Improved Simulation of the Nvidia Kepler Memory Hierarchy through Microbenchmarking

A Thesis Presented
by

Brad J Courville

to

The Department of Electrical and Computer Engineering
in partial fulfillment of the requirements
for the degree of

Master of Science

in

Electrical and Computer Engineering

Northeastern University
Boston, Massachusetts

April 2017
Contents

List of Figures iv
List of Tables v
List of Acronyms vi
Acknowledgments vii
Abstract of the Thesis viii

1 Introduction 1
  1.1 Motivating GPU Acceleration .................................. 2
  1.2 Challenges of Simulating GPU hardware .......................... 3
    1.2.1 Instruction Set Architecture .................................. 4
    1.2.2 Compute Pipeline ........................................ 4
    1.2.3 Device Architecture and Memory Hierarchy ..................... 4
    1.2.4 Contributions ........................................ 4
  1.3 Thesis Organization .................................. 5

2 Background 6
  2.1 Cache Memory .................................... 6
    2.1.1 Multi-level Caching ...................................... 7
    2.1.2 Set Associativity ...................................... 7
    2.1.3 Cache Memory Addressing .................................. 9
    2.1.4 Cache Replacement Policies .................................. 9
  2.2 Microarchitectural Simulation .................................. 10
  2.3 Nvidia Kepler Microarchitecture ............................... 10
    2.3.1 Streaming Multiprocessor - SMX .............................. 11
    2.3.2 Memory Hierarchy .................................... 12

3 Related Work 14
  3.1 Multi2Sim Heterogeneous System Simulator .................... 14
    3.1.1 Kepler Instruction Set Architecture ......................... 14
    3.1.2 Kepler Compute Pipeline .................................. 15
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Cache Scaling vs. Energy (Source [1, 2])</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Cache Scaling vs. Access Time (Source [1, 2])</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Image of Kepler GK110 (Source [3, 4])</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Kepler SMX (Source [3])</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Kepler Memory Hierarchy (Source [3])</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>K40c Read-only / Texture Cache</td>
<td>21</td>
</tr>
<tr>
<td>4.2</td>
<td>K40c L1</td>
<td>22</td>
</tr>
<tr>
<td>4.3</td>
<td>K40c L2</td>
<td>23</td>
</tr>
<tr>
<td>4.4</td>
<td>Estimating Cache Block Size By Varying Stride</td>
<td>24</td>
</tr>
<tr>
<td>5.1</td>
<td>Kepler Execution Model</td>
<td>26</td>
</tr>
<tr>
<td>6.1</td>
<td>Load/Store Instructions</td>
<td>30</td>
</tr>
<tr>
<td>6.2</td>
<td>Number of Instructions</td>
<td>31</td>
</tr>
<tr>
<td>6.3</td>
<td>SDK Instruction Breakdown</td>
<td>32</td>
</tr>
<tr>
<td>6.4</td>
<td>Scaled Timing Errors</td>
<td>33</td>
</tr>
<tr>
<td>6.5</td>
<td>CUDA 6.5 Benchmark Results</td>
<td>34</td>
</tr>
<tr>
<td>6.6</td>
<td>CUDA 6.5 Benchmark Error</td>
<td>35</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Set Associative Cache Indexing ........................................... 9
4.1 Set-associative Cache Parameters (Source [5]) ....................... 20
6.1 Measured Access Latencies (clock cycles) ............................. 28
6.2 Nvidia Kepler Cache Parameters ........................................ 28
6.3 Nvidia CUDA 6.5 Benchmarks ........................................... 29
List of Acronyms

CPU  Central Processing Unit
DRAM  Dynamic Random Access Memory
FIFO  First-In First-Out
GPGPU  General Purpose Graphics Processing Unit
GPU  Graphics Processing Unit
HPC  High Performance Computing
IR  Intermediate Representation
ISA  Instruction Set Architecture
LRU  Least Recently Used
LSU  Load/Store Unit
PTX  Parallel Thread Execution
SASS  Streaming Assembly
SIMT  Single-Instruction Multiple-Thread
SMX  Next Generation Streaming Multiprocessor
TLB  Translation Lookaside Buffer
Acknowledgments

I would like to thank my group office at MIT Lincoln Laboratory for being so supportive and accommodating, the NUCAR Lab members who assisted me including Fanny Nina-Paravecino, Xun Gong, and Xiang Gong, as well as Prof. Rafael Ubal Tena and the other developers of Multi2Sim whose work I relied on heavily to complete this effort. I would also like to extend a special thank you to my advisor Prof. David Kaeli without whose guidance this work would not have been possible.
Abstract of the Thesis

Improved Simulation of the Nvidia Kepler Memory Hierarchy through Microbenchmarking

by

Brad J Courville

Master of Science in Electrical and Computer Engineering
Northeastern University, April 2017
Prof. David Kaeli, Advisor

General Purpose Graphics Processing Unit (GPGPU)s are frequently used to accelerate the performance of many types of parallel scientific and engineering workloads. The advent of GPU programming frameworks, such as Nvidia’s CUDA and Khronos’s OpenCL, has made it far easier to program these devices by providing a familiar C-syntax programming environment. Along with the increased popularity of these accelerators, comes an increased demand for simulation software that is capable of emulating their performance. Such computer architecture simulation software can be immensely useful in the performance evaluation of new hardware and architectural design decisions, and can be used to influence future microarchitectures. One such heterogeneous simulator capable of emulating the Nvidia Kepler device is Multi2Sim. The utility of any simulation infrastructure is dependent upon how well it reflects the execution on actual hardware.

