (49) models the rising temperature, and Equation (50) models the decaying temperature.

$$Temperature_{rise}(t) = Idle\_temp + \delta\left(1 - e^{-t/RC\_Rise}\right) \tag{49}$$

$$Temperature_{decay}(t) = Idle\_temp + \gamma\left(e^{-t/RC\_Decay}\right) \tag{50}$$

$$\delta = Max\_temp - Idle\_temp \tag{51}$$

$$\gamma = Decay\_temp - Idle\_temp \tag{52}$$

Figure 38 shows the estimated and measured temperature variations. Both the chip temperature and the board temperature are measured with the built-in sensors in the GPU.

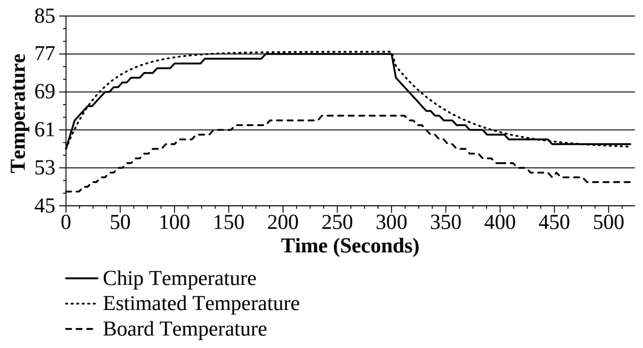


Figure 38: Modeling GPU temperature by using an RC function.

The effect of increasing temperature on power consumption is shown in Figure 39. The power data shows that with increasing temperature, the power consumption is increased. Max_temp is a function of run-time power, which depends on application characteristics. We discovered that the chip temperature is strongly affected by the rate of GDDR accesses, as modeled in Equation (53). The model parameters are summarized in Table 9.

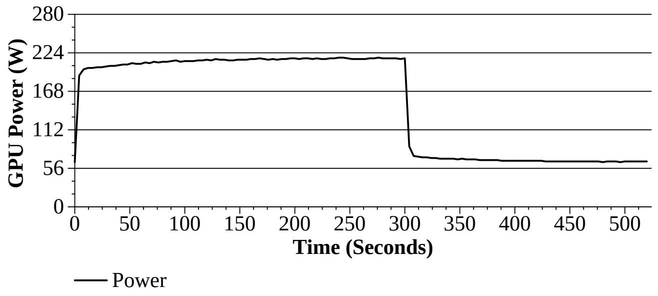


Figure 39: Effects of an increased temperature on GPU power consumption.

$$Max\_temp(Runtime\_Power) = \mu \times Runtime\_Power + \lambda + \rho \times MemAccess\_intensity \tag{53}$$

$$MemAccess\_intensity = \frac{Memory\_Insts}{NonMemory\_Insts} \tag{54}$$

Table 9: Temperature parameters for GTX280.

Parameter	**Value**
μ	0.120
λ	5.5
ρ	21.505
RC_Rise	35
RC_Decay	60

3.2.7 Modeling Increases in Static Power Consumption

Section 3.1.2 discussed the impact of temperature on static-power consumption. Because of the high number of processors in the GPU chip, we observed an increase in run-time power as the chip temperature increased, as illustrated in Figure 40. To model an increase in static-power consumption, the temperature model is included in the run-time power model, as shown in Equation (49) and Equation (50). Since we cannot control the operating voltage of the evaluated GPUs dynamically, only the operating temperature is considered.

Figure 40 shows that power increases gradually over time after the application starts[2] with the saturation power delta of 14 watts. This difference could be caused by the increase in static-power consumption or by the increase in fan power. To find the reason, the fan speed was changed from lowest to highest value. Unexpectedly, the increase of fan power was only 4 watts. Hence, the remaining 10 watts is modeled as the increase in static-power consumption. Equation (55) shows the comprehensive power equation that includes the increased static-power effect, which depends on σ, which is the ratio of power delta over temperature delta (i.e., $\sigma = 10 / 22$)). Note that $Runtime\_power_0$ is an initial power obtained from Equation (48), and the model assumes a cold start. $Temperature(t)$ in Equation (57) is obtained from Equation (49) and Equation (50).

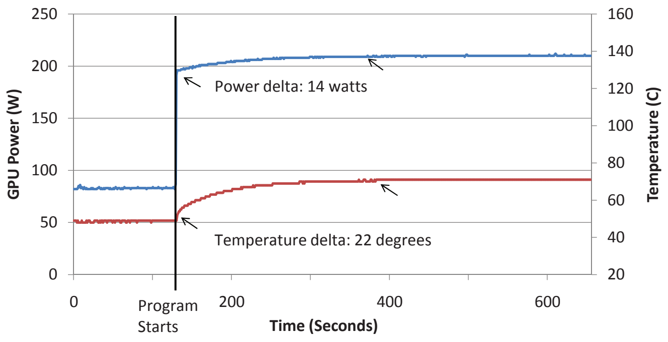


Figure 40: Effects of the static power.

$$GPU\_power(t) = Runtime\_power(t) + IdlePower \tag{55}$$

$$Runtime\_power(t) = Runtime\_power_0 + \sigma \times Delta\_temp(t) \tag{56}$$

$$Delta\_temp(t) = Temperature(t) - Idle\_temp \tag{57}$$

[2]The initial jump of power consumption exists when an application starts.

3.3 IPP: Integrated Power and Performance Model

In this section, the integrated power and performance (IPP) framework to predict performance per watt and the optimal number of active cores is discussed. IPP uses predicted execution times to estimate power consumption instead of using measured execution times.

3.3.1 Execution Time and Access Rate Prediction

In Section 3.2, the power model that computes access rates by using measured execution time information is developed. Predicting power at static time requires a knowledge of access rates in advance. In other words, execution time of an application is needed to predict power. We used a recently-developed GPU timing model [33] to predict the execution time.

In the timing model, the total execution time of an application is calculated with one of Equation (58), Equation (59), and Equation (60) based on the number of active threads, MWP, and CWP values. MWP represents the number of memory requests that can be serviced concurrently, and CWP represents the number of warps that can finish one computation-period during one memory-access period. N is the number of running warps. Mem_L is an average memory latency where a latency of 430 cycles is used for the evaluated GPU architecture. Mem_cycles is the processor-waiting cycles for memory operations. $Comp\_cycles$ is the execution time of all instructions. $Repw$

is the number of times that each SM needs to repeat the same set of computation.

Case1: If (MWP is N warps per SM) and (CWP is N warps per SM)

$$(Mem\_cycles + Comp\_cycles + \frac{Comp\_cycles}{\#Mem\_insts} \times (MWP - 1)) \times \#Repw \qquad (58)$$

Case2: If (CWP >= MWP) or (Comp_cycles > Mem_cycles)

$$(Mem\_cycles \times \frac{N}{MWP} + \frac{Comp\_cycles}{\#Mem\_insts} \times (MWP - 1)) \times \#Repw \qquad (59)$$

Case3: If (MWP > CWP)

$$(Mem\_L + Comp\_cycles \times N) \times \#Repw \qquad (60)$$

IPP calculates $AccessRate$ by using Equation (61), where $Predicted\_Exec\_Cycles$ is calculated with one of the Equation (58),Equation (59), and Equation (60).

$$AccessRate_{comp} = \frac{DAC\_per\_th_{comp} \times Warps\_per\_SM}{Predicted\_Exec\_Cycles/4} \qquad (61)$$

3.3.2 Optimal Number of Cores for Highest Energy Efficiency

IPP predicts the optimal number of SMs that achieve the highest performance per watt. As Figure 34 shows, the performance of an application can increase in two ways: linear curve, and non-linear curve. For the linear case, the optimal number of SMs is always the maximum number of cores. However, for the non-linear case, the optimal number of SMs is less than the maximum number of cores. Hence, performance per watt (Perf/W) can be calculated by using Equation (62).

$$Perf/W = \frac{work/execution\ time(\#cores)}{power(\#cores)} \qquad (62)$$

Equation (58),Equation (59), and Equation (60) calculate execution times. Among the three cases, only Case 2 has a memory bandwidth-limited case. Case 1 is used when there are not enough number of running threads in the system, and Case 3 models when an application is computationally intensive. So both Cases 1 and 3 would never reach the peak memory bandwidth.

Once MWP_peak_BW reaches N, the application usually reaches the peak bandwidth. Hence, based on Equation (6), the optimal number of cores can be calculated by using the following equations to simplify the calculation.

$$\begin{aligned}
&\text{if (1) (MWP == N) or (CWP == N) or} \qquad (63)\\
&\text{(2) MWP} > \text{CWP or}\\
&\text{(3) MWP} < \text{MWP\_peak\_BW}\\
&\qquad Optimal\ \#\ of\ cores = Maximum\ available\ \#\ of\ cores\\
&\text{else}\\
&\qquad Optimal\ \#\ of\ cores = \frac{Mem\_Bandwidth}{(BW\_per\_warp) \times N}
\end{aligned}$$

3.4 Methodology

3.4.1 Power and Temperature Measurement

GTX280 GPU, which has 30 SMs and uses a 65 nm technology, is used in the evaluation. Extech 380801 power analyzer to measure the overall system power consumption. The raw data is sent to a data-log machine every 0.5 second. Each Micro benchmark executes for an average of 10 seconds.

Since the input power to the entire system is measured, $Idlepower\_System$ of 159 watts, is subtracted from the system input power to obtain GPU_Power.[3] Hence, $Idle\_Power$ is 83 watts. GPU temperature is measured with the nvclock utility, which outputs board and chip temperatures. Temperature is measured every second.

To test the accuracy of the IPP system, the Merge benchmark [62, 33], five additional memory bandwidth-limited benchmarks (Nmat, Dotp, Madd, Dmadd, and Mmul), and one computation-intensive (i.e., non-memory bandwidth limited) benchmark (Cmem) are all used for evaluation. Table 4 describes each benchmark and summarizes the characteristics. To calculate the number of dynamic instructions, a

[3]IdlePower_System is obtained by measuring system power with another GPU card whose idle power is known.

GPU PTX emulator, Ocelot [49], is used.

3.5 Results

3.5.1 Evaluations of the Power Model

Figure 41 compares the predicted power consumption with the measured power for the Micro benchmarks. According to Figure 36, the global memory consumes the most amount of power. MB4, MB8, and MEM benchmarks consume much greater power than the FP benchmark, which consists of mainly floating-point instructions. Surprisingly, the benchmarks that use texture cache or constant cache also consume high power. This is because both the texture cache and the constant cache have higher $MaxPower$ than that of the FP unit. The geometric mean of the error in the power prediction for the Micro benchmark is 2.5%.

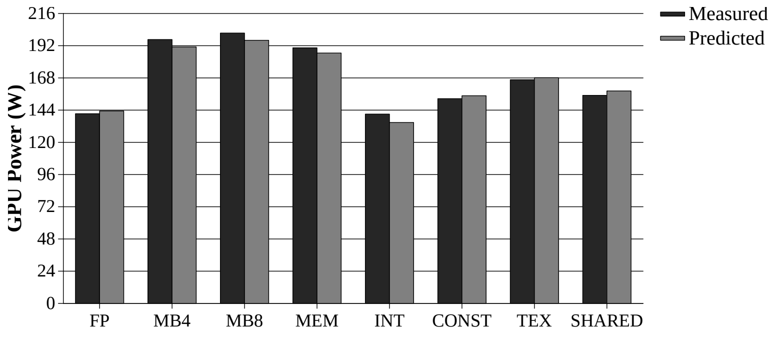


Figure 41: Measured and predicted power consumption of micro benchmarks.

Figure 42 shows the access rates for each benchmark. When an application does not have many memory operations, such as FP, dynamic access rates for FP and REG can be very close to one. The access rate for Fetch-decode-schedule (FDS) unit is one when an application reaches the peak performance of the machine.

Figure 43 compares the predicted power and the measured power consumptions for the evaluated GPGPU kernels. The geometric mean of the power prediction error is 9.18%. Figure 44 shows the dynamic access rates. The complete breakdown of the GPU power consumption is shown in Figure 36. Bino and Conv have lower global memory-access rates than others, which results in less power consumption than others. Sepia and Bs are high performance applications. This explains why they have high REG and FDS values. All the memory bandwidth-limited benchmarks have higher power consumptions even though they have relatively lower FP, REG, and FDS access rates.

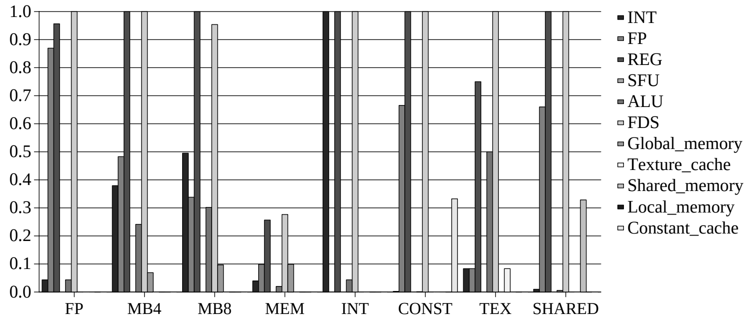


Figure 42: Dynamic access rates of the micro benchmarks.

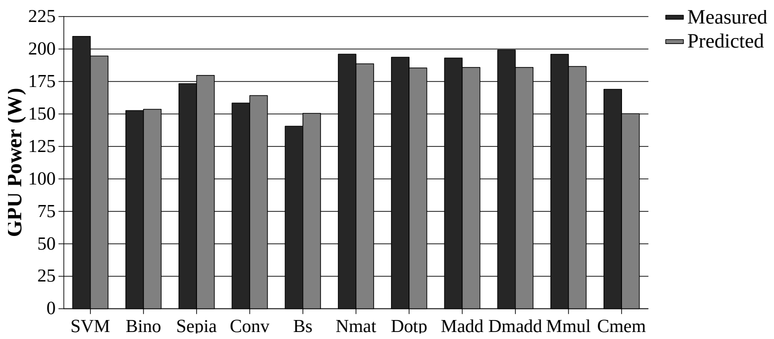


Figure 43: Power prediction using the measured time of merge benchmarks.

3.5.2 Temperature Model

Figure 45 displays the predicted chip temperature over time for all the evaluated benchmarks. The initial temperature is 57 degrees Celsius, the typical GPU cold state temperature in the evaluated system. The temperature is saturated after 600 secs. The peak temperature depends on the peak run-time power consumption. The final temperatures varies from 68 degrees Celsius for the INT benchmark to 78 degrees Celsius for the SVM benchmark. Based on Equation (56), we can predict that the run-time power of SVM will increase by 10 watts after 600 seconds. However, for the INT benchmark, it will increase by only 5 watts.

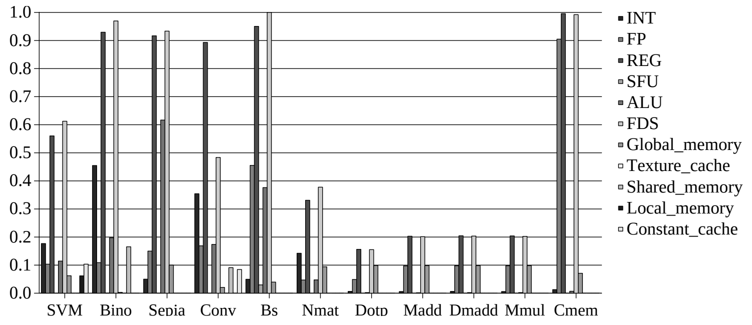


Figure 44: Dynamic access rates of merge benchmarks.

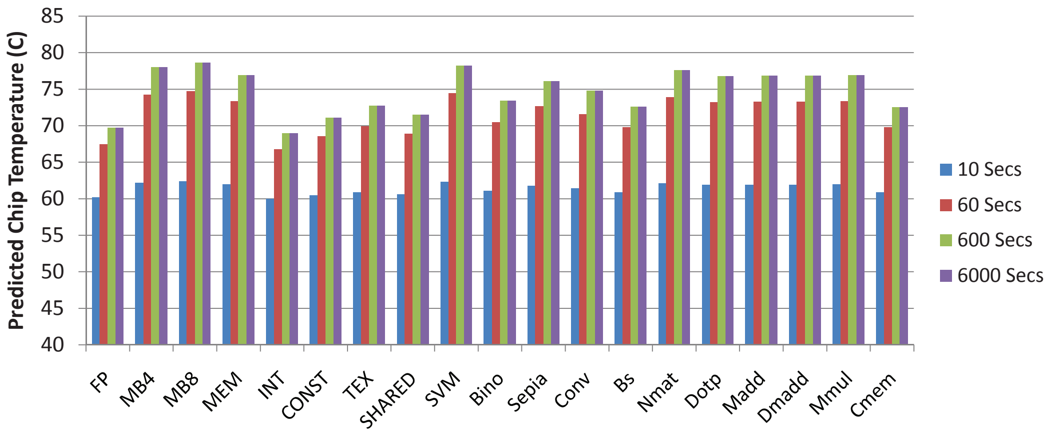


Figure 45: Peak temperature prediction for the benchmark.

3.5.3 Power Predictions Using IPP

Figure 47 shows the power prediction of IPP for both Micro benchmarks and Merge benchmarks. The main difference is that Section 3.5.1 requires measured execution times, while IPP predicts execution times. Using the predicted execution times could have increased the error in power prediction. However, since the error of timing model is not high, the overall error of the IPP system is not significantly increased. The geometric mean of the power prediction of IPP is 8.94% for the GPGPU kernels and 2.7% for the Micro benchmarks.

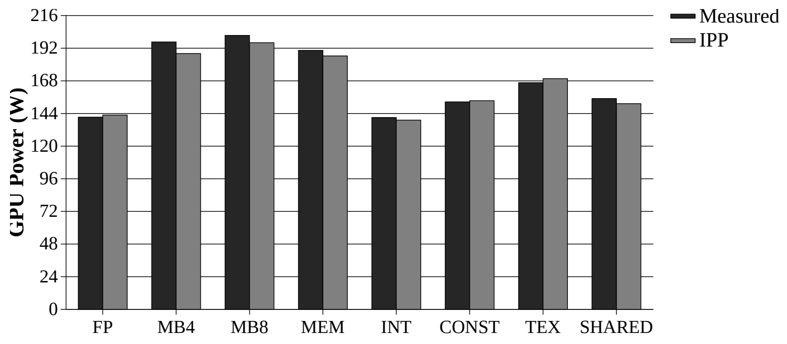


Figure 46: Power prediction using the IPP system for micro benchmark.

3.5.4 Performance and Power-Efficiency Predictions Using IPP

Based on the conditions in Equation (63), the benchmarks that reach the peak memory bandwidth are identified. The five merge benchmarks do not reach the peak memory bandwidth, as shown in Table 4. CWP values in Bino, Sepia and Conv are equal to or less than MWP values, so these benchmarks cannot reach the peak memory bandwidth. Both SVM's MWP of 5.878 and Bs's MWP value of three are less than MWP_peak_BW value of 10.8. Thus they cannot reach the peak memory bandwidth either.

To further evaluate the IPP system, the benchmarks that reach the peak memory bandwidth are synthesized; the third column in Table 4 shows the average memory bandwidth of each application. One non-bandwidth-limited benchmark, Cmem, is included as a comparison to bandwidth-limited benchmarks. For the experiment, the number of active cores is changed by varying the number of blocks. We design the applications such that one SM executes only one block. Note that, even though different number of SMs are invoked for execution, the total amount of work stays the same (i.e., the amount of work per SM changes). For the output metric, giga instructions per second (GIPS)[4] is used instead of giga floating-operations per second (GFLOPS).

[4]GIPS is used as a performance metric, because performance should include non-floating point instructions.

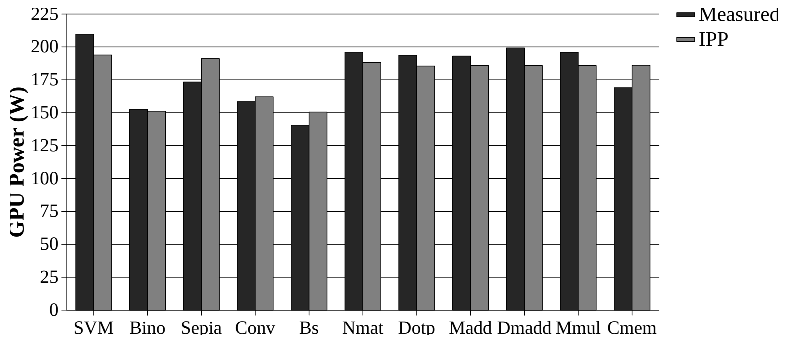


Figure 47: Power prediction using the IPP system for merge benchmarks.

Figure 49 shows how GIPS varies with the number of active cores for both the measured data and the predictions of IPP. Only Cmem has a linear performance improvement in both the measured data and the predicted values. The rest of the benchmarks show saturated performances as the number of active cores is increased. IPP still predicts the GIPS values accurately except for Cmem. Although the predicted performance of Cmem does not exactly match the actual performance, IPP still correctly predicts the trend. Nmat shows a higher performance than other bandwidth-limited benchmarks, because it has a higher arithmetic intensity.

Figure 48 shows the actual bandwidth consumption of the experiment in Figure 49. Cmem shows a linear correlation between the bandwidth consumption and the number of active cores, but Cmem still cannot reach the peak memory bandwidth. The memory bandwidths of the remaining benchmarks are saturated when the number of active cores is around 19. This explains why the performance of these benchmarks is not improved significantly after approximately 19 active cores.

Figure 50 and 51 shows GIPS per watt (GIPS/W) for the same experiment. The results show both the actual GIPS/W and the predicted GIPS/W using IPP. Nmat shows a salient peak point, but for the rest of benchmarks, GIPS/W has a very smooth curve. As we have expected, only GIPS/W of Cmem increases linearly in both the measured data and the predicted data.

Figure 52 shows GIPS/W for all the GPGPU kernels running on 30 active cores.

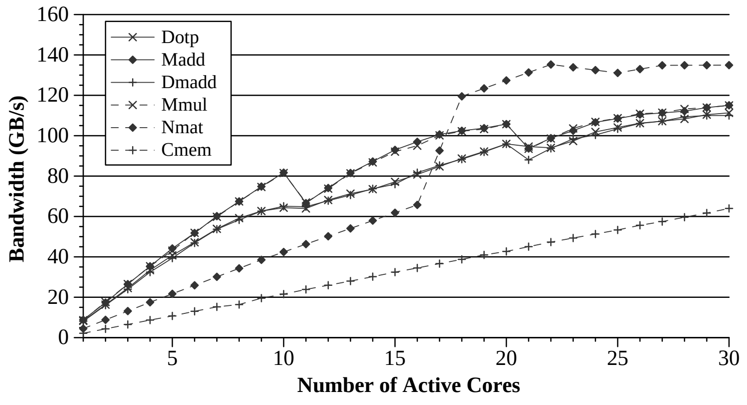


Figure 48: Bandwidth consumption vs. active cores.

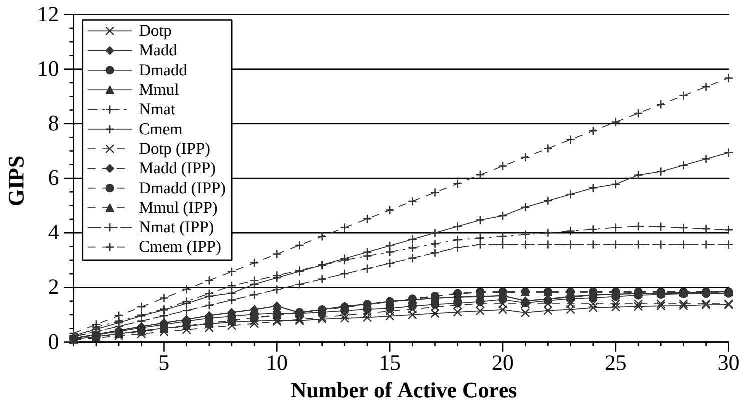


Figure 49: GIPS vs. active cores.

The GIPS/W values of the non-bandwidth-limited benchmarks are much higher than those of the bandwidth-limited benchmarks. GIPS/W values can vary significantly from application to application depending on the performance. The results also include the predicted GIPS/W using IPP. Except for Bino and Bs, IPP predicts GIPS/W values fairly accurately. The errors in the predicted GIPS/W values of Bino and Bs are attributed to the differences between the predicted and the measured execution times.

3.5.5 Energy Savings by Using Fewer Cores

Based on Equation (63), IPP calculates the optimal number of cores for a given application by choosing the highest GIPS/W point among different number of cores. IPP returns 20 for all the evaluated bandwidth-limited benchmarks and 30 for Cmem.

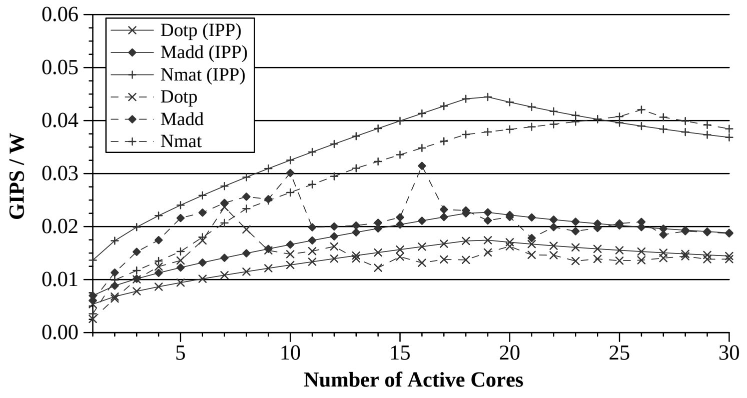


Figure 50: Perf/W vs. active cores for the first set of merge benchmarks.

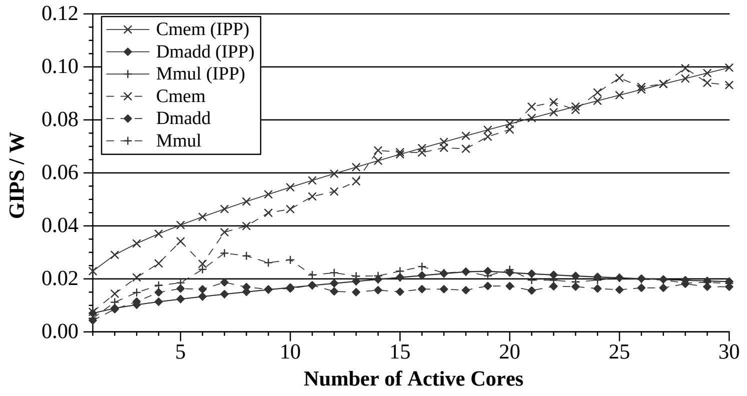


Figure 51: Perf/W vs. active cores for the second set of merge benchmarks.

Figure 53 shows the difference in energy savings between the use of the optimal number of cores and the maximum number of cores. The curve, Runtime+Idle, shows the energy savings when the total GPU power is used in the calculation. The curve, Runtime, shows the energy savings when only the run-time power from the Equation (35) is used. Finally, the curve of Powergating is the predicted energy savings if power-gating technique is applied. The average energy savings for Runtime cases is 10.99%.

3.5.6 Energy Savings from Power-Gating Technique.

The current Nvidia GPUs do not employ any per-core power-gating mechanisms. However, future GPU architectures could employ power-gating mechanisms as CPUs have already made use of per-core power-gating technique [40].

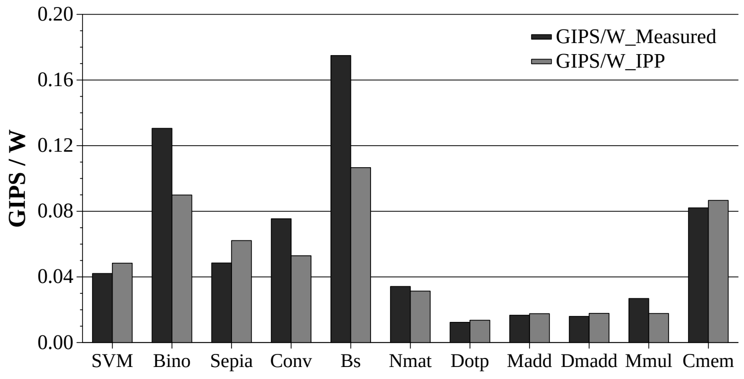


Figure 52: GIPS/W for the merge benchmarks.

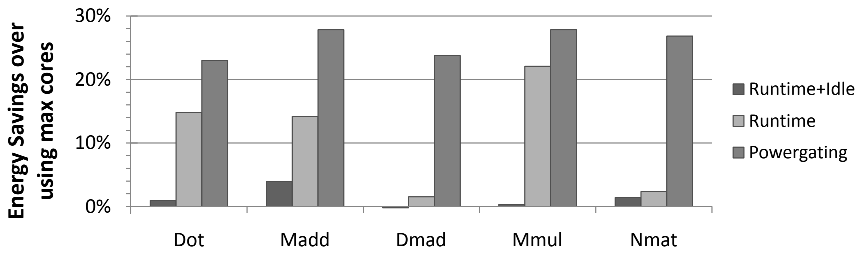


Figure 53: Energy savings using the IPP system.

To evaluate the energy savings in power-gating processors, we predict the GPU power consumption as a linear function of the number of active cores. For example, if 30 SMs consume 120 watts for an application, we assume that each core consumes 4 watts when per-core power-gating is applied. There is no reason to differentiate between Runtime+Idle and Runtime since the power-gating mechanism eliminates idle power consumption from in-active cores. Figure 53 shows the predicted amount of energy savings for the GPU cores that employ power-gating. Since power consumption of each individual core is much smaller in a power-gated system, the amount of energy savings is much higher than the current GTX280 processors. When power-gating is applied, the average energy savings is 25.85%. Hence, utilizing only fewer cores based on the outcomes of IPP will be more beneficial in future per-core power-gating processors.

3.6 *Extension of the work to Fermi GPU architecture*

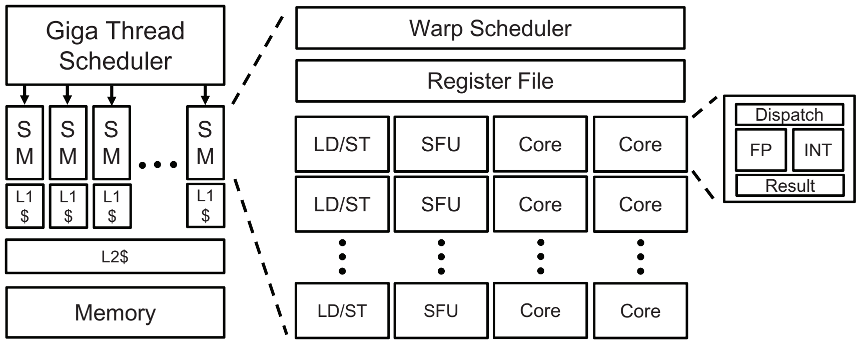


Figure 54: High-level view of Fermi GPU architecture (GTX580).

High-level characteristics of Fermi architecture are the following. (1) L1 cache (co-exists with shared memory), (2) L2 cache (connected to all SMs), (3) Six DRAM controllers, (4) GigaThread scheduler, (5) Register file, (6) 32 Cores (SPs) per SM, (7) Two schedulers per SM, (7) FP Unit, INT Unit per SP, (7) Interconnection Network, (8) SFUs, (9) LD/ST Units. The architecture contains both L1 and L2 caches, and six DRAM controllers. Gigathread scheduler schedules workloads to different streaming processors (SM). Inside the SM, a warp scheduler exists that schedules a workload from a gigathread scheduler to a series of cores that contain computation units such as FP and INT.

Table 10 summarizes the architectural changes. The notable changes include 32 streaming processors (SPs) and two warp schedulers inside the streaming multiprocessor (SM), whereas only eight SPs and one scheduler existed for previous architecture version (e.g., GTX280). Furthermore, L1 and L2 data caches are available. L1 cache co-exists with traditional shared memory and it can be configured at compile time or at run time by using special API calls. The rest of the units such as register file and shared memory show an increased capacity.

Table 10: Comparison with GTX280.

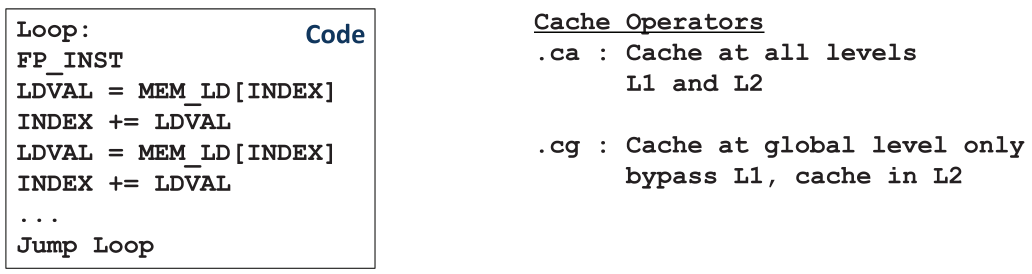
Units	GTX280	GTX580	Notes
SM	30	16	
SPs per SM	8	32	
Total SPs	240	512	
L1 Cache	None	16 KB / 48 KB	Per SM, shared with shared memory
L2 Cache	None	768KB	One Unit, shared by all SMs
Shared Memory	16 KB	16 KB / 48 KB	Per SM, shared with L1 cache
Register File	64 KB	128 KB	Per SM
FDS (Fetch/Decode/Schedule)	1	2	Per SM
FP Unit	8 per SM	32 per SM	Separate FP unit exists per SP
INT Unit	8 per SM	32 per SM	Separate INT unit exists per SP
SFU Unit	2 per SM	4 per SM	
LD/ST Unit	8	16	
Interconnection Network	Exist	Exist	
Compute Version	1.3	2.0	
CoreFreq	1.3 GHz	1.54GHz	
MemoryFreq	1.1 GHz	2.1GHz	

3.7 Designing stressing benchmarks for cache and DRAM

Designing benchmarks for analyzing cache and DRAM in terms of performance and power is not straightforward. Unlike CPU architecture design, GPUs have hundreds of threads per SM that can simultaneously access the cache structure. As a result, many lines could be evicted even before being used[57]. Hence, it is very important to design benchmarks appropriately to stress the memory system (cache, DRAM) correctly.

Figure 55 shows one example of our code design for stressing specific architectural units. The figure shows a series of dependent loads, where the next-load location depends on the previously-loaded value. To control this dynamic memory-access locations, we intentionally preset the values with either zero or a non-zero value to control cache accesses. To access DRAM mostly, the next loaded index should be after the previous cacheline.

PTX version 2.0 introduced several cache operators that determine memory access behavior, which can be set during the compilation time. The *.ca* operator is the default that caches data in L1 and L2 with normal eviction policy. On the other hand, the *.cg* operator bypasses L1 and caches data only in L2. In this study, we leverage these specifiers to facilitate cache control. Note that all these control for



```
Loop:                       Code
FP_INST
LDVAL = MEM_LD[INDEX]
INDEX += LDVAL
LDVAL = MEM_LD[INDEX]
INDEX += LDVAL
...
Jump Loop
```

```
Cache Operators
.ca : Cache at all levels
      L1 and L2

.cg : Cache at global level only
      bypass L1, cache in L2
```

Figure 55: Benchmark design for testing the memory system.

memory system is done without even making any changes to the actual kernel code itself.

3.8 Methodology

3.8.1 Measurement

Figure 56 shows the power decomposition graph that includes the idle and runtime power. First observation is that the idle power is very small (about 27 W). This is because Fermi incorporates aggressive idle-power optimization when no major workload is running in the GPU. When CUDA kernel is invocated, then we observed a high jump in power consumption, which was close to 64 watts. We believe that this is due to activating various architectural units such as instruction fetching from memory, gigathread scheduler being, and etc.

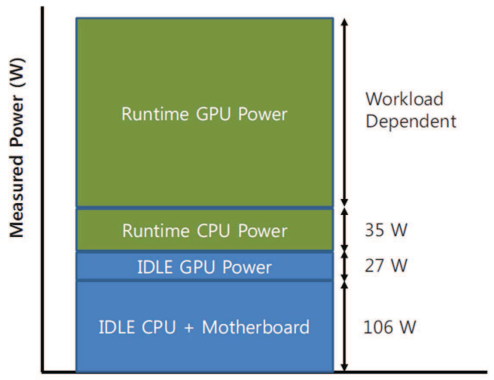


Figure 56: Idle-power decomposition.

Since we measure the total system power, we need to eliminate the CPU power.

Hence, to do this task, we take out the GPU physically from the motherboard, and the same CUDA workloads are executed after commenting the kernel call. This closely resembles the power that is consumed by CPU, that we want to eliminated. After the experiment, Figure 56 shows that the power removed from CPU is 35 W. Using this mechanism, approximating the power for GPUs is possible. We simply eliminate the runtime power for CPU and idle power including the motherboard.

3.8.2 Implementation using Ocelot Emulator

To facilitate the benchmark analysis, we implement an instruction count analyzer using Ocelot [17]. What we implemented is a tool that gets attached to the Ocelot emulator system. The following Figure 57 shows the average dynamic instructions per warp granularity, because the power and performance models are using the instructions reported at warp granularity. The sample output is shown in the below.

The first row shows the number of threads and blocks invoked for an application, followed by a total number of dynamic instructions. The next category shows the number of memory accesses, followed by the memory access type, which is either coalesced or uncoalesced with the expected number of transactions. Then, the output shows the instruction counts for all instructions types. These different instruction types are sub-categorized and associated with a specific architectural unit. These are used for power analysis purposes. Note that these numbers reported are at the warp granularity, hence, if there are ten warps executed, then that ten number has to be multiplied by this warp number. All these are taken cared by the analytical model.

3.8.3 Specialized Accessrate Functions

Section 3.2.3 first mentions about the specialized accessrate functions. This is necessary, because the accessrate is counted at an instruction granularity. For example, even if we have just one floating-point instruction, the corresponding hardware unit such as floating-point execution pipeline will be active for tens of cycles. This effectively means that the actual accessrate is much higher than the value obtained at an instruction granularity. Another graph feature is that, because aggressive power optimizations (i.e., clock-gating, power-gating, etc) are prevalent, the initial access to the structure produces a *jump* effect for power[44], as the unit becomes active from a low-power state. Since every hardware unit is different, a different accessrate function should be used.

Figure 58 shows different specialized accessrate functions that we designed and tuned. The figure shows which architectural unit is using which function. Note that the maximum power values for each unit and which specialized function is used is determined after design space exploration, as discussed in Section 3.2.4.

```
**************************************************************
Threads 256
Blocks 961
**************************************************************
Total 3159
**************************************************************
Memory_shared 0
Memory_local 0
Memory_const 1113
**************************************************************
CoalMem 0
UncoalMem 4
Coal_256bytes_transactions 0
Uncoal_256bytes_transactions 60
**************************************************************
abs 0             fp 0            int 0
add 449             fp 0            int 449
addc 0             fp 0            int 0

...
...

vshr 0             fp 0            int 0
vsub 0             fp 0            int 0
xor 0             fp 0            int 0
**************************************************************
Param 4
Texture 0
**************************************************************
Global_LD 3
Local_LD 0
Shared_LD 0
Global_ST 1
Local_ST 0
Shared_ST 0
**************************************************************
```

Figure 57: CUDA code of tiled matrix multiplication.

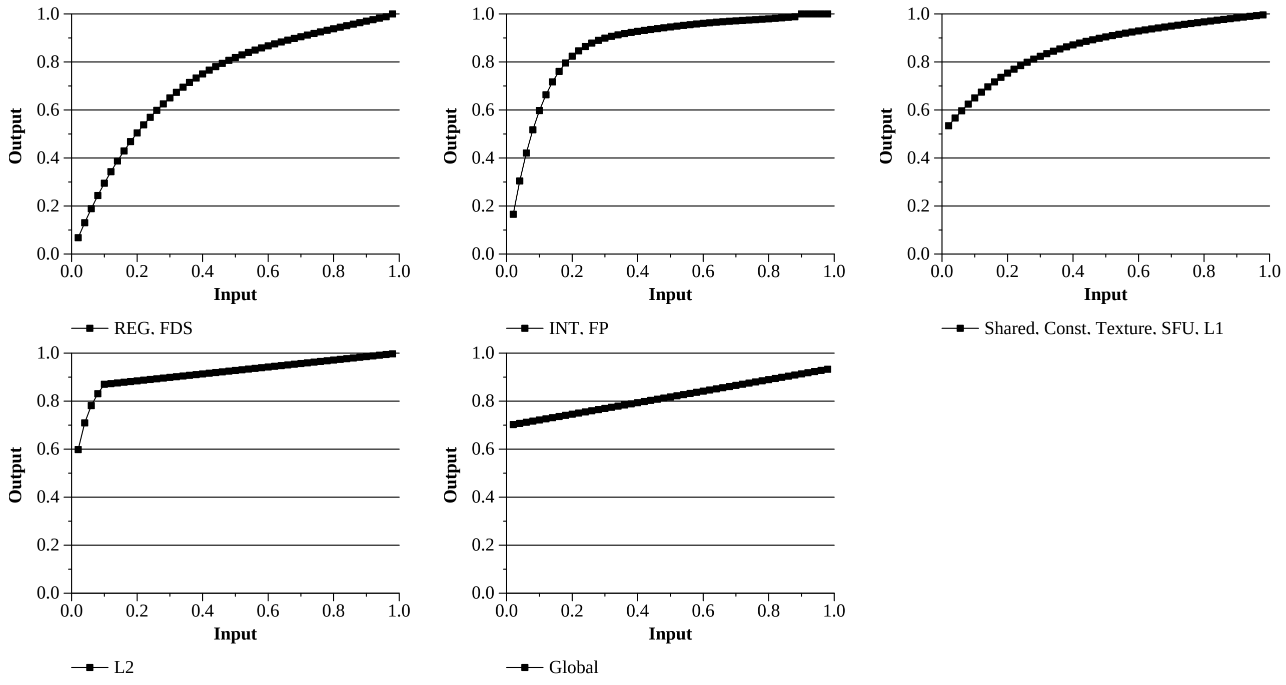


Figure 58: Accessrate conversion for architecture components.

