Understanding Latency Hiding on GPUs

Vasily Volkov

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-143
http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html

August 12, 2016

Understanding Latency Hiding on GPUs

by

Vasily Volkov

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

and the Designated Emphasis

in

Computational and Data Science and Engineering

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor James W. Demmel, Chair
Professor Krste Asanović
Professor Ming Gu

Summer 2016

Understanding Latency Hiding on GPUs

by

Vasily Volkov

Abstract

Understanding Latency Hiding on GPUs

by

Vasily Volkov

Doctor of Philosophy in Computer Science

and the Designated Emphasis in

Computational and Data Science and Engineering

University of California, Berkeley

Professor James W. Demmel, Chair

Modern commodity processors such as GPUs may execute up to about a thousand of physical threads per chip to better utilize their numerous execution units and hide execution latencies. Understanding this novel capability, however, is hindered by the overall complexity of the hardware and complexity of typical workloads. In this dissertation, we suggest a better way to understand modern multithreaded performance by considering a family of synthetic workloads, which use the same key hardware capabilities – memory access, arithmetic operations, and multithreading – but are otherwise as simple as possible.

One of our surprising findings is that prior performance models for GPUs fail on these workloads: they mispredict observed throughputs by factors of up to 1.7. We analyze these prior approaches, identify a number of common pitfalls, and discuss the related subtleties in understanding concurrency and Little's Law. Also, we help to further our understanding by considering a few basic questions, such as on how different latencies compare with each other in terms of latency hiding, and how the number of threads needed to hide latency depends on basic parameters of executed code such as arithmetic intensity. Finally, we outline a performance modeling framework that is free from the found limitations.

As a tangential development, we present a number of novel experimental studies, such as on how mean memory latency depends on memory throughput, how latencies of individual memory accesses are distributed around the mean, and how occupancy varies during execution.

Contents

Chapter 1
Introduction

Multithreading is a latency hiding technique that is widely used in modern commodity processors such as GPUs. It involves executing many instances of the same or different programs at the same time; the instances are called threads or warps. Multithreading allows better use of the widely replicated and deeply pipelined resources in modern, multi-billion transistor chips.

It is typically expected that executing more threads at the same time results in better performance, up to a limit. An example is shown in Figure 1.1. The vertical axis in the graphs is throughput, which is the amount of work done per unit time. The horizontal axis is occupancy, which is the number of warps executed at the same time per SM. SM stands for "streaming" multiprocessor, which is the replicated processor tile and a loose equivalent of CPU core. (It is more common to

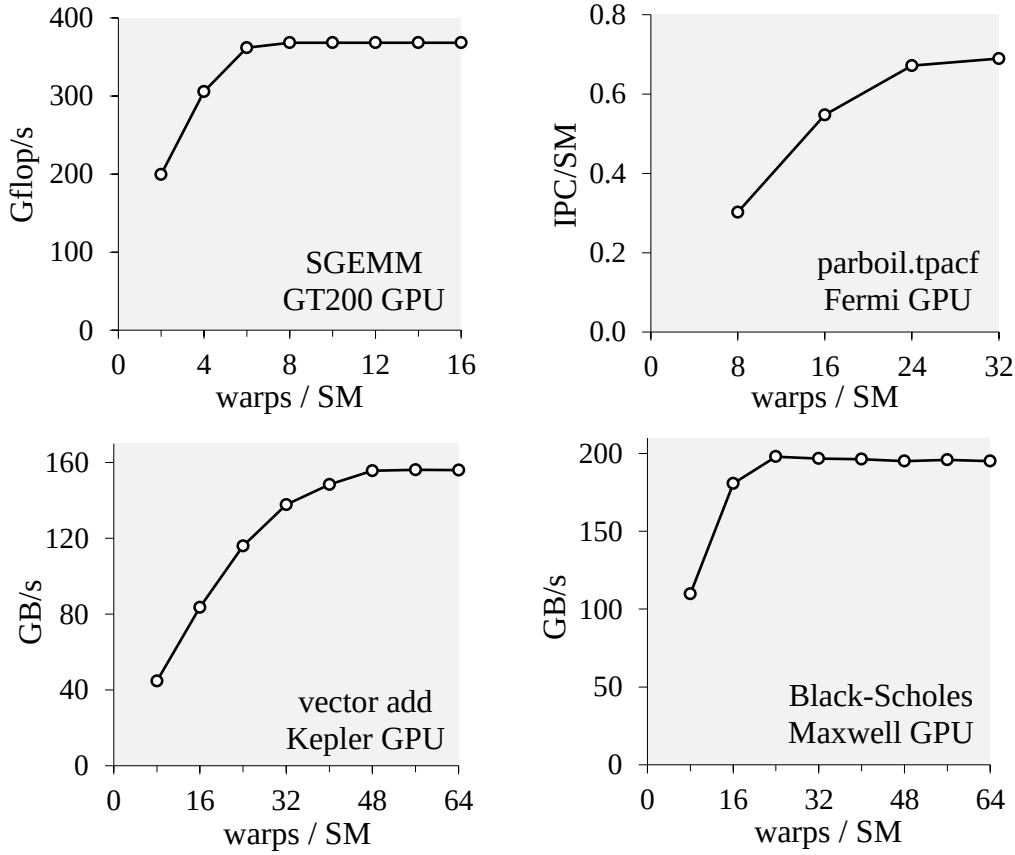


Figure 1.1: Executing more threads at the same time typically results in a better throughput, but only up to a limit[1].

[1] Data for SGEMM is from Volkov and Demmel [2008], Section 4.2. Other data is collected as described in Chapter 5.

define occupancy as fraction of the maximum supported number of warps; when speaking of the same processor, the difference is only in units.) All graphs in the figure share the same feature: throughput is larger at larger occupancy but not larger than a limit.

The improvements in throughput are due to latency hiding, i.e. overlapping execution times (latencies) of individual operations with execution times of other operations, such as operations in other warps.

How well do we understand this basic behavior? For example, what occupancy is needed to attain the best possible throughput? How does it depend on the GPU microarchitecture and executed code? How does one estimate execution time given occupancy, code, and any necessary information about the processor? Finally, what are the key differences between modeling multithreaded performance today and 25 years ago?

1.1 Focus on basics

Multithreading cannot be studied separately from other processor functions; rather, we may study how their performance depends on multithreading. Basic processor functions include computation, memory access and instruction issue. Each of them has a finer structure. There are different types of arithmetic units and different types of memories. Memory accesses may involve bank conflicts, memory divergence and caching. Instruction issue includes scoreboarding and dual-issue. Multithreading is implemented as part of instruction issue. GPU microarchitecture is thus fairly complicated.

The complexity of the hardware is reflected in complexity and diversity of the respective performance models. We are primarily interested in analytical performance models, which suggest formulas to predict execution time given a sufficiently detailed description of microarchitecture. There were many of them proposed. Some performance models are more complicated, such as Hong and Kim [2009], Sim et al. [2012] and Huang et al. [2014], which include over 20 equations each. Many feature unique approaches; for example, Zhang and Owens [2011] suggest using extensive experimental data instead of microarchitectural parameters, and Baghsorkhi et al. [2010] formulate their model as operations on a graph, such as weight assignments and reductions. The list of other performance models for GPUs includes Song et al. [2013], Kothapalli et al. [2009], Guz et al. [2010], Resios [2011], Ma et al. [2014], Alberto and Hiroyuki [2015], and Dao et al. [2015].

The suggested approaches share little in common, despite considering fundamentally similar GPUs, mostly from NVIDIA. There were several generations of them released: G80, GT200, Fermi, Kepler, Maxwell, and now Pascal. Newer generations include many improvements, most of which, however, – such as atomic operations, load-store architecture, static scheduling, dynamic frequency scaling, new graphics functions and elimination of shader clock – are finer than the level of detail considered in performance models. The basic microarchitecture is essentially the same in all cases: all general-purpose NVIDIA GPUs have similar fine-grained multithreading, similar types of functional units (e.g. CUDA cores and SFU units), and process basic kinds of memory accesses in a similar manner. Thus, we have no agreement on how to model performance of even the most basic GPU functionality.

The purpose of this work is to address this lack, even if in a limited manner. In order to do so we substantially simplify the problem.

Our reference point is the AMAT model for cached memory access [Hennessy and Patterson 2011]. It leaves aside many relevant and important features, such as memory bandwidth, cache bandwidth, prefetching, and concurrency. Yet, it is useful in understanding how cache parameters affect overall processor performance.

Pseudo-code	*Assembly code*
a = *memory*[*a*]	`LD R1, [R1]`
a = *a* + *b*	`FADD R1, R1, R2`
a = *a* + *b*	`FADD R1, R1, R2`
a = *a* + *b*	`FADD R1, R1, R2`
a = *memory*[*a*]	`LD R1, [R1]`
a = *a* + *b*	`FADD R1, R1, R2`
…	`...`

Listing 1.1: Each warp in our test workload executes what approximates an infinite instruction sequence similar to this.

Similarly, we develop a simplistic performance model that covers only a few basic processor functions. Yet, it provides new and surprising insights, not predicted in prior work.

1.2 An insightful test case

We prune the space of all possible GPU codes down to only those, where all arithmetic instructions are the same, all memory instructions are the same, the memory access pattern is one of the simplest and most common, all instructions are back-to-back dependent, and each warp executes what approximates an infinite instruction sequence. The only degrees of freedom kept are arithmetic intensity [Harris 2005; Williams et al. 2009] and occupancy. An example of the respective code is shown in Listing 1.1.

Estimating performance in such workloads, on one hand, requires considering the same key processor functions as in any other performance model, i.e. computation, memory access, and multithreading, but, on the other hand, is trivial and unambiguous – under reasonable assumptions. It is also not difficult to find ground truth by timing the execution of similar codes in practice. As at this level most GPUs are similar, the result is a generic performance model for GPU-like multithreading albeit with a minimalist feature set. If we are to understand performance of modern multithreaded processors, this may be a good starting point.

Another application of these simplistic workloads is getting a better insight into other performance models. In contrast to appreciating abstract concepts and logic behind the proposed equations, these workloads allow understanding performance models intuitively by working through a concrete example with pencil and paper. Doing so with realistic codes is more cumbersome, and the result is more difficult to evaluate. Due to the simplicity, even the more algebra-intensive performance models in this case are reduced to only a few relatively simple equations.

Finally, these workloads may also be used as a quick evaluation tool for performance models. Such evaluation is inexpensive, transparent, and easily reproducible. Yet, it tests some of their critical features.

1.3 Evaluation of prior performance models

We demonstrate the usefulness of this approach by using it to evaluate the following well-known performance models:

- Hong and Kim [2009],
- CUDA C Programming Guide [NVIDIA 2015, Ch. 5.2.3],
- Baghsorkhi et al. [2010],

- Zhang and Owens [2011] (slightly simplified),
- Sim et al. [2012], and
- Huang et al. [2014].

The surprising result is that all of them are found to have substantial limitations.

Several examples are given in Figure 1.2. The estimates are shown as a thick grey line, and experimental data is shown as dots.

The most common limitation is to substantially underestimate the occupancy that is needed to attain a maximum throughput, such as by a factor of up to about two. This limitation shows up when arithmetic intensity of the executed code is intermediate. We find it in such performance models as by Sim et al. [2012], Zhang and Owens [2011], Baghsorkhi et al. [2010] and Hong and Kim [2009], and in the CUDA C Programming Guide [NVIDIA 2015]. An example corresponding to the Sim et al. model is shown in Figure 1.2, a, left. A similar example for the CUDA C Programming Guide is shown in Figure 1.2, a, right. In the latter case the estimate is shown as a vertical bar because the respective analysis is limited to estimating the needed occupancy only.

Another repeating pattern is poor accuracy on memory-intensive codes, i.e. at small arithmetic intensities. In this case the performance model by Baghsorkhi et al. [2010] is found to severely underestimate throughput, as shown in Figure 1.2, b, left, and the performance model by Huang et al. [2014], in contrast, is found to substantially overestimate throughput, as shown in Figure 1.2, b, right.

Some performance models are found to have unique limitations. The performance model by Huang et al. [2014] is found to produce, in some cases, negative execution times. Also, one of its versions is found to substantially overestimate throughput in both memory-intensive and arithmetic-intensive codes. The performance model by Baghsorkhi et al. [2010], as shown in Figure 1.2, c, left, is found to suggest in some cases an odd dependence of throughput on occupancy, which is different from what we see in practice.

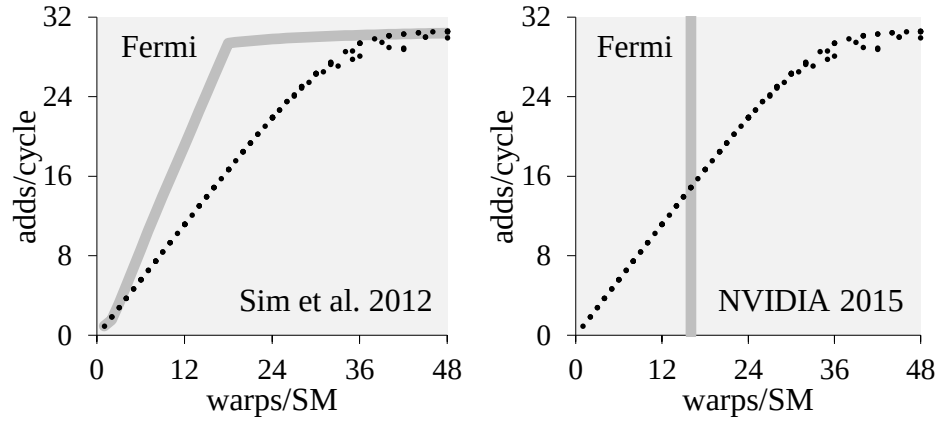
Despite these difficulties, the straightforward performance model that is easily found for this simplistic workload doesn't have these limitations. A typical example is shown in Figure 1.2, c, right. This simplistic workload thus helps funding "bugs" in performance models, which may not be found otherwise. Also, the fact that all of these better-known performance models are found to have substantial limitations suggests that our understanding of GPU performance is substantially limited.

1.4 The lessons

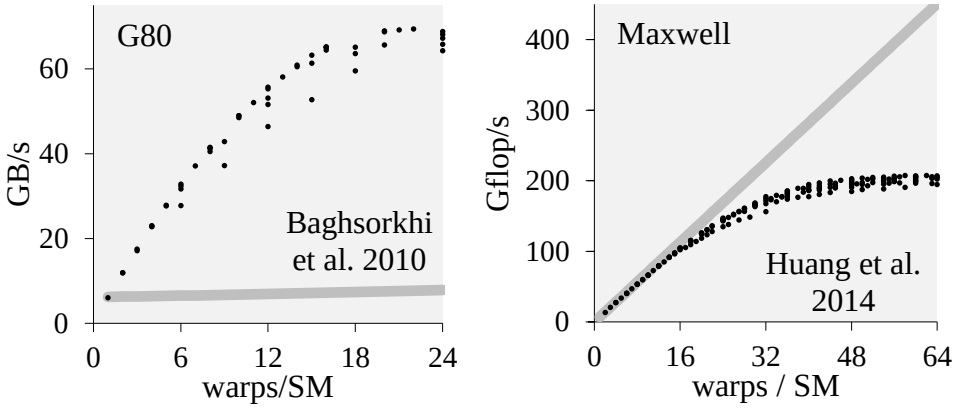
Given the simplicity of this workload, we can easily identify what factors are important in finding an accurate performance estimate.

One of these factors is found to be arithmetic latency. It is short compared to memory latency – only 6 to 24 cycles versus around 400 cycles – and is often deemphasized. Yet, ignoring arithmetic latency may be as detrimental to accuracy as ignoring memory latency. This is due to what is known as Little's law [Little 1961], which is a simple relation that states that latency multiplied by throughput equals concurrency. Arithmetic latency is short but maximum arithmetic throughput is large, so that arithmetic concurrency can be substantial and as substantial as memory concurrency if both are expressed in similar units. Ignoring small latency thus may imply ignoring substantial concurrency and lead to substantial inaccuracy.

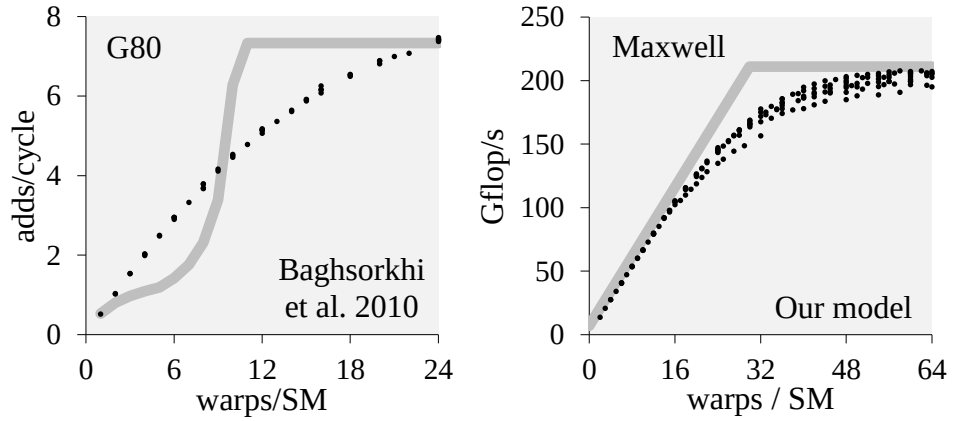
Another key factor is memory bandwidth. It was long understood that memory bandwidth is an important factor in processor performance, see for example Burger et al. [1996]. Yet, many



(a) Many prior performance models tend to underestimate the occupancy needed to attain a maximum throughput.



(b) Some performance models are substantially less accurate when arithmetic intensity is small.



(c) Some performance models show odd behavior (left). Our simple performance model has fewer limitations (right).

Figure 1.2: Estimated and observed performance in the simplistic workload.

performance models do not take it into account. Memory bandwidth is not taken into account in the well-known AMAT model cited earlier [Hennessy and Patterson 2011]. It is not taken into account in some of the early performance models for latency hiding, such as by Saavedra-Barrera et al. [1990] and Mowry et al. [1992]. It is not considered in a recent work on multithreaded

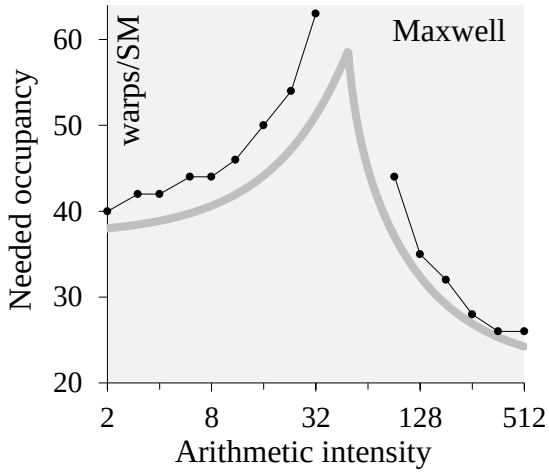


Figure 1.3: A characteristic feature of modern multithreaded performance, such as on GPUs, is cusp behavior. Here, shown as a grey line is an estimate, shown as points is experimental data. Arithmetic intensity is the number of arithmetic instructions per memory access instruction.

performance by Chen and Aamodt [2009]. Similarly, memory bandwidth is not considered in a number of GPU performance models, such as in Kothapalli et al. [2009], in many important cases in Huang et al. [2014], and in the brief review of latency hiding in the widely distributed vendor's programming guide [NVIDIA 2015, Ch. 5.2.3]. The GPU performance model by Baghsorkhi et al. [2010] does take memory bandwidth into account but does not integrate it well with a model for latency hiding. The popular Roofline model [Williams et al. 2009] does take memory bandwidth into account, but doesn't explain how to take into account latencies.

Thus, a key requirement for performance modeling for modern multithreaded processors such as GPUs is taking into account both multiple latencies and multiple throughput limits. (Other throughput limits besides memory bandwidth include, for example, the instruction issue rate.)

The problem of modeling latency hiding with several latencies to hide is relatively new. Some of the evaluated performance models do consider multiple latencies and yet fail on this simple test. The failure may be attributed, at least in some cases, to another missing component: understanding the difference between warp concurrency and instruction concurrency. A curious manifestation of this difference is what we call cusp behavior: attaining maximum throughput in the simplistic workload requires more warps when arithmetic intensity is intermediate than when arithmetic intensity is very small or very large, as shown in Figure 1.3. Putting it in other words, hiding both arithmetic and memory latency at the same time may require more warps than hiding either of them alone. We don't find this property in previously published performance models.

1.5 Towards a better performance model

The next step is extending the findings to performance estimation of realistic GPU workloads. We address it only in a limited manner.

We propose a generic framework for modeling GPU performance. Its key idea is to separately consider latency-bound and throughput-bound cases, which are the two distinct modes recognizable in many performance plots, such as those in Figure 1.1. Latency-bound mode corresponds to small occupancies, where throughput grows linearly or sublinearly with occupancy; as an approximation we assume that the growth is always linear. Throughput-bound mode corresponds to large occupancies, where throughput is constant with respect to occupancy and

attains a maximum. The two modes are similar to asymptotic bounds on performance of concurrent systems discussed in classical work, such as Lazowska et al. [1984] and Jain [1991]. In the latency-bound case we take into account various processor latencies. In the throughput-bound case we incorporate different throughput limits.

We show only a minimal demonstration of using this framework for performance estimation of realistic workloads. Instead we focus on showing that this framework, unlike prior performance models, is simple enough to be used in intuitive judgements. In particular, we are concerned with understanding how the occupancy needed to attain maximum throughput depends on various practical factors. We consider a few examples, in which we can easily predict at a qualitative level what happens with the needed occupancy when the same code is executed on different data, when the same code is compiled with different compiler options, when memory access pattern is altered, etc.

We also present a limited study of the sublinear behavior. It is found to be partially explained by memory contention. We measure how mean memory latency depends on memory contention, suggest a simple model for it, and show how this model can be integrated into the overall performance model.

1.6 Contributions of this work

We claim the following contributions:

1. We present a simple and insightful workload that helps understanding GPU performance and prior performance models (Chapter 4).
2. We evaluate a number of prior performance models in a uniform manner and identify some of their previously unknown limitations (Chapter 7).
3. We review common approaches to modeling GPU performance and highlight a number of pitfalls. For example, we point out that the problem of modeling latency hiding with multiple latencies to hide is relatively new and was not solved well before (Chapter 4).
4. We elaborate on using Little's law in GPU performance modeling and analysis (Chapters 3 and 4).
5. We suggest a simple framework for modeling GPU performance. It does not have some of the important limitations found in prior work and is based on classical work on concurrent systems (Chapter 3).
6. We measure and model how mean memory latency depends on memory contention. We show how to integrate such a latency model into an overall performance model without producing negative execution times, which we find in prior work (Chapter 6).
7. We measure the distribution of latencies of individual memory access instructions at different levels of memory contention (Chapter 6).
8. We measure how occupancy varies during execution and analyze a few patterns of such variance (Chapter 5).

1.7 Outline of this work

The presented topics are developed in the following text in order of increasing complexity.

In Chapter 2 we cover the necessary basics of GPU architecture and describe the specific GPUs used in this work. We use five GPUs representing five generations of GPU architecture: G80, GT200, Fermi, Kepler and Maxwell. This is to emphasize that topics addressed in this text are generic and our findings apply across architecture generations. Also, this allows evaluating prior performance models using GPUs that are similar to those used in the original work.

In Chapter 3 we present a novel discussion of GPU performance. We review basic concepts of concurrent execution such as latency, throughput and Little's law and consider a few examples of their application to GPUs. We point out an oversight shared by all previously published work that we know and present a new performance modeling framework that doesn't have it. We briefly show that this framework can be used to estimate throughput, and, in more detail, how it can be used to easily predict how needed occupancy changes when changing executed code or input data.

In Chapter 4 we study in detail the simplistic workload introduced above. First, we consider what occupancies are needed to hide arithmetic and memory latencies alone, and discuss a few relevant trends; this is to highlight the importance of arithmetic latency. Then we consider how occupancy that is needed to attain maximum throughput depends on arithmetic intensity. Thus we get cusp behavior; it is characteristic of GPUs and is a result of having different throughput limits and different latencies to hide. Then we examine prior approaches to performance modeling and identify a number of pitfalls.

In Chapter 5 we detail the setup used to generate experimental data shown throughout the text. Substantial attention is given to occupancy variance. We track how occupancy changes during execution and identify a few patterns where this variance is more pronounced.

In Chapter 6 we use the same experimental setup to extract microarchitectural parameters used in performance models, such as instruction latencies and throughput limits. Also, we present a novel study of memory latency. We measure how mean memory latency depends on attained memory throughput, and how latencies of individual memory accesses are distributed around the mean. The results are used to refine the performance model presented in Chapter 4. In addition, we measure latency and throughput metrics for unstructured global memory accesses and shared memory accesses with and without bank conflicts.

In Chapter 7 we present a detailed evaluation of prior performance models using the simplistic workload introduced earlier. They are found to be substantially limited. Results are compared with the model suggested in Chapter 4, which is found to be more accurate.

Chapter 2
Background on GPU architecture

In this chapter we cover the necessary background on GPU architecture.

GPUs are parallel processors designed to accelerate computer graphics applications. They have faster memory and faster arithmetic capabilities than CPUs, which makes them also attractive for applications other than graphics. Using these capabilities, however, requires a large degree of parallelism. A serial code, for example, typically runs faster on a CPU, not a GPU. As a result, GPUs are never used alone but only as an extension of a CPU-based host system. The main program is executed on the CPU and only small code fragments called kernels are executed on the GPU. If well-optimized, these kernels run at higher rates performing the bulk of the work. In this text we are exclusively interested in the performance of GPU kernels.

In the following, we describe GPU architecture at a level sufficient to understand the remaining text. For another entry-level introduction to GPU architecture see Nickolls and Kirk [2009]. For an introduction oriented towards kernel development see CUDA C programming guide [NVIDIA 2015]. Differences between different generations of GPU architecture are detailed in the respective vendor's whitepapers, such as those listed in Table 2.1. Additional details not published elsewhere are found in NVIDIA patents; some of them are cited below.

2.1 GPUs used in this work

The scope of this study is limited to NVIDIA GPUs. NVIDIA is the leading GPU vendor and also pioneered better support for general-purpose computing on GPUs. Their solution, called CUDA, allows programming NVIDIA GPUs using an extension of C language unburdened by graphics semantics. We use CUDA throughout this work, but the conclusions likely apply to kernels implemented using other frameworks, such as OpenCL [Howes 2015], as long as a similar hardware is used.

Six generations of CUDA-capable GPUs were released so far: G80, GT200, Fermi, Kepler, Maxwell and Pascal. In this study we consider one GPU in each of these generations except Pascal; these GPUs as listed in Table 2.1. Each of them was a flagship product at the time of its release: it offered the leading performance and was used to promote the respective architecture generation. We often refer to these particular GPUs by their generation name. For example, "the Kepler GPU" refers to the GeForce GTX 680 GPU that we use.

GPU	Year	Generation	Related publications and whitepapers by vendor
8800GTX	2006	G80	NVIDIA [2006]
GTX280	2008	GT200	NVIDIA [2008], Lindholm et al. [2008]
GTX480	2010	Fermi	NVIDIA [2009], NVIDIA [2010b], Wittenbrink et al. [2011]
GTX680	2012	Kepler	NVIDIA [2012a], NVIDIA [2012b]
GTX980	2014	Maxwell	NVIDIA [2014a], NVIDIA [2014b]

Table 2.1: GPUs used in this study. All of them are GeForce GPUs.

Serial code	SIMT code
`@!P0 BRA ELSE`	`SSY ENDIF`
`IF: IADD R1, R1, 1`	`@!P0 BRA ELSE`
`IADD R2, R2, 2`	`IF: IADD R1, R1, 1`
`IADD R3, R3, 3`	`IADD R2, R2, 2`
`BRA ENDIF`	`IADD.S R3, R3, 3`
`ELSE: IMUL R4, R4, 4`	`ELSE: IMUL R4, R4, 4`
`IMUL R5, R5, 5`	`IMUL R5, R5, 5`
`IMUL R6, R6, 6`	`IMUL.S R6, R6, 6`
`ENDIF:`	`ENDIF:`

Table 2.2: A serial code versus a GPU code.

2.2 SIMD architecture

One of the key features of GPU architecture is wide SIMD width and a novel hardware support for treating SIMD divergence.

SIMD stands for Single Instruction Multiple Data. All instructions on NVIDIA GPUs are SIMD instructions, i.e. operate on vectors, not scalars. The length of the vector is called SIMD width and equals 32 on all CUDA-capable GPUs released so far. For example, an add instruction adds element-wise 32 numbers in one register to 32 numbers in another register and writes the 32 results into a third register. Each register, therefore, is an array of 32 elements. Each element is 32-bit; dealing with wider data types such as double precision numbers and 64-bit pointers requires using two or more registers.

The GPU programming model conceals SIMD operations by exposing each physical thread as a number of logical threads – as many logical threads as the SIMD width. The GPU program is then written as operating on scalars, not vectors, and many logical threads are created to both populate the SIMD width and run multiple physical threads.

To differentiate the two types of threads, physical threads are called warps and the term "thread" is reserved for logical threads only. In this notation there are 32 threads in 1 warp.

2.3 Single Instruction Multiple Threads (SIMT)

The GPU programming model permits different threads in the same warp to take different paths in the program. But as the underlying architecture is SIMD, these different paths must be mapped into a single path for the warp to be executed. A key feature of GPUs is that this mapping is done during execution by the hardware itself. SIMD processors that implement this novel capability are said to have SIMT architecture; SIMT stands for Single Instruction Multiple Threads. SIMT architecture is a flavor of SIMD architecture.

For a basic understanding of SIMT operation consider execution of an "if … else …" statement, such as:

```
if(condition) { a=a+1; b=b+2; c=c+3; }
else          { d=d*4; e=e*5; f=f*6; }
```

The corresponding assembly code with no SIMD semantics is shown in Table 2.2, left. In the code, `@!P0 BRA` is a predicated branch instruction – the branch is taken if predicate `!P0` is 1, i.e. if `P0` is 0. `P0` is a predicate register that corresponds to variable `condition` above.

Suppose this code is executed in more than one thread. Different threads may branch differently: some may take the if branch, when others take the else branch. This may seem to be incompatible

Instructions		Threads				Warp	
		0	1	2	3	Order	Mask
	SSY ENDIF	0	0	0	0	0	1111
@!P0	BRA ELSE	1	1	1	1	1	1111
IF:	IADD R1, R1, 1		2	2		5	0110
	IADD R2, R2, 2		3	3		6	0110
	IADD.S R3, R3, 3		4	4		7	0110
ELSE:	IMUL R4, R4, 4	2			2	2	1001
	IMUL R5, R5, 5	3			3	3	1001
	IMUL.S R6, R6, 6	4			4	4	1001
ENDIF:	(next instruction)	5	5	5	5	8	1111

Table 2.3: Diverging control flow: P0 = (0,1,1,0).

with SIMD execution: if all of these threads are in the same warp, they all must follow the same instruction sequence – branching differently is not possible.

Yet, a SIMT processor can execute this code as is, despite the potentially diverging paths. When the processor finds that thread paths diverge, it disables some of the threads and executes instructions in one path; then it disables the other threads and executes instructions in the other path. Similar execution is possible on SIMD processors that don't have SIMT architecture but only if this operation is implemented explicitly in the assembly code.

Although the presented code can be executed on a SIMT processor as is, its performance may be lacking. A better SIMT code must specify not only where execution paths diverge – which is at branch instructions – but also where they converge back.

The better code is shown in Table 2.2, right. It has two new features. First, the program address where the threads converge is set prior to their divergence using instruction SSY. Second, the last instruction in each of the diverged paths is tagged with suffix ".S". ("S" stands for "synchronization".)

2.4 SIMT execution

Below we detail how the code in Table 2.2, right, is executed on a SIMT processor with two cases of input data. For the sake of example, we assume there are only four threads, all in one warp. In Case 1 the inputs in predicates P0 are set respectively to 0, 1, 1, 0 across the threads, which means that threads 1 and 2 take the if branch and threads 0 and 3 take the else branch. This case is detailed in Table 2.3. In Case 2 the predicates are set to 1, 1, 1, 1: all threads take the if branch. This case is detailed in Table 2.4.

Column "Threads" in the tables lists the logical execution order for each thread as defined by the program. Column "Warp" lists the physical execution order for the warp as executed on a SIMT processor. Also, this column lists the thread masks, which are explained below.

As SIMT architecture is a flavor of SIMD architecture, executing an instruction in one thread implies executing it also in all other threads of the same warp – unless some of them are disabled. A SIMT processor keeps track of which threads are currently disabled using a special bit mask called active mask. For example, executing a SIMD instruction when active mask is 0110 implies that the instruction is effective only for threads 1 and 2 in the warp. Active mask is a part of warp state, similarly to program counter. Below we refer to pair (*program counter*, *active mask*) as current execution state.

Instructions		Threads				Warp	
		0	1	2	3	Order	Mask
	SSY ENDIF	0	0	0	0	0	1111
@!P0	BRA ELSE	1	1	1	1	1	1111
IF:	IADD R1, R1, 1	2	2	2	2	2	1111
	IADD R2, R2, 2	3	3	3	3	3	1111
	IADD.S R3, R3, 3	4	4	4	4	4	1111
ELSE:	IMUL R4, R4, 4						
	IMUL R5, R5, 5						
	IMUL.S R6, R6, 6						
ENDIF:	(next instruction)	5	5	5	5	5	1111

Table 2.4: Threads don't diverge: `P0` = (1,1,1,1).

To keep track of the paths taken by different threads, SIMT processor employs another warp state called divergence stack. When a branch instruction diverges, one of the branches is taken by the warp immediately, and the other is taken later; the data necessary to take it later is put on the stack. This data is a similar pair of a program counter address and an active mask.

Additional entries are put on the stack by the SSY instructions. These entries include the address of the convergence point as specified in the instruction and the active mask at the time of executing the instruction, i.e. before the divergence. In both of our examples, this entry is (ENDIF, 1111).

Entries are popped from the stack upon executing the .S instructions. The taken entry is then assigned to the current execution state.

In Case 1, the branch instruction diverges into paths (IF, 0110) and (ELSE, 1001). Suppose the else branch is taken first. Then the current execution state is set to (ELSE, 1001) and pair (IF, 0110) is put on the stack. The warp state after executing the branch instruction then is:

```
Current execution state: (ELSE,  1001)
       Divergence stack: (IF,    0110)
                         (ENDIF, 1111)
```

As a result, the else branch is executed with mask 1001 first, then the if branch is executed with mask 0110, and then execution is transferred to label ENDIF with mask 1111; the transfers are at .S instructions. The effect is the same as if threads 1 and 2 took the if branch and threads 0 and 3 took the else branch. Note that different threads took different branches, but the warp took both.

In Case 2, the branch options are (IF, 1111) and (ELSE, 0000). The second pair has zero mask, which means that the branch is not diverging. The pair with zero mask is discarded and the other pair is set as the current execution state. The warp state after executing the branch instruction then is:

```
Current execution state: (IF,    1111)
       Divergence stack: (ENDIF, 1111)
```

Then, execution proceeds at the if branch with mask 1111, but only up to the .S instruction. After executing the .S instruction, the control is transferred to label ENDIF with the same mask. In effect all threads take the if branch, and the warp also takes only the if branch; instructions in the else branch are not executed.

This illustrates the basic operation of a SIMT processor. Further details can be found in NVIDIA patents such as Coon et al. [2008], Coon et al. [2009], Coon et al. [2010], and Coon et al. [2011].

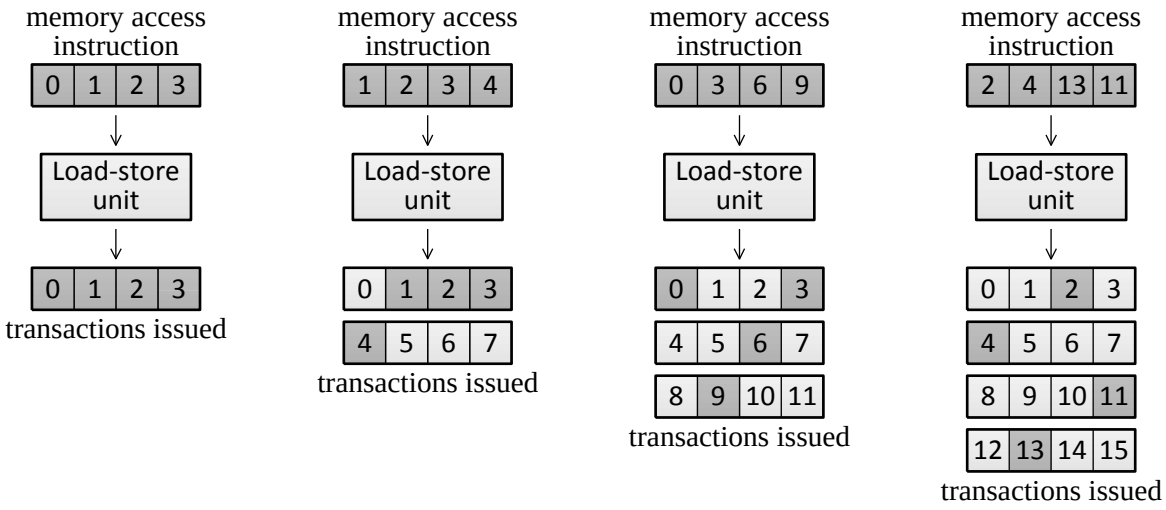


Figure 2.1: Coalescing memory accesses.

In particular, there is an additional detail needed to handle different types of control divergence, such as divergence at branch instructions, divergence at return instructions, and divergence at break instructions.

Thread divergence and SIMT processing are orthogonal to this work as they only determine in which order SIMD instructions are executed and with what masks.

2.5 Coalescing memory accesses

Similarly to independently taking branches, each GPU thread can also independently access memory. Given the SIMD width, this amounts to 32 independent memory accesses per instruction. Each access is typically to a four-byte data. The memory system, on the other hand, processes requests at a coarser granularity of 32 to 128 byte continuous and aligned blocks. Requests for such blocks are called memory transactions. When different thread accesses in the same SIMD instruction fit in same block, they may be "coalesced" and served with one transaction. Coalescing is done in load-store units.

Below we consider a few examples of memory coalescing; in all cases we assume there are no cache hits. For the purpose of illustration we assume there are only four threads per warp and only four words per transaction. Also, we assume that memory is word-addressed and each thread accesses a single word. Each warp access in this case can be described with a tuple (*a*, *b*, *c*, *d*), where *a*, *b*, *c*, and *d* are the memory addresses used in the respective thread accesses. Each transaction can be described with a similar tuple but under additional constraints: the addresses must be sequential and aligned as in (4*k*, 4*k*+1, 4*k*+2, 4*k*+3) for some integer *k*.

The simplest case of memory coalescing is when a warp access directly corresponds to a valid memory transaction, such as accesses (0,1,2,3) and (4,5,6,7). This is also the only case when accesses are coalesced on early CUDA-capable GPUs such as G80. This trivial case is shown in Figure 2.1, left.

When individual accesses in a SIMD instruction are similarly lined up but together are not aligned with transaction boundaries, an additional transaction is needed. For example, warp access (1,2,3,4) translates into two transactions: (0,1,2,3) and (4,5,6,7), as shown in Figure 2.1, center

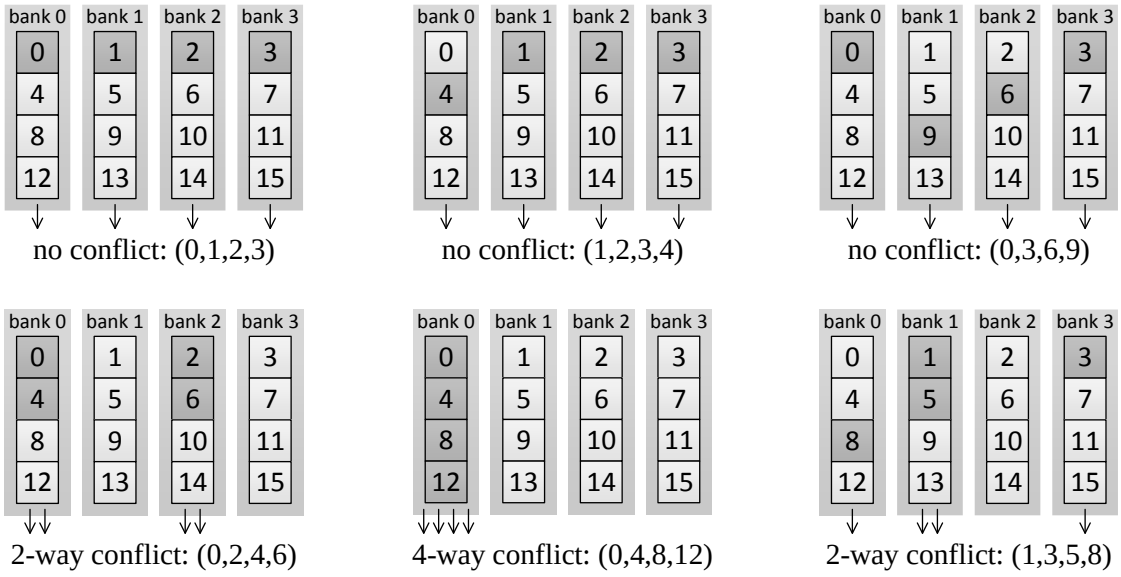


Figure 2.2: Shared memory bank conflicts.

left. The extra transaction poses an additional stress on the memory system and requires an additional time for processing.

If individual accesses are lined up but with a stride larger than one, proportionally more transactions are needed. For example, stride-2 access (0,2,4,6) requires two transactions: (0,1,2,3) and (4,5,6,7), and stride-3 access (0,3,6,9) requires three transactions: (0,1,2,3), (4,5,6,7), and (8,9,10,11). (As above, lack of alignment may further increase the transaction count.) An example with stride-3 access is shown in Figure 2.1, center right; two accesses are coalesced into a single transaction, but two others are not.

In the worst case, a separate transaction is needed for each individual thread access. This is the case, for example, with warp access (2,4,13,11). No coalescing is possible in this case, as shown in Figure 2.1, right.

Memory accesses on earlier GPUs may involve additional intricacies. For example, when different accesses in a warp fit in transaction boundaries but come in a wrong order, they are coalesced on GT200 and newer GPUs, but not on G80. In the G80-type coalescing rules, warp accesses (3,2,1,0) and (1,1,1,1) both allow no coalescing and are both translated to four identical transactions: (0,1,2,3), (0,1,2,3), (0,1,2,3), and (0,1,2,3).

GPUs in the GT200 generation have another feature: they may use transactions of different sizes when processing the same instruction. Suppose that in our case both 4-word and 2-word transactions are available. Then warp access (1,2,3,4) translates into transactions (0,1,2,3) and (4,5) instead of transactions (0,1,2,3) and (4,5,6,7). This may reduce the stress on the memory system.

Another feature of the G80 and the GT200 GPUs is that they process warp accesses one half-warp at a time. In terms of our example, warp access (1,2,3,4) is first split into half-warp accesses (1,2) and (3,4), which are then separately translated into transactions: (0,1,2,3) in the first case and (2,3) and (4,5) in the second.

This illustrates some of the basic options in coalescing memory accesses. The practical reference on coalescing rules is the CUDA C Programming Guide [NVIDIA 2010a; NVIDIA 2015].

Generation	Bank count	Bank width	Cycles per access
G80	16	32 bits	2
GT200	16	32 bits	2
Fermi	32	32 bits	2
Kepler	32	64 bits	1
Maxwell	32	32 bits	1

Table 2.5: Parameters of shared memory units.

2.6 Shared memory bank conflicts

GPUs have a special fast memory for sharing the current working set among threads executed at the same time. This fast memory is called shared memory. The "usual" memory, such as discussed above, is called global memory where it is necessary to differentiate. There are also local memory, constant memory and texture memory, which do not take a substantial part in this work.

Accesses to shared memory are processed differently. For example, they are not coalesced into larger blocks. Instead, the stored data is sliced into a number of banks, and each bank serves accesses independently, one access at a time. Ideally, different threads access different data in different banks or the same data in the same bank; in the latter case, the data is accessed once and then broadcast. Otherwise, some banks perform several data accesses, which takes an additional time; it is said, then, that the respective warp access causes bank conflicts.

For an illustration, assume there are, again, four threads, addressing is word-based, and each thread accesses a single word. Also, assume there are only four banks. Then, the stored data is distributed in the following manner: words 0, 4, 8, … are in bank 0, words 1, 5, 9, … are in bank 1, and so on (Figure 2.2).

Any stride-1 access in this case incurs no bank conflicts; a special alignment, other than by word boundary, is not necessary. For example, accesses (0,1,2,3) and (1,2,3,4) are both processed equally fast (Figure 2.2, top left and top center).

In contrast, stride-n accesses with even n – but not odd n – cause bank conflicts and require additional processing time. For example, warp access (0,2,4,6) is stride-2 and incurs a 2-way bank conflicts in banks 0 and 2: these two banks must serve all four accesses alone, two accesses per bank, as shown in Figure 2.2, bottom left. There are no bank conflicts if the access is stride-3, as in (0,3,6,9); this case is shown in Figure 2.2, top right. A stride-4 access causes a 4-way bank conflict; an example is (0,4,8,12), as shown in Figure 2.2, bottom center: all four thread accesses are served by bank 0. In practice, up to 32-way bank conflicts are possible.

Examples of a broadcast are warp accesses (1,1,1,1) and (1,1,1,2). They involve no bank conflicts: bank 1 serves many thread accesses, but fetches the requested data only once.

When different banks experience conflicts of different degrees, the largest degree determines the execution time. For example, access (1,3,5,8), shown in Figure 2.2, bottom right, incurs a 2-way bank conflict when accessing bank 1, and no conflicts when accessing banks 3 and 0. Execution time is then bound by the 2-way bank conflict.

Parameters of shared memory units, which process shared memory accesses, vary across GPU generations, as shown in Table 2.5. Earlier GPUs have 16 banks in each unit and process warp accesses one half-warp at a time (a half-warp is 16 threads). Newer GPUs have 32 banks per unit. Bank width is 32 bits on all but Kepler GPUs, which have 64-bit banks; these wider banks also imply more complicated conflict rules when accesses are 32-bit. (Bank width is how much data

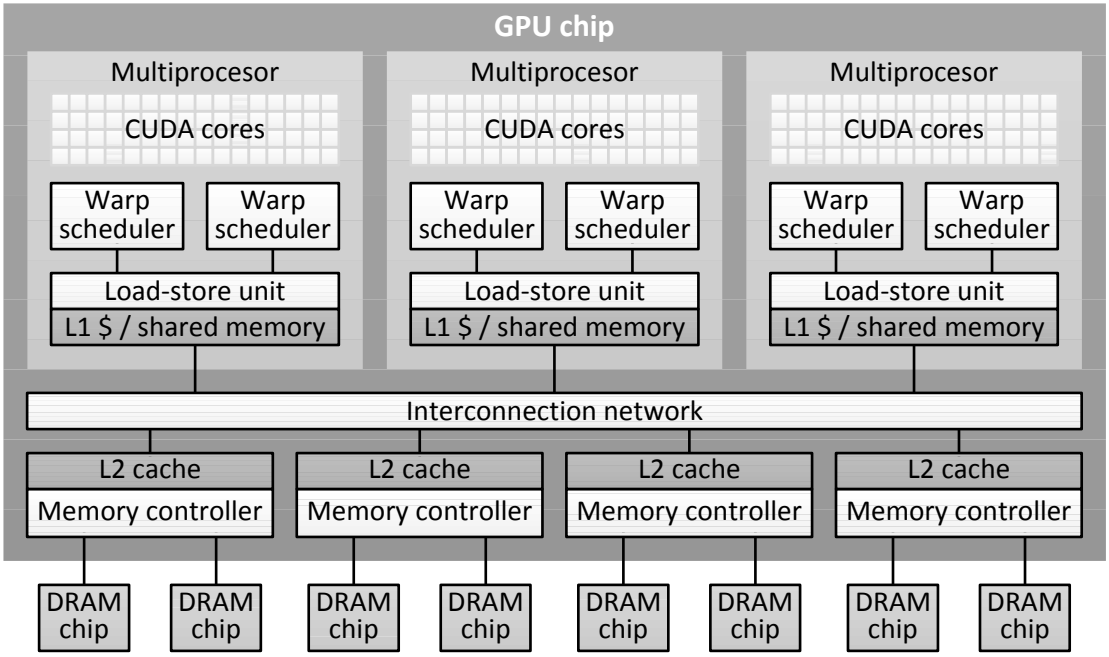


Figure 2.3: Some of the basic components of a GPU.