In this thesis we utilize microbenchmarks to highlight a number of microarchitectural properties of an NVIDIA Kepler GPU. The goal is to develop an understanding of the memory access timings and cache parameters present in the Kepler memory hierarchy. This knowledge is then integrated into the Multi2sim Kepler model to improve simulation accuracy. These enhancements improve the timing accuracy of the MultiSim Kepler model by an average of 10.8% and a maximum of 26.8% with for memory intensive benchmarks when compared to physical hardware.
Chapter 1

Introduction

Although the name remains unchanged, the utility of Graphics Processing Unit (GPU)s has grown far beyond their humble computer graphics specific origins. The term GPGPUs is often used to describe these devices when they are used more broadly for various scientific, medical, and engineering computing contexts. The range of applications running on these devices continues to grow, enabling us to detect skin cancer as well as a trained dermatologist, delivering the Artificial Intelligence processing power required for cars to drive themselves [6, 7, 8]. The advent of GPU programming frameworks such as NVIDIA CUDA and OpenCL has made programming these devices simpler and allows software developers to quickly and fully leverage the large amount of parallelism available in a modern GPU. There are also many CUDA extension libraries such as cuBLAS (linear algebra) and cuFFT (signal processing) that allow developers to accelerate programs that use common scientific and engineering functions [9, 10].

There are several key differences between a modern GPU and a more traditional multi-core Central Processing Unit (CPU). GPUs are considered to be throughput optimized devices while CPUs are latency optimized, and as such, have different strengths. The throughput focused GPUs are capable of a high degree of parallelism, and as such are able to execute many similar commands concurrently. While the clock frequencies in a modern GPU are generally a bit slower than in a CPU, the Single Instruction Multiple Thread (SIMT) design of the GPU allows it to perform a large amount of computation.

Computer architecture researchers often rely on computer simulations to evaluate and compare various hardware and software features. The evaluation of tradeoffs can require a significant amount of time and investment. It is generally difficult to alter a hardware design once it has been built. This burden can be significantly reduced by developing a simulation of the hardware elements
CHAPTER 1. INTRODUCTION

within a software framework. However, it can be challenging to build such a software framework that accurately reproduces hardware behavior. Many software simulators have been developed to evaluate various ISAs, cache memories, and even entire systems [11, 12]. These software abstractions are generally forced to make trade-offs between simulation detail and simulation runtime performance, and may provide different versions the simulate at the ISA, functional, and detailed timing levels.

The Nvidia Tesla class of accelerators are powerful GPGPU devices that are widely used across academia and industry. The Nvidia Tesla series are designed for High Performance Computing (HPC) systems and include the K20, K40, and K80 (Kepler microarchitecture). While the Kepler microarchitecture was first released in 2012, Nvidia continues to innovate their GPGPU microarchitecture, including the recently released P100 (Pascal microarchitecture) [3, 13]. Due to the widespread adoption of the Kepler class accelerators, the accurate simulation of such devices is of particular interest within the computer architecture field. Some aspects of the NVIDIA Kepler architecture are provided in the NVIDIA documentation, but many of the important microarchitecture details are withheld [3].

The aim of this work is to uncover some of these hidden implementation details about the memory hierarchy used in the NVIDIA Kepler GPU architecture, and then use those details to improve the accuracy of the Multi2Sim Kepler GPU Simulator model. Having an accurate memory model is incredibly important for a system simulator to achieve representative performance.

1.1 Motivating GPU Acceleration

\[ y = ax + y \]  

(1.1)

Listing 1.1: C SAXPY

```c

void saxpy(int N, float a, float *x, float *y) {
    for (int i = 0; i < N; i++) {
        y[i] = a*x[i] + y[i];
    }
}

```

A simple example that illustrates the benefits of GPU acceleration is the generalized vector addition subroutine in 1.1, commonly referred to as AXPY. The term SAXPY refers to a single-
CHAPTER 1. INTRODUCTION

precision floating-point implementation of AXPY. This and other AXPY subroutines are commonly used in linear algebra applications and workloads [14, 15, 9]. These types of subroutines are used in a distributed version of the LINPACK benchmark to evaluate the performance of the world’s top supercomputers [16, 17].

Listing 1.2: CUDA SAXPY (Source [18])

```c
__global__
void saxpy(int N, float a, float* restrict x, float* restrict y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) y[i] = a * x[i] + y[i];
}
...
int N = 1 << 20;
cudamemcpy(d_x, x, N, cudamemcpyHostToDevice);
cudamemcpy(d_y, y, N, cudamemcpyHostToDevice);
// Perform SAXPY on 1M elements
saxpy<<<4096, 256>>>(N, 2.0, x, y);
cudamemcpy(y, d_y, N, cudamemcpyDeviceToHost);
```

The C implementation of SAXPY shown in 1.1 is straightforward and serves as a reference implementation. This implementation performs the calculations serially in a for-loop and utilizes only a single CPU thread. The Nvidia CUDA implementation shown in 1.1 performs the same computation in parallel on the GPU and is able to utilize hundreds or thousands of GPU threads if they are available. When the vector size (N) is large, the GPU accelerated parallel code can achieve significant speed-ups.

1.2 Challenges of Simulating GPU hardware

It is a difficult task to accurately simulate a hardware system when many aspects of the underlying system are unknown. GPU and CUDA simulation are especially difficult because many of the details of the Instruction Set Architecture (ISA) are not available to the public. This work targets the Multi2Sim Kepler timing model for simulation with the understanding that it is not currently cycle-accurate, but with the intention of improving the accuracy of the model in terms of performance.
CHAPTER 1. INTRODUCTION

This effort focuses on the Kepler memory hierarchy implementation and the Load/Store Unit (LSU) for servicing memory access requests.

1.2.1 Instruction Set Architecture

An accurate functional simulation of the Nvidia Kepler microarchitecture requires a disassembler that can parse the natively-compiled CUDA assembly code back into instructions, as well as the logic to execute those instructions correctly. Timing simulation additionally requires detailed knowledge and modeling of the GPU pipeline dynamics, and requires knowledge about individual instruction execution and interaction between multiple active instructions.

1.2.2 Compute Pipeline

Effective timing simulation requires a detailed understanding of the microarchitectural pipeline and the functional units involved. These details can include specialized processing hardware such as single-precision floating-point units, double-precision floating-point units, load/store units, and special function units. The Nvidia Kepler microarchitecture in particular includes 192 single-precision cores, 64 double-precision cores, 32 load/store units, and 32 special function units per SMX [3]. The specific timings for these operations may not be known precisely and must be modeled approximately during simulation.