3.9 *Microbenchmark Analysis*

Figures 59, 60 show the series of microbenchmarks and the power-model predictions. Similar to GTX280, we have different sets of microbenchmarks with a different ratio of memory-to-computations.

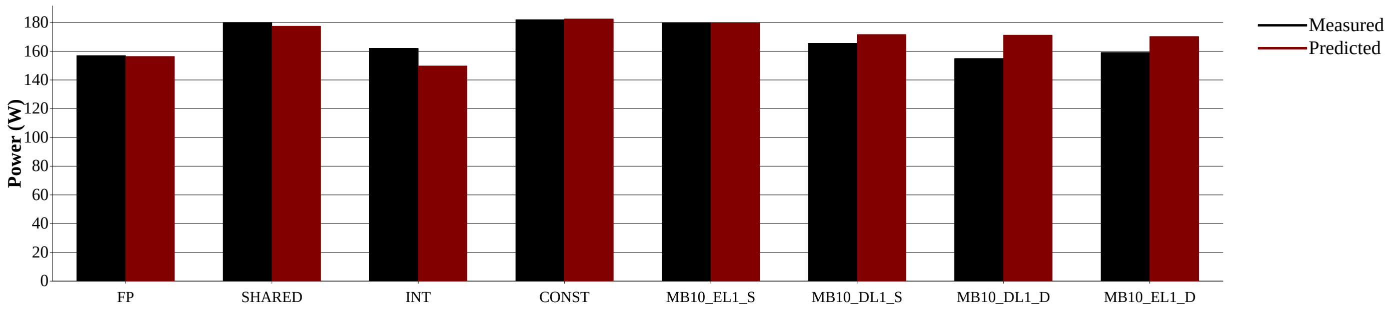


Figure 59: Microbenchmark power comparison (Set I).

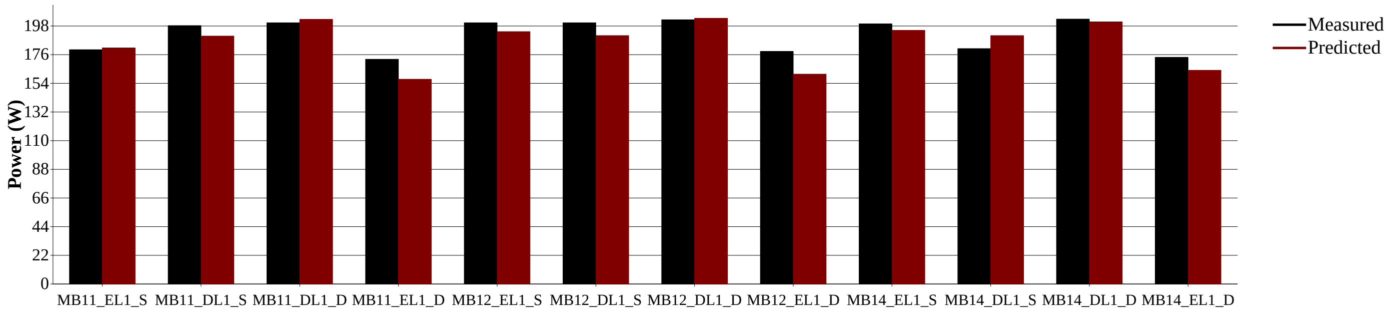


Figure 60: Microbenchmark power comparison (Set II).

The benchmark name is followed by either EL1 and DL1, and S and D keywords. EL1 means enabling L1 cache while DL1 disables it. S means accessing the same location, whereas D means a different location. For EL1 and DL1, the same benchmark code is executed, but the only difference is whether L1 cache is enabled or not. The same benchmark code is still executed for S and D, however, the difference is the *accessed* memory location. For S, the same memory is accessed every time, hence the L1 cache is mostly stressed out. However, for D, it is likely that L2 and DRAM are stressed including L1. The power model prediction is 2.12%.

Figures 61, 62 show the accessrates for each benchmark, incorporating the effect of specialized accessrate conversion. For FP, INT, Shared, Const benchmarks, the accessrate graph shows zero values for L1, L2, and DRAM. This is expected since these benchmarks do not access memory. For benchmarks with _D keywords, L1 access is disabled. With _S keywords, depending on the benchmark, we have a varying degree of L1, L2, DRAM accesses.

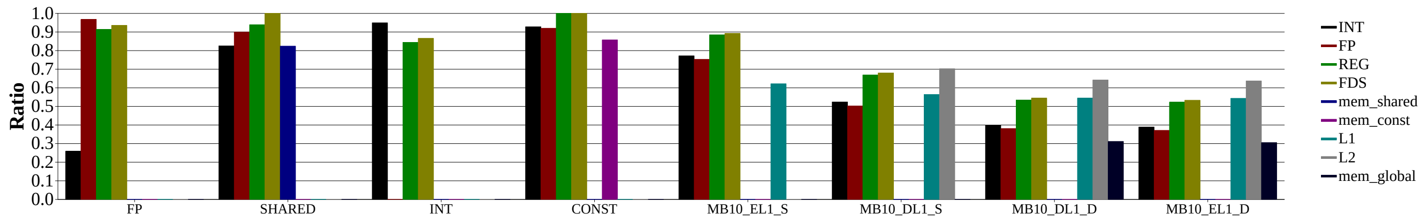


Figure 61: Effective microbenchmark access rates for architectural units (Set I).

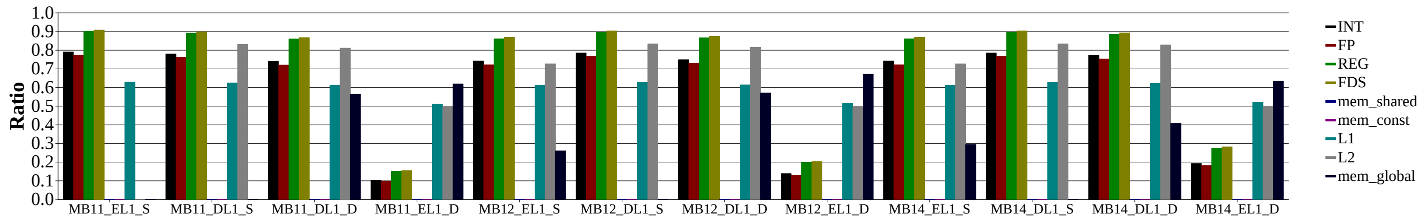


Figure 62: Effective microbenchmark accessrates for architectural units (Set II).

Figures 63, 64 show the power-breakdown graph for each benchmark. First, the highest bar is from idle power. Because Fermi incorporates an aggressive idle-power management, the idle power is very small. But a high increase of power consumption occurs when a CUDA program is executed, modeled by Dynamic_Const term. Computationally-intensive benchmarks such as FP, INT do not access the memory system in the kernel, so L1, L2, DRAM accessrate bars are zero. For memory benchmarks with _D keyword, L1 cache is zero as it is disabled. For benchmarks with _S keyword, all L1, L2, DRAM system are accessed with varying degrees.

3.9.1 Power-Parameter Results

Table 11 shows the power values for architectural units from the model and software perspective. Note that we are not attempting an exact back-engineering to find

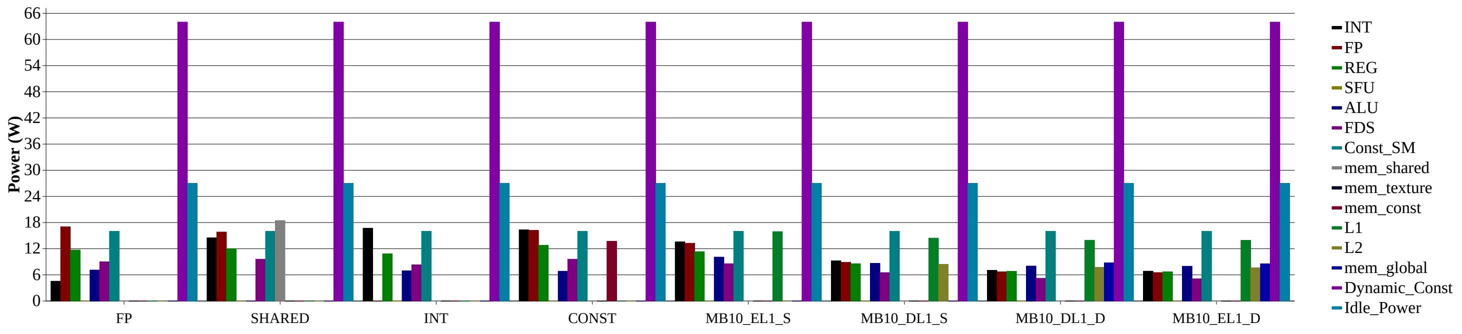


Figure 63: Power breakdown for micro benchmarks (Set I).

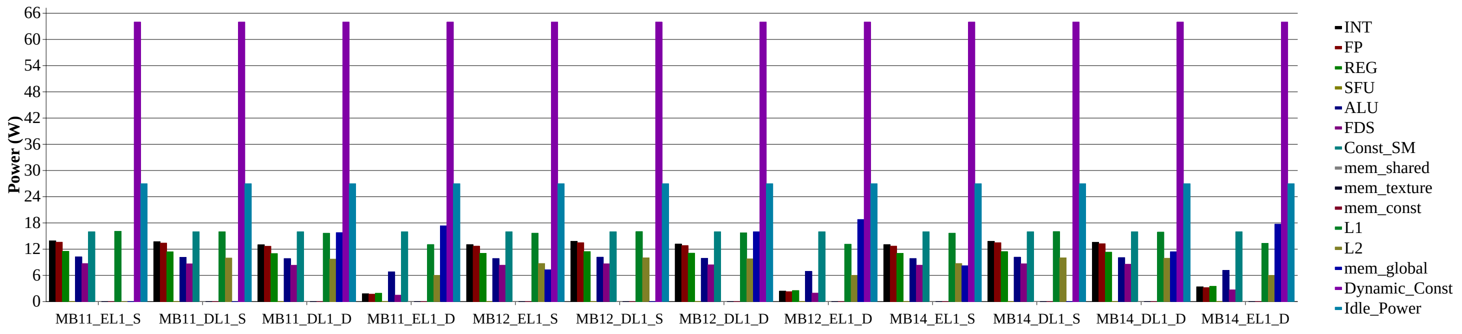


Figure 64: Power breakdown for micro benchmarks (Set II).

precise values for hardware units, but what hardware values can be used from software perspective. Nevertheless, the estimates conform to high-level architectural changes in Fermi. For example, the number of FP units have quadrupled, hence the values have increased close to a factor of four. The number of schedulers in SM have doubled in Fermi, and that effect is reflected in power values as well.

Table 11: Power parameters.

Units	**GTX280**	**GTX580**	**Increased by factor**
FP	0.2	1.1	4 (Size is increased by four times)
INT	0.25	1.1	4 (Size is increased by four times)
REG	0.3	0.8	2 (Expect much higher increased due to heavy porting)
FDS (Fetch/Dec/Sch)	0.5	0.6	2 (Dual issue scheduler, expect more increase)
Shared memory	1	1.4	4 (L1, SharedMem shared together, higher increase, more logic)
Constant cache	0.4	1.0	
Activation	Part of SM_Const	64	Constant increase when activated (Separate from idle power)
Global memory	52	28	
L1 cache	N/A	1.6	
L2 cache	N/A	12	
SFU	0.5	0.6	
Texture cache	0.9	0.9	

3.10 GPGPU Benchmarks

3.10.1 Results and Discussions

Figures 65, 66 show the power prediction for GPGPU benchmarks. The power model is able to predict accurately, considering different access rates to different architectural units. The overall prediction rate is 4.64%.

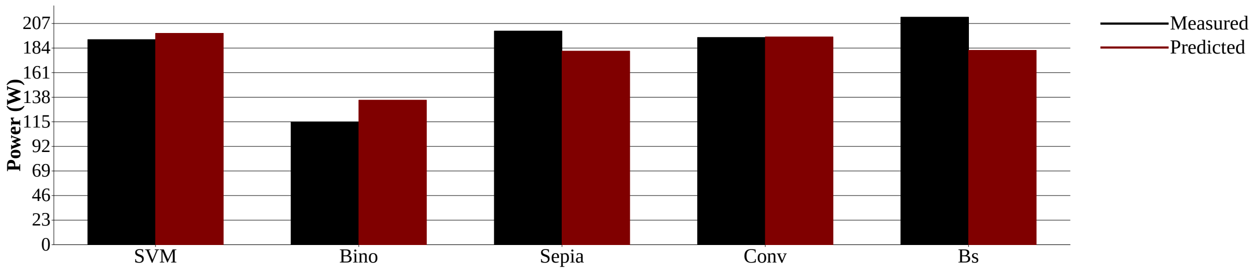


Figure 65: GPGPU benchmark power comparison (Set I).

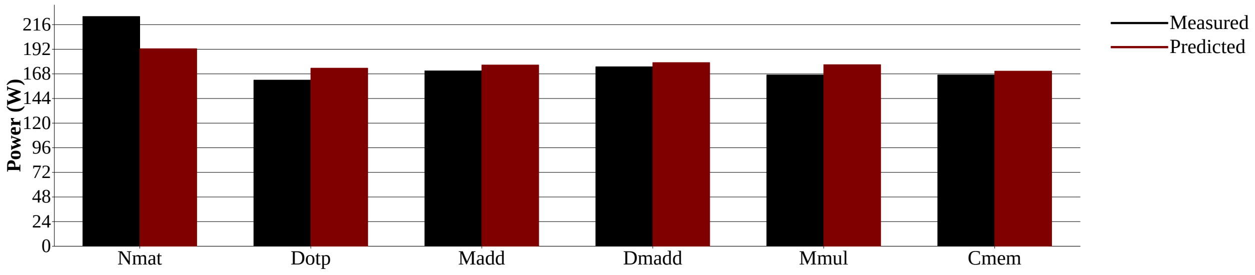


Figure 66: GPGPU benchmark power comparison (Set II).

First, a large power delta is observed between Bino (Binomial) and Bs (Blackscholes). This is due to cache hit/miss. For example, Binomial has large number of DRAM accesses, whereas Blackscholes and SVM have very high hit rates in L1 and L2. Hence, from a number of *accessed* architectural units, Binomial has the largest number of accesses as all L1, L2, and DRAM units are activated. However, the power consumption is the smallest. The reason is that because DRAM is frequently accessed, the *overall* execution is slowed down severely, and this effectively lowers the accessrates for all architectural units, hence the severe drop in power consumption.

Figures 68, 69 show accessrate values. SVM has the highest fetch/decode/schedule

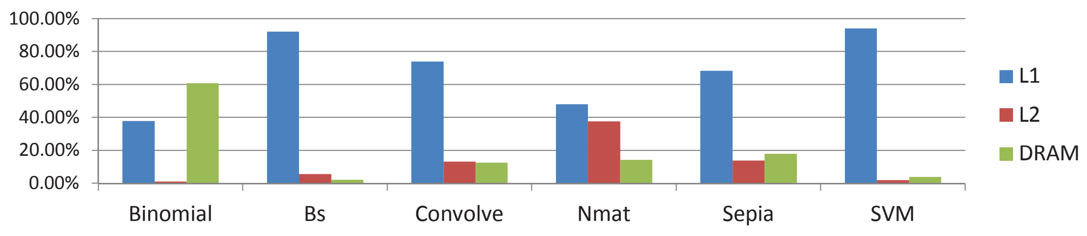


Figure 67: Cache access profile.

(FDS) value, meaning it is executed very efficiently, whereas Bino has a low corresponding value. Even though SVM does not heavily access L2 and DRAM, because of its highest execution efficiency, the measured power is similar to other benchmarks that use many hardware units (i.e., Bino, Sepia, Conv, Bs access L2, DRAM, shared memory, etc). Hence, it is not just the execution efficiency that determines power, but the power also depends on how many, and how often each architectural units are utilized.

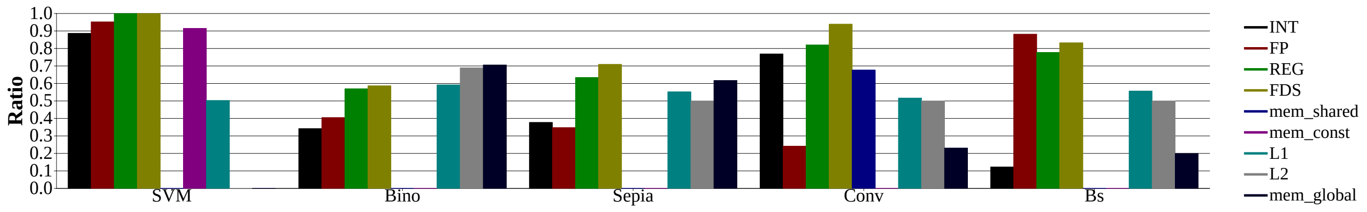


Figure 68: Effective GPGPU benchmarks accessrates (Set I).

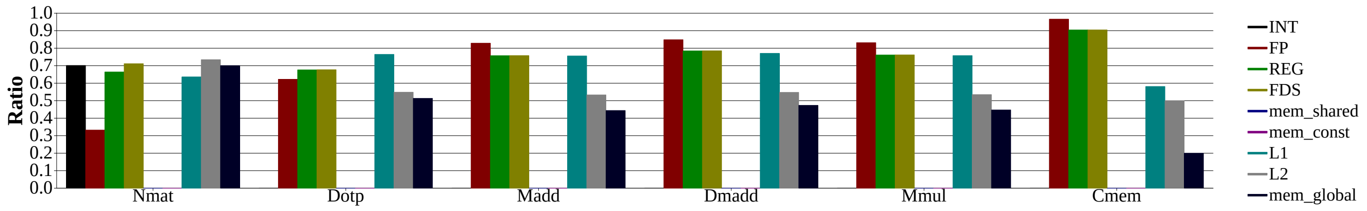


Figure 69: Effective GPGPU benchmarks accessrates (Set II).

Figure 70, 71 shows the power breakdown graph for GPGPU benchmarks. Each benchmark exhibits different accessrates to different hardware units. SVM shows higher power values compared to Bino, as SVM is executed more efficiently, utilizing much of const and L1 cache.

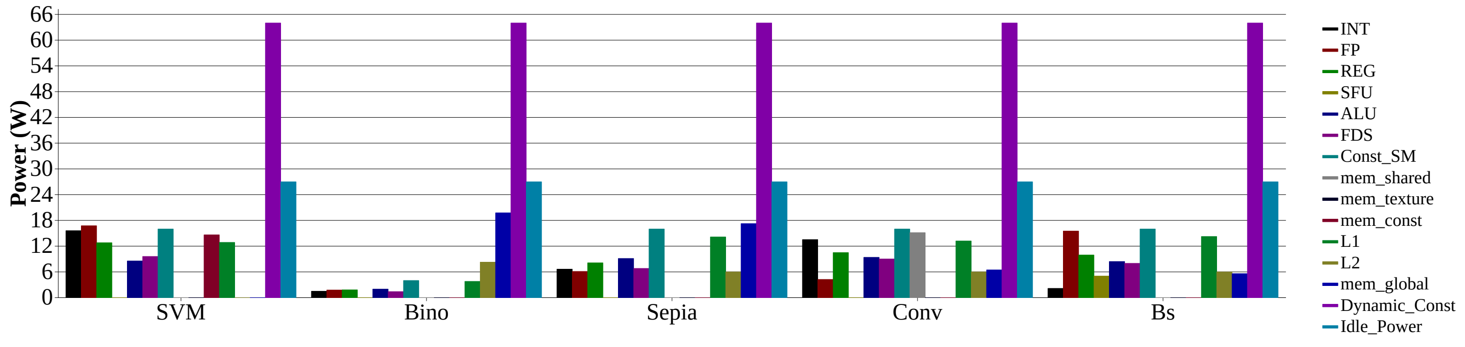


Figure 70: Power breakdown for GPGPU benchmarks (Set I).

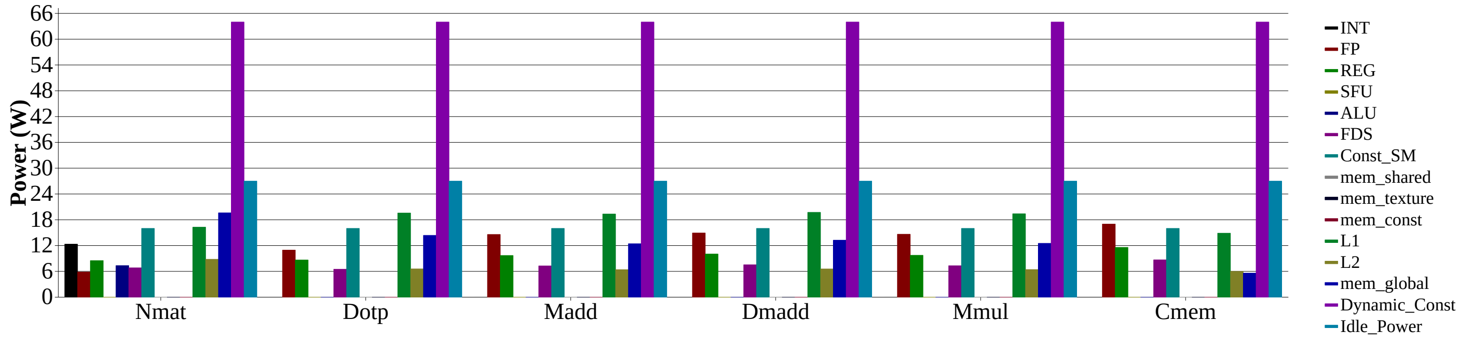


Figure 71: Power breakdown for GPGPU benchmarks (Set II).

An important point to observe is that even though measured and predicted power values for benchmarks might look very similar to each other from Figures 65, 66, analyzing power decomposition graphs concludes otherwise; different architectural units with varying accessrates are contributing to the final value. Moreover, the model is able to differentiate a large delta between SVM and Bino, which is more than 30 watts.

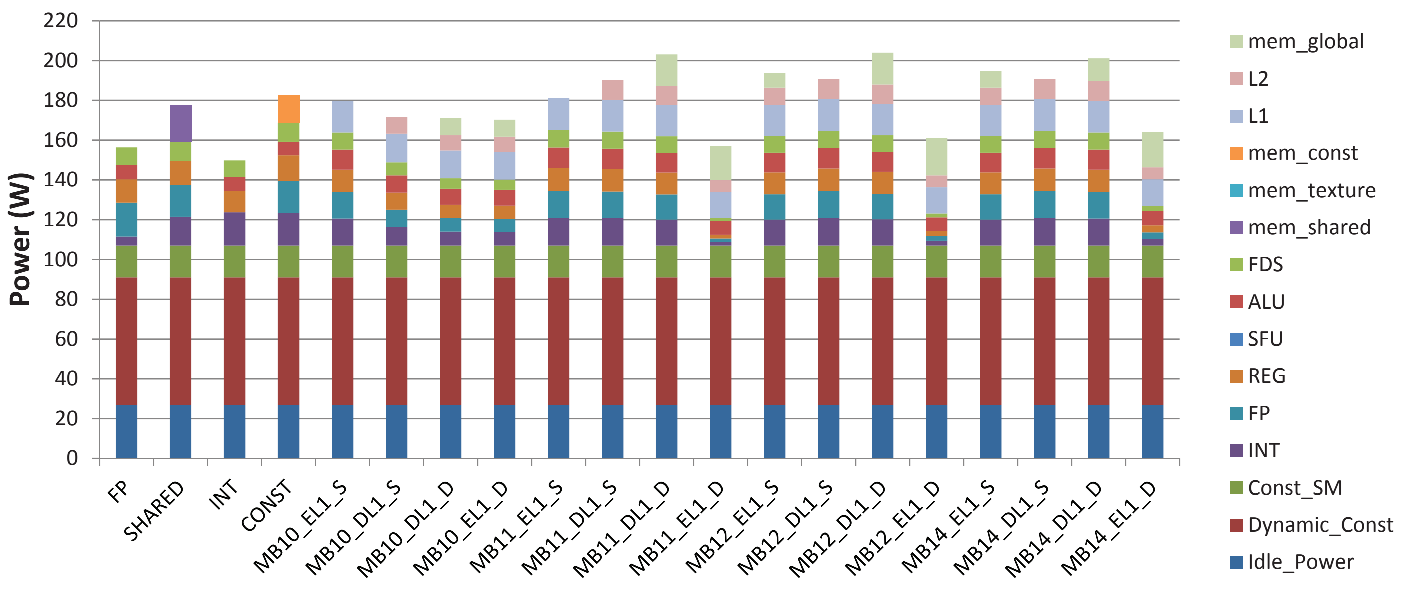


Figure 72: Comprehensive power breakdown graph for GTX580.

3.11 Energy-Efficient Execution

3.11.1 Controlling the Number of Active SMs

Unlike Figure 37 where the power increases as a sub-linear line, Fermi architecture produces a linear line in Figure 73. This shows that Fermi is able to minimize the power from unused streaming processors (SM) more effectively. We initially found

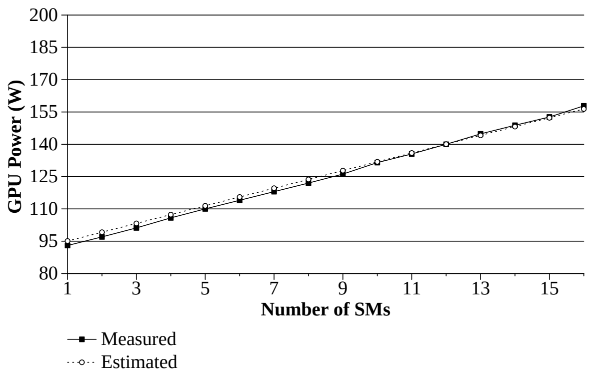


Figure 73: Controlling the number of active cores with model prediction.

that the idle power is at 27 W, which is a low value. But the power reading jumps to 93 W even if only one SM is activated. We observed that from that point, as more SMs are activated, only a little fraction of power value is added, eventually producing a linear line. This actual experiment producing a delta of 60 W is not small enough to be ignored. This reading should be taken advantage of when running real applications.

Figures 74, 75,and 76 show that depending on the number of warps per SM (N), the degree in which the bandwidth saturation occurs is different. For example, when N is 8, all the benchmarks show a linear line, which means there is no bandwidth degradation. Interestingly, when N is 16, Mmul and Cmem show not much degradation while the rest of the benchmarks show more degradations. And when N is 32, all show bandwidth degradation.

Because the bandwidth of GTX580 is very large, the model predicts no bandwidth saturation will occur for these benchmarks. Furthermore, the previous model approach does not distinguish between each application's demanded bandwidth (i.e., some applications shoot memory requests frequently due to independent memory accesses, but some do not due to large computations in between memory accesses, etc). For example, the previous model will predict the same point between Mmul and Dmadd, although the degree at which they saturate is clearly different.

To address these problems and to improve the analytical model, Chapter V revisits these benchmarks and explain more in detail, with model predictions shown in Figure 119.

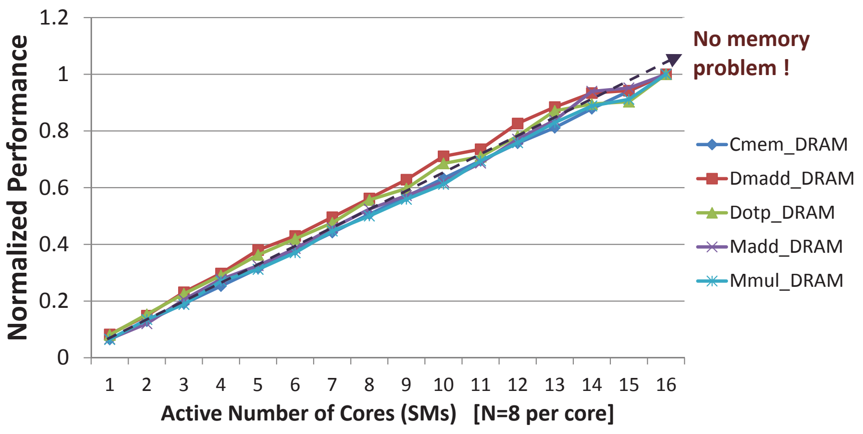


Figure 74: Memory benchmark result using N equal to eight.

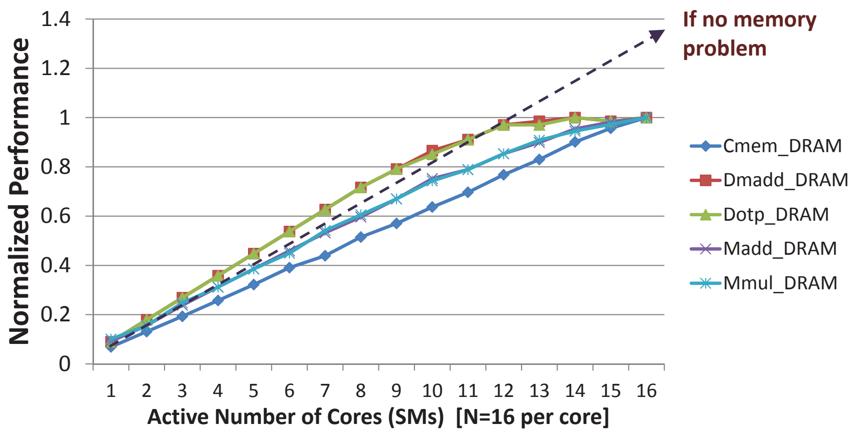


Figure 75: Memory benchmark result using N equal to 16.

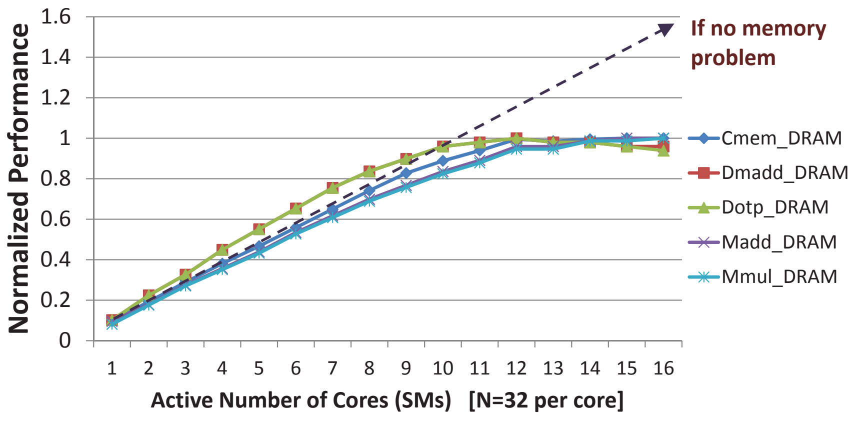


Figure 76: Memory benchmark result using N equal to 32.

3.12 Other Measurements

3.12.1 Correlation between GIPS vs. Power Consumption

Figure 77, 78 show the normalized giga instructions per second (GIPS) versus normalized power consumption, and the correction between those two values. The power consumption is very highly correlated with performance, as high performance means less idle cycles, which mean architectural units are utilized almost every cycle. However, simply knowing a high correlation with performance is not sufficient. Because the peak power value can not be predicted with the performance value only. For example, two applications might have similar performances, but very different power values as different architectural units might have been utilized.

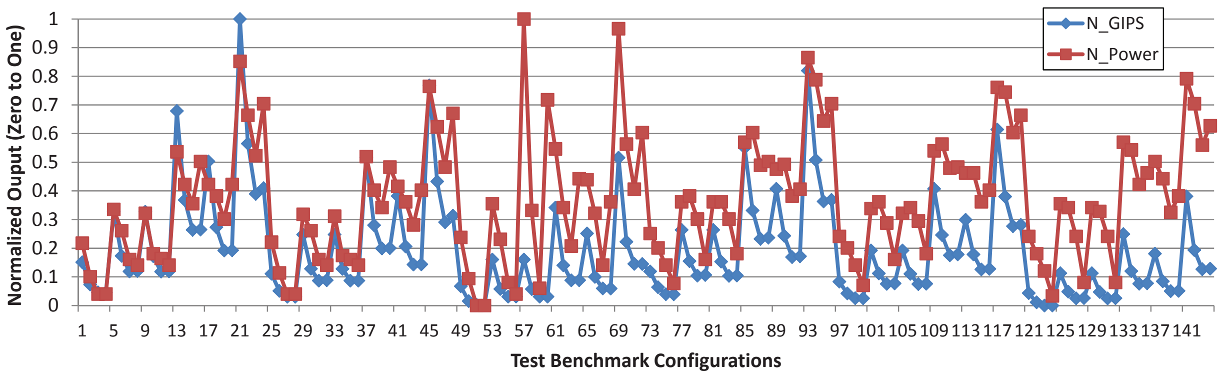


Figure 77: Normalized GIPS vs. power consumption.

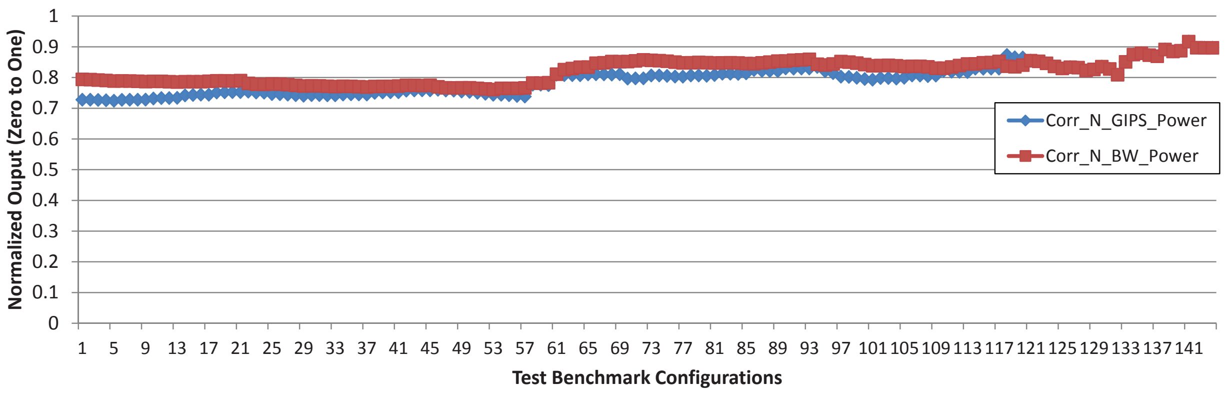


Figure 78: Correlation of GIPS vs. power consumption.

3.13 Summary

In this chapter, we proposed an integrated power and performance (IPP) modeling system for the GPU architecture and the GPGPU benchmarks. IPP extends the previous empirical CPU power work to model the GPU power consumption. IPP also considers the increases in leakage power consumption that results from the increases in temperature. Using the proposed power model and the newly-developed timing model, IPP predicts performance per watt and also the optimal number of cores to achieve energy savings.

IPP predicts the power consumption and the execution time with an average of 8.94% error for the evaluated GPGPU benchmarks. IPP predicts the performance per watt and the optimal number of cores for the five bandwidth-limited GPGPU benchmarks. Based on IPP, the system can save on average 10.99% of run-time energy consumption for the bandwidth-limited applications by using fewer cores. We demonstrated the power savings in the real machine. We also calculated the power savings if a per-core power-gating mechanism is employed, and the result shows an average of 25.85% in energy reduction.

Furthermore, we extended the power model to Fermi GPU architecture (GTX580). The prediction error for the microbenchmarks is 2.12% and for the GPGPU benchmarks is 4.64%. We found that the idle power for GTX580 is about 27 watts, which is considerably lower compared to the previous versions. But when a CUDA program is invoked, there is a increase of about 66 watts. We project that this is due to aggressive power optimization to minimize the idle running power.

The proposed IPP system can be used by a thread scheduler and the power management system. It can also be used by compilers or programmers to optimize program configurations.

CHAPTER IV

THERMAL ANALYSIS

4.1 Introduction

The number of cores inside a chip is increasing dramatically in today's processors. For example, NVIDIA's GTX280 has 30 streaming multiprocessors with 240 CUDA cores, and NVIDIA Fermi GPUs have 512 CUDA cores. On the multi-core front, the latest AMD processors have 12 cores. This high number of cores puts a lot of pressure on designing effective power and temperature-controlled architectures. Moreover, the work by Mesa-Martinez et al.[70] showed that temperature is becoming a dominant factor for determining performance, reliability, and leakage power consumption of modern processors.

In this dissertation, we use GPUs as a form of many-core processor. With GPUs, it is possible to validate that a temperature-aware thread scheduling can actually reduce power consumption. Unfortunately, unlike the state-of-the-art multicores, the current GPUs do not provide temperature sensors for each individual core. Usually, a board-level temperature sensor is provided. However, it cannot account for the rampant temperature variations across the chip due to hotspots. Hence, we propose a new temperature-measurement system that allows us to measure the temperature map, while also measuring the total power consumption.

Some efforts in academia have focused on measuring temperature using infrared (IR) cameras [71] (Although industries have better ways of measuring temperature, typically that information is not disclosed to the public). IR cameras provide an entire temperature distribution, but setting up cost is very high, and they require special oil cooling. In other words, a heatsink must be removed, which could interfere with natural heat distribution from a heatsink. Also, measurements performed through such a setup typically require some adjustments to the measured data, so as to accurately represent ideal measurements under actual working conditions of the processor (i.e., with a heatsink cooling solution). Thus, there is an opportunity for inaccuracies to creep in due to the nature of the modeling.

Hence, we propose a new cost-effective temperature-measurement system that uses thermocouples for the first time for GPU architectures. We devised a method to install thermocouples *between* a chip and a heatsink. With this system, we successfully measured the on-chip temperature distribution of a GPU processor. Thermocouples provide two benefits over IR cameras. First, they are very low cost and relatively easy to install, even in academia, without special expensive equipments. Second, a heatsink can still be placed, so we can measure power and temperature simultaneously. Then, we demonstrate the need for thermal-aware scheduling algorithms based on the correlation between the on-chip heatmap and power consumption.

4.2 Background

In this section, we discuss previous chip temperature measurement systems and provide a brief background of the evaluated GPU system.

Chip temperature characterization methods can be classified into two main branches: 1) modeling methods, and 2) measurement methods.

4.2.1 Modeling Methods

Temperature modeling methods are mainly relevant to design-time thermal characterization. They provide designers with the freedom to try out new designs and perform simulations to test its efficacy. Also, such thermal models can be plugged into microarchitecture simulators to see the effect of changing micro-architectural parameters on temperature or the effect of running different benchmarks. One of the most popular thermal models is HotSpot [55]. Based on the duality of heat transfer and electricity, the authors have modeled various microarchitecture components into equivalent thermal resistances and capacitances. HotSpot can be also used to model a particular thermal package for the chip and to observe its thermal characteristics. By plugging in the HotSpot thermal model into a simulator, one can track the thermal properties of individual components under load, understand a program's thermal behavior, evaluate thermal management techniques, and etc. The thermal model is portable, flexible and it can be built upon to cater to particular requirements. However, verification of the model is still a challenge.

4.2.2 Measurement Methods

Temperature measurement methods are mainly relevant to run-time thermal management techniques, which require a temperature measurement to occur in real time. Also, though thermal simulation models aim to faithfully mirror the behavior of the system, they are based on the designer's understanding of what factors affect the thermal characteristics of the system. So modeling methods need to be validated against actual measurements of some sort to ensure the accuracy of the model and thus require the existence of robust thermal measurement methods. In the realm of performance, modern processors provide measurement instruments in the form of hardware performance counters. However, for temperature, processors, especially many-core processors, do not yet have a concrete built-in measurement system. Though the exact methods used in the industry to measure temperature are not known, there are mainly two contemporary methods proposed in academia.

On-Chip Sensors

CMOS based on-chip sensors are mainly used to measure temperature at various points. This type of temperature sensing has been well-implemented in multi-core processors, with each core having its own thermal sensor. The IBM Power6 processor has 24 digital thermal sensors and three thermistors for monitoring temperature characteristics [23]. But in the case of many-core architectures like GPUs, so far there is just a board-level sensor [3] and one on-chip sensor whose location is unknown; the temperature of individual cores is not tracked. The advantage of using on-chip sensors is the accurate and real-time measurement of temperature across the chip, without the need for alternate cooling solutions as in the case of IR. Thus, temperature monitoring can be performed in the actual working conditions of the chip running real workloads. This translates to more accurate handling of DTS techniques. On the downside, some problems exist due to the sensors being integrated into the chip. Due to variations in the lithographic process, a complicated sensor circuit is required to achieve accurate results. This establishes a trade-off between accuracy and the amount of die area taken up by the sensor circuitry. Also, since sensor locations are discrete in nature, sensing all the hotspots on the chip is not possible, which leads to a spatial gradient of error if a sensor is not at the exact location of the hotspot.

IR-based Measurement

Infrared-based thermal imaging has gained popularity as a robust method of characterizing thermal behavior [71]. It provides good resolution and accuracy both in time and space. As such, it has been used in studying dynamic thermal management techniques. Its external nature also helps in making decisions regarding the placement location of thermal sensors on the chip at temperature-critical portions. However, there are a few limitations of using IR imaging, some of which have already been pointed out by Huang et al. [37]; IR rays cannot pass through metal. Generally, processors are encompassed with a metallic heat dissipation solution like a heatsink. So, for IR imaging to work, the heatsink needs to be removed and an alternate cooling solution needs to be provided. One of the prevalent methods in this case is removing the heatsink and providing laminar oil-cooling over a bare silicon die [71]. However, this results in different transient and steady-state thermal responses compared to a conventional cooling solution like a heatsink [37]. The other limitation is that the cooling capacity of oil is roughly proportional to the size of the oil tank and the velocity of the oil flow. In order to cool 100W-300W cores, the speed of oil flow has to be fast, thereby easily producing more distorted images. Although this method has high merits when done correctly, it comes with high setting up cost and time.