GPU	Number of SMs	Per multiprocessor		
		Warp schedulers	CUDA cores	SFU units
8800GTX	16	1	8	2
GTX280	30	1	8	2
GTX480	15	2	32	4
GTX680	8	4	192	32
GTX980	16	4	128	32

Table 2.6: The numbers of SMs and functional units per SM.

can be accessed at a time.) Each bank can finish one access every cycle on Kepler and newer GPUs, but only one access every two cycles on Fermi and earlier GPUs.

The reference on shared memory access performance and shared memory bank conflicts is the CUDA C programming guide [NVIDIA 2010a; NVIDIA 2015].

2.7 Other functional units and SMs

GPUs have a number of different functional units that participate in execution (Figure 2.3). We have already cited load-store units, which take part in processing global memory accesses, and shared memory units, which process shared memory accesses. There are also CUDA cores, SFU units, double precision units, warp schedulers, memory controllers and caches.

CUDA cores execute basic integer and single precision floating-point operations; each core is capable of finishing one such operation per cycle. 32 such operations are needed to fully execute an instruction.

Special Function Units, or SFUs, execute more complicated arithmetic operations, such as reciprocations, reciprocal square roots, and transcendental functions – also only in single precision. In most cases, each SFU unit is similarly capable of finishing one such operation per cycle.

GPU	Processor clock rate, GHz	Memory pin bandwidth, GB/s
8800GTX	1.35	86.4
GTX280	1.296	141.7
GTX480	1.4	177.4
GTX680	1.124*	192.3
GTX980	1.266*	224*

* *may vary during the execution, this is the maximum value*.

Table 2.7: Processor clock rates and memory pin bandwidths.

Double precision units execute double precision operations; we leave them out of scope of this work.

Warp schedulers are responsible for instruction issue. Each issue cycle each scheduler may issue one or two instructions selecting from a number of warps assigned to it; a different warp can be selected each time. This important mechanism is discussed in more detail shortly below.

A number of these units are assembled into one larger block – a tile – which is replicated across processor chip. The tile is called "streaming" multiprocessor, or SM. The number of units per SM varies with GPU generation (in some cases also within generation), and the number of SMs per GPU varies with the price segment the GPU is in. Table 2.6 lists these numbers for the GPUs used in this work; all of them are in the high-end segment. The number of shared memory units per SM is always one.

Additional processing units are found in the memory system. GPU memory system starts with load-store units, which are part of each SM. These units process memory access instructions and emit the necessary memory transactions. The data is stored in the DRAM devices, which are implemented as separate chips. Access to these devices is managed by memory controllers. Each transaction is delivered to the appropriate memory controller via the interconnection network. The controllers accumulate the incoming transactions and serve them in the same or a different order for a better DRAM operating efficiency. Reordering goals include, for example, improving row buffer hit rates and reducing the number of switches between reads and writes.

The memory system may also include caches: L1 cache in each multiprocessor and L2 cache in each memory controller. There are also texture caches, caches for address translation (TLBs), and, on some GPUs, read-only caches for global memory accesses.

2.8 Clock rates

For future reference, Table 2.7 lists processor clock rates and memory pin bandwidths for the GPUs used in this work. Memory pin bandwidth is the product of memory clock rate – which is different than processor clock rate –, the number of memory data pins, and the data rate per pin. The rate per pin is two bits per cycle for the double data rate (DDR) used in graphics DRAM.

Newer GPUs support dynamic frequency scaling, in which case some of these clock rates may vary during execution. For example, processor clock rate may be automatically decreased when there is little arithmetic but much memory activity, or when processor temperature exceeds a certain threshold. The numbers listed in the table for such GPUs represent the top of the allowed range. To leave dynamic frequency scaling out of scope, we ensure that only these rates are used in the reported measurements. The particular technique used to achieve this is explained in §5.4.

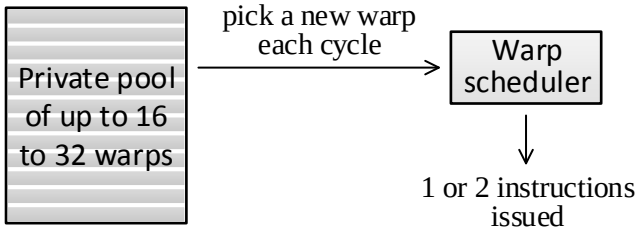


Figure 2.4: Each issue cycle a warp scheduler picks a different warp and issues one or two instructions from it if possible.

2.9 Fine-grained multithreading

Another distinctive feature of GPU architecture is multithreaded issue, also called fine-grained multithreading. Each instruction issue unit on GPU, i.e. each warp scheduler, may choose between a large number of warps when issuing an instruction and may issue from a different warp each cycle (Figure 2.4).

Multithreaded issue is used to satisfy the concurrency needs of pipelined execution: the execution is divided into a number of pipeline stages, where each instruction (or transaction, etc.) occupies one stage at a time and moves to the next stage when the next stage is available. A number of instructions are thus executed at the same time, each at a different stage. This improves instruction execution rate without improving execution times of individual instructions.

Two instructions can be executed at the same time only if they are independent, i.e. if output of one is not an input of another. One warp can supply a limited number of independent instructions at a time. More independent instructions are provided by executing a number of warps simultaneously.

Each warp scheduler is assigned a number of warps. All of these warps are current, i.e. the state of each is equally available for instruction issue. Context switches in terms of saving and restoring warp states are therefore not needed and don't occur in a regular execution. A similar mechanism with saving and restoring warp contexts, however, may still be used to implement advanced functionality, such as dynamic parallelism [NVIDIA 2012c] and debugging. The term context switching is also used for application contexts, such as when switching between using graphics and computational kernels; see, for example, whitepapers on the Fermi architecture [NVIDIA 2009; NVIDIA 2010b]. On all GPUs until the recent Pascal generation such switching does not involve saving warp contexts to DRAM [NVIDIA 2016].

The scheduler keeps track of which warps have an instruction ready for issue. If next instruction in a warp is not ready for issue, such as due to dependencies, the warp is stalled – instructions in the same warp are issued only in-order. One of the warps that are not stalled, if any, is selected each issue cycle and used to issue an instruction or two.

For example, at cycle 0 the scheduler may issue from warp 0, at cycle 1, from warp 1, etc. – not necessarily in a round-robin manner. A different warp is used independently of whether the last used warp is stalled for a long time, such as on memory access, or for a short time, such as on arithmetic operation. Moreover, there is evidence that schedulers on many GPUs cannot issue from the same warp again for a short time after each issue, even if the warp is not stalled on register dependency. We discuss this finer point in some additional detail in §3.7.

Table 2.8 lists some of the basic parameters of warp schedulers for different generations of GPU architecture. The maximum number of warps per scheduler is 16 to 32. Issue is done either each processor cycle, or once in two cycles – faster on newer GPUs. Dual-issue, i.e. the capability to

Generation	Warps per SM	Warps per scheduler	Issue rate, cycles	Issue width
G80	24	24	2	1
GT200	32	32	2	1
Fermi	48	24	2	1-2*
Kepler	64	16	1	2
Maxwell	64	16	1	2**

* *issue width is 1 on the GTX 480 used in this work.*
** *vendor's documentation is inconsistent.*

Table 2.8: Basic parameters of warp schedulers.

issue two independent instructions at a time, at the same warp scheduler, is available since the later versions of the Fermi architecture – the GPUs with "compute capability 2.1" [NVIDIA 2015]. It is not available on the Fermi GPU used in this work. Vendor's publications disagree on whether dual-issue is available on Maxwell GPUs: the respective whitepapers suggest it is [NVIDIA 2014a; NVIDIA 2014b], but the CUDA C programming guide suggests it is not [NVIDIA 2015; Ch. G.5.1]. Another reason to believe that it is available is that vendor's utility `cuobjdump` marks up instruction pairs in assembly codes for Maxwell GPUs.

It is not uncommon to assume that issue rate on G80 and GT200 processors is limited by one instruction per 4 cycles. However, this limit is effective only for CUDA core instructions, as bound by 8 CUDA cores per scheduler and SIMD width 32; the schedulers per se can issue twice as fast [Lindholm et al. 2008, p.45].

2.10 Thread blocks

As warps are executed asynchronously, an additional synchronization is needed to exchange data between warps using shared memory. To address this difficulty, GPUs provide a lightweight synchronization primitive: a barrier instruction.

Suppose that there are two warps: *A* and *B*, each writes its data to shared memory and then each reads the data written by the other warp. The read in warp *A*, then, must be processed after the write in warp *B*, which is not necessarily the case if the warps are executed independently. To ensure the intended access order, the reads and writes in each warp must be separated with a barrier instruction, as shown below:

```
warp A:          warp B:
write A data     write B data
barrier          barrier
read B data      read A data
```

When a barrier instruction is executed in one warp, the warp is stalled until a barrier instruction is executed also in another warp. This ensures that both reads are executed after both writes.

More generally, each warp is stalled until a barrier instruction is executed in every other warp of the same thread block, where thread block is a group of warps designated at the kernel launch time. All executed warps are divided into thread blocks; warps in the same thread block are executed on the same multiprocessor, access the same shared memory unit, and can be inexpensively synchronized. Different thread blocks are assigned to the same or different multiprocessors, but in either case access separate shared memory partitions.

Relative occupancy	Absolute occupancy, warps/SM				
	G80	GT200	Fermi	Kepler	Maxwell
100%	24	32	48	64	64
50%	12	16	24	32	32
25%	6	8	12	16	16
12.5%	3	4	6	8	8

Table 2.9: We use absolute occupancy in addition to the more commonly used relative occupancy.

2.11 Occupancy

Execution on a GPU starts with a kernel launch, where a GPU program (i.e. kernel) and its launch configuration are specified. The launch configuration includes the number of thread blocks to be executed, the number of warps per thread block, and the size of shared memory partition per thread block.

It is a common and recommended practice to launch many more thread blocks than can be executed at the same time. When all warps in one thread block complete, the next thread block in line is started in its place. The number of thread blocks executed at the same time depends on how much shared memory is allocated per thread block, and how many registers are used in each warp.

The number of warps executed at the same time is one of the key metrics characterizing execution on GPU; this metric is called occupancy in this text. Occupancy is also the number of warps executed at the same time divided by the maximum number of warps that can be executed at the same time. The second definition is prevalent elsewhere. We use occupancy in both meanings. The latter occupancy is quoted in percent and is, therefore, relative occupancy. The former occupancy is quoted in warps per SM, or warps per scheduler, and is absolute occupancy.

Table 2.9 lists a few examples of the correspondence between relative and absolute occupancies for different GPU generations. For example, occupancy 25% on a Maxwell GPU is the same as occupancies 16 warps/SM and 4 warps/scheduler; occupancy 25% on a G80 GPU is the same as occupancies 6 warps/SM and 6 warps/scheduler.

Chapter 3
Modeling latency hiding

The contributions of this chapter are the following. In §3.3 we point to an important oversight in prior work on GPU performance modeling, which is understanding the difference between occupancy and instruction concurrency. In §3.5 we suggest a simple performance modeling framework that doesn't have this oversight. In §3.9 we discuss what it means to have latency "hidden", correcting the view implied in vendors' programming guides. Finally, in §3.10 we show how to use Little's law to make quick intuitive judgements on GPU performance, such as on how the occupancy needed to attain a better throughput depends on various practical factors.

We start with introducing basic concepts, such as latency, throughput and Little's law, which are then developed into instruction latency, instruction throughput, warp latency, and warp throughput.

3.1 Latency, throughput and Little's law

In serial processing, the total execution time of a program can be broken up into a sum of execution times of individual instructions. In concurrent processing, in contrast, execution times of individual instructions may overlap and not sum to the total execution time. To address this difficulty, it is convenient to use other concepts than execution time, such as latency, throughput and concurrency.

Consider a generic concurrent activity, such as shown in Figure 3.1. It is comprised of a number of items, which can be instructions, memory transactions, warps, etc. Each item is defined with two numbers: its start time, and its end time. Separately given is the time interval during which the activity occurs, which is between cycle 0 and cycle 10 in this case. To characterize the entire process, we may use the following aggregate metrics.

The first metric is mean latency. This is the average of latencies of individual items, where latency of an individual item is the difference between its start time and its end time. In the figure, individual latencies are 1, 2, or 3 cycles, and mean latency is 2 cycles.

Another metric is throughput, or processing rate. It is defined as the number of items that fall within the given time interval divided by the duration of the interval. In this figure, it is 10 items over 10 cycles, or 1 item per cycle. We also often quote reciprocal throughput, which is 1 cycle per item in this case.

Concurrency is the number of items processed at the same time. It is defined for each particular moment of time and also as an average over the interval. In this example, concurrency varies between 1 and 3 items, and averages to 2 items.

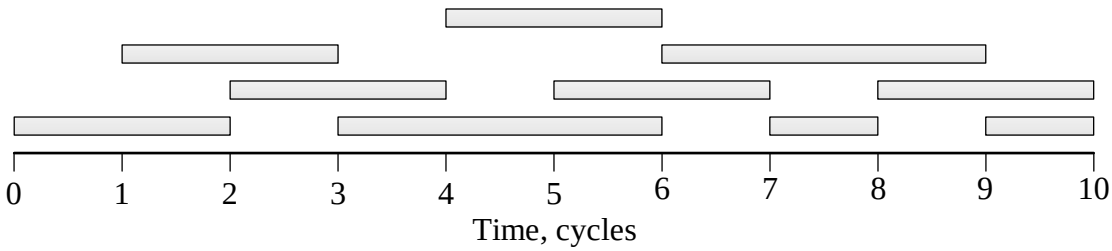


Figure 3.1: An example of a concurrent process.

Instruction type	Latency	Per multiprocessor	
		Peak Throughput	Needed Concurrency
Basic arithmetic	6 cycles	4 IPC	24 instructions
Memory access	368 cycles	0.081 IPC	30 instructions

Table 3.1: Latencies, peak throughputs and required concurrencies for basic instructions on the Maxwell GPU.

These metrics are related via Little's law (see, for example, Little [2011]):

$$mean\ concurrency = mean\ latency \cdot throughput.$$

For this relation to hold, it is important that all of these metrics are derived from the same set of start and end times, the same time interval is used to define both throughput and mean concurrency, and all items fall entirely within the interval. Little's law does not necessarily hold if some of the items do not fully fall within the interval – see Little and Graves [2008] for a discussion. But even in that case the relation may still apply, at least approximately, if the fraction of such items is small. Little's law is also used with infinite processes, for which case it was proven in Little [1961].

3.2 Instruction latency and throughput

Little's law is often used to estimate concurrency that is needed to attain a given throughput in executing any particular kind of instruction, such as arithmetic instructions or memory instructions (Table 3.1).

Consider a concurrent activity where items are instructions. Latency, throughput and concurrency in this case correspond to, respectively, instruction latency, instruction throughput and the number of instructions executed at the same time. Instruction latency is usually understood as register dependency latency. This is the part of the total instruction processing time that starts when an instruction is issued and ends when a subsequent register-dependent instruction can be issued. Register-dependent instructions are defined as instructions that use register output of a given instruction as input. This defines the start and end times for instruction execution, and thus also defines the respective throughput and concurrency metrics.

Little's law in this case reads, if reordering the terms:

$$instruction\ throughput = \frac{instructions\ in\ execution}{instruction\ latency}.$$

For example, the latency of basic arithmetic instructions on the Maxwell GPU is 6 cycles. Executing 1 such instruction at a time results in throughput of 1/6 instructions per cycle (IPC), executing 2 such instructions at a time, in throughput of 1/3 IPC, executing 3 such instructions at time, in throughput of 1/2 IPC, etc. We also may consider executing 0.5 such instructions at a time on average, such as if executing 1 such instruction at a time half of the time and 0, another half of the time. This would result in throughput equal 1/12 IPC. In all of these examples we assume that latency itself is constant and does not depend on concurrency.

Instruction throughput is limited by the capability of the hardware. For example, Maxwell GPUs have only 128 CUDA cores per SM and, as a result, can execute arithmetic instructions not faster than 4 IPC per SM – since 1 instruction does 32 arithmetic operations. This bound is called peak throughput. Instruction latency and the respective peak throughput are basic hardware parameters

that characterize instruction execution. They depend on instruction type, but may also depend on bank conflicts, memory coalescing and other factors.

It is easy to see that attaining peak throughput in this example requires executing 24 instructions per SM at the same time, where all instructions are implied to be the basic arithmetic instructions executed by CUDA cores. This result can be formally found by using Little's law: we multiply instruction latency, which is 6 cycles, by peak throughput, which is 4 IPC per SM, and get 24 instructions per SM. Note that by doing so we assume that this mean latency and this throughput are sustained at the same time.

A similar estimate can be also used for memory accesses [Bailey 1997]. Latency of memory access instructions on the Maxwell GPU is about 368 cycles – this is if accesses miss cache, are 32-bit, are fully coalesced and only one or a few such instructions are executed at the same time. For peak throughput it is common to use pin bandwidth, which is 224 GB/s on this processor. This rate corresponds to instruction throughput equal to 0.086 IPC per SM: to convert we divide by the clock rate (1.266 GHz), the number of SMs (16) and the number of bytes requested per instruction (128). Substituting this peak throughput and the above latency into Little's law suggests that 33 such memory access instructions must be executed at the same time per SM to attain the peak.

Complexity of the memory system creates a few extra difficulties. First, pin bandwidth is only an approximation of peak memory throughput – the best throughput that can be sustained in practice is noticeably less, such as by 3 to 20%, depending on GPU, according to our findings in Chapter 6. For this reason, we don't use pin bandwidths in this work and instead use peak throughputs found experimentally. On the Maxwell GPU, this throughput is found to be 211 GB/s, which corresponds to 0.081 IPC per SM. Substituting this number into the concurrency estimate above produces a new estimate equal 30 instructions per SM.

Another difficulty is with memory latency. As we show in Chapter 6, memory latency is not constant, but varies from one individual instruction to another, and its average, i.e. mean latency, tends to be larger when throughput is larger. Using unloaded latency and peak memory throughput in Little's law at the same time, therefore, does not correspond to a realistic execution scenario and serves only as an approximation. We keep our estimates approximate in this respect, assuming, for simplicity, that memory latency is constant relative to concurrency. This is a common practice.

Latency and peak throughput of memory access instructions depend on the memory access pattern. If accesses are not coalesced, instruction latency is larger due to issuing additional transactions, and peak instruction throughput is smaller due to transferring more data through the memory system.

3.3 Instruction concurrency vs occupancy

Throughput and concurrency estimates similar to the above are found in many prior GPU performance models – but with one important difference: the concurrency is understood, with some reservations, as the number of concurrently executed warps, not the number of concurrently executed instructions. This prior work includes CUDA C programming guide [NVIDIA 2015], and performance models by Hong and Kim [2009], Baghsorkhi et al. [2010], Sim et al. [2012] and Song et al. [2013].

There are at least two important reasons why the number of concurrently executed warps may be different than the number of concurrently executed instructions. These are instruction-level parallelism, which is well understood in prior work, and instruction mix, which is not.

Having instruction-level parallelism, or ILP, in a code means that the code permits executing more than one instruction at the same time from the same warp. In particular, this refers to register dependencies: two instructions may be executed at the same time only if register output of one

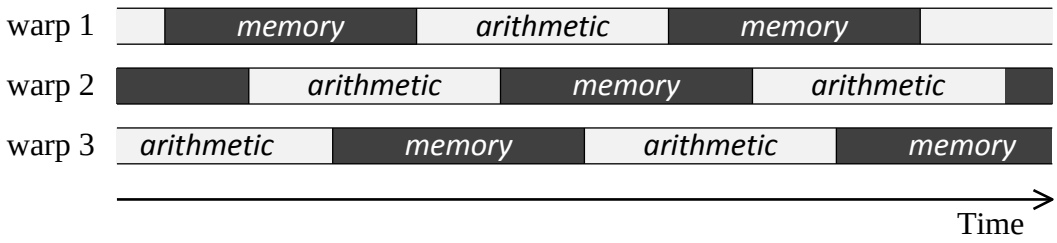


Figure 3.2: There are 3 concurrent warps, but only 1.5 concurrent memory access instructions on average.

instruction is not used as an input in the other instruction. Instruction-level parallelism is common in practice, and we consider a simple example that includes it – vector add kernel – later in this chapter.

ILP allows attaining the same instruction concurrency using fewer warps. For example, if on average 2 memory access instructions are executed in each warp at the same time, instruction concurrency equal to 30 memory instructions per SM may be attained by using 15 warps per SM. This effect is recognized in performance models by Sim et al. [2012] and Baghsorkhi et al. [2010], and is also noted in the CUDA C programming guide [NVIDIA 2015; Ch. 5.2.3]. Volkov [2010] presents a number of illustrations of this effect and its use in code optimization.

Another factor responsible for mapping between warps and instructions is instruction mix. It refers to the proportion in which different instruction types are present in the code. When several instruction types are present in the same code, which is the common case, executing an instruction of one type in a warp typically implies not executing instructions of other types in the same warp at the same time. In result, the number of concurrently executed instructions of any particular type may be smaller than the number of concurrently executed warps. This effect is overlooked in all previously published work that we know. This includes the occupancy estimate in Song et al. [2013], Section IV.E, a similar metric *MWP_peak_bw* in models by Hong and Kim [2009] and Sim et al. [2012], and throughput estimates in Baghsorkhi et al. [2010], Zhang and Owens [2011], and Sim et al. [2012]. A detailed study of prior performance models is presented later in this text.

Consider an example shown in Figure 3.2. It includes instructions of two types: arithmetic instructions and memory instructions. The instructions are present in such proportions that each warp has one arithmetic instruction in execution half of the time and one memory instruction in execution another half of the time. Thus, given 3 warps, the number of arithmetic instructions in execution is only 1.5 on average, and the number of memory instructions in execution is also only 1.5 on average. Attaining the concurrency equal 30 memory instructions in this case requires executing 60 warps at the same time – compared to 15 warps in the example with ILP. Even a stronger effect may be expected if kernel includes more than two instruction types, such as four. These may be, for example, CUDA core instructions, SFU instructions, global memory access instructions and shared memory access instructions – all executed in different functional units and having different latencies and peak throughputs.

To summarize, the number of warps needed to attain a peak throughput – unlike the needed number of instructions – depends not only on hardware parameters, such as instruction latencies and peak throughputs, but also on executed code, such as register dependencies and the instruction mix.

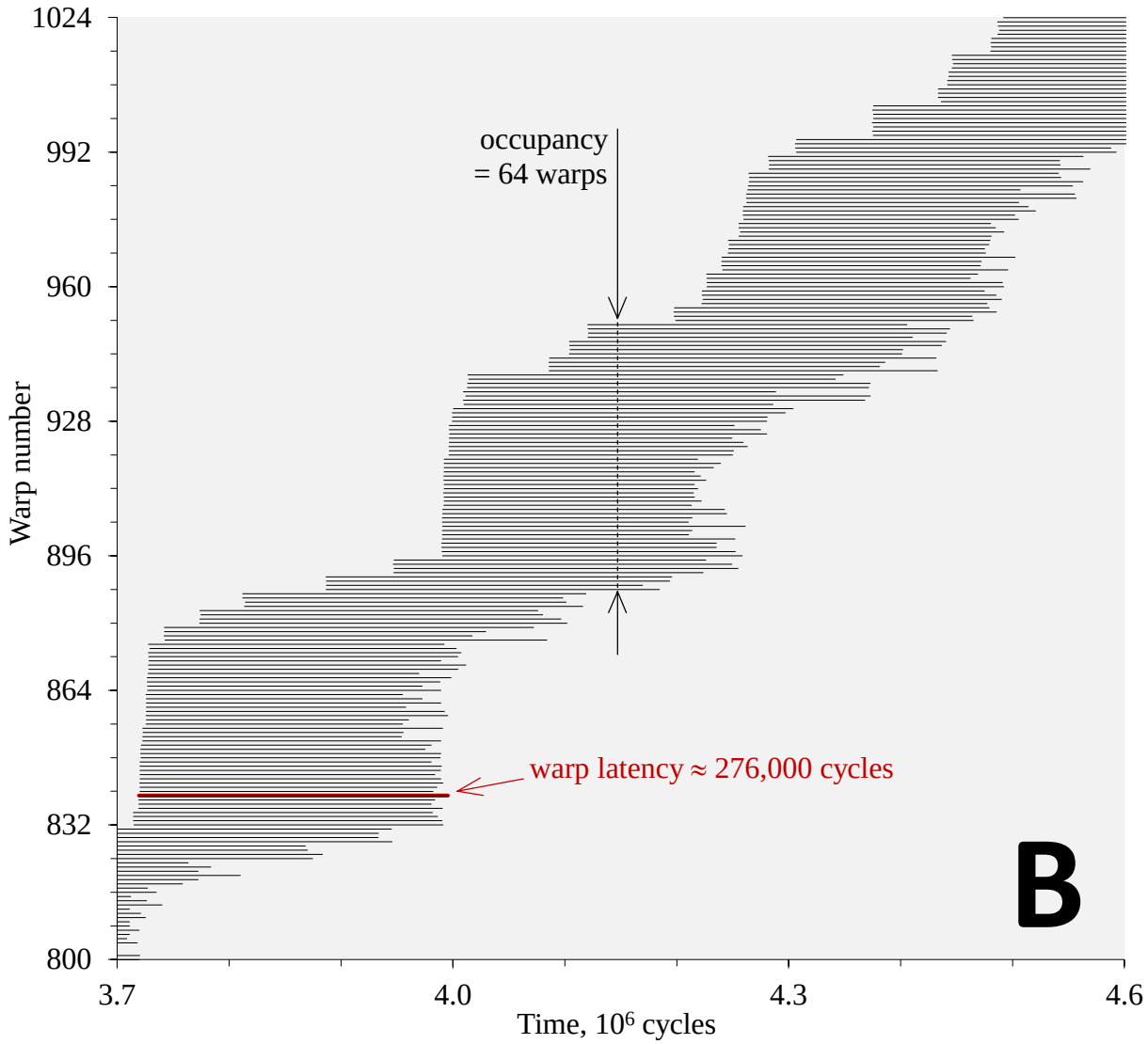


Figure 3.3: An example of a concurrent execution of warps. It corresponds to interval “B” in Figure 5.1.

3.4 Warp latency and throughput

To establish a connection between occupancy and kernel performance, we introduce a few intermediate metrics, which are directly connected with occupancy via Little’s law. These are warp latency and warp throughput.

Consider a concurrent process where items are warps. Each item in this case is defined by start time and termination time of the respective warp. The concurrency metric, then, corresponds to the number of warps executed at the same time, which is occupancy. The throughput metric corresponds to the total number of executed warps divided by the total execution time, which is warp throughput. The difference between start time and termination time of an individual warp corresponds to warp latency. The metrics are connected via Little’s law as:

$$\textit{mean occupancy} = \textit{mean warp latency} \cdot \textit{warp throughput}.$$

An example of such a concurrent process is shown in Figure 3.3. The shown data corresponds to an execution of a particular kernel on the Kepler GPU and is recorded as explained in §5.5. Individual warps are shown as horizontal lines. Length of a line corresponds to the respective warp latency, and the number of lines crossing a vertical section, to occupancy. Examples of these metrics are shown in the figure. Plotted are only a part of the total execution time, and only the warps that were executed on a particular SM.

According to the figure, latencies of individual warps may differ. This is despite executing the same number of instructions in each warp, which is the case here. A possible explanation for the difference is the variance in latencies of individual memory access instructions; we study this variance in more detail in §6.6.

Also noticeable in the figure is a variance in occupancy. This is a result of the differences in warp latencies: warps in the same thread block terminate at different times, but new warps may be assigned in their place only simultaneously. There are also other factors contributing to occupancy variance; we review some of them in Chapter 5. Occupancy variance is usually not taken into account in performance modeling, and we also consider it only to a limited degree.

With these metrics, we formulate the problem of performance modeling as finding warp throughput given occupancy and other parameters. Having found warp throughput, other integral performance metrics are also easily found. For example, the overall instruction throughput is found as the average number of instructions executed per warp multiplied by warp throughput, and the total execution time, as the total number of executed warps divided by warp throughput.

3.5 Two basic performance modes

We divide the problem of estimating warp throughput into two independent subproblems. First, we find a bound on warp throughput that is due to processor throughput limits. Second, we find a bound on warp latency that is due to processor latencies. The bounds are then merged using Little's law to produce the estimate. The two bounds correspond to two distinct performance modes.

We write the bound on warp throughput in a generic form:

$$\textit{warp throughput} \leq \textit{throughput bound}.$$

This bound is due to hardware limits on throughput: the number of CUDA cores and other functional units, peak issue throughput, peak throughput of memory system, etc. Unlike the bounds on instruction throughputs, this bound also depends on executed code. Its estimation is discussed in §3.6.

Another factor that may contribute to execution time is processor latencies. It imposes a lower bound on latency of each individual warp and, similarly, a lower bound on mean warp latency:

$$\textit{mean warp latency} \geq \textit{latency bound}.$$

For example, register dependency latencies limit how soon a dependent instruction can be issued from the same warp; another latency parameter limits how soon an independent instruction can be issued from the same warp. There are latencies specific for memory access instructions and a latency associated with replacing a retired thread block. We detail how they contribute to the bound on warp latency in §3.7.

We reduce the bound on mean warp latency to a bound on warp throughput by using Little's law. This produces:

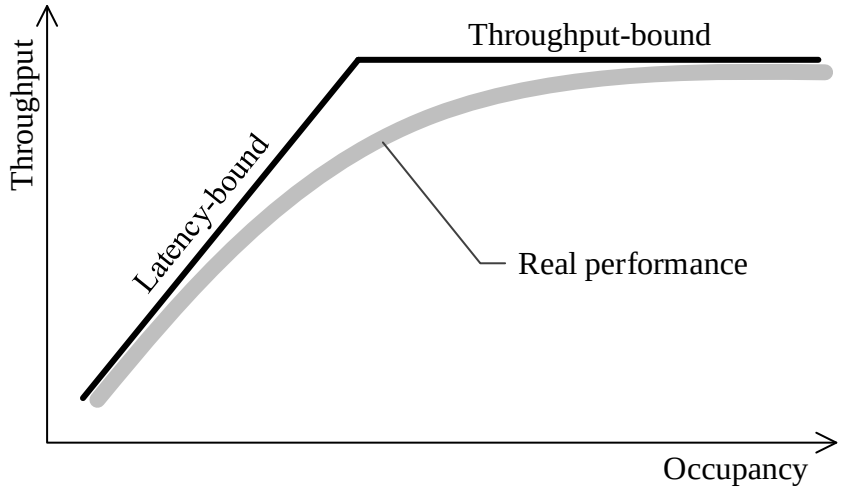


Figure 3.4: The basic performance pattern.

$$warp\ throughput \leq \frac{occupancy}{latency\ bound}\ .$$

Note that a similar reduction is not possible with a bound on individual warp latencies: the averaging is important.

Putting the two bounds together we get:

$$warp\ throughput\ \leq\ \min\left(\frac{occupancy}{latency\ bound}, throughput\ bound\right).$$

If there were no other factors contributing to kernel execution time, we would expect this combined bound to be an accurate estimate of warp throughput. But otherwise we expect it to be an approximate estimate. For the performance estimate we therefore use:

$$warp\ throughput\ \approx\ \min\left(\frac{occupancy}{latency\ bound}, throughput\ bound\right),$$

which summarizes our performance model at a high level.

This model has two modes shown in Figure 3.4. When occupancy is small, throughput is linear in occupancy and depends on processor latencies: this is latency-bound mode. When occupancy is large, throughput is fixed and depends on processor throughput limits: this is throughput-bound mode.

In practice, throughput usually doesn't grow linearly all the way to a maximum. This is due to interference between warps, which is not included in the bounds. For example, when two warps are executed at the same scheduler, both warps may have an instruction ready for issue, but only one can be selected at a time. Another warp, therefore, is delayed, which increases its latency and increases mean warp latency. For a similar reason, mean memory latency tends to increase when memory concurrency increases. Due to these additional delays, which are not part of the model, we might expect warp throughput to fall somewhat short of the estimate in both latency-bound

Instruction type	Per warp
CUDA core instructions	100
SFU instructions	5
Shared memory access	
-“- no conflicts	10
-“- 2-way conflict	10
DRAM access (no cache hits)	
-“- fully coalesced	5
-“- stride-2	5
Total	135

Table 3.2: A sample instruction mix.

Hardware throughput limit	Per SM
CUDA cores	128 cores
SFU units	32 units
Shared memory banks	32 banks
Memory system	10.4 B/cycle
Instruction issue	4 issues/cycle

Table 3.3: Sample hardware throughput limits.

mode and throughput-bound mode. This blurs the boundary between the modes so that one gradually translates into another, as schematically shown in the same figure. We call this “the gradual saturation effect”. We consider it the next level of accuracy and leave it out of scope until §6.4.

The model suggests that throughput may never decrease when occupancy increases. This is not always the case; it may decrease, for example, if cache hit rate decreases. This framework can be extended to cover such less common cases, but instead we choose to focus on more basic behaviors, which are yet not sufficiently well understood in prior work.

Similar bounds are discussed in classical work on concurrent systems such as Lazowska et al. [1984], Chapter 5.2, and Jain [1991], Chapter 33.6. Hong and Kim [2009] also separately consider what we call latency-bound and throughput-bound modes, but disregard arithmetic latency. This latency is important as we show later in Chapter 4. Sim et al. [2012] improves Hong and Kim model to include arithmetic latency and other factors, but the improved model has other limitations and is not similar to our approach. Huang et al. [2014] suggest a model that includes a similar latency-bound solution, but doesn’t include similar throughput bounds. Saavedra-Barrera et al. [1990], Chen and Aamodt [2009], and Huang et al. [2014] model the gradual saturation effect but not some of the other factors that are important on GPUs. The Roofline model [Williams et al. 2009] includes a number of throughput bounds but doesn’t provide a specific solution for the latency-bound case. Limiter theory [Shebanow 2008] considers similar throughput bounds, but does not consider latency bounds, at least not explicitly.

3.6 Throughput bound

To find the throughput bound we consider all hardware throughput limits one by one and convert each to a respective bound on warp throughput by taking into account instruction mix.

Used by:	CPI	Instructions per warp	Cycles total
Resource: CUDA cores			
CUDA core instruct.	0.25	100	25
Total:			25
Resource: SFU units			
SFU instructions	1	5	5
Total:			5
Resource: Shared memory banks			
No conflict access	1	10	10
2-way conflict	2	10	20
Total:			30
Resource: Memory system			
Fully coalesced	12.3	5	61.5
Stride-2 access	24.6	5	123
Total (the tightest bound):			184.5
Resource: Warp scheduler			
Single issue	0.25	125	31.25
Dual-issue	0.25	5	1.25
Reissue	0.25	15	3.75
Total:			36.25

Table 3.4: A worksheet for finding throughput bound.

Consider an example detailed in Tables 3.2, 3.3 and 3.4. Table 3.2 details instruction mix of a hypothetical kernel; the numbers are averages across executed warps. Table 3.3 details hardware throughput limits; they correspond to the Maxwell GPU. Table 3.4 details the resulting bounds on warp throughput and the involved cycle-counting.

The first throughput limit we consider is due to the limited number of CUDA cores. There are 128 CUDA cores per SM, which limits throughput of CUDA core instructions to 4 IPC per SM, or 0.25 cycles per instruction at each SM. The kernel has 100 such instructions per warp, which therefore limits warp throughput to 25 cycles per warp at each SM.

Similarly, given 32 SFU units per SM and 5 SFU instructions per warp, the bound on instruction throughput is 1 cycle per SFU instruction, and the respective bound on warp throughput is 5 cycles per warp at each SM.

The calculations are slightly more complicated if instruction throughput limit depends on other factors, such as memory access pattern. Consider the bound due to the limited number of shared memory banks. There are 32 of them per SM. We model them as a single resource that is exclusively used for 1 cycle by each shared memory access instruction with no bank conflicts, and for 2 cycles by each shared memory access instruction with 2-way bank conflicts. There are 10 instructions of both kinds per warp in this kernel, which sum up to 30 cycles of exclusive use. The respective throughput bound, therefore, is 30 cycles per warp at each SM.

Global memory accesses, both coalesced and non-coalesced, are considered in a similar fashion. It is given that peak throughput of memory system is 10.4 B/cycle per SM. We model it as a single

Issue cycle	Latency	Program code
0	301	`LD R1, [R1]`
301	24	`LDS R1, [R1]`
325	9	`FADD R1, R1, R2`
334	9	`MUFU.RSQ R1, R1`
343	–	`ST [R3], R1`

Listing 3.1: Register dependency latencies, the Kepler GPU.

Cycle	Instruction
0	`FADD R1, R1, R1`
6	`FADD R2, R2, R2`
12	`FADD R3, R3, R3`
18	`FADD R4, R4, R4`
24	`FADD R5, R5, R5`
30	`FADD R6, R6, R6`
36	`FADD R7, R7, R7`

Cycle	Instruction
0	`ST [R1], R1`
34	`ST [R2], R2`
68	`ST [R3], R3`
102	`ST [R4], R4`
136	`ST [R5], R5`
170	`ST [R6], R6`
204	`ST [R7], R7`

Cycle	Instruction
0	`LD R1, [R1]`
6	`LD R2, [R2]`
12	`LD R3, [R3]`
38	`LD R4, [R4]`
44	`LD R5, [R5]`
526	`LD R6, [R6]`
534	`LD R7, [R7]`

Listing 3.2: ILP latencies on the Fermi GPU.

resource independently available at each SM that is exclusively used for 1/10.4 cycles when transferring each byte. Each fully coalesced access instruction in this case, assuming cache misses, involves transferring 128 bytes and thus exclusively uses this resource for 12.3 cycles. Each memory access instruction that has stride-2 access pattern involves transferring twice as much data and therefore exclusively uses this resource for 24.6 cycles. As there are 5 instructions of each kind per warp, the total resource use is 184.5 cycles. The throughput bound therefore is also 184.5 cycles per warp at each SM.

The last throughput bound we consider is due to instruction issue. It involves such additional details as dual-issue and reissue. For the sake of example, we assume that each SFU instruction is dual-issued with a CUDA core instruction, and all instructions that involve non-coalesced accesses or bank conflicts are reissued once. Then, for 135 executed instructions per warp we have 5 dual issues, 125 single issues, and 15 reissues, which constitute 145 issue events in total. Given 4 warp schedulers per SM, the respective throughput limit is 0.25 cycles per issue at each SM, and the resulting bound on warp throughput is 36.25 cycles per warp at each SM.

The tightest of these bounds is 184.5 cycles per warp at each SM, or 0.00542 warps per cycle per SM. This is the best bound on warp throughput given the data. The more such hardware throughput limits we consider, and in more detail, the more accurate a bound we may construct.

3.7 Latency bound

A lower bound on mean warp latency is found by averaging lower bounds on individual warp latencies. A lower bound on each individual warp latency is found by simulating instruction issue in the respective warp assuming no throughput limits and no inter-warp interference. This allows finding instruction issue times in an inexpensive manner, using only processor latencies and executed code. Some interference between warps still has to be taken into account to model barrier synchronization.

The most widely recognized processor latency is register dependency latency; we discussed it before. If an instruction is issued at cycle X, and its register dependency latency is Y cycles, then its register output is ready at cycle $X + Y$. An instruction that uses this output cannot be issued

Warp 1		Warp 2	
Cycle	Instruction	Cycle	Instruction
0	`@P0 BRA label`	0	`@P0 BRA label`
28	`LD R1, [R1]`		
540	`FADD R1, R1, R2`		
	`label:`		`label:`
546	`BAR.RED.POPC`	32	`BAR.RED.POPC`
580	`FADD R1, R1, R2`	580	`FADD R1, R1, R2`

Listing 3.3: Latency of barrier synchronization.

sooner than then. An example is shown in Listing 3.1. Here, all instructions are back-to-back dependent. Issue time of the first instruction is cycle 0. Issue time of each following instruction is the sum of register dependency latencies of all instructions before it.

Another, less well known latency, is the latency to issue the next independent instruction from the same warp. We call it ILP latency for instruction-level parallelism. If an instruction is issued at cycle X, and ILP latency is Z cycles, next instruction from the same warp can be issued not sooner than at cycle $X + Z$. This is even if the next instruction does not depend on the recently issued instructions. An example is shown in Listing 3.2, left. This is a sequence of CUDA core instructions as executed on the Fermi GPU. All instructions are independent and issued 6 cycles after one another, where 6 cycles is the ILP latency. We find nontrivial ILP latencies on all five GPU architectures used in this study except Maxwell. Similar latency on a GT200 GPU was earlier noted by Lai and Seznec [2012].

Global memory access instructions may incur a larger ILP latency. For example, store instructions on the Fermi GPU, according to our study, cannot be issued more often than once in 34 cycles, as shown in Listing 3.2, center. This is if one warp is executed at a time, and all accesses are fully coalesced. ILP latencies for load instructions may even be larger and have a complicated structure, as shown in the same listing, right. The first five instructions in the example are issued 6 cycles after one another, except that the fourth instruction is issued 26 cycles after the third. What is more notable, is that the sixth instruction is issued 482 cycles after the fifth. This delay is about as long as memory latency itself, which equals approximately 513 cycles on this GPU (as we find in Chapter 6, Table 6.2). The delay may be due to a limitation on the number of concurrently processed memory loads allowed for each individual warp. This limitation was studied earlier by Nugteren et al. [2014].

Similar long sequences of independent instructions are common in highly optimized kernels such as matrix multiply [Volkov and Demmel 2008; Gray 2014], FFT [Volkov and Kazian 2008] and radix sort [Merrill and Grimshaw 2010]. They are also not unusual in other codes. For example, `lbm` kernel in the Parboil benchmarking suite [Stratton et al. 2012] includes a sequence of 20 independent loads and a sequence of 19 stores.

Another potential contribution to minimum warp latency is due to barrier synchronization. When a barrier instruction is executed in one warp, the warp is stalled until all other warps in the same thread block also execute a barrier instruction. The stall times can easily be found if issue times of the barrier instructions are known. To find these issue times we schedule all instructions in the participating warps up to the barrier instruction, inclusively. In doing so we assume that each warp starts at cycle 0 and is executed alone. Then we take the largest of these times, add pipeline latency of barrier instruction if known, and use the result as issue time for the next instruction in each warp.

```
__global__ void add(float *a, float *b, float *c)
{
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   c[i] = a[i] + b[i];
}
```

Listing 3.4: Addition of vector elements written in CUDA.

An example for the Fermi GPU is shown in Listing 3.3. There are two warps. One warp takes a longer path that includes an additional stall on memory access. The warps are synchronized at a barrier instruction, which is denoted as `BAR.RED.POPC`. Both warps start at the same time but arrive to the barrier instruction at different times: cycle 546 in one case and cycle 32 in another. Assuming that pipeline latency of barrier instruction is 34 cycles, issue time of the next instruction in both warps is cycle 580.

Also used in the example is the latency of a branch instruction. We find it equals 32 cycles if the branch is taken and 28 cycles if it is not. These and other latencies for the Fermi GPU are found by timing similar instruction sequences using clock register on the GPU. The instruction sequences were created using a custom assembler similar to `asfermi`[2]. There are other similar latencies, which may be used to refine the bound.

As discussed earlier, terminated warps are not replaced immediately, which results in occupancy variance. We can compensate for this effect by considering the time it takes to replace a warp as a part of the warp latency itself. There are two such delays that can be easily taken into account. The first is the time until other warps in the same thread block also terminate. This may be taken into account by rounding up each warp latency to the largest warp latency in the same thread block. The second is the time to assign a new thread block after previous thread block terminates. This latency can be found by timing execution of an empty kernel such as

```
__global__ void empty() { }
```

when it is launched with a large number of thread blocks. We find this latency equal to 150 to 300 cycles, depending on GPU. This number is then added to the latency of each individual warp.

3.8 An example

Consider an application of the presented methodology to performance estimation of a simple kernel such as vector add. The kernel is shown in Listing 3.4. It is written in CUDA; for a quick but sufficient introduction to CUDA see Nickolls et al. [2008]. CUDA code resembles C code but includes a few additional features. For example, the first line in the kernel computes global thread index using thread index in a thread block and index of the thread block.

We consider execution of this kernel on the Kepler GPU. We choose this, less recent GPU because this results in a slightly shorter, by two instructions, assembly code than if using the newer Maxwell GPU and thus allows keeping the following presentation a little less cumbersome.

The respective assembly code is shown in Listing 3.5. The code is easy to understand. The first instruction is redundant; according to Greg Smith, of NVIDIA, it sets the stack pointer[3]. The next two instructions read thread index and thread block index from special registers. The indices are then combined into a global thread index using integer multiply-add instruction, `IMAD`. Instructions

[2] http://code.google.com/p/asfermi/

[3] http://stackoverflow.com/questions/14483077/how-cuda-constant-memory-allocation-works

```
MOV R1, c[0x0][0x44]
S2R R3, SR_TID.X
S2R R0, SR_CTAID.X
IMAD R2, R0, c[0x0][0x28], R3
ISCADD R3, R2, c[0x0][0x140], 0x2
ISCADD R0, R2, c[0x0][0x144], 0x2
LD R3, [R3]
LD R0, [R0]
ISCADD R2, R2, c[0x0][0x148], 0x2
FADD R3, R3, R0
ST [R2], R3
EXIT
```

Listing 3.5: Assembly code for the vector add kernel.

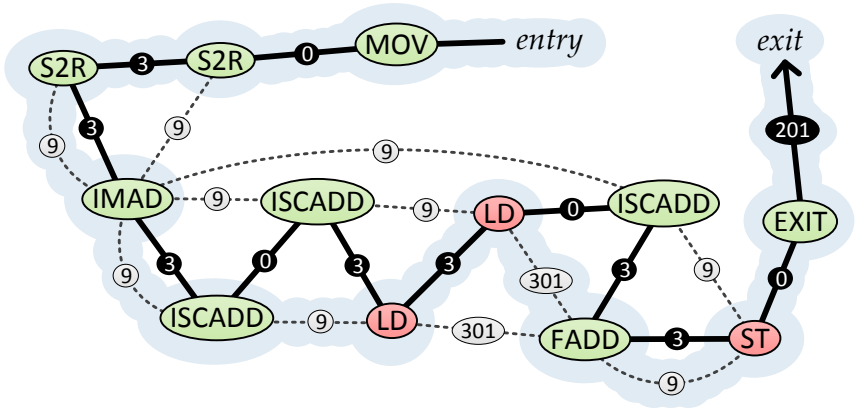


Figure 3.5: Program execution graph for the kernel.

`ISCADD` are used to compute memory addresses in the arrays, and instructions `LD` and `ST` are used to access the arrays. `FADD` is a floating-point add instruction. Operands such as `c[0x0][0x44]` refer to the data stored in constant memory, such as kernel arguments and thread block size. Operands `R0`, `R1`, etc. are registers. Bars on the left denote the dual-issued instruction pairs. The assumed pairing rule is explained below.

To find a bound on individual warp latencies we represent all related latency constraints with a graph. The graph is shown in Figure 3.5. The nodes correspond to executed instructions, and the edges, to processor latencies. The graph is directed. Each edge points down the execution order and is annotated with the respective latency. Thick edges connect instructions in program order and represent ILP latencies and the warp termination latency. Dashed edges correspond to register dependency latencies, for which we use the numbers found in Chapter 6. These are 301 cycles for memory loads and 9 cycles for floating-point adds. All other instructions that are not memory accesses are assumed to be executed using CUDA cores and have similar 9 cycle latency. ILP latency is separately found to be 3 cycles. Dual-issue is modeled with zero ILP latency. We assume that load instructions are not paired with other load instructions, which leaves four dual-issue opportunities in the code as shown in the listing and the figure. Latency to replace a terminated thread block is found to be 201 cycles. This graph is similar to the work flow graph introduced in Baghsorkhi et al. [2010] but with substantially different edge assignments.