1.2.3 Device Architecture and Memory Hierarchy

Not all of the microarchitectural details about a GPU device and associated memory hierarchy will be known, and so some assumptions will need to be made for simulation. For the case of the Nvidia Kepler, some parameters, such as the number of Next Generation Streaming Multiprocessor (SMX)s and size of each cache, are published [3]. Other parameters must be inferred or estimated through experimentation. While the total size of the shared L2 data cache is known to be 1,536KB for the Nvidia Kepler GK110 implementation, some detailed cache organization, such as the set-associativity and replacement policy, require further investigation [3].

1.2.4 Contributions

This thesis focuses on improving the accuracy of a state-of-the-art simulator, the Multi2Sim simulator [19]. The outcome of this work will be to benefit the broader computer architecture research community. The specific contributions of this thesis include:
CHAPTER 1. INTRODUCTION

• measurement and approximation of different access latencies across multiple levels of the Nvidia Kepler cache hierarchy,

• the design/implementation of experiments and discovery of a variety of Kepler cache parameters, and

• implementation/evaluation of incorporating this new knowledge in the Kepler memory hierarchy model in the Multi2sim framework.

1.3 Thesis Organization

This thesis is organized as follows: Chapter 2 provides background necessary to understand GPU memory microarchitecture, and associated technologies discussed in this thesis, Chapter 3 describes related work around microarchitecture exploration performed on other platforms, Chapter 4 describes the experimental methodology used to obtain the organization parameters of the Kepler cache hierarchy, Chapter 5 discusses the integration of our findings in the Multi2sim Kepler simulator, Chapter 6 describes the results and findings of this work, and Chapter 7 provides a summary of this thesis, and directions for future work.
Chapter 2

Background

GPGPUs are now widely used to tackle challenging computational problems within many scientific and engineering disciplines. When computational workloads can be divided into sub-tasks and processed concurrently, they are often able to achieve a large performance improvement through GPGPU acceleration. Modern GPU devices offer some limited support for synchronization and communication between different threads, so even though tasks are not completely independent, they are able to achieve a speedup through parallel computation.

Nvidia CUDA and OpenCL are two leading software frameworks for compiling and running programs on these highly-parallel GPU devices[20, 21]. CUDA and OpenCL provide a familiar programming language with C-style syntax, allowing a large number of developers to quickly learn and write efficient programs for these devices.

2.1 Cache Memory

Cache memories are present on most CPU and GPU systems. The goal of the cache is to improve memory performance. Caches exploit the temporal and spatial locality present in memory address streams. Caches are typically developed in static RAM (SRAM) technology, though can also be implemented in dynamic RAM or Flash technologies. When data is stored in main memory of the system, if we *cache* a copy of the data closer to the processing pipeline, we can significantly reduce memory access latencies and improve overall performance [1].

Trade-offs involving memory storage capacity, access latency, chip area, complexity, power usage, cost and other metrics must be considered when designing the memory hierarchy for a computer system. Knowledge about the characteristics of representative workloads and their memory
access patterns should be leveraged during the design phase. The computer architecture needs to understand how best to leverage the spatial and temporal locality present in the workload. When a memory hierarchy is well matched to a workload, the working set of application should fit within the capacity of the cache structure.

2.1.1 Multi-level Caching

When a program requests a block of data from memory using a load instruction, the amount of time it takes to actually retrieve the data is highly dependent on which memory level contains a valid copy of the data. When using multiple cache levels in the memory hierarchy, each level is checked subsequently from the innermost storage level (generally smallest and fastest) to the outermost storage level (generally largest and slowest). Effective multi-level cache designs leverage the spatial and temporal locality inherent in common memory access patterns. However, memory access patterns are can vary significantly based on application workload and processing pipeline must be generalized to work well under a variety of conditions.

In contrast to the requirements for a CPU’s cache structure, which must work reasonably well for a broad range of workloads, a GPU is optimized for memory patterns that can be expected in most data parallel applications. These designs may even incorporate aggressive prefetching of data and coalescing related memory operations to further leverage the available memory-level parallelism, inherent in GPGPU applications [1].

When it comes to the performance of retrieving data on a memory load instruction, the latency increases at each level of the memory hierarchy. Some typical access times are given in [1]- registers: 0.15-0.30ns, cache: 0.5-15ns, main memory: 30-200ns, and disk storage: 5,000,000ns. Using a shared last-level cache can be critically important for performance as there is significantly latency penalty when a memory access is required to go off chip to either main memory or disk [22]. Due to the large performance penalty associated with off-chip accesses, the last-level cache capacity and replacement policy should be considered carefully to achieve optimal system performance.

2.1.2 Set Associativity

When designing a cache structure, computer architects make design decisions about how the underlying memory addresses are mapped to cache block addresses. The simplest method is to use a direct-mapped cache, whereby each memory address maps to exactly one location in the cache. Since the underlying memory space does not fit entirely in the cache, many memory addresses will
CHAPTER 2. BACKGROUND

Figure 2.1: Cache Scaling vs. Energy (Source [1, 2])

index into the same location in the cache, creating potential conflicts. These conflicts can cause poor cache performance by prematurely evicting useful cache blocks, even though there may be other cache block entries that are older or unused.

One way to address this issue is to allow each new cache entry to be stored in any of the possible cache block locations, resulting in a fully associative cache. While this approach will produce a much better cache hit rate, the added complexity in terms of finding an address can be prohibitive in terms of the access latency and hardware logic requirements. The cache block replacement policy for a fully associative cache needs to consider all entries in the cache to determine which block to evict, and the cache also needs to be able to search through every single cache entry to determine if the requested block is present or not. The decision logic required for a fully-associative cache does not always scale effectively.

Most modern cache designs opt for a trade-off between these two extreme cases, known as a set associative cache. Set associative caches allow each address in memory to be placed into a finite number of locations, referred to as the number of ways. For example, a 4-way set associative cache allows for each memory address to be placed in one of four possible locations in the cache. This design limits the amount of hardware complexity required in the physical chip layout and design, while exhibiting increased performance over a direct-mapped cache by reducing the number of block address conflicts. Direct-mapped caches can be thought to have a set associativity of 1-way, and fully associative caches can be thought of as N-way (where N is the number of cache entries). There are often diminishing returns as the number of ways is increased, and the number of way should be chosen carefully to provide adequate performance without introducing added complexity that can
increase memory access latency and power consumption.