Thermocouples

The most notable advantage of using thermocouples is the cost and the ease of use for the measurement. Not only it is suitable for measurements up to 750 degrees Celsius, but it has a very thin diameter and being a wire, it can be placed anywhere easily. However, placing this wire in a specific location is a challenge as the pressure applied on it could affect the temperature readings. Furthermore, the resolution of the readings are very limited to IR measurement. However, with a very well designed thermo-spacer and some knowledge of the GPU processor layout, using thermocouples provides the best cost-effective solution. Also, reconstructing heatmap from thermocouple readings is much simpler than IR case since IR method involves high-velocity oil flow. To the best of our knowledge, actual temperature measurement on a GPU chip has not been done before, and unlike CPU architecture, GPU has very high number of cores and has more opportunities for temperature and power reduction from this study.

4.3 Experimental Setup

Figure 79 shows the block diagram of the entire temperature and power measurement system. The AC power is intercepted by the EXTECH power analyzer, which then is connected to the test computer. The computer has 8800GT GPU with thermocouples and the spacer installed. Thermocouple readings are measured by another computer using Labview software.

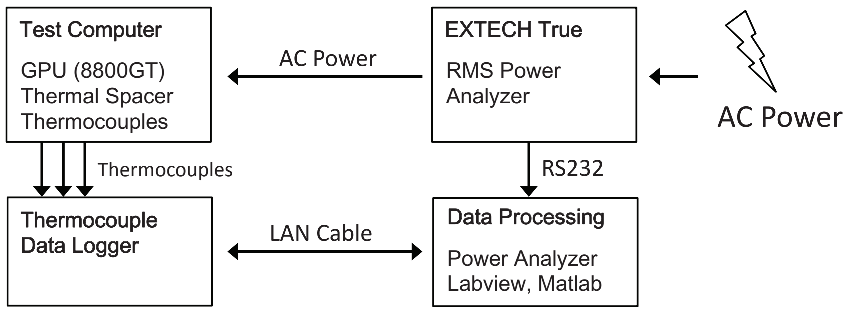


Figure 79: Temperature and power measurement system.

4.3.1 Temperature Measurement System

We propose a thermal-measurement method where thermocouples are used as temperature sensors. We have designed a thermal spacer with grooves cut in, to hold the thermocouples at desired locations. The thermocouples are embedded in these grooves. The spacer has raised edges and a shape such that it fits perfectly over the GPU chip, consequently establishing a contact between the thermocouples and the chip surface.

Thermocouple

J-type thermocouples are used in our measurement system. It is suitable for measurements ranging from 0 to 750 degrees Celsius, which is more than enough to cover the spectrum of temperatures encountered in a working GPU chip. It has a high sensitivity of around 55 uV/degrees Celsius. The J-type is one of the most popular thermocouple types because of its wide measurement range and superior voltage output, which translates to greater temperature resolution.

Thermal Spacer

Figure 80 shows the customized thermal spacer, which is made of copper, the same material as the heatsink on the GPU. Consequently, it transfers heat from the GPU to the heatsink very well. The thermal resistance of the spacer is so low that it can be ignored for all practical purposes. Thus, our temperature-measurement methodology does not affect the working of the GPU in any detrimental way.

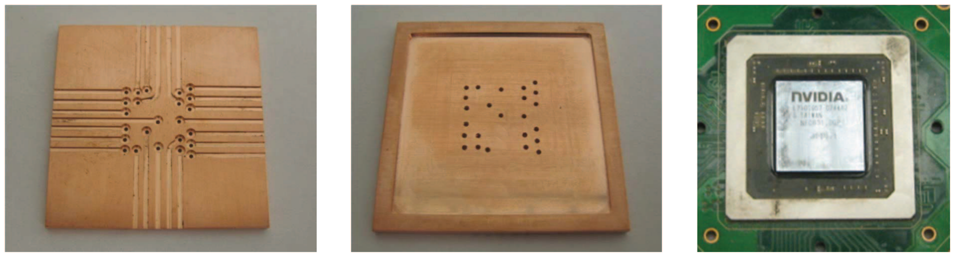


Figure 80: Customized thermospacer for 8800GT GPU.

Data Logger The thermocouples are connected to a data-logger unit NI FP-TC 120, three 8-channel thermocouple modules for FieldPoint [2]. We use a 10/100 MBps Ethernet interface for FieldPoint to communicate the sensor data to the data-logging machine.

4.3.2 Power Measurement System

We use the Extech 380801 AC/DC Power Analyzer [1] to measure the overall system power consumption. The raw power data is sent to a data-log machine every 0.5 seconds through an RS232 interface. Note that multiple computers are involved in recording power and thermocouple readings, so timing is synchronized.

4.3.3 Reconstructing Images

To reconstruct temperature images, an interpolation using Matlab is used. To interpolate between each thermocouple readings, *contourf* function is used to reconstruct an overall thermal image. This function groups a subset of temperature values from a two-dimensional processor die (x and y axis), and plots colors that represent different temperature readings (z axis).

4.3.4 Installation-Methods Previously Attempted

Taping using heat transfer tapes, soldering, and gluing using thermo-epoxy are other possible installation options, but we learned that they are not feasible. Soldering does not work because the surface of a chip cannot be soldered. Both taping and gluing allow installation of thermocouples but they have two serious problems. First, both tape and glue material themselves prevent heat transfer from the chip to heatsink. Even with material specifically designed for high temperature, it is still not good enough to transfer all the heat from the chip. The second problem is that placing thermocouples exactly at the desired locations is not a trivial task.

Therefore, we used grooves in the thermal spacer to hold the thermocouples in place. The sensor placement pattern is uniform in nature so as to take temperature measurements on the GPU chip over a uniform pattern grid. A layer of thermal paste is applied on the GPU chip as well as on the thermal spacer to ensure smooth thermal contact throughout.

Figure 81 shows the thermal spacer and the locations of the thermocouples. The figure also shows an an estimated floor plan of SMs. This floor plan is estimated based on GTX280[4], which has the same microarchitecture but different number of SMs. The floor plan shows the location of cores and texture processing clusters (TPC), which is a large architectural unit that contains two SMs, instruction cache, and work scheduler. We estimate the core locations based on our one-core active experiments in Section 4.5.2.

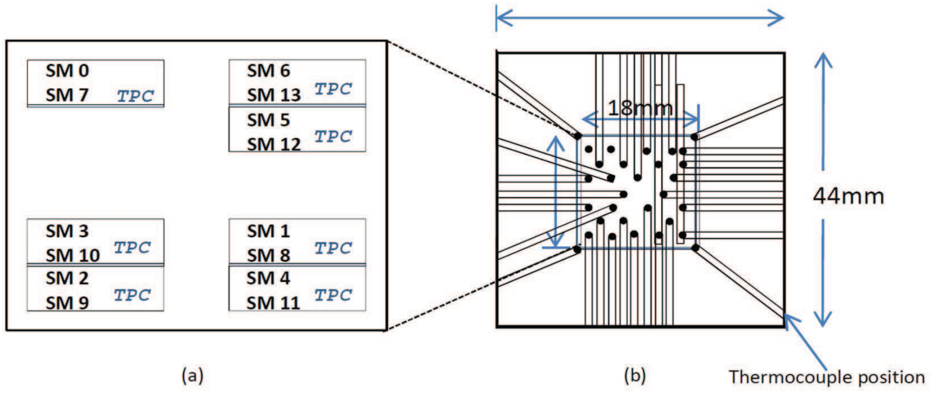


Figure 81: Temperature measurement system design (a) estimated floor plan of the GPU, (b) thermal spacer design.

Figures 82 shows a picture after the thermocouples are placed on the thermospacer, and Figure 83 shows a side view of installed thermocouples and the spacer between the heatsink and the chip.

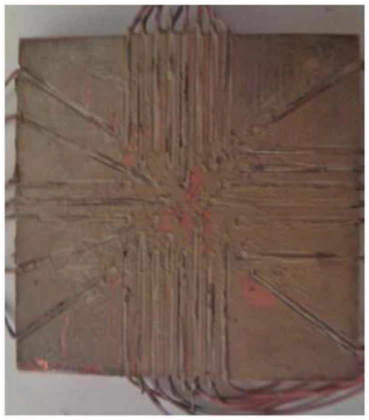


Figure 82: Picture after the thermocouples are placed on the spacer.

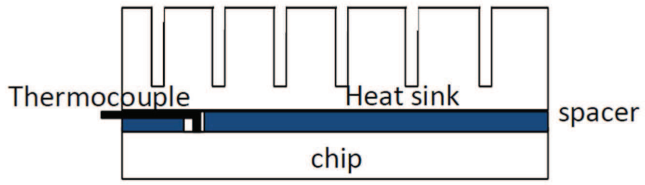


Figure 83: A side view of the installed thermocouple and spacer.

4.4 Many-Core Architecture

Figure 84 shows the high-level view of a heavily multithreaded and many-core GPU architecture (NVidia's 8800GT is used). Series of streaming multiprocessors (SM) are connected by an interconnection network and to a DRAM system.

On the top of the figure shows a series of workloads that gets scheduled by work scheduler unit. Unlike CPU architecture, scheduling is done purely by hardware. As a result, the number of activated SMs and which workload gets assigned to which specific SM is not determined at static time. Hence, the next section discusses how we designed the benchmarks to overcome this problem.

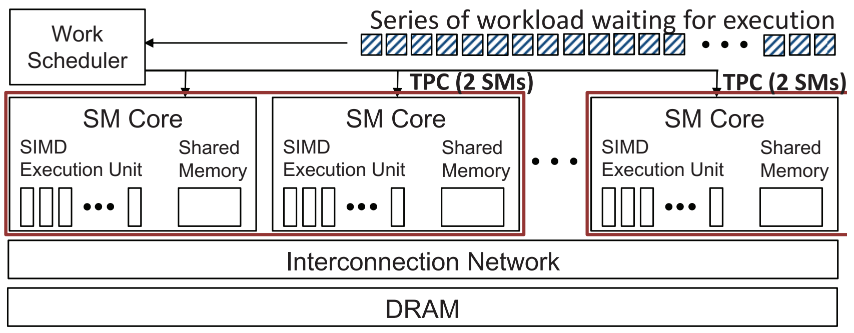


Figure 84: High-level GPU architecture and workload execution.

4.4.1 How to control which core for execution

Currently, GPU vendors do not disclose information on how to control the scheduling and other essential information. Hence, to overcome this problem, we devise a new technique in software to make sure that only a *single* workload gets assigned to each SM. [1] Each workload is intentionally modified to use just a right amount of SM resources (i.e., increasing shared memory and register usage), so that only one workload gets assigned to SM. Then, we intentionally invoke a number of workloads that is identical to the number of SMs in the GPU. Another modification is that we made each workload run for sufficiently long time as we do not want frequent context switching between workloads. For verification, when we increased just one more workload, the execution time is doubled, which shows that all SMs were activated just before the workload addition. Figure 85 shows that by controlling *act* value, we can control which active core is used for execution. Note that not all real GPU benchmarks are constructed in this manner, and currently controlling specific core from this technique using those benchmarks is not possible.

The high-level view of this specialized benchmark has a number of floating point multiply-adds and coalesced memory loads inside a loop. We supply as parameters to the kernel all the SM numbers that should be active for the run. Figure 85 shows

[1]Multiple workloads (i.e., CUDA blocks) can be assigned to SM depending on the resource usage.

```
__global__ void kernel(
int Num_Iterations, int blocksize, float *dm_src,
int act1, int act2, int act3, int act4)
{
    int bix = blockIdx.x;
    if ((bix==act1)||(bix==act2)||(bix==act3)||(bix==act4))
    {
      // A loop of computations and memory accesses

    } //end block Id
}

// Kernel Invocation
// dimGrid == #SM, dimBlock == 256 or 512
kernel<<<dimGrid, dimBlock>>>(dm_input1, dm_output);
```

Figure 85: Simplified view of code example.

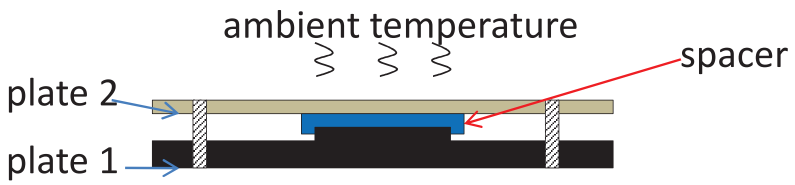
an example of activating four blocks. The benchmark is run for a fixed amount of time (120 seconds in the above case[2]) during which, the host code calls the kernel in a loop till the specified time is elapsed. We use the nvclock utility to record the GPU board temperature. Based on the benchmark output and the nvclock utility output, we calculate a running average of GPU board temperature and also note the maximum temperature for each configuration run.

4.5 Verification Results

4.5.1 Calibration Experiments

We design a calibration experiment system as shown in Figure 86. Two plates have been designed and manufactured, as shown in Figure 86. Plate 1 mimics the thermal behavior of a processor (heat source), and Plate 2 mimics the thermal behavior of a heatsink. One side of Plate 1 has the exact same shape of the chip, so we can place the spacer on which the thermocouples are already installed between two plates. We uniformly increase the temperature of Plate 1. After that, we place Plate 1 in the

[2]We choose 120 seconds for execution time to reach a steady state.



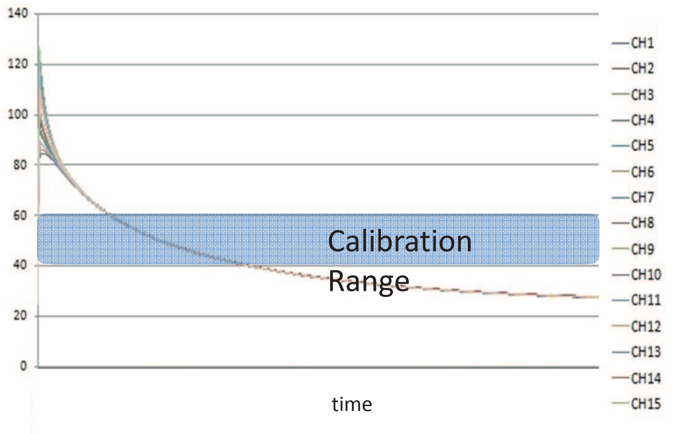


Figure 86: Thermocouple calibration system (top) and the results (bottom).

ambient temperature and install the spacer and Plate 2 in order. Then, Plates 1, 2, and spacer reach the steady state, which is at the room temperature. Figure 86 shows the calibration result, which shows that during the transient period, temperature differences occur, especially in the initial stage. We believe that these initial differences are primarily due to different physical pressures applied to some thermocouples *during* putting Plate 2 on top of Plate 1 physically. Once the weight of Plate 2 is stabilized on Plate 1, only minor temperature differences exist, especially in the calibration range (operation range). Hence, this shows that we can use thermocouples to measure the heat distribution on the surface of a processor.

4.5.2 One-core Activation

One of the important questions is whether there will be enough of a temperature difference between active cores and idle cores. To answer this question, we activate only one core at a time and vary the active core locations. We adjust the time of execution such that the temperatures reflected by the thermocouples reach a saturated

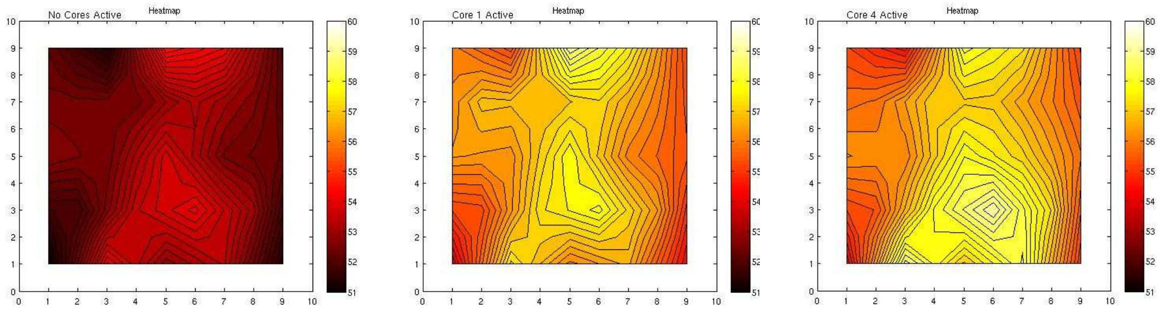


Figure 87: Difference in heatmaps for idle and active cores (left: no active core, middle: Core 1 active, right: Core 4 active).

value. We take an average of 30 readings after saturation for each thermocouple location and plot the heatmap at the saturation point taking this value. Figure 87 shows the heatmap of two different active cores (Core 1 and Core 4) and the idle state. The results show that when a core is active, the temperature is higher than in other areas by around 5 degrees. Please note that, even though the rest of the cores are idle, because there is no power gating or clock gating, those cores are still on, consuming some power.

On performing the one-core activation experiment, we observed that the heatmaps for 0 and 7 were very similar. This was also true for SM combinations of (1,8), (2,9), (3,10), (4,11), (5,12) and (6,13). So it is apparent that 0 and 7 belong to the same TPC, and the same can be said about the other combinations. Figure 88 shows the similarity in thermal maps for combinations (0,7) and (3,10). Note that neither a default GPU scheduling algorithm nor exact core locations are disclosed by GPU vendors.

4.5.3 Repeatability and Rotation Test

To test the stability of the thermocouple measurements, we performed a rotation test (the chip is isotropic). In this experiment, we insert the spacer after rotating 90 degrees from the original position. If the temperature deltas that we have observed in the original position were caused by the thermocouples themselves instead of actual

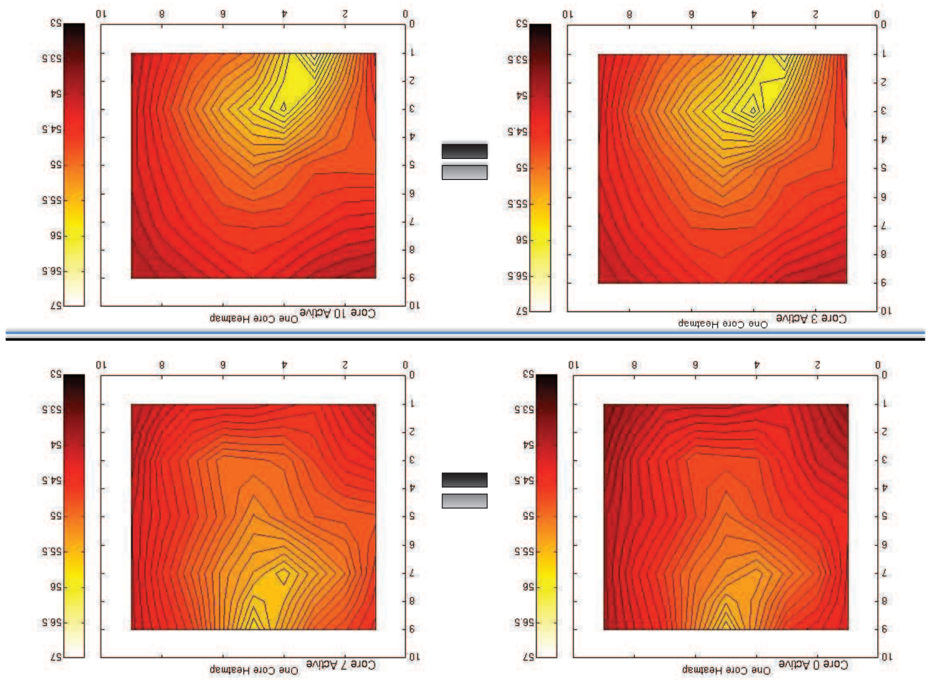


Figure 88: SMs belonging to the same TPC (top left: Core 0 active, top right: Core 7 active, bottom left: Core 3 active, bottom right: Core 10 active).

hotspot of the GPU, if we rotate the spacer, the hotspots would have rotated together. Please note that the thermocouples are already glued in the spacer, so when we rotate the spacer, the thermocouples are also rotated together. The default configuration is called 0 degree, and we plotted the heatmap with the spacer at 90 degrees. Figure 89 shows the results of the 0- and 90-degree experiments (the 90-degree data is also drawn based on the core locations in the 0-degree data). The results show that the heatmap at 90 degrees looks similar, so the hotspot is still found correctly by other thermocouples. Hence, we can say that the thermocouples are laid out properly to detect hotspots irrespective of the orientation. Although we do not present the results in this chapter, we also did the repeatability test. We rotate the spacer back to the original position and compare the results with the initial the 0-degree experiment data. The repeatability test shows very similar results. These experiments point to the robustness of the temperature-measuring method using thermocouples with a

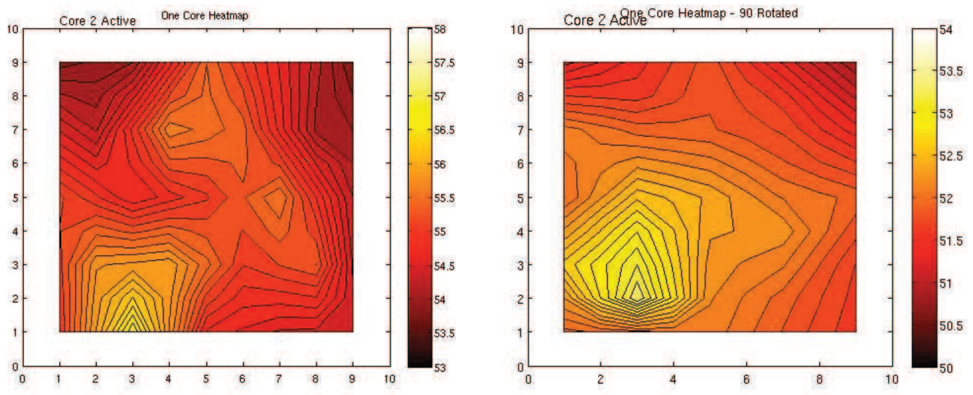


Figure 89: The 0- and 90-degree heatmaps of thermal spacer with Core 2 active.

custom-designed thermal-spacer.

4.6 Temperature Aware Scheduling

To save energy, many temperature-aware thread-scheduling algorithms have been proposed. The advantage of certain core combinations being thermally optimal or generating lower power can be explained by thinking about the layout from a thermal perspective. As explained in detail [38], interleaving high power density elements with lower power ones leads to virtual lateral heatsinks. Thus, when scheduling work on cores that are distant from high power density elements, scheduling such that active cores are separated by low power/cooler running components would give such a combination of active cores an edge from the thermal and power point of view. Thus, having an idea about the layout, one can intelligently schedule work to minimize thermal stress and power consumption. Also, a more uniform power and thermal distribution leads to lower hotspot formation.

4.6.1 Temperature and Power Measurement

Using our power-measurement system and results of the on-chip sensor, we can find the delta in power as well as temperature for different combinations of active cores.

We measure temperature and power together by activating one, two, four, and seven cores. For one-core and two-core tests, power consumption is almost the same regardless of which core(s) is(are) active. This is because one or two cores do not generate enough power to create severe hot spots. The seven-core test also shows similar power consumption behavior. This is because more than half of the chip is activated so the entire chip becomes hot (i.e., no temperature distributions.) We observe that activating four cores provides a significant delta, depending on core positions. Hence, we report the results of the four-core test.

4.6.2 Multiple-Core Tests on 8800GT

We tried different combinations of four-active cores in 8800GT and measured the power and temperature for each of the cases. Table 12 summarizes the results.

Table 12: Four active cores - measured power vs on-chip sensor.

Active Cores	Avg. Power (Watts)	Avg. Temp (Celsius)	#Active TPCs (Estimated)
0-7-1-8	253.68	76.99	2
4-11-6-13	253.77	76.59	2
2-9-5-12	254.44	77.42	2
4-11-0-1	256.36	77.44	3
3-10-5-6	257.23	78.01	3
6-7-8-9	261.04	79.47	4
10-11-12-13	261.66	78.88	4
2-3-4-5	261.77	79.60	4
0-1-2-3	262.53	80.41	4

The results show a strong correlation between temperature and power. Higher temperature consumes more power. From the table we can see that the core combination of 0-7-1-8 consumes the least amount of power and temperature, while the combination 0-1-2-3 induces maximum thermal stress and power. This is a very interesting phenomenon, because we are executing the same code on the same number of SMs (processors). This fact can be corroborated by looking at the heatmaps for the two cases shown in Figure 90.

For the 0-7-1-8 case, we project that two TPC units are activated, while four TPC units are activated for the 0-1-2-3 case. Because activating a TPC unit activates several architectural units such as instruction cache and work scheduler to SMs, we

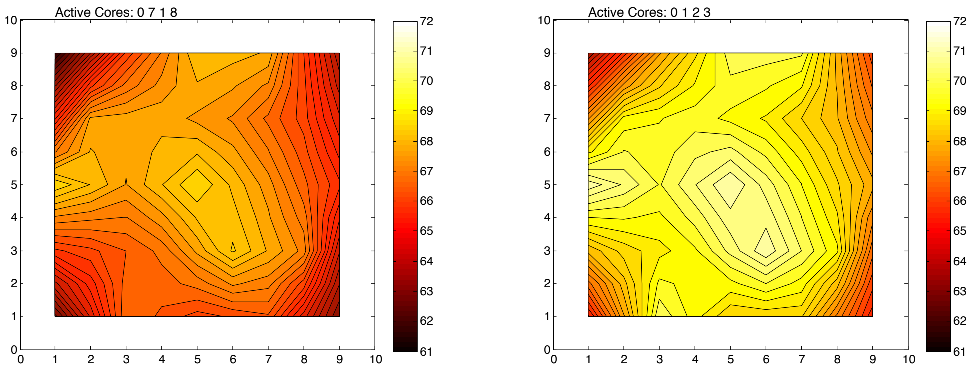


Figure 90: Difference in heatmaps between high and low thermal stress (left: 0-7-1-8 right: 0-1-2-3).

believe that the higher peak power for the 0-1-2-3 case comes from activating those other units.

Based on this results, we can conclude that, temperature aware thread/core scheduling can actually change the power consumptions. The first observation is that minimizing the number of big architectural units such as TPC reduces peak temperature considerably, and furthermore, maximizing the distance of active TPCs is recommended as heat can be spread to idle units nearby.

4.6.3 Projection of Thermal Effect with Higher Number of Cores

Section 4.6.2 showed that depending on the active core location, temperature and power consumption can be severely affected, even though the same number of cores is used for execution. We project that this will become more apparent in the architecture with many more number of cores. For example, NVidia GTX280 has 30 SMs, compared to 12 SMs of 8800GT. This GPU will give us more rooms of choosing the number of active cores and their locations. However, we do not have the thermospacer and leave this for the future work. Nevertheless, we have successfully done a similar experiment using the on-chip temperature sensor and power meter.

Figure 91 shows that peak power is proportional to temperature. Hence, temperature-aware thread/core scheduling can actually reduce the peak power.

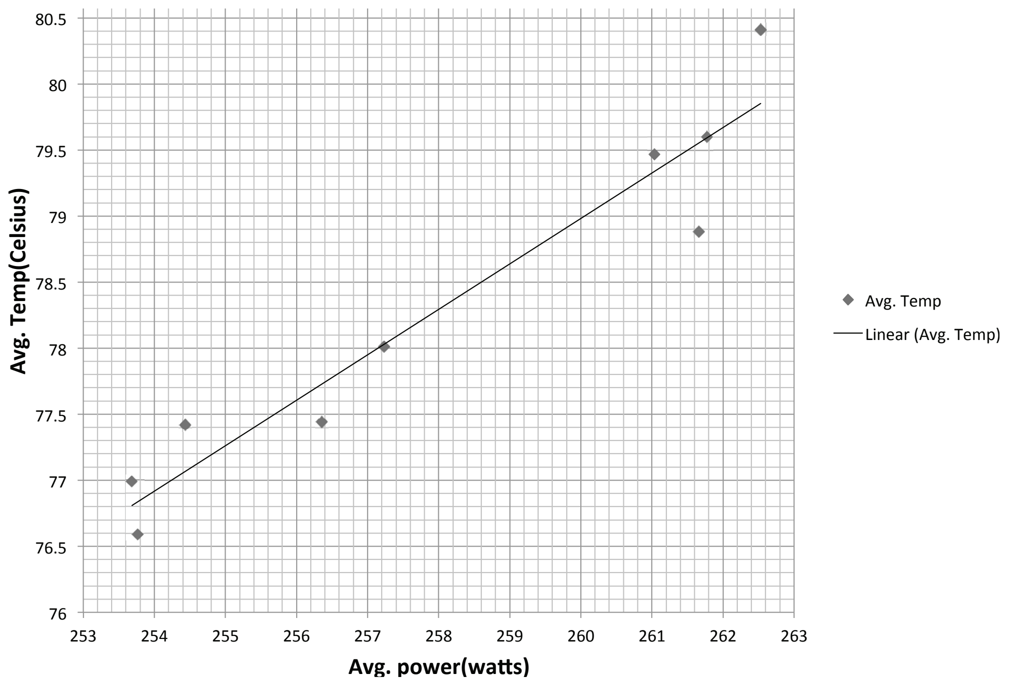


Figure 91: Effect of high temperature on the peak power achieved.

4.7 Implications and Future Work

The results in Section 4.6.2 presented that the number of TPCs activated should be minimized to reduce power and temperature. To avoid confusion, minimizing the number of active TPCs is not the same as minimizing the number of active cores; this is transparent to a programmer.[3] Another implication is that those active TPCs should be as far apart as possible. This fact was actually considered in this study [38] that if hot and cold area are interchangeably placed, they create a virtual heatsink effect. The difference is that they used a simulator, and the granularity of control was different (i.e., controlling CPU units vs. GPU cores). We actually controlled scheduling at core and TPC granularity and confirmed this effect in a real experiment.

To maximize the virtual heatsink effect, separating the active TPCs as far apart as possible is clearly the one step. However, there is a complicated trade off between (1) maximizing all SMs in a *single* TPC vs. (2) minimizing the number of active

[3]Number of active cores (SMs) is the same for all configurations in Table 12.

SMs in a TPC, which results in more number of active TPCs. It would seem that activating *all* SMs within the same TPC is better since SMs share some texture and shared cache. Furthermore, activating another TPC unnecessarily could result in more energy use. But there could be another trade off. For some types of applications with heavy memory use, SMs in the same TPC could compete each other for memory load/store units, which could degrade performance. For this case, invoking multiple TPCs would result in better performance. This deep level of investigation is future work. Nevertheless, we have managed to measure temperature of a GPU processor and perform explicit work scheduling despite no disclosed information from vendors.

To the best of our knowledge, this is the first study to analyze the thermal behavior of a GPU processor using thermocouples and extends [34] by adding one more dimension of energy optimization, which is changing active core location, not just limiting the number of cores.

4.8 Summary

In this chapter, we present a robust and reliable temperature measurement system using thermocouples. Furthermore, we overcome the GPU scheduling problem despite lack of documentations on scheduling.

We discuss the importance and an application of such a system by describing its relevance to a thermal-aware scheduling scheme for many-core systems. With power and temperature having become primary level design parameters and with the advent of many-cores, we believe that this field of research has many opportunities to be explored and needs robust tools to achieve that exploration. To this effect we feel that the system described in this chapter would prove to be very beneficial.

CHAPTER V

THROUGHPUT MODEL

5.1 Introduction

Heterogeneous architectures have been popular, and more future processors will be heterogeneous architectures. For example, AMD introduced the Fusion architecture, and Intel's Sandy Bridge and NVidia's Denver have been introduced. China's Tianhe incorporated CPU and GPU cores at a system level and has built one of the fastest supercomputers in the world.

OpenCL [79] is introduced to increase the programmability and portability in heterogeneous computing. OpenCL, an open standard for parallel programming, has emerged from the Khronos group. OpenCL is becoming increasingly popular with many vendors. Currently, many companies are actively releasing OpenCL implementation and compilers for their architectures such as Intel, NVIDIA, AMD, and IBM. The main benefit of OpenCL is that the same source code can run on multiple devices, including mobile, personal desktop computers, and even in super-computing centers. However, to achieve the best performance, the programmer still needs to spend significant amount of tuning and changing the code manually. Moreover, the optimum distribution ratios of a workload among multiple devices can not be known before actual execution or profiling.

Hence, to alleviate those issues, we propose a generic analytical model that predicts the performance of OpenCL programs for CPU and GPU. All overhead such as off-line efforts and runtime profiling overhead are eliminated. To make the analytical prediction as general and portable as possible, we use an intermediate representation (IR) rather than an actual binary information. Despite using only the IR information and a work group size, potential performance can be predicted. For devices that require a data transfer time such as GPUs, we also consider this effect in the equation. Our approach enables a programmer to know what the potential performance is before running the application. Applications are characterized based on a model. This is clearly different from repeating experiments without understanding the performance. Furthermore, as our approach is applicable as the static-time approach, it provides insights that enable further optimizations that provide energy and performance benefits. Compiler optimizations such as changing instruction mixture ratio (i.e., more memory versus more computations) and invoking fewer cores could be available to eliminate the performance bottleneck.

5.2 OpenCL Usage on Different Architectures

The same OpenCL kernel can be executed on both CPU and GPU architectures, as illustrated in Figure 92. Hence, if there is a generic performance and power model that works using LLVM IR, then the inputs related to an application do not need to be produced separately for CPUs, GPUs, and other devices. The model can simply use the same input and produce different outputs for different devices and architectures.

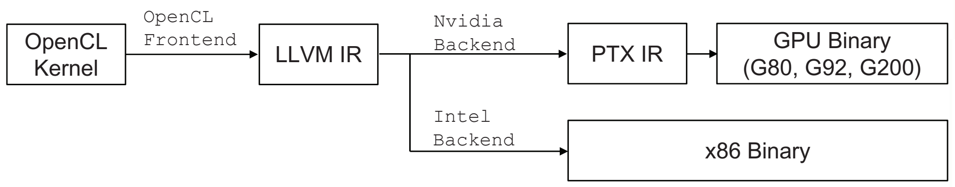


Figure 92: OpenCL compilation and execution framework.

However, predicting the performance is not as straightforward, as CPU and GPU execution styles are inherently different due to their different architectural design. CPU focuses on maximizing ILP using a few threads, while GPU utilizes thread-level parallelism by executing hundreds of lightweight threads using many execution units as single-instruction multiple-thread (SIMT). Nevertheless, multiple performance-related concepts are interchangeable from a performance and power perspective: effective useful instructions ratio, memory-level parallelism, back-to-back instructions dependency, ILP, and etc. First, to examine how hardware is different in more detail, Figures 93 and 94 show the Nehalem CPU and the NVidia GPU GTX580 (Fermi) architectures [74], respectively.

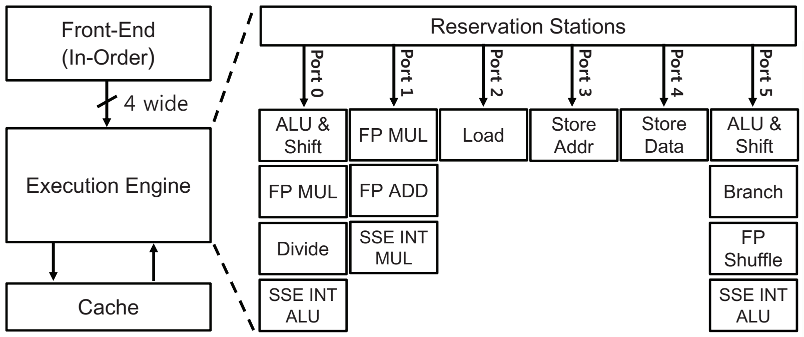


Figure 93: High-level view of CPU execution units.

Figure 93 shows a four-wide issue to the execution engine [98]. Four independent instructions from the same thread can be issued in the same cycle, provided that corresponding execution units and ports are available for scheduling and pipelining. Note that not all types of computation execution units exist in every port; ports two to four are used mainly for memory accesses, whereas scalar and vector-type floating-point (FP) units and integer ALU units exist in ports zero, one, four, and five. The CPU architecture attempts to maximize throughput by using vector instructions (SSE, AVX) when applicable; in Intel OpenCL, four or eight logical OpenCL threads are combined to generate one vector instruction [5].

While the CPU architecture is very effective in maximizing instruction-level parallelism (ILP) of a single thread or two threads as hyperthreading, the architecture is not very efficient for context-switching. Moreover, the number of FP execution units is far less than a GPU architecture that has hundreds of FP units. Hence, because of these architectural differences, there are pros and cons of the performance perspectives between CPUs and GPUs.

Figure 94 shows that inside each streaming multiprocessor (SM), a series of streaming processors (SP) exist that execute hundreds of GPU threads in parallel.

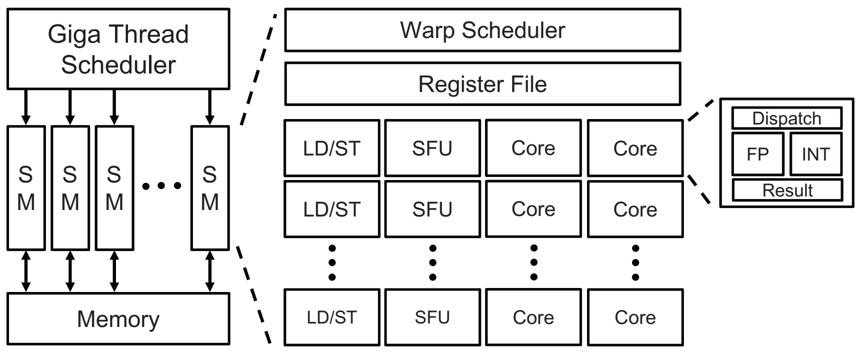


Figure 94: High-level view of GPU execution units.

Whenever GPU execution encounters a long-latency instruction and/or a memory instruction, the architecture efficiently switches to a different group of threads (warp). Hence, even most of the computation instruction latencies (not just memory cycles) are hidden by other warps. As a result, the throughput for each computation instruction is almost close to one cycle, given a sufficient number of warps per core (SM).[1] Contrary to the CPU architecture, the OpenCL programming model naturally maps to a GPU architecture; each OpenCL logical thread maps to each physical GPU thread, whereas in CPU, many OpenCL logical threads are combined to generate vector instructions (e.g., SSE, AVX).

5.3 Completing the Model

This chapter discusses much improvements to the first performance model proposed in Chapter II. First, rather than predicting mere execution time as the output, this new work shifts to predicting performance as a throughput, similar to the roofline work [103]. The roofline work uses DRAM intensity as an input. But, in addition to

[1]Back-to-back instruction dependency (DEP) is not very critical due to the high number of HW threads assigned.

using that input, our new model uses other values such as number of cores, number of warps per SM, threads (warps) per core (N), etc. Furthermore, the focus is different. We attempt to predict the actual specific performance considering architectural and application inputs: N, the effect of memory-level parallelism, bandwidth saturation, active cores, and other relevant inputs, rather producing upperbound and lowerbound values.

5.3.1 Effective Ratio

The most notable benefit of using OpenCL is that the same kernel can be used for execution on CPU, GPU, and other architectures. From a throughput perspective, remarkably, many of the same concepts can be shared regardless of which architecture is used for execution.

First, what constitutes *useful* instructions needs to be defined. In this work, we choose the FP instruction as a useful instruction. Regardless of the architecture, the higher the ratio of FP instructions to the total number of instructions, the higher performance should be achieved. If the memory effect is not considered, then this phenomenon can be clearly predicted and even measured. Hence, we define performance in Equation (64), which is intuitively very simple to understand.

$$Perf = PeakAchi \times EffRatio \tag{64}$$

The term $PeakAchi$, which represents peak achievable performance, depends only on the useful instructions (i.e., FP instruction); whether it is an SSE, fused-multiply-add, or a scalar FP instruction. For example, if only an FP instruction is executed and produces an output every cycle, then the peak performance depends on the number of physical FP units in the hardware. By using the number of those useful instructions, $PeakAchi$ is calculated. This term represents a meaningful performance number, similar to the roofline, that an application performance cannot pass beyond this limit. [2]

[2]Knowing this limit is very useful for programmer, compiler, and runtime system.

Unlike the $PeakAchi$ term, finding the $EffRatio$ is more complicated because actual

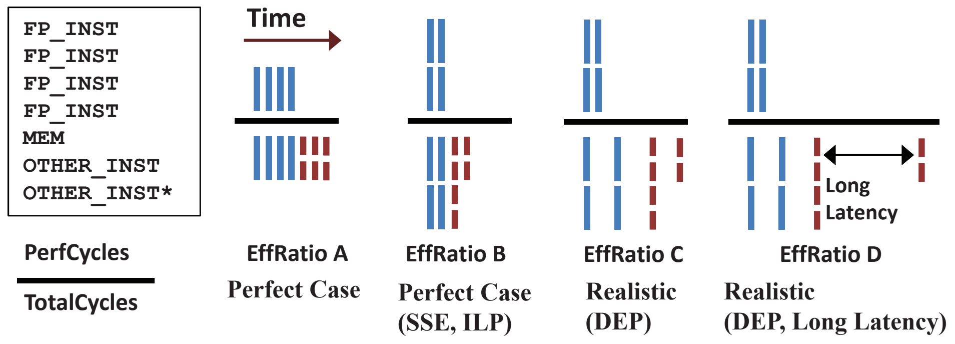


Figure 95: Effective useful instructions ratio within only computations.

execution cycles need to be considered. The numerator of the $EffRatio$ represents the *minimum* number of cycles to execute useful instructions (i.e., FP insts); ILP and SSE effects will affect this term. On the other hand, the denominator represents the number of cycles to execute all instructions. Unlike the numerator, the denominator also depends on instruction dependency (DEP), long-latency computation instructions, and memory-waiting idle cycles. For example, if five FP instructions exist out of 10 total instructions, then the ratio is at most 0.5. If there are back-to-back dependent instructions, then this ratio will go down further . In a subsequent section where we consider idle memory cycles, this ratio will go down even further. The best case is when the $EffRatio$ approaches one, however if the $PeakAchi$ term is not good enough to start with, then even if the $EffRatio$ is one, the maximum achievable performance is not very good.

To quantify the $EffRatio$ in more detail, we take the following approach, as illustrated in Figure 95. The box shows FP, MEM, and other instructions. The fraction below the box shows that the numerator contains FP instructions, while the denominator contains total instructions including FP instructions. The *Perfect* case is when every instruction, including FP-instruction, produces an output every cycle (i.e., EffRatio is one). However, for most of the cases, the effective ratio cannot be *one* because of other non-useful instructions that always exist, such as branch and integer-computation instructions.