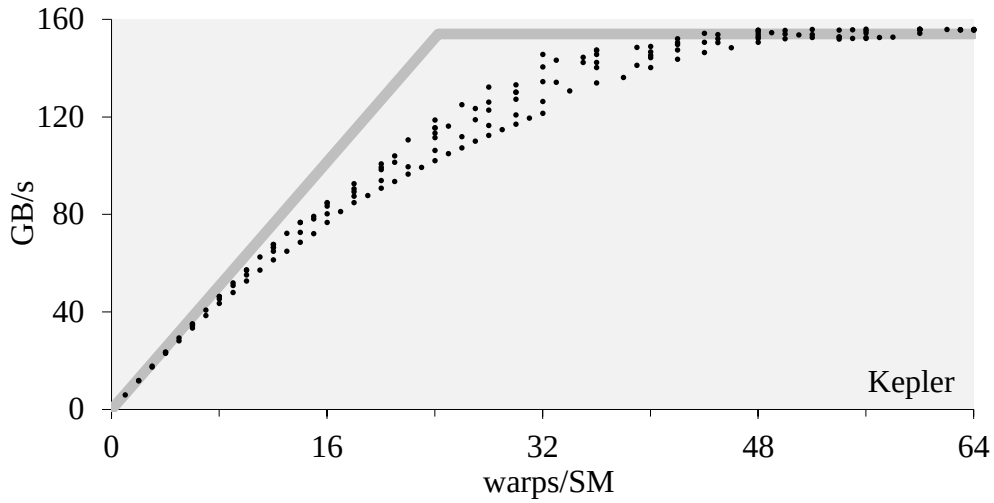


Figure 3.6: Comparison with experiment.

As edges represent latency constraints, length of a longest path in the graph corresponds to a lower bound on warp latency. The longest path in this graph is 544 cycles long; it is highlighted in the figure. Since all warps are similar, this is both a bound on individual warp latency and a bound on mean warp latency. Thus:

$$\textit{mean warp latency} \geq 544 \text{ cycles}.$$

For a bound on warp throughput we consider three bottlenecks: instruction issue, CUDA cores, and memory system. Given 12 instructions and 4 dual-issues, there are 8 issues per warp. Peak instruction issue rate is 1 issue per cycle per warp scheduler. The bound on warp throughput is thus 8 cycles per warp at each scheduler. Given 4 warp schedulers per SM, this is 0.5 warps / (cycle · SM).

Out of these 8 issue events, 5 include one CUDA core instruction and 2 include two CUDA core instructions. Given 48 CUDA cores per scheduler, at least one CUDA core instruction can be finished every cycle at each scheduler, and two can be finished every second cycle. Therefore, finishing all CUDA core instructions in this kernel, when executing many warps, requires at least 7 cycles per warp at each warp scheduler. This bound is less tight than the bound above.

Peak throughput of the memory system is found in Chapter 6: it is 154 GB/s or 17.1 B / (cycle · SM). This is if all accesses are reads and miss cache. The amount of data transferred by the memory system is 3×128 B / warp, and, therefore, warp throughput is bound by 0.0445 warps / (cycle · SM). This is if we don't differentiate read traffic and write traffic.

The tightest bound is the one due to the memory system. So:

$$\textit{warp throughput} \leq 0.0445 \text{ warps / (cycle} \cdot \text{SM)}.$$

Putting the two bounds together we get the following estimate:

$$\textit{warp throughput} \approx \min(\, n \,/\, 544,\, 0.0445 \,),$$

where *warp throughput* is in warps per cycle per SM, and n is occupancy in warps per SM.

It is convenient to express this result as memory throughput in GB/s. In order to do so we multiply it by the number of bytes transferred per warp (3 × 128 B), the number of SMs (8) and the clock rate (1.124 GHz). The result is:

$$\textit{memory throughput} \approx \min(\, n \cdot 6.35 \text{ GB/s}, 154 \text{ GB/s} \,).$$

It is plotted in Figure 3.6 as a thick grey line. Also plotted, as dots, are the throughputs attained in experiment when executing this kernel on large arrays. Multiple dots at each occupancy correspond to attaining the same occupancy with different thread block sizes; details of the experimental setup are given in Chapter 5. The figure shows that the model is highly accurate at both large and small occupancies but is less accurate otherwise due to the gradual saturation effect.

3.9 Latency hiding

The ability to attain a better throughput by executing many warps at the same time is often called "latency hiding". However, the use of this term requires a clarification, in which case we can use the above example as an illustration.

The widely used introductory materials on GPU programming, such as CUDA C Programming Guide [NVIDIA 2015; Ch. 5.2.3] and similar manuals for AMD GPUs [AMD 2015; Ch. 2.6.1], suggest considering latency "hidden" when attaining full utilization of functional units on a multiprocessor if on NVIDIA GPUs, and when keeping compute units busy if on AMD GPUs. This is in line with how latency hiding – also called latency tolerance – is understood in classical work, such as on coarse-grained multithreaded processors and prefetching: its purpose was to maximize processor utilization by avoiding stalls on memory accesses. We briefly review this prior work later, in §4.9.

However, this understanding of latency hiding leads to surprising results if applied to GPUs. For example, we find that in this case latency is not hidden in the example above and, moreover, cannot be hidden even theoretically. Indeed, the best attained throughput as per Figure 3.6 is 156 GB/s, which corresponds to the overall instruction throughput of 0.54 IPC/SM, which is a small fraction of both peak issue throughput (4 to 8 IPC/SM), and peak arithmetic throughput (6 IPC/SM). Functional units on multiprocessors are thus poorly utilized, and, in the above understanding, latency is not hidden. Yet, execution times at large occupancies are as small as if latencies were nil: the attained memory throughput equals peak memory throughput and therefore would not improve even if processor latencies were smaller. In this, more intuitive sense, latencies are hidden.

It is therefore useful to differentiate hiding latency and avoiding stalls. Stalls in instruction issue can be roughly divided into two types: latency stalls and bandwidth stalls. Latency stalls are the stall cycles that would be eliminated if some of the processor latencies were shorter. Bandwidth stalls are the stall cycles that would be eliminated if some of the processor throughput limits were looser. Latency hiding techniques help avoiding only latency stalls. A similar classification is suggested in Burger et al. [1996], except in application to memory stalls only. The two types of stall cycles correspond to the two execution modes discussed in §3.5: latency-bound mode and throughput-bound mode. Hiding latency, thus, means attaining throughput-bound mode.

To summarize, we understand latency hiding as synonymous to attaining any hardware throughput limit. This may be a limit due to a full utilization of functional units on a multiprocessor, or due to a full utilization of processing units elsewhere, such as in memory system. We use this understanding later, in Chapter 4.

```
__global__ void permute( int *a, int *b, int *c )
 {
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   a[i] = b[c[i]];
 }
```

Listing 3.6: Permutation of array elements in CUDA.

3.10 Back-of-the-envelope analysis

An important property of each GPU kernel is the occupancy needed to minimize its execution time or, what is the same, to attain a maximum throughput. In our model it is found as the product of a lower bound on mean warp latency and an upper bound on warp throughput. For short, below we also refer to these bounds as simply warp latency and warp throughput. Thus we can write:

$$\textit{needed occupancy} \approx \textit{warp latency} \cdot \textit{warp throughput.}$$

This is similar to how the needed instruction concurrency equals the product of instruction latency and peak instruction throughput, and is also a consequence of Little's law.

This simple relation, at least in some cases, provides an intuitive aid for understanding how the needed occupancy is affected by changes in executed code or input data. Below we consider a few examples of such back-of-the-envelope analyses.

Coalesced versus non-coalesced accesses. In the first example we consider how needed occupancy may depend on memory access pattern used in a kernel.

Consider the kernel shown in Listing 3.6. It performs a permutation of the contents of an array as described by assignment $a[i] := b[c[i]]$, where $i = 0, \ldots, m-1$ and m is the size of each array. Of the given three arrays, two are always accessed in a coalesced manner, and one, array b, is accessed in either a coalesced or a non-coalesced manner, depending on the contents of array c.

We compare the occupancies needed in the following two cases. In the first, the base case, the permutation is trivial as defined by $c[i] = i$ for all i. All accesses to array b are then fully coalesced. In the second case, the permutation is random with repetition: each $c[i]$ is set to a random number between 0 and m–1 so that all or virtually all accesses to array b are fully diverging.

What we show below is that the needed occupancy can be easily guessed to be substantially smaller in the second case, because the drop in warp throughput caused by memory divergence is much more severe than the associated increase in warp latency.

It is well known that memory throughput dramatically deteriorates when memory accesses are diverging. In part this is due to the increase in data traffic. Suppose that all thread accesses are 4 bytes wide. Then a fully coalesced access instruction requires transferring 128 bytes. A fully diverging access instruction, on the other hand, results in transferring 32 transactions with 32 or 128 bytes per transaction [NVIDIA 2010a; NVIDIA 2015]. The respective increase in data traffic, thus, is 8x or 32x. Similarly, the number of bytes transferred in the permutation kernel is 3×128 per warp in the base case and 2×128+32×32 or 2×128+32×128 in the second case; the respective increases in the overall data traffic are 3.3x are 11x. Warp throughput may be expected to drop by a similar factor.

In practice, however, memory divergence often causes a larger throughput deterioration than can be explained by data traffic alone. In Chapter 6 we find that on the five GPUs used in this study peak instruction throughput in fully diverging accesses is smaller than peak instruction throughput in fully coalesced accesses by either a factor of 28 to 33 or a factor of 56. (These numbers

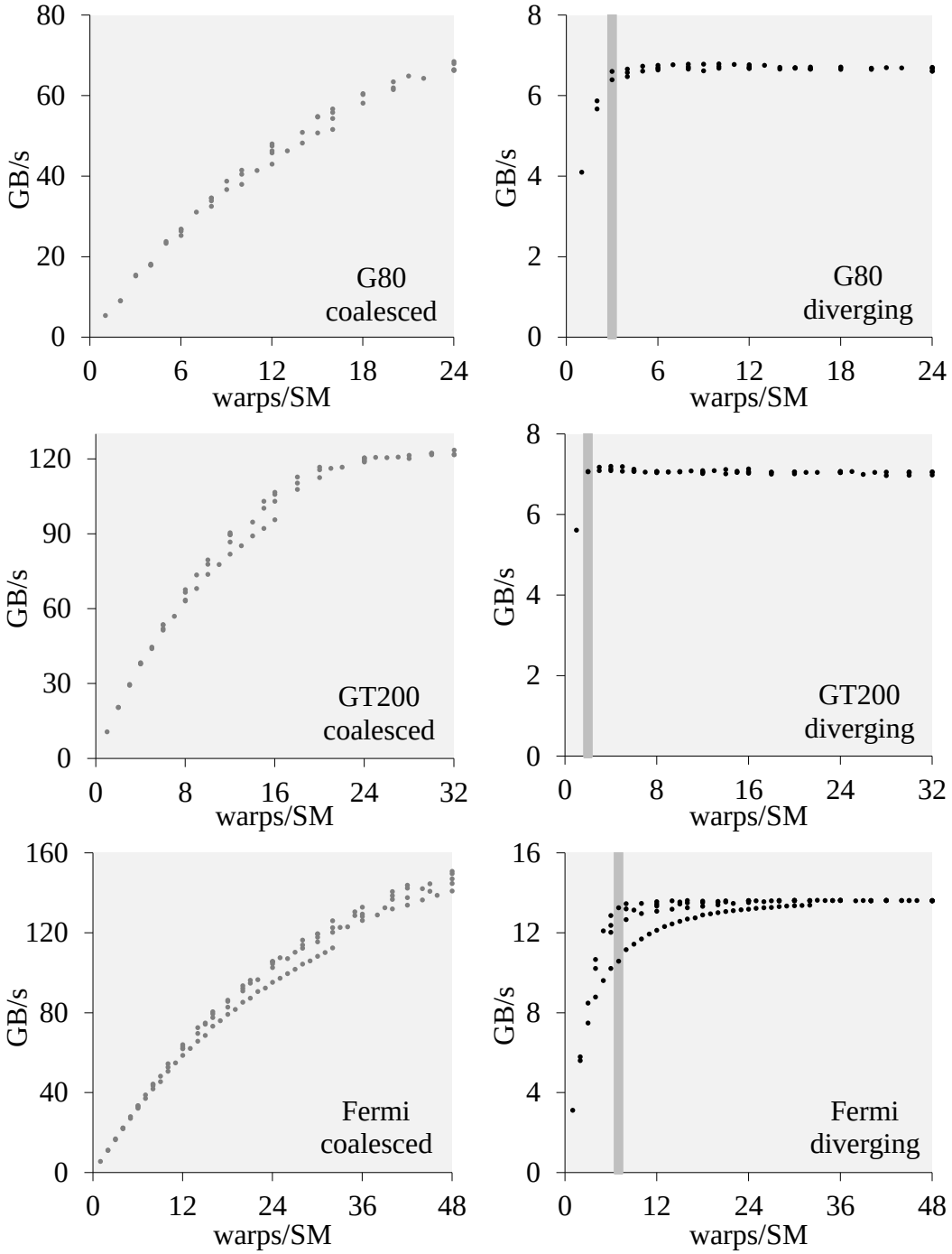


Figure 3.7: Performance of the permutation kernel.

correspond to the particular array sizes used here and in §6.7.) This is substantially larger than the factor of 8 expected with 32-byte transactions and is similar to as if all transactions were not shorter than about 128 bytes. Assuming that this observation also applies in execution of the permutation kernel, the deterioration of warp throughput in it must be around 11x or larger on each of the five GPUs.

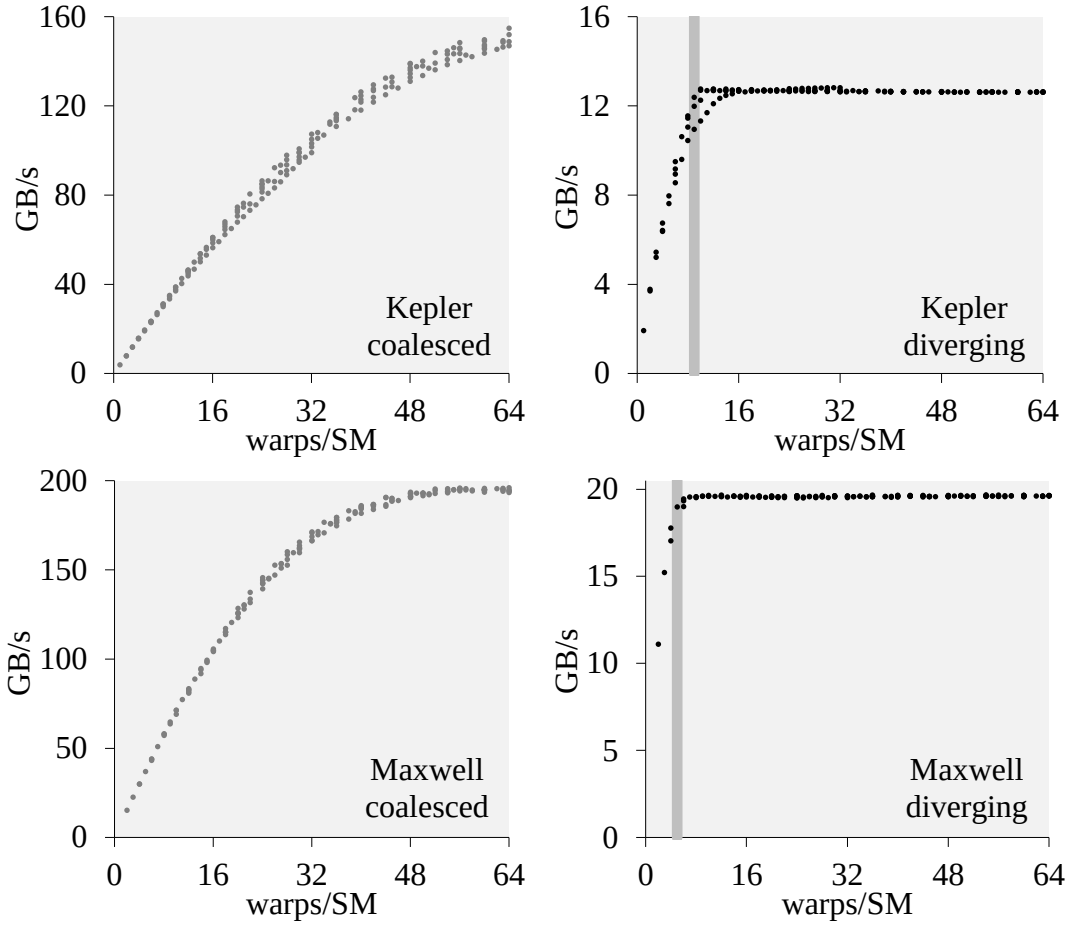


Figure 3.8: Performance of the permutation kernel (continued).

The respective increase in warp latency, in contrast, is relatively small. Indeed, issuing each additional transaction requires only a small additional delay since all of the transactions are independent and can be processed at the same time. In Chapter 6 we find that this increment is around 6 to 8 cycles per transaction on some GPUs and about 33 cycles per transaction on other GPUs. In the latter case, given SIMD width 32, the increments may yet add up to about a thousand cycles, but even then warp latency only about doubles.

To summarize, warp latency increases only moderately, but warp throughput decreases dramatically. Therefore, the needed occupancy, which equals their product, must also substantially decrease.

To compare this expectation with experiment, we run the kernel on all five GPUs. The code is slightly adapted to fit each GPU better. For the G80 and GT200 GPUs we replace integer multiplication done to compute `i` with the `__mul24` intrinsic function, and for the G80, GT200 and Fermi GPUs we organize thread blocks into a two-dimensional grid. Both are common practices described in the CUDA C programming guide [NVIDIA 2010a]. The array sizes are large and are the same as selected in §6.7, where non-coalesced accesses are studied in more detail. Otherwise the experimental setup is described in Chapter 5.

The result is shown in Figures 3.7 and 3.8. Throughput is reported in GB/s, counting each thread as 12 bytes. As expected, the needed occupancy is substantially smaller if the permutation is

```
__global__ void vabs( float *a )
 {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    float b = a[i];
    if( b < 0 ) a[i] = -b;
 }
```

Listing 3.7: Computation of the absolute value of array elements.

random: only about 6% to 15%, as shown with vertical bars (right column). This is compared to over 50% to 100% occupancy needed when the permutation is trivial (left column).

Reads versus writes. In the second example, we consider how the needed occupancy may depend on execution path taken in a kernel.

The kernel is shown in Listing 3.7. It computes the absolute value of array elements. As above, we slightly modify the code to fit each GPU better, such as use the `__mul24` intrinsic function and two-dimensional grid of thread blocks on earlier GPUs. The array is assumed to be large. When finding memory throughput, each thread is counted as 4 or 8 bytes.

We compare the occupancies needed in the following two cases. In the first case, all array elements are positive, so that data is read but not written back as no change in sign is required. In the second case, all array elements are negative; execution takes a different path, so that the data is both read and written. In both cases, memory is accessed in a fully coalesced manner.

By using Little's law, we can easily see that occupancy needed in the second case is smaller. Indeed, the additional writes don't substantially increase warp latency, but do substantially reduce warp throughput. Therefore, they also reduce the product of warp latency and warp throughput. This product is the estimate for the needed occupancy.

Warp latency is not substantially affected because execution paths taken in the two cases differ by only a few instructions, none of which require a substantial stall. These are the store instruction and the arithmetic instruction that changes sign. Unlike loads, stores don't incur dependency stalls and thus add only a small number of cycles to warp latency. Latency of the arithmetic instruction is also small. The drop in warp throughput, in contrast, is substantial: the kernel is memory-bound, warp throughput is bound by data traffic, and twice as much data is transferred in the second case. Transferring twice as much data at about the same latency saturates memory system twice as fast and at half the occupancy.

This is similar to what we see in practice on the Maxwell GPU, as shown in Figure 3.10, bottom. Throughput equal to 190 GB/s, which is close to the maximum, is attained at occupancy 60 warps per SM if all data is positive, and at 40 warps per SM if all data is negative. The difference is somewhat less than the expected factor of 2. There are a few reasons for this: the small but nonzero increase in warp latency, the gradual saturation effect, and a slightly lower peak throughput in the case when half of the accesses are writes (about 200 GB/s) compared to the case when all accesses are reads (about 210 GB/s).

On the other GPUs, as shown in the same figure and Figure 3.9, the same pattern manifests itself in a different fashion. When all data is positive, the needed occupancy is so large that it, apparently, exceeds 100% (left column). Peak throughput, in this case, is never attained and the needed occupancies cannot be compared directly. Instead, we compare the occupancies needed to achieve a smaller throughput: the largest throughput attained in the all-positive case. It is attained at 100% or similar occupancy if all data is positive, and at 47% to 63% occupancy if all data is negative, as shown with vertical bars in the figure. This is a similar difference of about a factor of 2.

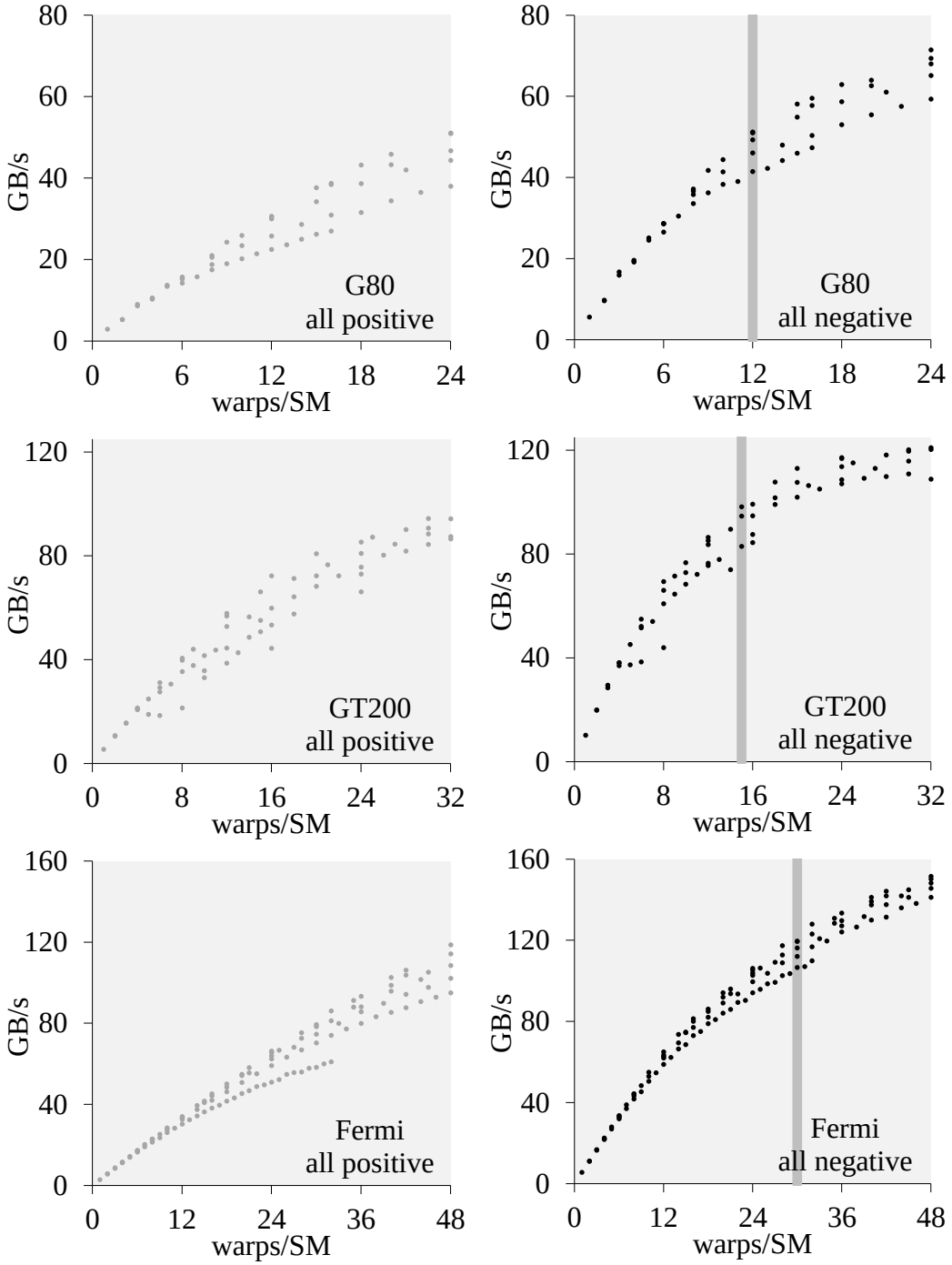


Figure 3.9: Performance of the "vector abs" kernel.

A larger instruction count. In the last example, we consider how the needed occupancy may depend on compilation options.

We consider the Black-Scholes kernel in the CUDA SDK [Podlozhnyuk 2007] as executed in our setup. In one case, it is compiled with option `-use_fast_math`; in the other, using no such option. Using the option reduces instruction count for the cost of reduced accuracy.

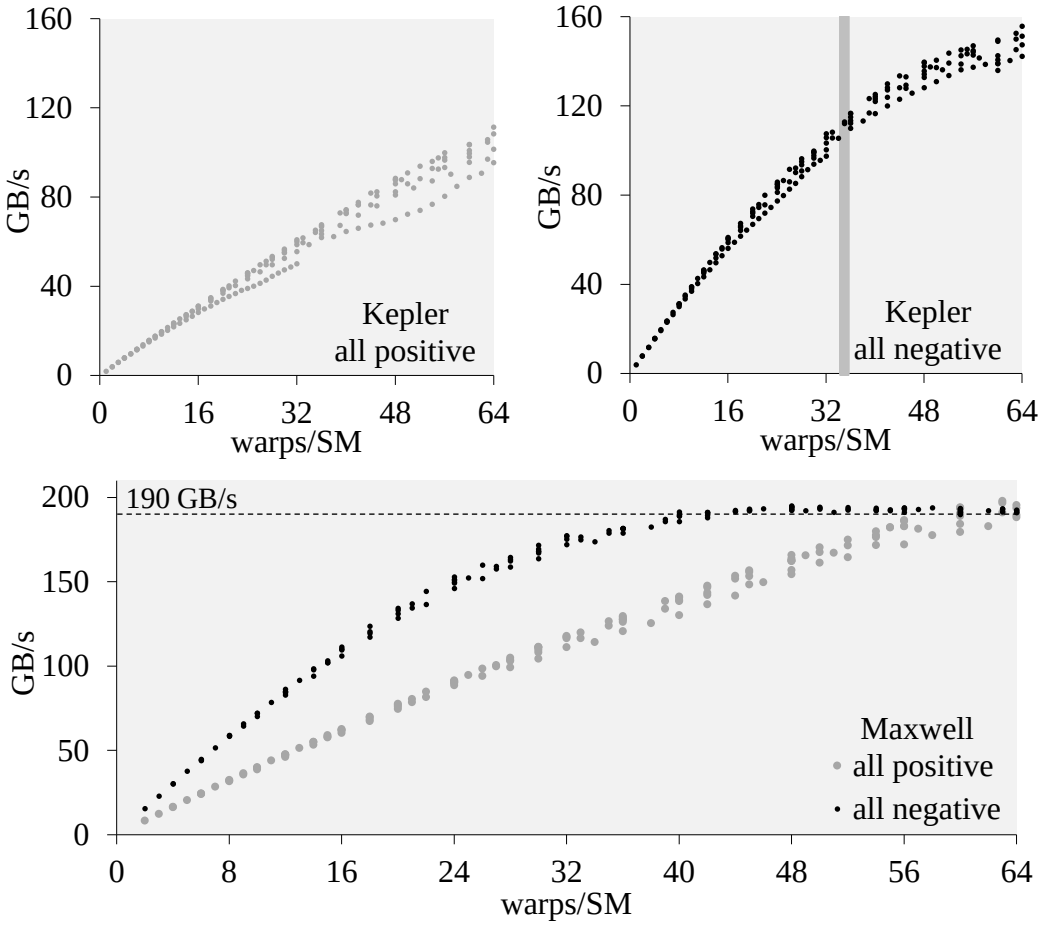


Figure 3.10: Performance of the "vector abs" kernel (continued).

Our interest is to find how the needed occupancy is affected by using this option. We assume it is known that the kernel is memory-bound in both cases, when the instruction count is larger and not. By being memory-bound we understand that overall throughput is bound by memory system, not other throughput limits.

It is easy to see that occupancy requirements are smaller when using the option. Again, we break down the needed occupancy into a product of warp latency and warp throughput. Warp throughput is the same in both cases, as the kernel is memory-bound, and using the option does not affect memory accesses. On the other hand, the reduced instruction count does results in a smaller warp latency. The required occupancy, therefore, is also smaller.

This is what we see on the Maxwell GPU, as shown in Figure 3.11. Close to peak throughput, 190 GB/s, is attained at occupancy equal about 44 or 48 warps per SM if not using the option, and at about 18 warps per SM if using the option. A similar result is found on the Kepler GPU: peak memory throughput is attained at about 50% occupancy if using the option, and no bound is visibly attained at even 100% occupancy if not using the option. In the latter case, however, it is not clear if peak memory throughput can be attained at even larger than 100% occupancies (if such were supported) – the code may be not memory-bound. On the Fermi GPU the code is clearly not memory-bound when instruction count is larger; the presented analysis does not apply. The compiler option does not substantially affect code generation on the G80 and GT200 GPUs, and we don't consider them here.

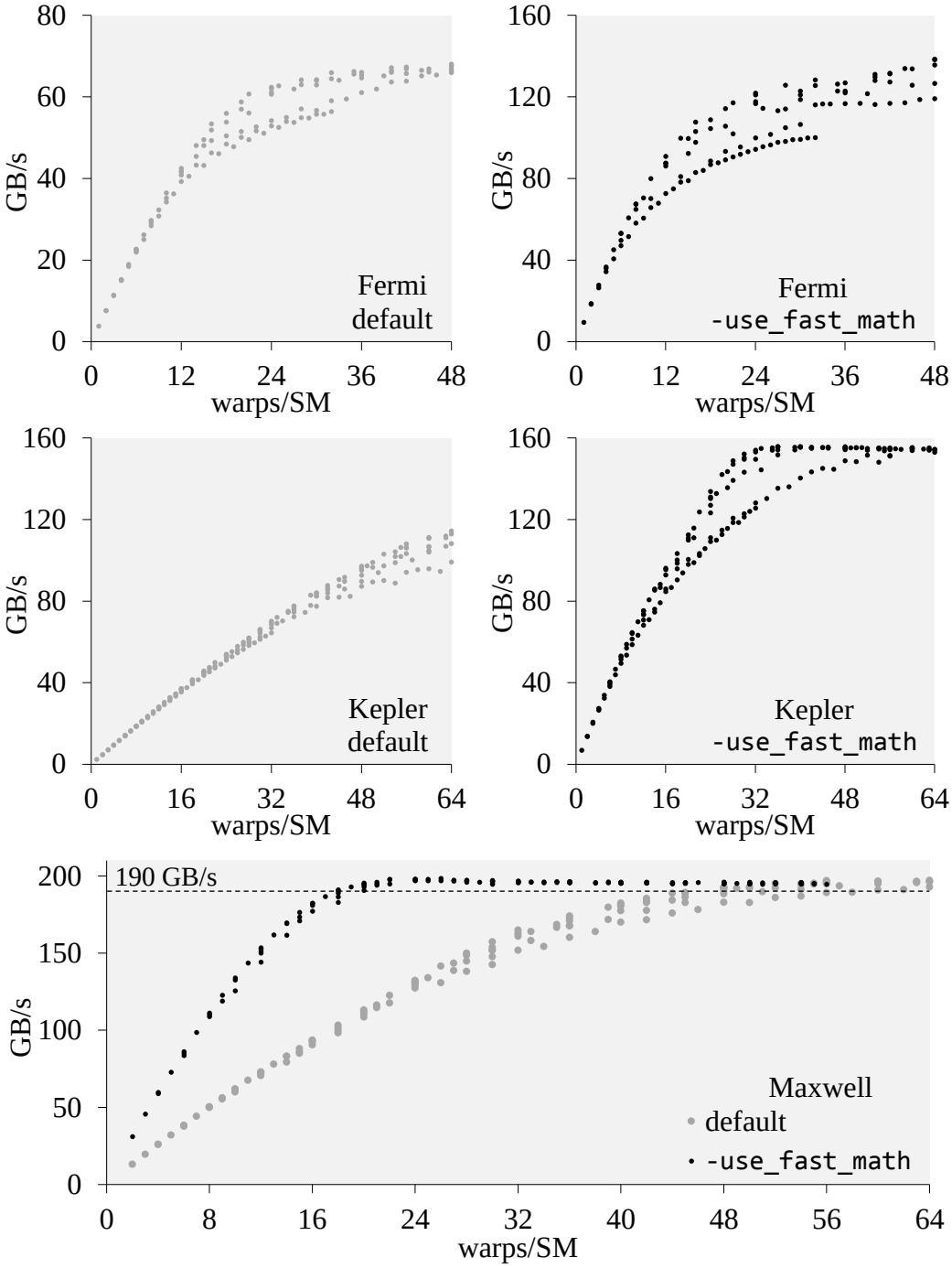


Figure 3.11: Performance of the Black-Scholes kernel.

Chapter 4
Understanding latency hiding

In this chapter we make the following contributions.

In §4.1 we present a simple workload that brings into focus some of the basic properties of GPU operation. We use it in §§ 4.5–4.7 to study how the occupancy needed to hide latency depends on arithmetic intensity of executed code. Surprisingly, the dependence, at least in this simplistic workload, is found to be distinctly non-monotonic: the needed occupancy increases with arithmetic intensity when arithmetic intensity is small and decreases otherwise.

Prior to that, in §4.3, we consider a simpler question: how the occupancy needed to hide arithmetic latency compares with the occupancy needed to hide memory latency. The first case corresponds to the execution of a workload where all instructions are arithmetic instructions, and the second case, where all instructions are memory instructions. Contrary to what is often assumed, the occupancies are found to be similar. In §4.4 we discuss a connection between Little's law and the "memory wall", which partially explains the finding.

In §§ 4.8–4.13 we discuss prior approaches to GPU performance modeling and identify a number of pitfalls. In particular, we show that some prior models are limited due to ignoring arithmetic latency, others are limited due to ignoring memory bandwidth, and yet others are limited despite including all relevant latencies and throughput limits into consideration – due to considering execution of different instruction types in isolation.

4.1 The setup and the model

We consider a kernel reduced to its most basic features: a mix of arithmetic and memory instructions, where all arithmetic instructions are identical and all memory instructions are also identical. The only variable parameter in the kernel is the proportion in which the two instructions are mixed. The instructions are selected to approximate the common case: memory accesses are coalesced and miss in the cache, arithmetic instructions are some of the more common CUDA core instructions – floating-point adds. The instructions are evenly interleaved and back-to-back dependent. The proportion in which instructions are mixed is called arithmetic intensity, or α: it is α arithmetic instructions per 1 memory access instruction, as shown below (to the left is the pseudo-code):

$a = memory[a]$	`LD R1, [R1]`
$a = a + b$	`FADD R1, R1, R2`
$a = a + b$	`FADD R1, R1, R2`
…	`...`
$a = a + b$	`FADD R1, R1, R2`
$a = memory[a]$	`LD R1, [R1]`
$a = a + b$	`FADD R1, R1, R2`
…	`...`

} α instructions

This instruction sequence is executed in each warp.

GPU	*mem_lat*, cycles	*mem_thru*, IPC/SM	*alu_lat*, cycles	*alu_thru*, IPC/SM	*issue_thru*, IPC/SM
G80	444	0.0268	20	0.25	0.5
GT200	434	0.0277	24	0.25	0.5
Fermi	513	0.0599	18	1	1
Kepler	301	0.1338	9	4	4
Maxwell	368	0.0814	6	4	4

Table 4.1: Hardware parameters used in our model.

For the purpose of performance modeling, we assume that the instruction sequence is infinite and includes no other instructions than these two. In practice, a small number of additional instructions still have to be used: a loop structure to avoid thrashing the instruction cache, an initialization code, timing, etc. A finer detail is using floating-point arithmetic on memory pointers. To avoid ruining the pointers, the added values are always zeros; yet, this does not suffice in floating-point semantics and we implement a few additional precautions. Note that adding zeroes does not mean that all loads in the same warp are from the same memory location. The locations are different: the address of each location is fetched by the previous load, so that the resulting memory access pattern depends on the contents of memory, as in the traditional pointer-chasing benchmark. The experimental setup is described in more detail in Chapter 5.

The kernel is inspired by the Roofline model [Williams et al. 2009] and a discussion in the CUDA C programming guide, Ch. 5.2.3 [NVIDIA 2015].

The following is the solution according to our model. First, we find a latency bound. Suppose that latency of arithmetic instructions is *alu_lat* cycles and latency of memory access instructions is *mem_lat* cycles. Warp latency per a repeating group of $\alpha + 1$ instructions, when only one warp is executed at a time, then is

$$Latency = mem\_lat + \alpha \cdot alu\_lat.$$

For convenience, we consider the throughput sustained in executing memory access instructions. When executing 1 warp at a time, it equals 1 / *Latency* instructions per cycle (IPC). When executing *n* warps at a time, it is at most *n* times larger:

$$Memory\ throughput \leq \frac{n}{Latency}.$$

We usually deal with throughput and occupancy metrics as given per SM, in which case *n* is in warps per SM and throughput is in IPC per SM.

Next step is finding a throughput bound. We consider three bottlenecks: CUDA cores, memory system and instruction issue. For the issue bottleneck, we assume that each executed instruction is issued only once and consumes a single issue cycle. Suppose that the throughputs sustained in each of these bottlenecks are bound by constants *alu_thru*, *mem_thru* and *issue_thru* respectively, i.e.:

$$\begin{aligned} Arithmetic\ throughput &\leq alu\_thru, \\ Memory\ throughput &\leq mem\_thru, \\ Instruction\ throughput &\leq issue\_thru. \end{aligned}$$

GPU	Estimated arithmetic throughput, adds/cycle per SM
G80	$32 \cdot \alpha \cdot \min (n / (444 + 20 \cdot \alpha),$ 0.0268, $0.25 / \alpha)$
GT200	$32 \cdot \alpha \cdot \min (n / (434 + 24 \cdot \alpha),$ 0.0277, $0.25 / \alpha)$
Fermi	$32 \cdot \alpha \cdot \min (n / (513 + 18 \cdot \alpha),$ 0.0599, $1 / (\alpha+1))$
Kepler	$32 \cdot \alpha \cdot \min (n / (301 + 9 \cdot \alpha),$ 0.1338, $4 / (\alpha+1))$
Maxwell	$32 \cdot \alpha \cdot \min (n / (368 + 6 \cdot \alpha),$ 0.0814, $4 / (\alpha+1))$

Table 4.2: The resulting solutions.

The throughputs are related by the fraction of arithmetic instructions in the mix. Since α arithmetic instructions and α+1 overall instructions are executed for each memory instruction, the relationship is:

$$\begin{aligned} \textit{Arithmetic throughput} &= \alpha \cdot \textit{Memory throughput}, \\ \textit{Instruction throughput} &= (\alpha+1) \cdot \textit{Memory throughput}. \end{aligned}$$

This allows reducing the bounds on different throughputs to a bound on one throughput, such as memory throughput. We get:

$$\textit{Memory throughput} \le \min\left(\textit{mem\_thru}, \frac{\textit{alu\_thru}}{\alpha}, \frac{\textit{issue\_thru}}{\alpha+1} \right).$$

This concludes the derivation of throughput bound. Adding in the latency bound, we get an overall bound on throughput, which we treat as the expected value. The expected memory throughput, thus, is:

$$\textit{Memory throughput} \approx \min\left(\frac{n}{\textit{Latency}}, \textit{mem\_thru}, \frac{\textit{alu\_thru}}{\alpha}, \frac{\textit{issue\_thru}}{\alpha+1} \right).$$

This solution can be somewhat simplified if we note that only two of the three throughput limits matter on any particular GPU. Indeed, on the G80 and GT200 GPUs issue throughput is overprovided and therefore the respective bound can be omitted. The solution, then, reduces to:

$$\textit{Memory throughput} = \min\left(\frac{n}{\textit{Latency}}, \textit{mem\_thru}, \frac{\textit{alu\_thru}}{\alpha} \right).$$

On the Fermi and later GPUs, on the other hand, we may omit the throughput limit due to CUDA cores: it is the same or looser than the limit due to instruction issue unless dual-issue is possible, which is not the case here as all instructions are back-to-back dependent. The resulting solution is:

$$\textit{Memory throughput} = \min\left(\frac{n}{\textit{Latency}}, \textit{mem\_thru}, \frac{\textit{issue\_thru}}{\alpha+1} \right).$$

These are our final solutions for memory throughput. We get the respective arithmetic throughputs in adds per cycle per SM by multiplying them by $32 \cdot \alpha$.

Input parameters for the GPUs used in our study are listed Table 4.1. Issue throughputs are taken from vendor's documentation (as quoted in Tables 2.6 and 2.8), but otherwise the numbers are

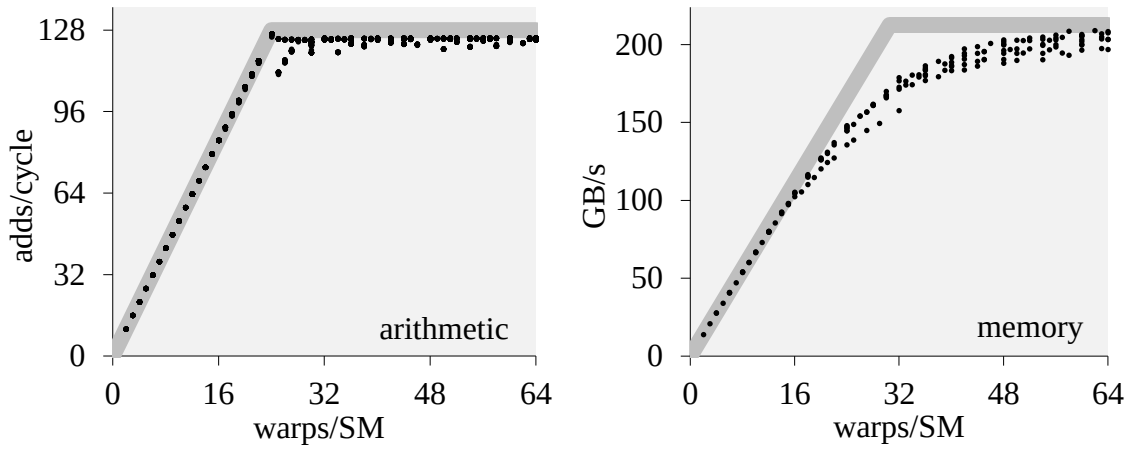


Figure 4.1: Hiding arithmetic and memory latencies on the Maxwell GPU.

found experimentally as described in Chapter 6. For example, peak memory throughputs are taken from Table 6.4 and translated to IPC/SM units by dividing them by the number of SMs (Table 2.6), the clock rate (Table 2.7), and the number of bytes transferred per instruction (128 bytes). For convenience below, single-issue is factored into both issue and arithmetic throughput limits. For example, arithmetic throughput on the Kepler GPU is limited to 6 IPC/SM if taking into account CUDA cores only, and to 4 IPC/SM if also taking into account single-issue.

For convenience, Table 4.2 lists the resulting solutions with numerical values of constant parameters substituted in.

4.2 Limit cases

At limits $\alpha = 0$ and $\alpha = \infty$, the kernel reduces to the most basic forms, performance in which is well-understood. At $\alpha = \infty$, all instructions are adds:

$a = a + b$	`FADD R1, R1, R2`
$a = a + b$	`FADD R1, R1, R2`
$a = a + b$	`FADD R1, R1, R2`
...	`...`

and the model becomes:

$$Arithmetic\ throughput = \min\left(\frac{n}{alu\_lat}, alu\_thru\right).$$

This model matches experiment well; Figure 4.1, left, shows an example for the Maxwell GPU. The thick grey line in the figure is the model, the dots are the experiment. This case is the best understood. As we discuss later, a number of prior performance models use this model as a building block.

In the case $\alpha = 0$, all instructions are memory accesses:

$a = memory[a]$	`LD R1, [R1]`
$a = memory[a]$	`LD R1, [R1]`
...	`...`

and the model reduces to:

$$\textit{Memory throughput} = \min\left(\frac{n}{\textit{mem\_lat}}, \textit{mem\_thru}\right).$$

This case is shown in the same figure, right. This time the difference with experiment is noticeable: in the experiment the latency bound smoothly translates into the throughput bound, whereas the model has a sharp knee. This difference is a basic limitation of our model.

The focus in the discussion below is on the numbers of warps needed to hide latency, i.e. attain a maximum throughput. In these two cases the model suggests the following two solutions:

$$\begin{aligned} &\textit{warps needed} \text{ (arithmetic)} = \textit{alu\_lat} \cdot \textit{alu\_thru}, \\ &\textit{warps needed} \text{ (memory)} = \textit{mem\_lat} \cdot \textit{mem\_thru}. \end{aligned}$$

These relations can be recognized as applications of Little's law. Note that in the case of memory instructions the solution is approximate.

These solutions for occupancies are well-known. The case of hiding arithmetic latency ($\alpha = \infty$) is considered in the CUDA C programming guide, Ch. 5.2.3 [NVIDIA 2015]. The case of hiding memory latency is considered in Bailey [1997], and the result can be recognized as *MWP_peak_BW* in Hong and Kim [2009] and Sim et al. [2012]. A minor difference in our approach is that we treat arithmetic and memory instructions in a similar manner.

4.3 Does hiding memory latency require more warps?

Substituting the hardware parameters listed in Table 4.1 into the solutions above suggests that hiding arithmetic latency requires 24 warps per SM and hiding memory latency requires 30 warps per SM if on the Maxwell GPU – which are similar numbers. This is despite the dramatic difference in latencies – 6 cycles in one case and 368 cycles in another – and in contrast with the prevailing common wisdom.

Indeed, latency hiding on GPUs is almost universally associated with the longer memory latency. This is how GPU's massive multithreading was originally introduced in vendor's publications, such as Lindholm et al. [2008], Nickolls et al. [2008], and Nickolls and Dally [2010]; this view is still dominant today. Even publications that do recognize arithmetic latency, such as Gebhart et al. [2011], tend to downplay it as requiring "a much smaller pool of warps". Our findings are contrary: the required pools of warps are similar in both cases, at least on recent GPUs.

The result must not be surprising given Little's law: the required concurrency is a product of latency and throughput, not a function of latency alone. Memory latency is indeed larger than arithmetic latency but, at the same time, memory throughput is smaller than arithmetic throughput, so that the respective latency-throughput products are similar. Putting it in other words, accessing memory one access at a time is slow, but accessing memory is slow anyway, so that processing too many accesses at the same time is not needed and not helpful.

The comparison between hiding memory and arithmetic latencies is made more complicated by the gradual saturation effect, which is found with memory accesses but not arithmetic operations: we find that only about 80% of peak memory throughput is typically attained at the occupancies suggested by the model. Therefore, for a more complete account we also find the occupancies where 90 and 95% of the peaks are sustained in practice. For the Maxwell GPU, such as shown in Figure 4.1, these are 40 and 46 warps per SM respectively – the latter is about 1.5 times larger than the estimate.

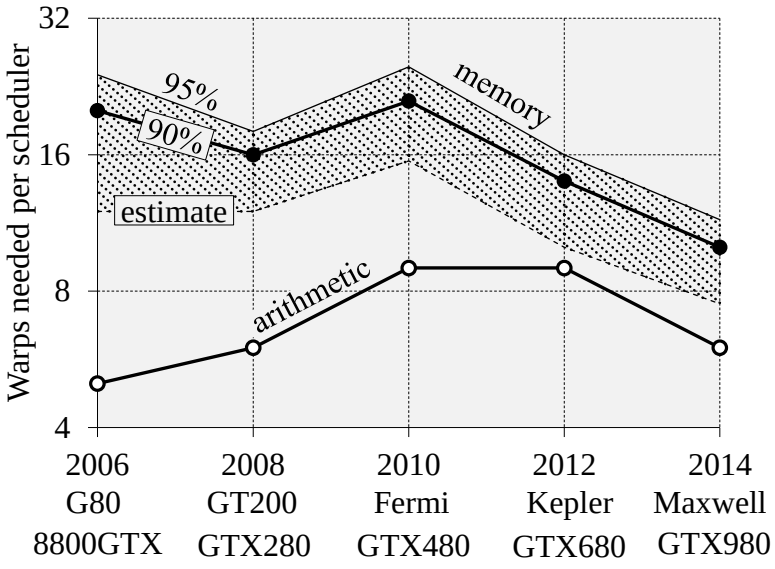


Figure 4.2: Occupancies needed to hide arithmetic and memory latencies.