### 2.1.3 Cache Memory Addressing

The address structure shown in 2.1 describes how a memory address is mapped into a set-associative cache [1]. Note that it is not necessary to store the block offset portion of the address for each block, because those bits all index into the same cache block and can be treated as a single entity. The tag portion of the block address is used to identify when block in the indexed cache block is the correct one to access.

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Index</td>
</tr>
</tbody>
</table>

### 2.1.4 Cache Replacement Policies

When a cache memory attempts to store a new cache block entry, but the cache set is already full, the cache must decide which one of the cache blocks to evict to make room for the new entry. The logic that controls how a block is chosen for eviction is referred to as the replacement policy for the cache.

Some commonly used cache replacement policies are First-In First-Out (FIFO) and Least Recently Used (LRU). A FIFO cache replacement policy assumes that older cache blocks are less likely to be reused, and uses counters to track the age of cache blocks, with the oldest cache
CHAPTER 2. BACKGROUND

block being chosen for eviction. The LRU cache replacement policy is similar to FIFO, with the modification that all accesses to the cache block reset the age counter, so that the age counters are instead tracking the last usage of each entry. There are also more complex cache replacement policies such as RRIP that can outperform LRU at the expense of additional design complexity and hardware elements [23]. If our address stream does not exhibit locality, we can use Random replacement. These scheme makes no assumptions about the likelihood of each cache block being reused and simply chooses a random cache block for eviction.

2.2 Microarchitectural Simulation

When designing next-generation microarchitectures, hardware designers rely on software simulation to evaluate the performance and reliability of their proposed designs. There are several different types of architectural simulators that each target different levels of detail, accuracy, and simulation runtime performance. A functional simulator captures the basic functionality of a design, but is not likely to be useful for evaluating the runtime performance of the system. This type of simulator would be sufficient when measuring the hit-rate in a cache memory system or other event-based details that are less timing dependent [12]. When the metric being studied involves time, a detailed timing simulator is required [19]. In the extreme case, a detailed full-system simulator may even be capable of running an entire operating system with a cycle-accurate simulated processor [11]. Full-system simulators can provide the most accurate simulation, but require a significant amount of knowledge of the hardware design and can have a prohibitive computational cost. Some simulators are capable of integrating temperature and power estimation, based a particular hardware layout model, measuring the activity level of different components using performance counters [24].

2.3 Nvidia Kepler Microarchitecture

The Nvidia Kepler GPU microarchitecture builds upon the previous Fermi microarchitecture by focusing on energy-efficient computing, making some architectural changes to the compute units, and adds some new capabilities including Dynamic Parallelism and Hyper-Q [3]. The Nvidia Tesla HPC series of GPUs is focused on server-grade GPU accelerators and offers impressive levels of computational power; the GK110 used in the Tesla K40 offers over 1 TFlop of double-precision throughput [3]. Hardware releases in the Tesla series that utilize the Kepler microarchitecture include the K20, K20X, K40, and K80. The K40 is considered a full implementation of the GK110 design,
CHAPTER 2. BACKGROUND

Figure 2.3: Image of Kepler GK110 (Source [3, 4])

and contains fifteen SMX and six 64-bit memory controllers. The K20 is a scaled-down implementation of the GK110 design and includes 13 SMX, five 64-bit memory controllers, and a smaller device memory. The K80 was released later and utilizes the GK210 design instead of GK110. The GK210 is similar to the GK110, but offers an increased CUDA compute capability level of 3.7, as well as additional registers and shared memory for each multiprocessor [3].

Nvidia Kepler devices utilize a CUDA compute capability in the 3.X series. The K20 and K40 have a CUDA compute capability of 3.5, while the K80 has a CUDA compute capability of 3.7 [3]. New atomic and shuffle CUDA intrinsics are introduced in GK110/210. Support for additional registers per SMX is introduced in the GK210 [3].

2.3.1 Streaming Multiprocessor - SMX

The main processing pipelines within a Kepler GPU are grouped into functional units called SMXs, rebranded from the Streaming Multiprocessor (SM) name used in previous microarchitectural generations. Each SMX contains 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units [3]. The Nvidia Kepler uses an Single-Instruction Multiple-Thread (SIMT) processing model in which similar threads are grouped into blocks of size 32 (i.e., warps). Note that the number of functional units are multiples of 32, so they can be executed concurrently across an entire warp.
As seen in Figure 2.3.1, each SMX also contains four warp schedulers and eight dispatch units, issuing enough instructions to keep the device busy. There is also a 64KB block of memory that can be split between shared memory and L1 cache. The possible splits for the Kepler GK110 are 16KB/48KB, 32KB/32KB, and 48KB/16KB between shared and L1 cache, respectively. The Kepler GK210 increases the amount of shared memory to 128KB, and offers similar splits between shared and L1 cache memory [3]. The Kepler also offers 48KB of read-only data cache for constant memory. This offers an improvement over the Fermi generation of GPUs in that this memory can be used directly through CUDA intrinsics, and does not require the use of indirect access through the texture units [3].

2.3.2 Memory Hierarchy

The memory hierarchy for the Kepler microarchitecture is shown in Figure 2.3.2. Each GK110 SMX contains 64KB of shared memory an L1 cache, and 48KB of read-only constant/texture memory. Beyond the scope of an individual SMX, a thread has access to a larger amount of shared L2 cache memory and device memory Dynamic Random Access Memory (DRAM).
Kepler devices do not use the L1 data cache by default for global memory accesses, and first use the L2 cache before going out to device memory DRAM. This behavior can be modified in some Kepler systems using the `-dlcm=ca` nvcc compiler flag. This default behavior is a departure from the Fermi architecture, which used both L1 and L2 caches by default and had a compiler flag option to disable L1 and use only the shared L2 cache. When enabled, L1 caches use a portion of the 64KB of memory per SMX that is allocated for shared memory and L1 usage. The L2 cache sizes can vary across hardware implementations. For example, the K20 and K40 have 1280KB and 1536KB of shared L2 data cache memory, respectively.