The next case shows when SSE and ILP effects are applied. Note that these effects are applied to both the numerator and denominator, as the numerator is a subset of the denominator. The figure shows that in terms of a ratio, the value is still similar. The third case illustrates when dependency effect (DEP) is applied. It is important to see that this is only applied to the denominator, as the numerator specifies the minimum number of cycles when everything is perfect (i.e., no back-to-back instruction delay and output is produced every cycle). The back-to-back dependency is especially observed in the CPU architecture because of inefficient hiding by other threads. From the experiment, we observed that there was no instruction-latency hiding by other CPU threads, as the context-switching penalty from runtime or OS could be much higher. Hence, the number of cycles in the denominator was much larger due to the instruction latency and the distance between the producing instruction and the consumer. This part is described in more detail in Chapter 5.3.4 using the DEP term.

5.3.2 Moving the Model to Higher Level

Finding $PeakAchi$ and the $EffRatio$ from computations only is not sufficient, as there are other factors that play a crucial role in performance [33], which is the memory effect. This is not straightforward since *some* memory cycles are hidden by computations, and some are not. Furthermore, the bandwidth effect and the speed at which a memory instruction is requested between computations make this analysis very challenging.

First, we have multiple threads or warps per core, not just one. Second, when there is a memory instruction from one warp, the next warp will be executed. Possibly, there could be multiple independent memory instructions from the same warp even before changing to the next warp. So how is throughput calculated correctly considering all these factors? The challenge is that, depending on the number of warps, how much memory cycles can be hidden will be different, which directly affects throughput (i.e., more idle cycles means lower throughput). Since there are parameters (N, memory-waiting cycles, how much can be hidden, etc), quantifying the correct throughput is a challenge. Hence, we develop and propose a new systematic series of steps that are simple, but detailed enough for modeling performance effectively, even in heavily multithreaded architectures.

Figure 96 shows the methodology. Each box represents either C (computation) or M (memory access), followed by a number, which represents a thread or a warp number.

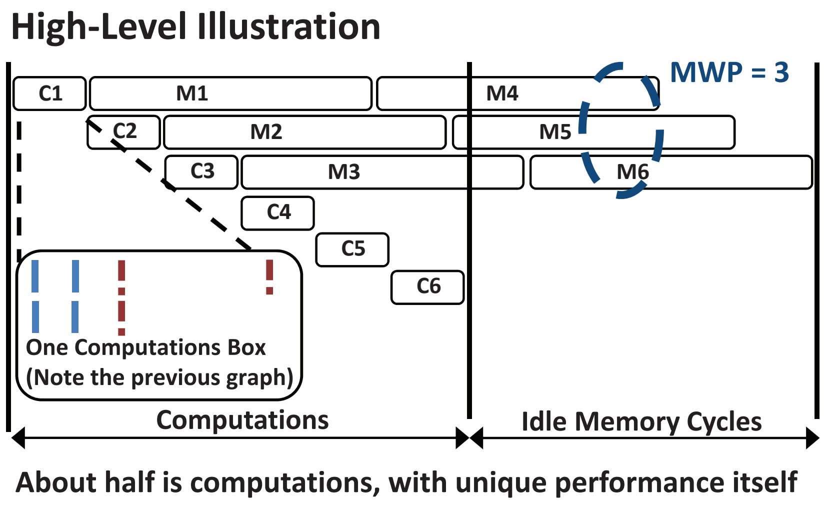


Figure 96: High-level view that illustrates the series (N) of computations with overlapping memory accesses (MWP).

Inside each computation box, the magnified box shows the *PeakAchi* effect, as discussed in the prior section. What is interesting is that from a throughput perspective, these series of computation boxes are effectively executed next to each other, *preserving* the throughput. However, because of memory-waiting cycles, the overall throughput will further drop below the previously calculated value based on only computation instructions. Because only three warps' memory-cycles are overlapped (assuming memory-level parallelism, MWP, is three), a significant number of "idle memory cycles" with respect to "computations" exist, as shown in the figure. These idle memory-waiting cycles further degrade the performance. Figure 96 represents the total execution showing computations, memory-waiting cycles, and how much they are overlapped. Using these characteristics, the final throughput value can be quantified and calculated. In the analytical model, the following parameters are proposed and used: effective computation cycles for one thread (denoted by computation box C), effective memory cycles for one thread (M), and memory-level parallelism (MWP).

5.3.3 Transforming an OpenCL Kernel

The left side of Figure 97 shows an OpenCL kernel without the effects of multithreading (i.e., one thread). The goal of the transformation is to find effective computations, memory-access cycles, and the strength of memory requests (i.e., strength is more than one if multiple independent memory requests exist from the base memory instruction).

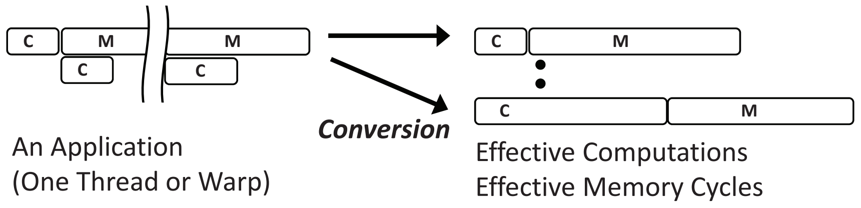


Figure 97: Transforming an OpenCL kernel into effective computation and memory cycles.

For the transformation, we simply count the number of total computations and memory instructions. The outputs of the transformation process are the following: effective computations per memory access, number of memory requests, memory cycles, DEP, ILP, and memory-level strength (MSTR). As mentioned in Chapter 5.3.6, the MSTR term represents the number of overlapping memory accesses from a single thread, primarily due to the next independent memory instruction in the stream. The term plays a crucial role, especially in determining bandwidth saturation of an application, and is integrated into the new analytical model as an improvement from the previous model.

5.3.4 Summarizing the Model using Computation Instructions

Equation (64) requires two terms: $PeakAchi$ and $EffRatio$. Figure 95 shows that SSE and ILP directly affect both the numerator ($PeakAchi$) and denominator ($EffRatio$), while DEP affects only the denominator. Since an application could contain a mixture of scalar and vectorized FP instructions, we take an average as the overall peak-achievable performance, as shown in Equation (67).

For the GPU architecture, $PeakFusedFP$ replaces $PeakVecFP$, where each instruction counts as two floating-point operations. The term simply finds an upper performance limit correctly for the GPU architecture, as a fused multiply-add instruction is not supported in Nehalem CPUs. The $EffRatio$ term is also affected as two CPU instructions (multiply, add) become one instruction; hence these factors are considered correctly for modeling DEP, ILP, and thread-level and memory-level parallelism.

$$PeakFP = DevPeakFP \times \frac{\#FP}{\#FP + \#VecFP} \tag{65}$$

$$PeakVecFP = DevPeakVecFP \times \frac{\#VecFP}{\#FP + \#VecFP} \tag{66}$$

$$PeakAchi = PeakFP + PeakVecFP \tag{67}$$

Finding the $EffRatio$ is illustrated in Equation (68). The numerator, $PerfCycles$, represents the minimum number of cycles needed to execute useful instructions, where each instruction produces an output every cycle.[3]

$$EffRatio = \frac{PerfCycles}{TotalCycles} \tag{68}$$

$$PerfCycles = \sum Useful\_Insts \;\; / \;\; ILP \tag{69}$$

$$TotalCycles = \sum Total\_Insts \; \times \; \frac{DEP\_Effect}{ILP} \tag{70}$$

If two FP units are available in the hardware, and useful (i.e., FP) instructions are independent, then those two works can be done in *one* cycle; hence that is where the ILP term is applied to both $PerfCycles$ and $TotalCycles$. This is also illustrated in Figure 95. On the other hand, the dependency factor (DEP) only affects the denominator, $TotalCycles$. If back-to-back instructions are independent, then the instruction can be issued (pipelined) every cycle. However, if the subsequent instruction is dependent on the previous instruction, then instead of pipelining an instruction every cycle, the next instruction cannot be issued for the duration of previous instruction latency (i.e., about four cycles for CPU, about 20 cycles for GPU), which lowers the throughput significantly. Hence, to model this behavior, $DEP\_Effect = MAX(Inst\_Latency/DEP, 1)$ is needed. When the next instruction is dependent on the previous instruction, then DEP is 1. This will increase $totalcycles$ by the factor of instruction latency. If the dependency distance is greater than the instruction latency, then DEP_Effect becomes one, which is not harmful to the performance.

5.3.5 Memory-Level Parallelism

The memory-level parallelism metric is a very important metric that defines the memory behavior of an application, introduced in Chapter II. The contribution is that the metric considers both software and hardware parameters, including the number of hardware threads, memory bandwidth, application characteristics, etc. Chapter III

[3]Architecture is heavily multithreaded, and execution units are pipelined.

extends this term for GPU architectures for energy efficiency. Finally, this chapter further makes more advancements and clarifications, and improves the work to one step further. This enables visualization of the performance-affecting parameters and bottlenecks.

MWP is defined as in Equation (71). The metric can be easily thought of as how many parallel memory requests can be serviced in one core (or SM). Although multiple parameters exist that affect MWP, the minimum value always determines the final value as a limiting factor. The subsequent section discusses each term in detail.

$$MWP = MIN(MWP\_App, MWP\_BW, MWP\_Proc, N) \tag{71}$$

5.3.6 Memory-Level Parallelism: MWP_App

A new metric is proposed, MWP_App. This metric is related to the previous CWP metric [33], but with major modifications applied both in concept and equations. This metric removes the classifications between the scenarios (MWP > CWP, CWP < MWP, etc.), and the metric is more intuitively easier to understand as well as making the model more generic. The definition of this metric is how many parallel memory requests *need* to be overlapped for an application. This metric also takes into account of the number of independent memory accesses from *one* warp by using a new term called MSTR (i.e., strength of overlapping independent memory accesses) and the number of warps per SM (N). When the MWP_App metric is compared to MWP_BW, which is the machine's peak bandwidth per SM, whether an application is likely to saturate bandwidth or not can be determined.

Therefore, this new approach not only enables easier conceptual understanding on the metric itself, but also enables significantly improved bandwidth saturation analysis at a finer detail. For example, whereas the previous model always predicted the same number of cores that saturate bandwidth, this new model not only predicts the better optimal number of cores, but also predicts a different number of cores depending on the application.

First, we propose the MWP_App_InfiN metric, which assumes we have infinite "N" available. This metric simply tells us how many memory requests can overlap given an application, an infinite bandwidth, and N. Figure 98 illustrates that although we have infinite N, because of large computations, the maximum possible parallelism is different and limited. The first application requires three overlapping memory

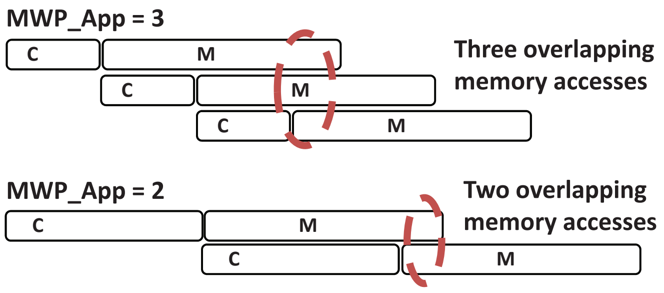

MWP_App = 3
C
M
C
M
C
M
Three overlapping memory accesses
MWP_App = 2
C
M
C
M
Two overlapping memory accesses

Figure 98: Depending on the number of computations, parallel memory-access demand is different (MWP_App).

accesses. Hence, if hardware (i.e., MWP_BW) is capable of overlapping those three overlapping memory accesses, and if there are at least three warps, then idle memory cycles will be hidden by computations. The second application, because computations are so large, only requires two overlapping memory accesses even though hardware may be capable of providing more bandwidth. This metric is shown in Equation (72).

$$MWP\_App\_InfiN = MAX(EffMem/EffComp, 1) \tag{72}$$

The metric, MWP_App_InfiN, is very useful in knowing how many warps or threads per core are necessary to *completely* hide the memory-waiting cycles. While this metric is useful as a suggestion, it can be further modified and be made more practical. We propose another metric called MWP_App. This metric considers actual number of N and memory strength.

The metric is very useful for determining bandwidth saturation if the MWP_App metric is compared to MWP_BW, which tells how many requests the hardware can handle. If MWP_App is greater than MWP_BW, then there will be a bandwidth saturation problem. On the other hand, if MWP_App is much smaller than MWP_BW, then no bandwidth problem will exist. An improvement to the previous work is that we can now quantify how *much* MWP_App is greater than MWP_BW rather than just knowing whether a bandwidth problem exists or not. We also have done experiments to verify that when MWP_App is much higher than MWP_BW, the performance even degrades significantly.

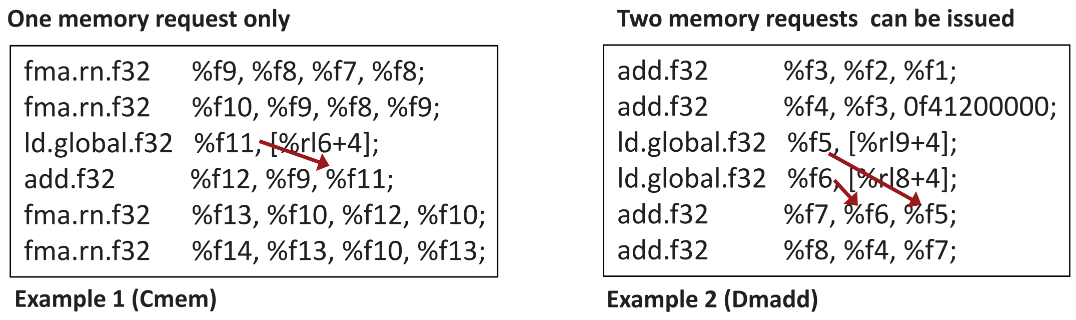
However, MWP_App is not obtained easily, as there could be multiple independent memory requests from the same thread. To address this issue, the subsequent section discusses more on the bandwidth demand of an application, especially the MSTR term.

5.3.7 Independent Memory Accesses: MSTR

While MWP_App_InfiN is useful from the perspective of knowing how many memory requests (equivalently the number of warps) need to be overlapped given an application to hide idle memory cycles, the metric can be further modified to make it more useful. If we constrain N to be the actual number of warps allocated, then the upperbound of MWP_App metric is either N and/or the number of warps that hide the memory latency. When this metric is compared to the MWP_BW metric, which represents how many parallel memory requests can be possibly overlapped per core, then whether or not this application is bandwidth limited can be easily determined.

Furthermore, the previous mechanism predicts the same number of cores; however, this is improved. In this work, we propose a simple heuristic to determine memory strength per thread or warp, and apply that to MWP_App metric. In other words, MWP_App is the true metric that shows how many memory accesses need to be overlapped for *this* application, considering computations, memory instructions, independent memory accesses, N, etc. For applications that have a high number of computations, MWP_App is small. On the other hand, applications with few computations will have high MWP_App, but can be greater if memory strength is more than one.

Figure 99 shows one example for obtaining memory-level strength (MSTR). The example on the left shows one memory access, and the next is the consumer instruction. As there is no independent memory instruction in between, the strength is one. On the other hand, the example on the right shows two independent memory accesses before the consumer instruction. Hence, the strength is two. This simple but effective heuristic is used for finding the MSTR value.



```
fma.rn.f32     %f9, %f8, %f7, %f8;
fma.rn.f32     %f10, %f9, %f8, %f9;
ld.global.f32  %f11, [%rl6+4];
add.f32        %f12, %f9, %f11;
fma.rn.f32     %f13, %f10, %f12, %f10;
fma.rn.f32     %f14, %f13, %f10, %f13;
```

```
add.f32        %f3, %f2, %f1;
add.f32        %f4, %f3, 0f41200000;
ld.global.f32  %f5, [%rl9+4];
ld.global.f32  %f6, [%rl8+4];
add.f32        %f7, %f6, %f5;
add.f32        %f8, %f4, %f7;
```

Figure 99: Different memory-level strengths (MSTR) depending on the application.

Figure 100 illustrates the effect of independent memory accesses. Two warps exist (N = 2), and each warp can issue two memory requests together because MSTR is equal to two. From the bandwidth perspective, this application's demand for memory bandwidth is higher than two, which is the number of warps in an application.

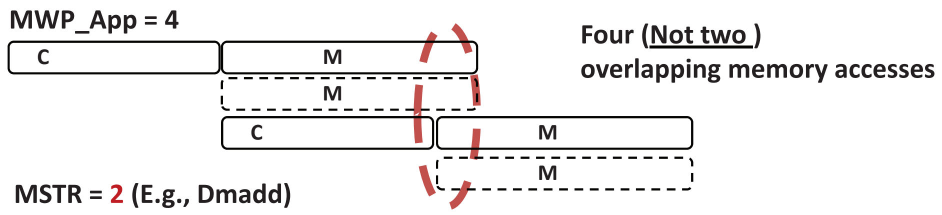


Figure 100: Increased MWP_App metric from increased MSTR.

Equation (73) and Equation (74) show the MWP_App_OneMemWarp and MWP_App metrics. First, the MWP_App_InfiN metric is compared to the actual number of warps (N), as the number of parallel memory requests cannot be greater than N. Then, MWP_App is multiplied by the MSTR term from an application. Both the MWP_App_InfiN and MWP_App terms have practical uses.

$$MWP\_App\_OneMemWarp = MIN(MWP\_App\_InfiN, N) \tag{73}$$

$$MWP\_App = MWP\_App\_OneMemWarp \times MSTR \tag{74}$$

5.3.8 Memory-level Parallelism: MWP_BW and MWP_Proc

Chapter 2.3.3 introduces the MWP_BW and MWP_Proc terms. They represent how many concurrent outstanding memory accesses the hardware can sustain. MWP_BW considers the memory bandwidth budget, while MWP_Proc considers internal memory cycles that prevent effective overlapping.[4] The MWP_BW metric has been defined as follows, where Load_bytes_per_warp and Mem_L represent amount of bytes loaded, and memory latency.

$$MWP\_BW = \frac{Mem\_Bandwidth}{BW\_per\_warp \times \#ActiveSM} \tag{75}$$

$$BW\_per\_warp = \frac{Freq \times Load\_bytes\_per\_warp}{Mem\_L} \tag{76}$$

The potential issue of this equation is that when Mem_L is increased, the intuition tells us that performance should be decreased because of the higher memory latency. However, this is not always so with the previous model. Increasing the Mem_L term lowers the BW_per_warp term, which increases the MWP_peak_BW term significantly. Also, this increases BW_Proc as well, thereby increasing the overall performance

[4]For uncoalesced memory accesses that generate many memory transactions, high internal memory cycles exist between each transaction, which prevent amount a high amount of overlapping between warps.

(decreasing the execution time) from a very high MWP value, which is not very likely to happen.

This issue can be addressed by fixing and associating Load_bytes_per_warp and Mem_L as a pair. If there are more than one transaction per warp (i.e., uncoalesced access), effective memory latency per warp will be a sum of those multiple accesses from the uncoalesced access. The important concept to preserve is the amount of bytes loaded and associated load latency. The overall ratio is the same in either case. What is an important concept for bandwidth is the vertical direction (i.e., how many different warps's memory accesses can be overlapped) should be considered.

5.3.9 Final MWP Metric

Figure 101 illustrates one example that shows calculating the overall MWP. The output of the MWP_App metric is four, which implies that at least four warps are necessary to hide all idle memory cycles. However, because hardware is capable of supporting only three concurrent memory accesses, the MWP metric cannot be greater than three. Unfortunately, this application ends up using large resources, so only two warps (N) are allocated per core (SM). Hence, the final MWP value is only two. One further insight other than the final MWP value of two is that those MWP_App and MWP_BW values can still be meaningful.

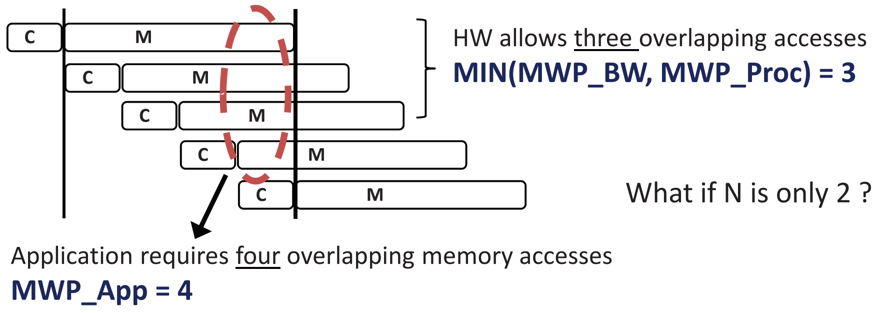

C
M
C
M
C
M
C
M
C
M
HW allows three overlapping accesses
MIN(MWP_BW, MWP_Proc) = 3
What if N is only 2 ?
Application requires four overlapping memory accesses
MWP_App = 4

Figure 101: Final MWP metric depends on MWP_BW, MWP_App, N, and MWP_Proc. The overall MWP is two (N).

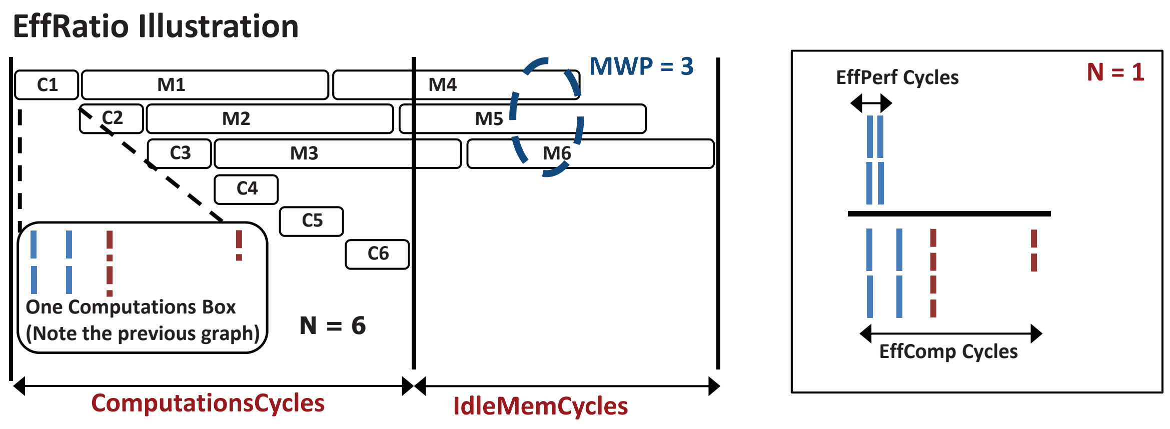
Even if there are only two warps, the memory can still be saturated; the application requires three concurrent memory accesses, but the hardware can only support two concurrent memory accesses. Later in the chapter, both the experiment and the model confirm that higher the MWP_App value than the MWP_BW value, the higher degree of performance degradation that occurs.

5.3.10 Bringing the Model to Higher Dimension using MWP

Simply calculating the MWP value is not sufficient to know what goes on in an application execution. Figure 96 showed that even in a perfect-cache case, depending on the useful instruction mixture ratio (i.e., useful instructions ratio to the total instructions), the roofline of peak performance can vary significantly. Furthermore, even knowing that is not sufficient. In an application execution, we could have a different number of warps or threads per core (N), a different number of MWP, and so on. Hence, all these factors have to be taken into account, and the challenge is how to calculate and quantify each parameter's effect on overall performance. Furthermore, not only is the final performance number important, but the parameters that were used to find the final performance provide deep insights regarding bottlenecks such as bandwidth saturation and peak potential performance.

Figure 102 illustrates one comprehensive example, assuming that the final MWP value is three. The first observation is that we have six warps per SM (N = 6), and inside each computation box, due to the back-to-back dependency latency, the throughput even without memory accesses is not very high. Given all this information, the remaining question is how to put all this information together and quantify a meaningful performance-related output such as throughput.

Note that each application has gone through a transformation process, as described in Chapter 5.3.3, in which an application has average computation cycles and average memory latency with the memory strength, MSTR. Hence, there is one computation period and one memory period for each thread (The number followed after



EffRatio = (EffPerf x **N**) / ((EffComp x **N**) + IdleMemCycles)

Figure 102: Final throughput is further reduced by idle memory cycles. However, how many idle cycles exist depends on multiple interrelated parameters, making the analysis complicated. The figure illustrates how to quantify computation cycles and idle memory cycles using computations, effective memory latency, N, MWP, and throughput for computations.

either M or C is a thread or a warp number). The first observation is that because there are multiple warps or threads, memory accesses are overlapped, while computations are executed next to each other. Note that inside each computation box, the throughput associated with *only* computations is being preserved at some value. But as soon as an execution enters the "IdleMemCycles" period, no more computations are done, just idle cycles. Hence, the key to finding overall throughput is to identify the computations-only throughput and predict the idle memory cycles that depend on N and MWP.

The solution to finding the $EffRatio$, assuming we know $IdleMemCycles$, is to use the following proposed Equation (77). The value of N is multiplied to both $EffPerf$ and $EffComp$ terms since the $EffPerf$ term is a subset of $EffComp$. If a value of zero is used for $IdleMemCycles$, then these N values get canceled out. It makes sense that if only computations are executed, then it does not matter how many warps or threads that get executed. The throughput value is preserved at some rate.

$$EffRatio = (EffPerf \times N)/((EffComp \times N) + IdleMemCycles) \quad (77)$$

However, a complex case occurs when memory accesses are considered. The larger the computations, the more memory-latency hiding that takes place. But this also depends on N, average memory latency, and MWP. Hence, the challenge is how to utilize these factors to find IdleMemCycles. Figure 103 shows the proposed approach.

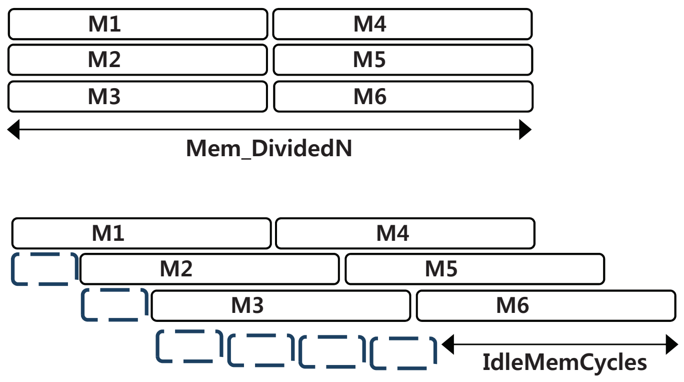


Figure 103: Idle memory cycles using MWP value of three and computation boxes.

First, Equation (78) shows the serialized memory cycles by multiplying with N. Then, by using the memory parallelism metric (MWP), perfectly divided memory cycles are obtained in Equation (79). Depending on the number of computations, the overall memory latency when the first warp requests an access to where the last warp's memory access ends could vary; hence the overall memory cycles are obtained in Equation (80).

$$Mem\_SerialN = EffMem \times N \tag{78}$$

$$Mem\_DividedN = Mem\_SerialN/MWP \tag{79}$$

$$Mem\_N = Mem\_Divided + EffComp \times (MWP - 1) \tag{80}$$

$$CompcyclesN = EffComp \times N \tag{81}$$

$$IdleMemCycles = max(0, (Mem\_N - CompcyclesN)) \tag{82}$$

But because computations can hide some of the memory latency, that memory latency should be subtracted by the computations, which results in overall $IdleMemCycles$. The benefit of deriving the model in this fashion is that categories are not necessary, and $IdleMemCycles$ is calculated accordingly to a different ratio of computation cycles, memory cycles, N, and MWP.

5.4 Results

The execution configuration uses the full occupancy (i.e., N = 48 for Fermi architecture) and large input size. Figure 104 shows the MWP-affecting parameters that determine the final MWP. Note that the MWP value is limited by MWP_BW, but not by that much difference. For this type of case where MWP_App is slightly greater than or equal to MWP_BW, we categorize this type of applications differently from an application whose MWP_App is much greater than MWP_BW. This fact is demonstrated both by the experiment and the model in Figure 119.

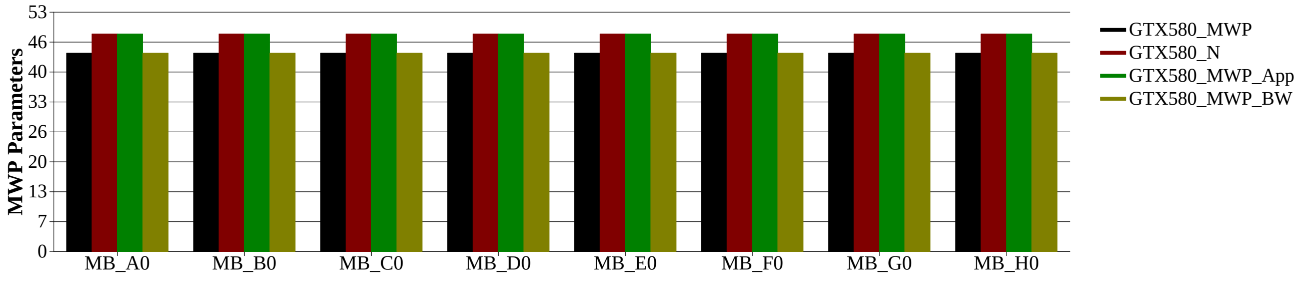


Figure 104: MWP determining factors, where the final value is limited by MWP_BW.

Figure 105 shows the performance when memory access is not considered. In other words, this is the peak achievable performance. This is especially useful in knowing and verifying the expected performance of computations only. Depending on whether the benchmark uses a scalar FP instruction (e.g., multiply instruction) or an FMA instruction (e.g., fused multiply-add instruction), the peak performance is doubled for MB_E0 to MB_H0.[5]

On the other hand when memory effect is considered, the performance drops significantly, as shown in Figure 106, because of hundreds of memory-access latency cycles. But as discussed in the analytical model section, quantifying how many idle memory cycles exist for an application is not trivial. Unlike the previous model, this new model enables quantifying how much is the peak achievable performance and

[5]For CPUs, this FMA instruction is translated to a separate multiply instruction and add instruction, and the peak performance is not changed.

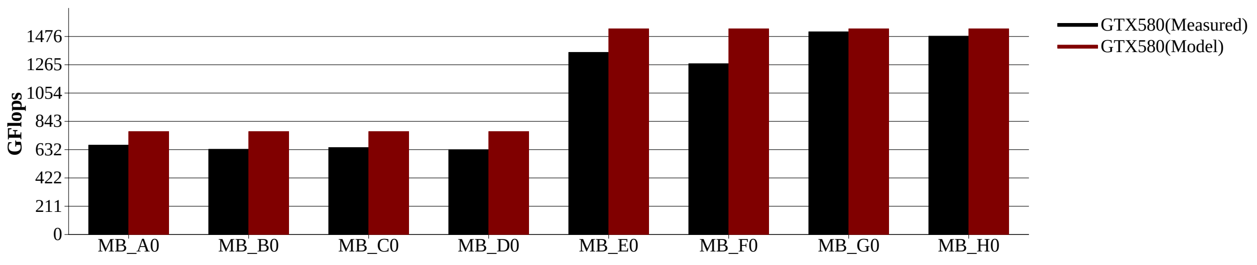


Figure 105: Baseline performance comparison by disabling memory effect.

what contributed to idle memory cycles (from N, MWP_App, MWP_BW, MWP_Proc, instruction mixture ratio).

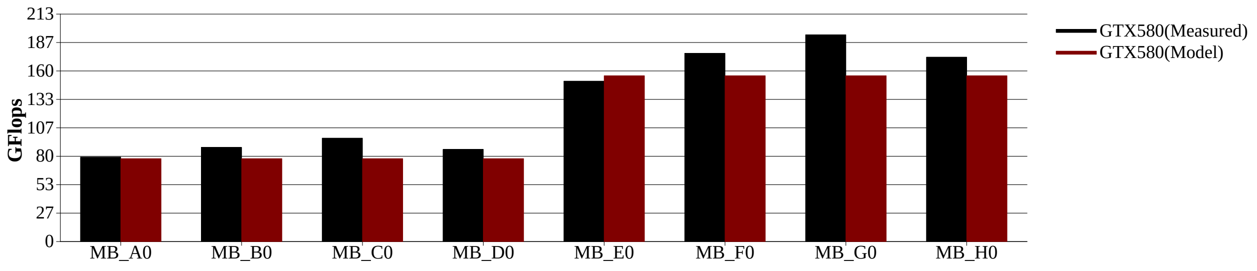


Figure 106: Performance comparison considering memory access effects.

The amount of performance degradation can be visualized by graphing the $EffRatio$ metric as in Figure 107. Furthermore, these values can be put together like Figure 108. The graph shows the predicted performance when all memory accesses are not considered in the model (i.e., also used as an approximation of perfect cache hit in L1).

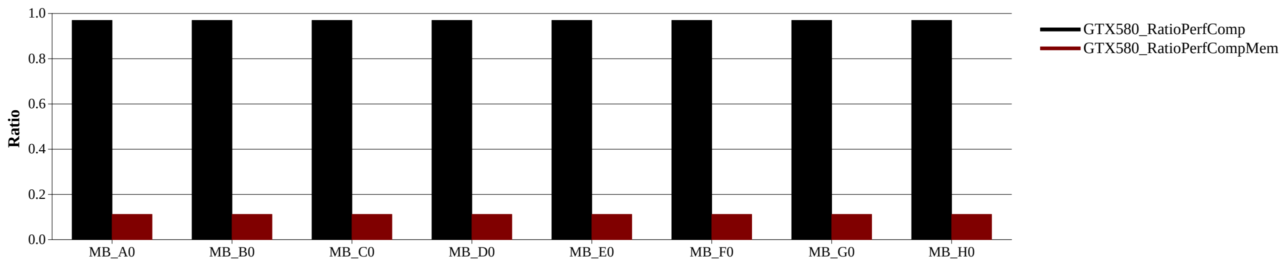


Figure 107: Visualizing the effect of memory accesses on the effective performance ratio.

Another benefit of the model is to use the model parameters to find out how much of the memory access latency has been hidden, whether by computations only, or by memory-level parallelism (i.e., if two memory accesses are overlapped, then effectively

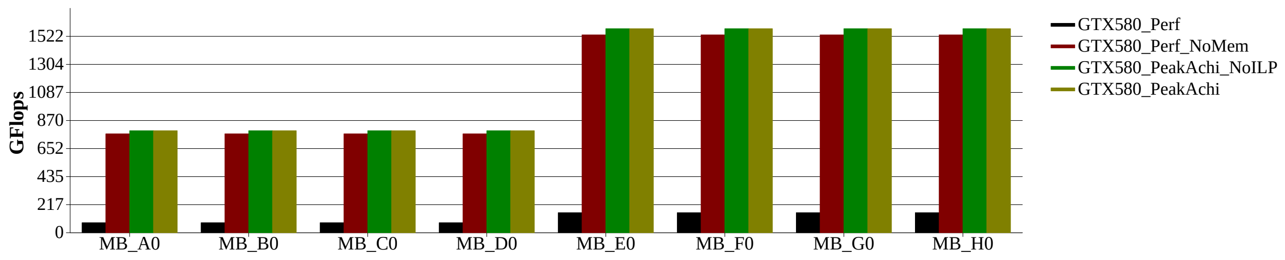


Figure 108: Illustrating the different levels of performances: computations only, memory effect, no ILP, and peak achievable performance.

one memory access is hidden by MWP).

5.4.1 Computation Mixture Difference

Chapter 5.4 discussed the results that have useful FP computations and memory accesses inside a kernel. However, that is not always the case in many benchmarks, as they contain many types of instructions such as integer instructions, index calculations, and branch instructions. Hence, we modify the benchmark to put those additional instructions in between the memory and FP instructions. This effectively changes the peak achievable performance even if memory is not considered.

Figure 109 shows that the model is able to distinguish between two cases: the left side that ends with "0" has only FP and memory instructions, while the right side additionally contains other types of instructions. This effectively lowers the instruction mixture ratio such that the FP ratio is lower, and hence the overall performance is lower. The model successfully distinguishes and predicts the actual performance correctly. Figure 110 shows that the instruction mixture change has not changed the MWP information, because the number of additional instructions was not significantly high. If that were the case, the MWP_App metric would be lower, which would bring down the overall MWP further (i.e., more computation instructions lower the MWP_App term). Since the MWP values are similar to each other, we can know that only performance differences are from the mixture ratio of useful instructions to the total instructions.

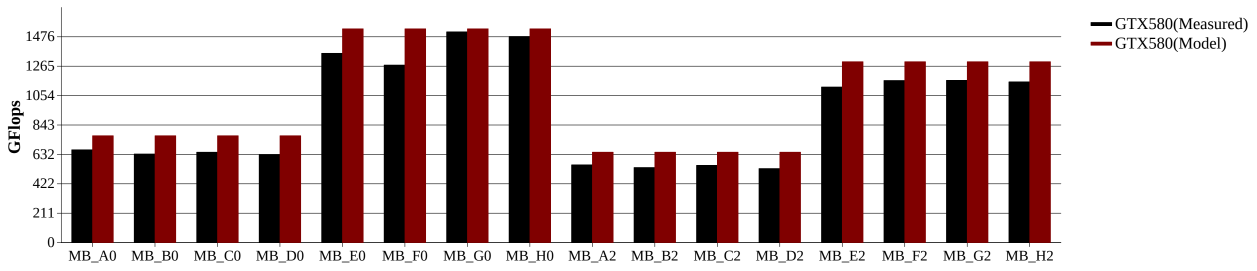


Figure 109: Computations result with different instruction mixtures. Left: only FP instructions, Right: other computation instructions.

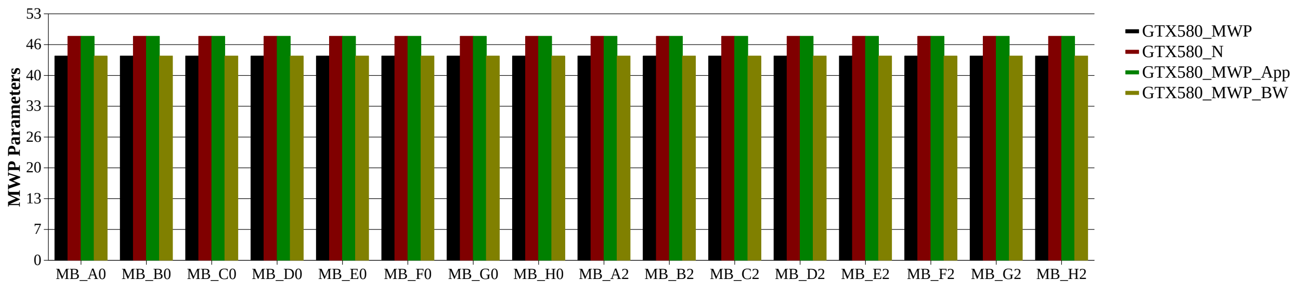


Figure 110: MWP determining factors, where the final value is limited by MWP_BW.

Figures 111, 112, and 113 show the ratio for the computations, the peak achievable performances, and the quantitative value for the memory hiding, respectively. Note that computations ratio for the left side is higher than the values on the right due to different instruction mixture. But when memory effect is considered, because of memory-access latencies, on average, the effective ratio values become similar, as the memory is the dominant performance-affecting factor.

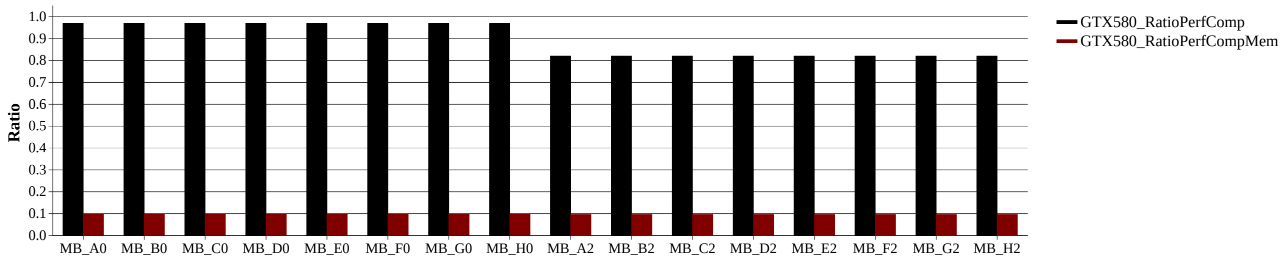


Figure 111: Effective performance ratio. Note the changing ratio between 0_group and 2_group due to different mixture ratios.

Figure 112 shows various performance predictions. The first bar shows the overall performance considering memory effect, where the second bar predicts the peak achievable performance considering only computations. The third bar is the performance when ILP is not considered, and the fourth bar is the peak achievable performance. Since N is sufficiently large, the ILP effect is not relevant for this result. Figure 113 shows a very small value for hidden memory ratio. A very low ratio of "idle memory cycles" to "serialized memory cycles" simply means that most of the memory cycles are hidden by overlapping memory effects.

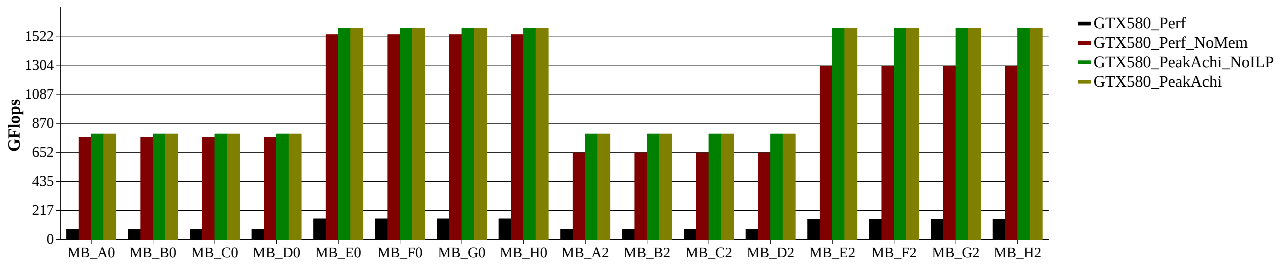


Figure 112: Illustrating the different levels of performances for FP-only and mixture benchmarks.