We collected similar numbers for all five GPUs used in the study; they are plotted in Figure 4.2. The collection of data is detailed in Chapter 6, and the numbers are listed in Tables 6.3 and 6.4. The figure shows that the difference in occupancies has nearly monotonic dynamics: it is usually smaller in newer GPUs. If we divide the occupancy needed to sustain 90% of memory peak by the occupancy needed to sustain arithmetic peak, we see that this ratio decreases with every GPU generation except the last, as shown in the following table:

Generation Year	G80 2006	GT200 2008	Fermi 2010	Kepler 2012	Maxwell 2014
Ratio	4x	2.7x	2.3x	1.6x	1.7x

In the last generation the ratio increases back from 1.6x to 1.7x, but this is not a substantial reversal. The 4x difference in year 2006 is partially a feature of our setup; if we used multiply-add instructions instead of add instructions, this ratio would be 3.3x with other data unchanged. Also, we wouldn't see a different trend if were not limited by single-issue: dual-issue doesn't change the numbers of warps needed – as both instructions in the issued pairs come from the same warp – but does result in a larger number of concurrent arithmetic instructions on the Kepler GPU. Thus, if we considered similar ratios for instruction concurrency, the ratio for the Kepler GPU and this GPU alone would be smaller.

Note that the five GPUs we use are comparable: all are high-end GPUs used to promote each respective new GPU generation; each offered the leading performance at the time of its release and therefore represents the best effort in scaling GPU architecture.

Occupancy required to hide memory latency depends on the memory access pattern. Our primary focus is on the common access patterns where all accesses are fully coalesced and miss in the cache. If we consider accesses that are not so well coalesced but similarly miss in the cache, we find they require lower occupancy, not higher. This is because each instruction in this case is served with a larger number of transactions, so that the memory system is saturated with fewer concurrent instructions and fewer concurrent warps.

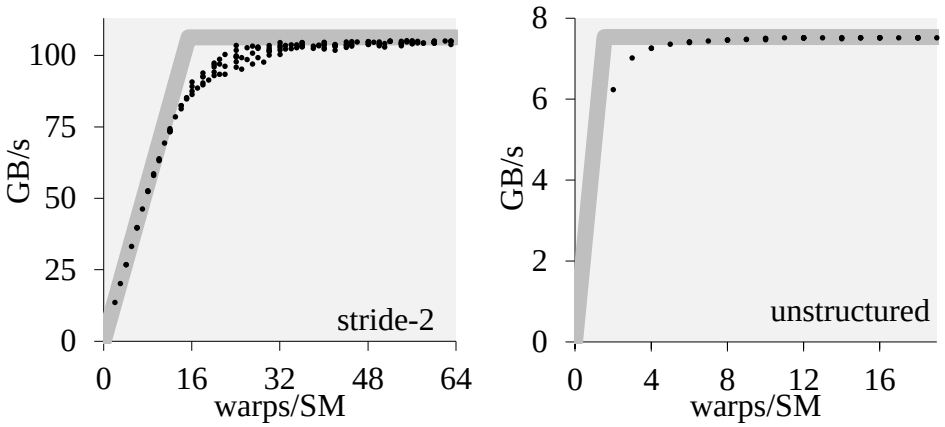


Figure 4.3: Hiding latency of non-coalesced memory accesses requires fewer warps.

Figure 4.3 shows two such examples for the Maxwell GPU. The graph on the left shows the throughput attained when using stride-2 access pattern. Peak throughput is 2x smaller – only 0.041 IPC/SM – whereas memory latency is only a little larger: 376 cycles. As a result, the required occupancy is 2x lower: 20 warps per SM for 90% of the peak. If the accesses are random, i.e. unstructured and fully divergent, latency increases less than 2x compared to the base level – to 534 cycles – whereas throughput drops by more than an order of magnitude, down to 0.0029 IPC/SM. As a result, only 3 warps per SM are needed to attain 90% of the peak. This case is shown in Figure 4.3, right. Also shown in the figure, in a thick grey line, is our model plotted using the quoted metrics. In §6.7 we find that in the case of unstructured accesses the same pattern also applies to other GPUs in the study – see Table 6.3.

These findings suggest that arithmetic latency must not be considered a second-class citizen in GPU performance modeling; yet, some of relatively recent models, such as Song et al. [2013], entirely omit it. A number of GPU performance models do consider arithmetic latency but with limited success, as we discuss later.

4.4 Latency hiding and the "memory wall"

To understand the forces behind the dynamics shown in Figure 4.2 and to speculate on its future development, it is instructive to factor the ratio between the occupancies into ratios between the latencies and the throughputs, i.e. as:

$$\frac{warps\ needed\ (memory)}{warps\ needed\ (arithmetic)} = \frac{memory\ latency}{arithmetic\ latency} \cdot \frac{memory\ throughput}{arithmetic\ throughput}.$$

The last ratio in this product – the ratio between memory and arithmetic throughput limits of a processor – has an important historical trend: it has been long observed that it tends to decrease, i.e. that improvements in memory throughput tend to lag behind improvements in arithmetic throughput. This trend is a part of what is known as "memory wall" – see, for example, Burger et al. [1996] and McKee [2004]. The trend is not exclusive to GPUs but is observed on GPUs as well.

To qualify the trend better, in Figure 4.4 we plot the ratios between memory and arithmetic throughputs versus the respective release dates for CUDA-capable GeForce GPUs; GeForce is NVIDIA's primary GPU product line. The vertical axis in the graph is pin memory bandwidth divided by the throughput of CUDA cores, where the latter is found as the number of CUDA cores

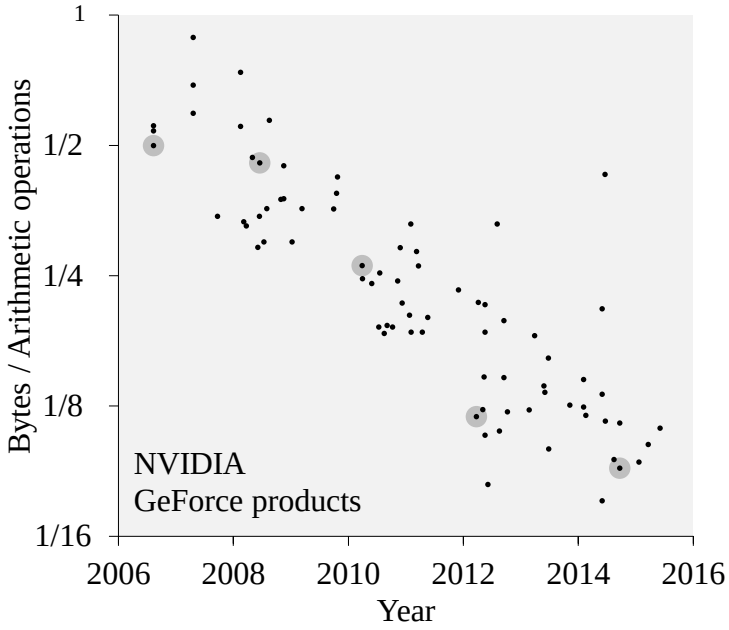


Figure 4.4: Improvements in memory bandwidth lag behind improvements in arithmetic throughput.

multiplied by the processor clock rate. The numbers are taken from vendor's website *www.geforce.com*. When two processor clock rates are given, we use the larger, "boost" clock rate. The horizontal axis in the graph is the release date found as the earliest date the respective product was first available at the popular online store *amazon.com*. The published online data, in both cases, includes a few obvious errors, which were corrected. A few GPU versions listed in vendor's website were not found in the store and are not plotted. Also excluded are laptop and OEM products. Highlighted in the figure are the GPUs used in this work. These are the key products associated with releases of new architecture generations.

The data is plotted in log-linear coordinates to highlight the well-defined exponential trend. The best fit (not shown) suggests that improvements in GPU's arithmetic throughput outpace improvements in GPU's memory throughput by a factor of about 1.24 per year on average. The trend survived transition from GDDR3 memory technology (G80, GT200) to GDDR5 technology (Fermi and newer); some of the GPUs in the figure use DDR2 technology. The most recent generation of GPU architecture, Pascal, features even newer memory technology: HBM2 [NVIDIA 2016]. We leave finding if this affects the trend to future work.

As provision of arithmetic throughput increases, so do the respective concurrency requirements. They could have already exceeded the concurrency requirements for memory accesses if not the improvements in arithmetic latency. Latency of floating-point add instructions, for example, was reduced from 24 cycles in the GT200 generation to 18 cycles in the Fermi generation to 9 cycles in the Kepler generation to 6 cycles in the Maxwell generation. Memory latency, in contrast, did not change as substantially (Table 4.1).

If the trend is to continue, we may eventually find that hiding arithmetic latency requires more warps than hiding memory latency, which is already the case with non-coalesced memory accesses.

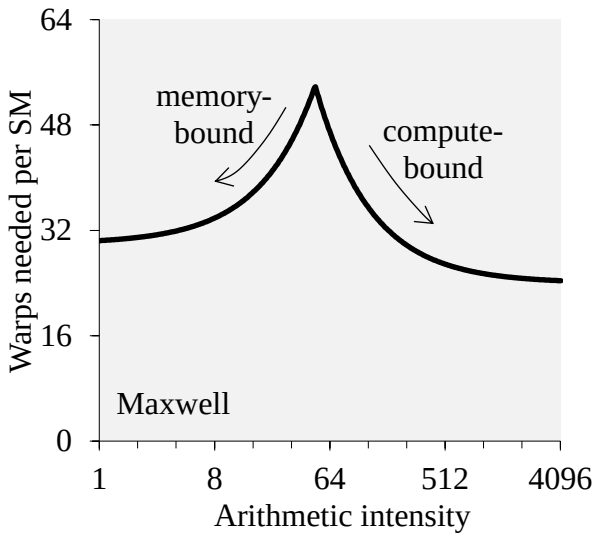


Figure 4.5: Occupancy needed to hide latency depends on arithmetic intensity of the kernel in a substantially non-monotonic fashion. Shown here is the estimate.

4.5 Cusp behavior

We return to the consideration of instruction mixes. Again, our interest is how many warps are needed to hide latency, i.e. to attain one of the peak throughputs. Given the model for throughput, the solution for occupancy in the general case is easily found as

$$n = Latency \cdot \min\left(mem\_thru,\ \frac{alu\_thru}{\alpha},\ \frac{issue\_thru}{\alpha+1}\right).$$

The solution has a distinct property: it depends on arithmetic intensity in a substantially non-monotonic fashion. Indeed, when throughput is bound by the memory system, i.e. at small arithmetic intensities, it reduces to

$$n \text{ (small } \alpha) = mem\_lat \cdot mem\_thru + \alpha \cdot alu\_lat \cdot mem\_thru,$$

which increases with arithmetic intensity. (We used that *Latency* equals $mem\_lat + \alpha \cdot alu\_lat$.) In the opposite case, i.e. when α is large and throughput is bound by either instruction issue or arithmetic capability, the needed number of warps decreases with arithmetic intensity. For example, on GPUs where throughput is bound by arithmetic capability, not instruction issue, the solution is

$$n \text{ (large } \alpha) = mem\_lat \cdot \frac{alu\_thru}{\alpha} + alu\_lat \cdot alu\_thru.$$

On other GPUs the number is slightly smaller.

The largest occupancy is required where the two branches meet, i.e. where the code is both compute- and memory-bound. Again, if we assume that throughput is bound by arithmetic capability, not instruction issue, the largest occupancy is required at $\alpha = alu\_thru / mem\_thru$ and equals

$$n \text{ (maximum)} = alu\_lat \cdot alu\_thru + mem\_lat \cdot mem\_thru.$$

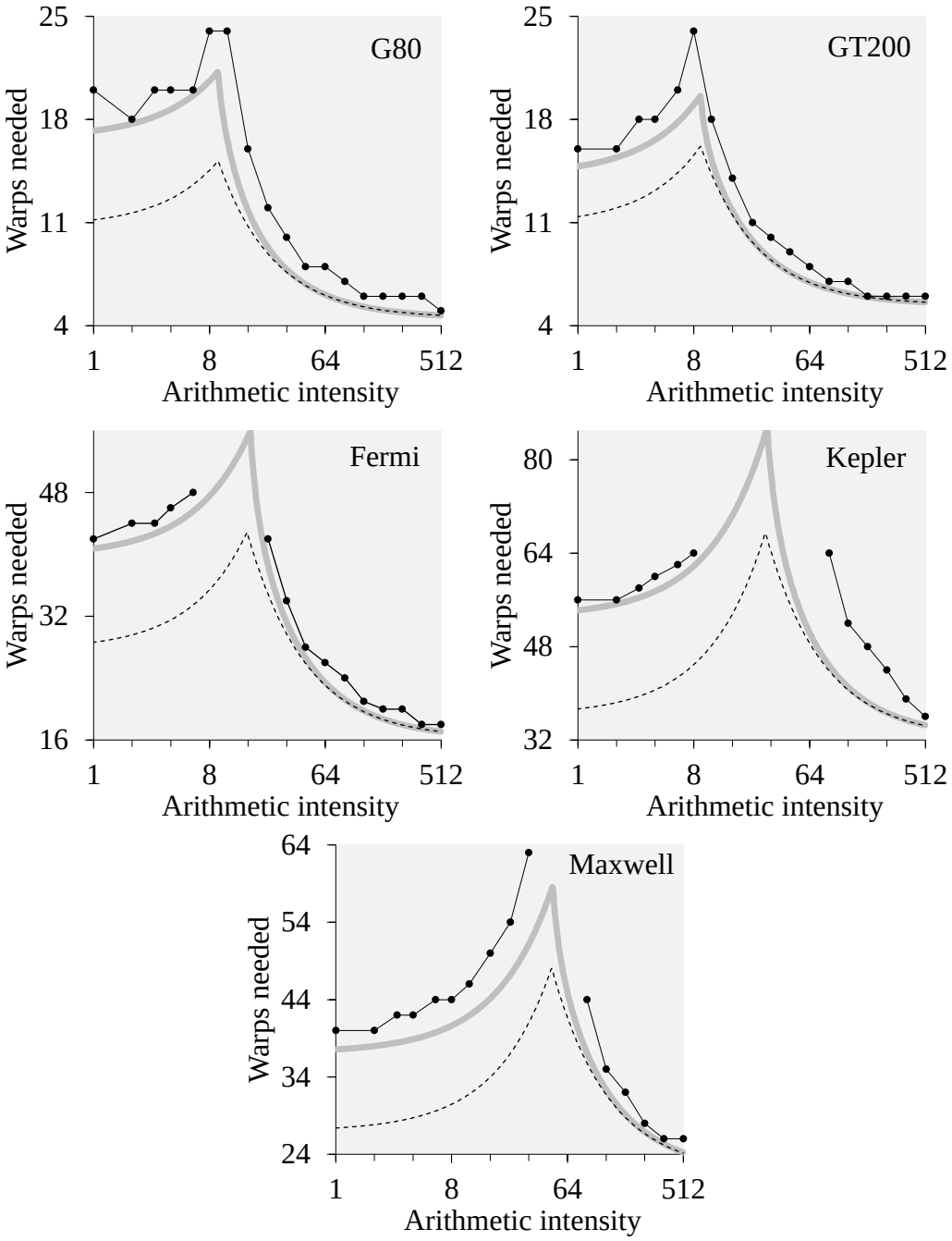


Figure 4.6: Cusp behavior in the model and the experiment.

This number can be recognized as sum of the occupancies required at limits $\alpha = 0$ and $\alpha = \infty$. The sum is substantially larger than either occupancy (at most by a factor of 2) and the solution is substantially non-monotonic only if the occupancies are comparable – but they are comparable, as we found earlier in this chapter.

The solution for the Maxwell GPU is plotted in Figure 4.5; the two branches are easily recognized in the figure. The values attained on the left- and the right-hand sides of the graph are approximately 30 and 24 warps found previously for limits $\alpha = 0$ and $\alpha = \infty$. In the middle of the

graph ($\alpha \approx 49$) the solution achieves approximately 54 warps, which is the sum of the two values. We call this pattern "cusp behavior" by the look of the graph.

In Figure 4.6 the model is compared with experiment. The model is shown as a dashed line, the experiment is shown as dots, and another model, explained below, is shown as a thick grey line. The values plotted in each case are the smallest occupancies where 90% of one of the peak throughputs is attained. No experimental data is plotted if the target throughput of 90% of peak is not attained at any occupancy, such as in cases where more warps are needed than physically supported.

According to the figure, cusp behavior is experimentally observed on each of the five GPUs, as predicted. This qualitative accuracy is a unique feature of our model. Later, in §7.10, we show that prior performance models suggest no similar behavior.

Quantitatively, the accuracy of our model is limited, which is due to the unaccounted gradual saturation effect. In §6.5, we describe a refined model that partially addresses the gradual saturation effect by taking into account memory contention. The respective new estimates are shown in the same figure as thick grey lines. They closely follow the experimental data except when using large α on the Kepler GPU.

4.6 Concurrent instructions are not concurrent warps

For a better understanding of cusp behavior we also find the numbers of arithmetic and memory instructions executed at the same time. They are different than the number of warps executed at the same time.

Consider latency hiding when bound by arithmetic throughput. In this case, when latency is hidden, arithmetic instructions are executed at rate *alu_thru*, and their latency equals *alu_lat*. The average number of arithmetic instructions executed at the same time then can be found by using Little's law as the product of these numbers:

$$\textit{arithmetic instructions in execution} = \textit{alu\_lat} \cdot \textit{alu\_thru}.$$

The number of memory instructions executed at the same time is found similarly. It is the product of the rate sustained in executing memory instructions, which is *alu_thru* / α, and the latency of these instructions, which is *mem_lat*:

$$\textit{memory instructions in execution} = \textit{mem\_lat} \cdot \frac{\textit{alu\_thru}}{\alpha}.$$

Since all instructions are back-to-back dependent, one warp can have only one instruction in execution at a time, such as one arithmetic instruction or one memory instruction, but not both. Therefore, the required number of warps is the sum of these numbers:

$$\textit{warps needed} = \textit{arithmetic instructions in execution} + \textit{memory instructions in execution}.$$

This produces the solution found previously, i.e.

$$\textit{warps needed} = \textit{mem\_lat} \cdot \frac{\textit{alu\_thru}}{\alpha} + \textit{alu\_lat} \cdot \textit{alu\_thru},$$

but also suggests an insight into the origins of the terms.

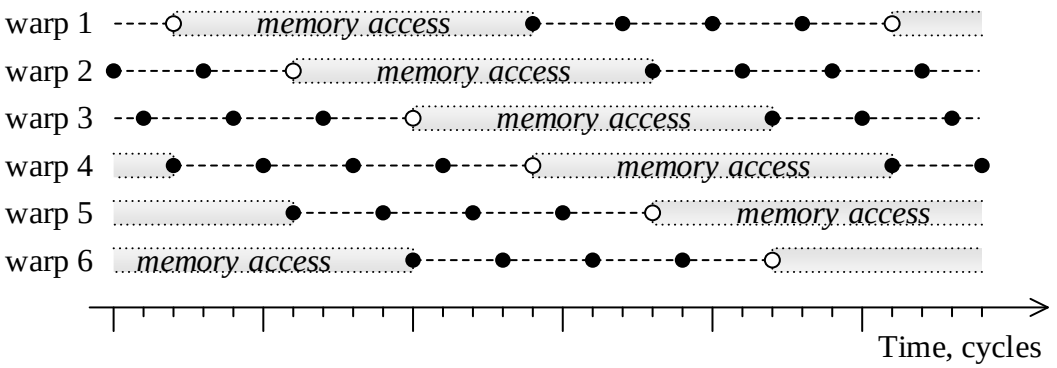


Figure 4.7: An example of a concurrent execution.

As arithmetic intensity increases, memory access rate decreases and the number of memory instructions executed at the same time also decreases. The number of arithmetic instructions executed at the same time, on the other hand, does not change because the same number is needed to sustain the same peak throughput. As a result, fewer warps are needed at larger arithmetic intensities.

The case when it is the memory system that is the bottleneck is considered similarly. The number of concurrently executed memory access instructions in this case is constant: it is the number needed to sustain peak memory throughput. The number of concurrently executed arithmetic instructions, on the other hand, increases when α increases – because arithmetic throughput increases. The needed occupancy then increases too.

The difference between concurrent instructions and concurrent warps is illustrated in Figure 4.7. There are $\alpha = 4$ arithmetic instructions per memory instruction and $n = 6$ warps being executed at the same time. Instruction issue times are shown with circles, filled for arithmetic instructions and empty for memory instructions. Shown with dashed lines are the execution times of arithmetic instructions. Arithmetic latency is *alu_lat* = 3 cycles, memory latency is *mem_lat* = 12 cycles. Peak arithmetic throughput is *alu_thru* = 1 IPC, which is sustained in the example.

For 6 concurrently executed warps, there are only 3 concurrently executed arithmetic instructions and only 3 concurrently executed memory access instructions at any moment of time. This can be found in the figure or by using Little's law. Indeed, the product of arithmetic latency and arithmetic throughput is 3 cycles multiplied by 1 IPC, which equals 3 instructions. The product of memory latency and sustained memory throughput is 12 cycles multiplied by 0.25 IPC, which also equals 3 instructions. The sustained memory throughput, 0.25 IPC, is found as $1 / \alpha$ multiplied by arithmetic throughput.

Executing 5 or fewer warps in this example would result in attaining less than peak arithmetic throughput, which is in contrast to only 3 warps being sufficient to attain peak throughput when no memory instructions are present.

4.7 Cusp behavior with other instruction types

Cusp behavior is also found with other instruction mixes. We briefly consider two of them: a mix of adds and reciprocal square roots:

$a = 1 / \sqrt{a}$	`MUFU.RSQ R1, R1`
$a = a + b$	`FADD R1, R1, R2`
$a = a + b$	`FADD R1, R1, R2`
…	`...`

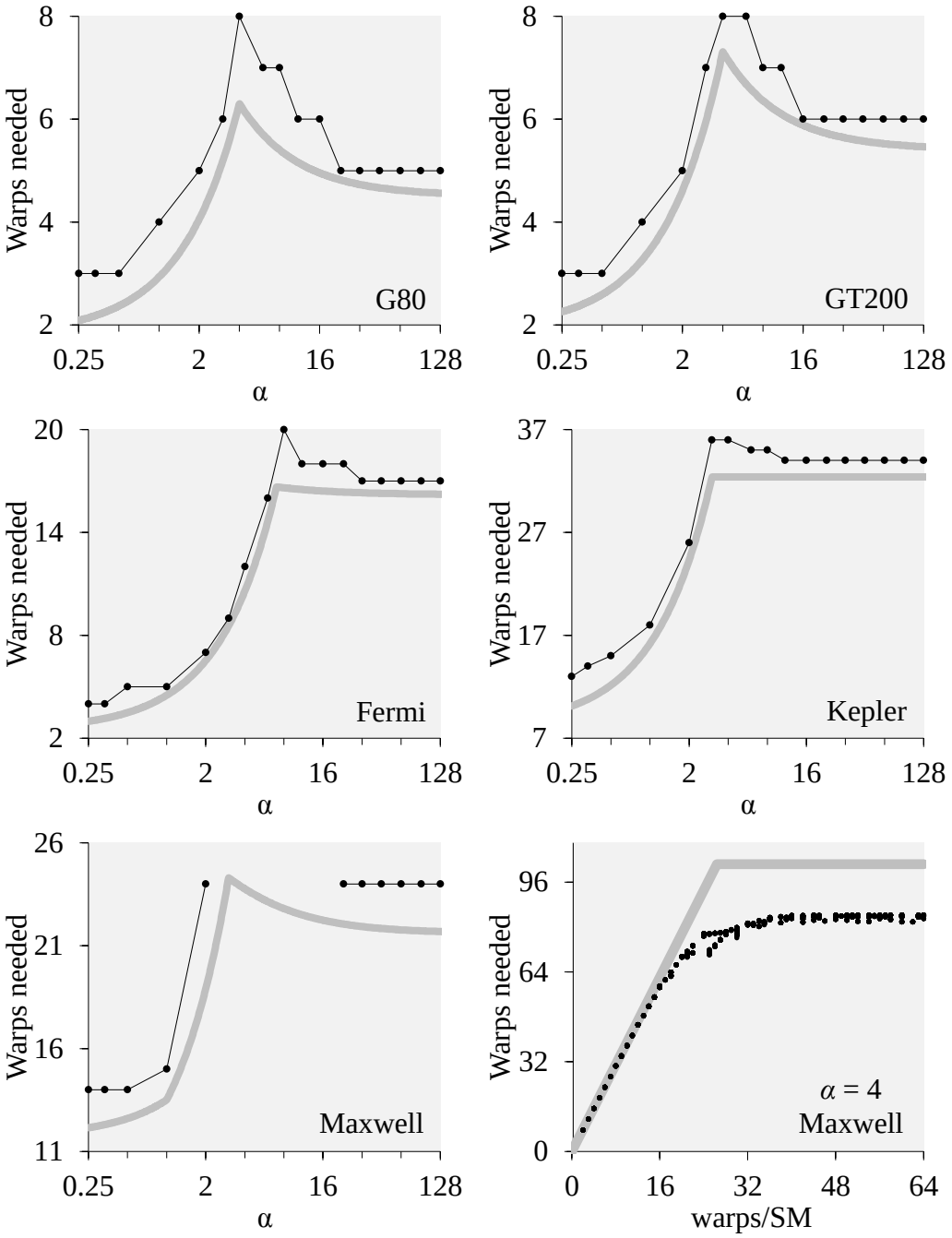


Figure 4.8: Cusp behavior with SFU instructions.

and a mix of adds and shared memory accesses:

a = *shared memory* [*a*]	`LDS R1, [R1]`
a = *a* + *b*	`FADD R1, R1, R2`
a = *a* + *b*	`FADD R1, R1, R2`
…	`...`

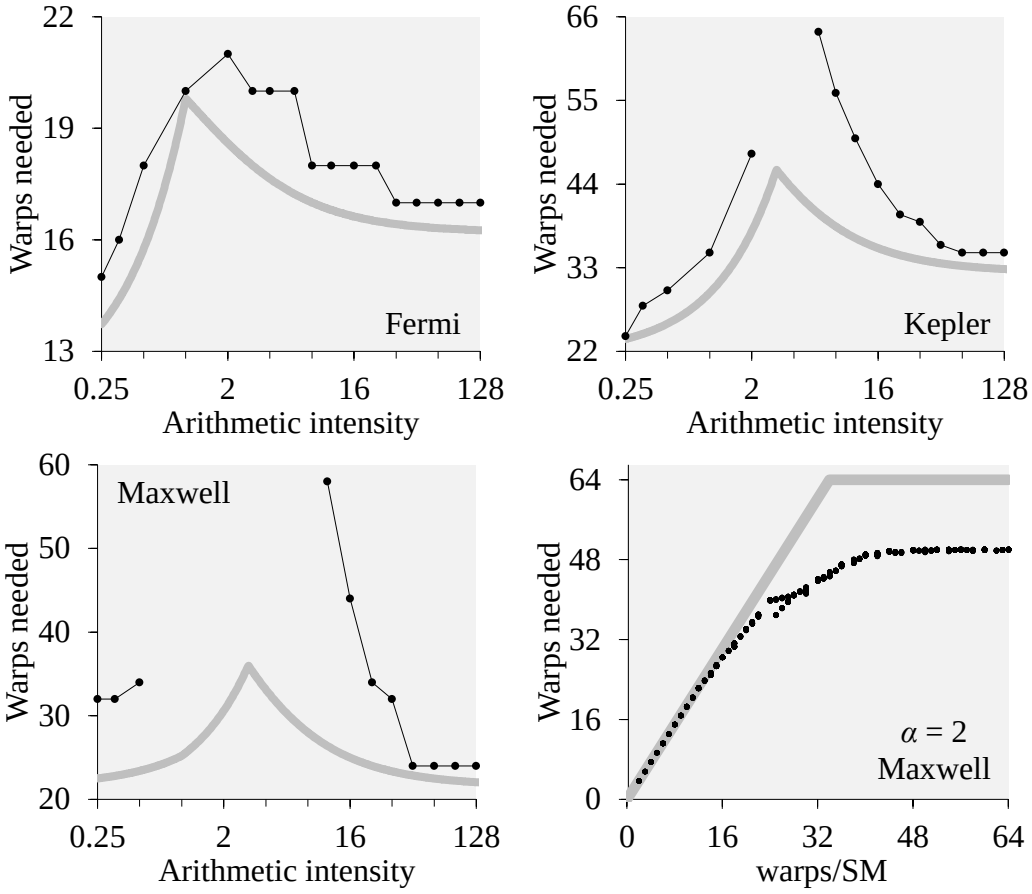


Figure 4.9: Cusp behavior with shared memory accesses.

In the latter case the accesses are set up to cause no bank conflicts. The second mix is not feasible and is not considered on the G80 and the GT200 GPUs. On these GPUs shared memory is addressed using special registers, which requires using additional instructions. In both mixes the instructions are evenly interleaved and all are back-to-back dependent. There are α add instructions per 1 other instruction, or, if α is less than one, 1 add instruction per 1 / α other instructions.

The performance model is derived similarly as before. Instruction latencies are used to find the latency bound, and peak throughputs are used to find the throughput bound. There are two latencies and three bottlenecks; the difference with the prior case, therefore, is only in the metrics used. In place of latency and throughput of global memory access instructions, we use similar metrics for SFU and shared memory access instructions. The needed metrics are found in Chapter 6, Tables 6.1 and 6.2.

In the case of the Maxwell GPU we consider an additional detail: we note that some of the register dependency latencies on this processor depend on the type of the dependent instruction. Specifically, they are shorter if dependent instruction is a CUDA core instruction. For example, register dependency latency of an SFU instruction is 13 cycles if the dependent instruction is also an SFU instruction, and 9 cycles if the dependent instruction is a CUDA core instruction. Similarly, latency of a shared memory access instruction is 24 cycles if the dependent instruction is also a shared memory access instruction, and 22 cycles if the dependent instruction is a CUDA core instruction. We use these details when computing term *Latency* in the model.

The results are compared with experiment in Figures 4.8 and 4.9. The thick grey line is where 90% of one of the peak throughputs is attained in the model, and the dots are where the same throughput is attained in experiment. Cusp behavior is pronounced in a number of cases such as with SFU instructions on the G80 and GT200 GPUs, and with shared memory instructions on the Fermi and Kepler GPUs. This suggests that cusp behavior is not uncommon and not specific to global memory accesses.

In the case of the Maxwell GPU, the model is noticeably less accurate. This may be due to unaccounted issue costs. We assumed that issuing each instruction takes a single issue cycle, but this is not necessarily the case. Indeed, throughput in the experiment is found to saturate at lower values than suggested by the model; two such examples are shown in the same figures, bottom right graphs. The discrepancy is larger when the code is issue-bound or nearly so and SFU or shared memory instructions are more frequent, which suggests that these instructions consume an additional issue throughput. If we assume that issuing each SFU instruction takes 2 issue cycles, and issuing each shared memory instruction takes 3 issue cycles, the estimated saturated throughputs in the two examples match the experiment well. We leave further investigation to future work.

4.8 Solution in the CUDA C Programming Guide

In the remainder of this chapter we review several approaches taken in prior work. The purpose is to illustrate challenges in modeling latency hiding and to discuss common pitfalls. This review is done at a basic, conceptual level. A more "to the letter" review follows in Chapter 7.

We start with the CUDA C programming guide [NVIDIA 2015], which briefly discusses a similar problem: given arithmetic intensity of a kernel, find the occupancy needed to attain the best possible throughput. The discussion is found in Chapter 5.2.3 of the programming guide starting with version 3.0 [NVIDIA 2010a]. It is based on considering a few numerical examples; in the following, we use the numbers cited in version 5.5 of the guide [NVIDIA 2013]. The discussion is simplistic but important, as a similar pattern recurs in a number of other, more elaborate models.

Suppose memory latency is *mem_lat* = 600 cycles, peak instruction throughput is *instruction time* = 4 CPI, and there are $\alpha = 30$ arithmetic instructions per memory access instruction. If one warp stalls at a memory access, but instruction issue continues at the peak rate – which means that latency is hidden – then 150 instructions are issued by the time the stalled warp may continue. These instructions can only be issued from other warps. As only 30 arithmetic instructions can be issued from any particular warp before it similarly stalls on a memory access, "about 5 warps" are needed. This result can be summarized as:

$$\textit{warps needed}\ (\text{CUDA guide}) \approx \frac{\textit{mem\_lat}}{\textit{instruction time} \cdot \alpha}.$$

This is the result presented in the programming guide. There are a few trivial elaborations it does not include. First, we didn't count the stalled warp – the total warp count therefore is 6. Second, we did not include memory instructions – in total there are 31 instructions issued from each warp per memory stall. With these refinements, we get:

$$\textit{warps needed}\ (\text{refined}) = \frac{\textit{mem\_lat}}{\textit{instruction time} \cdot (\alpha + 1)} + 1.$$

However, this introduces only a little difference, and we use the original version below.

GPU	α	*mem_lat*	*instruction time*	*estimate*
G80	16	444 cycles	4 CPI	6.9 warps
GT200	16	434 cycles	4 CPI	6.8 warps
Fermi	32	513 cycles	1 CPI	16 warps
Kepler	32	301 cycles	0.25 CPI	38 warps
Maxwell	64	368 cycles	0.25 CPI	23 warps

Table 4.3: The data sheet for the analysis in the vendor's guide.

The substantial factor missing in this analysis is arithmetic latency. This is in line with the popular understanding of GPUs as processors that use multithreading to hide the long memory latency, in which case the shorter arithmetic latency is often deemphasized. This omission results in a substantial error.

The error can be found by rewriting the estimate as

$$warps\ needed\ \text{(CUDA guide)} \approx mem\_lat \cdot \frac{alu\_thru}{\alpha}.$$

Here, we use the fact that most instructions in the example are arithmetic, so that peak instruction throughput approximately equals peak arithmetic throughput. We, therefore, replace *instruction time* with 1 / *alu_thru* and get the above.

In this form, the estimate may be easily recognized as the number of memory instructions executed at the same time, which was found in §4.6. We also found that the number of warps executed at the same time is larger and equals

$$warps\ needed = mem\_lat \cdot \frac{alu\_thru}{\alpha} + alu\_lat \cdot alu\_thru.$$

The difference is the additional term $alu\_lat \cdot alu\_thru$ that accounts for hiding arithmetic latency.

To highlight the importance of this term, we compare the estimate with experiment. We choose similar arithmetic intensities to those used in the programming guide: large enough to keep the kernel compute-bound but small otherwise. Indeed, the estimate is not intended for cases where α is too small – throughput is then bound by memory bandwidth, which is not considered in the analysis. Also, the estimate is obviously inaccurate when α is very large: it converges to zero at $\alpha = \infty$, where $alu\_lat \cdot alu\_thru$ warps are needed in practice. We, therefore, set arithmetic intensities to the values approximately equal but somewhat larger than ratios *alu_thru* / *mem_thru*. These are larger values for newer GPUs, as these ratios on newer GPUs are similarly larger – this trend was discussed in §4.4. A similar trend is present in the programming guide: the considered arithmetic intensity is 10 in version 3.0, 15 in version 3.2 and 30 in version 5.0 – these are larger values in newer versions covering newer GPUs. The arithmetic intensities we choose are listed in Table 4.3.

For other input parameters – memory latencies and peak arithmetic throughputs – we use the same values as previously in this chapter. They are not always the same as the numbers quoted in the programming guide, but are more appropriate: they are found experimentally, tuned for the specific GPUs and the specific setup we use (e.g. assume no dual-issue) and are better consistent with prior work, such as with the latencies reported in Volkov and Demmel [2008], Hong and Kim [2009] and Wong et al. [2010]. These parameters and the resulting estimates are also listed in Table 4.3.

The comparison with experiment is shown in Figure 4.10. Experimental data is shown as dots, the original estimate is shown as a thick grey line, and the better estimate, which includes the

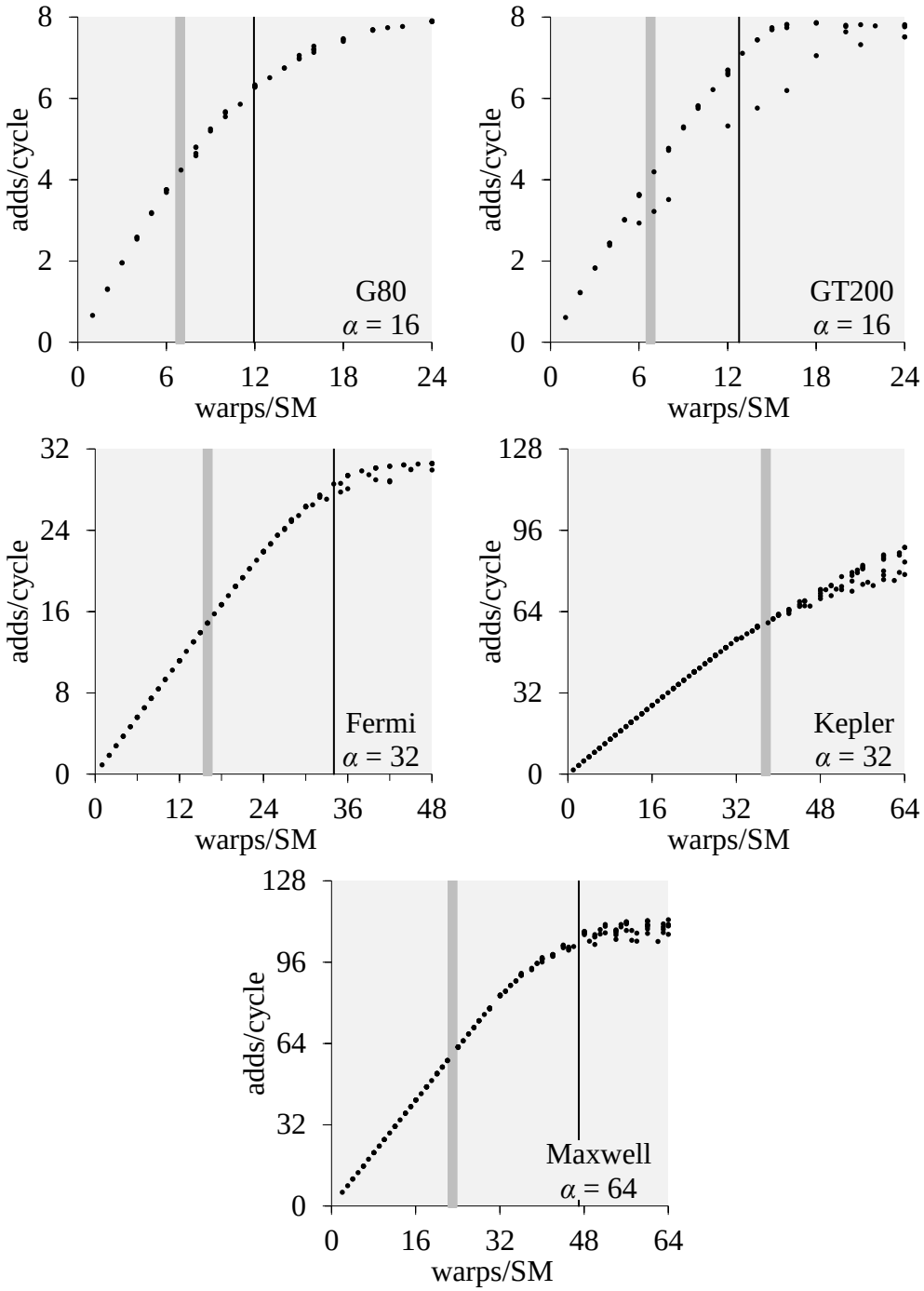


Figure 4.10: The analysis in vendor's programming guide substantially underestimates the needed occupancy (thick grey line). A better estimate takes into account both arithmetic and memory latency (thin black line).

additional term *alu_lat* · *alu_thru*, is shown as a thin black line. According to the figure, the vendor's model systematically underestimates the needed occupancy by a factor of about 2. The better estimate is nearly twice as large and is substantially more accurate, demonstrating that arithmetic latency is an important factor. The accuracy is still limited due to the unaccounted

gradual saturation effect, which is especially pronounced on the G80 GPU. In the case of the Kepler GPU, the better estimate (74 warps) exceeds 100% occupancy and is not shown.

The programming guide is not fully oblivious of arithmetic latency. For example, it is considered in another example which involves no memory accesses. The difficulty, therefore, is in understanding how to include both arithmetic and memory latency in the same analysis at the same time – or in realizing that it is necessary.

4.9 Similar solutions in prior work

The scheme described in the CUDA C programming guide also appears in a number of other performance models. We review them below to contrast their underlying assumptions with GPU operation.

Hong and Kim [2009] is one of the better known GPU performance models. It arrives at a similar solution by being consistently oblivious of arithmetic latencies. All instructions are assumed to execute in *instruction time* equal to 4 cycles on G80 and GT200 GPUs. Also, arithmetic instructions are assumed to incur no stalls on register dependencies. This is similar to assuming that arithmetic latency is also equal 4 cycles, or, more generally, that *alu_lat* = 1 / *alu_thru*. Under this assumption, the additional factor *alu_lat* · *alu_thru* discussed above is indeed trivial.

The number of warps needed to fully overlap memory stalls with execution is denoted in this model as *CWP_full* and is found as [Hong and Kim 2009, Eq. 8]:

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

Here, *Comp_cycles* is the total execution time for a warp and equals the time it takes to execute one instruction (i.e. *instruction time*) multiplied by the total number of instructions executed per warp. *Mem_cycles* is the total stall time for a warp and equals memory latency (*mem_lat*) multiplied by the number of memory instructions executed per warp. For our simplistic kernel this reduces to:

$$warps\ needed\ (\text{Hong and Kim}) = \frac{mem\_lat}{(\alpha+1) \cdot instruction\ time} + 1,$$

which is similar to the solution found in the programming guide.

Another source of similar results is the performance models for coarse-grained multithreaded processors, such as by Saavedra-Barrera et al. [1990] and Agarwal [1989]. These processors switch to a different thread only on long latency events, such as stalls on memory accesses; arithmetic latencies, thus, are not hidden by multithreading. In this respect, these models are similar to Hong and Kim model, which also implies that warps are switched only on memory accesses.

The models are derived as follows. Suppose each context switch takes C cycles, which is typically a non-zero number for coarse-grained multithreading, memory latency is L cycles, and each thread is executed for R cycles until stalling on a memory access. If n threads are executed at the same time, then each stall can be overlapped with execution of $n - 1$ threads and n context switches. Therefore, latency is hidden if $(n - 1) \cdot R + n \cdot C \geq L$. Solving this for n gives (Section 2 in Agarwal [1989], here in a different notation):

$$threads\ needed\ (\text{Agarwal}) = \frac{R + L}{R + C}.$$

Alternatively, we may assume that each stall can be overlapped with only $n - 1$ context switches, because the n-th context switch returns the control to the stalled thread, and thus is initiated only after the stall is finished. In this case, we solve $(n - 1) \cdot (R + C) \geq L$ and get [Saavedra-Barrera et al. 1990, Eq. 4]:

$$\textit{threads needed} \text{ (Saavedra-Barrera et al.)} = \frac{L}{R + C} + 1.$$

The two approaches are the same if $C = 0$ as on GPUs. If we also ignore arithmetic latency and assume that each instruction is executed in *instruction time* equal 1 / *alu_thru*, we get $R = (\alpha + 1) \cdot$ *instruction time*, which produces the same result for the needed number of physical threads as in the programming guide and the Hong and Kim model. If we do include arithmetic latency, such as by putting $R = \alpha \cdot$ *alu_lat*, the result is smaller and therefore is even less accurate.

Similar solutions were also suggested for prefetching, such as by Klaiber and Levy [1991] and Mowry et al. [1992]. Prefetching is another technique for latency hiding that hides memory but not arithmetic latency. It is typically applied to execution of a loop, in which case, instead of overlapping memory accesses with execution of other threads, memory accesses are overlapped with execution of other loop iterations in the same thread. This is done by requesting data several iterations ahead. The needed number of iterations is found similarly to the needed number of threads in multithreading. If memory latency is l cycles, each instruction executes in *instruction time* = 1 cycle, and s instructions are executed in each loop iteration, then the number of iterations to prefetch ahead is found as [Mowry et al. 1992]:

$$\textit{prefetch distance} \text{ (Mowry et al.)} = \frac{l}{s}.$$

This is similar to the solution suggested in the programming guide, if the loop is similar to our kernel, i.e. if each iteration is comprised of one memory access instruction and α other instructions.

The key feature of all these models is that they are built around hiding one latency only – memory latency. In contrast, GPUs hide a number of different latencies at the same time, including both arithmetic and memory latency. In modeling GPU performance, therefore, we must take a similar leap from hiding one latency to hiding different latencies.

4.10 Similar models but with arithmetic latency

There were attempts to improve the popular scheme above by including a separate model for hiding arithmetic latency. Such approaches are found in Baghsorkhi et al. [2010] and Sim et al. [2012]. Considering arithmetic latency separately, however, does not necessarily result in a better accuracy.

If we consider execution of arithmetic instructions alone, reciprocal instruction throughput can be easily found as:

$$\textit{arithmetic time}\ (n) = \max\left(\frac{\textit{alu\_lat}}{n},\ 1 / \textit{alu\_thru}\right),$$

where n is occupancy in warps per SM, and *arithmetic time* (n) is the expected throughput in cycles per instruction. As we have seen in §4.2, this estimate is indeed accurate when memory instructions are not present.

This throughput is then assumed to apply even if memory instructions are present, except when all warps are stalled on memory accesses at the same time. A scheme similar to above is then used to find if such overall stalls do happen.

Again, we start with one warp stalling on a memory access and consider instructions issued from other warps during the stall. Since only up to $(n-1) \cdot (\alpha+1)$ such instructions may be issued until all other warps also stall, instruction issue may continue at the new rate only for additional $(n-1) \cdot (\alpha+1) \cdot$ *arithmetic time* (n) cycles. The memory stall is hidden if this number of cycles is the same or larger than the duration of the stall, i.e. if the following quantity is zero or negative (after Baghsorkhi et al. [2010], Eq. 6):

$$\textit{Exposed latency} = \textit{mem\_lat} - (n-1) \cdot (\alpha+1) \cdot \textit{arithmetic time}\ (n).$$

By rearranging terms in $\textit{Exposed latency} \leq 0$, we get

$$n \geq \frac{\textit{mem\_lat}}{(\alpha+1) \cdot \textit{arithmetic time}\ (n)} + 1.$$

This is similar to the solution in the programming guide, except *arithmetic time* (n) is used in place of *instruction time*. Since *arithmetic time* (n) is not smaller than *instruction time* (the latter equals 1 / *alu_thru*), the new solution is not larger and therefore is not more accurate than before.

The factor *alu_lat* · *alu_thru* discussed earlier also appears in this model – it is the number of warps needed to hide arithmetic latency, i.e. to have *arithmetic time* (n) = 1 / *alu_thru* – but it is not added to the number of warps needed to hide memory latency, unlike in our model. Cusp behavior, therefore, is not possible.

Baghsorkhi et al. [2010] use this approach with minor differences: they assume that memory instructions are issued in 1 / *alu_thru* cycles, not in *arithmetic time* (n) cycles, and that memory latency is hidden, i.e. does not contribute to execution time, even if the exposed latency is a small positive number.

Sim et al. [2012] use a similar approach to define metric *CWP_full*, which was originally introduced in Hong and Kim [2009]. The new definition, if written in our notation and for our kernel, is:

$$\textit{CWP\_full}\ (\text{Sim et al.}) = \frac{\textit{mem\_lat}}{(\alpha+1) \cdot \textit{arithmetic time}\ (n)} + 1,$$

which is similar to the above.

Using rate *arithmetic time* (n) does not improve accuracy because this rate doesn't apply when memory stalls are present – even if all warps never stall at the same time. Indeed, after having $(n-1) \cdot (\alpha+1)$ instructions issued, the first stalled warp is assumed to continue execution, but alone, with other warps still being stalled. Instructions, then, may be issued only from this warp, and, therefore, only as often as one instruction in *alu_lat* cycles, independently of n – not one instruction in *arithmetic time* (n) cycles as was assumed.