To use the read-only/texture memory within a CUDA application, a developer can use either the `const`, `restrict` or the `ldg()` CUDA keywords [3]. The shared memory within an SMX can be utilized by different threads within the same thread block and can be declared using the `__shared__` CUDA specifier. This shared memory space allows for some limited inter-thread communication and data sharing within a thread block.
Chapter 3

Related Work

3.1 Multi2Sim Heterogeneous System Simulator

This thesis focuses primarily on understanding the Kepler memory hierarchy to improve the accuracy of the Multi2Sim Kepler simulator [25, 19]. Multi2Sim was originally developed to target the MIPS ISA, but has since grown to include the x86 and ARM CPUs, and AMD Southern Islands, Nvidia Kepler, and HSA architectures GPU architectures [26]. This work primarily focuses on the detailed timing implementation of the Nvidia Kepler microarchitecture, as modeled by Multi2Sim.

3.1.1 Kepler Instruction Set Architecture

While the specific details of the native Kepler assembly instructions Streaming Assembly (SASS) are not made available publicly, some information about SASS can be deduced through the Parallel Thread Execution (PTX) Intermediate Representation (IR) (i.e., the NVIDIA intermediate representation assembly), and the cuobjdump utility, which prints a listing of SASS instructions from a binary file [27]. There have also been some research efforts to reproduce the SASS format and instructions for the Nvidia Kepler microarchitecture [19, 28]. A detailed understanding of the Kepler ISA is of needed to produce accurate execution within anx architectural simulator. Inaccurate or unsupported ISA instructions will limit the number of binary executables that will execute correctly and limit the overall utility of the simulator.
CHAPTER 3. RELATED WORK

3.1.2  Kepler Compute Pipeline

The Kepler compute pipeline within the Multi2Sim simulator was largely adapted from the AMD Southern Islands GPU microarchitecture, with some modifications to support the naming conventions and architectural behavior specific to the Nvidia Kepler microarchitecture. The Kepler simulation target device is the Nvidia Kepler K20c and is referred to as the Multi2Sim Tesla K20c. The pipeline contains several functional units that are based on the Kepler SMX functional components.

3.2  Microbenchmarking

Microbenchmarking is the process of using carefully designed access patterns to uncover microarchitectural behavior about the underlying system. This work uses memory access timings based on the CUDA device clock to uncover information about the cache memory hierarchy used in the Nvidia Kepler microarchitecture. This thesis leverages the pchase microbenchmarking algorithm described by Saavedra-Barrera [29], and later extended by Mei and Chu [5], to uncover some of the design details about the Nvidia K20c memory hierarchy.

3.2.1  Pointer Chasing Algorithm

The pointer chasing or pchase algorithm is useful for understanding the characteristics of a processor’s memory hierarchy [29, 30]. This methodology has been extended to GPU memory hierarchies as well [31, 5]. Our research builds upon the pchase algorithm discussed by Mei and Chu [5], and our results generally agree with these earlier findings. The targeted device in prior work was an Nvidia GTX 780 [5], which is a consumer-grade desktop implementation of the GK110 design. Mei and Chu’s work examines both the tvalue-s [31] and tvalue-N [29] methodologies for estimating the cache size, associativity, and cache way size parameters.

```plaintext
\begin{verbatim}
\\\n// Initialization of memory addresses
for (int i = 0; i < array_size; i++) {
    A[i] = (i + stride) / array_size;
}
\\
// Pointer Chase
\end{verbatim}
```

Listing 3.1: Pointer Chase Algorithm (Source [5])
A key attribute of the pchase algorithm is that each memory access depends on the successful resolution of the previous one. These data dependencies allow the algorithm to avoid instruction level parallelism, memory level parallelism, and data prefetching behavior that can confuse experimental results. Isolating each memory access helps to provide a clearer picture of the caching behavior of the device. This algorithm works well in practice for the Nvidia Kepler Texture/Read-only cache, as well as the L1 data cache, but the picture is murkier when examining the L2 data cache.

A similar approach was applied by Wong et al. [31] in an attempt to understand the Tesla microarchitecture of the Nvidia GTX 280. This prior study examines the Texture cache and Translation Lookaside Buffer (TLB) caches within the Tesla microarchitecture. Their work produced estimates for the cache size, cache way size, and cache line size based on the tvalue-N graph [31, 5].
Chapter 4

Microbenchmark Experiments

Our strategy in this thesis includes performing several timing experiments to better understand the read memory access latency for the Nvidia Kepler microarchitecture for all levels of the memory hierarchy. Timing results are shown for the Texture/Read-only cache, the L1 data cache, the L2 data cache, and device memory DRAM. The pointer chasing algorithm was also used to estimate various cache parameters within the Kepler memory hierarchy.

4.1 Latency Timing Experiments

The approach used to measure the mean round-trip latency for each level of the memory hierarchy is fairly straightforward. A single-threaded microbenchmark running the pchase memory access algorithm, as described by Saavendra-Barrera [5], is run for each level in the memory hierarchy. The memory access timings are measured using the Nvidia CUDA `clock()` function, which returns the number of device clock ticks. The difference between the start time and end time can be used to deduce the memory access latency for a single access and is measured in GPU clock cycles. By applying a loop-unrolling technique and gradually increasing the number of memory accesses the timing measurement can attempt to remove the `clock()` timing overhead and isolate the timing for a single memory access. The arithmetic mean of a large number of these timings is then used to estimate the memory hit latency at each memory level of the hierarchy. The `nvprof` utility is used to report on the number of memory accesses being serviced from the targeted memory level at the expected rate. A two-pass microbenchmark is used to produce a 50% hit-rate, with compulsory misses recorded on the first load of the data into the cache, and then on the second pass, accesses hit in the cache.
CHAPTER 4. MICROBENCHMARK EXPERIMENTS

4.1.1 Timing Measurements

Listing 4.1: Empty Timing Loop

```c
/* 2570 */ S2R R6, SR_CLOCKLO;  // begin = clock();
/* 2578 */ S2R R7, SR_CLOCKLO;  // end = clock();
/* 2588 */ ISUB R6, R7, R6;   // diff = end − begin;
/* 2590 */ STS [R8], R6;    // s_timings[i] = diff;
```

Listing 4.1 shows the SASS output for an empty loop generated by running the pchase algorithm without any memory operations. This empty loop requires 16 cycles to execute and may be indicative of the minimum overhead associated with `clock()` measurements.