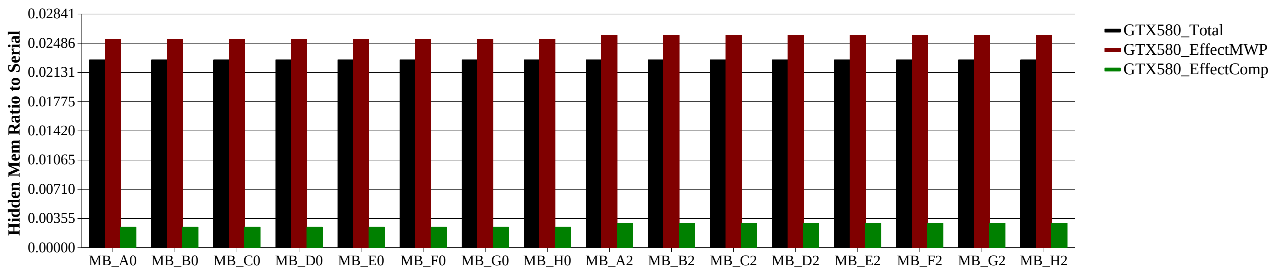


Figure 113: Illustrates the effect of the overlapping memory accesses as a ratio from the serial version.

5.4.2 Categories

The benefits of deriving the model from bottom-up and decomposing terms is that this enables visualization and categorization of an application. By using N and MWP-affecting parameters, two big categories can be proposed: one in which enough threads (warps) per core (N) exists, and the other one that does not have sufficient threads to completely hide the memory-waiting cycles (i.e., N >= MWP_App_InfiN). Note that even if not enough warps exist per core to completely hide the idle memory cycles, memory bandwidth saturation as well as other performance-affecting problems can still occur.

Cases 4 and 9 show the case that even though MWP_App is greater than MWP_BW, it is not too far off, and hardware is almost capable of sustaining that demand. Hence, this is put into a separate category.

```
SUB-CATEGORY I

# Case that N < MWP_App_InfiN
# Having insufficient number of warps to hide memory-latency completely

Case 1
((N < MWP_App_InfiN) and (MWP_Process < MWP_App) and (MWP_BW < MWP_App))
(Bad) Number of warps per SM is not enough for an application
(Bad) Memory problem from MWP_BW and MWP_Process
(Note) Not much you can do unless compiler optimization
(Note) Increase N does not help much as memory problem will occur
       but at least some computations can execute in between
(Note) Reduce core frequency

Case 2
((N < MWP_App_InfiN) and (MWP_Process < MWP_App) and (MWP_BW >= MWP_App))
(Bad) Number of warps per SM is not enough for an application
(Bad) MWP_Process problem with N. Improve memory type
      Reducing cores does not help really because single mem. latency issue from one sm
(Note) Increase N does not help much as memory problem will occur
       but at least some computations can execute in between
(Note) Reduce core frequency

Case 3
((N < MWP_App_InfiN) and (MWP_Process >= MWP_App) and (MWP_BW < (0.85 * MWP_App)))
(Bad) Number of warps per SM is not enough for an application, but not too much
(Bad) MWP_BW problem
(Note) Use other device with larger BW, reduce active cores & data usage per thread
(Note) Increase N does not help much as memory problem will occur
       but at least some computations can execute in between

Case 4
((N < MWP_App_InfiN) and
(MWP_Process >= MWP_App) and (MWP_BW >= (0.85 * MWP_App)) and (MWP_BW < MWP_App))
(Bad) Number of warps per SM is not enough for an application
(OK) MWP_BW may be just enough
(Note) Use other device with larger BW, reduce active cores & data usage per thread
(Note) Increase N does not help much as memory problem will occur
       but at least some computations can execute in between
(Note) Fourth Best Case !

Case 5
((N < MWP_App_InfiN) and (MWP_Process >= MWP_App) and (MWP_BW >= MWP_App))
(Bad) Number of warps per SM is not enough for an application
(Good) No HW saturation problem in Memory BW and Memory Process from N perspective
(Note) Increase N helps definitely
(Note) Reduce core frequency
(Note) Third Best Case !
```

Figure 114: Categorization of an application using model insights: N < MWP_App.

```
SUB-CATEGORY II

# Case that N >= MWP_App_InfiN
# Having sufficient number of warps

Case 6
((N >= MWP_App_InfiN) and (MWP_Process < MWP_App) and (MWP_BW < MWP_App))
(Good) Enough number of warps (N) to hide memory latency
(Bad) Memory problem from MWP_BW and MWP_Process.
      Memory saturation overlapping problem, could result in extra cycles
(Note) Computations to fit idle cycles are enough
       But memory problems could result in more idle cycles
(Note) Possibly reduce N to reduce stressing register/fetch/decode/scheduler

Case 7
((N >= MWP_App_InfiN) and (MWP_Process < MWP_App) and (MWP_BW >= MWP_App))
(Good) Enough number of warps (N) to hide memory latency
(Bad) Memory problem from MWP_BW and MWP_Process.
      Memory saturation overlapping problem, could result in extra cycles
(Note) Computations to fit idle cycles are enough
       But memory problems could result in more idle cycles

Case 8
((N >= MWP_App_InfiN) and (MWP_Process >= MWP_App) and (MWP_BW < (0.85 * MWP_App)))
(Good) Enough number of warps (N) to hide memory latency
(Bad) MWP_BW problem. Bandwidth problem
(Note) Use other device with larger BW, reduce active cores & memory data usage per thread

Case 9
((N >= MWP_App_InfiN) and
(MWP_Process >= MWP_App) and (MWP_BW >= (0.85 * MWP_App)) and  (MWP_BW < MWP_App))
(Good) Enough number of warps (N) to hide memory latency
(OK) MWP_BW may be just enough
(Best) Second best case !
       Further improvement comes from compiler optimization

Case 10
The Best Case !
((N >= MWP_App_InfiN) and (MWP_Process >= MWP_App) and (MWP_BW >= MWP_App))
(Good) Enough number of warps (N) to hide memory latency
(Good) No HW saturation problem in Memory BW and Memory Process from N perspective
(Best) The best case.
       Further improvement comes from compiler optimization
```

Figure 115: Categorization of an application using model insights: N >= MWP_App.

5.4.3 Power Measurements on GTX580

Figure 116 shows the measured power data for the memory-intensive benchmarks. The raw power data has been processed to produce 16 points, where each point represents a different number of active cores. The first point refers to using only one active core, whereas the last point refers to using all 16 cores. The benchmark, CMEM, is computationally intensive, so this does not saturate bandwidth, while the benchmarks DMADD and DOTP highly saturate memory bandwidth. The benchmarks MADD and MMUL fall in between. Note that regardless of whether or not a benchmark saturates in memory behavior, the consumed power is always increased with an increasing number of cores. Contrary to computations-only benchmarks, these memory-intensive benchmarks vary more widely in their power numbers with a delta of 170 W (computations-only delta is about 60 W). This power data is used to produce energy-efficiency results in the subsequent sections.

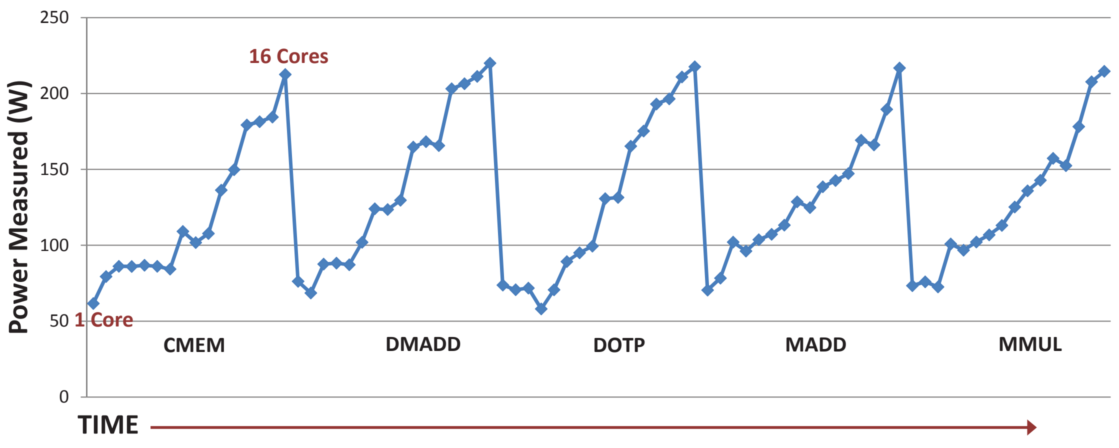


Figure 116: Processed power measurement data on GTX580 GPU.

5.4.4 More Benchmark Analysis on Bandwidth

First, the bandwidth evaluating benchmarks from the previous power work are used for testing memory bandwidth. However, because of the cache in GTX580, the benchmarks are modified while keeping the original structure as much possible. Figure 117 shows a portion of code that shows the kernel. The memory access depends on the previous loaded value. Regardless of whether it is a multiplication or an addition, note that the stride is always 16384. By confirming with the profiler, this stride makes sure that the memory access always misses both in L1 and L2 caches. The reason for this high stride is that from a core's perspective, stride that is bigger than a cacheline size (i.e., 128-bytes) is sufficient. However, because L2 is shared and the other cores *already* used that index earlier, simply increasing the stride does not work.

```
// Dmadd Kernel. Each memory data is pre-allocated with the value of 8192, overall stride is 16384
for (int i=0; i<LOOPITER; i++)
{
  loadedvalue = dm_input1[index] + dm_input2[index];
  index += loadedvalue;
  loadedvalue = dm_input1[index] + dm_input2[index];
  index += loadedvalue;
  loadedvalue = dm_input1[index] + dm_input2[index];
  index += loadedvalue;
}

// Dotp Kernel. Each memory data is pre-allocated with the value of 128, overall stride is 16384
// 128 * 128 = 16384
for (int i=0; i<LOOPITER; i++)
{
  loadedvalue = dm_input1[index] * dm_input2[index];
  index += loadedvalue;
  loadedvalue = dm_input1[index] * dm_input2[index];
  index += loadedvalue;
  loadedvalue = dm_input1[index] * dm_input2[index];
  index += loadedvalue;
}

// Cmem Kernel. Each memory data is pre-allocated with the value of 16384
// Overall stride is 16384
for (int i=0; i<LOOPITER; i++)
{
  loadedvalue = dm_input1[index];
  index += loadedvalue;
  FMAD4(sum, multipler)
  FMAD4(sum, multipler)
  loadedvalue = dm_input1[index];
  index += loadedvalue;
  FMAD4(sum, multipler)
  FMAD4(sum, multipler)
  FMAD4(sum, multipler)
  FMAD4(sum, multipler)
}

// Mmul Kernel. Each memory data is pre-allocated with the value of 8192 with multipler of 2
// Overall stride is 16384
for (int i=0; i<LOOPITER; i++)
{
  sum = multipler * dm_input1[index];
  index += sum;
  sum = multipler * dm_input1[index];
  index += sum;
  sum = multipler * dm_input1[index];
  index += sum;
}

// Madd Kernel. Each memory data is pre-allocated with the value of 11384 with adder of 5000
// Overall stride is 16384
for (int i=0; i<LOOPITER; i++)
{
  sum = adder + dm_input1[index];
  index += sum;
  sum = adder + dm_input1[index];
  index += sum;
  sum = adder + dm_input1[index];
  index += sum;
}
```

Figure 117: The set of code designs to control cache accesses in GPU Fermi architecture with L1 and L2 caches.

Figure 118 shows the version that has the L1 and L2 cache hit, so that the performance scales linearly with more cores even if an N of 32 and all 16 cores are used. On the other hand, Figure 119 shows the case when DRAM is stressed 100% of the time for each memory instruction.

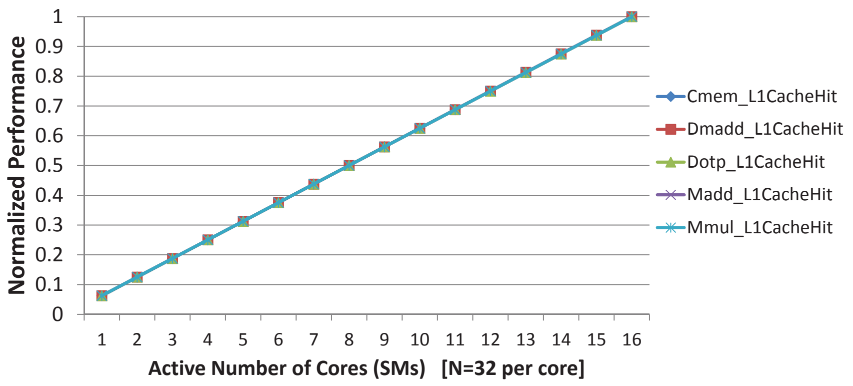


Figure 118: Benchmarks with L1 hit. Note that even with many cores, the performance scales linearly.

Figure 119 shows multiple performance versions varying N from 8 to 32. A different N means a different number of allocated hardware threads per core (SM), and the benefit is that when there is a long-latency instruction such as memory access, a processor switches to the other warps and continues execution. What we expected is that as N is increased, more parallel memory requests are put in the system.

According to the expectation, when N is small (N = 8), the performance scales linearly even if the memory system is intensively used (i.e., each memory access is accessing DRAM and verified in the CUDA profiler). However, as N is increased, DMadd and Dotp start to show some degradations, because of their higher memory strength per thread (i.e., MSTR). The degradation becomes clearly visible at N = 24. Interestingly, at N = 32, Cmem still scales linearly, while Dmadd and Dotp show significant performance degradations. The performances of Madd and Mmul fall in between.

One significant improvement from the previous model used for performance and power is that the previous model always predicted the same number of cores that saturates bandwidth. This is because the model assumes that at any given moment of time, each warp is requesting one memory transaction. However, this is not always so, because there can be multiple independent memory requests as in the Dmadd case, illustrated in Figure 99. Furthermore, even knowing this issue would not be sufficient since integrating into the model would not be very straightforward.

Another contribution of this new model is the following example. Even if those independent memory requests are handled, however, if an application has so much computation that not many warps can be overlapped anyway, then the application's *demand* for MWP should be lower than expected. Hence, the MWP_App metric is proposed to solve the above two cases: independent memory requests and actual memory demand considering the amount of computation (i.e., a very computation-intensive benchmark does not demand high bandwidth). In other words, MWP_App indicates how many overlapping memory requests are at demand, and when hardware supports the sufficient bandwidth (i.e., MWP_BW is sufficient), then there is no bandwidth problem. Furthermore, we can even quantify the degree of bandwidth saturation (i.e., how much MWP_App is greater than MWP_BW) with the new model.

Because of these contributions, Figure 121 shows the case where MWP_App is different depending on the application. Dmadd and Dotp show much higher bandwidth demand than the rest. For the first figure, MWP_BW is much higher than MWP_App; hence this shows the case in Figure 119. However, as N is increased, MWP_App approaches MWP_BW, which in turn puts pressure on the memory system and performance degradation starts to occur.

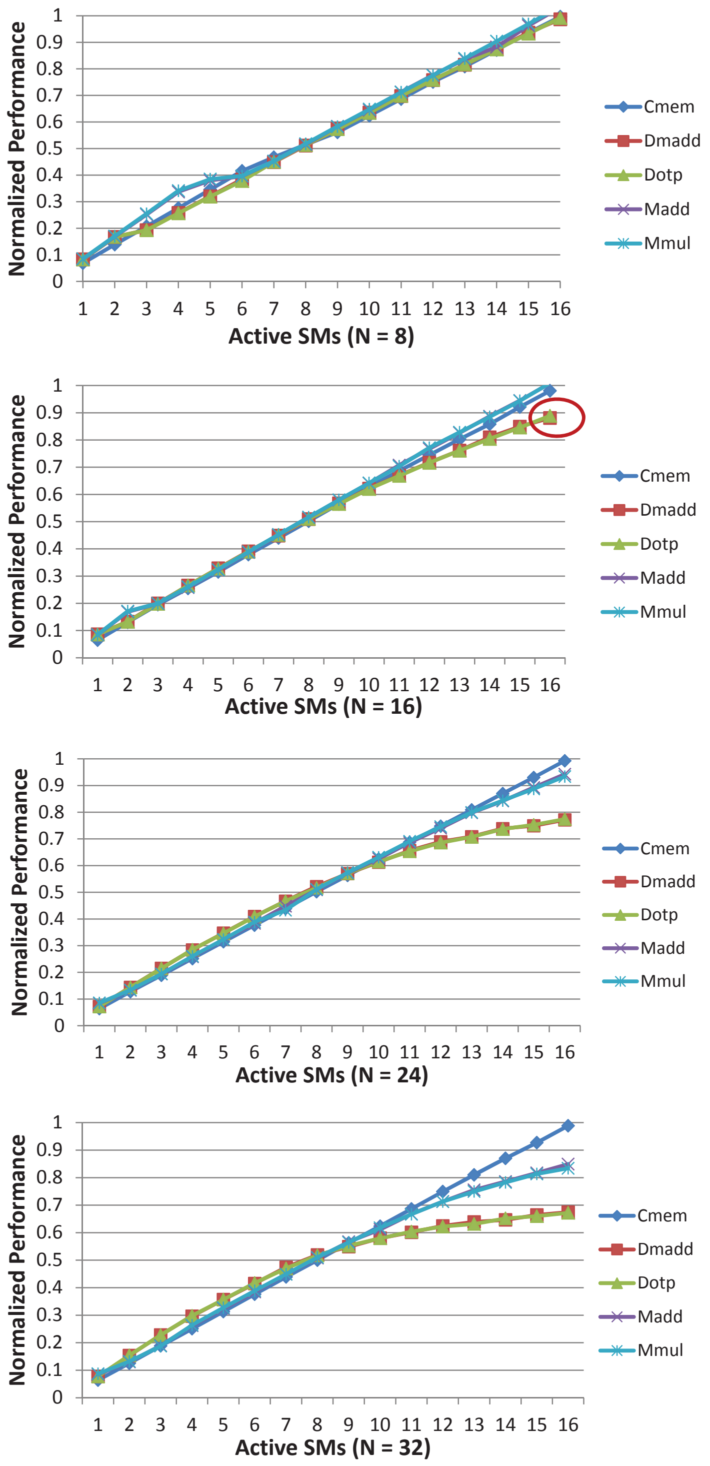


Figure 119: Performance data for memory-intensive benchmarks. Each benchmark is normalized to the best expected performance if no BW saturation has occurred. The Cmem performance does not degrade with increasing N, while Dmadd's performance degrades significantly. The Madd's performance degradation is in between Cmem and Dmadd.

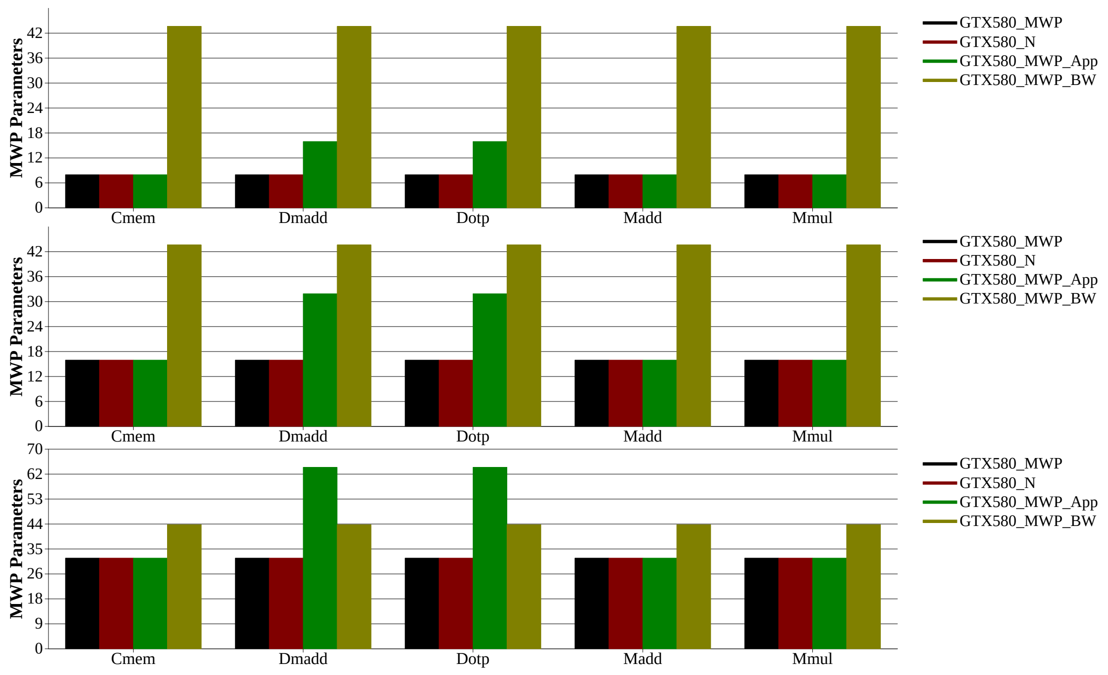


Figure 120: Effects of different N on MWP_App vs. MWP_BW, where N values are 8, 16, and 32. Dmadd and Dotp significantly overpass MWP_BW, while the rest do not. This effect can be correlated to Figure 119.

Figure 121 shows the raw normalized performance data for these benchmarks. As these show actual performance data, three categories can be clearly seen. For Cmem, there is no saturation effect at all, while Dmadd shows severe performance degradation as N is increased. For Madd, the model predicts that there is no memory saturation effect; however, we can see a slight performance degradation for N = 32. For this reason, we make a change in the category that when MWP_App is very similar to MWP_BW within 15%, we project that saturation is not severe, but can be slightly observed.

Figures 122 and 123 show a different perspective of analyzing the memory effect. The x-axis is the number of active cores, while the y-axis shows the corresponding MWP values. For Cmem, both the new model and old model predicts that memory bandwidth is enough. However, for Dmadd, the previous model predicts otherwise, while the new model predicts a bandwidth problem. The reason is that the previous model compares MWP_BW to N, while the new model has a new metric, MWP_App, that considers both computation amount and memory strength (MSTR). The new model can even detect two cases when N = 16 and N = 32. When N = 16, MWP_BW nearby approaches MWP_App, so a little bit of bandwidth saturation can occur. However, when N = 32, much degradation will occur because MWP_BW is much smaller than MWP_App.

Figure 124 shows another perspective. This time N is changed. As N is changed, we can see that MWP_App for Dmadd and Cmem are increased at a different rate. If the MWP_App term hits the MWP_BW line (the MWP_BW line is drawn at different active cores), this is the maximum N that avoids further bandwidth saturation. The benefit of using the model this way is that given an application, the optimum number of N can be found, not just finding the optimal number of active cores for an application. Hence, rather than only predicting active cores, the benefit of the new model is that all the terms can be rearranged to predict and suit the purpose.

5.4.5 MWP Metrics Usability and Insights

Figure 125 shows that the model predicts different categories with different N, even for the same benchmarks. As N is increased, much pressure is put on the memory system. Hence, the categories are shifted. The benefit of using categorization is that any benchmark can be categorized from a modeling perspective, which is unique compared to other categorization mechanisms. On the other hand, Figure 126 shows the changing the active number of cores for a fixed N value. For the benchmarks that demand a high bandwidth such as Dmadd and Dotp, the category number is smaller than other benchmarks' values (i.e., the higher the better). When N is 48, then because of even more bandwidth demand, the categories become even more smaller than when N is 24.

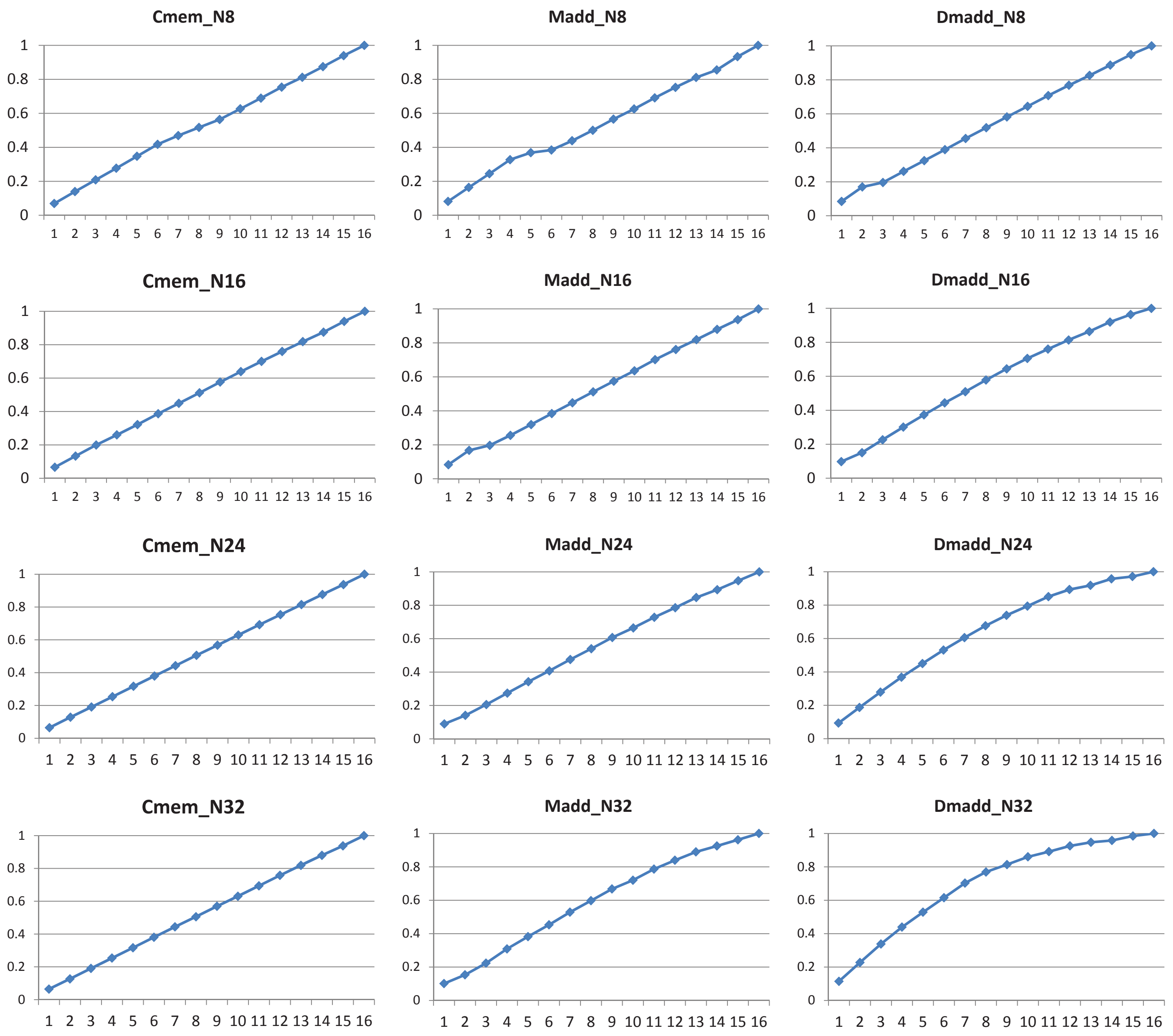


Figure 121: Normalized performance data for memory benchmarks. Each benchmark is normalized to its best performance point. The Cmem performance does not degrade with increasing N, while Dmadd's performance is degraded significantly (i.e., not linear) with increasing N. The Madd's performance degrades at a rate that is between Cmem and Dmadd.

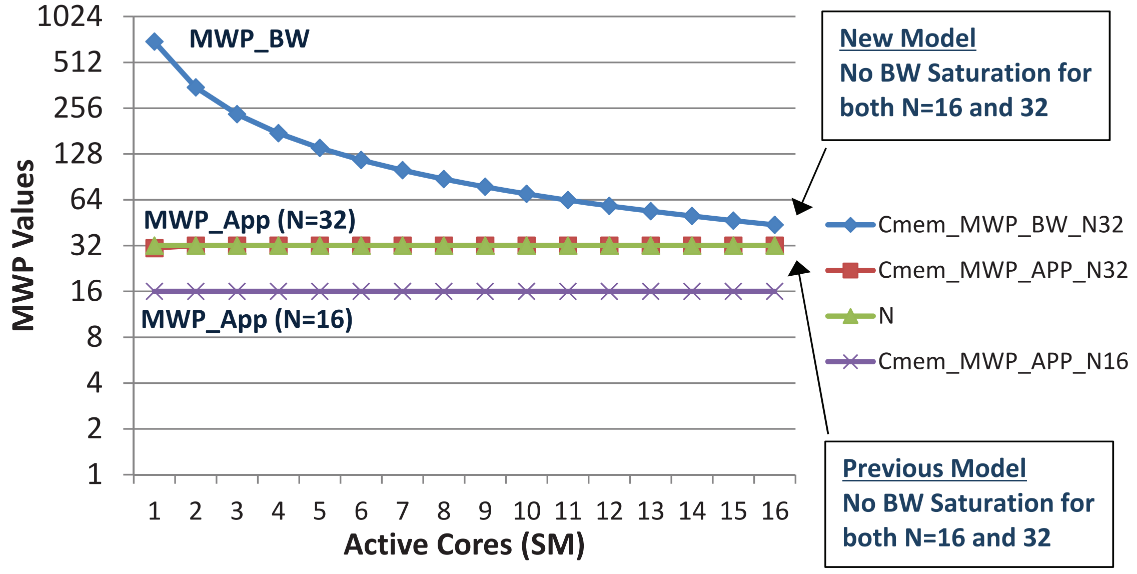


Figure 122: Memory-affecting Parameters Graph: Cmem.

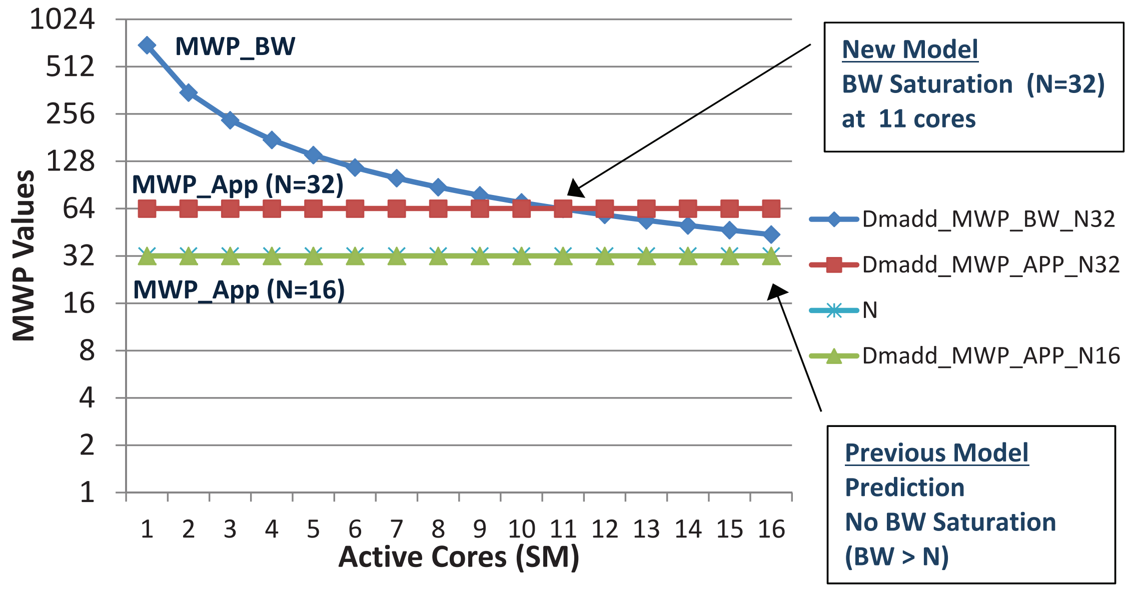


Figure 123: Memory-affecting Parameters Graph: Dmadd.

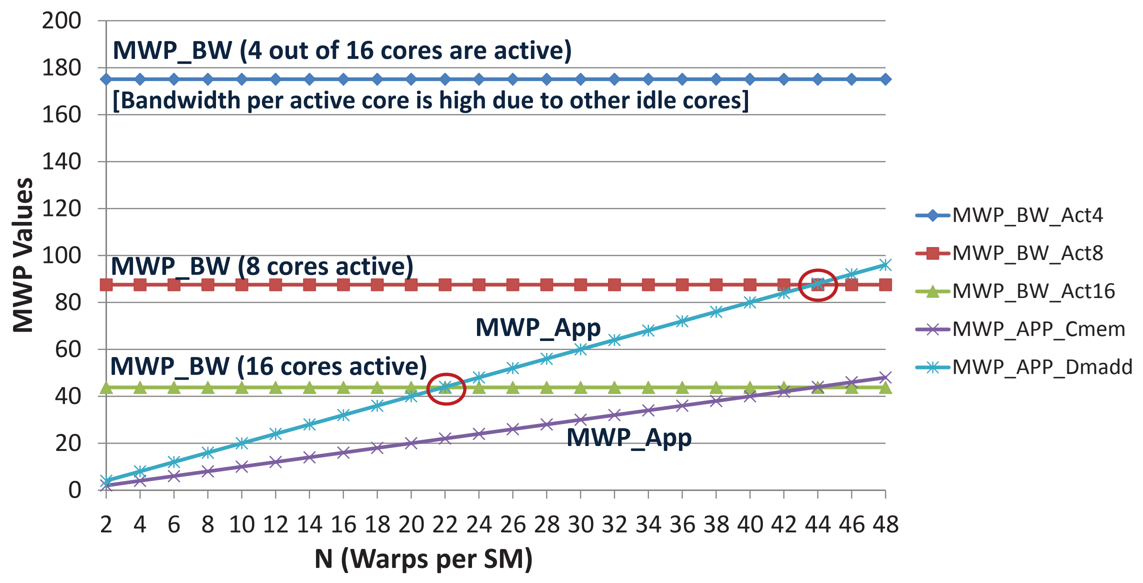


Figure 124: Memory-affecting parameters from different perspective, changing N.

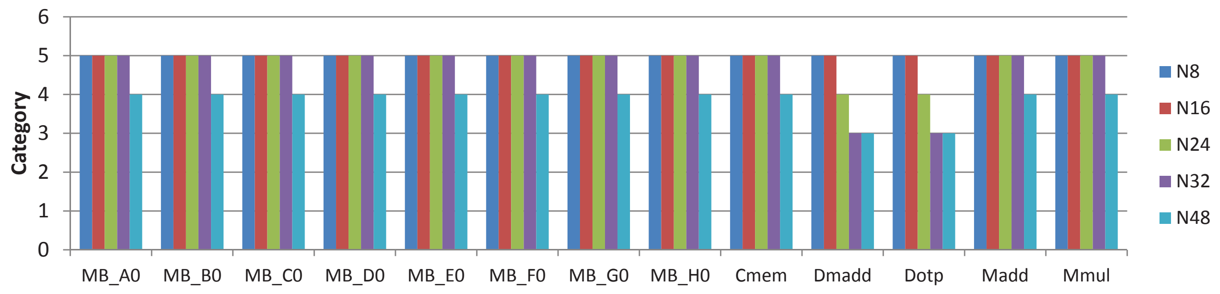


Figure 125: Categories affected with respect to changing N. The category of five is the best case, as described in Figure 114.

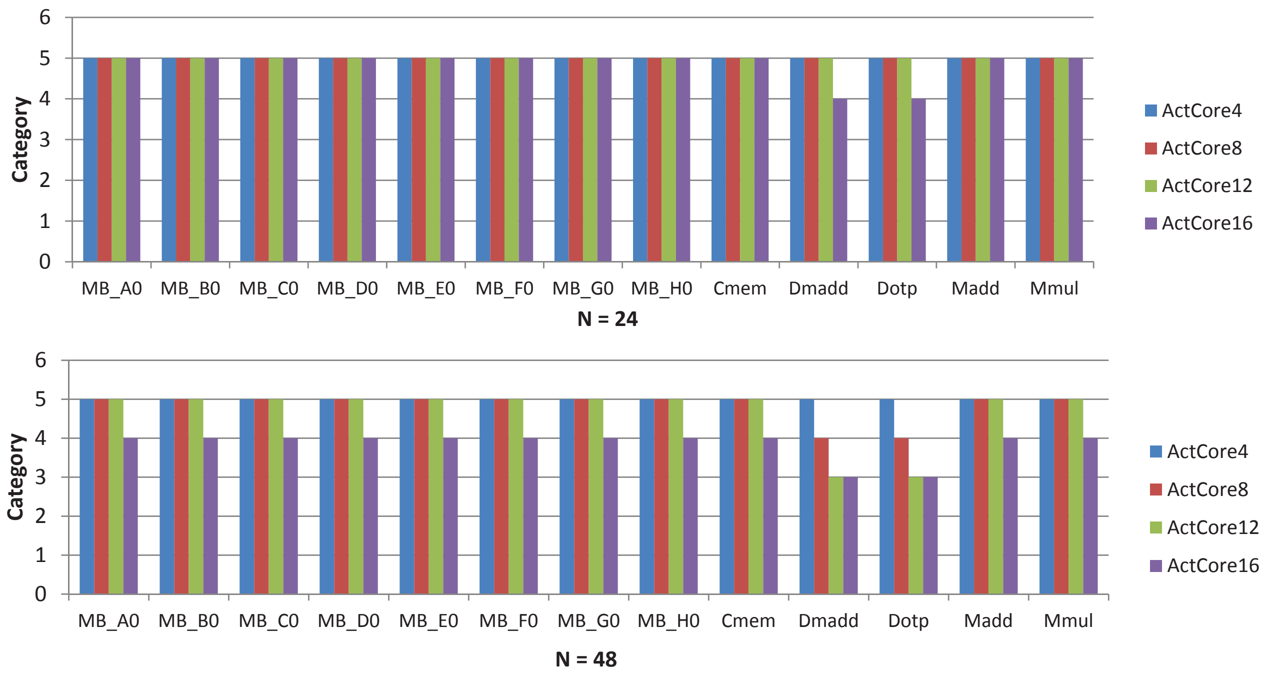


Figure 126: Categories with respect to changing active cores with different N (warps per SM). The category of five is the best case, as described in Figure 114.

5.5 *How to Derive Quantitative Model Suggestions?*

The model that is restructured and improved has high benefits and extendability as feedback to a programmer, compiler, and even to a runtime system. This work is not based on on machine-learning or profiling mechanisms. Furthermore, this work is generic enough that it can be applicable to other architectures such as CPUs. This section discusses how this model derives for giving further suggestions that can improve not only performance, but also less peak power and even optimum energy efficiency.

5.5.1 Finding the Optimum Number of Active Cores

The basis for this suggestion is that if too many cores are active, since each core consumes some portion of available memory bandwidth, the overall bandwidth might not be enough for *all* threads or warps inside a core. Even if this happens, the model (especially the improved version) can quantify a different bandwidth *demand* for an application (i.e., independent memory accesses, amount of computations between memory accesses, etc.). Because the model identifies the demand bandwidth (i.e., MWP_App) of an application, and how much bandwidth a device can provide (i.e., MWP_BW), calculating the overall effect is possible. Hence, given the following situation (MWP_App > MWP_BW), the solution is either to (1) increase MWP_BW, or (2) decrease MWP_App. First, since (MWP > MWP_BW), how much to MWP_BW should be increased is derived as follows.

$$MWP\_BW\_new = \alpha \times MWP\_BW \tag{83}$$

$$MWP\_BW\_new \geq MWP\_App \tag{84}$$

$$\text{Therefore, } \alpha \geq (MWP\_App/MWP\_BW) \text{ where } \alpha \geq 1 \tag{85}$$

The α term tells how much MWP_BW should be increased. Since the MWP_BW term is represented by the following equations, either the numerator is increased by

α, or the denominator should be decreased by $1/\alpha$.

$$\begin{aligned} MWP\_BW\_new &= \alpha \times MWP\_BW && (86) \\ &= \frac{\alpha \times Mem\_Bandwidth}{BW\_per\_warp \times \#ActiveSM} && (87) \\ &= \frac{Mem\_Bandwidth}{(1/\alpha) \times BW\_per\_warp \times \#ActiveSM} && (88) \end{aligned}$$

This representation shows that either the device with larger memory bandwidth should be used instead if available, or BW_per_warp or $\#ActiveSM$ should be reduced (i.e., divided by α). The first method requires code changes, while the latter option of reducing number of active cores requires either code techniques or using special internal API calls, which are not publicly available.

The model's improved prediction mechanism has the clearest advantage when the performance-per-watt metric is plotted. Figure 127 shows four cases. The y-axis shows the normalized performance-per-watt value (higher is the better), and the x-axis shows the number of active cores. When N is eight, both the previous model and the new model predict using the maximum number of cores to maximize performance per watt, as well as for N equals 16. However, when N is 24, the new model predicts 13 cores for Dmadd and Dotp (the rest is still the maximum number of cores), while the previous model still predicts 16 cores. Finally, when N is 32, the new model shifts its previous prediction further to the left and predicts ten cores, while the previous model predicts 16. The only time that the previous model predicts fewer than the maximum number of cores is when N is equal to 48. However, even in this case, the previous model predicts the *same* number of cores like the previous work. The new model not only predicts a different number of cores, but even distinguishes between different benchmarks.

Figure 128 compares the energy efficiency as a bar graph. The figure shows that Dmadd and Dotp show the worst prediction from the previous model, while Cmem's prediction is correct. Madd and Mmul fall in the mid-range. This shows that improvement in the MWP_BW and MWP_App metrics lead to better predictions and thus better energy efficiency.