4.11 Considering execution of arithmetic and memory instructions in isolation

Another popular approach is to consider execution of both arithmetic and memory instructions separately, as if instructions of the other type were not present in the kernel, and then to merge the

results. This approach, with refinements, appears in Zhang and Owens [2011] and Sim et al. [2012]. It suffers from a similar difficulty: throughputs found when considering different instruction types in isolation don't apply in general.

Suppose, we know instruction throughputs when arithmetic instructions are executed alone, as in case $\alpha = \infty$, and when memory instructions are executed alone, as in case $\alpha = 0$. For example, we may use the models found earlier, i.e.:

$$\textit{arithmetic time}\ (n) = \max\left(\frac{\textit{alu\_lat}}{n},\quad 1\ /\ \textit{alu\_thru}\right)\ \text{and}$$

$$\textit{memory time}\ (n)\quad = \max\left(\frac{\textit{mem\_lat}}{n},\ 1\ /\ \textit{mem\_thru}\right),$$

and this is what we do below, or we may use the throughputs sustained in respective micro-benchmarks, as suggested in Zhang and Owens [2011]. Here, both *arithmetic time* (*n*) and *memory time* (*n*) are reciprocal throughputs in cycles per instruction at each SM, and *n* is occupancy in warps per SM.

These throughputs are then assumed to apply in general, when executing an arbitrary kernel. In this case, the time to execute all arithmetic instructions in the kernel can be found as the product of *arithmetic time* (*n*) and the number of arithmetic instructions executed per SM, and the time to execute all memory instructions, as the product of *memory time* (*n*) and the number of memory instructions executed per SM. Since arithmetic and memory instructions are processed at the same time, the larger of these two is understood as the total execution time. (Sim et al. [2012] suggest a slightly more complicated merging procedure.) To summarize:

$$\begin{aligned}\textit{Time}\ (n) = \max\ (\ &\textit{arithmetic time}\ (n)\ \cdot\ \textit{number of arithmetic instructions},\\ &\textit{memory time}\ (n)\ \cdot\ \textit{number of memory instructions}\ ).\end{aligned}$$

For our simplistic kernel this equals

$$\textit{Time}\ (n) = \max\left(\alpha \cdot \textit{arithmetic time}\ (n),\ \textit{memory time}\ (n)\right)$$

per the repeating group of $\alpha + 1$ instructions. Arithmetic throughput in adds per cycle per SM is then found as $32 \cdot \alpha\ /\ \textit{Time}\ (n)$.

This approach is compared with experiment in Figure 4.11. As usual, the model is shown as a thick grey line, and experimental data is shown as dots. Arithmetic intensities used in the comparison are the same as used previously, when discussing the programming guide; for this model they correspond to nearly the worst case accuracy. In the shown cases, the model overestimates throughputs by factors of about 2 on all five GPUs when occupancy is small.

The throughputs are misestimated because rates *arithmetic time* (*n*) and *memory time* (*n*) don't apply when both arithmetic and memory instructions are present at the same time. Indeed, if *n* is small, *arithmetic time* (*n*) corresponds to executing *n* arithmetic instructions at the same time, and *memory time* (*n*) corresponds to executing *n* memory instructions at the same time. Using both rates together corresponds to executing both *n* arithmetic and *n* memory instructions at the same time – which is two instructions per warp. However, only one instruction per warp can be executed at a time in this particular kernel, as all instructions in it are back-to-back dependent. The model, thus, overestimates concurrency and, as a result, overestimates throughput.

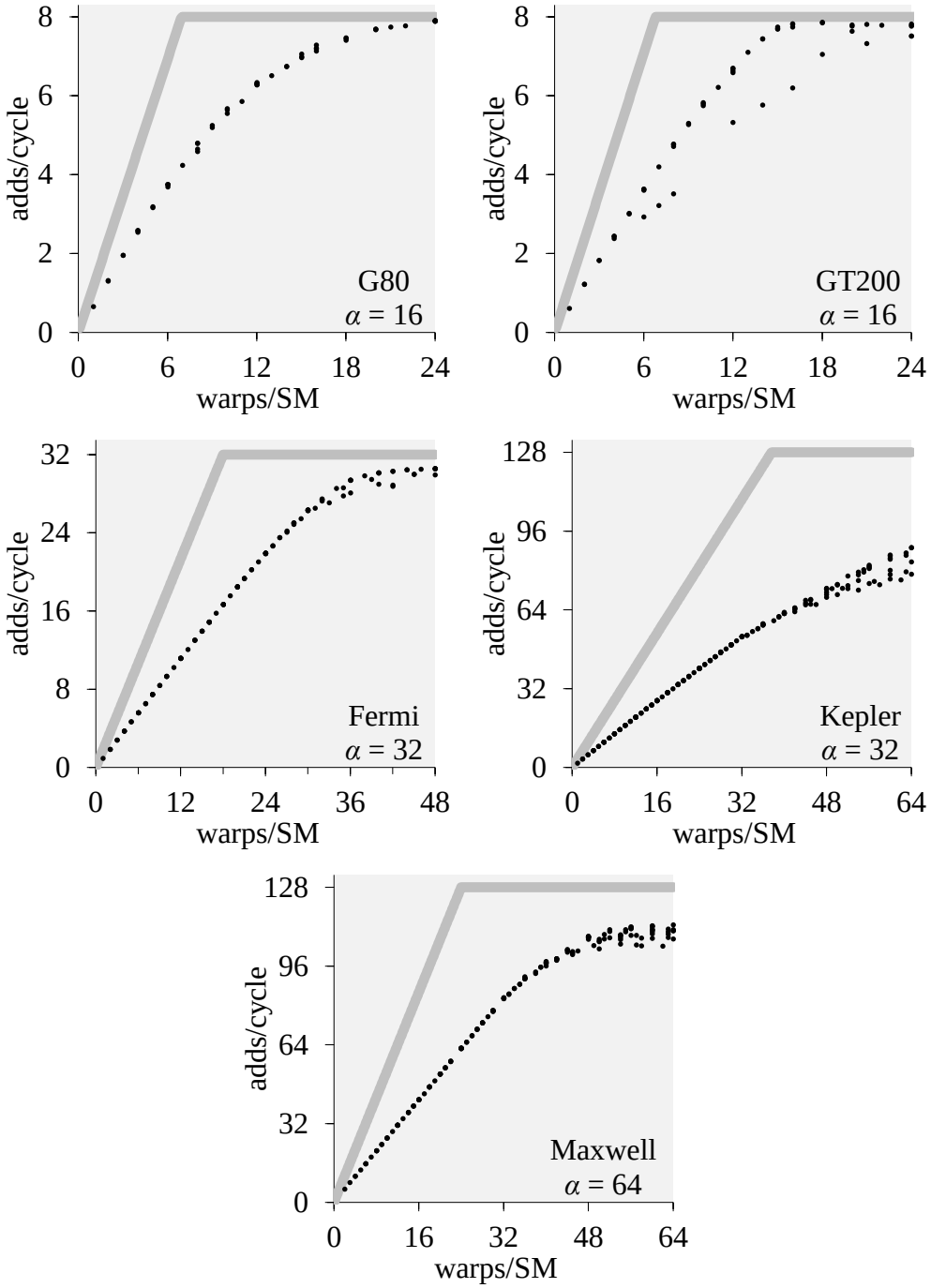


Figure 4.11: Assuming that arithmetic and memory instruction times overlap results in poor accuracy at small occupancies.

Yet, the model is reasonably accurate at large occupancies, where performance is throughput-bound. In this case, the concurrencies are sufficiently large to saturate the respective bottlenecks in both the model and the experiment.

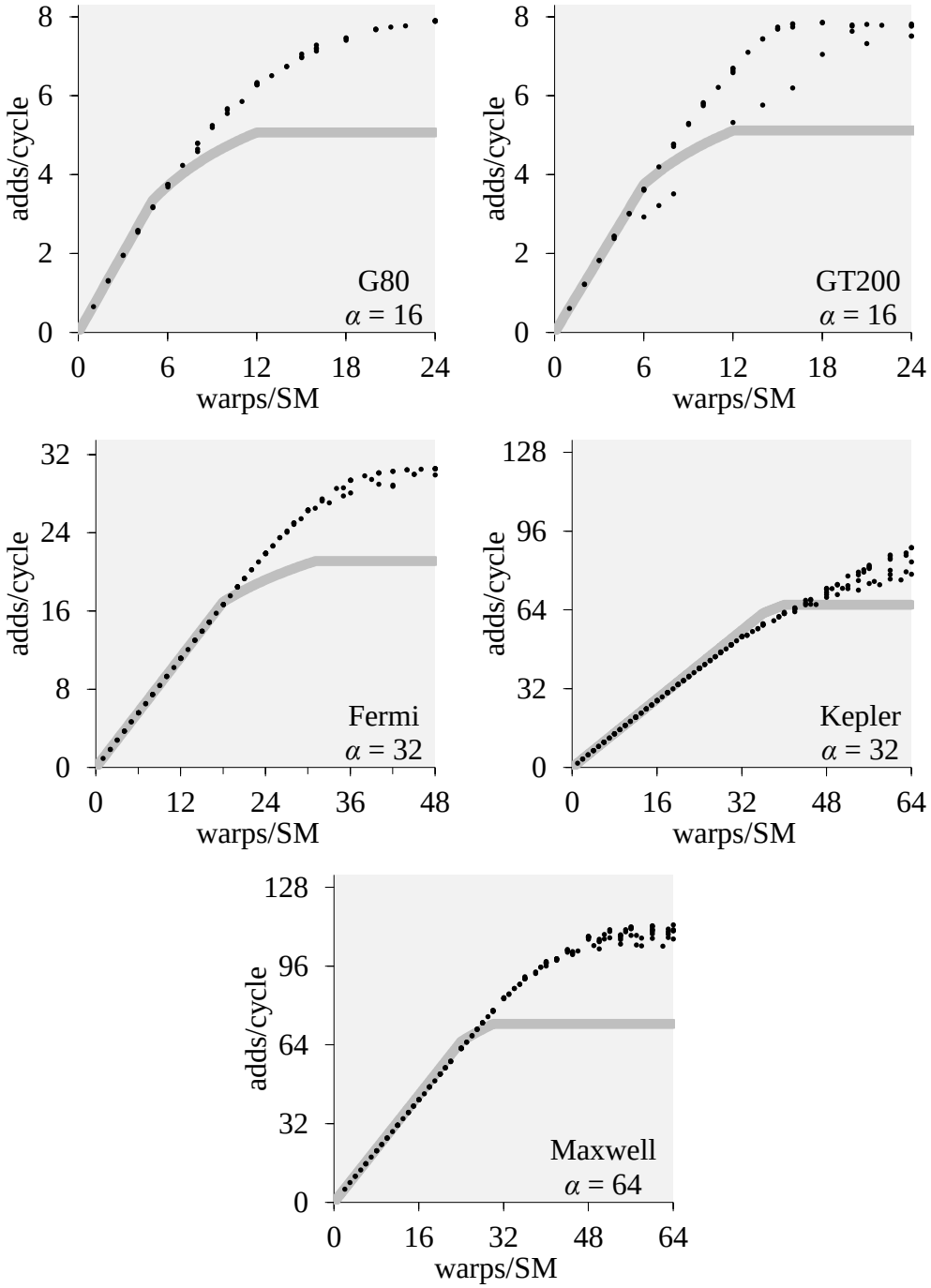


Figure 4.12: Assuming that arithmetic and memory instruction times add results in poor accuracy at large occupancies.

4.12 Do instruction times add or overlap?

A possible variation of the above model is to assume that execution times of arithmetic and memory instructions add, not overlap. In this case *Time* (*n*) is replaced, for our kernel, with

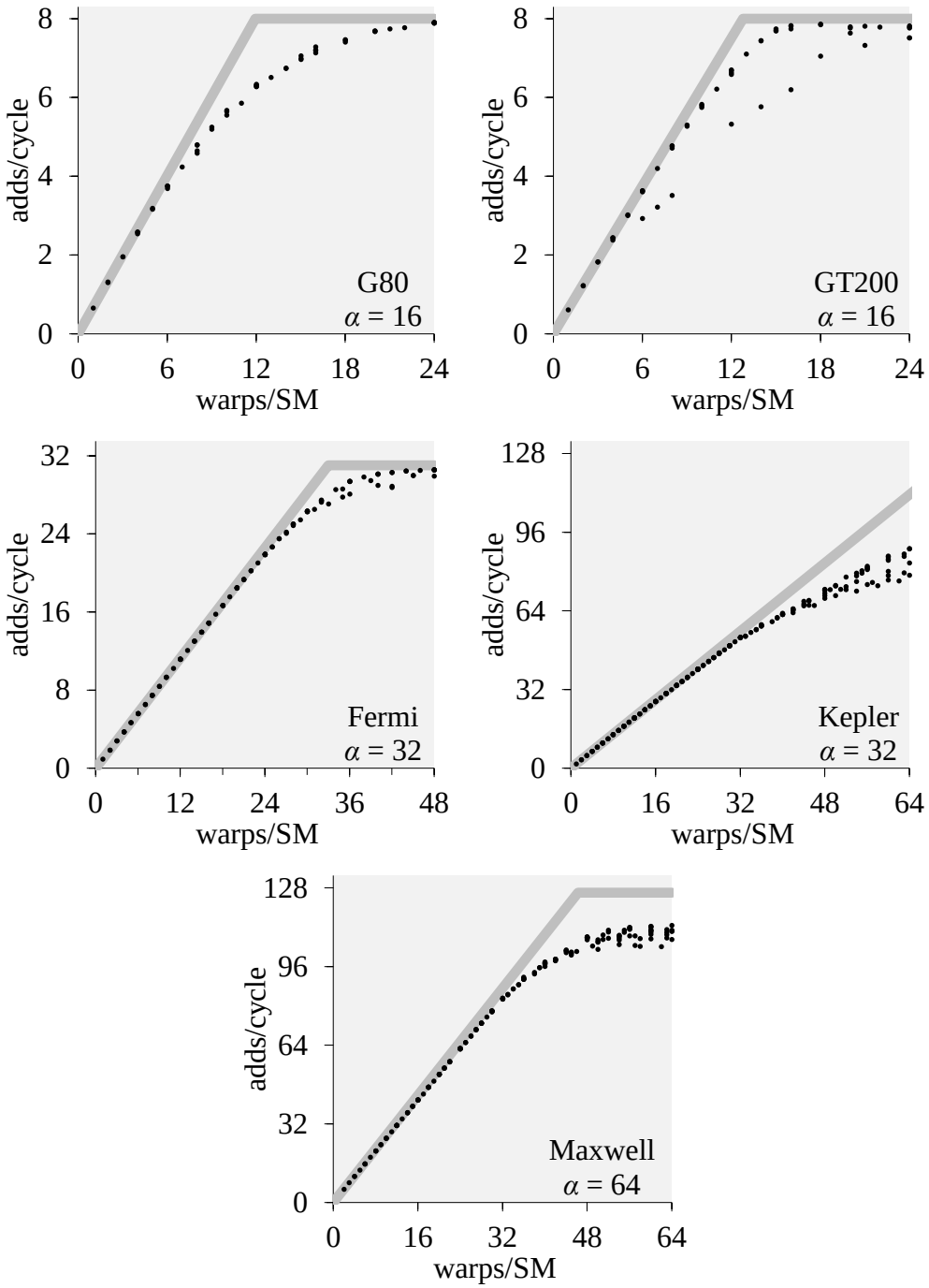


Figure 4.13: Our model is accurate at both small and large occupancies.

$$\textit{Time}\,(n) = \alpha \cdot \textit{arithmetic time}\,(n) + \textit{memory time}\,(n),$$

and arithmetic throughput is found as before. The new result is shown in Figure 4.12. The pattern is the opposite: in this variant, the model is accurate at small occupancies, where performance is latency-bound, but not at large occupancies, where performance is throughput-bound.

Two similar approaches, i.e. one with adding memory and arithmetic times and another with taking the maximum of them, were earlier discussed by Kothapalli et al. [2009]. However, their discussion was more simplistic, as they don't consider how these times depend on occupancy.

Our performance model can be understood as a combination of the two above approaches. Indeed, to find throughput in the latency-bound case we add instruction latencies, and to find throughput in the throughput-bound case we take the largest exclusive use time among different resources. As a result, our model is accurate at both small and large occupancies, as shown in Figure 4.13.

The figure also highlights some of the limitations of our model. First, the gradual saturation effect is not modeled. Second, saturation throughput on the Maxwell GPU is overestimated by about 12%. The latter may be due to similar reasons as in §4.7: issuing global memory access instructions might be taking more than a single issue cycle per instruction even when accesses are fully coalesced. The accuracy is improved if we assume that each memory instruction takes several cycles to issue; the best fit we found is 8 cycles. A further investigation is needed to better justify this assumption and find if this fit is accurate in general.

Separately considering latency-bound and throughput-bound cases is the key feature of our model and is in contrast to separately considering execution of arithmetic and memory instructions suggested in prior work.

4.13 Models with no memory bandwidth limits

Some of the previously published models, such as by Chen and Aamodt [2009] and Huang et al. [2014], also include a component similar to our latency bound. Yet, they don't include similar throughput bounds, in particular the bound due to the limited memory throughput.

Chen and Aamodt [2009] suggest four models for processors such as Sun's Niagara, which are not GPUs, but are also fine-grained multithreaded. None of these models consider memory bandwidth. Suppose that all executed threads are similar, as in our kernel, and λ is the throughput when executing one thread at a time. Then, their first three models can be summarized as $\lambda_1(n) = \lambda$, $\lambda_2(n) = n\lambda$, $\lambda_3(n) = 1 - (1 - \lambda)^n$, where $\lambda_1(n)$, $\lambda_2(n)$, and $\lambda_3(n)$ are the estimated throughputs when executing n threads at a time. The first model ignores any effect of multithreading. The second model is similar to our latency bound but with no throughput bounds. The third model includes one throughput bound: 1 IPC for instruction issue. Also, it features a model for the gradual saturation effect. Their fourth model is more complex, suggests no simple closed-form solutions, and still doesn't include memory bandwidth.

Huang et al. [2014] propose a GPU performance model where limits on memory bandwidth are not considered if all memory accesses are coalesced. They assume that "*in a balanced GPU design the resources used for normal operations are sufficient for each warp*". This means, for example, that if peak instruction issue throughput is 1 IPC per SM, then the number of CUDA cores per SM must be at least 32. This is a sufficiently realistic assumption as far as CUDA cores are concerned but not for memory system. For example, the GPU they consider has peak issue throughput equal 16 IPC per device, processor clock rate equal 1 GHz, and pin bandwidth equal 192 GB/s – all are realistic parameters similar to those on the Fermi GPU. But issuing coalesced memory access instructions at this issue rate amounts to attaining memory throughput equal 2048 GB/s, which exceeds pin bandwidth by an order of magnitude. The provided resources are therefore not sufficient, unlike assumed, and the respective throughput limits must hence also be considered. Memory bandwidth is similarly important on all five GPUs used in our work, as can be inferred

from Table 4.1. Indeed, in Chapter 3 we have already seen a number of kernels that have only coalesced memory accesses and yet are bound by peak memory throughput.

Memory bandwidth is also not considered in some of the early models for latency hiding, such as by Saavedra-Barrera et al. [1990] and Mowry et al. [1992]. This is in line with understanding latency hiding as keeping processor busy, not as saturating one or another bottleneck. In our performance model, in contrast, we consider several different throughput limits – and several different latencies.

Chapter 5
Experimental setup

In this chapter, we describe a novel experimental setup that is used to produce the experimental data shown throughout the text. It is based on the well-known pointer-chasing benchmark (see, e.g., Volkov and Demmel [2008] and Wong et al. [2010]), but includes a few substantially new features.

First, in contrast to executing one thread at a time, as done in the traditional pointer-chasing benchmark, we execute a number of threads at the same time. This requires a specific setup for memory addresses, which is discussed in §5.10. This feature allows studying memory latency under contention; the respective results are presented in Chapter 6.

Second, we mix memory access instructions with arithmetic instructions, which is to introduce an additional degree of freedom and match the workloads theoretically studied in Chapter 4. These instruction mixes are described in §§ 5.1–5.2. Arithmetic instructions are set to incur no register bank conflicts, as described in §5.3. The resulting experimental data is used in Chapters 4 and 7.

Another novel feature of our setup is tracking how occupancy varies during execution. This is explained in §5.6. A few found patterns of occupancy variance are discussed in §§ 5.8–5.9. The technique is also used to better control the maximum attained occupancy, as explained in §5.7.

In §5.4 we explain how we enforce that GPU clock rates are constant in each measurement. This is to leave dynamic frequency scaling out of scope.

Finally, in §5.11 we describe how to use a similar experimental setup with realistic kernels.

5.1 The primary instruction mix

A substantial part of this work is focused on understanding GPU performance when executing a simplistic kernel: a synthetic, highly controlled mix of arithmetic and memory instructions. The instructions used in the mix are selected to be the most common instructions. Memory access instructions correspond to what we call “streaming memory access”, in which case data is “streamed” from DRAM in the most efficient manner; the respective setup for memory accesses is described later in this chapter. For arithmetic instructions we use floating-point adds.

The reasons for using floating-point adds are the following. First, they are executed by CUDA cores, which execute most arithmetic instructions. In this respect floating-point adds are similar to other common arithmetic instructions – and are common instructions per se. Second, they cannot be executed by any other units but CUDA cores, unlike floating-point multiplies, which can be executed on some of the earlier GPUs by both CUDA cores and SFU units. Third, their performance features are relatively simple; for example, their theoretical peak throughput is one add per CUDA core per cycle on all GPUs. This is unlike some of the other arithmetic operations, such as integer multiply-adds, which have lower peak throughput on some GPUs [NVIDIA 2015, Ch. 5.4.1]. Fourth, add instructions have only two input operands, unlike multiply-add instructions. This reduces the likelihood of register bank conflicts, which are discussed later. Finally, sequences of floating-point adds are not collapsed by the CUDA compiler into fewer operations, unlike sequences of integer adds.

The resulting instructions mixes have the following form:

```
LD R1, [R1]
FADD R1, R1, R2
FADD R1, R1, R2
FADD R1, R1, R2
LD R1, [R1]
FADD R1, R1, R2
...
```

Here we use the fact that GPUs don't have separate floating-point registers, which allows transparently using floating-point results as memory addresses. Note, however, that G80 and GT200 GPUs do have separate registers for addressing shared memory. Creating a similar instruction mix but with shared memory accesses on these GPUs is thus not possible, which was earlier noted in §4.7.

The downside of using floating-point adds is the need to work around floating-point semantics. To ensure that floating-point results are valid memory addresses, and, moreover, are the intended memory addresses, we set up the adds to produce outputs identical to their inputs. First, we always add only zeros. Second, we never use memory addresses that correspond to denormalized numbers, such as the first 8 MB of address space. This is necessary on the G80 and GT200 GPUs, which flush denormalized numbers to zero. These two conditions suffice in practice. We implement a few safety checks to ensure that the behavior is as expected.

As explained in §4.1, adding zeros doesn't mean that each load instruction loads from the same address. Which memory locations are accessed depends on the memory contents, which are set up as explained in §5.10. Specifically, we ensure that there are no cache hits.

The repeating sequence of 1 memory load and α floating-point adds is wrapped up into a loop of about 500 to 1,000 instructions per iteration to avoid thrashing the instruction cache. The prologue and epilogue of each kernel are optimized down to 10–20 instructions each. The number of iterations, the number of thread blocks and other parameters are set as explained later in this chapter. The kernels are written in CUDA C and compiled for 32-bit platform. The resulting binary is disassembled using `cuobjdump` utility to ensure that the produced instructions are the same as expected. On the Maxwell GPU, we enforce that each global memory access is compiled into a single instruction by using inline assembly, instruction `ld.global.u32`. We use the default cache settings, which imply using both L1 and L2 caches when on the Fermi GPU. A number of different versions of CUDA Toolkit were used throughout this work, starting with version 5.5 to version 7.5.

5.2 Other instruction mixes

In §4.7, we briefly considered other instructions mixes, where global memory loads are replaced with shared memory loads and reciprocal square roots. The kernels that use reciprocal square roots are compiled using flag `-use_fast_math` to ensure that each use of the respective mathematical function (`rsqrtf`) is compiled into a single instruction. Shared memory accesses are configured to use the default addressing mode when on the Kepler GPU, which is the 32-bit mode.

All shared memory loads in all threads are set to access address zero to produce broadcast with no bank conflicts. When using reciprocal square roots, the iterated variable is set to 1. Latency and throughput metrics of the reciprocal square root instruction are found to be independent of whether input operand is trivial, i.e. equals 1, or not. Similarly, floating-point adds are found to have the same latency and throughput metrics independently of whether the added number is zero or a

nonzero. Finally, shared memory loads are found to have the same latency and peak throughput metrics independently of whether they correspond to broadcast or stride-1 access.

In Chapter 6 we consider another type of instruction mix: homogeneous instruction sequences. In this case all instructions in the mix (i.e. not including the auxiliary instructions such as loop control, timing, etc.) are the same: all floating-point adds, all global memory loads, all shared memory loads, or all reciprocal square roots. Otherwise, the experimental setup is the same.

5.3 Register bank conflicts

Our synthetic kernels heavily use add instructions with two register operands. Such instructions may have different latency and throughput metrics depending on register assignments, such as due to register bank conflicts. Lai and Seznec [2013] and Gray [2014] reverse engineered how registers are mapped to register banks on Kepler and Maxwell GPUs. We performed a similar study for all five GPUs used in this work. The results for the Kepler and Maxwell GPUs are in line with the earlier findings. The G80 and GT200 GPUs are found to expose no such differences in instruction metrics as long as the operands are not identical. On the Fermi GPU we find that instruction latency is 2 cycles longer if the operands are the same modulo 4, but peak throughputs are the same.

We disassemble the binaries to ensure that there are no register bank conflicts, and manipulate the source code to eliminate conflicts if they are found.

5.4 Dynamic frequency scaling

Some of the GPUs used in this work implement dynamic frequency scaling. It affects measurements but is orthogonal to the study. We therefore ensure that it is never effective. We use the following technique to ensure that the used clock rates are always the same and are the maximum supported.

We query GPU clock rates once in 10 microseconds in a separate CPU thread using NVAPI [NVIDIA 2014c]. If they are not the maximum supported, a signal is sent to the primary CPU thread using a shared variable. The primary CPU thread executes GPU kernels. When it receives the signal, it resets GPU processor clock rate to maximum, and, if the variable was set during an execution of a GPU kernel under study, the respective measurement is redone.

To increase GPU processor clock rate to maximum, we execute an arithmetically intensive kernel. If the clock rate does not increase to maximum after a few executions of this kernel, the device temperature must be too high; we wait until it cools down and retry.

To set GPU memory clock rate to maximum, we use a third party overclocking tool called NVIDIA Inspector[4]. It is needed only on the Maxwell GPU: other GPUs in the study always use the same memory clock rate when executing CUDA kernels. Using the tool, we manually set memory clock rate in performance level P2 to the advertised 3.5 GHz, which solves the problem.

5.5 Timing

Timing is done on the GPU side to avoid the overheads associated with CUDA API calls. Each warp is timestamped in the beginning and the end of its execution using the 32-bit clock register. Also recorded is the ID of the multiprocessor on which the warp is being executed. The timestamps and the ID are written to global memory when the warp terminates. The writing is done only in one thread of each warp to minimize the produced memory traffic. Given the large number of instructions executed in each warp, the associated overhead is assumed to be negligible.

[4] http://www.guru3d.com/files-details/nvidia-inspector-download.html

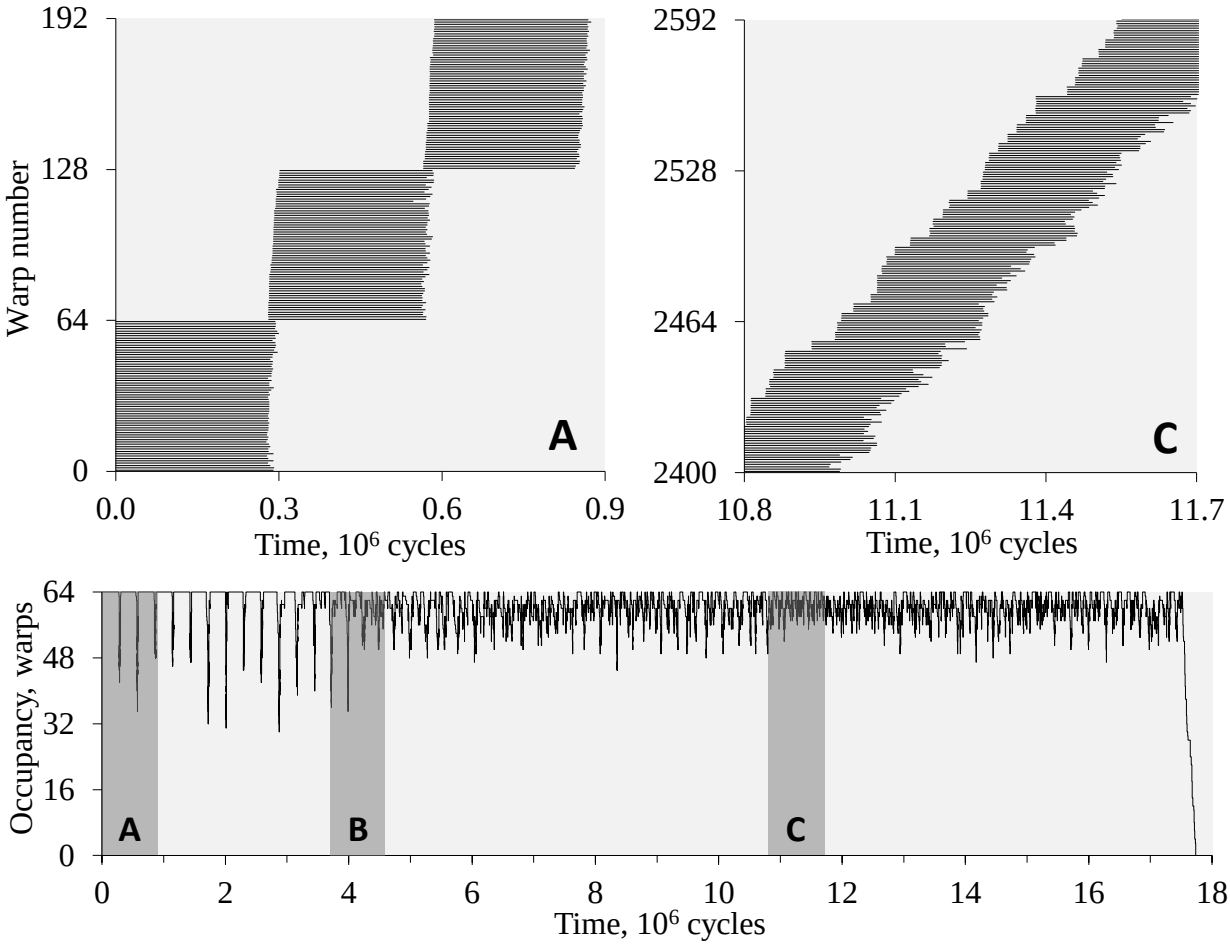


Figure 5.1: We record the start and the termination times for each warp. This allows tracking occupancy during the execution. The "B" case is detailed in Figure 3.3.

A few examples of the collected data are shown in Figures 5.1, top, and 3.3. These are three graphs, each corresponding to a different short interval in a particular execution of a kernel. Each plotted horizontal line corresponds to lifetime of an individual warp according to the recorded timestamps. Shown are only the warps that were executed on a particular SM.

Clock registers on different SMs are not necessarily synchronized. To address this difficulty, timestamps recorded on different SMs are processed separately. For each SM, we find the earliest and the latest of the timestamps. The difference between them gives the execution time on this particular SM. The largest of such execution times across all SMs is taken as the total execution time. It is used to find the attained throughput.

Also extracted for each execution is average warp latency, where individual warp latencies are found as differences between the respective timestamps. Average warp latency is used in Chapter 6 to find instruction latencies.

5.6 Occupancy tracking

The timestamps are also used to keep the concurrency under control. The recorded timestamps are sorted in increasing order, keeping track for each timestamp whether it corresponds to the start or the termination time of a warp. The sorted sequence is traversed from the beginning to the end to

find occupancy at each moment of time on each SM. An example is shown in Figure 5.1, bottom. It corresponds to the same dataset as shown in the same figure, top, and Figure 3.3. The correspondence is indicated using letters A, B and C. (The case "B" is shown in Figure 3.3.)

As the figure shows, occupancy may substantially vary during execution. Moreover, the character of variance may be substantially different at different times.

When execution of a kernel begins, each SM is assigned as many warps as possible given the resource constraints. In the shown case, as detailed in graph "A", the execution starts with assigning 64 warps, which is 16 thread blocks with 4 warps each. The warps start at about the same time and terminate also at about the same time. A new round of thread blocks is then assigned in their place. These thread blocks also start at about the same time, and so on. In this execution mode occupancy equals 64 warps most of the time, except during the short intervals when it drops nearly in half.

As execution proceeds, thread block start times gradually fall out of sync, and, eventually, none of the initial structure persists. In this, different execution mode, start times of new thread blocks are distributed more or less uniformly, as shown in graph "C". Occupancy attains the maximum of 64 warps only during relatively short periods and otherwise oscillates between approximately 48 and 64 warps. The transition between the two phases is shown in Figure 3.3. It corresponds to interval "B" in Figure 5.1.

5.7 Occupancy control

Occupancy is controlled by allocating shared memory. Such allocations limit the maximum number of thread blocks that can be executed at the same time at each SM; occupancy is limited by the product of this number and thread block size. The same occupancy, as a result, may often be achieved in several ways, using different thread block sizes.

When recording each throughput sample reported in this work for synthetic kernels, we ensure that target occupancy is attained at least once at each SM. This is done by using the occupancy tracking technique described above. Occupancy plotted on horizontal axis in graphs, such as in Figure 5.2, left, corresponds to this maximum attained occupancy.

The needed allocation sizes are found experimentally using the same occupancy tracking technique. The allocations are found to have substantial and sometimes unexpected granularity. For example, shared memory allocations on the Kepler GPU appear to be rounded up to multiples of 256 bytes. This may be expected as this is the width of shared memory interface: 32 banks, 8 bytes per bank [NVIDIA 2015]. On the G80 and GT200 GPUs, on the other hand, allocations are rounded to multiples of 512 bytes, which we don't know how to explain.

Due to the large granularity, shared memory allocations cannot be used to constrain occupancy to every physically supported number of thread blocks. For example, allocating 3072 bytes per thread block on the Kepler GPU constrains occupancy down to 16 thread blocks per SM. (This GPU has 49152 bytes of shared memory per SM in total.) But allocating 3073 bytes per thread block, i.e. only one byte more, constrains occupancy down to 14, not 15, thread blocks per SM. This is because the allocation size is rounded up to 3328 bytes. Similarly, occupancy cannot be constrained down to 7 blocks per SM on the G80 and GT200 GPUs. For a different reason we cannot constrain occupancy down to 1 thread block per SM on the Maxwell GPU: this GPU has 96 KB shared memory per SM, whereas maximum shared memory allocation per thread block is 48 KB.

On the G80 and GT200 GPUs we found additional overheads in shared memory allocations: a fixed 16-byte overhead per thread block, and a 4-byte overhead per kernel argument. These

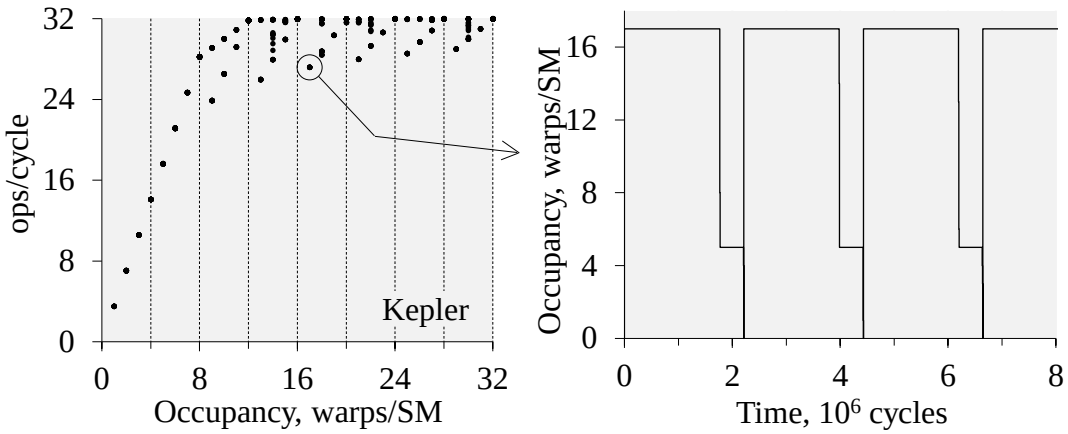


Figure 5.2: Occupancy may vary due to load imbalance.

overheads, although undocumented, pose no difficulty in our setup, because attained occupancy is always tracked using the timestamping technique.

It may be not necessary to constrain occupancy if using the persistent threads technique explained later. But this technique has other limitations, and we use it only on the GT200 GPU.

5.8 Occupancy oscillations

Shared memory allocations impose only an upper bound on occupancy. Otherwise, occupancy may vary, which may have a noticeably negative effect on throughput. Below, we consider a few cases of such negative effect.

One of the reasons for occupancy oscillations is load imbalance across warp schedulers. An example is shown in Figure 5.2. It corresponds to an execution of a synthetic kernel on the Kepler GPU, where the repeated instruction is reciprocal square root. The graph on the left shows the throughputs attained with different occupancy constraints. At larger occupancies we see a jigsaw pattern, which is characteristic of load imbalance. The same figure, right, details how occupancy varied in a particular execution, where it was constrained down to 17 warps per SM as 1 thread block per SM, 17 warps per thread block. Given that this GPU has 4 warp schedulers per SM, this results in load imbalance. As a result, some warps finish substantially sooner than others in the same thread block, and occupancy equals only 5 warps per SM during a substantial fraction of execution time. The overall throughput, as a result, is 15% less than the throughputs attained at smaller occupancies that involve no load imbalance, such as at 12 or 16 warps per SM.

Occupancy variance may also depend on thread block size. An example is shown in Figure 5.3, left. It includes several samples at occupancy equal 64 warps per SM. They correspond to different thread block sizes. Occupancy variance for two such samples is detailed in the same figure, right. According to the figure, using shorter thread blocks resulted in less variance and a larger throughput.

Occupancy variance is usually not taken into account in GPU performance modeling, and, when evaluating GPU performance models in Chapter 7, we don't use samples that are likely more compromised by occupancy variance. Specifically, we consider only the occupancies that correspond to integer numbers of warps per scheduler, and only the largest throughputs attained at each such occupancy.

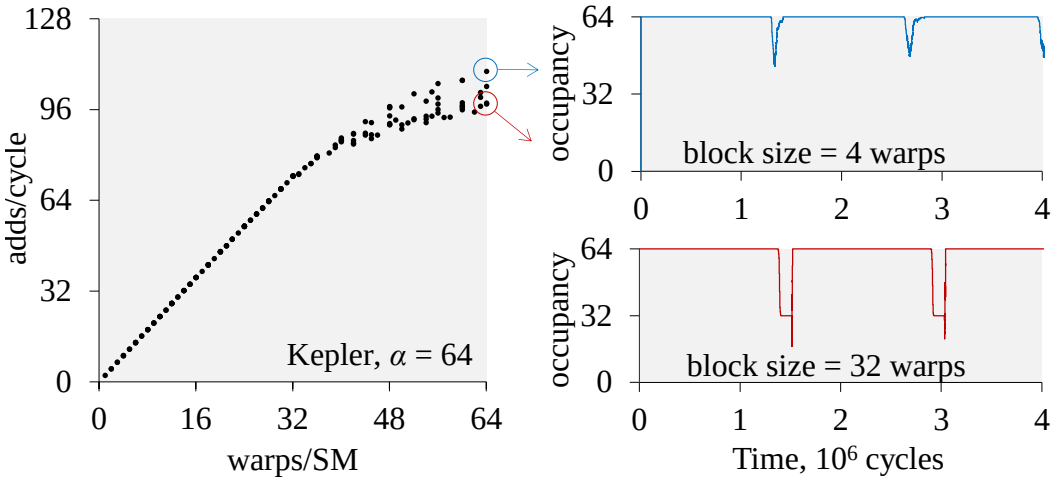


Figure 5.3: Throughput may depend on thread block size.

5.9 Persistent threads vs oversubscription

Occupancy variance may also explain why throughput may be different when executing the same instruction mix with the same occupancy, the same thread block size, but different numbers of launched thread blocks, and different numbers of iterations per warp.

The most common practice in GPU programming is to launch a large, application-dependent number of thread blocks, and do a conveniently small amount of work in each thread block. This typically results in oversubscribing the GPU, so that most of the launched thread blocks start executing only after others terminate. We use this approach on all but the GT200 GPU. The number of iterations per warp is set so that warp latency at minimal occupancy (i.e. latency that is due to register dependencies only) is about 10^6 cycles. If the instruction mix doesn't include global memory accesses, the total number of thread blocks is set to 100 times the number of thread blocks that can be executed at the same time. If global memory accesses are involved, the number of thread blocks is set depending on the size of the used array, as explained later.

Another practice is to use persistent threads [Aila and Laine 2009]. In this case we launch only as many thread blocks as can be executed at the same time and do more work per each thread block. For synthetic kernels, we set iteration count so that warp latency at minimal occupancy is about 10^8 cycles, except, again, if global memory accesses are involved. If all warps start at the same time and do the same amount of work, we may expect them to terminate also at about the same time. Indeed, this is what we see in Figure 5.1, top left. However, this is not necessarily the case.

An example is shown in Figure 5.4, left. It corresponds to an execution of a synthetic kernel on the G80 GPU. The repeated instruction is shared memory access. Thread block size is 10 warps and the maximum occupancy is 20 warps per SM. Warps start executing at about the same time and each executes the same number of instructions. Yet, their termination times are substantially different: about half of them terminate about two thirds through the execution. Moreover, during the last 7% of the execution time, occupancy was less than 9 warps, where 9 warps is the occupancy needed to hide shared memory latency as we find later in Chapter 6 (Table 6.3). If instead we execute the same instruction mix with the same occupancy constraints, the same thread blocks size, but using the oversubscription setup described above, occupancy also varies, but mostly

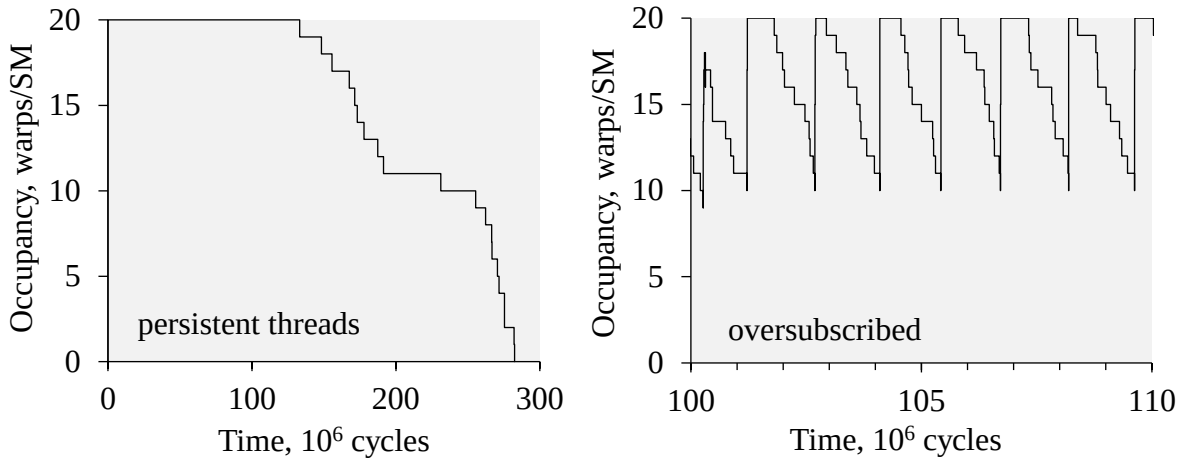


Figure 5.4: Using persistent threads may result in a more substantial occupancy deterioration.

between 10 and 20 warps per SM, as shown in the same figure, right. Occupancy is almost never too small, which results in a better overall throughput, larger by a few percent.

On the GT200 GPU, on contrary, using persistent threads results in a smaller occupancy variance and better throughputs. Therefore, we use persistent threads on the GT200 GPU and oversubscribe otherwise.

5.10 Memory access setup

An important part of our experimental setup is the setup of memory accesses. We discuss it last to highlight some of the difficulties that are more evident in throughput versus occupancy plots.

Our primary interest is what we call streaming memory access, which corresponds to reading a contiguous array once in the most efficient manner, such as if using only fully-coalesced accesses with no cache hits. This is one of the most common types of memory accesses on GPU.

Memory accesses occur in our synthetic kernels in a form of pointer chasing. Each thread starts at some memory address, which is used to read the following memory address, and so on. Therefore, there are two components of the setup: the start addresses and the array of pointers.

We use a 512MB pointer array on all five GPUs. Entry i of the array holds address of the $i+N$-th entry, where N is thread block size. Thread i of thread block j starts at array entry $i + j \cdot spacing$, where parameter *spacing* is selected to space start positions in different thread blocks sufficiently apart from each other; also, this parameter is aligned to multiples of 32. Each thread block, thus, sweeps through a contiguous section of the array. The number of thread blocks and the number of iterations are set to be sufficiently large to scan through the entire or the most of the array; also, we ensure that sections corresponding to different thread blocks do not overlap. If using persistent threads, the array size is used to set the number of iterations. Otherwise, it is used to set the number of thread blocks. When α is large, the resulting execution time may be too large, in which case we use fewer thread blocks or fewer iterations. Note that the pointer chain is not looped and no array entry is accessed twice in the same kernel execution, which is to avoid cache hits.

The resulting throughputs for case $\alpha = 0$ are plotted versus occupancy in Figure 5.5. In the graphs corresponding to the G80 and GT200 GPUs, samples are substantially scattered. We believe that this is due to TLB thrashing, which occurs when the number of concurrently executed thread blocks is too large, such as larger than the number of available TLB entries. The GT200 GPU has

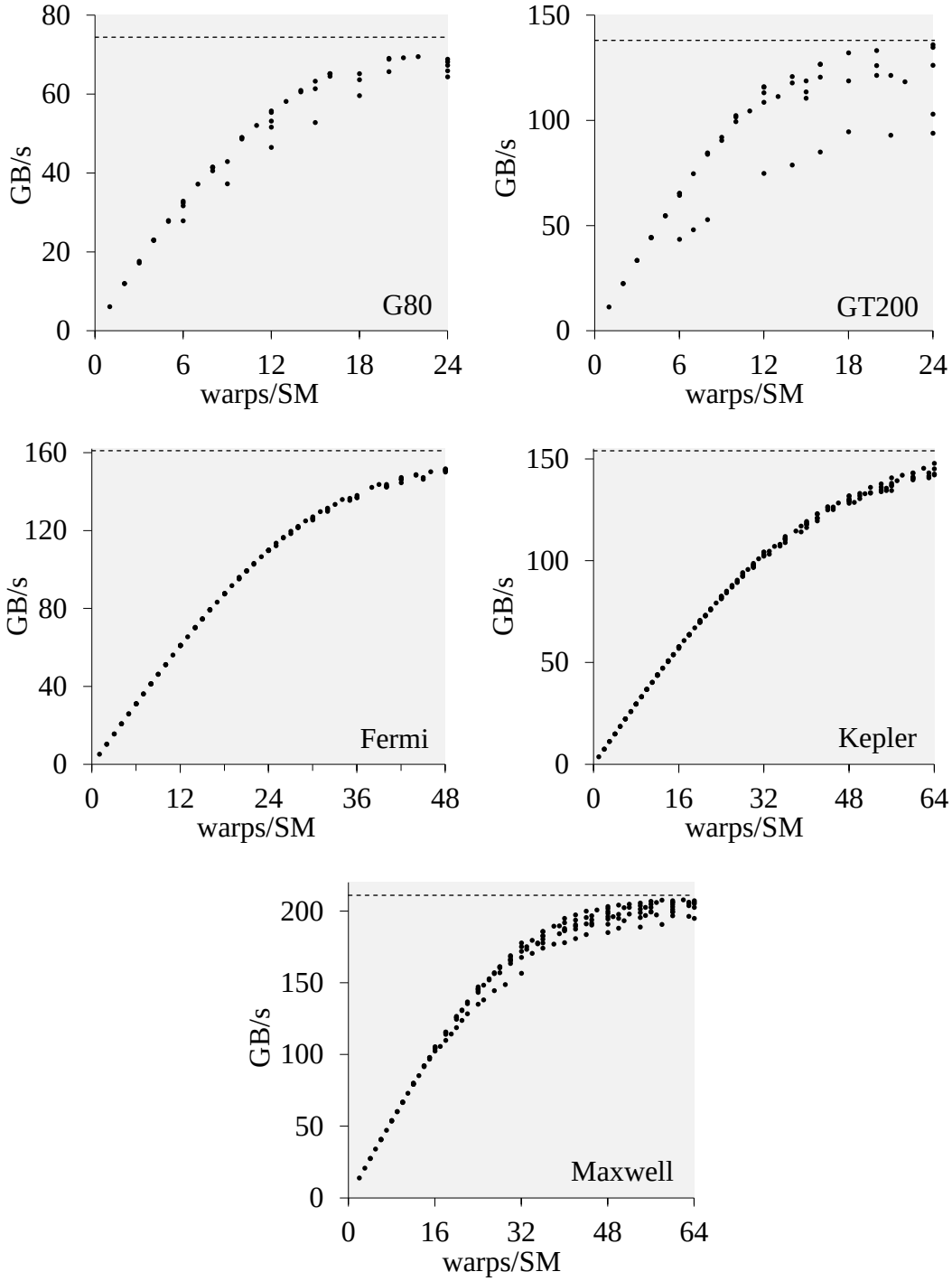


Figure 5.5: Throughputs attained in streaming memory accesses. Dashed lines indicate peak throughputs, which are found in §6.3.

more SMs, can run more thread blocks at a time, and therefore suffers more from this effect. We somewhat reduce the scatter by tuning parameter *spacing*. On these two GPUs, we find other effects that may be explained with TLB misses, as discussed later in §6.7.