Listing 4.2: Unrolled Timing Loop

```c
/* 25a0 */ S2R R6, SR_CLOCKLO;  // begin = clock();
/* 25a8 */ LD.E R4, [R10];     // j = A[j];
/* 25b0 */ MOV32I R7, 0x4;
/* 25b8 */ IMAD.U32.U32 R12.CC, R4, R7, c[0x0][0x140];
/* 25c8 */ IMAD.U32.U32.HI.X R13, R4, R7, c[0x0][0x144];
/* 25d0 */ LD.E R4, [R12];    // j = A[j];
/* 25d8 */ IMAD.U32.U32.R10.CC, R4, R7, c[0x0][0x140];
/* 25e0 */ IMAD.U32.U32.HI.R11, R4, R7, c[0x0][0x144];
/* 25e8 */ LD.E R4, [R10];   // j = A[j];
/* 25f0 */ STS [R9], R4;    // shared.a[i] = j;
/* 25f8 */ S2R R7, SR_CLOCKLO; // end = clock();
```

The overhead of the `clock()` and `shared.a[i] = j` operations can be mitigated by using the timings from multiple unrolled loops of increasing size. Listing 4.2 shows an example of an unrolled loop that performs three successive global memory loads. The timings reported in this study include a small overhead due to the memory address computation. The timing results for each memory level in the K20c and K40c are shown in Chapter 6.
CHAPTER 4. MICROBENCHMARK EXPERIMENTS

4.1.2 Texture / Read-only Cache

The memory access latency for the Texture/Read-only cache can be measured with a minor modification of the pchase algorithm. Targeting the read-only cache, we implemented our kernel by using the \texttt{ldg()} CUDA intrinsic to tell the nvcc compiler that it is safe to cache this data in the read-only memory space. It is also possible to target the read-only portion of memory using the \texttt{const} \texttt{restrict} CUDA keywords.

4.1.3 L1

Similarly, the L1 data cache can be targeted within the Kepler GPU by using the \texttt{-dlcm=ca} nvcc compiler flag when compiling a CUDA program. This allows the Kepler device to store data in L1. Note that this behavior is not available on all Kepler devices and so the compiler flag may be ignored if the feature is not available. For example, the Tesla K20c is unable to cache data in L1, and will effectively ignore the compiler flag, while the driver will allow caching if targeting the Tesla K40c.

4.1.4 L2

The default behavior within the Nvidia Kepler microarchitecture for global memory accesses is to cache them in the L2. No special compiler flags are required to exercise this behavior.

4.1.5 Device Memory

The access latency for the GPU device memory is equivalent to the timing for a miss in the L2 cache. The device memory DRAM contains the entire working memory for the application, and data can be explicitly copied from host-to-device and device-to-host with the CUDA \texttt{cudaMemcpy()} function.

4.2 Measuring Cache Parameters

Our measurement strategy relies on a few assumptions about the underlying cache structure. One assumption is that the cache sets are divided evenly and exhibit a uniform replacement policy. The set-associative cache model, as described in Chapter 2, is assumed as the general model used when exploring the various Kepler cache parameters. If there were significant deviations from these
CHAPTER 4. MICROBENCHMARK EXPERIMENTS

underlying assumptions, such as a non-uniform set organization, or non-standard memory addressing, this would impact our investigation utilizing the microbenchmarks and their subsequent results.

\[
t_{\text{avg}} = \frac{1}{N} \sum_{i=1}^{N} t_{i,\text{hit}} q(i) + t_{i,\text{miss}} 1 - q(i)
\]

(4.1)

\[
q(i) = \begin{cases} 
0 & \text{if } f(i) \text{ is cache miss} \\
1 & \text{if } f(i) \text{ is cache hit}
\end{cases}
\]

(4.2)

Table 4.1: Set-associative Cache Parameters (Source [5])

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>cache size</td>
</tr>
<tr>
<td>b</td>
<td>cache line size</td>
</tr>
<tr>
<td>a</td>
<td>cache associativity</td>
</tr>
<tr>
<td>T</td>
<td>number of cache sets</td>
</tr>
</tbody>
</table>

This work uses the set-associative cache notation described by Mei and Chu [5], and assumes that the total size of the cache can be determined by the product of the number of cache sets, the cache block size, and the cache associativity, as shown in Equation 4.3 [1, 5].

\[
T \times a \times b = C
\]

(4.3)

4.2.1 Cache Memory Addressing

As shown in Table 2.1, the memory addressing scheme for a traditional set-associative cache can be determined by the cache block size \(b\) and set associativity \(a\). This work assumes that the cache memory addresses follow a typical set associative addressing scheme, as described in Chapter 2. The cache memory address must contain enough bits to represent the tag, set, and offset. One of the example use cases described in the SASSI GPU instrumentation software computes the cache block address by doing a bit shift of 5 bits [32]. This bit shifting method agrees with the 32 byte value that was experimentally determined in this study (\(2^5 = 32\)).

4.2.2 Overall Cache Size

The total cache size can be determined by performing a pchase microbenchmarking experiment that slowly increases the working set size of the benchmark, and examines the arithmetic
mean of the memory access latency. See Appendix A for a review of the arithmetic and other Pythagorean means.