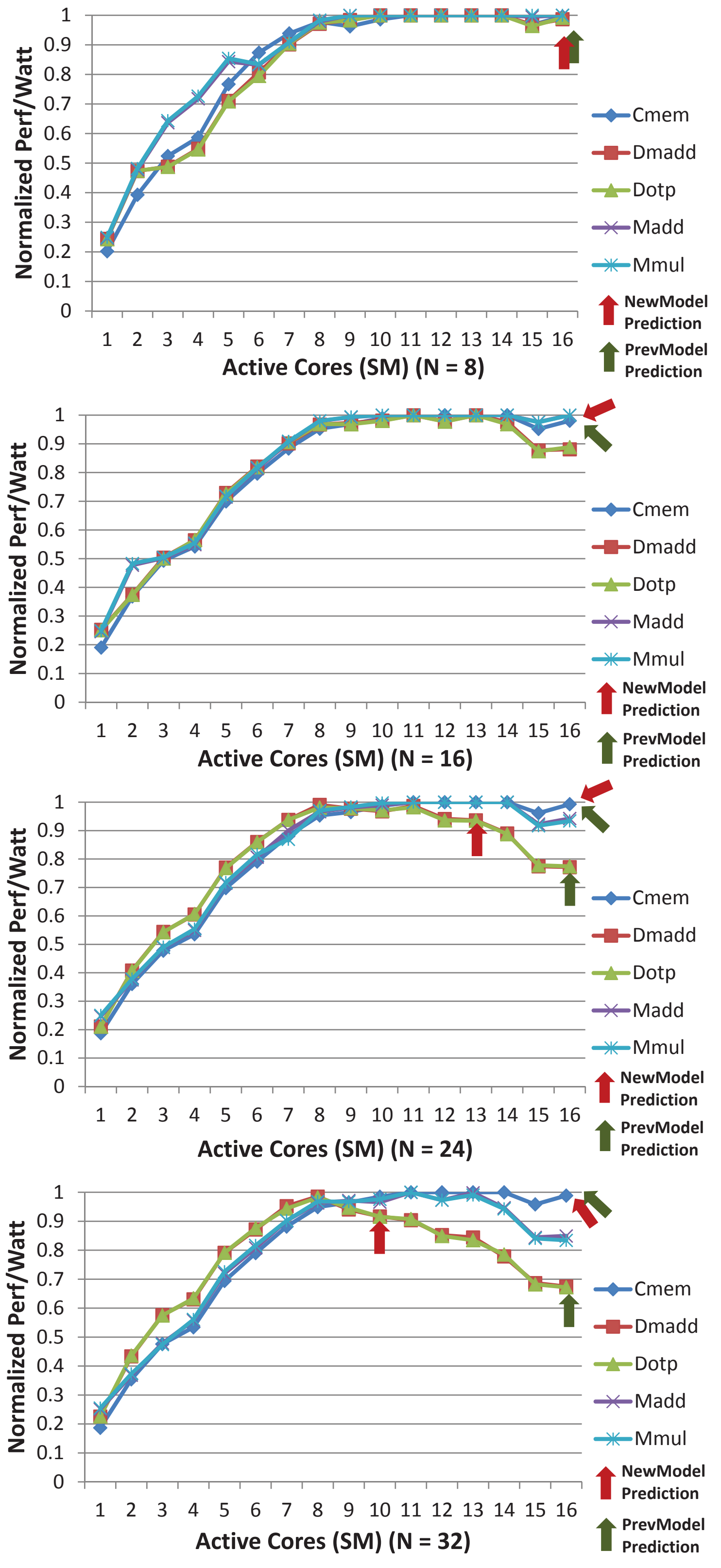


Figure 127: Energy-efficiency graph vs. N between the previous and the new model. Each line is normalized to the benchmark's best performance point. The previous model predicts all cores, while the new model predicts a different number of cores for the application, and even the accuracy is better.

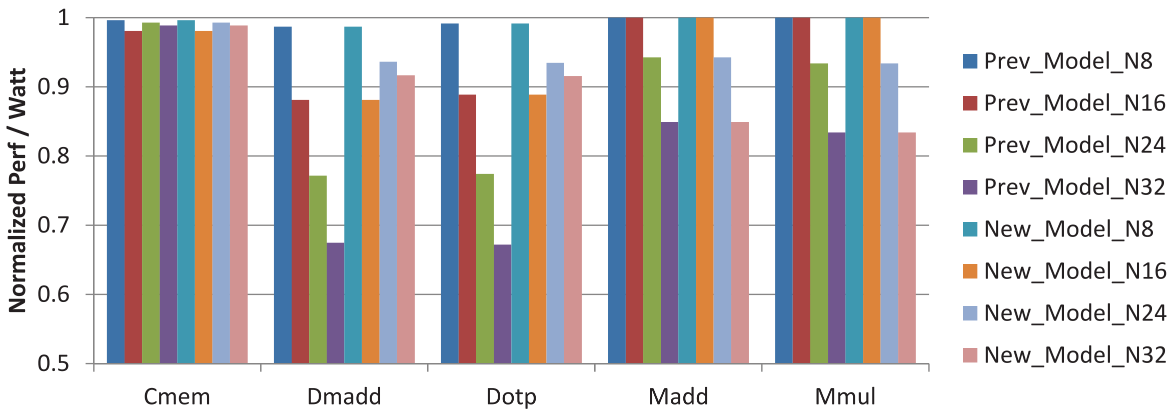


Figure 128: Energy-efficiency predictions between the previous and the new model. The efficiency value of one is the best. On average, the new model achieves 94.76% toward optimum efficiency, while the previous model achieves 90.09%. For the worst case, the new model achieves 89.91%, and the previous model achieves 79.5% for N = 32.

5.5.2 Better Number of Threads per Core (N)

The new model suggests the number of active cores to maximize energy efficiency, and the model terms can also be arranged to suggest a different N (i.e., related to occupancy). Similar to the reasoning in Section 5.5.1, the derivation is as follows, with the only difference being decreasing MWP_App (such that $MWP\_App \leq MWP\_BW$) rather than increasing MWP_BW.

$$MWP\_App\_new = \alpha \times MWP\_App \tag{89}$$

$$MWP\_App\_new \leq MWP\_BW \tag{90}$$

$$\text{Therefore, } \alpha \leq (MWP\_BW/MWP\_App) \text{ where } \alpha \leq 1 \tag{91}$$

Since MWP_App is calculated as follows, the easiest method to control MWP_App is by decreasing the N value by (1 - α)%.

$$MWP\_App = MIN(MWP\_App\_InfiN, N) \times MSTR \tag{92}$$

When this suggestion is applied to the benchmarks, the result in Figure 129 shows that the energy efficiency is preserved quite well except for a little degradation for Dmadd and Dotp. Nevertheless, this is much better than about a 30% degradation as shown in the fourth case of Figure 127. Note that energy efficiency is preserved even if all the active SMs of 16 are used.

5.5.3 Core Frequency Reduction

This new model can provide further insights to better energy efficiency. When memory-waiting idle cycles are significant, the core performance such as frequency can be reduced. The question is how to derive this term from the analytical model. The assumption is that memory clock is not reduced, but only the core frequency is changed.

Figure 130 shows that only two warps exist, so the MWP_App_InfiN metric (not

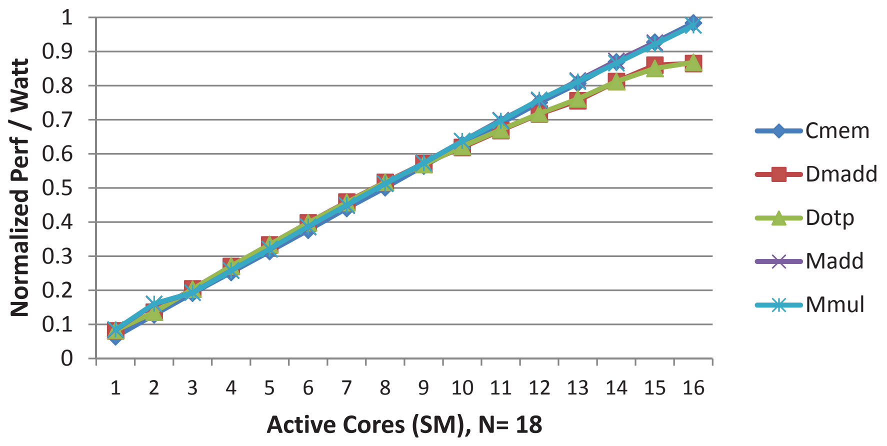


Figure 129: Activating all 16 SMs but just using N = 18 as the model suggested. Note that performance degradation for Dmadd and Dotp is not severe, about 10%, compared to using N = 32, which degrades performance by more than 30%.

MWP_App that considers N) is higher than N, which means there will be idle memory cycles because not enough computations and N exist to hide all idle memory cycles. Quantitatively, the figure shows that only one (C2) computation is hiding M1's memory access, while three threads are needed. Since the ratio is three to one, the computation core frequency can be reduced by a factor of three. Equation (93) shows the derivation that specifies how much a core frequency can be reduced that minimizes performance degradation.

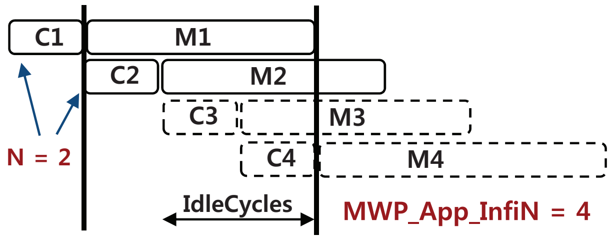


Figure 130: Quantifying the core reduction frequency.

$$CoreReductionPercent \approx (MWP\_InfiN - 1)/(N - 1) \qquad (93)$$

5.6 Case Study: Matrix Multiplication

Figure 131 illustrates two benchmarks: naive and tiled matrix multiplications in a performance-space graph. A performance can be anywhere from 0 to 1600 GFlops for a GTX580 GPU. Despite this large range of possible performances, two major contributions from the model can be stated. First, the model successfully predicts the actual measured performances in the correct range. Second, one interesting fact can be derived from the prediction: the model is able to adjust the width from L1 and DRAM as block size is increased.

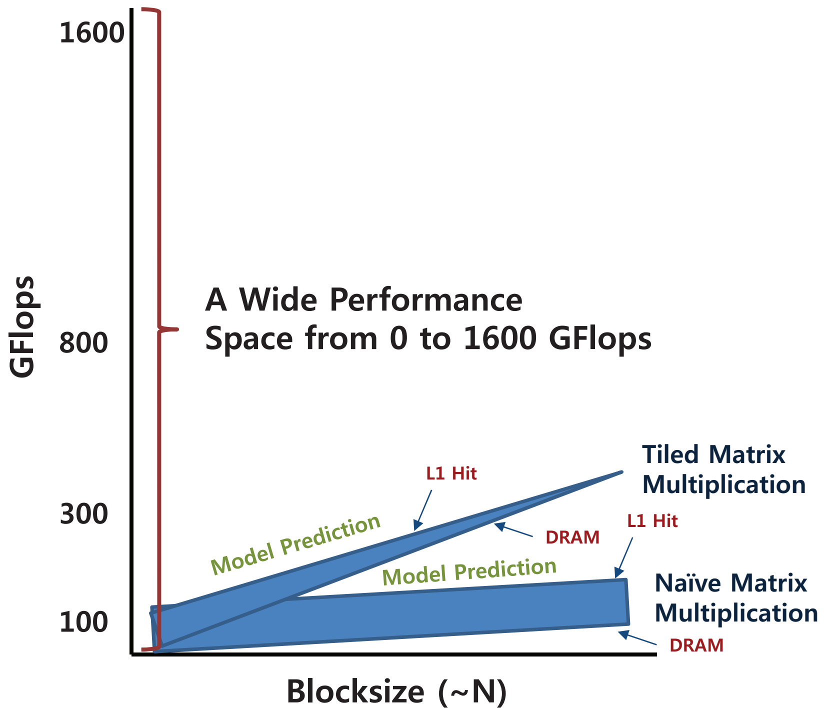


Figure 131: High-level illustration showing two different matrix multiplication implementations. The model not only predicts a correct performance range for different block sizes, but also spots the thinning gap effect between L1 and DRAM predictions lines.

For example, for the naive matrix multiplication, the distance between the L1-hit prediction line and the DRAM-prediction line is preserved. However, for the tiled matrix multiplication, the gap is thinning. The primary reason is that as block size is increased, for the tiled matrix multiplication, there are more computations with fewer global memory accesses *per warp*.[6] Since only a finite number of warps can be allocated per SM, effectively more computations can hide memory-waiting cycles as block size is increased, whereas for the naive matrix multiplication, the block size does not have any impact on the number of instructions per warp.

The model also predicts the L1-hit prediction and DRAM-only prediction lines. These performance prediction lines provide useful information to the compiler or a programmer, as this gives them a lower-bound and an upper-bound of performance. Furthermore, the model can even give future performance predictions, unique to an application, when one parameter, such as bandwidth and number of cores, is changed.

Figure 132 shows the naive matrix multiplication result. The line, MatNaive_Measured, is the measured data. The line, Model_High_L1, is the model prediction when all global memory accesses are L1-cache hits, while Model_High_DRAM and Model_DRAM lines represent if most global memory accesses are DRAM access, or all accesses are DRAM accesses.

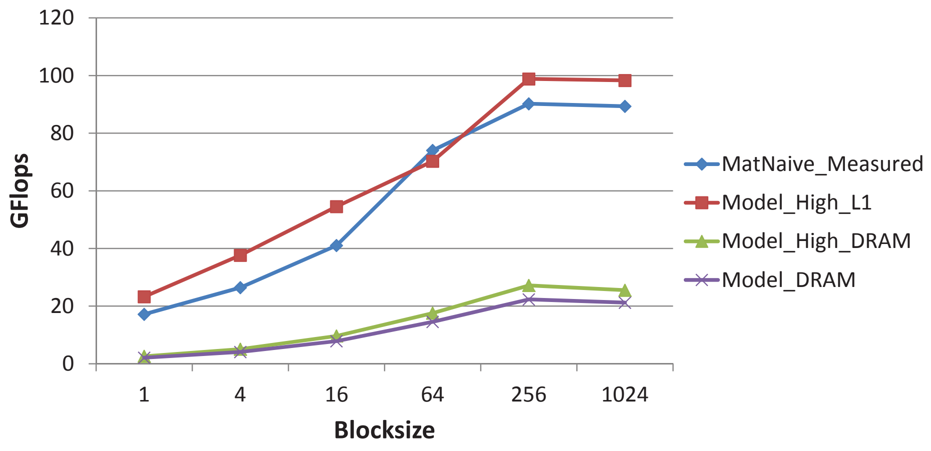


Figure 132: Naive matrix multiplication result. The model predicts high L1-cache hit, high DRAM-accesses, and all-DRAM accesses cases.

The first observation is that the gap between L1 and DRAM is much wider than the tiled matrix multiplication case. For this type of application, having a high cache

[6]With larger block size, more number of global memory accesses are changed to shared memory accesses, which we consider as computation accesses.

hit is very important. However, the upperbound (Model_High_L1) is not very high in the first place. Hence, for this type of application, making an algorithm change is suggested.

Figure 133 shows the profiler result for memory accesses. The graph shows the distribution ratio among L1, L2, and DRAM. The measured result closely follows the mostly-L1-hit line, which the model predicted. However, because the algorithm itself generated very low useful instructions to the total instructions ratio, even with a very high-cache hit in L1, the overall performance is not very good.

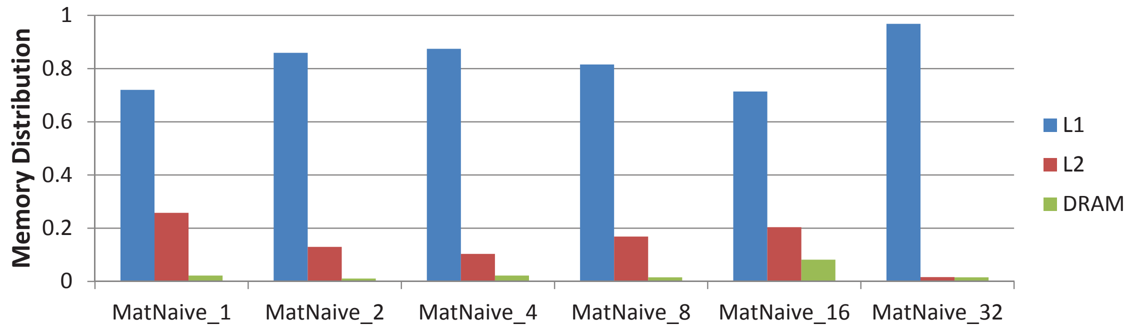


Figure 133: Profiler result for tiled matrix multiplication on GTX580, showing the memory distribution ratio among L1, L2, and DRAM.

Figure 134 shows MWP values. The figure shows that MWP_App is much higher than MWP_BW for block sizes of 16 and 32 (256, 1024 in other figures). As a result, this is going to saturate memory bandwidth. However, because of a large ratio of L1-cache hit, this saturation effect is not observed in the Fermi architecture.

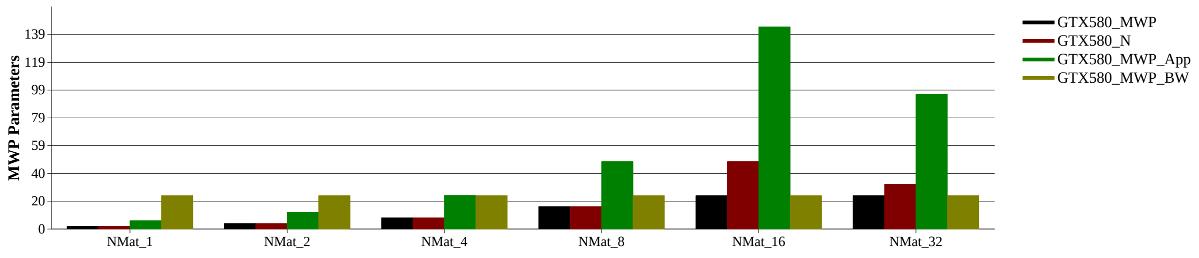


Figure 134: MWP values for naive matrix multiplication for GTX580. If cache did not exist, this benchmark would have generated a very high degree of memory saturation.

On the other hand, Figure 135 shows the result for the tiled matrix multiplication. The analysis is more complicated than the naive case for several reasons: the number of instructions is changed as block size is changed. Nevertheless, the model does predict a correct performance range since the measured data is inside the upper and lower bound lines.

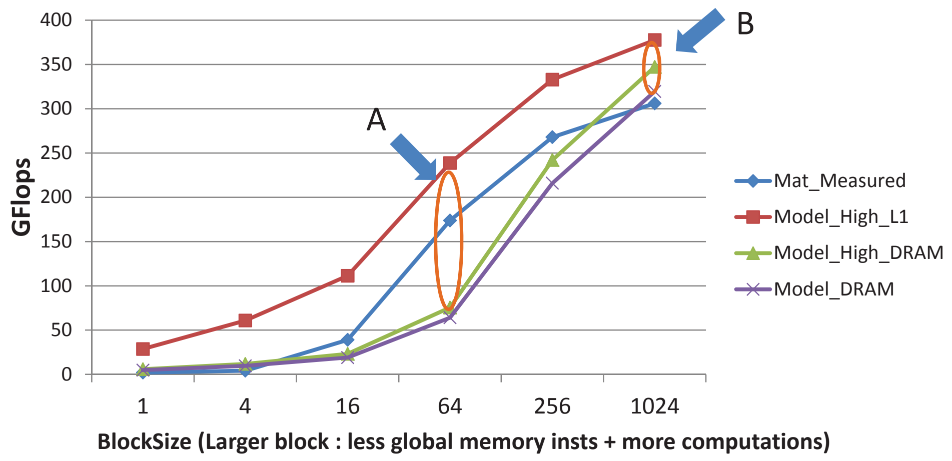


Figure 135: Blocked matrix multiplication result. The interesting case is that as the block size becomes larger, the number of global memory accesses is decreased, while the computations are increased per warp. Note the *decreasing* gap between L1 and DRAM as the block size is increased. The new model is able to spot this effect since the gap between the L1 and DRAM lines is getting smaller.

A few important things can be derived from the model prediction lines. First, the circle "A" shows the distance between Model_High_L1 and Model_DRAM, which is disproportionately wider compared to the circle "B" and even when compared to the naive matrix multiplication case. The reason is that as previously mentioned, as block size is increased, more computations with fewer global memory accesses exist *per warp*. Since N is limited per SM, for tiled matrix multiplication, effectively more DRAM-waiting cycles are hidden by computations. Hence, the insight from the model predictions only is that for this type of application where the gap is not significantly large, optimizing for better cache hits is not a good strategy.

Figure 136 shows the profiler result. The result shows that for small block size, which mimicks a naive matrix multiplication, the cache hit ratio is high. But as the block size is increased, memory accesses are mostly hit in L2 and DRAM. One possible analysis for why the measured data increases at the same rate as Model_DRAM (i.e., the measured data is not increased at a high rate) is that MWP_App is higher than MWP_BW at those points. These MWP values assume DRAM-only accesses, but we can assume this since the profiler result shows a high DRAM-accesses ratio.

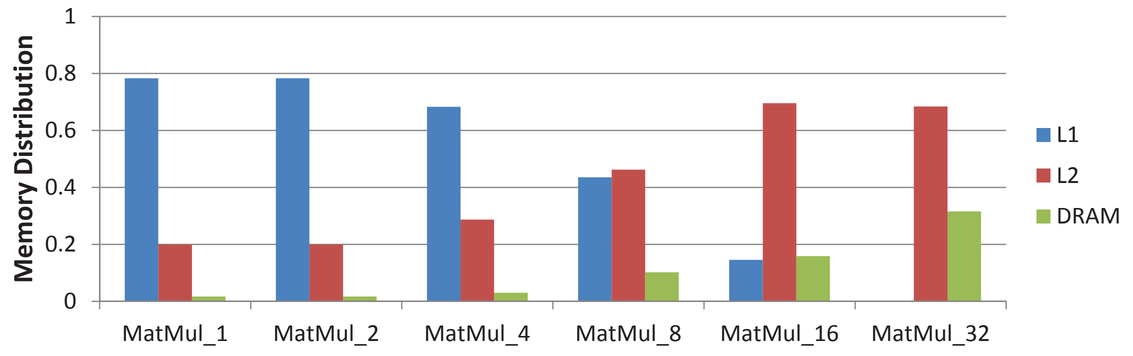


Figure 136: Profiler result for blocked matrix multiplication on GTX580, showing the memory distribution ratio among L1, L2, and DRAM.

Because of the high DRAM accesses for a block size of 256 and 1024, and MWP_App is higher than MWP_BW, as shown in Figure 137. We project that the memory system is much stressed. Hence, the increasing rate of the measured data is low when compared to the DRAM-only predictions from the model.

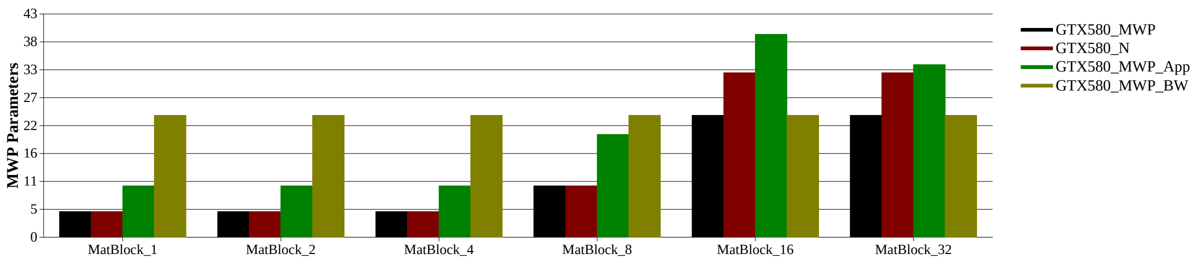


Figure 137: MWP values for the tiled matrix multiplication for GTX580. Note that assuming all DRAM accesses, the MWP_App is larger than MWP_BW for larger block sizes, generating memory bottleneck.

5.6.1 Category with Suggestions

The category information is related to the memory saturation effect. When most memory requests are L1-hit, the category output is the best case. As mentioned previously, the categories from one to five are one big category when not enough N exists (i.e., N < MWP_App_Infi), with number 5 being the best case (no memory saturation problem). The categories from six to ten are another big category that specifies that N is sufficient (i.e., N >= MWP_App_Infi). The number six is the worst case with the memory saturation problem, while the number ten is the best case.

Figure 138 shows the category information for the naive matrix multiplication. The top graph assumes most memory accesses were L1-cache-hit accesses, while the bottom graph assumes DRAM-intensive accesses. Because the model takes into account the idle DRAM cycles, the categories are different depending on memory access patterns.

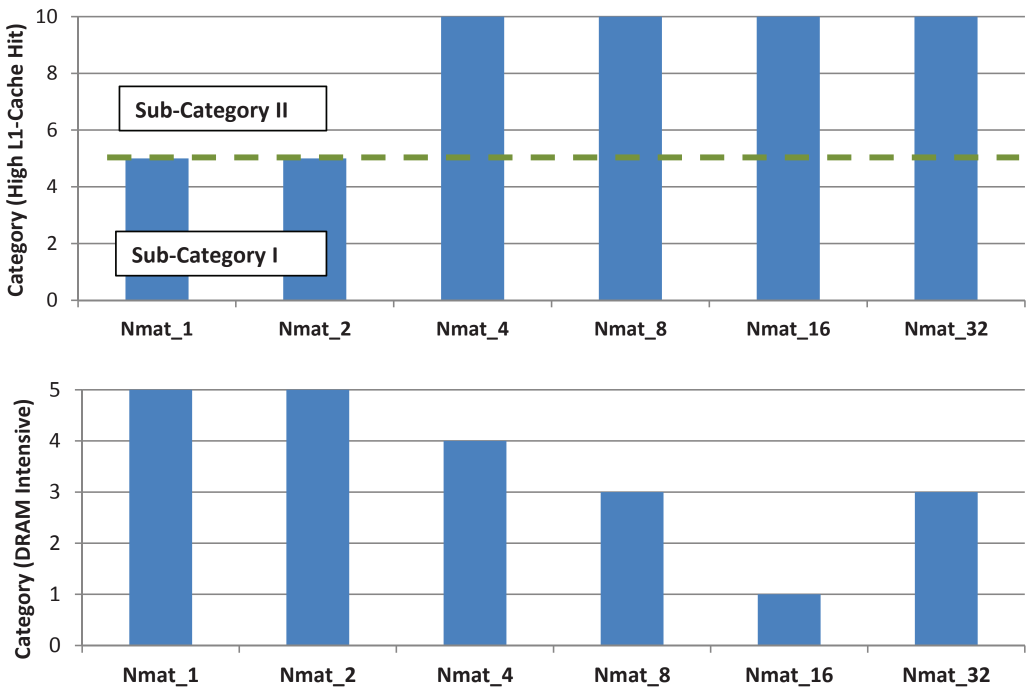


Figure 138: Category for naive matrix multiplication. Top: L1 intensive, Bottom: DRAM intensive. As the block size is increased, good categories are preserved for the L1-intensive case, while the categories become worse for the DRAM-intensive case.

The top figure starts with the category value of five, then ten. Both the values of five and ten are the best cases, with only the difference being in N. Since not enough N exists for Mat_1 and Mat_2 but memory accesses are good (i.e., L1 hit), the category of five is used. Later, when N is increased, the category value is changed to ten. However, for the DRAM-intensive case, the categories initially start with 5, but as much stress on the memory system emerges, the categories get reduced. The reason NMat_32 is higher than NMat_16 is that N for NMat_16 is higher since only *one* 1024 threads per block are assigned per SM, whereas multiple 256 threads per block can be assigned for the prior case.

Figure 139 shows the category information for the tiled matrix multiplication. Interestingly, the categories are improved with a larger block size. As previously mentioned, the number of global memory accesses and total computations *per* warp are reduced with increased block size. Hence, this is better for the memory system. Furthermore, as large computations are able to hide idle DRAM cycles effectively, the category is very much improved with larger block size. This case illustrates a very good case when an algorithm is well designed.

Table 13 shows the core predictions between the previous and the new model for L1-intensive and DRAM-intensive cases. The parentheses for the new model specifies the case where *all* memory accesses are DRAM accesses, whereas the numbers outside the parenthesis specify the case where most memory accesses are DRAM accesses. Note that for all DRAM access cases, the new model does adjust the number of cores to be even smaller than what it predicted earlier.

Figure 140 shows the expected energy efficiency. By using the same methodology from Figure 127 in which the new model predicts a better point for core prediction, the energy efficiency estimation is produced. The values are normalized to the default value that uses all number of cores. For example, each value of DRAM_Intensive and DRAM_only are normalized to their default configuration that uses all available cores

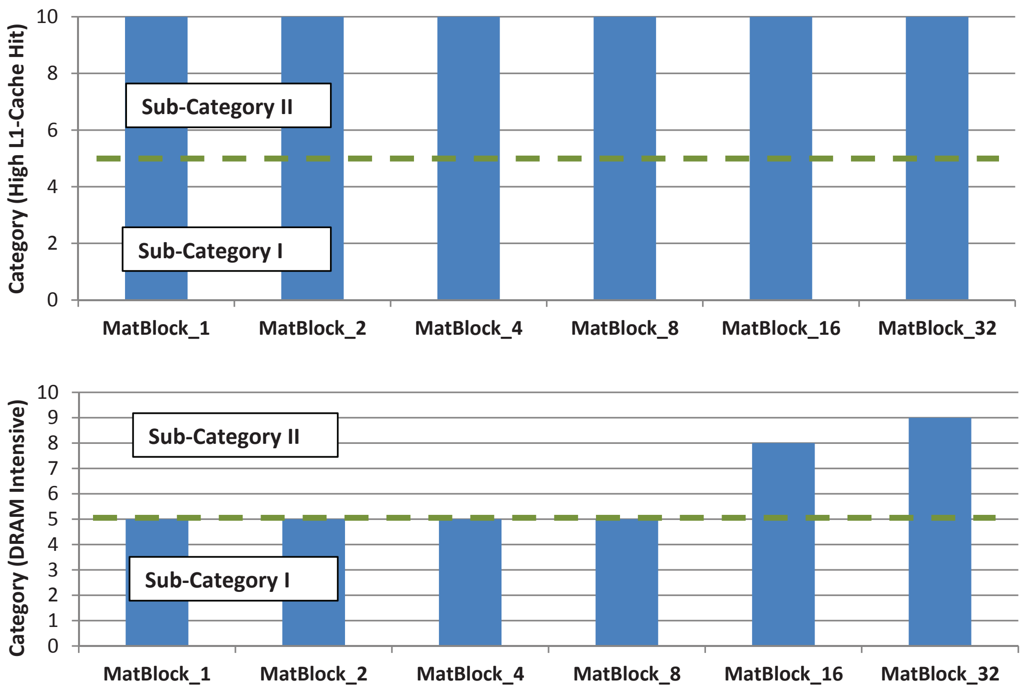


Figure 139: Categories for different block sizes in the tiled matrix multiplication. Top: L1 intensive, Bottom: DRAM intensive. Unlike naive matrix multiplication, the categories are improved for larger block sizes because the number of computations is larger, leading to hiding more idle memory cycles.

(16 cores for GTX580). The result shows that even for benchmarks that can hide idle memory cycles well like the tiled matrix multiplication, there is a room for improvement by reducing the number of cores, only one core for the DRAM_intensive case, and four cores for the pure DRAM accesses case. For naive matrix multiplication, the energy efficiency improvement will be more significant.

Table 13: Core predictions out of 16 total cores. P_Model means the previous model in Chapter II, N_Model means the new model. L1 specifies highly L1-intensive accesses, while DRAM specifies highly DRAM-intensive accesses. The number inside the parentheses predicts for *all* DRAM accesses.

Core Suggestion	P_Model L1	N_Model L1	P_Model DRAM	N_Model DRAM
NMat_4	16	16	16	16
NMat_32	16	16	11	4
NMat_4	16	16	16	16
NMat_16	16	16	11	13 (10)
NMat_32	16	16	11	15 (12)

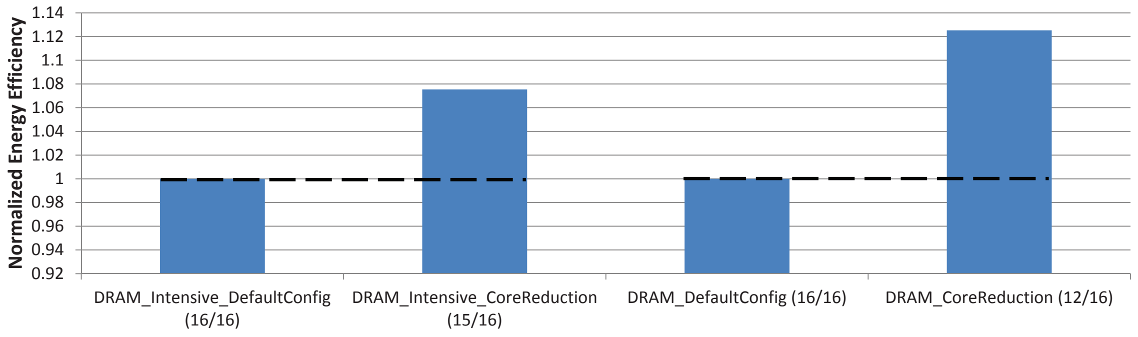


Figure 140: Expected projection of energy efficiency improvement for tiled matrix multiplication.

5.7 Model Parameters and Inputs

Table 14 shows the model parameters for a device. As illustrated, the model only considers essential high-level information of a device such as number of cores, DRAM latency, number of FP units per core, and bandwidth.

Table 14: Model parameters for GTX580 and Nehalem E5645.

	Frequency	DRAM Latency	NumCores	FP Units Per Core	Bandwidth
GTX580	2.4 GHz	450	1	32 (SIMT)	150
E5645 (x2)	1.544 GHz	250	12	2 (Non-SSE), 2 (4-wide SSE)	50

Table 15 shows the model input for the benchmark. Similar to the device input, the input for the benchmark is very compact. Since the model considers FP performance, the input shows many versions of FP instructions. The primary difference among these versions is whether a FP instruction is a *vectorizable* version or not. If it is vectorized, then number of floating-point operations will be calculated accordingly for CPU case, but for GPU case, this instruction is considered just like normal FP instruction. It is important to distinguish a fused-multiply (FMA) instruction as well since two floating-point operations are done per cycle for GPU, but there is no FMA instruction supported in CPU. Hence, for CPU, an FMA instruction is decomposed into two scalar FP instructions.

Table 15: Model parameters for benchmark input.

	Description
fp_insts	Number of scalar FP instructions per warp (thread)
fp_fused_insts	Number of fused multiply-add instructions per warp (thread)
fp_vec_insts	Number of vectorizable scalar FP instructions per warp (thread)
fp_vec_fused_insts	Number of vectorizable fused multiply-add instructions per warp (thread)
int_insts	Number of integer instructions per warp (thread)
br_misc_insts	Number of other instructions excluding FP, INT, Memory instructions per warp (thread)
mem_insts	Number of memory instructions per warp (thread)
mem_indep_strength	Number of independent memory strength (MSTR) per warp (thread)
ILP	Effective ILP number per warp (thread)
SSE_ILP	Effective ILP number for SSE instructions per warp (thread)
DEP	Dependence strength between consecutive instructions per warp (thread)
Bytes_per_thread	An average number of bytes each thread requests per thread
DistanceMemBytes	An average distance in bytes between each thread (affects different number of transactions)
N	Number of warps or threads per core or streaming processor (SM)

5.8 Applicability to CPU Architectures

The model is generic enough to be applicable for a CPU architecture. The primary issues to consider for CPU are modeling the ILP effect, and to correctly use the number of FP instructions as CPU has both scalar FP and vectorized FP instructions. For example, a vectorized FP instruction utilizes four SIMD lanes, whereas a non-vectorized FP instruction only uses a scalar FP execution unit. In terms of throughput, this is four factor of difference, hence it is very important to consider these factors correctly. Furthermore, unlike GPU execution, OpenCL implementation for CPUs does not support efficient multithreading between computation instructions. Hence, depending on the back-to-back instruction dependency, the performance degradation is affected significantly. For example, the FP instruction latency is about four cycles [30], so if there is no independent FP instruction within those four cycles, performance is degraded. For example, if the instruction is back-to-back dependent, performance degradation is by a factor of four from the ideal case.[7] If instructions are dependent every two instructions, then the degradation factor is by a factor of two. And, when the distance is greater than four, no performance degradation exists, as the pipeline is hidden by other same-type instructions. Finally, we have not observed a context-switching of threads when there is a memory access like the GPU does. For this reason, we model this behavior simply by setting the number of warps (threads) per core (i.e., N term) to the value of one, and the model automatically takes care of this phenomenon.

Figure 141 shows that the performance from MB_A to MB_D, and from MB_E to MB_H is linearly increased. The reason is that the back-to-back instruction dependency distance is one for MB_A, and it is four for MB_D. MB_D and MB_H produce the expected performance, because the FP latency for the CPU architecture is about four cycles. Hence, all the subsequent instructions are pipelined without delay, and thus the expected performance is obtained.

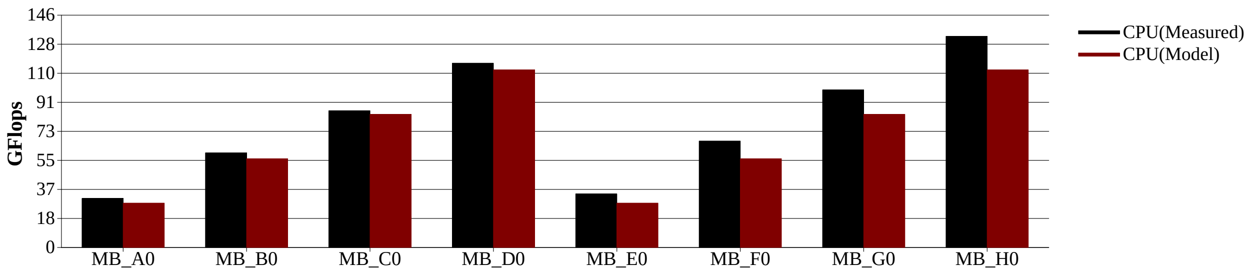


Figure 141: Baseline performance comparison by disabling the memory effect. Note that the dependency effect (DEP) is visible from A to D and E to H. Because there is no fused-multiply-add instruction, the performance does not double for E to H compared to A to D.

Figure 142 shows that because we have not observed an efficient multithreading effect during memory accesses, the MWP value is fixed at one for CPUs in the model.

Figure 143 shows the ratio for computations only and then the ratio considering memory accesses. Similar to GPUs, the effect of memory access is significant. In other words, when the CPU does not have a cache hit, an efficient multithreading

[7]Ideal case is when an output is produced every cycle.

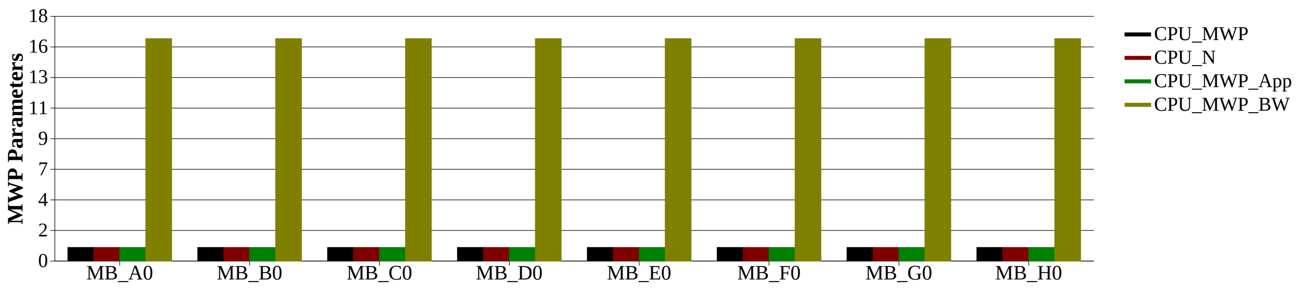


Figure 142: MWP determining factors for CPU.

effect by other threads does not occur.

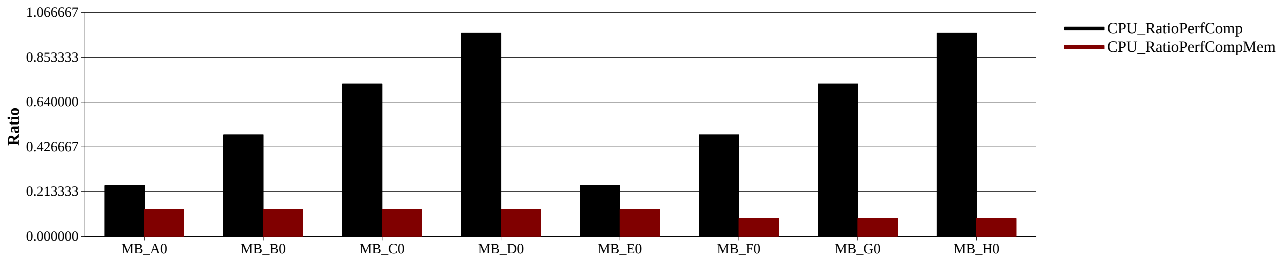


Figure 143: Ratio values for CPU, demonstrating the effect of the memory access on the overall performance.

5.9 Heterogeneous System and Execution

The ability of the new throughput model to generate multiple performance values has many benefits, including its applicability to a heterogeneous systems. By predicting the upperbound and lowerbound of the potential performance on both CPUs and GPUs, even before profiling on actual hardware, a better starting point of heterogeneous execution can be provided to the runtime scheduler. For example, rather than starting 50 to 50 percent between the CPU and GPU, the model could suggest only running on one architecture (not running at all), or the model can provide a better starting point for execution.

Figures 144 and 145 show the performance values predicted by the model for CPU and GPU, respectively. The model predicts different cases such as the high-L1 hit and the DRAM-intensive. Because the performance gap between L1 and DRAM is different (as shown for matrix multiplication case), this information is also important for runtime scheduling.