We also experimented with another memory access setup, where stride in the pointer array (in words) is set to equal the total number of launched threads, and thread *i* starts at entry *i* of the array; *i* is the global thread index equal `threadIdx.x + blockIdx.x * blockDim.x`. On the Fermi and newer GPUs, both setups perform similarly. On the G80 and GT200 GPUs, this version produces more scatter in the data, perhaps due to more severe TLB thrashing, and therefore is not used.

5.11 Realistic kernels

We use elements of the above experimental setup when studying realistic kernels, such as the kernels discussed in Chapter 3 and shown in Figure 1.1. First of all, we use the same clock rate control. We cannot use the same timing, as this would require modifying the kernels. Therefore, we time execution on the CPU side. If possible, such as if the studied kernel doesn't overwrite its input data and there is no potential data reuse in cache, we time several back-to-back executions of the kernel to amortize overheads. To filter out noise, we often collect several throughput samples at each execution configuration and report the median.

Occupancy is set using the same shared memory allocations minus the static allocation in kernel, if any. We also check the number of used registers, which may also limit occupancy. Mostly for debugging purposes, for each kernel we also consider its modified version that does include timestamping. If the program is too short, such as in the case of the vector add considered in §3.8, warp latencies in the modified version are increased to a million cycles by using artificial delays. The modified versions of the kernels are expected to attain target occupancies according to the timestamps, except if other differences, such as in register use, make it not possible.

Chapter 6
Extracting hardware parameters

In this chapter, we consider simpler instruction mixes, in which all instructions are the same and back-to-back dependent. This is to extract basic hardware parameters, such as instruction latencies, peak instruction throughputs, and concurrencies needed to attain these peak throughputs. One of the contributions of this chapter is extraction of such hardware parameters for a few different instruction types and a few different GPUs. The results are summarized in Tables 6.1, 6.2 and 6.3.

Another contribution is a novel study of memory latency. In §6.4, we measure how mean memory latency depends on memory throughput and suggest an analytical model for this dependence. In §6.5, we explain how this latency model can be integrated into an overall performance model. In §6.6, we measure distributions of latencies of individual memory accesses at different levels of memory contention. In §6.7, we briefly consider non-coalesced memory accesses, and, in §6.2, shared memory accesses with bank conflicts.

Some of the found metrics are used in other chapters of this work. Unlike in other chapters, throughput and concurrency metrics here are quoted per scheduler, not per SM. This is to highlight that in this form they vary less across GPUs.

6.1 Arithmetic and shared memory instructions

First, we look at the performance metrics that describe execution of CUDA core instructions (floating-point adds), SFU instructions (reciprocal square roots) and shared memory instructions (loads). They are listed in the first three data columns of Tables 6.1, 6.2, and 6.3.

Table 6.1 lists the maximum attained throughputs. The units are ops per cycle per scheduler, where one op is one operation in one thread, and 32 ops/cycle is 1 IPC. The numbers are rounded to nearest integers, which introduces less than 1% change.

All found throughputs but one match hardware specifications. The exception is shared memory access on the G80 GPU. The specifications suggest that this throughput must be 8 ops/cycle per SM as there are 16 banks per SM and each bank is capable of fetching a new word every 2 cycles [NVIDIA 2010a]. In practice, we find that this throughput never exceeds 6 ops/cycle. A similar finding is reported in Volkov and Demmel [2008].

Another number that requires a commentary is the throughput of floating-point adds on the Kepler GPU. This GPU has 48 CUDA cores per scheduler, but the best recorded throughput is only 32 ops/cycle per scheduler. This is the expected number when no instruction-level parallelism (ILP) is present in the executed code: dual-issue in this case is not possible, and throughput is bound by warp schedulers, not CUDA cores.

Our best sustained throughputs approach theoretical peak numbers better than similar numbers reported in some of the prior work. For example, Zhang and Owens [2011] report sustaining only 84% of theoretical peak in executing CUDA core instructions on a GT200 GPU (Figure 2 in their paper), and Sim et al. [2012] report sustaining only 93% of peak throughput in executing CUDA core instructions on a Fermi GPU when not using ILP (Figure 6 in their paper). We attain over

GPU	Maximum throughput **per scheduler**, ops/cycle					
	add	*SFU*	*smem*	*smem2*	*stream*	*random*
G80	8	2	6	3	0.86	0.027
GT200	8	2	8	4	0.89	0.016
Fermi	16	2	8	4	0.97	0.029
Kepler	32*	8	8	4	1.07	0.034
Maxwell	32	8	8	4	0.65	0.023

* *larger if using instruction level parallelism*

Table 6.1: Peak instruction throughputs, 1 instruction = 32 ops.

GPU	Instruction latency, cycles					
	add	*SFU*	*smem*	*smem2*	*stream*	*random*
G80	20	32	38	54	444	644
GT200	24	34	40	56	434	662
Fermi	18	22	26	58	513	1571
Kepler	9	9	24	56	301	1213
Maxwell	6	13*	24*	26	368	534

* *if the dependent instruction is the same* (*see* §4.7)

Table 6.2: Register dependency latencies.

GPU	Warps needed per scheduler					
	add	*SFU*	*smem*	*smem2*	*stream**	*random**
G80	5	2	9	6	20	< 1
GT200	6	3	11	8	16	< 1
Fermi	9	1.5	6.5	7.5	21	2
Kepler	9	3	6	7	14	1.75
Maxwell	6	4	8	3.75	10	0.75

**at 90% of peak throughput*

Table 6.3: Concurrencies needed to attain peak throughputs.

99% of the respective theoretical peaks in all cases except the one described above: shared memory accesses on the G80 GPU. Our experimental setup is thus highly precise.

Table 6.2 lists the instruction latencies. They are found in the following manner. Mean warp latency is reported as a part of the experimental setup. It is divided by the number of instructions executed per warp. (We count only the repeated instructions, such as adds.) The results are accumulated across executions with different occupancies. The smallest such result is taken as instruction latency. All these latencies are within about 1% of the nearest integers and are rounded when listed in the table. These are the first three columns in Table 6.2.

The found latencies are in a minor disagreement with the latencies reported in prior work, such as in Wong et al. [2010] and Volkov and Demmel [2008]. Specifically, the latencies found on the G80 and GT200 GPUs are larger than the previously reported numbers by 6 cycles in the case of SFU instructions and by 2 cycles in the case of shared memory instructions. This is due to the difference in experimental methodologies. In the prior work, latencies were found by executing 1 thread at a time, whereas in this work they are found by executing 1 or more warps per SM. This

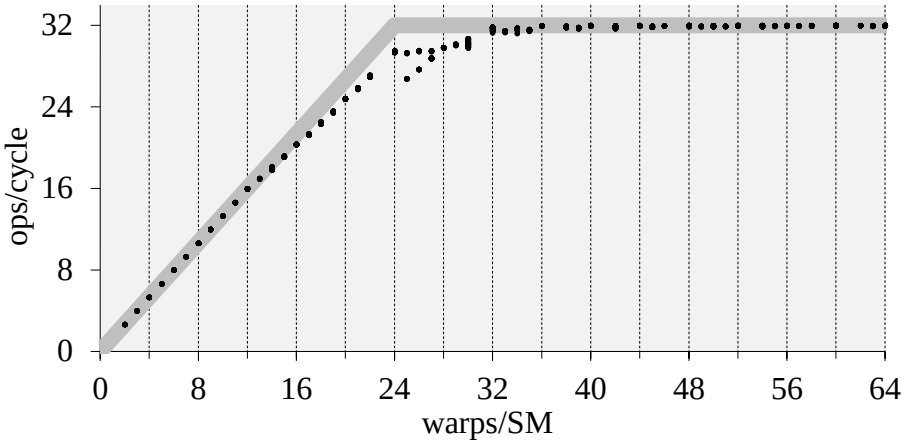


Figure 6.1: Throughput in shared memory accesses on the Maxwell GPU.

difference is important on G80 and GT200 GPUs, where each instruction is executed as two sets of 16 threads (half-warps) [Lindholm et al. 2008].

Mei and Chu [2016] report a larger, 28 cycle latency for shared memory accesses on a similar Maxwell GPU, GTX980. Their data correspond to accesses such as $j = A[j]$, which include additional pointer arithmetic. Latency of pointer arithmetic is likely the same as of floating-point adds, i.e. 6 cycles, as both are executed using CUDA cores. Latency of shared memory access instructions, as we found in §4.7, is different if the dependent instruction is a CUDA core instruction: 22 cycles, not 24 cycles listed in Table 6.2. Their result is therefore consistent with ours.

Mei and Chu report difficulties in approaching maximum theoretical throughput in shared memory accesses. Their best throughputs are only 33% to 84% of theoretical peak values. In contrast, we attain over 99% of theoretical peak values in shared memory accesses on Maxwell, Kepler, Fermi, and GT200 GPUs. Their reported lack of throughput is likely due to the experimental setup. They time execution of short instruction sequences, such as 8 reads followed by 8 writes in each warp[5], in which case attaining a better throughput is indeed difficult as GPUs are optimized for large workloads. We better eliminate numerous irrelevant factors and thus better expose key hardware features.

In Table 6.3, we list the smallest occupancies at which peak throughputs are attained, i.e. approached within a small threshold. They correspond to the needed instruction concurrencies. In most cases, the number is approximately equal to the respective instruction latency (Table 6.2) multiplied by the respective throughput (Table 6.1), divided by SIMD width (32), and, perhaps, also rounded up to an integer number of warps per SM, or an integer number of warps per scheduler. For example, latency of SFU instructions on the Kepler GPU is 9 cycles, their peak throughput is 8 ops/cycle per scheduler, so the required concurrency is 2.25 concurrent instructions per scheduler, or 9 concurrent instructions per SM. In practice, peak throughput is attained at 12 warps per SM, as shown in Figure 5.2, left.

In some cases, needed occupancy is different than expected, such as with shared memory accesses on the Maxwell GPU shown in Figure 6.1. Given the latency and throughput metrics, we expect attaining peak throughput at 24 warps per SM. Yet, despite this is an integer number of

[5] http://www.comp.hkbu.edu.hk/~chxw/Code/shared_bandwidth.cu

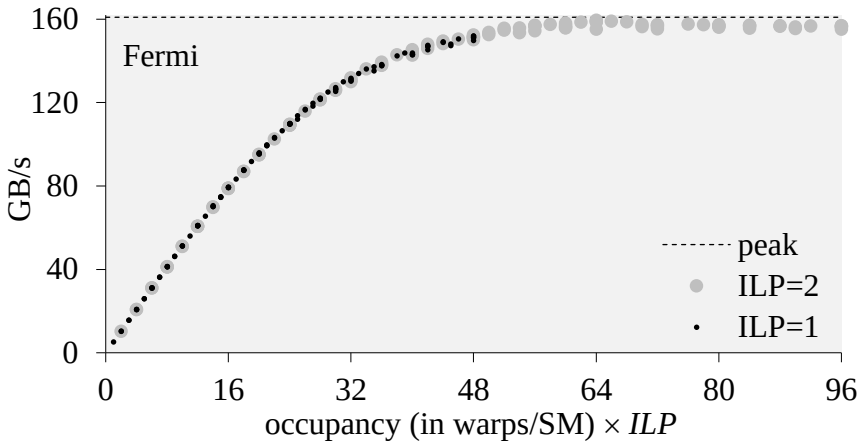


Figure 6.2: Instruction-level parallelism (ILP) often has a similar effect as thread-level parallelism.

warps per scheduler, executing at this occupancy results in attaining only 92% of peak throughput. A better fraction, over 99%, is attained at occupancy equal 32 warps per SM; this number is listed in the table. Wong et al. [2010, Figure 6] report a similar odd behavior with CUDA core instructions on a GT200 GPU. In both cases, this may be a feature of experimental setup.

6.2 Shared memory bank conflicts

Shared memory accesses perform differently when they cause bank conflicts. Peak throughput in this case is expected to be proportionally lower, such as by a factor of 2 if the conflicts are 2-way [NVIDIA 2015]. Expectations for other metrics are less clear.

We slightly modify the setup to cause 2-way bank conflicts in accesses; the scope is limited to this simplest case to build the most basic expectations. The results are reported in Tables 6.1, 6.2, and 6.3, column *smem2*.

As expected, peak instruction throughput drops by a factor of 2 on all five GPUs. Impact on latency, in contrast, varies. Instruction latency increases by 16 cycles on the G80 and the GT200 GPUs, by 32 cycles on the Fermi and the Kepler GPUs, and by only 2 cycles on the most recent Maxwell GPU. These are 8% to 133% increases. Similar results for 2-way bank conflicts are reported in Mei and Chu [2016]. They find 37 cycle, 35 cycle and 2 cycle latency increments on Fermi, Kepler and Maxwell GPUs respectively.

The impact on the needed occupancy, and, therefore, the needed instruction concurrency, also varies. On the Maxwell GPU, the needed occupancy drops by a factor of about 2: throughput falls by half, whereas latency changes only a little. On the G80 and the GT200 GPUs, the needed occupancy drops by factors of 1.4 to 1.5. On the Fermi and the Kepler GPUs, the needed occupancy, on contrary, increases: throughput, again, drops by half, but latency more than doubles.

Bank conflicts, thus, may both increase or decrease the instruction concurrency needed to attain peak throughput in shared memory accesses.

6.3 Streaming memory access

We are to find similar metrics for global memory accesses: mean latency of memory accesses, their best possible throughput, and the smallest occupancy where this throughput is attained. All of these metrics depend on memory access pattern. Below we consider streaming accesses, which are fully coalesced and miss in the cache, as described in §5.10.

Latencies of memory access instructions are listed in Table 6.2, column *stream*. They are found similarly to other instruction latencies, i.e. as the smallest warp latency per instruction observed across executions with all possible occupancies. The resulting numbers are similar to the latencies reported in prior work. For example, Wong et al. [2010] cite a similar 440 cycle latency on a similar GT200 GPU (GTX280). Volkov and Demmel [2008] cite 470 cycle latency on a similar G80 GPU (8800GTX), which includes 20 or 24 cycle latency of pointer arithmetic. Chu and Mei [2016] report 383 cycle memory latency on a similar Maxwell GPU (GTX980), which also includes the latency of pointer arithmetic – likely, 6 cycles. The methodologies used in the prior work are different. For example, in all of them latency is measured by executing 1 thread at a time, whereas we never execute less than 1 warp per SM at a time.

Finding peak throughputs requires an extra effort. Figure 5.5 shows the collected throughput samples plotted versus the respective occupancies for all five GPUs. The figure suggests that on some of the GPUs the best possible throughput might be not attained in this type of setup: throughput keeps increasing at around 100% occupancy as if an even larger occupancy, if it was supported, resulted in an even higher throughput.

To better estimate the best attainable throughput, we use a slightly modified experimental setup, where concurrency is amplified with instruction-level parallelism (ILP). In this case, instead of having all memory accesses back-to-back dependent, each access instruction is set to depend on the instruction *ILP* instructions before. For example, case *ILP* = 2 corresponds to the following sequence:

```
LD R1, [R1]
LD R2, [R2]
LD R1, [R1]
LD R2, [R2]
...
```

Pointer setup is also appropriately modified, so that the accessed memory locations are similar to those accessed in the original setup. We find that using a small *ILP* has a similar effect on throughput as using an *ILP*-times larger occupancy, as shown in Figure 6.2.

For each GPU, we find the best throughput attained over a number of kernels with different *ILP* parameters. The results are listed in Table 6.1, column *stream*, in Table 6.4, and plotted as a dashed line in Figure 5.5. These are our best estimates of peak memory throughputs for streaming 32-bit accesses. They equal 80 to 97% of the respective pin bandwidths.

Finally, we find the smallest occupancies where these peak throughputs are attained. We usually understand attaining a peak throughput as approaching it within a sufficiently small threshold. In this case, due to substantial gradual saturation effect, the result is highly sensitive to the magnitude of this threshold. We, therefore, use sufficiently large thresholds and report several such occupancy numbers for each GPU.

First, we estimate the needed occupancies using latency and throughput metrics. The estimates are found as usual, i.e. as products of instruction latencies listed in Table 6.2 and the respective peak throughputs listed in Table 6.1, divided by SIMD width. The results are listed in Table 6.4, column *linear*. In practice, only 75 to 84% of peak throughputs are attained at these occupancies, as also detailed in the table.

Second, we find the occupancies where 90 and 95% of peak throughputs are attained in the original setup, i.e. when not using ILP. These numbers are listed in the same table. On the Fermi GPU, 95% of peak throughput is never attained if not using ILP, in which case we list twice the

GPU	Peak, GB/s	Warps/scheduler needed			% of peak at *linear*
		linear	at 90%	at 95%	
G80	74	12	20	24	75%
GT200	138	12	16	18	84%
Fermi	161	15.5	21	25*	80%
Kepler	154	10	14	16	78%
Maxwell	211	7.5	10	11.5	80%

* *attained using 12.5 warps/scheduler in ILP = 2 setup*

Table 6.4: Occupancy needed to hide memory latency is characterized using several numbers.

occupancy where it is sustained in the *ILP* = 2 setup. These two cases, no ILP and *ILP* = 2 for the Fermi GPU, are shown in Figure 6.2.

According to these findings, sustaining 90 and 95% of peak throughput requires, respectively, about 1.3x and about 1.5x larger occupancy than suggested by the estimate. This characterizes how large the gradual saturation effect is. The numbers corresponding to the 90% threshold are listed in Table 6.3, column *stream*.

Mei and Chu [2016] report attaining only up to 156 GB/s on a similar Maxwell GPU. We consistently attain over 200 GB/s in this and other kernels, including such complicated kernels as Black-Scholes (see §3.10). This, again, reflects the quality of our experimental setup which is highly optimized to maximize throughput by minimizing irrelevant overheads. A particular factor important in this case is dynamic frequency scaling. Memory clock rate on the Maxwell GPU tends to drop to 3.0 GHz. We ensure it is always the maximum possible: 3.5 GHz. (This is if including the factor of 2 for the double data rate.) A similar technique would, perhaps, increase the result in Mei and Chu by the factor of 3.5/3.0, i.e. to 182 GB/s, which is within about 10% of our data. Note that Mei and Chu report peak throughput in copy, whereas the data in Table 6.4 is for reads only. Our best result for copy on this GPU, found using a different kernel executed in the same setup, is 199 GB/s.

6.4 Modeling memory contention

Memory latency, as other instruction latencies, is usually characterized with a single number. This number is usually understood as the round-trip time to DRAM when memory system is otherwise idle. This is also how memory latency is measured in prior work. However, when the memory system is not otherwise idle, latency may be different. By using our setup, we can measure how different memory latency is at different levels of memory access intensity.

As usual, we find memory latency as average warp latency per instruction. This time, however, we consider all latency samples as collected at different occupancies. They are plotted versus the respective attained throughputs in Figure 6.3. Dots show the experimental data, thick grey line shows the model introduced below.

According to the figure, memory latency is not always the same, but gradually increases with throughput and increases faster when throughput is large. Similar behavior is known in interconnection networks, as discussed, for example, in Dally and Towles [2003]. The GPU memory system includes a similar interconnection network, which may, at least partially, explain the latency increase.

Many models used in analysis of interconnection networks are based on queuing theory. Given a stream of incoming requests, such as memory transactions, they estimate the average delay experienced by these requests due to queuing for service, which is necessary if some requests

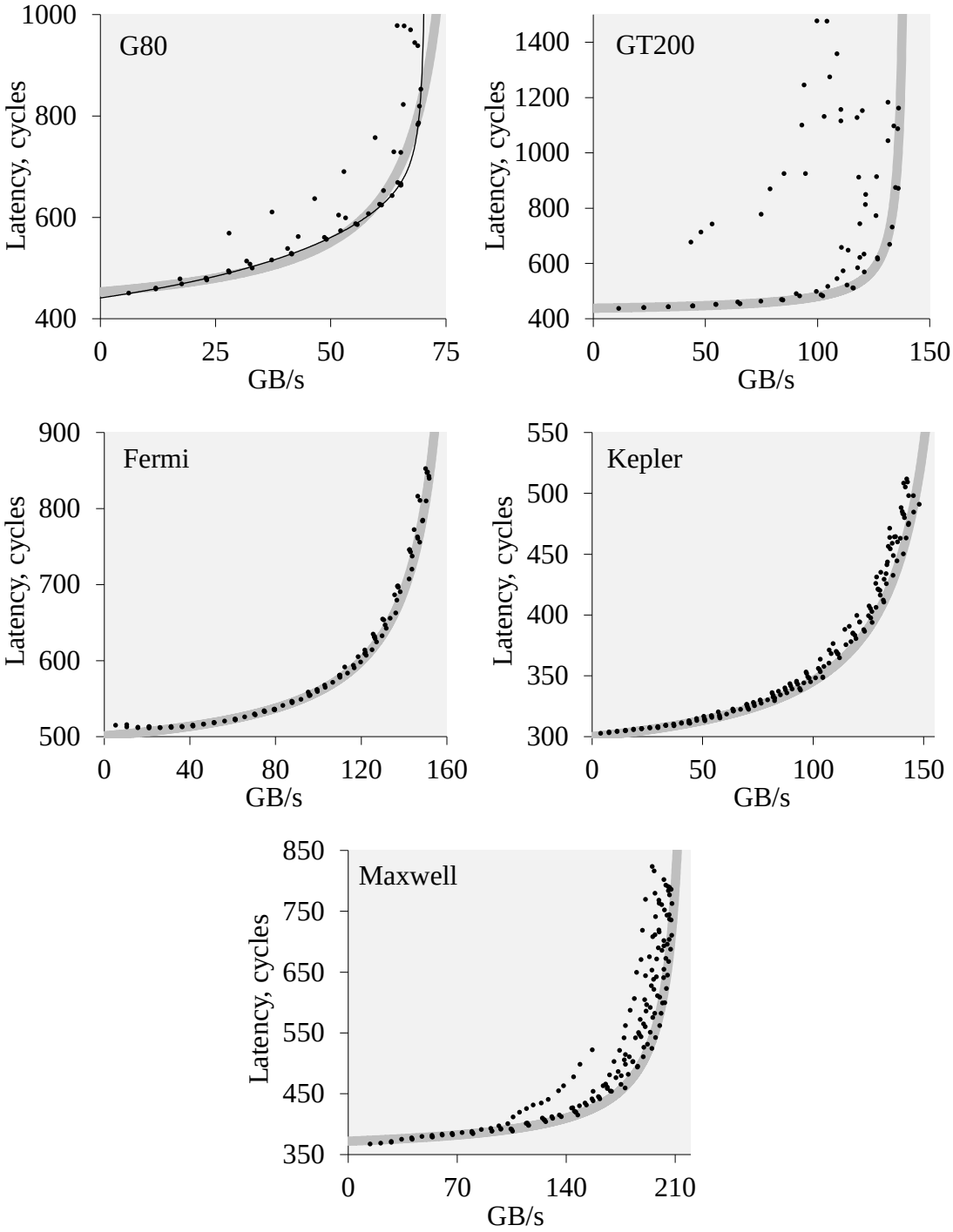


Figure 6.3: Mean memory latency increases with throughput.

arrive when previous requests are still being processed. Many of these models have a common feature: the delay is found to be proportional to $\rho/(1-\rho)$, where ρ is the sustained fraction of peak throughput. This is the case with such well-known queuing models as M/D/1 queue, M/M/1 queue, M/G/1 queue, and heavy traffic approximation for the more general G/G/1 queue. See Gross et al. [2008] for explanation and further details.

	G80	GT200	Fermi	Kepler	Maxwell
Latency, cycles	444	434	513	301	368
a, cycles	453	438	501	300	372
b, cycles	61	17	41	32	22
c, GB/s	81	140	170	170	221
Peak, GB/s	74	138	161	154	211

Table 6.5: Fitted coefficients *a*, *b*, and *c* for the memory latency model. Shown for comparison are latency and peak throughput metrics reported previously in Tables 6.1 and 6.4.

Motivated by this pattern, we fit our data using the following function:

$$Memory\ latency = a + b \cdot \frac{Throughput}{c - Throughput},$$

where *a*, *b*, and *c* are the fitting coefficients, and *Throughput* is the attained throughput in GB/s. Coefficients *a* and *c* are similar, respectively, to unloaded memory latency and peak memory throughput. Coefficient *b* is a new parameter that describes how quickly latency increases with throughput when throughput is small. Fitting is done numerically by using a trivial discrete optimization technique. We give preference to samples with higher throughputs to find an outer envelope to the data if the data is scattered. This is consistent with the approach taken elsewhere in this work.

The best fits are listed in Table 6.5 and shown in Figure 6.3. For comparison, the table also includes latency and peak throughput metrics reported earlier. According to the figure, the fits are reasonably tight. The fit for the G80 GPU is less tight; it can be further improved by introducing an additional term of similar form. Specifically, we find that model

$$Memory\ latency\ (\text{G80}) = 441 + 4 \cdot x / (71 - x) + 156 \cdot x / (121 - x),$$

where *x* is throughput in GB/s, fits the data's outer envelope better. This refined model is shown with thin black line in the same figure. These latency models can be used to refine the overall GPU performance model as discussed below.

Huang et al. [2014] similarly suggest using queuing theory to estimate delays in the GPU memory system. But, as we show in §§ 7.7–7.8, the delays they find may be negative. This has the following explanation. They find the queuing delay using ρ, find ρ using the arrival rate of memory requests, and find the arrival rate using memory latency. However, memory latency is assumed to be constant, whereas in practice it also depends on some factor, which is the queuing delay itself. This simplification results in larger arrival rates, larger ρ, and, in some cases, in ρ larger than 1, which implies negative queuing delays, and, possibly, negative execution times. This difficulty can be avoided as explained below.

6.5 Using the memory contention model

The above latency model can be easily integrated into the GPU performance model presented earlier. Below, we detail how this may be done in the case of the synthetic instruction mix considered in §4.1.

Originally, the latency bound was found as

$$Latency = mem\_lat + \alpha \cdot alu\_lat,$$

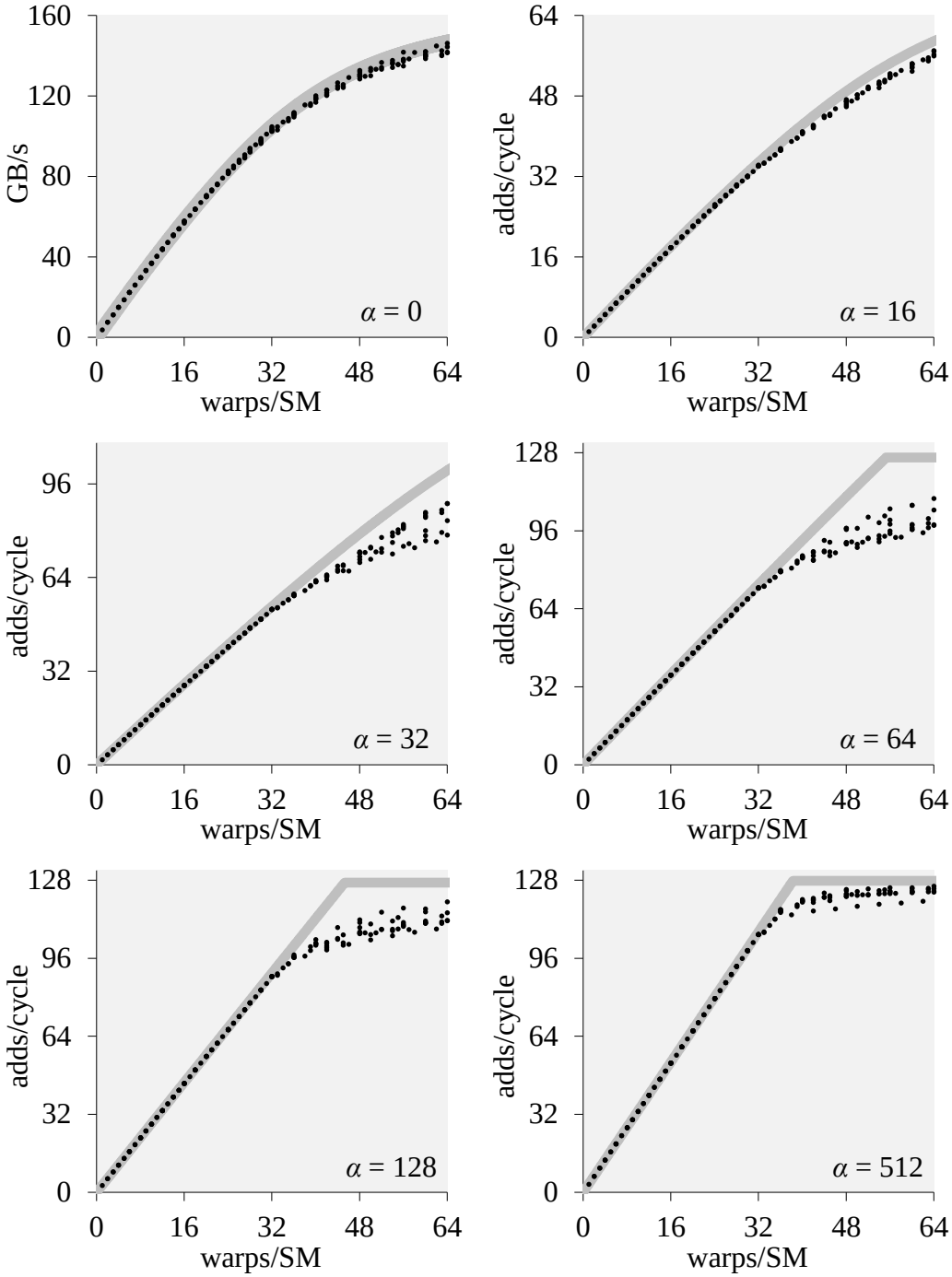


Figure 6.4: Performance of the improved model on the Kepler GPU.

where memory latency was assumed to be constant and equal *mem_lat*. We replace this constant with the new model for memory latency and get:

$$Latency = a + b \cdot \frac{\lambda}{c - \lambda} + \alpha \cdot alu\_lat,$$

where λ is memory throughput in GB/s. The rest of the model is the same; it concludes with the following solution for memory throughput:

$$\textit{Memory throughput} = \min\left(\frac{n}{\textit{Latency}}, \textit{mem\_thru}, \frac{\textit{alu\_thru}}{\alpha}, \frac{\textit{issue\_thru}}{\alpha+1}\right).$$

Memory throughput and λ describe the same quantity but in different units: IPC/SM in one case, and GB/s in another. They are, therefore, connected as:

$$\lambda = \textit{Memory throughput} \cdot 128\ \text{B} \cdot \textit{number of SMs} \cdot \textit{clock rate}.$$

This allows finding memory throughput given occupancy n. Arithmetic throughput in adds per cycle per SM is then found as *Memory throughput* $\cdot\ \alpha \cdot 32$.

For the Kepler GPU this model reduces to solving the following equation, where *Memory throughput* is denoted as x for short:

$$x = \min\left(\frac{n}{300 + 32 \cdot x\ /\ (0.1477 - x) + 9 \cdot \alpha}, 0.1338, \frac{4}{\alpha + 1}\right).$$

This is to replace the solution found in §4.1, which is

$$x = \min\left(\frac{n}{301 + 9 \cdot \alpha}, 0.1338, \frac{4}{\alpha + 1}\right).$$

The equation can be solved analytically, but we solve it numerically using bisection. This allows using the same solver with more complicated latency models, such as in the case of the G80 GPU.

The result is compared with experiment in Figure 6.4. Experimental data is shown as dots, and the model is shown as a thick grey line. The new solution is accurate at small and large α, but is less accurate at intermediate α, such as 64 or 128. This suggests that increase in memory latency is not the only cause of gradual saturation effect.

This model is evaluated in additional detail in §7.9. Also, it is used to produce the data shown as a thick grey line in Figure 4.6, as was discussed in §4.5.

6.6 Latency of individual memory accesses

Memory latencies discussed so far are mean latencies, which is implied by how they are found. Latencies of individual accesses are not necessarily the same. A minor modification of our setup, however, allows extracting latencies of individual accesses as well.

We want to measure latency of individual accesses with a minimal impact on performance. In order to do so, we time individual instructions in only 1 of the executed warps and only 1 memory access instruction out of each 8. The latency values are accumulated in shared memory and are flushed to DRAM when the warp terminates, so that the flushing does not affect the timing. We run the same kernel a large number of times to accumulate about 10 million samples.

We restrict these extended measurements to the Kepler GPU, and only to a few of the many execution configurations shown earlier in Figure 6.3. The results are presented in Figure 6.5 and Table 6.6. The table lists the minimum and the maximum latency samples, and percentiles that divide this range. The same percentile quantities are plotted in Figure 6.5, top left. The curve in

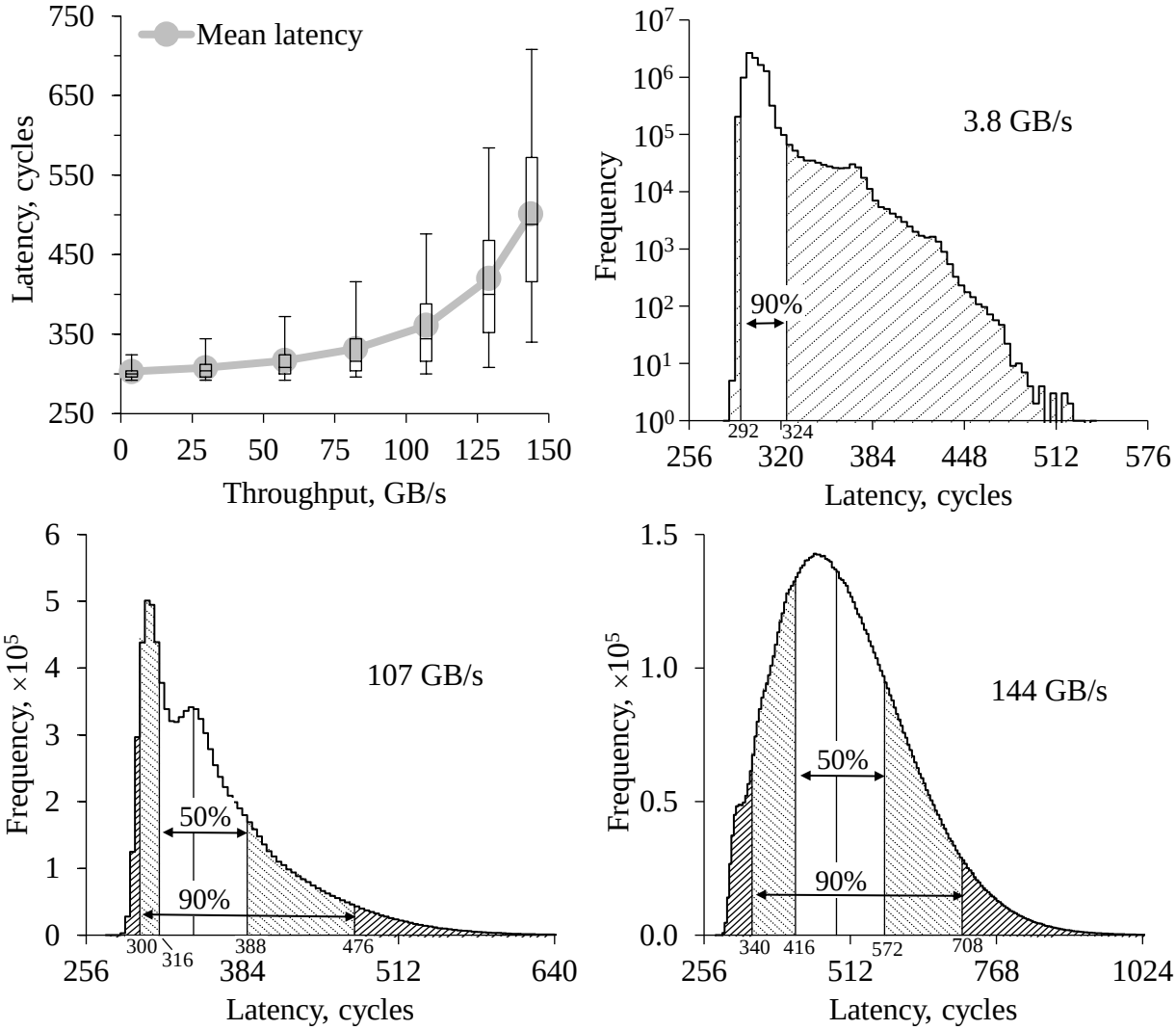


Figure 6.5: Distribution of individual memory access latencies on the Kepler GPU.

the graph is shown for comparison – these are the mean latencies reported before. They are not substantially different than median latencies.

The data shows that individual memory latencies may substantially differ from their averages, especially when throughput is large. When throughput is 94% of peak throughput, average latency is 501 cycles, and yet some of the accesses took 1453 and 278 cycles. These are extreme, unlikely values, each represented with a single sample. But even if we discard top 5% and bottom 5% of all samples, the rest is still spread across a wide range between 340 to 708 cycles. This distribution is shown in more detail in Figure 6.5, bottom right; it resembles the familiar bell curve. The area under the curve is divided to highlight the percentiles.

The distribution is narrower when memory traffic is not as intense. If throughput is only 2.5% of peak throughput, the middle 90% of samples fit within a 32-cycle range, which is small compared to 303 cycle average latency. Yet, at least once, latency was 536 cycles, which is 1.8 times the average. The respective distribution density is shown in Figure 6.5, top right. This time it resembles a shifted exponential distribution. Only 5-th and 95-th percentiles are shown in this figure.

GB/s	Latency, cycles (percentiles)						
	min	5th	25th	50th	75th	95th	max
3.8	280	292	296	300	304	324	536
30	272	292	296	304	312	344	588
58	272	292	300	308	324	372	680
82	272	296	304	316	344	416	756
107	275	300	316	344	388	476	961
129	275	308	352	400	468	584	1331
144	278	340	416	488	572	708	1453

Table 6.6: Distribution of individual memory access latencies on the Kepler GPU.

Figure 6.5, bottom left, shows an intermediate case, where throughput is 69% of peak throughput. The distribution has a more complicated structure with two local maxima.

To summarize, latencies of individual memory accesses may substantially differ from their mean, especially when memory traffic is intense.

6.7 Unstructured memory access

For comparison, we also look into performance of unstructured memory accesses, where each thread accesses what approximates a random memory location. Each memory access instruction in this case fully diverges, i.e. splits into the maximum number of memory transactions (32) and thus maximally consumes memory bandwidth. Instruction throughput is then about the smallest possible and instruction latency is about the largest possible. In this sense, unstructured memory access is opposite to streaming memory accesses.

We modify the setup by using a differently initialized pointer array and different starting pointers. The starting pointers are set with a sufficiently large stride, so that the first memory access in every warp is also fully divergent. The pointers in the array are initialized so that the resulting pointer chain connects all array entries into a single loop in a pseudo-random order.

The best attained throughput substantially depends on array size, as shown in Figure 6.6. (The throughputs are found by counting each thread access as 4 bytes.) Some of the features in the plotted data are due to caching. For example, the single major feature in the graphs for the Maxwell and Kepler GPUs is due to caching in L2 cache. G80 and GT200 GPUs don't cache memory accesses unless they are specifically set up to use texture caches – which is not the case in our setup. Yet, their graphs show similar memory hierarchy effects: throughput deteriorates at larger array sizes by up to about 25x in one case, and 100x in another case. This is likely due to the hierarchy of TLBs, which implements address translation, and is consistent with the analysis in Wong et al. [2010]. This analysis suggests that GTX280 GPUs have two-level TLB with the second level covering 32 MB of address space. In line with this, we see a substantial drop in throughput on the GT200 GPU, which is also a GTX280 GPU, when array size exceeds 32 MB.

Data for the Fermi, Maxwell and Kepler GPUs show a moderate, 20–30% throughput deterioration at 512 MB array size, which we cannot explain with caching. This deterioration may also be due to TLBs. We didn't use larger array sizes, or sizes other than powers of two.

For the limited purpose of this study, we want to characterize unstructured memory accesses with a single set of metrics that correspond to a generic or common case. For this purpose, for each GPU, we pick a size that neither involves substantial caching nor substantial TLB thrashing and thus, in some respect, represents the simplest and the most basic case. These choices are shown in Figure 6.6 with vertical bars.

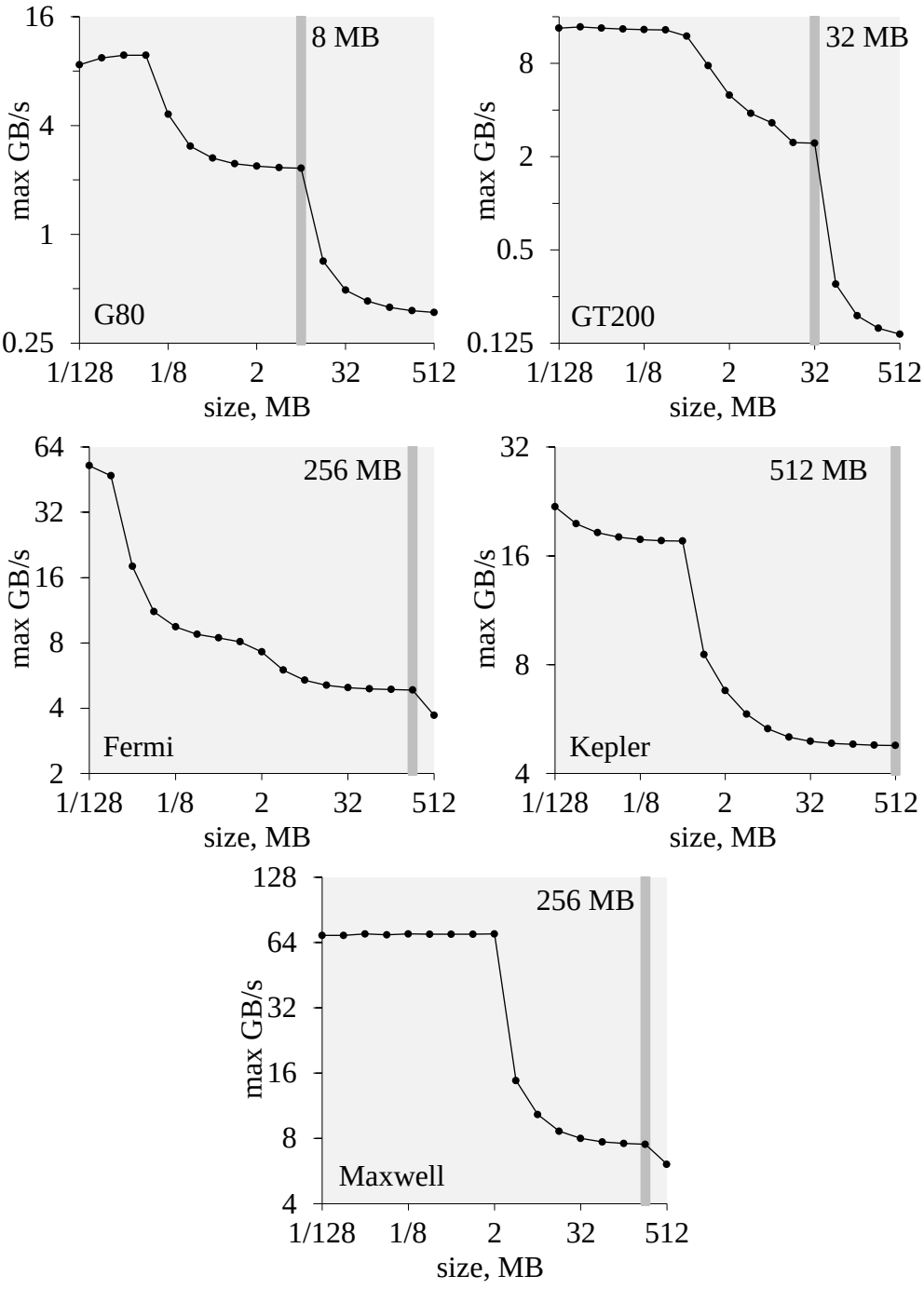


Figure 6.6: Throughput in unstructured memory accesses strongly depends on the size of the dataset.

Using these array sizes, we measure the same three metrics as before: register dependency latency, the best attained throughput, and the smallest occupancy where 90% of the best throughput is attained. The results are listed in Tables 6.1, 6.2, and 6.3, column *random*.

Although the array sizes we selected don't correspond to the lowest throughputs in the graphs, the best sustained throughputs on the G80 and GT200 GPUs are so low that they are attained with

GPU	G80	GT200	Fermi	Kepler	Maxwell
Latency, cycles	6.7	7.6	34	33	5.9

Table 6.7: Latency of issuing an additional transaction in a fully diverging memory access.

less than 1 warp per SM, such as 17 warps per 30 SMs on the GT200 GPU. With this in mind, we measure unloaded latency using 1 warp per device, not 1 warp per SM as we do otherwise.

As listed in Table 6.1, peak throughputs in unstructured accesses are 28 to 33 times smaller than in streaming accesses on all but the GT200 GPU, where the difference is 56x. We, however, expect only an 8x difference on all but the Fermi GPU. These GPUs use 32-byte transactions in fully diverging accesses [NVIDIA 2010a; NVIDIA 2015], in which case for each requested word (4 bytes), memory system transfers a 32-byte block, which suggests the 8x factor. The result is approximately similar to as if all transactions were never shorter than about 128 bytes. A further analysis is needed to explain this behavior.

Finally, we look at instruction latencies. They are longer than latencies in streaming memory accesses due to issuing additional memory transactions. To find the latency increment per transaction, we assume that each non-coalesced access instruction requires 32 transactions, and each coalesced access instruction requires 2 transactions if on the G80 and GT200 GPUs, 1 transaction if on the Fermi GPU, and 4 transactions if on the Kepler and Maxwell GPUs; all of this is according to the CUDA C Programming Guide. The result is listed in Table 6.7. Latency increment per transaction is found to be about 6 to 8 cycles on the G80, GT200 and Maxwell GPUs, and 33 or 34 cycles on the Fermi and Kepler GPUs.

Chapter 7
Evaluation of prior models

The main contribution of this chapter is an evaluation of a number of well-known GPU performance models using the simplistic workload introduced in Chapter 4. Surprisingly, we find that all of them have substantial limitations. The models by Hong and Kim [2009], Sim et al. [2012], Zhang and Owens [2011] and Baghsorkhi et al. [2010] are found to substantially overestimate throughput in latency-bound case, such as by up to a factor of at least 1.7; this occurs at intermediate arithmetic intensities. The model by Huang et al. [2014] is found to substantially overestimate throughput in throughput-bound case, such as, on some GPUs, by a factor of 2 and above. This occurs at small arithmetic intensities. In one version of this model, this also occurs at large arithmetic intensities. In some cases, this model is found to produce negative execution times. The considered prior models are found to show no cusp behavior.