### 4.2.2.1 Read-only / Texture Cache

Figure 4.1: K40c Read-only / Texture Cache

The Read-only or Texture cache exhibits a very clear step function when using the pchase algorithm. Figure 4.1 shows the experimental results for the algorithm, as applied to the Nvidia Tesla K40c Texture cache. In this figure, all of the working sets below 12KB contain exclusively texture cache hits, and all working sets above 12.5KB produce texture cache misses. The four observed steps indicate that the Texture cache is four-way set-associative, based on the tvalue-N cache parameter estimation method [5].
The L1 cache is unused for global memory accesses by default in Kepler, but is enabled with the `-Xptxas -dlcm=ca` compiler flag. The L1 data cache uses a variable capacity of 16/32/48KB per SMX, with the remaining space being allocated for shared memory. Figure 4.2 shows the pchase algorithm, as applied to the Kepler K40c L1 cache. Below 16KB, all of the cache accesses hit in the L1 data cache, and above 20KB, all memory accesses miss in L1 and hit in L2. Between these two points, the average memory access increases linearly due to the linear increase in the number of L1 cache misses.
The L2 data cache is the first level cache for global memory accesses by default, and is shared across all of the SMXs. The L2 cache in the Kepler microarchitecture is larger and more complex than the L1 and Texture caches. Figure 4.3 shows the average read access latency, as a function of working set size. This pattern is much noisier than the figures for the Texture and L1 caches, so it would be unwise to suggest that this pattern can be explained neatly by either the tvalue-N or tvalue-s cache parameter estimation methods [5].

4.2.3 Cache Block Size

The block size of a cache memory can also be determined through microbenchmarking. Since we have previously determined the mean access latency of hits and misses at each cache level, we can choose a threshold between these values and use it to classify individual memory accesses as either hits or misses. This binary classification of accesses as hits and misses makes it easier to detect patterns in the trace.

To calculate the cache block size, we can gradually increase the cache access stride and examine the hit and miss pattern. The use of the pChase algorithm for this microbenchmark minimizes
the effects of prefetching and ILP optimizations, which can influence the results. Once the stride has exceeded the cache block size, the pattern should contain only misses, and this value can then be used as an estimate of the cache block size.

![Figure 4.4: Estimating Cache Block Size By Varying Stride](image)

The experimental hit rates used to determine these cache block sizes were validated with the Nvidia profiler `nvprof` tool, to ensure that the memory accesses were actually being served from the expected cache memory level.

### 4.2.4 Set Associativity

After applying the tvalue-N method to Figures 4.1 and 4.2, we found the values for the set associativity of both the Texture cache and the L1 data caches were 4-way set associative. The data for the L2 cache is shown in Figure 4.3, which too noisy to directly apply the tvalue-N technique.

### 4.2.5 Replacement Policy

Based on the application of the tvalue-N methodology to Figures 4.1 and 4.2, the cache replacement for the Texture cache is determined to be LRU, but the L1 data cache is unknown, and is likely not to be LRU. These estimates are in agreement with the findings from Mei and Chu [5].
Chapter 5

Multi2sim Software Integration

5.1 Introduction

This work aims to improve the accuracy of the Nvidia Kepler model within the Multi2Sim Kepler simulator. The Multi2Sim Kepler model is based on the Nvidia Tesla K20c hardware. Our implementation is based on the parameters deduced using both the Tesla K20c and Tesla K40c hardware, with the goal of improving simulator accuracy when compared to hardware runtimes.

This work builds upon the Nvidia Kepler model version of the Multi2sim model in the multi2sim-kepler and iiswc2016 branches of the Multi2Sim GitHub code repository. While the Nvidia Kepler ISA is only partially supported, the supported 32-bit instructions are sufficient to support the execution of the Nvidia CUDA SDK 6.5 applications, which we use in this thesis in our performance evaluation.

5.1.1 Modeling the Memory Hierarchy

The Kepler LSU model from the iiswc2016 branch was incomplete as originally modeled, and simply acted as pass-through with a constant memory access timing latency. This meant that even though the Kepler simulation model contained a memory hierarchy complete with L1 caches, an L2 cache, and device memory, the models were not actually being utilized during simulation. Since the load and store instructions were not using the memory hierarchy, memory usage statistics were not meaningful. This research added logic to the LSU so that the caches and device memory, as specified in the configuration, are actually enabled and utilized during detailed timing simulation.

The original Kepler memory hierarchy, as modeled in the iiswc2016 branch, contains
CHAPTER 5. MULTI2SIM SOFTWARE INTEGRATION

a 16KB L1 cache for each SMX, and six shared L2 caches (each 64KB), for a total L2 capacity of 384KB. The L2 cache block size is also incorrect for Kepler, using a 128B block, instead of the 32B block. This configuration is likely based on the original Fermi simulation target design, and was never updated appropriately for the Kepler memory hierarchy.

5.2 Validation

To compare the accuracy of the updated memory hierarchy, we can measure the timing error relative to the physical K20c hardware for both the original memory hierarchy model in Multi2sim,
and our updated version that include our enhancements. We expect that the accuracy of the model should improve, and should be closer to the execution time of the physical hardware. Several tests are performed that leverage the benchmarks in the CUDA 6.5 SDK. The results of these timing experiments are presented in Chapter 6.

5.3 Future Work

While this work makes improvements to the detailed Kepler timing model in Multi2Sim, there are still a few areas where the model could benefit from additional development effort. As is the case in the Nvidia Tesla K20c and K40c, the cache sizes used in physical hardware are not necessarily powers of two, and this implementation issue in Multi2Sim needs to be fixed. A second limitation in the Kepler model is the requirement for 32-bit program compilation on the host through the usage of the \texttt{-m32} compiler flag. The overall utility of the Multi2Sim simulator would improve significantly with added 64-bit application support as this would allow for the execution of many additional benchmarks and programs. This change is currently in progress.
Chapter 6

Simulator Accuracy

6.1 Kepler Microbenchmarking Results

Table 6.1: Measured Access Latencies (clock cycles)

<table>
<thead>
<tr>
<th></th>
<th>L1</th>
<th>Tex</th>
<th>L2</th>
<th>DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>K40c</td>
<td>46</td>
<td>132</td>
<td>216</td>
<td>308</td>
</tr>
<tr>
<td>K20c</td>
<td>-</td>
<td>143</td>
<td>222</td>
<td>331</td>
</tr>
</tbody>
</table>

Table 6.1 shows the round-trip access times for each level in the memory hierarchy, as measured for a Nvidia Tesla K40c. The units are GPU clock cycles as returned by the `clock()` function. These timings are collected using the default compilation flags, and we computed the mean value from a large number of memory access measurements. The K20c does not support enabling the L1 via a compiler flag, so the K40c was used for these experiments.