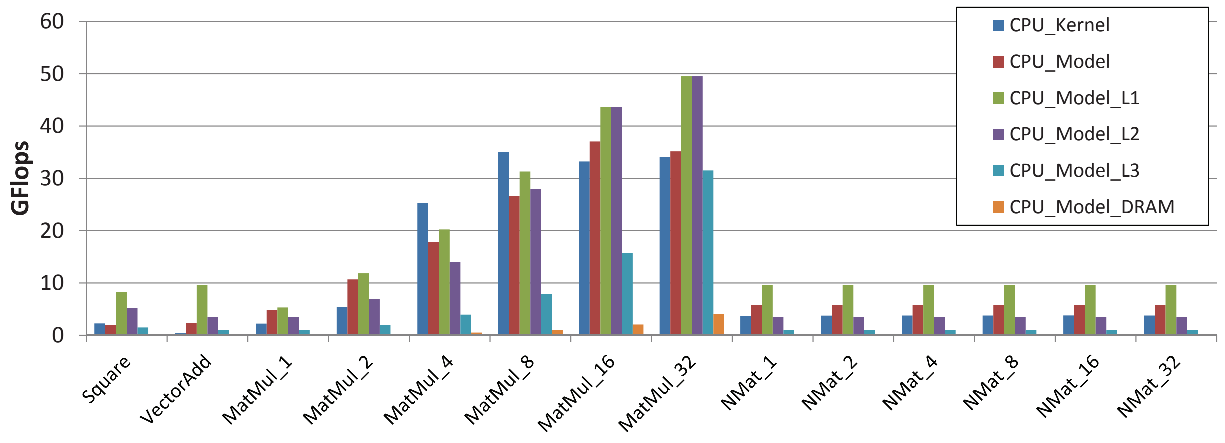


Figure 144: Multiple model performance predictions for CPU.

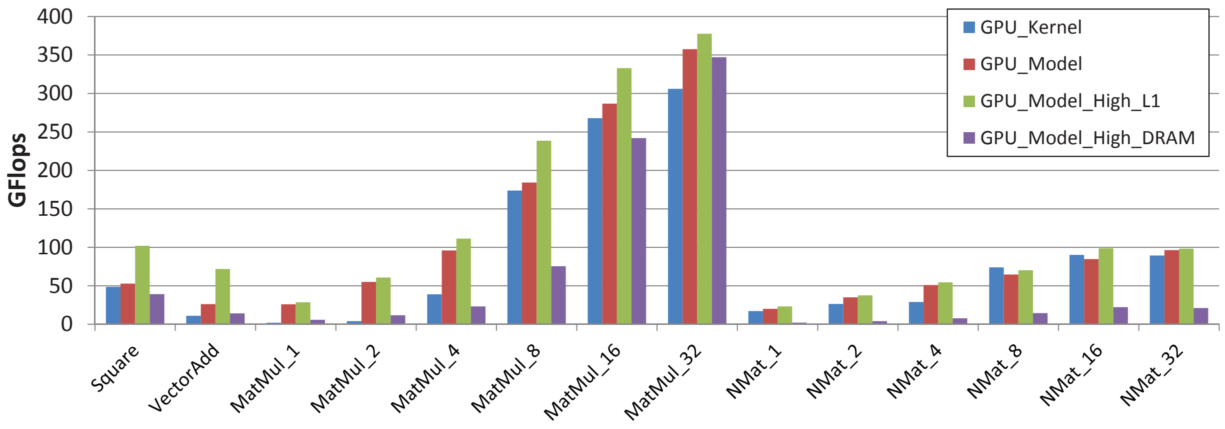


Figure 145: Multiple model performance predictions for GPU.

Figure 146 shows the final predictions with the measured data for a CPU and GPU. Despite having cache hierarchy for both the CPU and GPU, this demonstrates that the model successfully predicts the kernel performance. The model uses a heuristic to project cache hit/miss information by using a small input and using a GPU profiler for both the CPU and GPU. This mechanism can be further improved by using Ocelot, which can dynamically analyze a subset of memory addresses and predict cache hit/miss information, even before the *actual* scheduling.

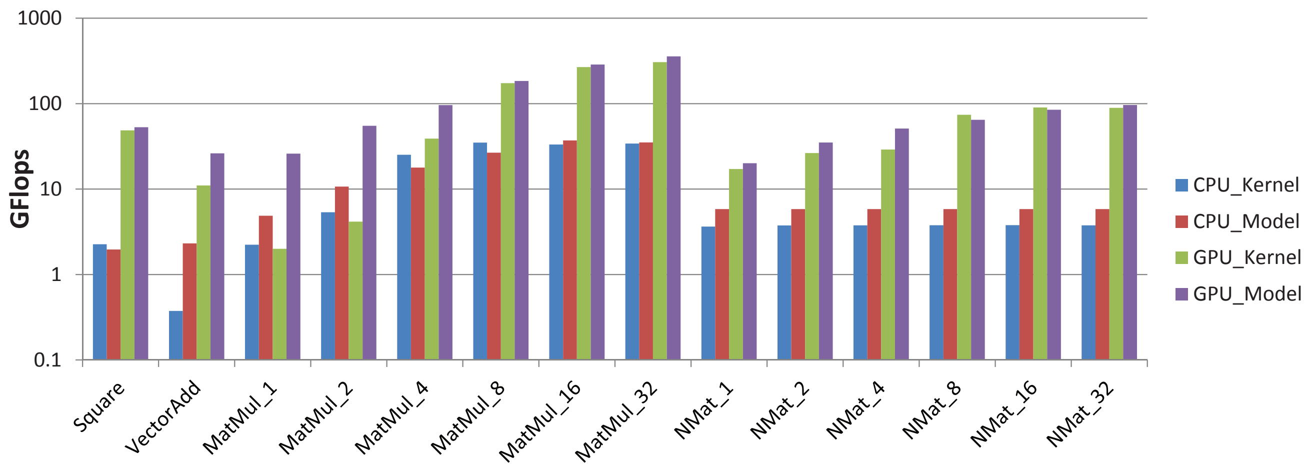


Figure 146: Comparison between the model and measured performances for CPU and GPU.

Figure 147 shows the scheduling decisions projected by the model and the manually-found best mapping ratios. The scheduling ratio from the model is within 17% of the manually found method. Finally, Figure 148 shows the final result graph for heterogeneous execution.

This section demonstrated the potential applicability to both CPU and GPU architectures and heterogeneous system. Surprisingly, the model successfully predicted multiple performance values for different cache hierarchy levels, and despite using a simple heuristic for finding cache and DRAM memory access ratios, the final predictions matched the final measured data within 17% for the GPU and 28% for the CPU. These errors can be further reduced if a better memory distribution ratio is found by either using a small input on a profiler or advancing the heuristic and using Ocelot at the same time. Furthermore, by considering the data transfer time to GPU, the scheduling decision will be more practical. This concludes the section on the advanced model and its multiple practical uses, which are not limited to scheduling.

Manual Ratio	
CPU	GPU
4.44%	95.56%
3.28%	96.72%
52.72%	47.28%
56.30%	43.70%
39.29%	60.71%
16.76%	83.24%
11.04%	88.96%
10.03%	89.97%
17.50%	82.50%
12.43%	87.57%
11.47%	88.53%
4.85%	95.15%
4.02%	95.98%
4.04%	95.96%

Model Ratio	
CPU	GPU
3.58%	96.42%
8.11%	91.89%
15.78%	84.22%
16.24%	83.76%
15.66%	84.34%
12.64%	87.36%
11.44%	88.56%
8.95%	91.05%
22.51%	77.49%
14.24%	85.76%
10.24%	89.76%
8.26%	91.74%
6.43%	93.57%
5.70%	94.30%

Figure 147: Scheduling decision predicted by the throughput model.

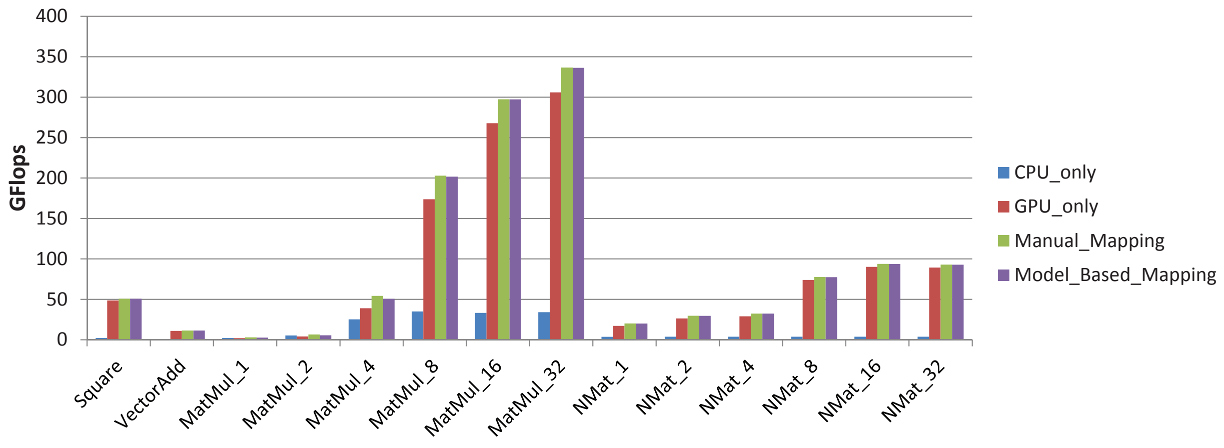


Figure 148: Heterogeneous performance using the model output.

CHAPTER VI

RELATED WORK

6.1 Recent GPU Performance Work

Recent relevant work on the GPU performance model is GROPHECY by Meng et al. [68]. Their work uses our analytical performance work [33] for performance prediction, but extends the performance work by projecting performance on CPU code. The idea is that since converting the CPU code to GPU code takes time and effort the code conversion should be done only when the model predicts that sufficient benefits exist. Hence, this work primarily focuses on the issues of transforming CPU code. Contrary to this work, our work extends the analysis further to bandwidth saturation and using OpenCL. Furthermore, the model is revised extensively so that performance-affecting hardware and software parameters can be easily simulated, producing an upperbound and lowerbound of expected performances.

Sim et al. [89] has done a practical extension of our work [33]. Rather than predicting the performance itself, the work focuses on quantifying the benefit of an optimization as many optimization techniques are available. Furthermore, the work visualizes such optimization action in compute-and-memory-bound space. The primary difference from our improved work in Chapter V is that the focus is different. Because our improved model is restructured and mathematically built using a top-down approach, given an application, our model can immediately calculate the upper-bound and lower-bound of performance while still preserving ideas such as the MWP and CWP concepts. Furthermore, we improved the MWP-CWP relationship to remove corner cases and further extend the work to target energy-efficient execution by finding optimum N and number of cores, and producing detailed category information with optimization suggestions.

The work by Jia et al. proposed GPURoofline [46], which is an extension of the previous work [103]. While conceptually the topic is practical and useful, using just the compute intensity to find the upperbound is too simplified for a GPU architecture. Unlike CPU execution, there is an inter-complicated relationship between multiple factors such as N, memory-level parallelism, bandwidth saturation, etc. Hence, given instruction information and N, it is not very clear how to use the model, whereas our work clearly shows how to use the model and produces more performance values, not limited to the values of the roofline.

Zhang and Owens proposed a quantitative performance analysis model [105]. The major difference is that while our work is analytically built-up using architectural information, their work is built from the opposite direction, which is using microbenchmark outputs. Their work focuses on instruction pipeline throughput and shared memory. The work is more on analyzing CUDA performance on a few real benchmarks with qualitative discussions rather than using the model inputs and outputs. Contrary to this work, our work uses explicit model outputs to discuss the output results.

Baghsorkhi et al. [7] proposed a GPU performance model using a work-flow graph as an abstract interpretation of a GPU kernel. The program dependence graph (PDG), which contains control and data dependence information, is used to predict performance. This work primarily focuses on a compiler-based approach that determines the path and weight of each path to find the dynamic instruction counts. While this work would provide very good analysis on obtaining more accurate inputs, the output is mere execution cycles with limited further insights, unlike our work. Nevertheless, there is a high potential to improve the model significantly by leveraging this work for the input portion of our model.

Kothapalli et al. [54] proposed a performance prediction model similar to our work.

However, the model is a simplified version, which is limited in handling bandwidth and multiple N that affects memory performance.

The community for general-purpose GPU (GPGPU) computing provides insights into how to optimize GPU code to increase memory-level parallelism and thread-level parallelism [29]. 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. Ryoo et al. [83] proposed two metrics to reduce an optimization space for programmers by calculating the utilization and efficiency of applications, but their work only focused on non-memory-intensive workloads. In comparison, we thoroughly analyzed both memory-intensive and computation-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 execution time.

Predicting multiple GPU performances using a single performance model is proposed by Schaa et al. [86]. Recent work by Zhang and Owens [105] proposed a performance model from a quantitative perspective, where the model is based on the throughput of the instruction pipeline, shared-memory access, and global-memory access.

Luk et al. [66] empirically modeled the performance of GPGPU applications as a linear model using run-time information for a dynamic-compilation system. Williams et al. proposed the model, called Roofline, to visualize the performance of multi-core architectures [103]. The model sets an upper bound on the performance of a kernel that depends on memory-intensity and computation-intensity metrics.

Several application programmers have developed a performance model for specific applications. Choi et al. [15] proposed a GPU-kernel performance model of a sparse matrix vector multiply (SpMV) kernel for auto tuning. The proposed model guides the auto-tuning process, which is input-matrix dependent. Meng et al. [69] presented a model for optimizing iterative-stencil loops used for image processing, data mining, and physical simulations. Govindaraju et al. [28] presented a memory model to improve the performance of applications by improving the texture-cache usage. The work by Liu et al. [65] modeled the performance of bio-sequence applications written in OpenGL shading language (GLSL) [50].

6.2 Performance Models for CPU Architecture

Karkhanis and Smith [48] 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. The model is analytical based and is one of the first works in the CPU domain. Currently, the most relevant works using analytical models in CPU domains are from the group led by Eeckhout in Belgium. For example, Eyerman and Eeckhout [21] extended the work by Karkhanis thoroughly using the mechanistic performance model using simulation. Then another work by Eyerman used the identical research on real hardware [22].

Hence, later Heirman et al. proposed using cycle stacks [32] to understand performance bottlenecks for multi-core environments as there could be other factors that were not seen in a single core environment. The insight is that overall CPI is made up of different CPI stacks, which are contributed from different execution units. However, the benchmark is still single threaded. Hence, to further improve this work using multithreaded workloads, Eyerman et al. [20] extended the work. The primary difference from our work is that their work scope focuses on CPU architectures. For GPUs, there is one more level of parameters that needs to be considered. For example, it is not only the ILP, but as the GPU very efficiently switches between different warps to hide long-latency instructions or idle DRAM cycles, TLP should be considered as well. As we have seen, considering this part is non-trivial. The relationship between ILP, TLP, etc. with actual performance is very complicated, considering N, finite bandwidth, and memory types.

More analytical models have been proposed for superscalar processors [72, 73, 75]. However, most work did not consider memory-level parallelism or even cache misses. Chen and Aamodit [14] improved the first-order superscalar processor model by considering the cost of pending hits, data prefetching, and miss status holding register (MSHR). They showed that not modeling prefetching and MSHR can increase errors significantly in the first-order processor model. However, only the cycle per instruction (CPI) of memory instructions was compared with the cycle-accurate simulator.

A rich body of work exists that predicts parallel-program performance prediction using stochastic modeling or task-graph analysis. Saavedra-Barrera and Culler [84] proposed a simple analytical model for multi-threaded 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 multi-threaded architectures. However, they have not considered synchronization effects. Furthermore, the application characteristics are represented with statistical modeling, which cannot provide a detailed performance estimation for an application.

Sorin et al. [91] developed an analytical model to calculate the throughput of processors in the shared memory system. They developed a model to estimate the stall times of a processor due to cache misses or resource constraints. They also discussed coalesced-memory effects inside the MSHR. The majority of the model is also based on statistical modeling.

6.3 Power Models for CPU Architecture

Isci and Martonosi proposed a power model using an empirical method [44]. There have been follow-up studies that use similar techniques for other architectures [16]. Wattch [10] has been widely used to model dynamic-power consumption using event counters from architectural simulations. HotLeakage [104] models a leakage current based on a circuit model and dynamic events. Skadron et al. proposed the temperature-aware micro-architecture model [90] and released a software called HotSpot. The software requires architectural simulators to model the dynamic power consumption. However, all these studies were done only for CPUs. Sheaffer et al. studied a thermal management for GPUs [88]. In their work, the GPU was a fixed graphics hardware. Fu et al. presented the experimental data of a GPU system and evaluated the efficiency of energy and power [24].

Huang et al. evaluated the energy efficiency of GPUs for scientific computing [35]. Their work demonstrated the efficiency for only one benchmark, and concluded that using all the cores provides the best efficiency. They did not consider any bandwidth-limitation effects. Li and Martinez studied power and performance considerations for a chip multi-processor (CMP) [61]. They also analytically evaluated the optimal number of processors for the best energy savings. However, their work focused on CMP and presented heuristics to reduce a design-space search using power and performance models. Suleman et al. proposed a feedback-driven threading mechanism [93]. By monitoring the bandwidth consumption using a hardware performance counter, the feedback system decides how many threads can be run without degrading performance. Unlike our work, it requires run-time profiling to know the minimum number of threads that reaches the peak bandwidth. Furthermore, they demonstrate power savings through simulation only.

Recently, large-scale architectural design space was simulated [6, 43, 56] by statistical sampling and regression techniques. Rai et al. proposed the temperature prediction mechanism that is simpler and even faster than Hotspot [81]. The authors claim that not all the details such as processor floor plan and thermal structure are necessary. Lewis et al. proposed a runtime model for server energy consumption [60].

Esmaeilzadeh et al. analyzed and quantified the power and performance perspectives for commercial processors for the past 10 years. This work provides a good overview of the power and performance perspectives and predicts the future trend [19]. Taylor discusses the trend of dark silicon and how specialized architectures can be leveraged to obtain high energy efficiency [96]. The author also published the work [18] that combines technology scaling models, performance models, and empirical results to answer how much more performance can be extracted from the multicore path in the future.

Kong et al. present recent thermal management techniques for microprocessors [53]. The authors categorize the techniques six main parts: temperature monitoring, floor planning, OS/compiler techniques, cooling solutions, etc.

6.3.1 OpenCL

OpenCL [79] is an open standard for parallel programming of heterogeneous systems. The kernel code design is flexible and generic to be applicable for many different types of architectures and devices. However, there are no official schedulers for heterogeneous computing and no official release for device-specific runtime optimizations. Hence, this has been a hot research domain previously as well as for the future.

For the framework, Kim et al. proposed the OpenCL framework for CPU and GPU clusters in [52], and another work by Kim achieves a single compute device image in OpenCL for multiple GPUs [51]. Since CPU and GPU memory spaces are disjoint [85], Jablin et al. proposed CPU-GPU communication management and optimization [45]. OpenCL is also popular in industry as well, as Intel and NVidia released their own runtimes [41, 78].

OpenCL naturally maps to a GPU architecture. However, for CPUs, the task is not very straightforward. A CPU needs to utilize vectorized instructions (SSE, AVX) [67]. However, because CPU hardware does not have a high number of execution units (i.e., thread-level parallelism is limited in CPUs), optimizing OpenCL performance is difficult. For this reason, Intel provides a list of guidelines for optimizing OpenCL on CPUs [42].

6.3.2 Energy-Efficient Execution

CPU and GPU co-executes from performance, energy, and temperature perspectives [47]. Benchmark suite is developed to target heterogeneous computing [13]. A similar approach is taken in modeling for performance and energy efficiency for the FT benchmark, but this is specifically for one benchmark [26]. As a general article regarding energy-efficient computing, Brown and Reams discuss this topic, including GPUs [11]. Because power and temperature are now one design constraint, another article that discusses design parameters for predicting the future of big chips is discussed in [36]. Lee et al. claimed that throughput computing is an important topic for the future and did an architectural-trade-off study between CPUs and GPUs [59], where the conclusion is that the CPU is not an order-of-magnitude behind GPU performances.

Hamano et al. proposed the metrics that calculate EDP, and a power-aware scheduling algorithm [31]. Similarly, SPRAT [94] compilation framework is proposed that translates code to run efficiently on CPU and GPU. Another work that schedules task in heterogeneous scheduling is done [58, 100]. They not only schedule the workloads, but further they apply DVFCS technique (dynamic-voltage-frequency-core scaling) to reduce power consumption, and the method is based on the machine-learning mechanism. Liu et al. proposed power-efficient, time-sensitive mapping technique [64] for heterogeneous systems consisting of CPUs and GPUs. The mechanism is based on mathematical modeling and power-related concepts along with DVFS are explained.

Wang and Ren also have done a similar work that efficiently schedules tasks in heterogeneous system [99].

Benchmark suites are also proposed. Che et al. proposed the Rodinia benchmark suite for heterogeneous computing [13]. Each benchmark has CPU and GPU versions available for analysis. Later, Seo et al. proposed the NAS parallel benchmark suite in OpenCL in the same conference (IIWSC) [87]. Another GPGPU benchmark suite, Parboil, is provided [97].

CHAPTER VII

CONCLUSION AND FUTURE RESEARCH

First, the analytical model-based approach provides valuable insights into processors [48]. Second, the model enables a fast trade-off analysis between architectural parameters such as number of cores, number of hardware threads per core, bandwidth, etc. Rather than using a simulator, the model-based approach has the advantage that this trade-off analysis can be done fast and also provides insights to a computer architect. Furthermore, this information is vital and can be used just-in-time for a programmer, compiler, or even a runtime system. All of these insights are directly applicable to this thesis work.

As for the accomplishments on the path, the following provides a summary.

- **Chapter II** proposed and evaluated a memory parallelism-aware analytical model to estimate the execution cycles for the GPU architecture. The key idea is to find the maximum number of memory warps that can execute in parallel, a metric called MWP, to estimate the effective memory instruction cost. The evaluation shows that the geometric mean of absolute error of the analytical model on micro-benchmarks is 5.4% and on GPU computing applications is 13.3%. The work provides model-based insights and has established a starting step for heavily-multithreaded architectures including GPUs.

- **Chapter III** proposed an integrated power and performance (IPP) modeling system for the GPU architecture and the GPGPU benchmarks. This work extends the previous empirical CPU power work [44] for modeling GPU power. Furthermore, we considered the increased leakage power effect from the temperature increase. IPP predicts the power consumption and the execution time with an average of 8.94% error for the evaluated GPGPU benchmarks. IPP predicts the performance per watt and the optimal number of cores for the five bandwidth-limited GPGPU benchmarks. Based on IPP, the system can save on average 10.99% of run-time energy consumption for the bandwidth-limited applications by using fewer cores. This has been demonstrated in a real machine; the model also projected the energy savings if power gating is employed.

- **Chapter IV** proposes a new cost-effective temperature-measurement system that uses thermocouples for the first time for GPU architectures. We devised a method to install thermocouples *between* a chip and a heatsink. With this system, we successfully measured the on-chip temperature distribution of a GPU processor. The conclusion is that even if the same number of cores that are activated, depending on what cores are activated, the peak achieved power can be reduced. Unlike previous works that used simulators, we used a real measuring system using thermocouples and thermospacers. When this location information is used in conjunction with the thesis, the peak power can be further reduced.

- **Chapter V** significantly improves the analytical framework for performance. The main concept of memory-level parallelism is still preserved in this work. Rather than just calculating final execution cycles, the model is more sophisticated on how much *this* application requires bandwidth by proposing a term called MWP_App given a finite N. We also propose MWP_App_Infi metric, which provides how many N are necessary to completely hide the idle DRAM waiting cycles.

  The model is also built in a different way, while preserving the previous concepts as much as possible. The biggest modification is that the model now provides different levels of performance predictions: (1) peak achievable performance assuming cache-hit, (2) lower-bound assuming DRAM-only memory accesses, and (3) performances that fall in the middle. In other words, the model now produces a throughput prediction. This requires only the number of instructions per basic block for producing performance and power information.

In high level, the thesis work aims to achieve energy-efficient execution on many-core architectures, not limited to GPUs. Achieving energy-efficient execution, which is unique to an application, is not only the final outcome. The paths that lead to the result produces many other insights that visualize bottlenecks for performance and peak power. For example, depending on the application, the profiler result may return good cache hit information; however the performance might not be good enough for a programmer such as naive matrix multiplication. The new model not only provides an upperbound (perfect L1-cache hit) and lowerbound (only DRAM accesses), but it also provides additional information that the issue with the application lies in the effective mixture ratio (i.e., useful instructions to the total instructions). Furthermore, this analytical model can provide further interesting insights and answer questions of a programmer such as (1) what will happen with just one more core?, (2) is buying a GPU with more BW or more cores beneficial for *this* application?

This work has much potential to be applicable to different domains. First, knowing the upperbound and lowerbound information is crucial for a programmer since a decision such as whether or not it is beneficial to keep optimizing can be determined earlier. Furthermore, the model facilitates a trade-off analysis among changing architectural parameters such as bandwidth, number of cores, number of hardware threads per core, and etc. In addition, this work is extended to using OpenCL language and also demonstrates its effective usefulness to CPU architecture as well.

7.1 Future Research Directions

The thesis concludes with potential future research discussions.

7.1.1 IR-based Instruction Analyzer

A natural next step to this research is writing an IR-based instruction analyzer. This would enable applying the model to a greater variety of benchmarks as well as automating the process of passing the instruction information to the analytical model. First, how to count instructions given many basic blocks needs to be decided. The immediate step is to count different types of instructions per basic block granularity. Then, different heuristics can be proposed for how to determine the execution path.

Another possible implementation is to make this analyzer a feedback driven mechanism. By using Ocelot implementation and a small input set, the execution path can be approximated before actually running on real hardware. This would raise the accuracy of the instruction count mechanism. Furthermore, this whole process of obtaining model input and model implementation can be integrated with Ocelot along with the power model.

7.1.2 Providing Feedback

Programmer: A programmer can leverage the model insights to check the expected performance, a peak achievable performance, and lower bound of performance along with expected power numbers. The effect of changing architectural parameters such as bandwidth, number of cores, number of hardware threads, etc can be efficiently simulated uniquely to an application. Most of all, by knowing the peak achievable performance, either he can keep optimizing a program or change the hardware or an algorithm itself.

Compiler: A compiler can leverage the model insights significantly, controlling the register usage that directly affects N and/or the number of cores. Figure 149 shows further insight and opportunity for compiler optimization.

If the model predicts that there is enough bandwidth available on a given hardware, then the compiler could try to generate more memory instructions, if that

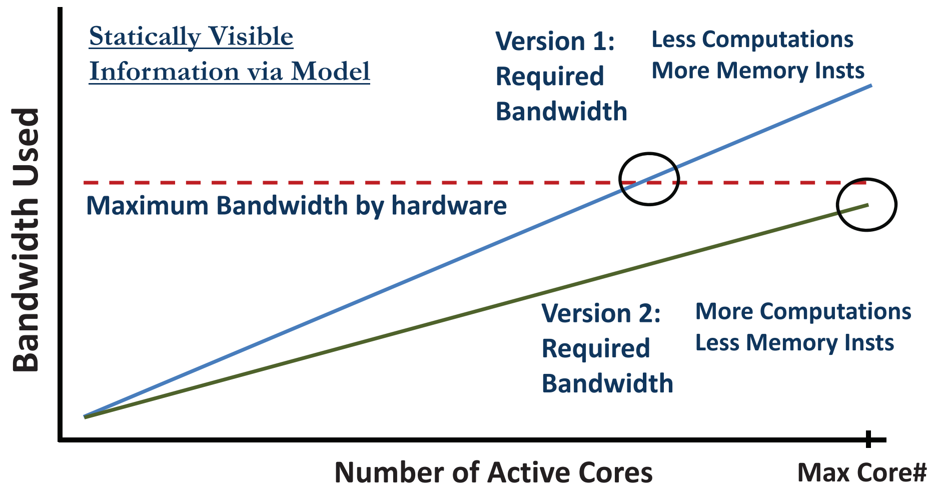


Figure 149: Using the analytical model at static time enables visualization of possible, further optimizations that still do not degrade performance.

results in fewer overall instructions. On the other hand, if the model expects memory saturation, the compiler could instead attempt to (1) reduce the number of memory instructions and/or (2) try to increase the amount of computations between memory instructions (i.e., reduce MWP_App), and/or (3) either reduce the number of active cores by the compiler itself or pass this information to the runtime system. The applicability of the model is that it has benefits even at static compilation time.

Another potential optimization is that if the compiler could change the reference pattern to reduce the cache miss ratio, it effectively reduces effective memory latency, which then reduces MWP_App. As this has less pressure on the memory bandwidth, the compiler could focus on other optimizations.

Another benefit of leveraging the model at static time is that if multiple types of devices are available, the compiler can also make a smart decision about where to offload a work. This is not only the runtime's task. To make this more accurate, the data transfer cost for GPUs has to be taken into account.

Runtime: The immediate use for the runtime is to determine where to offload a work, for example, between a CPU and a GPU. This decision can be determined by finding the expected performances on a CPU and a GPU, where on a GPU, the data transfer time has to be taken into account.

A more interesting challenge lies in the heterogeneous execution. This topic is a big topic that is extendable by using this thesis work. Many previous works have had used the profiling method, such as the previous work I have co-authored in [66]. However, most previous works have used either runtime profiling or very simplified mathematical fitting. Contrary to those works, if my work is extended, not only is simple curve fitting possible, but by leveraging the insight of an application, energy-efficient execution is possible. However, this can even give just-in-time opportunities for the compiler and even provide feedback to a programmer or an algorithm designer.

Heterogeneous Execution: Finally, an additional extension of the model to heterogeneous execution is to determine if an overall target is speed-up or an energy efficiency. These two terms are highly correlated but with some exceptions. This information is very useful and practical to the runtime system, especially the system that uses OpenCL since the model is applicable.

When an application runs very efficiency on a CPU, but not so much on a GPU, then it is better not to schedule *any* work to the GPU in the first place. Because of the GPU data transfer time as well as power increase, the scheduling is not worth it for both performance and power. On the other hand, if an application runs very non-efficiently on a CPU, but the GPU is very efficient, then it is necessary to schedule as much work as possible to the GPU *regardless* of whether the CPU or GPU is power efficient. The model suggests that power does have an impact; however, performance is the main contributing factor.

The future potential research is to further develop this heterogeneous energy-efficiency and speed-up analytical model along with the new throughput model. As this whole infrastructure is integrated, this thesis work will be a major contribution to assisting energy-efficient execution on heterogeneous platforms.

APPENDIX A

ENERGY EFFICIENCY

This chapter introduces a high-level work using mathematical modeling, for the purpose of deriving the relationship between speed-up and and energy-efficiency. It has long been said that maximizing speed-up results in the best energy efficiency. Hence, the purpose of this chapter, as a preliminary study, is to investigate further this phenomenon deriving the relationship mathematically for understanding and gaining insights. I project that when this work is further integrated with the performance throughput model in Chapter V, the work will give further insights into energy-efficient execution using the analytical approach.

A.0.3 High-level Modeling

First, we analytically show the relationship between performance and energy-efficiency using mathematical derivations. Then, by leveraging the derived model, we explain the previous results, which mostly showed a linear relationship. Finally, we explore the trade-off between heterogeneous execution versus homogeneous execution (i.e., trade-off between reduced execution time versus higher power).

A.0.4 Introduction of the Parameters

Definitions

$$T_o : \text{Baseline Execution time}$$

$$P_o : \text{Baseline Average Power}$$

$$E_o : \text{Baseline Energy Consumption} \qquad (E_o = P_o \times T_o) \tag{94}$$

$$\mathsf{S} = Speedup = \frac{T_o}{T_N} \quad \text{where } T_o \text{ is the baseline, } T_N \text{ is the new time} \tag{95}$$

$$\mathsf{E} = EnergyEfficiency = \frac{E_o}{E_N} \quad \text{where } E_o \text{ is the baseline, } E_N \text{ is the new value} \tag{96}$$

A.0.5 The Relationship Between Energy-Efficiency and Performance

We define performance as the speed-up over the baseline, as shown in Equation (95). And energy efficiency, shown in Equation (96), represents how much energy is saved with the new execution over the baseline. Then, how is energy-efficiency related to performance?

First, define the new power and execution time as follows, where new execution time and power are multiplied by α terms.

$$T_N = \alpha_T \times T_o \tag{97}$$

$$P_N = \alpha_P \times P_o \tag{98}$$

How about energy efficiency metrics?

$$E_o = P_o \times T_o \tag{99}$$

$$E_N = P_N \times T_N \tag{100}$$

This can be further decomposed into the following.

$$\begin{aligned} E_N &= P_N \times T_N \qquad (101)\\ &= \alpha_P \times P_o \times \alpha_T \times T_o \\ &= \underbrace{\alpha_P \times \alpha_T}_{\alpha_{ov}} \times \underbrace{P_o \times T_o}_{E_o} \\ E_N &= \alpha_{ov} \times E_o \end{aligned}$$

Then, energy-efficiency is related to performance

$$\mathsf{E} = \frac{E_o}{E_N} = \frac{E_o}{\alpha_{ov} \times E_o} = \frac{1}{\alpha_{ov}} = \frac{1}{\alpha_P \times \alpha_T} = \frac{P_o}{P_N} \times \frac{T_o}{T_N} \tag{102}$$

Since T_o/T_N is equal to speed-up S, Equation (102) further simplifies to the following.

$$\mathsf{E} = \frac{P_o}{P_N} \times \mathsf{S} \tag{103}$$

Therefore, energy efficiency E is related to the change in power and speed-up of an application, as represented by Equation (103). This equation shows that if an increase in power is not significantly different from the baseline power (i.e., P_o is similar to P_N), then E is a linear relationship to S.

A.1 Projections

In this section, we project the different scenarios of executions and what it means in terms of energy and performance. Figure 150 shows different cases of energy-efficiencies and speed-ups. The x-axis and y-axis are plotted in log-scale in powers of two to match the experimental results obtained earlier. The assumption of the figure is that a power increase as speed-up increases is a linear relationship.

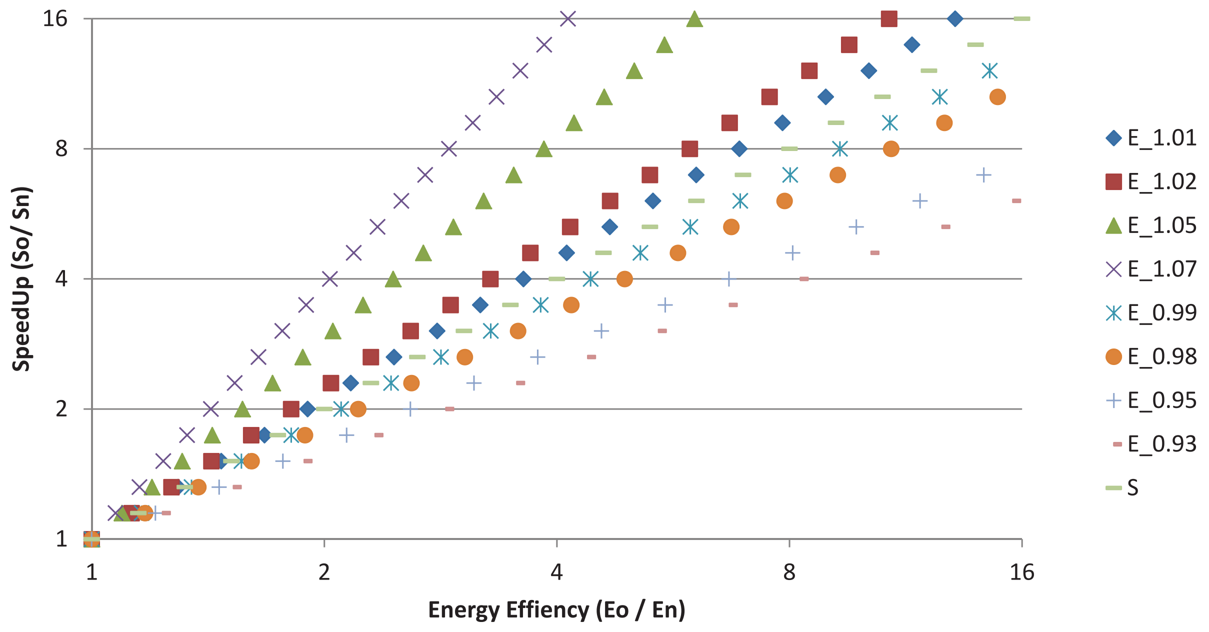


Figure 150: Model projections on energy-efficiency vs. speed-up on an application.

A.2 Effects of multiple cores and heterogeneous executions

Essentially, the noticeable effects of multiple core executions are reflected as changes in S and P_N terms. For example, if performance increases linearly with an increasing number of cores, then the S term will linearly increase as well. We also expect the power term, P_N, to increase in a similar fashion if we assume higher performance comes with more cores. The challenge here is that if we know how the S and P terms change with respect to multiple core or heterogeneous executions, then finding energy efficiency, E, is straightforward.

A.2.1 Performance

To model S from the high level, the following architectural parameters are used: F: frequency, C: number of cores, ζ: throughput per core, D data. T, execution time, is proportional to the terms as follows.

$$T \propto \frac{D \times \phi}{C \times \zeta \times F} \qquad C \text{ cores, } \zeta \text{ throughput per core, } F \text{ frequency, } \phi \text{ benchmark} \tag{104}$$

Since we focus on a relative speed-up (not an absolute performance itself), we can approximate the speed-up as follows.

$$\mathsf{S} = \frac{T_o}{T_N} = \frac{D_o \phi_o}{C_o \zeta_o F_o} \times \frac{C_N \zeta_N F_N}{D_N \phi_N} = \frac{C_N \zeta_N F_N}{C_o \zeta_o F_o} \times \frac{D_o \phi_o}{D_N \phi_N} \tag{105}$$

D represents an amount of data to be processed. ϕ models the performance-degrading effect that a benchmark has on the architecture such as memory saturation. If the same machine configuration is used, then Equation (105) can be further decomposed into the following since the other terms cancel out.

$$\mathsf{S} = \frac{C_N}{C_o} \times \frac{D_o}{D_N} \tag{106}$$

A.2.2 Power

To model with an emphasis on determining the *change* in power, we primarily consider the factors that have direct impact. These include C : #active cores, F : frequency, λ : benchmark characteristics on the architecture, B : baseline idle power. We assume that static power is mostly reflected in the B term, and the αV^2 term in the traditional dynamic power equation is not changed. Instead, we only consider F and the number of cores C that can be controlled in the experiment.

$$P \propto C F^3 \lambda + B \tag{107}$$

$$\frac{P_o}{P_N} = \frac{C_o {F_o}^3 \lambda_o + B_o}{C_n {F_n}^3 \lambda_n + B_n} \tag{108}$$

The baseline idle power, B, might not seem relevant since we are modeling the change in power. However, this is not true. For example, even if there is an increase in the $CF^3\lambda$ term, if the baseline B is very large, then the overall change (P_o/P_N) will be very small. For a heterogeneous system that has a CPU and a GPU, B_o and B_n are the same, and only one term needs to be added instead of two , as the idle system contains both CPU and GPU idle powers.

A.2.3 Usage and Insights of the Model

The overall equation for energy efficiency can be expressed as follows.

$$\mathsf{E} = \frac{P_o}{P_N} \times \mathsf{S} = \frac{P_o}{P_N} \times \frac{T_o}{T_N} = \left(\frac{C_o{F_o}^3\lambda_o + B_o}{C_n{F_n}^3\lambda_n + B_n}\right)\left(\frac{C_N\zeta_N F_N D_o\phi_o}{C_o\zeta_o F_o D_N\phi_N}\right) \tag{109}$$

We can see that the trend in Figure 151 tends to be linear regardless of whether we change the frequency and power (different configurations 1,2,3,4,5) in the efficiency-speedup plotspace. But, depending on the rate of power change, we can see sub-linear and super-linear lines.

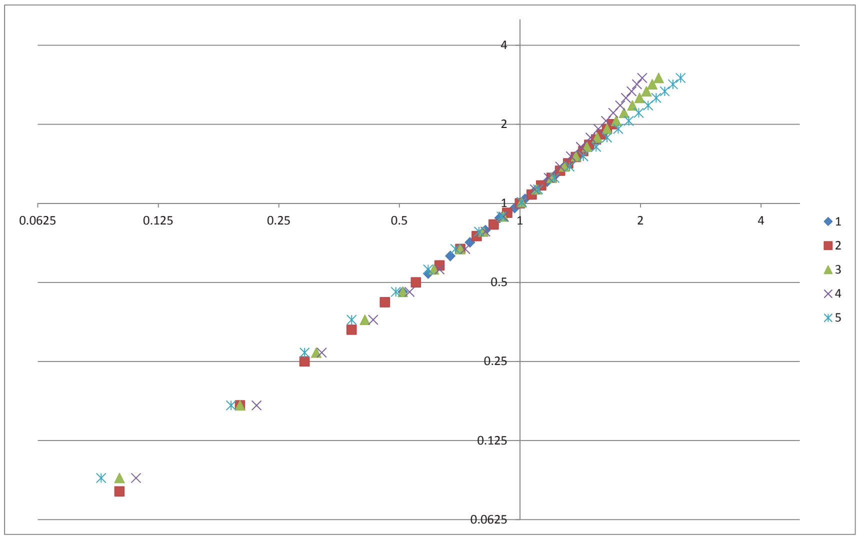


Figure 151: Different configurations of just CPU tends to be linear in energy-efficiency vs. speed-up space, as the increase tends to be canceled out by the increase in the denominator unless the rate is different.