The new model presented in §4.1 is found to have better accuracy: it overestimates throughput by a smaller factor, such as up to 1.28 on all but the G80 GPU, and up to 1.34 on the G80 GPU. This error is mostly due to the unaccounted gradual saturation effect. We also briefly consider the refined model suggested in §6.5, which takes into account memory contention. The refined model is found to be even more accurate.

To disencumber the algebra, we consider execution times per the repeating group of $\alpha + 1$ instructions, keeping in mind that this group is repeated many times in each warp. Unless specially noted, n denotes occupancy in warps per SM. When comparing estimates with experimental data, we consider only the largest throughput sample at each occupancy and only the occupancies that correspond to integer numbers of warps per scheduler. This is to compensate for various unaccounted effects, such as those discussed in §5.8.

7.1 Hong and Kim 2009 model

Hong and Kim [2009] model is one of the better known performance models for GPUs. We evaluate it on the GTX280 GPU, the same as one of the GPUs used in the original work.

One of the basic limitations of this model is that it ignores arithmetic latency. Therefore, it cannot possibly be accurate in the case when all executed instructions are adds ($\alpha = \infty$): the model suggests a flat throughput of 4 CPI at all occupancies, whereas the throughput attained in experiments is substantially lower when occupancy is small – down to 24 CPI. On the other hand, in the case when all instructions are memory loads ($\alpha = 0$), the model is similar to and is as accurate as the respective basic model discussed in §4.2. Below we consider the case of an intermediate arithmetic intensity, $\alpha = 32$.

The model starts with computing metrics *MWP* and *CWP*. *MWP* is the maximum number of warps per SM that may access memory at the same time. It equals n when n is small, but otherwise is bound by MWP_{peak_bw}, which is the number of warps needed to attain peak memory throughput. MWP_{peak_bw} is found by using what we may recognize as Little's law: these are Eq. 6 and 7 in the Hong and Kim paper. Given 141.7 GB/s pin bandwidth, 1.296 GHz clock rate, 30 SMs, 128 bytes accessed per warp, and 434 cycle memory latency (here we use the number found in Chapter 6), we find that attaining memory throughput equal to pin bandwidth requires executing on average

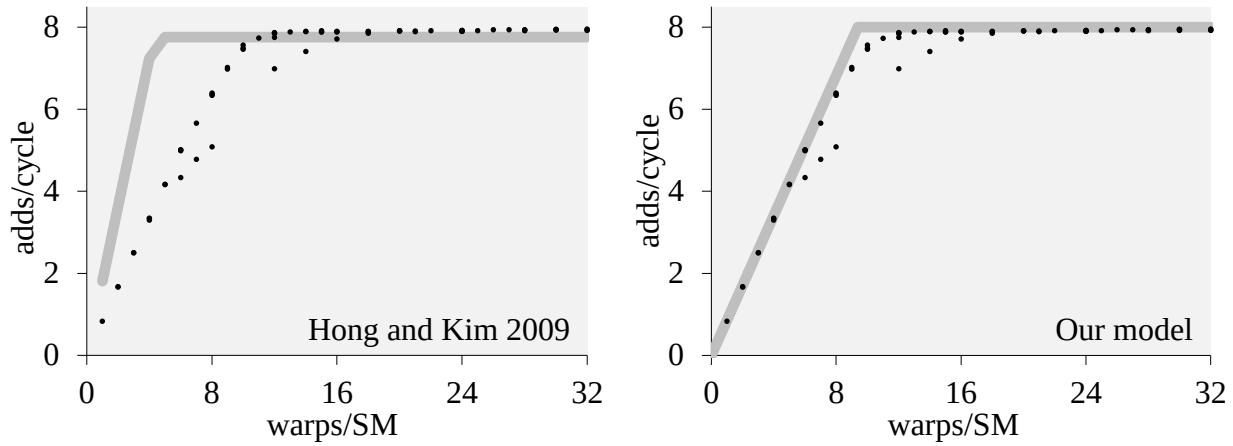


Figure 7.1: Hong and Kim [2009] and our model compared with experiment on the GT200 GPU, $\alpha = 32$.

about 12.4 memory access instructions per SM at the same time. Hong and Kim interpret this number as the needed number of warps, i.e. as 12.4 warps per SM, which is misleading, as discussed in §3.3. So far we have:

$$MWP = \min(\, n, 12.4 \,).$$

CWP is the number of computational periods, i.e. groups of α arithmetic instructions, that can be executed when waiting for one memory access, plus one (Eq. 8 and 9 in Hong and Kim [2009]). We have already discussed this metric in §4.9. Given 434 cycle memory access latency, CUDA core throughput of 4 CPI at each SM, and arithmetic intensity 32, we get:

$$CWP = \min(\, n, 4.3 \,).$$

CWP and *MWP* metrics are used to differentiate three execution modes, which may be recognized as one latency-bound mode and two throughput-bound modes: memory-bound and compute-bound.

Latency bound mode corresponds to small occupancies where $CWP = MWP = n$. In our case this is when $n \leq 4.3$. The total execution time per the repeating group of $\alpha+1$ instructions is then (Eq. 22 in their paper):

$$Time\ (n \leq 4.3) = 434 + 4 \cdot (\alpha + 1) + \varepsilon_1,$$

where ε_1 is negligibly small if the number of instructions in each warp is very large. This can be converted to arithmetic throughput in adds per cycle per SM as:

$$Throughput\ (\text{adds/cycle}) = 32 \cdot n \cdot \alpha \,/\, Time,$$

where 32 is the instruction SIMD width. Since *Time* does not depend on *n*, throughput grows linearly with *n*, which is what we usually expect in latency-bound mode.

Larger occupancies correspond to throughput-bound mode. If *MWP* is less than *CWP*, throughput is assumed to be bound by memory system, and otherwise, by CUDA cores. In our case *CWP* is smaller: it equals 4.3 when $n \geq 4.3$, whereas *MWP* keeps increasing up to 12.4.

Therefore, throughput is bound by CUDA cores, and all instructions are executed at the rate equal 4 CPI (Eq. 24 in the paper):

$$Time\ (n \geq 4.3) = 4 \cdot n \cdot (\alpha + 1) + \varepsilon_2,$$

where ε_2 is another term that is negligibly small in our case.

The result is plotted in Figure 7.1, left. Our model is shown for comparison in the same figure, right. At this arithmetic intensity Hong and Kim model overestimates throughput by a factor of up to 2.1, but only at small occupancies. At occupancy 10 warps and above, the model is highly accurate. In contrast, our model is within 2% of the experimental data everywhere but at the knee, where the error is up to 9%. This is if considering only the largest throughput samples at each occupancy.

This substantial lack of accuracy is due to ignoring arithmetic latency. Arithmetic latency can be introduced into the model by replacing estimate for the latency-bound case with

$$Time\ (n \leq 4.3) = 434 + 24 \cdot \alpha,$$

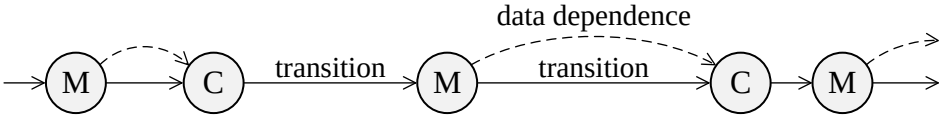
where 24 is arithmetic latency in cycles. This reduces throughputs at small occupancies by the needed factor of 2.1. Additional modifications are needed to correctly separate latency- and throughput-bound cases.

7.2 Baghsorkhi et al. 2010 model

The model by Baghsorkhi et al. [2010] is the first performance model for GPUs to include both arithmetic and memory latencies. We find it accurate when all executed instructions are arithmetic instructions ($\alpha = \infty$), but not when all executed instructions are memory accesses ($\alpha = 0$). In this respect it is opposite of the Hong and Kim model. We evaluate it on the 8800GTX GPU, the same as used in the original paper.

In this model we find "latencies" of individual operations and combine them into "latency" of entire warp. These "latencies", however, are understood differently than elsewhere in this text. We therefore put them in quotes.

To keep track of "latencies" of different instructions, this model employs an annotated program dependence graph – we adopted a similar graph in §3.8, Figure 3.5. Each memory instruction is represented with a node *M*, and each sequence of arithmetic instructions is represented with a node *C*. The nodes are connected in program order using edges called transition arcs. Each arc is assigned a weight equal to the "latency" of the instructions represented by the source node. There is a second type of edge: data dependence arcs. They represent stalls on memory accesses. Each data dependence arc connects the respective memory node with the node that uses the fetched data. In our case, it is always the next node. Thus, the graph for our workload has the following structure:



Each compute node corresponds to α add instructions. As we found in Chapter 6, these instructions on the G80 GPU have peak throughput equal to 4 CPI at each SM and register dependency latency equal to 20 cycles. If using these metrics, the "latency" of each arithmetic instruction is (see Eq. 3 and the following passage in the original paper; "*alu*" is our notation):

$$alu = \max(\, 4,\, 20 / n \,).$$

This number, multiplied by α, is assigned to each transition arc that starts at a compute node. Transition arcs for memory nodes are temporarily assigned 4 cycle weights, where 4 cycles is understood as a generic issue latency.

Summing the weights assigned so far produces "the number of compute cycles" $CYC_{compute}$. It equals, per the repeating group of $\alpha + 1$ instructions,

$$\frac{CYC_{compute}}{NUM_{mem}} = \alpha \cdot \max\,(\, 4,\, 20 / n \,) + 4,$$

where NUM_{mem} is the number of memory instructions per warp.

The next step is to find the number of cycles CYC_{mem} needed to transfer the requested data from memory at peak memory throughput. Peak memory throughput per SM on this GPU is 0.86 ops/cycle or 37 CPI (Table 6.1). Therefore:

$$\frac{CYC_{mem}}{NUM_{mem}} = 37.$$

This number is used to adjust the weights at memory transition arcs so that the total warp "latency" was not less than CYC_{mem}. The new weights are (Eq. 4 in their paper):

$$Latency_{BW} = 4 + \max(\, 0,\, 37 - \alpha \cdot \max(\, 4,\, 20 / n \,) - 4 \,).$$

Finally, we consider the stalls on memory accesses. As we found in Chapter 6, memory latency on this GPU is 444 cycles. In this model, it is assumed that when one warp stalls on a memory access or on a barrier instruction, the processor switches to executing another warp, until that warp stalls too, and so on. Each warp is understood to stall after being executed for the following number of cycles (Eq. 5 in their paper):

$$NBC_{avg} = \frac{CYC_{compute}}{NUM_{mem} + NUM_{sync} + 1} \approx \frac{CYC_{compute}}{NUM_{mem}},$$

where NUM_{sync} is the number of barrier instructions and is zero in our case. Note that on a realistic GPU such context switches happen every issue cycle, not only at memory and barrier instructions [Lindholm et al. 2008].

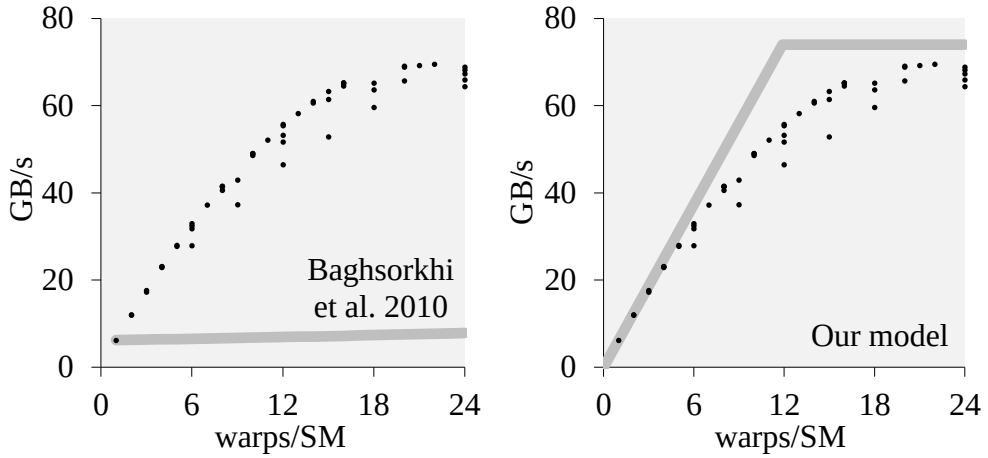
Since the number of warps available for switching is limited, the latency may be not fully hidden. The exposed part is found as (Eq. 6 in the paper):

$$Latency_{exposed} = 444 - (n - 1) \cdot NBC_{avg}.$$

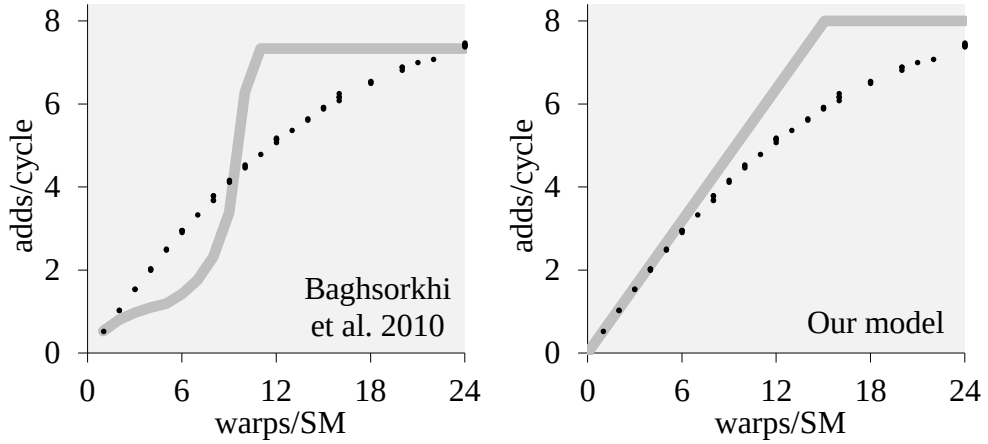
This number is assigned to each data dependence arc.

The graph and the assigned weights are then reduced to produce the total warp latency. In our case, this corresponds to finding a longest path in the graph. Warp latency, per the repeating group of $\alpha + 1$ instructions, thus, equals:

$$Warp\ latency = \alpha \cdot alu + \max(\, Latency_{BW},\, Latency_{exposed} \,),$$



(a) Memory-only case, $\alpha = 0$.



(b) Intermediate arithmetic intensity, $\alpha = 11$.

Figure 7.2: Baghsorkhi et al. model and our model compared with experiment on the G80 GPU.

which can be summarized as:

$$\textit{Warp latency} = \max(\, 37,\, x,\, 440 - (\, n - 2\,) \cdot x\,), \quad \text{where}$$
$$x = \alpha \cdot \max(\, 4,\, 20 / n\,) + 4.$$

The total execution time is found as the sum of all warp latencies. Arithmetic throughput, therefore, is

$$\textit{Throughput}\ (\text{adds/cycle}) = 32 \cdot \alpha \,/\, \textit{Warp latency}.$$

If all instructions are memory accesses ($\alpha = 0$), the overall instruction throughput is

$$\textit{Throughput}\ (\text{IPC/SM}) = 1 / \max(\, 448 - 4 \cdot n,\, 37\,).$$

This is a dramatic underestimate, as shown in Figure 7.2, a, left. The model is shown as a thick grey line, and experimental data is shown as dots. At 100% occupancy, the estimate is more than 8x smaller than what is observed in experiment. The model suggests that over 100 warps per SM

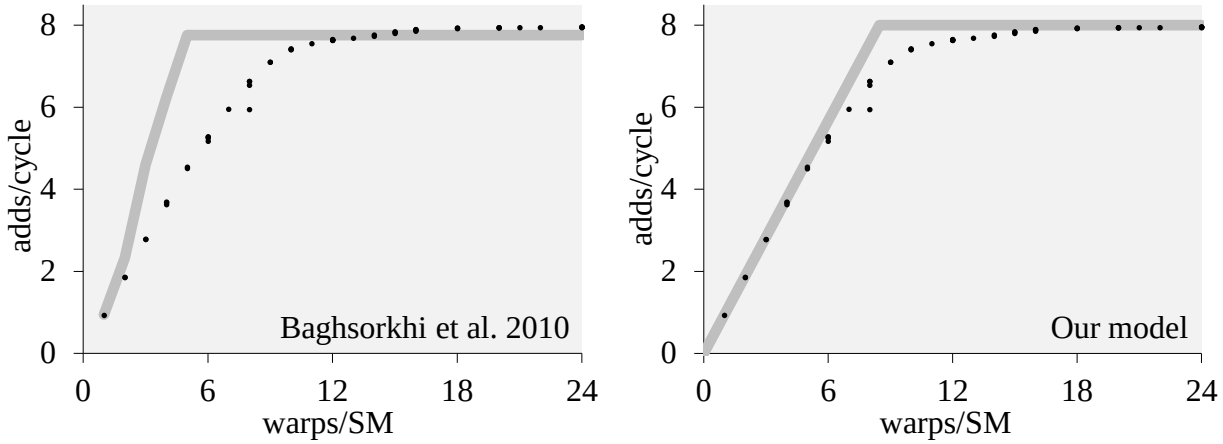


Figure 7.3: Baghsorkhi et al. model and our model compared with experiment on the G80 GPU with $\alpha = 32$.

are needed to attain peak memory throughput, whereas in practice 20 warps are sufficient to attain 94% of the peak. For comparison, the result according to our model is

$$\textit{Throughput}\ (\text{IPC/SM}) = 1 / \max(\ 444 / n,\ 37\ ),$$

which is much more accurate, as shown in the same figure, right.

The Baghsorkhi et al. model shows unusual behavior at intermediate α values, such as $\alpha = 11$ shown in Figure 7.2, b, left. Throughput at 100% occupancy is estimated accurately, but otherwise the behavior is neither what we expect, which is a linear or sublinear growth up to peak throughput, nor what is observed in experiment. Our model, shown in the same figure, right, does not have such an unusual feature.

This odd behavior is not as pronounced when arithmetic intensity is higher. An example with $\alpha = 32$ is shown in Figure 7.3, left. The model performs similarly to Hong and Kim model: it overestimates throughput in latency-bound case but is accurate otherwise. The worst disagreement with experiment is 1.7x. Our model is shown for comparison in the same figure, right. It is accurate in both latency-bound and throughput-bound cases but doesn't model the smooth transition between them.

If we use pin bandwidth (86.4 GB/s) and 24 cycle arithmetic latency, as suggested in the original paper, the results are not substantially different.

7.3 Zhang and Owens 2011 model

The GPU performance model by Zhang and Owens [2011] is the first to be accurate in both $\alpha = 0$ and $\alpha = \infty$ limits. Yet, it is less accurate at intermediate arithmetic intensities. We evaluate it on the GTX280 GPU, which is similar to the GTX285 GPU used in the original work; both are in the GT200 generation. In this evaluation, we make an approximation, as explained below.

The model employs a more detailed microbenchmarking than is used otherwise in GPU performance modeling. It is typically required to know instruction latencies, which are not disclosed by GPU vendors and are therefore found using reverse-engineering, as we did in Chapter 6. However, a similar technique may also be used to produce more detailed timing data, such as the throughputs attained in executing every specific instruction type at every specific occupancy.

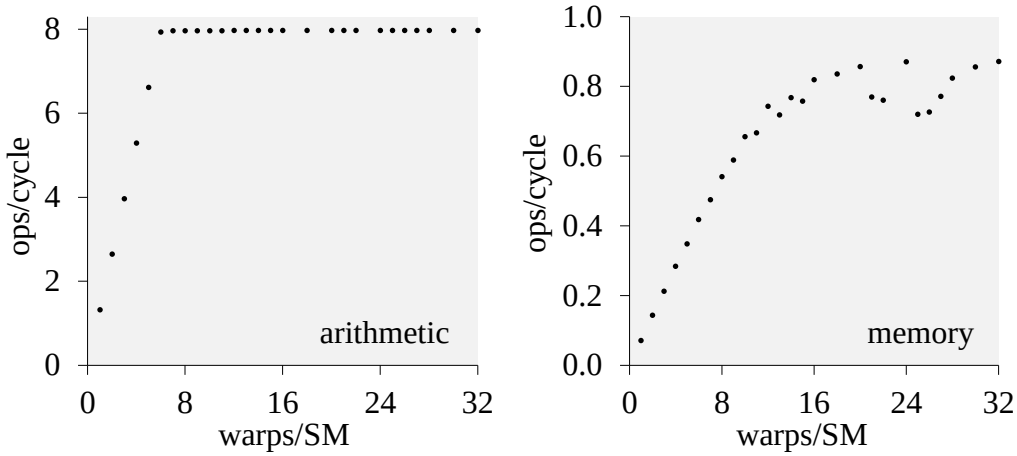


Figure 7.4: Input data for Zhang and Owens model on the GT200 GPU.

Such data for floating-point adds is shown in Figure 7.4, left. This is similar to the data used in Chapter 6 and shown in Figures 6.1 and 5.2, except limited to a single sample at each occupancy. These are the largest throughput samples, which is consistent with using only the largest throughput samples when comparing estimates with experiment. The throughputs are plotted in operations per cycle per SM. The plotted numbers are denoted below as *alu* (*n*).

Similar throughputs for memory instructions are shown in the same figure, right. They are denoted below as *mem* (*n*). In the case of memory accesses, Zhang and Owens suggest measuring throughputs not only at specific occupancies, but also at specific thread block sizes, specific thread block counts, and specific numbers of memory accesses per warp – these must be the same as used in the kernel, performance of which is being estimated. In this evaluation, we use only the same occupancies. The evaluation, thus, is only approximate. However, the omitted factors have little effect in our setup.

If throughput in ops per cycle is *alu* (*n*), then execution time in cycles per instruction is 32 / *alu* (*n*). By multiplying it by the number of such instructions executed in a given kernel, we find the total time it takes to execute all of these instructions. Similarly, we find the time it takes to execute all memory instructions. (The procedure is slightly more complicated if the kernel includes barrier instructions.) This can be written as:

$$\begin{aligned} alu\_time &= \#alu\_instructions \cdot 32 / alu\ (n), \\ mem\_time &= \#mem\_instructions \cdot 32 / mem\ (n). \end{aligned}$$

Since GPUs can execute arithmetic and memory instructions simultaneously, the total execution time is found as the larger of these times:

$$Time = \max(\ alu\_time, mem\_time\ ).$$

If *Time* is given per the repeating group of $\alpha + 1$ instructions, then arithmetic throughput in adds per cycle per SM is $32 \cdot \alpha$ / *Time*. Since this repeating group includes α arithmetic and 1 memory instruction, the solution for arithmetic throughput is:

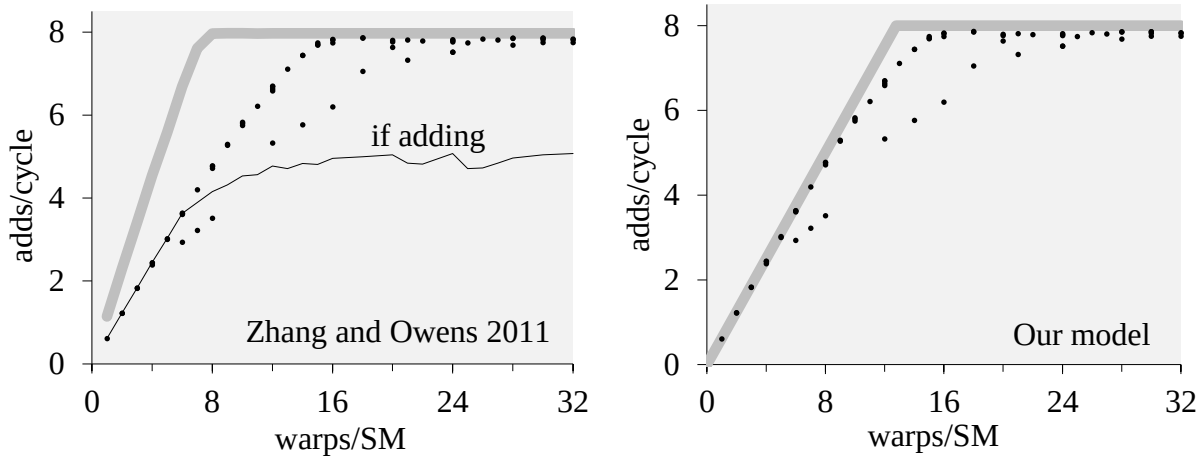


Figure 7.5: Zhang and Owens [2011] and our model compared with experiment on the GT200 GPU, $\alpha = 16$.

$$\textit{Throughput}\ (\text{adds/cycle}) = \min(\ \textit{alu}\ (n),\ \alpha \cdot \textit{mem}\ (n)\ ).$$

The result for $\alpha = 16$ is shown as a thick grey line in Figure 7.5, left. It is an overestimate in the latency-bound case: the difference is up to a factor of 1.9. The model is accurate at large occupancies, where performance is throughput-bound.

For comparison, in the same graph we show a similar estimate if assuming that arithmetic and memory times add, not overlap, i.e. if we take

$$\textit{Time} = \textit{alu\_time} + \textit{mem\_time}.$$

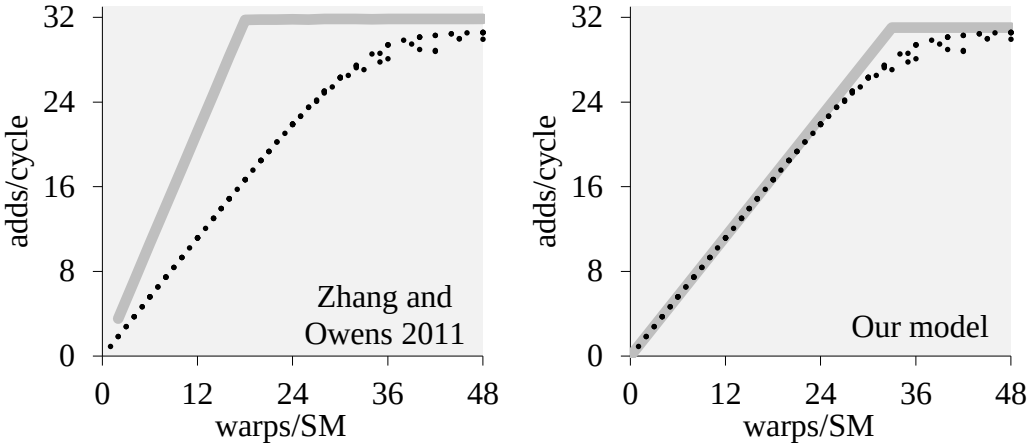
It is plotted in the same graph as a thin black line. The result has opposite character: it is accurate at small but not large occupancies. We discussed similar options in §4.12. Our model is shown in the same figure, right. It is accurate at both small and large occupancies.

We also evaluate Zhang and Owens model on several newer GPUs. The worst cases are summarized in Figure 7.6. In all of them, the model overestimates throughput in latency-bound case. The gap with experiment is up to 1.9x or 2.0x. On the Kepler GPU, case $\alpha = 32$, this model is substantially inaccurate at all occupancies (Figure 7.6, b). On the Maxwell GPU, case $\alpha = 64$, this model slightly overestimates throughput even in throughput-bound mode (Figure 7.6, c), which is the same as in our model, as was briefly noted in §4.12. Our model is shown in the same figure, right. It is substantially more accurate.

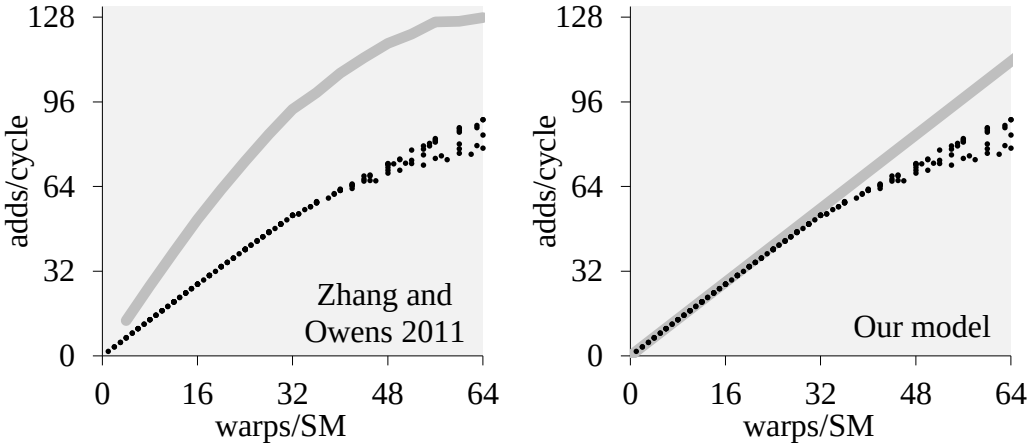
7.4 Sim et al. 2012 model

The Sim et al. [2012] model is a further development of the Hong and Kim model discussed before. An important improvement is inclusion of arithmetic latencies into consideration. The resulting model suffers from the same limitation as Zhang and Owens model: it is fairly accurate at limits $\alpha = 0$ and $\alpha = \infty$, but not at intermediate arithmetic intensities.

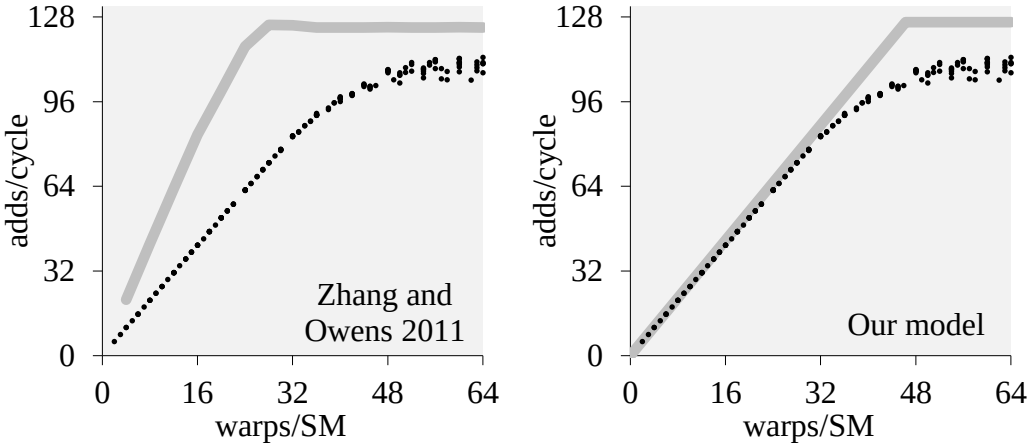
We evaluate this model on the GTX480 GPU, which is similar to the Tesla C2050 GPU used in the original work. Both of them have the Fermi architecture and similar multiprocessors of compute capability 2.0. The differences include: C2050 has one fewer multiprocessor, a lower processor clock rate, and a lower memory bandwidth.



(a) Fermi, $\alpha = 32$.



(b) Kepler, $\alpha = 32$.



(c) Maxwell, $\alpha = 64$.

Figure 7.6: Zhang and Owens [2011] model and our model on the newer GPUs.

Similarly to Zhang and Owens model, in this model we first estimate the time to process all arithmetic instructions and, separately, the time to process all memory instructions. The total time is then found as either the larger of these two times or using a somewhat more complicated formula.

Peak throughput of floating-point add instructions on the GTX480 GPU is 1 IPC per SM, and their latency is 18 cycles. This is the same as on the C2050 GPU. Given these metrics, arithmetic throughput at occupancy n is assumed to equal max(18/n,1) cycles per instruction. The total processing time in CUDA cores in our example, per the repeating group of α+1 instructions, is found as (Equations 2 to 6 in Sim et al. [2012]):

$$T_{comp} = (\alpha+1) \cdot n \cdot \max(18 / n, 1).$$

Processing time in the memory system is found in a similar manner. It equals the total number of executed memory instructions multiplied by their estimated reciprocal throughput (Equations 11 to 15 in the paper; also using that $MWP \le MWP_{peak\_bw}$):

$$T_{mem} = n \cdot 513 / \min(MWP, \max(1, CWP–1)).$$

Here, 513 cycles is memory latency on the GTX480 GPU (Table 6.2), and *MWP* and *CWP* are defined similarly as before – for example, both equal n when n is small – but with a few differences explained below.

When there is only one warp, each memory access takes 513 cycles. When there are n warps and n is small, one might expect an execution rate of 513 / n cycles per instruction – similarly to 18 / n cycles per arithmetic instruction above. However, the formula for T_{mem} suggests using instead a slower rate of 513 / (n – 1) cycles per instruction. The difference is substantial at smaller n such as 2 or 4, and at these occupancies we find that this model substantially underestimates throughput if arithmetic intensity is low, such as by a factor of 2 at n = 2. This explains a feature in Figure 7.11 that we discuss later in §7.9 but is not discernible in other graphs. If we replace *CWP*–1 with *CWP*, the feature is gone, but other limitations are not.

CWP is the same concept as in Hong and Kim [2009] but enhanced to include arithmetic latencies. Given that arithmetic instructions are assumed to be executed at the rate equal to max(18/n,1) cycles per instruction, we have (Equations 4, 5, and A.1 to A.4 in the original paper):

$$CWP = \min(n, 1 + 513 / ((\alpha+1) \cdot \max(18/n, 1))).$$

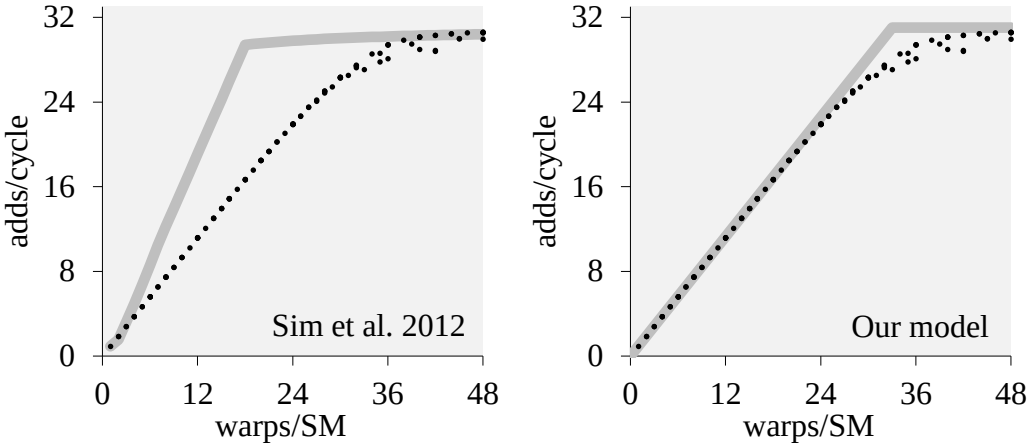
The definition of *MWP* is the same as in Hong and Kim model, except this time we have to discuss a term that was not previously important – the departure delay Δ (Equation A.5 in their paper):

$$MWP = \min(n, MWP_{peak\_bw}, avg\_DRAM\_lat / \Delta).$$

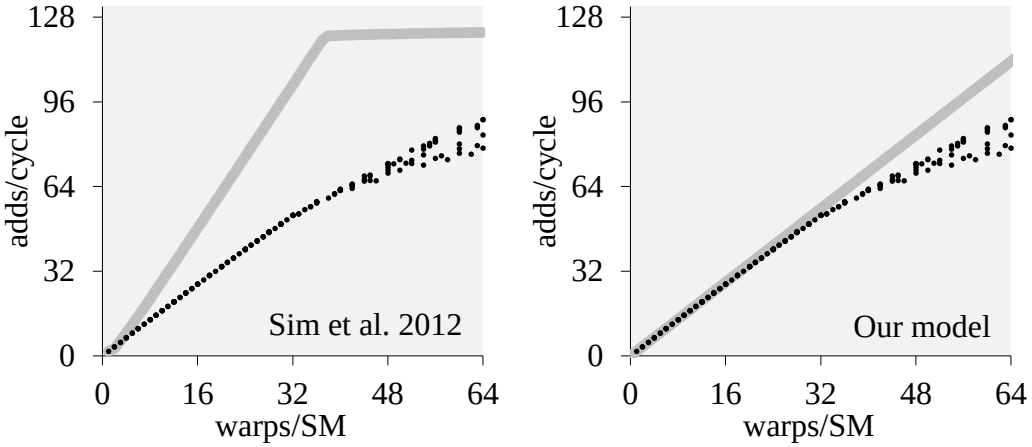
Here, *avg_DRAM_lat* is memory latency.

MWP is the largest number of warps that may access memory at the same time. *MWP* is n at best, but is further capped. The model suggests two such caps. One is the same as in §7.1: it is MWP_{peak_bw} to ensure that peak memory throughput is not exceeded. If we use the best sustainable throughput listed in Table 6.4 for the Fermi GPU as peak memory throughput, we get MWP_{peak_bw} = 30.7 warps/SM.

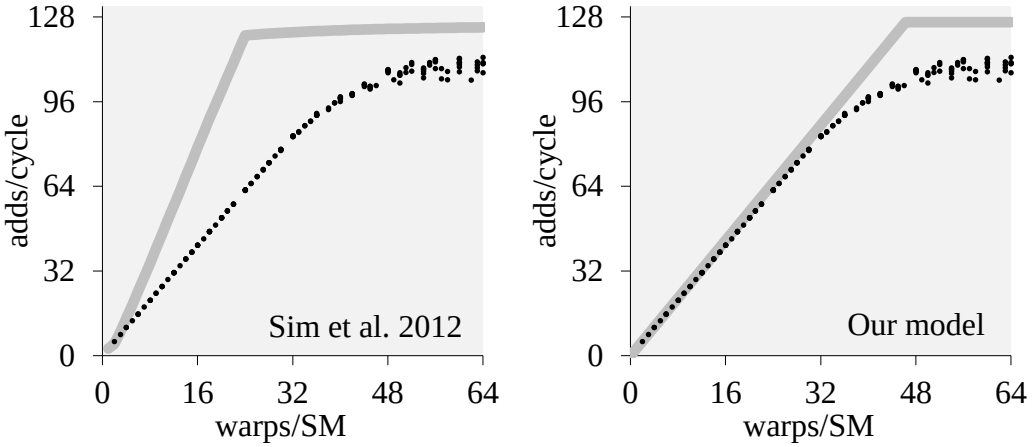
The second cap is due to departure delay. Departure delay Δ is the smallest time between issuing individual transactions on a multiprocessor. This concept is covered in more detail in the original Hong and Kim [2009] paper – see, for example, Figures 9 and 10 there. The departure delay on the GTX280 GPU, according to Hong and Kim, is small, only 4 cycles if accesses are coalesced,



(a) Fermi, $\alpha = 32$.



(b) Kepler, $\alpha = 32$.



(c) Maxwell, $\alpha = 64$.

Figure 7.7: Sim et al. [2012] model and our model compared to the experiment.

in which case the respective cap is loose and never effective; we quietly omitted it in §7.1. Departure delay on the GTX480 GPU is not generally known, but Sim et al. suggest it is not small on the C2050 GPU. It therefore requires a further consideration.

Indeed, if we use the data given in Sim et al. [2012] for the C2050 GPU, we find that MWP_{peak_bw} = 30.7, and *avg_DRAM_lat* / Δ = 22. This means that the cap due to departure delay is more pressing than the cap due to peak memory throughput, and the latter, therefore, is never effective. The effective peak throughput is then defined by the departure delay itself. This throughput is easy to find: on the C2050 GPU, we get 0.335 GB/s if executing one memory access instruction at a time – this is denoted *BW_per_warp* in the paper – so that with 22 warps, which is the cap, and 14 multiprocessors, we get 103 GB/s. This is similar to the peak memory throughput cited in other sources for this GPU. For example, Ruetsch and Fatica [2011, Table 2.1] quote 107 GB/s when ECC is on, which is the default setting (this is the case when memory error-correction functionality is enabled).

Using departure delay, thus, is used as a way to introduce a more realistic throughput bound into the model. Otherwise, bandwidth is limited only by 144 GB/s of pin bandwidth, which is noticeably higher than the best throughput attainable in practice. However, we have already addressed this concern by computing MWP_{peak_bw} for the GTX 480 GPU using the best sustainable throughput, unlike in the original paper. Therefore, the Δ term is redundant and may be omitted.

The model by Sim et al., however, is ambiguous in terms of the use of departure delay. Below we review some of the other options and explain why they are not relevant.

The original definition of Δ, as given in Hong and Kim [2009], is the smallest time between issuing individual transactions on the same multiprocessor. On one hand, if we know peak memory throughput we can convert it to the respective time between transactions. On the other hand, we may be able to attain a smaller time between transactions if using only one multiprocessor to access memory. To find if this is the case, we implement a microbenchmark where we run only one thread block on device. All warps in the thread block execute a pointer-chasing sequence with fully diverging accesses. We vary the number of warps and the number of threads enabled in each warp. The result on the GTX480 GPU is that throughput increases with the total number of active threads, but only up to 8.6 cycles per transaction. This number can be understood as departure delay. However, it suggests a looser cap on *MWP* than is due to MWP_{peak_bw} and therefore doesn't affect the result.

Another definition of departure delay is implied by Eq. 13 in Sim et al. [2012]: it is the increment in instruction latency due to issuing each additional memory transaction when accesses are diverging. If base memory latency is 513 cycles, and memory access instruction diverges into k transactions, then instruction latency is assumed to be $513 + (k - 1) \cdot \Delta$. Ideally, departure delay in this definition is the same as above, i.e. 8.6 cycles on average, but this is not necessarily the case. Using the CUDA profiler, we find that for every additionally issued memory transaction the diverging instruction is reissued. Reissue latency is another, independent parameter. To find it we implement another microbenchmark. Again, it is a pointer chasing with fully diverging accesses. But this time, we run only 1 warp per device with only k threads enabled, so that only k transactions are produced per instruction. By considering the resulting warp latencies, we find that Δ = 32 cycles. This is similar to the latency increment per transaction found in §6.7. If we use this number to define metric *MWP*, the throughput cap for the Fermi GPU drops down to 84 GB/s, which is not realistic: we observe substantially higher throughputs in practice.

To summarize, we use the best sustained memory throughput, not pin bandwidth, to find MWP_{peak_bw} and omit the term that depends on departure delay. The result is:

$$MWP = \min(n, 30.7).$$

	Fermi	Kepler	Maxwell
Arithmetic latency	18 cycles	9 cycles	6 cycles
Arithmetic throughput per SM	1 CPI	0.25 CPI	0.25 CPI
Memory latency	513 cycles	301 cycles	368 cycles
MWP_{peak_bw} (per SM)	30.7 warps	40.3 warps	29.9 warps

Table 7.1: The parameters used with Sim et al. model.

The total execution time is then found as (Eq. 1, 16 and 17 in the paper):

$$T_{exec} = \max(T_{comp}, T_{mem}) \quad \text{if } CWP > MWP,$$
$$T_{exec} = \max(T_{comp}, T_{mem} + T_{comp} / n) \quad \text{if } CWP \leq MWP.$$

Arithmetic throughput per SM equals:

$$\textit{Arithmetic throughput} (\textit{adds/cycle}) = 32 \cdot n \cdot \alpha / T_{exec}.$$

Comparison with experiment on the Fermi GPU is shown in Figure 7.7, a. As other models, Sim et al. model overestimates throughput in latency-bound case. The worst disagreement with experiment is 1.8x. The case in the figure is the worst observed; it corresponds to $\alpha = 32$. Our model, shown in the same figure, is substantially more accurate.

We also evaluated the model on the Kepler and Maxwell GPUs. The respective hardware parameters are listed in Table 7.1. The results are compared with experiment in Figure 7.7, b and c. Again, the model overestimates throughput in latency-bound case, this time by up to 2.0x on the Kepler and 1.9x on the Maxwell GPU. On the Kepler GPU the model is substantially inaccurate at all occupancies – similarly to Zhang and Owens model. Our model is more accurate.

7.5 Huang et al. 2014 model: round-robin policy

The performance model by Huang et al. [2014] is the most recent model that we evaluate in this text. If previous models lack accuracy in latency-bound case, this model lacks accuracy in throughput-bound case. The original paper includes several typos, such as in Eq. 7, 15, 16, 21, and 22. We confirmed them with the primary author and cite when using a respective equation.

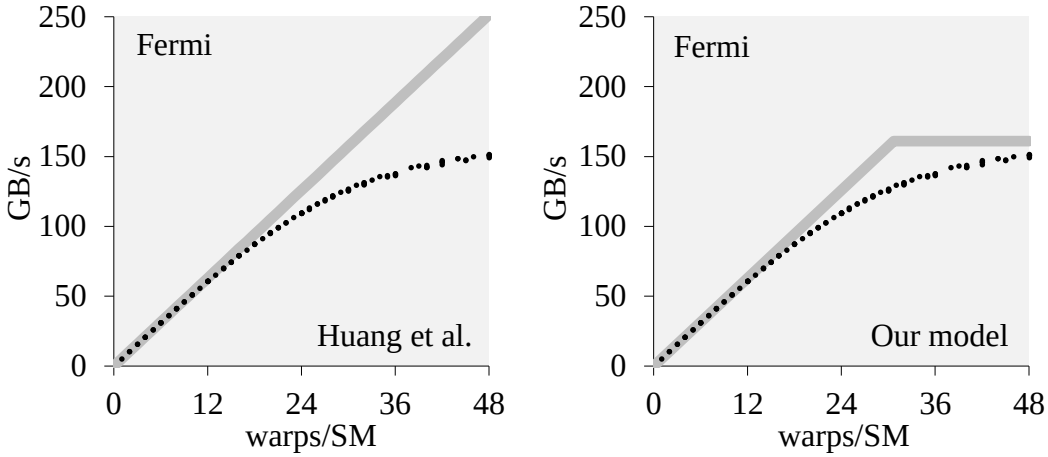
The model finds the reciprocal execution rate as a sum of two terms (Eq. 3 in their paper): the delays due to contention in the memory system and all other delays. If memory accesses are coalesced, as in our setup, the delays due to contention are assumed to be zero (their Section IV.B). In this case Eq. 3 becomes:

$$CPI_{final} = CPI_{multithreading}.$$

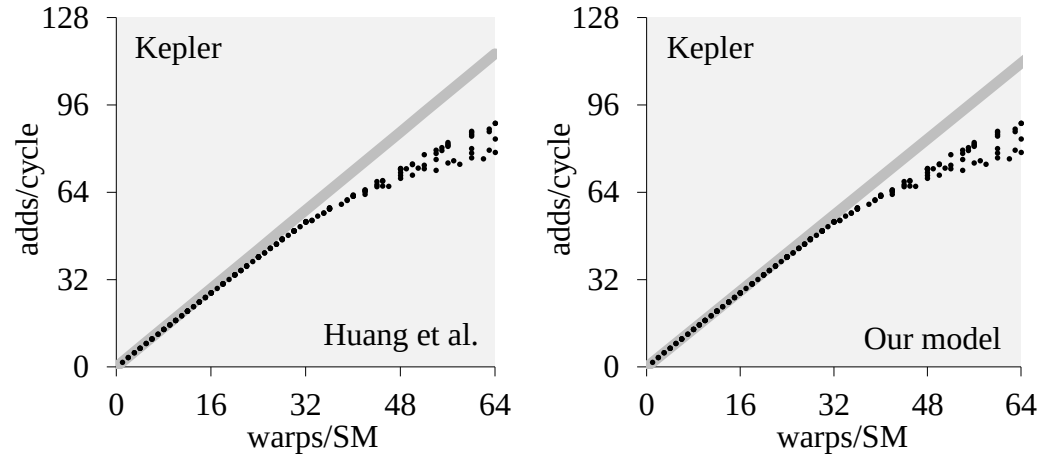
Below we use *IPC* as synonymous to 1 / *CPI*. For example, IPC_{final} is the same as 1 / CPI_{final}, and $CPI_{multithreading}$ is the same as 1 / $IPC_{multithreading}$.

It is assumed that peak instruction issue rate is 1 issue per cycle per SM, which matches the Fermi GPU. We also use this model on the Kepler and Maxwell GPUs, in which case n is interpreted as occupancy per scheduler, and the result, as throughput per scheduler.

First, we find a representative warp, which can be any in our case, and find its execution time as if it was executed alone with no throughput limits. This is similar to finding the latency bound in our model, as explained in §3.7. This execution time is denoted *total_cycles* and, in our case, equals, per the repeating group of α+1 instructions:



(a) Fermi, memory-intensive case ($\alpha = 0$).