A.3 Effects of multiple cores and heterogeneous executions

To model heterogeneous execution, only two terms need to be modified: P and T. The power is increased as two architectures are activated, but the execution time is reduced.

$$T = Max(T_{CPU}, T_{GPU}) \tag{110}$$

$$P = P_{CPU} + P_{GPU} \tag{111}$$

Equation (109) can be further modified as follows. Note that the base P_o and P_N parameters are still the CPU-only configuration to match the previously-obtained measured graphs. Hence, we apply the heterogeneity to T_N and P_N configurations.

$$\mathsf{E} = \frac{P_o}{P_N} \times \frac{T_o}{T_N} = \frac{P_o}{P_{N\_CPU} + P_{N\_GPU}} \times \frac{T_o}{Max(T_{N\_CPU}, T_{N\_GPU})} \tag{112}$$

where the CPU and GPU terms are as follows ($_{\_C}$ means CPU and $_{\_G}$ means GPU)

$$T_N = Max\left(\frac{D_{N\_C}\phi_{N\_C}}{C_{N\_C}\zeta_{N\_C}F_{N\_C}}, \frac{D_{N\_G}\phi_{N\_G}}{C_{N\_G}\zeta_{N\_G}F_{N\_G}}\right) \tag{113}$$

$$\begin{aligned} P_N &= \left(C_{N\_C}{F_{N\_C}}^3\lambda_{N\_C} + B_{N\_C}\right) + \left(C_{N\_G}{F_{N\_G}}^3\lambda_{N\_G} + B_{N\_G}\right) \\ &= \left(C_{N\_C}{F_{N\_C}}^3\lambda_{N\_C}\right) + \left(C_{N\_G}{F_{N\_G}}^3\lambda_{N\_G}\right) + B_o \end{aligned} \tag{114}$$

The data distribution terms, D_{N_C} and D_{N_G}, simply mean the distributed data size to the CPU and GPU. And the B_o term comprehensively refers to the idle system power that has CPU and GPU cards installed.

A.4 Projections of multiple heterogeneous cores executions based on analytical approach

Since the main goal is to look at the high-level trend of energy-efficiency and speed-up changes with respect to the number of cores, frequency, and other factors, the model can not model each benchmark's effect in detail. Nevertheless, by categorizing performance and power into high and low values, we can see the effects they have on a heterogeneous execution.

To simulate the effects of heterogeneous execution, each series has 11 points that represent the partitioning ratio (i.e., 0 means 100% CPU, 100 means 100% GPU). Not only that, we model different performance and power characteristics for each series. In other words, the CPU could run efficiently for Series A but not GPUs, and vice versa. Performance efficiency is controlled by ϕ value and can be obtained either by machine learning or by the detailed performance model. Power efficiency is controlled by *coreweight* terms.

Table 16: Each series represents an unique benchmark type. The goal is to analytically find out the effect on the heterogeneous execution.

Terms	**Performance: CPU**	**Performance: GPU**	**Power: CPU**	**Power: GPU**
Series 1	Slow	Slow	Low	Low
Series 2	Slow	Slow	Low	High
Series 3	Slow	Slow	High	Low
Series 4	Slow	Slow	High	High
Series 5	Slow	Fast	Low	Low
Series 6	Slow	Fast	Low	High
Series 7	Slow	Fast	High	Low
Series 8	Slow	Fast	High	High
Series 9	Slow	Slow	Low	Low
Series 10	Slow	Slow	Low	High
Series 11	Slow	Slow	High	Low
Series 12	Slow	Slow	High	High
Series 13	Slow	Fast	Low	Low
Series 14	Slow	Fast	Low	High
Series 15	Slow	Fast	High	Low
Series 16	Slow	Fast	High	High

Figure 152 shows the results for various series. The overall trend looks very similar to the measured data. However, the graph looks a bit shifted leftward, which means that more power is used in the script for heterogeneous execution than for the measured case. In a real execution, it shows that even though it is a heterogeneous execution, the power consumption does not linearly add up (CPU + GPU); instead, the power settles at a lower value, which is not too far off from using a CPU or GPU alone. However, for this analytical model, since we add the powers together for a heterogeneous execution, the predicted power tends to be higher than the measured case, hence the graph shifts leftwards. If we consider this issue to the analytical model, the graph will shift rightwards. Nevertheless, this graph still provides the valuable information that the dots in the left-bottom quadrant are the ones that we should avoid.

Further configurations are displayed in Figures 153, 154, and 155. Depending on different CPU and GPU performances, along with different performance and power values, the whole graphs shift accordingly.

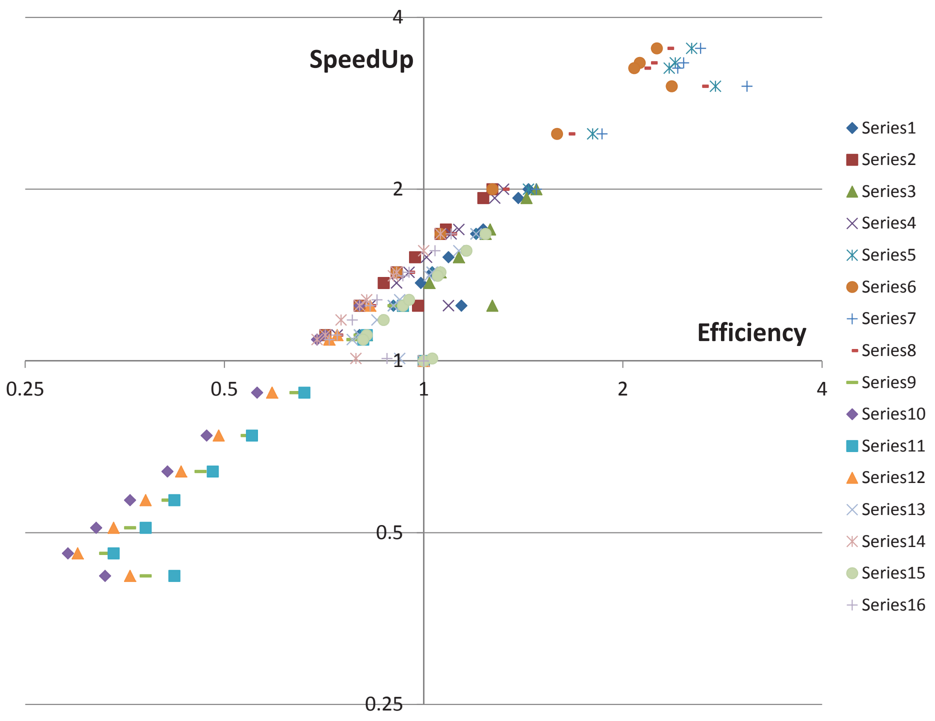


Figure 152: Showing simulations for data partitioning between CPU and GPU. First Configuration Version: Default.

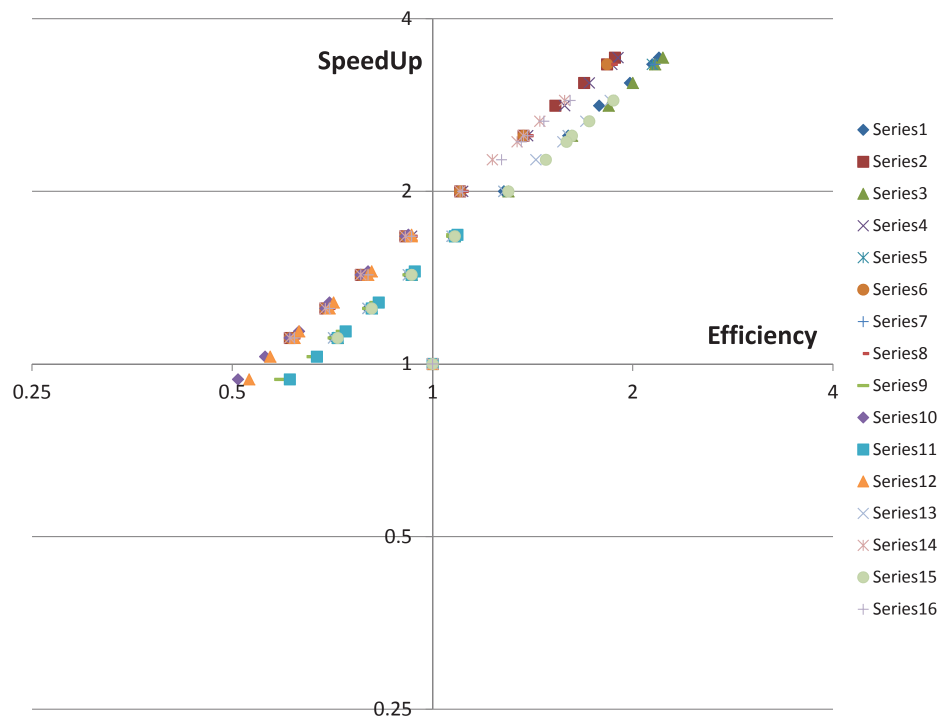


Figure 153: Showing simulations for data partitioning between CPU and GPU. Second Configuration Version: Less CPU performance and power consumption.

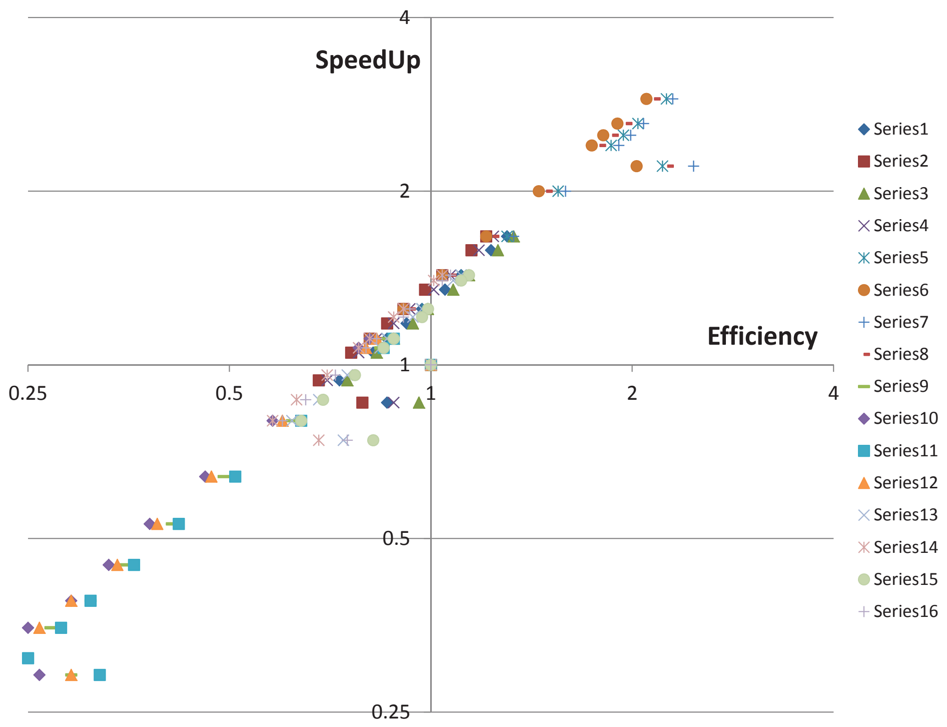


Figure 154: Showing simulations for data partitioning between CPU and GPU. Third Configuration Version: Less GPU performance and power consumption.

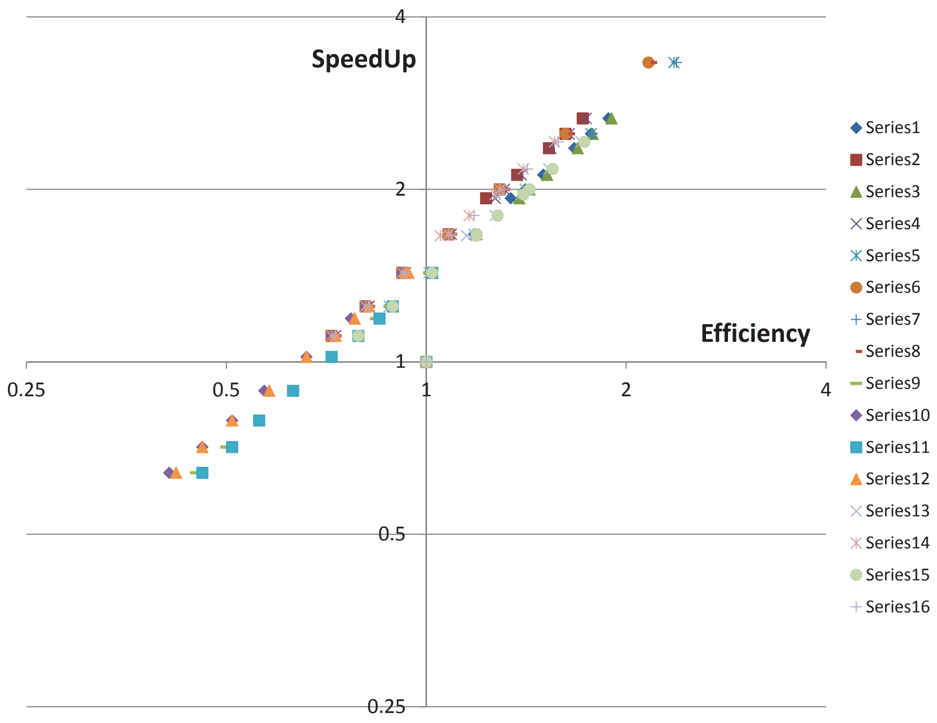


Figure 155: Showing simulations for data partitioning between CPU and GPU. Fourth Configuration Version : Less CPU and GPU performance and power consumptions.

A.5 Discussions

First, the analytical model predictions project that the relationship between energy efficiency and speed-up is linear. Second, we can observe that some sets of series are clustered in the middle portion of the graph, while some are in the upper-right and lower-left quadrants. This information is summarized as follows in Figure 156. Some

```
The series in the bottom-left quadrant
phi_cpu Fast ||  phi_gpu SLOW ||  P_coreweight_cpu_iter LOWPower ||  P_coreweight_gpu_iter LOWPower       9
phi_cpu Fast ||  phi_gpu SLOW ||  P_coreweight_cpu_iter LOWPower ||  P_coreweight_gpu_iter HighPower      10
phi_cpu Fast ||  phi_gpu SLOW ||  P_coreweight_cpu_iter HighPower ||  P_coreweight_gpu_iter LOWPower      11
phi_cpu Fast ||  phi_gpu SLOW ||  P_coreweight_cpu_iter HighPower ||  P_coreweight_gpu_iter HighPower     12

The series in the top-right quadrant
phi_cpu SLOW ||  phi_gpu Fast ||  P_coreweight_cpu_iter LOWPower ||  P_coreweight_gpu_iter LOWPower       5
phi_cpu SLOW ||  phi_gpu Fast ||  P_coreweight_cpu_iter LOWPower ||  P_coreweight_gpu_iter HighPower      6
phi_cpu SLOW ||  phi_gpu Fast ||  P_coreweight_cpu_iter HighPower ||  P_coreweight_gpu_iter LOWPower      7
phi_cpu SLOW ||  phi_gpu Fast ||  P_coreweight_cpu_iter HighPower ||  P_coreweight_gpu_iter HighPower     8
```

Figure 156: The series in the left-bottom and top-right quadrants.

immediate conclusion that can be drawn from the result is that when an application runs very efficiently on a CPU, but not so much on a GPU, it is better not to schedule *any* work to a GPU in the first place. Because of the GPU data transfer time as well as the power increase, the scheduling is not worth the performance and power. The contribution of this work is to show this fact analytically.

On the other hand, if an application runs very non-efficiently on a CPU, but the GPU is very efficient, then it is necessary to schedule as much work as possible to the GPU *regardless* of whether the CPU or GPU is power efficient or not, as shown in Figure 156. The model suggests that power does have an impact; however, performance is the main contributing factor.

A.6 Projections of Different Frequencies

Figure 157 shows the effects of changing frequencies in the speedup and efficiency space for different data partitioning points. As the frequency has a cubic relationship on power, the graph tends to shift to the left as well (i.e., overall direction is moving top left). Figures 158 and 159 show the effects of changing frequency in more detail.

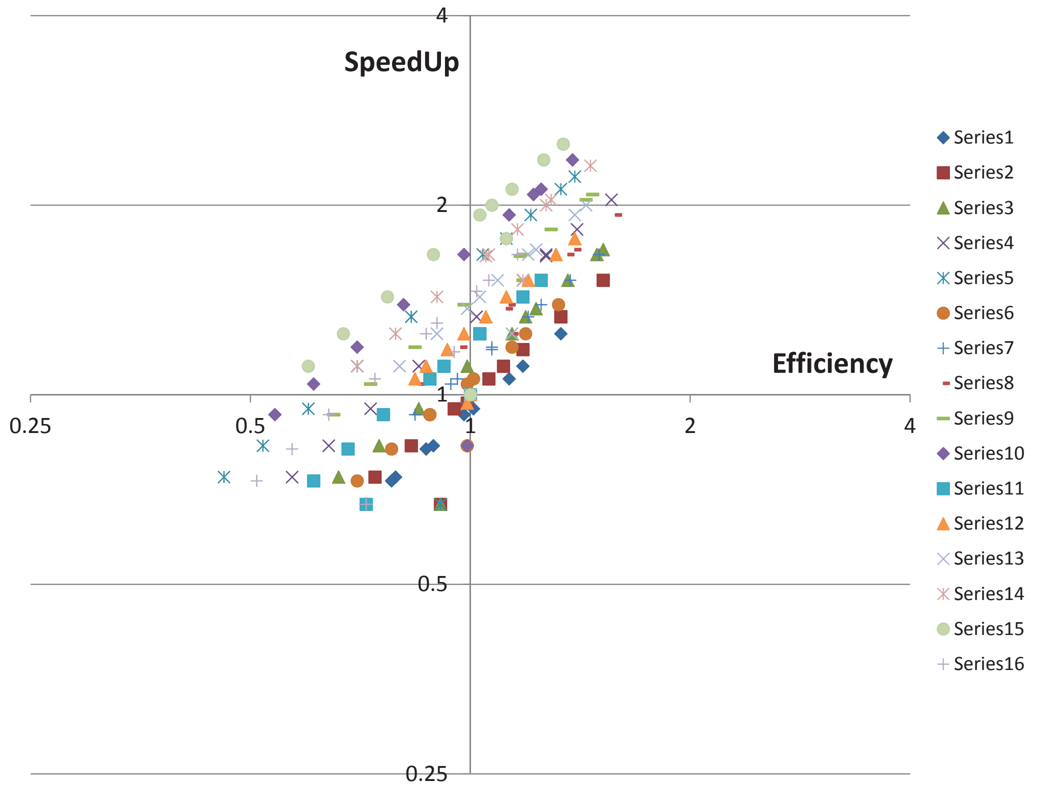


Figure 157: Space exploration for changing frequencies.

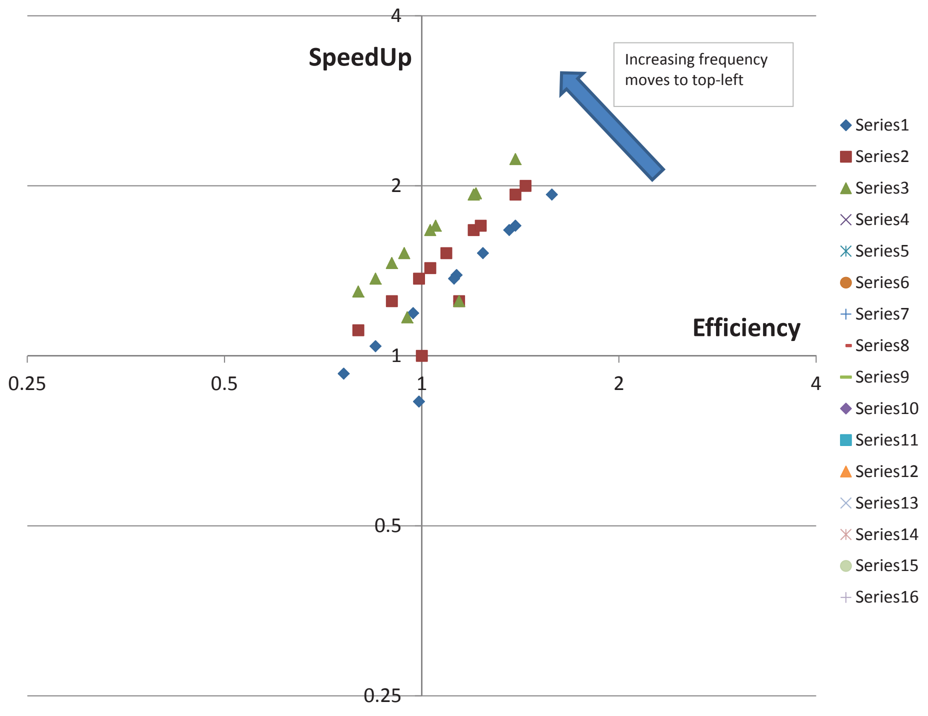


Figure 158: Showing the effects of increasing CPU frequency.

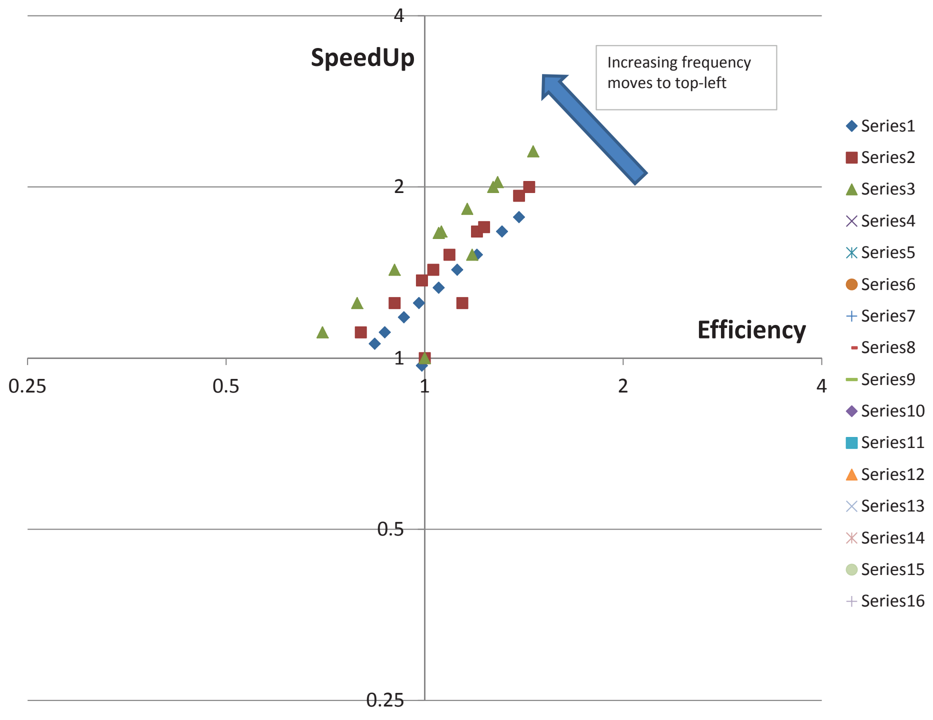


Figure 159: Showing the effects of increasing GPU frequency.

A.7 Discussion: Factors that Determine the Rate of Change in Energy Efficiency

For most cases, we observe a linear relationship. But what are the baseline factors and the underlying reasons ? To investigate this effect, we revisit Equation (103).

To see if it is a linear relationship, we need to check the following condition.

$$\mathsf{E}(\mathsf{S}) = \frac{P_o}{P_N} \times \mathsf{S} \tag{115}$$

$$\frac{\partial^2}{\partial \mathsf{S}^2}\mathsf{E}(\mathsf{S}) = 0 \qquad \text{For a linear increase} \tag{116}$$

$$\frac{\partial^n}{\partial \mathsf{S}^n}\mathsf{E}(\mathsf{S}) \neq 0 \qquad \text{For a non-linear increase, } n \geq 2 \tag{117}$$

The challenging part is to know whether or not P_o is a function of S. And this will affect the shape of an energy-efficiency function.

A.8 Summary

The contributions of this work are to clearly derive the relationship between the energy-efficiency and speed-up among heterogeneous architectures. First, the model shows why the relationship is mostly linear. The underlying reason is that the rate of power change with respect to cores, frequency, etc. is not fast enough. Second, the model has been implemented in the script for simulation purposes, and it showed the effects of different configurations of heterogeneous executions including different high/low power and fast/slow performances. Another implication is that performance has a much higher impact on energy efficiency if the GPU is very fast in execution, as discussed in Section A.5.

Finally, the model provides insight into the linear relationship between energy-efficiency and speed-up. The equation claims that unless the rate of change for power increase is in a square or more, it will be most likely a linear relationship. The future work is to improve the power and performance models and provide further insights on how to optimize energy efficiency given many parameters during runtime, analytically or statically, using a machine-learning mechanism.

REFERENCES

[1] "Extech 380801." http://www.extech.com/.

[2] "National instruments fp-tc 120 thermocouple data logger."

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

[4] "Nvidia's geforce gtx280 graphics processor."

[5] Ali, A., Dastgeer, U., and Kessler, C., "Opencl for programming shared memory multicore cpus," in *MULTIPROG '12: Fifth Workshop on Programmability Issues for Multi-Core Computers*, 2012.

[6] Azizi, O., Mahesri, A., Lee, B. C., Patel, S. J., and Horowitz, M., "Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis," in *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, (New York, NY, USA), pp. 26–36, ACM, 2010.

[7] Baghsorkhi, S. S., Delahaye, M., Patel, S. J., Gropp, W. D., and Hwu, W.-m. W., "An adaptive performance modeling tool for gpu architectures," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, (New York, NY, USA), pp. 105–114, ACM, 2010.

[8] Borkar, S., "Design challenges of technology scaling," *IEEE Micro*, vol. 19, no. 4, pp. 23–29, 1999.

[9] Branover, A., Foley, D., and Steinman, M., "Amd fusion apu: Llano," *Micro, IEEE*, vol. 32, pp. 28 –37, march-april 2012.

[10] Brooks, D., Tiwari, V., and Martonosi, M., "Wattch: a framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, (New York, NY, USA), pp. 83–94, ACM, 2000.

[11] Brown, D. J. and Reams, C., "Toward energy-efficient computing," *Queue*, vol. 8, pp. 30:30–30:43, Feb. 2010.

[12] Butts, J. A. and Sohi, G. S., "A static power model for architects," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, (New York, NY, USA), pp. 191–201, ACM, 2000.

[13] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.-H., and Skadron, K., "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44 –54, oct. 2009.

[14] Chen, X. E. and Aamodt, T. M., "A First-Order Fine-Grained Multithreaded Throughput Model," in *IEEE International Symposium on High Performance Computer Architecture (HPCA 2009)*, pp. 329–340, Feb. 2009.

[15] Choi, J. W., Singh, A., and Vuduc, R. W., "Model-driven autotuning of sparse matrix-vector multiply on gpus," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, (New York, NY, USA), pp. 115–126, ACM, 2010.

[16] Contreras, G. and Martonosi, M., "Power prediction for intel xscale&#174; processors using performance monitoring unit events," in *Proceedings of the 2005 international symposium on Low power electronics and design*, ISLPED '05, (New York, NY, USA), pp. 221–226, ACM, 2005.

[17] Diamos, G. F., Kerr, A. R., Yalamanchili, S., and Clark, N., "Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 353–364, ACM, 2010.

[18] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D., "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, (New York, NY, USA), pp. 365–376, ACM, 2011.

[19] Esmaeilzadeh, H., Cao, T., Yang, X., Blackburn, S. M., and McKinley, K. S., "Looking back and looking forward: power, performance, and upheaval," *Commun. ACM*, vol. 55, pp. 105–114, July 2012.

[20] Eyerman, S., Bois, K., and Eeckhout, L., "Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '12, 2012.

[21] Eyerman, S., Eeckhout, L., Karkhanis, T., and Smith, J. E., "A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst.*, vol. 27, pp. 3:1–3:37, May 2009.

[22] Eyerman, S., Hoste, K., and Eeckhout, L., "Mechanistic-empirical processor performance modeling for constructing cpi stacks on real hardware," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, (Washington, DC, USA), pp. 216–226, IEEE Computer Society, 2011.

[23] Floyd, M., Ghiasi, S., Keller, T., Rajamani, K., Rawson, F., Rubio, J., and Ware, M., "System power management support in the ibm power6 microprocessor," *IBM Journal of Research and Development*, 2007.

[24] Fu, R., Zhai, A., Yew, P.-C., Hsu, W.-C., and Lu, J., "Reducing queuing stalls caused by data prefetching," in *The 11th Annual Workshop on the Interaction between Compilers and Computer Architecture*, 2007.

[25] Fung, W. W. L., Sham, I., Yuan, G., and Aamodt, T. M., "Dynamic warp formation and scheduling for efficient gpu control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, (Washington, DC, USA), pp. 407–420, IEEE Computer Society, 2007.

[26] Ge, R., Feng, X., and Cameron, K., "Modeling and evaluating energy-performance efficiency of parallel processing on multicore based power aware systems," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1 –8, may 2009.

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

[28] Govindaraju, N. K., Larsen, S., Gray, J., and Manocha, D., "A memory model for scientific algorithms on graphics processors," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, (New York, NY, USA), ACM, 2006.

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

[30] Hackenberg, D., Molka, D., and Nagel, W. E., "Comparing cache architectures and coherency protocols on x86-64 multicore smp systems," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 413–422, ACM, 2009.

[31] Hamano, T., Endo, T., and Matsuoka, S., "Power-aware dynamic task scheduling for heterogeneous accelerated clusters," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1 –8, may 2009.

[32] Heirman, W., Carlson, T., Che, S., Skadron, K., and Eeckhout, L., "Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pp. 38 –49, nov. 2011.

[33] Hong, S. and Kim, H., "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, (New York, NY, USA), pp. 152–163, ACM, 2009.

[34] Hong, S. and Kim, H., "An integrated gpu power and performance model," in *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, (New York, NY, USA), pp. 280–289, ACM, 2010.

[35] Huang, S., Xiao, S., and Feng, W., "On the energy efficiency of graphics processing units for scientific computing," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, (Washington, DC, USA), pp. 1–8, IEEE Computer Society, 2009.

[36] Huang, W., Rajamani, K., Stan, M., and Skadron, K., "Scaling with design constraints: Predicting the future of big chips," *Micro, IEEE*, vol. 31, pp. 16 –29, july-aug. 2011.

[37] Huang, W., Skadron, K., Gurumurthi, S., Ribando, R. J., and Stan, M. R., "Differentiating the roles of ir measurement and simulation for power and temperature-aware design," in *ISPASS '09: Proc. of the IEEE Int'l. Symp. on Performance Analysis of Systems and Software, 2009*, April 2009.

[38] Huang, W., Stant, M. R., Sankaranarayanan, K., Ribando, R. J., and Skadron, K., "Many-core design from a thermal perspective," in *DAC '08: Proc. of the 45th conference on Design automation*, 2008.

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

[40] Intel, "Intel®Nehalem Microarchitecture." http://www.intel.com/technology/architecture-silicon/next-gen/index.htm?iid=tech_micro+nehalem.

[41] Intel Corporation, "Intel OpenCL SDK." http://software.intel.com/en-us/articles/intel-opencl-sdk/.

[42] Intel Corporation, *Writing Optimal OpenCL Code with Intel OpenCL SDK*.

[43] Ïpek, E., McKee, S. A., Caruana, R., de Supinski, B. R., and Schulz, M., "Efficiently exploring architectural design spaces via predictive modeling," *SIGPLAN Not.*, vol. 41, pp. 195–206, Oct. 2006.

[44] Isci, C. and Martonosi, M., "Runtime power monitoring in high-end processors: Methodology and empirical data," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, (Washington, DC, USA), pp. 93–, IEEE Computer Society, 2003.

[45] Jablin, T. B., Prabhu, P., Jablin, J. A., Johnson, N. P., Beard, S. R., and August, D. I., "Automatic cpu-gpu communication management and optimization," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, (New York, NY, USA), pp. 142–151, ACM, 2011.

[46] Jia, H., Zhang, Y., Long, G., Xu, J., Yan, S., and Li, Y., "Gpuroofline: A model for guiding performance optimizations on gpus.," in *Euro-Par* (Kaklamanis, C., Papatheodorou, T. S., and Spirakis, P. G., eds.), vol. 7484 of *Lecture Notes in Computer Science*, pp. 920–932, Springer, 2012.

[47] Kang, S., Choi, H. J., Kim, C. H., Chung, S. W., Kwon, D., and Na, J. C., "Exploration of cpu/gpu co-execution: from the perspective of performance, energy, and temperature," in *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, (New York, NY, USA), pp. 38–43, ACM, 2011.

[48] Karkhanis, T. S. and Smith, J. E., "A first-order superscalar processor model," in *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, (Washington, DC, USA), pp. 338–, IEEE Computer Society, 2004.

[49] Kerr, A., Diamos, G., and Yalamanchili, S., "A characterization and analysis of ptx kernels," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, (Washington, DC, USA), pp. 3–12, IEEE Computer Society, 2009.

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

[51] Kim, J., Kim, H., Lee, J. H., and Lee, J., "Achieving a single compute device image in opencl for multiple gpus," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, (New York, NY, USA), pp. 277–288, ACM, 2011.

[52] Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., and Lee, J., "Snucl: an opencl framework for heterogeneous cpu/gpu clusters," in *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, 2012.

[53] Kong, J., Chung, S. W., and Skadron, K., "Recent thermal management techniques for microprocessors," *ACM Comput. Surv.*, vol. 44, pp. 13:1–13:42, June 2012.

[54] Kothapalli, K., Mukherjee, R., Rehman, M., Patidar, S., Narayanan, P., and Srinathan, K., "A performance prediction model for the cuda gpgpu platform," in *High Performance Computing (HiPC), 2009 International Conference on*, pp. 463 –472, dec. 2009.

[55] LAVA research group, "Hotspot." `http://lava.cs.virginia.edu/HotSpot`.

[56] Lee, B. C. and Brooks, D. M., "Accurate and efficient regression modeling for microarchitectural performance and power prediction," *SIGARCH Comput. Archit. News*, vol. 34, pp. 185–194, Oct. 2006.

[57] Lee, J. and Kim, H., "Tap: A tlp-aware cache management policy for a cpu-gpu heterogeneous architecture," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2012.

[58] Lee, J., Sathisha, V., Schulte, M., Compton, K., and Kim, N. S., "Improving throughput of power-constrained gpus using dynamic voltage/frequency and core scaling," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, (Washington, DC, USA), pp. 111–120, IEEE Computer Society, 2011.

[59] Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P., "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," in *Proceedings of the 37th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 451–460, ACM, 2010.

[60] Lewis, A. W., Tzeng, N.-F., and Ghosh, S., "Runtime energy consumption estimation for server workloads based on chaotic time-series approximation," *ACM Trans. Archit. Code Optim.*, vol. 9, pp. 15:1–15:26, Oct. 2012.

[61] Li, J. and Martínez, J. F., "Power-performance considerations of parallel computing on chip multiprocessors," *ACM Trans. Archit. Code Optim.*, vol. 2, pp. 397–422, December 2005.

[62] Linderman, M. D., Collins, J. D., Wang, H., and Meng, T. H., "Merge: a programming model for heterogeneous multi-core systems," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, (New York, NY, USA), pp. 287–296, ACM, 2008.

[63] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J., "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, pp. 39–55, March 2008.

[64] Liu, C., Li, J., Huang, W., Rubio, J., Speight, E., and Lin, X., "Power-efficient time-sensitive mapping in heterogeneous systems," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, (New York, NY, USA), pp. 23–32, ACM, 2012.

[65] Liu, W., Muller-Wittig, W., and Schmidt, B., "Performance predictions for general-purpose computation on gpus," in *Proceedings of the 2007 International Conference on Parallel Processing*, ICPP '07, (Washington, DC, USA), pp. 50–, IEEE Computer Society, 2007.

[66] Luk, C.-K., Hong, S., and Kim, H., "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42nd*

Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, (New York, NY, USA), pp. 45–55, ACM, 2009.

[67] Maleki, S., Gao, Y., Garzarán, M. J., Wong, T., and Padua, D. A., "An evaluation of vectorizing compilers," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, 2011.

[68] Meng, J., Morozov, V. A., Kumaran, K., Vishwanath, V., and Uram, T. D., "Grophecy: Gpu performance projection from cpu code skeletons," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 14:1–14:11, ACM, 2011.

[69] Meng, J. and Skadron, K., "Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus," in *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, (New York, NY, USA), pp. 256–265, ACM, 2009.

[70] Mesa-Martinez, F. J., Ardestani, E. K., and Renau, J., "Characterizing processor thermal behavior," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, 2010.

[71] Mesa-Martinez, F. J., Nayfach-Battilana, J., and Renau, J., "Power model validation through thermal measurements," in *ISCA '07: Proc. of the 34th annual Int'l. Symp. on Computer Architecture*, 2007.

[72] Michaud, P. and Seznec, A., "Data-flow prescheduling for large instruction windows in out-of-order processors," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, (Washington, DC, USA), pp. 27–, IEEE Computer Society, 2001.

[73] Michaud, P., Seznec, A., and Jourdan, S., "Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors," in *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, (Washington, DC, USA), pp. 2–, IEEE Computer Society, 1999.

[74] Nickolls, J. and Dally, W., "The gpu computing era," *Micro, IEEE*, vol. 30, pp. 56 –69, march-april 2010.

[75] Noonburg, D. B. and Shen, J. P., "Theoretical modeling of superscalar processor performance," in *Proceedings of the 27th annual international symposium on Microarchitecture*, MICRO 27, (New York, NY, USA), pp. 52–62, ACM, 1994.

[76] NVIDIA, "Fermi: Nvidia's next generation cuda compute architecture." http://www.nvidia.com/fermi.

[77] NVIDIA Corporation, *CUDA Programming Guide, V4.0.*

[78] NVIDIA Corporation, "NVIDIA OpenCL SDK." http://developer.nvidia.com/cuda/opencl/.

[79] OpenCL, "The open standard for parallel programming of heterogeneous systems."
http://www.khronos.org/opencl.

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

[81] Rai, D., Yang, H., Bacivarov, I., and Thiele, L., "Power agnostic technique for efficient temperature estimation of multicore embedded systems," in *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '12, (New York, NY, USA), pp. 61–70, ACM, 2012.

[82] Roy, K., Mukhopadhyay, S., and Mahmoodi-Meimand, H., "Leakage current mechanisms and leakage reduction techniques in deep-submicrometer cmos circuits," *Proceedings of the IEEE*, vol. 91, pp. 305 – 327, Feb 2003.

[83] Ryoo, S., Rodrigues, C. I., Stone, S. S., Baghsorkhi, S. S., Ueng, S.-Z., Stratton, J. A., and Hwu, W.-m. W., "Program optimization space pruning for a multithreaded gpu," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, (New York, NY, USA), pp. 195–204, ACM, 2008.

[84] Saavedra-Barrera, R. H. and Culler, D. E., "An analytical solution for a markov chain modeling multithreaded," tech. rep., Berkeley, CA, USA, 1991.

[85] Saha, B., Zhou, X., Chen, H., Gao, Y., Yan, S., Rajagopalan, M., Fang, J., Zhang, P., Ronen, R., and Mendelson, A., "Programming model for a heterogeneous x86 platform," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation.*

[86] Schaa, D. and Kaeli, D., "Exploring the multiple-gpu design space," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2009.

[87] Seo, S., Jo, G., and Lee, J., "Performance characterization of the nas parallel benchmarks in opencl," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pp. 137 –148, nov. 2011.

[88] Sheaffer, J. W., Luebke, D., and Skadron, K., "A flexible simulation framework for graphics architectures," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '04, (New York, NY, USA), pp. 85–94, ACM, 2004.

[89] Sim, J., Dasgupta, A., Kim, H., and Vuduc, R., "A performance analysis framework for identifying potential benefits in gpgpu applications," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), pp. 11–22, ACM, 2012.

[90] Skadron, K., Stan, M. R., Sankaranarayanan, K., Huang, W., Velusamy, S., and Tarjan, D., "Temperature-aware microarchitecture: Modeling and implementation," *ACM Trans. Archit. Code Optim.*, vol. 1, pp. 94–125, March 2004.

[91] Sorin, D. J., Pai, V. S., Adve, S. V., Vernon, M. K., and Wood, D. A., "Analytic evaluation of shared-memory systems with ilp processors," in *Proceedings of the 25th annual international symposium on Computer architecture*, ISCA '98, (Washington, DC, USA), pp. 380–391, IEEE Computer Society, 1998.

[92] Su, H., Liu, F., Devgan, A., Acar, E., and Nassif, S., "Full chip leakage estimation considering power supply and temperature variations," in *Proceedings of the 2003 international symposium on Low power electronics and design*, ISLPED '03, (New York, NY, USA), pp. 78–83, ACM, 2003.

[93] Suleman, M. A., Qureshi, M. K., and Patt, Y. N., "Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, (New York, NY, USA), pp. 277–286, ACM, 2008.

[94] Takizawa, H., Sato, K., and Kobayashi, H., "Sprat: Runtime processor selection for energy-aware computing," in *Cluster Computing, 2008 IEEE International Conference on*, pp. 386 –393, 29 2008-oct. 1 2008.

[95] Tarjan, D., Meng, J., and Skadron, K., "Increasing memory miss tolerance for simd cores," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, (New York, NY, USA), pp. 22:1–22:11, ACM, 2009.

[96] Taylor, M. B., "Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse," in *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, (New York, NY, USA), pp. 1131–1136, ACM, 2012.

[97] The IMPACT Research Group, UIUC, "Parboil benchmark suite." http://impact.crhc.illinois.edu/parboil.php.

[98] Thomadakis, M., "The architecture of the nehalem processor and nehalem-ep smp platforms," tech. rep., Texas A&M University, 2011.

[99] Wang, G. and Ren, X., "Power-efficient work distribution method for cpu-gpu heterogeneous system," in *Parallel and Distributed Processing with Applications (ISPA), 2010 International Symposium on*, pp. 122 –129, sept. 2010.

[100] Wang, H., Sathish, V., Singh, R., Schulte, M. J., and Kim, N. S., "Workload and power budget partitioning for single-chip heterogeneous processors," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, (New York, NY, USA), pp. 401–410, ACM, 2012.

[101] Waring, C. and Liu, X., "Face detection using spectral histograms and svms," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 35, pp. 467 –476, June 2005.

[102] Wil Braithwaite, "The Art of Performance Optimization." SIGGRAPH 2009 Tutorial.

[103] Williams, S., Waterman, A., and Patterson, D., "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, April 2009.

[104] Zhang, Y., Parikh, D., Sankaranarayanan, K., Skadron, K., and Stan, M., "Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects," tech. rep., University of Virginia, 2003.

[105] Zhang, Y. and Owens, J. D., "A quantitative performance analysis model for GPU architectures," in *Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA 17)*, pp. 382–393, Feb. 2011.