(b) Kepler, intermediate arithmetic intensity ($\alpha = 32$).

Figure 7.8: Huang et al. [2014] model and our model, both scheduling policies.

$$total\_cycles = mem\_lat + \alpha \cdot alu\_lat.$$

The probability of having an instruction issued on any particular cycle is (Eq. 9 in their paper):

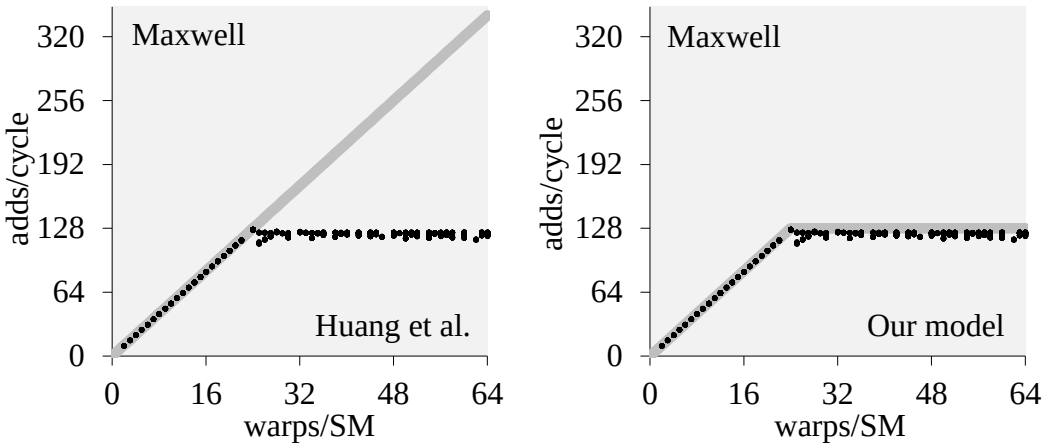
$$p = \frac{1 + \alpha}{mem\_lat + \alpha \cdot alu\_lat}.$$

Instruction throughput at occupancy n is then found as (Eq. 7 in their paper):

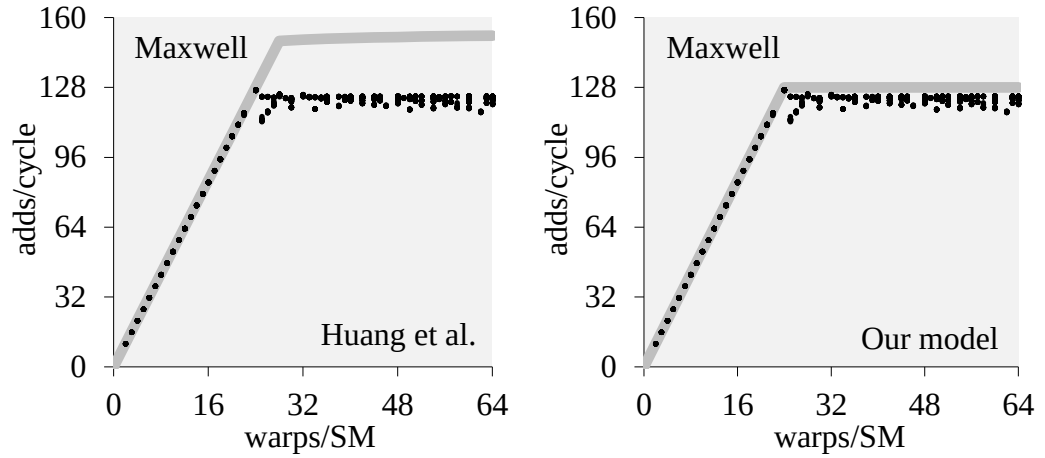
$$IPC_{multithreading} = \frac{n \cdot (1 + \alpha)}{total\_cycles + NO}.$$

Here, for short, we use *NO* in place of *#total_nonoverlapped_insts*. The original Eq. 7 has a typo: it has $CPI_{multithreading}$ on the left-hand side, which is the reciprocal of what is intended.

Term *NO* equals the sum of similar terms NO_i found for each interval *i*. Interval is a group of instructions that can be executed with no dependency stalls. In our case, all instructions are back-



(a) Round-robin scheduling policy.



(b) Greedy-then-oldest scheduling policy.

Figure 7.9: Huang et al. model, compute-intensive case ($\alpha = \infty$).

to-back dependent so that each instruction corresponds to a separate interval. The size of each interval is, therefore, one:

$$\#interval\_insts_i = 1.$$

Two approaches are suggested for finding terms NO_i. We consider the simpler solution first, which corresponds to the round-robin warp scheduling policy. In this case, we have (Eq. 10 and 11 in their paper):

$$NO_i = p \cdot (n - 1) \cdot (interval\_inst_i - 1),$$

which, for our kernel, is zero. Therefore, NO is also zero, and the throughput estimate is:

$$IPC_{final} = n \cdot (1 + \alpha) / (mem\_lat + \alpha \cdot alu\_lat).$$

By multiplying this by $32 \cdot \alpha / (\alpha+1)$ we get arithmetic throughput in adds/cycle per SM.

The result is similar to our model except includes no throughput limits. Therefore, it substantially overestimates throughput when occupancy is large. An example is shown in Figure 7.8, a, left. This is memory-intensive case, $\alpha = 0$, on the Fermi GPU. At 100% occupancy the estimate is 48/513 IPC per SM, or 252 GB/s. This substantially exceeds both pin bandwidth, which equals 177 GB/s, and the largest throughput attained in experiment, which equals 151 GB/s.

Another example is shown in Figure 7.9, a, left. This is compute-intensive case, $\alpha = \infty$, on the Maxwell GPU. At 100% occupancy the estimate is 341 adds/cycle per SM, whereas throughput cannot exceed 128 adds/cycle because there are only 128 CUDA cores per SM.

In both cases, our model is substantially more accurate and does respect hardware throughput limits, as shown in the same figures, right.

In some of the other cases, there is no difference between Huang et al. and our model. These are the cases where hardware throughput bounds are not attained at even 100% occupancy. An example is shown in Figure 7.8, b. It corresponds to intermediate arithmetic intensity, $\alpha = 32$, and the Kepler GPU. According to both our model and Huang et al. model, throughput at 100% occupancy is 115 adds/cycle per SM. In practice, throughput is only 89 adds/cycle per SM due to the gradual saturation effect.

7.6 Huang et al. model: greedy-then-oldest policy

The second solution suggested by Huang et al. [2014] corresponds to the greedy-then-oldest warp scheduling policy. Given that all intervals in our case have size one, we have (Eq. 13 in the paper):

$$avg\_interval\_insts = 1,$$

in which case Eq. 12, 14 and 15 can be simplified down to:

$$\#issue\_insts\_in\_stall_i = \min(p \cdot stall\_cycles_i, 1) \cdot (n - 1).$$

Here, $stall\_cycles_i$ equals latency of the respective instruction minus one. This is the latency of arithmetic instruction if interval i is comprised of an arithmetic instruction, and the latency of memory instruction otherwise. The original Eq. 15 and Eq. 16 have typos: max is used in place of min and vice versa. By using Eq. 16 we get:

$$NO_i = \max (0, \min (p \cdot stall\_cycles_i, 1) \cdot (n - 1) - stall\_cycles_i).$$

Thus, NO equals, per the repeating group of α+1 instructions:

$$NO = NO_{mem} + \alpha \cdot NO_{alu}.$$

where:

$$NO_{mem} = \max(0, \min(p \cdot (mem\_lat - 1), 1) \cdot (n-1) - mem\_lat + 1),$$
$$NO_{alu} = \max(0, \min(p \cdot (alu\_lat - 1), 1) \cdot (n-1) - alu\_lat + 1).$$

The new throughput estimate can then be summarized as:

$$IPC_{final} = n \cdot (1 + \alpha) / (mem\_lat + \alpha \cdot alu\_lat + NO_{mem} + \alpha \cdot NO_{alu}).$$

In cases $\alpha = 0$ and $\alpha = \infty$, this can be further simplified to:

$$IPC_{\alpha=0} = n \,/\, \max(\, mem\_lat,\, (n-1) \cdot (1 - 1 \,/\, mem\_lat) + 1\,).$$
$$IPC_{\alpha=\infty} = n \,/\, \max(\, alu\_lat,\ \ (n-1) \cdot (1 - 1 \,/\, alu\_lat)\ \ + 1\,).$$

Since *mem_lat* is in hundreds, and *n* is 48 at most, we have:

$$IPC_{\alpha=0}\ (\text{reasonable } n) = n \,/\, mem\_lat,$$

which is the same as with round-robin scheduling policy and is similarly inaccurate, see Figure 7.8, a. There is also no difference between the two policies if arithmetic intensity is not zero but small, such as when $\alpha = 32$ and GPU is the Kepler GPU. This is the same as the case shown in Figure 7.8, b.

If $\alpha = \infty$, then at sufficiently large *n* we have:

$$CPI_{\alpha=\infty}\ (\text{large } n) = 1 - 1/alu\_lat,$$

which exceeds the peak issue throughput of 1 IPC. If arithmetic latency is 25 cycles, as on the synthetic GPU considered in the original work, peak throughput is exceeded by only a little – 4%. But if arithmetic latency is 6 cycles, as on the Maxwell GPU, peak throughput is exceeded substantially – by 20%. This case is shown in Figure 7.9, b.

7.7 Huang et al. model with bandwidth term

The major limitation of Huang et al. [2014] model is that it ignores limits on memory throughput when memory accesses are coalesced. However, it does consider these limits when accesses are diverging. The solution proposed for diverging accesses can also be used with coalesced accesses, but it doesn't rectify the problem, as we show below.

In this case $CPI_{rc\_contention}$ is not zero but equals (Eq. 17 in the paper):

$$CPI_{rc\_contention} = \frac{MSHR\_delay + Bandwidth\_delay}{1 + \alpha}.$$

Here and below, we omit subscripts *i* present in the original text because all intervals comprised of memory instruction are similar, and all intervals comprised of arithmetic instruction have zero *MSHR_delay* and *Bandwidth_delay*.

Instruction throughput is then found as (Eq. 3):

$$CPI_{final} = CPI_{multithreading} + CPI_{rc\_contention},$$

where $CPI_{multithreading}$ is found using one of the approaches above.

We assume that all transactions have size 128 bytes. This is the case on the Fermi GPU and the synthetic GPU considered in the paper. Each coalesced access instruction in this case is mapped to a single memory transaction, so that Eq. 18 in the paper reads:

$$\#core\_reqs = n.$$

We also assume that the number of MSHR registers, which are used to keep track of the issued memory transactions, is sufficiently large to allow at least one transaction per warp. This is the

case on the synthetic GPU, which has 32 MSHR registers and allows executing up to 32 warps per SM at the same time (Table I in the paper), and with the Fermi GPU, which allows executing up to 48 warps per SM at the same time and, according to the study by Nugteren et al. [2014], has 64 MSHR registers per SM. Therefore, according to Eq. 20 in Huang et al., *MSHR_delay* is zero.

The model for *Bandwidth_delay* roots in queuing theory. Arrival rate λ for memory requests is found as (Eq. 23 in the paper):

$$\lambda = \frac{n \cdot \#cores}{mem\_lat}.$$

This quantity is defined for each interval but is nonzero only for memory intervals. Therefore, again, we omit index *i*.

The time to serve each request is found as:

$$s = clock\ rate \cdot 128\ \text{B} / memory\ pin\ bandwidth,$$

where 128 bytes is the transaction size.

Utilization factor ρ is the attained fraction of peak throughput. It is found as (Eq. 22):

$$\rho = \lambda s.$$

Finally, *Bandwidth_delay* is (Eq. 21 in the paper):

$$Bandwidth\_delay = \min\left(\frac{\lambda s^2}{2 \cdot (1-\rho)}, \frac{s \cdot n \cdot \#\text{cores}}{2}\right).$$

These two equations (Eq. 21 and 22) have typos in the original text: some instances of ρ and λ miss subscript *i*. We omit the subscripts on purpose for the same reason as above.

A difficulty with this model is that it allows utilization factor ρ to exceed 1, as if peak memory throughput was exceeded. For example, on the synthetic processor considered in the paper, we have: $s = 0.667$ cycles, *#cores* = 16, maximum occupancy is $n = 32$, and memory latency is 300 cycles. This produces $\rho \approx 1.14$.

Since ρ can exceed 1, *Bandwidth_delay* can be negative. Moreover, it can attain arbitrarily large negative values as the formula is singular at $\rho = 1$. For example, if $n = 29$ and the GPU is the synthetic GPU considered in the paper, we have $\rho \approx 1.03$ and *Bandwidth_delay* ≈ -11.

Negative *Bandwidth_delay*, in turn, may result in negative throughput and, therefore, in negative execution times. For example, if in addition to the above we assume $\alpha = 0$, then we get:

$$CPI_{final} = mem\_lat / n + Bandwidth\_delay \approx -0.7.$$

Figure 7.10, left, shows the case of $\alpha = 16$ on the same synthetic GPU. The solution is the same for both scheduling policies. It is singular at $n \approx 28.1$ warps per SM.

The same figure, right, shows a similar singularity in the case $\alpha = 32$ on the Fermi GPU. Experimental data is shown as dots, the solution for round-robin policy is shown as thick grey line, and the solution for greedy-then-oldest policy is shown as thin black line. The estimated throughput at $n = 34$ is negative and equals approximately –12 adds/cycle per SM in both cases.

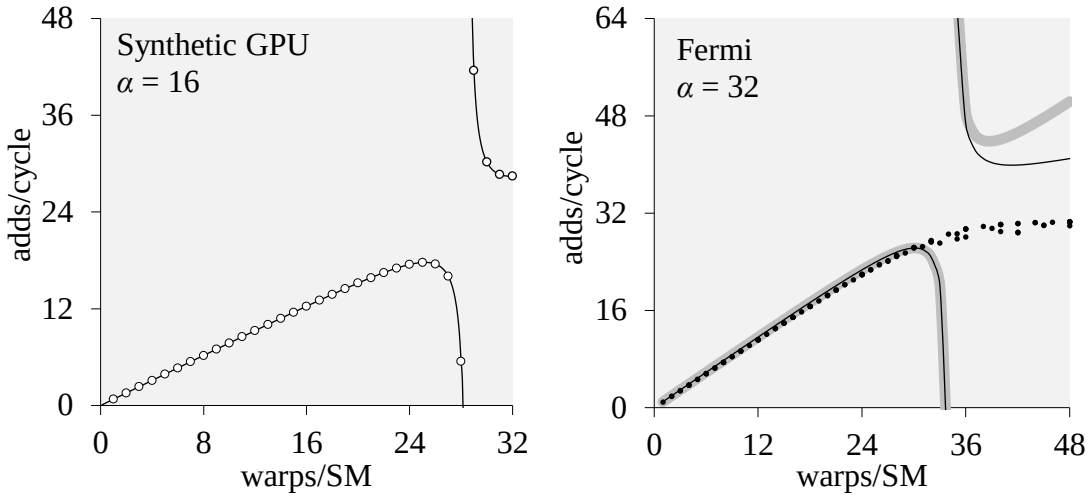


Figure 7.10: Huang et al. model with bandwidth terms.

This, however, is not the intended use of this model: terms *Bandwidth_delay* and $CPI_{rc\_contention}$ are supposed to be used only with diverging accesses.

7.8 Huang et al. model: non-coalesced accesses

If we use the resource contention model described above with diverging accesses, which is the intended use, we still may get negative execution times.

Consider a slightly modified workload, where all memory accesses are stride-2 instead of stride-1, but which otherwise is the same. Then, each memory access instruction maps into two 128-byte transactions, so that

$$\#core\_reqs = 2 \cdot n, \quad \text{and} \quad \lambda = 2 \cdot n \cdot \#cores \,/\, mem\_lat.$$

Let $n = 17$ and the GPU be the Fermi GPU. Then $\#core\_reqs = 34$, which is less than the limit of MSHR registers (64). Therefore *MSHR_delay* is zero according to Eq. 20 in the original paper. If we also assume that $\alpha = 0$, then, for both scheduling policies, we get:

$$CPI_{final} = mem\_lat \,/\, n + \min\left(\frac{\lambda s^2}{2 \cdot (1 - \lambda s)}, s \cdot n \cdot \#\text{cores}\right).$$

Given 15 cores, 513 cycle memory latency, 1.4 GHz clock rate, and 177.4 GB/s memory bandwidth, we get *Bandwidth_delay* ≈ -120 and $CPI_{final} \approx -89$ cycles per instruction, which is a negative throughput.

7.9 Summary

So far we compared models with experiment only at specific arithmetic intensities, such as those where the models are the least accurate. Below and in Figures 7.11, 7.12 and 7.13, we summarize how the same models perform across a range of arithmetic intensities. We don't include the Hong and Kim model in this summary as it ignores arithmetic latency and therefore is highly inaccurate when arithmetic intensity is large and occupancy is small.

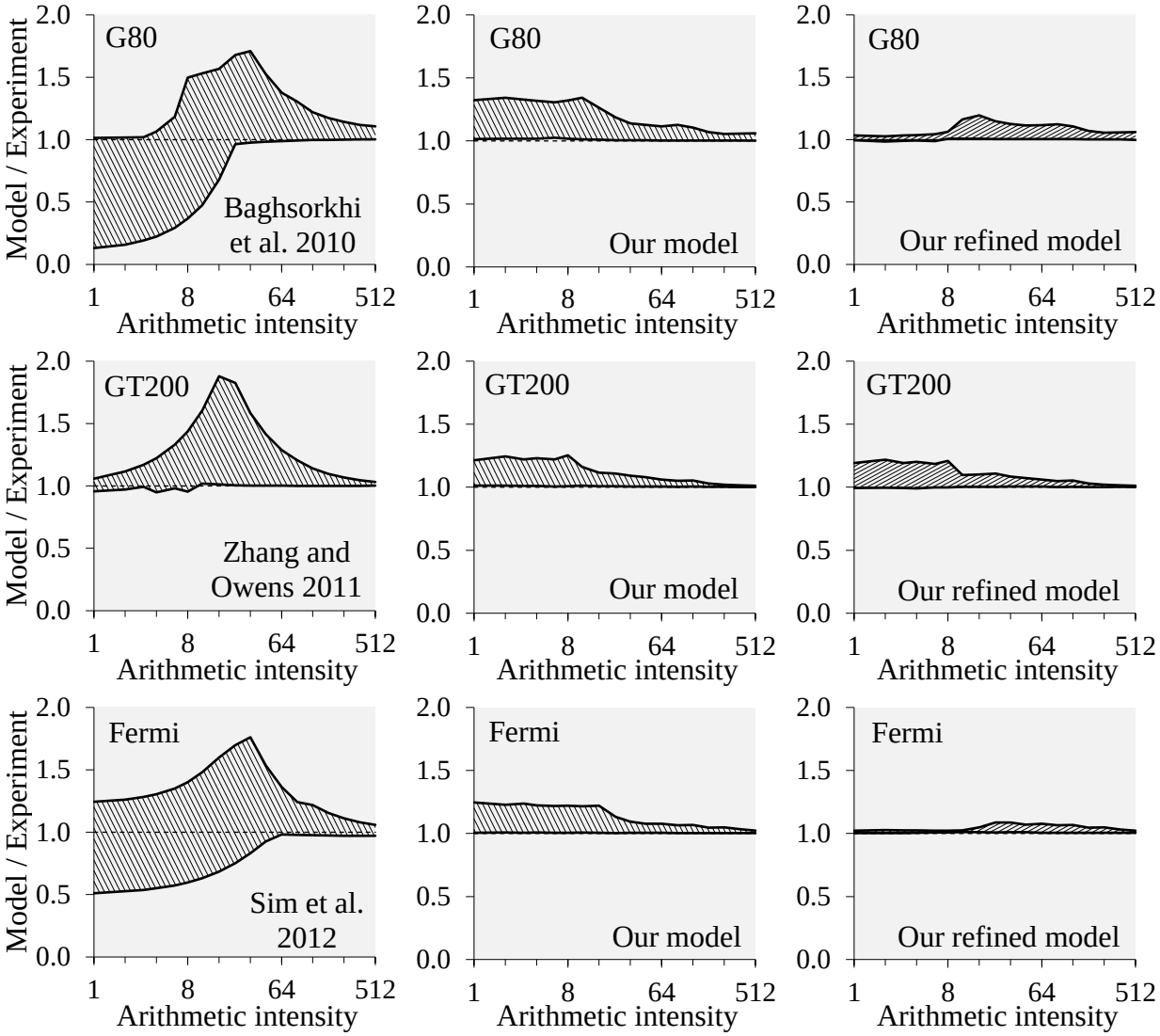


Figure 7.11: Estimation error vs arithmetic intensity in different performance models.

We consider α in range between 1 and 512, sampled at approximately every integer power of $\sqrt{2}$. For each such α, for each particular performance model, and separately for individual GPUs, we collect data similar to what was shown in numerous graphs earlier in this chapter: the estimated and the observed throughputs at each occupancy. As usual, we consider only the occupancies that correspond to integer numbers of warps per scheduler and only the largest experimentally observed throughput at each such occupancy. To characterize the accuracy, each estimate is divided by the respective observed value. If the resulting quotient is greater than 1, the respective estimate is too large; if the quotient is less than 1, the estimate is too small. The smallest and the largest of such quotients are plotted versus α in Figures 7.11, 7.12 and 7.13 as boundaries of the shaded regions; the extent of these regions, thus, describes the range of the error. Different graphs correspond to different GPUs and different performance models.

Consider, for example, the data presented earlier in Figure 7.3, left. It includes estimates due to the Baghsorkhi et al. model for the case $\alpha = 32$ and the respective experimental data. Here,

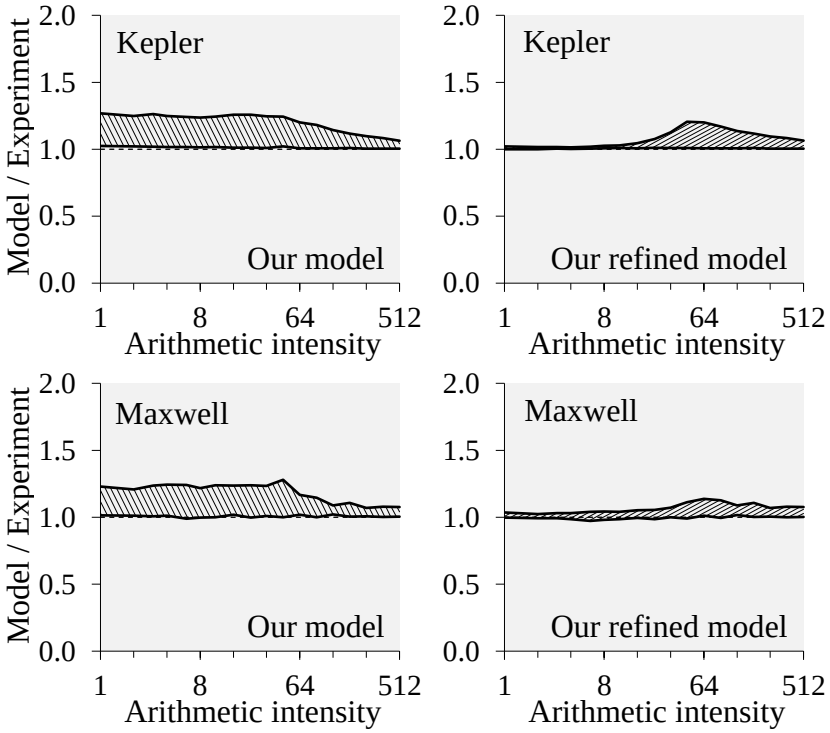


Figure 7.12: Estimation error in our performance models on newer GPUs.

throughput is overestimated by up to a factor of 1.710 and underestimated by up to a factor of 1/0.975. The respective quotients, 0.975 and 1.710, are plotted in Figure 7.11, top left, at $\alpha = 32$ as boundaries of the shaded region.

Figure 7.11 shows such data for the models that were found to be substantially inaccurate in latency-bound case. These are the models by Baghsorkhi et al. [2010], Zhang and Owens [2011], and Sim et al. [2012]. With each of these models we use a GPU similar to what was used in the original publication. The figure also includes similar plots for both our models: the basic model presented in §4.1 and the refined model presented in §6.5. Figure 7.12 shows similar data for our models and the two newer GPUs.

As the figures show, all models are increasingly accurate when fraction of arithmetic instructions in the mix approaches 100%. Indeed, the case when all instructions are CUDA core instructions and are back-to-back dependent is the simplest and the best understood.

Memory-intensive case is a more difficult challenge. The model by Baghsorkhi et al. is the least accurate in this respect: it dramatically underestimates throughput at small arithmetic intensities, such as by a factor of 7.6 at $\alpha = 1$ and a factor of 2.7 at $\alpha = 8$. The model by Sim et al. underestimates throughput by up to a factor of 2. This is the effect of term $CWP–1$ discussed in §7.4. Such a substantial underestimate occurs only at small occupancies such as 2 warps per SM. At occupancy 4 warps per SM the difference is already less: a factor of only 1.3. The model by Zhang and Owens is the most accurate: at $\alpha = 1$ its estimates are within 6% of the experimental data. This is the result of extensively using experimental data as input.

Our basic model doesn't capture the gradual saturation effect and therefore its accuracy is also limited in memory-intensive case. The errors are up to a factor of between 1.25 and 1.28 on all but the G80 GPU, where the error is up to a factor of 1.34. A similar overestimate by a factor of 1.25

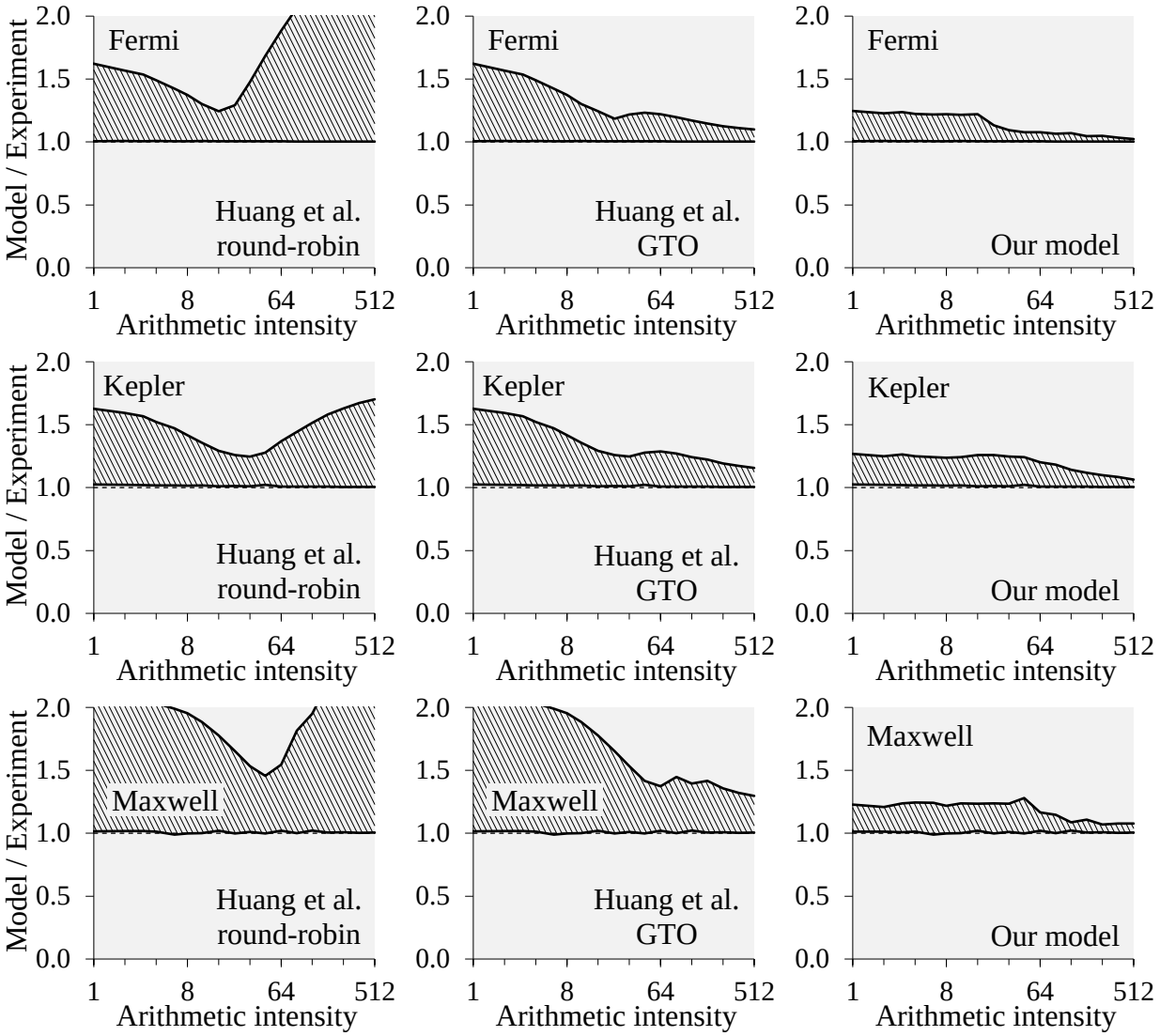


Figure 7.13: Errors vs arithmetic intensity in Huang et al. 2014 model.

at $\alpha = 1$ is also observed with Sim et al. model, which also doesn't capture the gradual saturation effect.

The case that proves to be the most challenging is the case of intermediate arithmetic intensities. This case is not accurately resolved in any of these three prior models. In each of them, at some α, throughput is overestimated by a factor of over 1.7. Our model doesn't have this feature. At intermediate arithmetic intensities it is about as accurate as in the memory-intensive limit.

The accuracy of the refined model is shown in the same graph. It is substantially more accurate in memory-intensive cases on all but the GT200 GPU; on the GT200 GPU experimental data includes a substantial scatter, due to TLB, which is not modeled. On the other GPUs, this refined model still does not fully solve the problem of gradual saturation. On the Kepler GPU, for example, the refined model overestimates throughput by a factor of about 1.2 at some α. The best overall improvement is found on the Fermi GPU, where the worst overestimate is by a factor of only 1.09. These worst performing cases also correspond to intermediate arithmetic intensities.

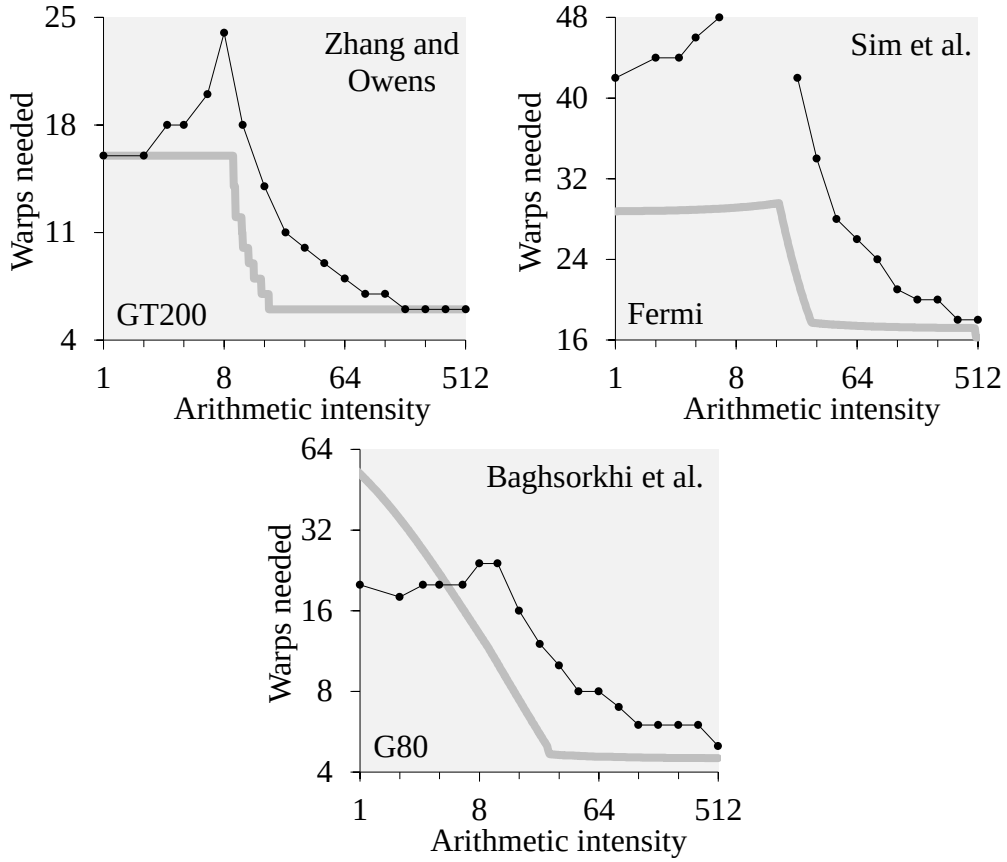


Figure 7.14: Prior models don't have cusp behavior.

Figure 7.13 summarizes the errors found in Huang et al. model. Here we consider the original version of this model, which does not include bandwidth terms discussed in §7.7. The model is accurate in latency-bound case but is substantially inaccurate in throughput-bound case. In result, the overall accuracy is better at intermediate arithmetic intensities, where peak throughput is attained at larger occupancies, and worse at very small and very large arithmetic intensities. In the case of the round-robin scheduling policy, the model is substantially inaccurate in both compute-intensive and memory-intensive cases. In the case of the greedy-then-oldest policy (GTO), it is substantially inaccurate in memory-intensive cases only. Our basic model has a better overall accuracy in all cases, as shown in the same figure.

The prior models are inaccurate in either latency-bound or throughput-bound case and, as a result, are substantially inaccurate at either intermediate or all but intermediate arithmetic intensities. Our model, in contrast, is reasonably accurate over the entire range of arithmetic intensities. Its accuracy is limited only by the gradual saturation effect, finer performance features of the Maxwell GPU, and the TLB-induced scatter in the case of the GT200 GPU. The accuracy of our model is further improved if using the memory contention model introduced in §6.4.

These findings correspond to the synthetic instruction mixes introduced in Chapter 4. Accuracy on realistic kernels may be different.

7.10 Prior models show no cusp behavior

Cusp behavior is a basic property found both in our model and experiment, as shown in §4.5. It means that attaining a peak throughput at intermediate arithmetic intensities requires more warps than attaining a peak throughput at very small or very large arithmetic intensities. This is due to the difference between warp concurrency and instruction concurrency discussed in §3.3.

Prior models don't expose cusp behavior, as shown in Figure 7.14. It includes similar graphs to those considered in §4.5. Plotted as dots are the smallest occupancies where 90% of a peak throughput is attained in experiment. Plotted as thick grey line are the smallest occupancies where the same throughput is attained in the respective performance model. As usual, we consider using the same or similar GPUs to those used in the respective original publications.

The figure shows the result for three models: Baghsorkhi et al. [2010], Zhang and Owens [2011], and Sim et al. [2012]. We don't consider Hong and Kim model, because it ignores arithmetic latency and therefore cannot have a noticeable cusp effect. We also don't consider Huang et al. model, because it does not predict that throughput is saturated, except at instruction issue limit with the greedy-then-oldest policy.

According to the figure, none of the three shown models have cusp behavior – except a subtle, insubstantial effect in the Sim et al. model. Also, the Baghsorkhi et al. model is shown to dramatically overestimate the needed occupancy in memory-intensive cases, which was noted earlier.

Conclusion

Concurrency is becoming even more ubiquitous, but is still not widely understood. To understand performance of new concurrent devices such as GPUs, we may consider them in a more general context and adopt the same basic concepts as developed in previous studies of concurrent systems. This requires understanding the subtleties of both the classical concurrency concepts and the GPU microarchitecture. This is the approach we tried to follow in this work.

Central in our discussions was Little's law. It is simple and yet requires care in use. For example, we found it important to distinguish different types of concurrency, such as warp concurrency, instruction concurrency, arithmetic instruction concurrency, and memory instruction concurrency. Also, we found that some implications of Little's law are counter-intuitive. For example, short arithmetic latency was found to have a similar impact on performance as the notably longer memory latency.

The key feature of fine-grained multithreading, such as used on GPUs, is the ability to transparently hide different latencies, such as arithmetic latency, memory latency, warp termination latency, and many others. This is in contrast with some of the previously studied techniques, such as coarse-grained multithreading and prefetching, which address hiding memory latency only. This narrow focus on memory latency is unwarranted on modern GPUs. Also a common pitfall is disregarding memory bandwidth. Although memory bandwidth is widely recognized as a key factor in processor performance, it is not always given the appropriate attention in performance modeling.

This work is in also line with prior work on understanding GPU performance by using microbenchmarking, such as Buck et al. [2004], Volkov and Demmel [2008], and Wong et al. [2010]. The simplistic workload we suggested may, too, be classified as a microbenchmark. It is only incrementally more complex than similar microbenchmarks used before and exposes new aspects of hardware operation.

We have shown that performance estimation on GPUs can be reduced to a relatively unsophisticated cycle counting. The key idea was to separately count cycles in latency-bound and throughput-bound cases. An immediate future work is to populate this framework with additional microarchitectural details, such as additional latencies and throughput limits, and evaluate it on a larger set of realistic kernels and more recent GPUs.

Another possible bound on throughput, not considered in this work, is due to hardware limits on concurrency. An example of such a limit is the maximum number of memory transactions that can be processed at the same time. Similar bounds were previously discussed by Shebanow as "space limiters" [Shebanow 2008; Shebanow 2010]. Taking them into account may improve performance modeling accuracy in some of the less common cases.

References

Agarwal. A. 1989. Performance tradeoffs in multithreaded processors. MIT VLSI Memo No. 89-566.

Aila, T., and Laine, S. 2009. Understanding the efficiency of ray traversal on GPUs. *Conference on High Performance Graphics 2009* (HPG '09), 145–149.

Alberto, C. M. M., and Hiroyuki, S. 2015. Linear performance-breakdown model: A framework for GPU kernel programs performance analysis. *International Journal of Networking and Computing 5*, 1, 86–104.

AMD. 2015. AMD OpenCL optimization guide (rev 1.0), August 2015.

Baghsorkhi, S. S., Delahaye, M., Patel, S. J., Gropp, W. D., and Hwu, W. W. 2010. An adaptive performance modeling tool for GPU architectures. *Symposium on Principles and Practice of Parallel Programming* (PPoPP '10), 105–114.

Bailey, D. H. 1997. *Little's Law and High Performance Computing*. Manuscript. http://www.davidhbailey.com/dhbpapers/little.pdf

Buck, I., Fatahalian, K., and Hanrahan, P. 2004. GPUBench: Evaluating GPU performance for numerical and scientific applications, *GP^2 Workshop on General Purpose Computing on Graphics Processors*, C-20.

Burger, D., Goodman, J. R., and Kägi, A. 1996. Memory bandwidth limitations of future microprocessors. *International Symposium on Computer Architecture* (ISCA '96), 78–89.

Chen, X. E., and Aamodt, T. M. 2009. A first-order fine-grained multithreaded throughput model. *International Symposium on High Performance Computer Architecture* (HPCA '09). 329–340.

Coon, B. W., and Lindholm, J. E. 2008. System and method for managing divergent threads in a SIMD architecture. US Patent 7,353,369.

Coon, B. W., and Lindholm, J. E. 2009. System and method for managing divergent threads using synchronization tokens and program instructions that include set-synchronization bits. US Patent 7,543,136.

Coon, B. W., Lindholm, J. E., Mills, P. C., and Nickolls, J. R. 2010. Processing an indirect branch instruction in a SIMD architecture. US Patent 7,761,697.

Coon, B. W., Nickolls, J. R., Lindholm, J. E., and Tzvetkov, S. D. 2011. Structured programming control flow in a SIMD architecture. US Patent 7,877,585.

Dao, T. T., Kim, J., Seo, S., Egger, B., and Lee, J. 2015. A performance model for GPUs with caches. *IEEE Transactions on Parallel and Distributed Systems 26*, 7, 1800–1813.

Dally, W., and Towles, B. 2003. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann.

Gebhart, M., Johnson, D. R., Tarjan, D., Keckler, S. W., Dally, W. J., Lindholm, E., and Skadron, K. 2011. Energy-efficient mechanisms for managing thread context in throughput processors. *International Symposium on Computer Architecture* (ISCA '11), 235–246.

Gray, S. 2014. A full walk through of the SGEMM implementation, https://github.com/NervanaSystems/maxas/wiki/SGEMM

Gross, D., Shortle, J. F., Thompson, J. M., and Harris, C. M. 2008. *Fundamentals of Queuing Theory* (4th ed.). Wiley.

Guz, Z., Itzhak, O., Keidar, I., Kolodny, A., Mendelson, A., and Weiser, U. C. 2010. Threads vs. caches: Modeling the behavior of parallel workloads. *International Conference on Computer Design* (ICCD), 274–281.

Harris, M. 2005. Mapping computational concepts to GPUs. *ACM SIGGRAPH Courses*, Chapter 31 (Los Angeles, July 31-Aug. 4).

Hennessy, J. L., and Patterson, D. A. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach* (5th ed.). Morgan-Kaufmann.

Hong, S., and Kim, H. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *International Symposium on Computer Architecture* (ISCA '09), 152–163.

Howes, L. 2015. The OpenCL Specification Version: 2.1 Document Revision: 23. Khronos OpenCL Working Group.

Huang, J.-C., Lee, J. H., Kim, H., and Lee, H.-H. S. 2014. GPUMech: GPU performance modeling technique based on interval analysis. *International Symposium on Microarchitecture* (MICRO-47), 268–279.

Jain, R. 1991. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience.

Kothapalli, K., Mukherjee, R., Rehman, M. S., Patidar, S., Narayanan, P. J., and Srinathan, K. 2009. A performance prediction model for the CUDA GPGPU platform. *International Conference on High Performance Computing* (HiPC '09), 463–472.

Klaiber, A. C., and Levy, H. M. 1991. An architecture for software-controlled data prefetching. *International Symposium on Computer Architecture* (ISCA '91), 43–53.

Lai, J., and Seznec, A. 2012. Break down GPU execution time with an analytical method. *Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools* (RAPIDO '12), 33–39.

Lai, J., and Seznec, A. 2013. Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs. *International Symposium on Code Generation and Optimization* (CGO '13), 1–10.

Lazowska, E. D., Zahorjan, J. G., Graham, G. S., and Sevcik, K. C. 1984. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall.

Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro 28*, 2, 39–55.

Little, J. D. C. 1961. A proof for the queuing formula: $L = \lambda W$. *Operations Research 9*, 3, 383–387.

Little, J. D. C., and Graves, S. C. 2008. Little's Law. In *Building Intuition: Insights From Basic Operations Management Models and Principles (Eds.: Chhajed, D., and Lowe, T. J.)*. 81–100, Springer.

Little, J. D. C. 2011. Little's Law as viewed on its 50th anniversary, *Operations Research 59*, 3, 536–549.

Ma, L., Agrawal, K., and Chamberlain, R. D. 2014. A memory access model for highly-threaded many-core architectures, *Future Generation Computer Systems 30*, 202–215.

McKee, S. A. 2004. Reflections on the memory wall. *Computing Frontiers* (CF '04), 162–167.

Mei, X., and Chu, X. 2016. Dissecting GPU memory hierarchy through microbenchmarking. *ArXiv:1509.02308*, http://arxiv.org/abs/1509.02308.

Merrill, D., and Grimshaw, A. 2010. Revisiting sorting for GPGPU stream architectures. Technical Report CS2010-03, Department of Computer Science, University of Virginia.

Mowry, T. C., Lam, M. S., and Gupta. A. 1992. Design and evaluation of a compiler algorithm for prefetching. *International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS V), 62–73.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. 2008. Scalable parallel programming with CUDA. *ACM Queue 6*, 2, 40–53.

Nickolls, J., and Kirk, D. 2009. Graphics and computing GPUs. In *Computer Organization and Design: The Hardware/Software Interface, 4th ed. (Eds.: Patterson, D. A., and Hennessy, J. L.).* C1–C83, Elsevier.

Nickolls, J., and Dally, W. J. 2010. The GPU computing era. *IEEE Micro 30*, 2, 56–69.

NVIDIA. 2006. *NVIDIA GeForce 8800 GPU Architecture Overview*. Technical Brief. November 2006.

NVIDIA. 2008. *NVIDIA GeForce GTX 280 GPU Architectural Overview*. Technical Brief. May 2008.

NVIDIA. 2009. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. Whitepaper. V1.1.

NVIDIA. 2010a. *NVIDIA CUDA Programming Guide v3.0*. February 2010.

NVIDIA. 2010b. *NVIDIA GF100*. Whitepaper. V1.5.

NVIDIA. 2012a. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. Whitepaper. V1.0.

NVIDIA. 2012b. *NVIDIA GeForce GTX 680*. Whitepaper. V1.0.

NVIDIA. 2012c. *Dynamic Parallelism in CUDA*. Technical Brief. NVIDIA.

NVIDIA. 2013. *CUDA C Programming Guide v5.5*. NVIDIA. July 2013.

NVIDIA. 2014a. *NVIDIA GeForce GTX 980*. Whitepaper. V1.0.

NVIDIA. 2014b. *NVIDIA GeForce GTX 750 Ti*. Whitepaper. V1.1.

NVIDIA. 2014c. *NVAPI Reference Documentation*. Release 337: May 2, 2014. NVIDIA.

NVIDIA. 2015. *CUDA C Programming Guide v7.5*. NVIDIA. May 2015.

NVIDIA. 2016. *NVIDIA Tesla P100*. NVIDIA. April 2016.

Nugteren, C., van den Braak, G.-J., and Corporaal, H. 2014. Detailed GPU cache model based on reuse distance theory. *International Symposium on High Performance Computer Architecture* (HPCA '14), 37–48.

Podlozhnyuk, V. 2007. Black-Scholes option pricing. NVIDIA SDK.

Resios, A. 2011. GPU performance prediction using parametrized models. Master's thesis, Utrecht University.

Ruetsch, G., and Fatica, M. 2011. *CUDA Fortran for Scientists and Engineers: Best Practices for Efficient CUDA Fortran Programming* (1st ed.). Morgan Kaufmann.

Saavedra-Barrera, R., Culler, D., and von Eicken, T. 1990. Analysis of multithreaded architectures for parallel computing. *Symposium on Parallel Algorithms and Architectures* (SPAA '90), 169–178.

Sim, J., Dasgupta, A., Kim, H., and Vuduc, R. 2012. A performance analysis framework for identifying potential benefits in GPGPU applications. *Symposium on Principles and Practice of Parallel Programming* (PPoPP '12), 11–22.

Shebanow, M. 2008. GPU Computing: Pervasive massively multithreaded processors. *Workshop on Architectures and Languages for Throughput Applications (ALTA 2008)*.

Shebanow, M. 2010. Personal communication.

Stratton, J. A., Rodrigues, C., Sung, I-J., Obeid, N., Chang, L.-W., Anssari, N., Liu, G. D, and Hwu, W-m. W. 2012. Parboil: A revised benchmark suite for scientific and commercial

throughput computing. IMPACT Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Center for Reliable and High-Performance Computing.

Song, S., Su, C., Rountree, B., and Cameron, K. W. 2013. A simplified and accurate model of power-performance efficiency on emergent GPU architectures. *International Symposium on Parallel & Distributed Processing* (IPDPS), 673–686.

Volkov, V. 2010. Better performance at lower occupancy. *GPU Technology Conference* (GTC '10).

Volkov, V., and Demmel, J. W. 2008. Benchmarking GPUs to tune dense linear algebra. *ACM/IEEE Conference on Supercomputing* (SC '08). Article 31, 11 pages.

Volkov, V., and Kazian, B. 2008. Fitting FFT onto the G80 architecture. CS 258 final project report, University of California, Berkeley.

Williams, S., Waterman, A., and Patterson, D. 2009. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM 52*, 4, 65–76.

Wittenbrink, C. M., Kilgariff, E., and Prabhu, A. 2011. Fermi GF100 GPU architecture. *IEEE Micro 31*, 2, 50–59.

Wong, H., Papadopoulou, M. M., Sadooghi-Alvandi, M., and Moshovos, A. 2010. Demystifying GPU microarchitecture through microbenchmarking. *International Symposium on Performance Analysis of Systems and Software* (ISPASS-2010), 235–246.

Zhang, Y., and Owens, J. D. 2011. A quantitative performance analysis model for GPU architectures. *International Symposium on High Performance Computer Architecture* (HPCA '11). 382–393.