Table 6.2: Nvidia Kepler Cache Parameters

<table>
<thead>
<tr>
<th></th>
<th>Texture</th>
<th>L1</th>
<th>L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Size</td>
<td>32B</td>
<td>128B</td>
<td>32B</td>
</tr>
<tr>
<td>Cache Size</td>
<td>12KB</td>
<td>16/32/48KB</td>
<td>1280/1536KB</td>
</tr>
<tr>
<td>Associativity</td>
<td>4-way</td>
<td>4-way</td>
<td>-</td>
</tr>
<tr>
<td>Cache Replacement</td>
<td>LRU</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The texture cache was experimentally determined to have a 32B block size, 4-way set associativity, and an LRU cache replacement policy. These results were obtained using the methodologies described in Chapter 4. These values are consistent with prior studies and the information provided by Nvidia.
CHAPTER 6. SIMULATOR ACCURACY

Similarly, the L1 data cache parameters were estimated based on the procedures described in Chapter 4. The L1 data cache was found to have a 128B cache block size, 4-way set associativity, but an unknown cache replacement policy. These findings are consistent with the limited information available.

The methodology used for determining the texture cache and L1 data cache parameters was not able completely expose the L2 cache parameters. The cache block size was determined to be 32B, and the total sizes of the L2 are 1280KB (K20c) and 1536KB (K40c). There are still too many unknowns to directly apply the techniques that worked for the L1 and texture caches.

6.2 Multi2Sim Evaluation Results

Table 6.3: Nvidia CUDA 6.5 Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CppIntegration</td>
<td>Feature</td>
<td>Nvidia CUDA 6.5 sample that integrates CUDA into an existing C++ application/ framework.</td>
</tr>
<tr>
<td>InlinePTX</td>
<td>Feature</td>
<td>Nvidia CUDA 6.5 sample that demonstrates how to use inline PTX (assembly language) directly in a CUDA kernel.</td>
</tr>
<tr>
<td>MatrixMul16</td>
<td>Compute</td>
<td>Nvidia CUDA 6.5 sample that performs a matrix multiplication. This matrix dimensions have been modified from 128x128 to 16x16 to increase simulation performance.</td>
</tr>
<tr>
<td>VectorAdd</td>
<td>Compute</td>
<td>Nvidia CUDA 6.5 sample that performs vector addition. The default vector length is 50,000 elements.</td>
</tr>
<tr>
<td>SimpleMultiGPU</td>
<td>Feature</td>
<td>Nvidia CUDA 6.5 sample that demonstrates how to leverage multiple GPUs through the CUDA API. Note that only one GPU was utilized for these experiments.</td>
</tr>
</tbody>
</table>

Table 6.3 lists the Nvidia CUDA 6.5 SDK programs that were used in our detailed timing analysis of the Kepler model within Multi2Sim. These benchmarks can be divided into two categories, feature-driven workloads and compute-driven workloads. The feature-driven workload do not perform much computation, but serve as a reference code that highlights one of the CUDA SDK features. The compute-driven examples perform significant computation that more closely reflects the performance of an actual CUDA kernel.
CHAPTER 6. SIMULATOR ACCURACY

6.2.1 Benchmark Instruction Results

Figure 6.1: Load/Store Instructions

Figure 6.1 shows how the number of Load/Store instructions scale with the array size in the VectorAdd benchmark. The number of Load/Store Instructions exactly matches the metric returned from running on the Nvidia K40c hardware. This indicates that the multi2sim code is successfully dissasembling these instructions and handling the correct number of memory transactions.
Figure 6.2 shows that the number of instructions used by Multi2Sim Kepler scales consistently with the K40c, but does not match exactly. This shows that representative performance may be achievable with the current model, but not cycle-accurate performance is not yet possible. Instruction composition results are shown for both the feature-driven and compute-driven workloads. The instruction composition results are shown in Table 6.3.
The three benchmarks that exhibit the highest percentage of memory instructions are used to evaluate the memory hierarchy changes in this study. The three benchmarks that show the highest memory activity include the two compute-driven code samples (VectorAdd and MatrixMul16), as well as one of the feature-driven samples (CppIntegration).

Figure 6.3: SDK Instruction Breakdown
CHAPTER 6. SIMULATOR ACCURACY

6.2.2 VectorAdd Scaling Results

Figure 6.4: Scaled Timing Errors

The VectorAdd benchmark was run for several different array size between 32 and 2048 array elements. The results of our scaling study are shown in Table 6.4. The performance disparities between the updated version of Multi2Sim Kepler actually decrease as a function of the array size. The accuracy of the memory model is vastly improved over the original Multi2Sim Kepler implementation, which shows much larger disparities across all array sizes.

The baseline version of Multi2Sim does not utilize the memory hierarchy elements in the Load/Store Unit, and simply uses a fixed timing value that scales linearly with the array size. The more detailed version of the Multi2Sim Kepler timing model however, does include the full memory hierarchy, and so accuracy is very good, especially for the larger array sizes.
CHAPTER 6. SIMULATOR ACCURACY

6.2.3 CUDA Benchmark Results

Figure 6.5: CUDA Benchmark Comparison

Closer to 1 is better

![CUDA Benchmark Comparison Chart](image)

Figure 6.5: CUDA 6.5 Benchmark Results

Figure 6.5 shows the timing results for the three memory intensive benchmarks as run on the Multi2Sim Kepler detailed timing simulator. These three benchmarks, CppIntegration, MatrixMul16, and VectorAdd, all exhibit improved accuracy when using the enhanced memory model in Multi2Sim. The timing results shown are normalized to the K20c hardware, which is used as a baseline. The CppIntegration, MatrixMul16, and VectorAdd workloads exhibit an improvement of 13.9%, -8.4%, and 26.8%, respectively. This produces an average timing improvement of 10.8%, for these memory intensive benchmarks.
CHAPTER 6. SIMULATOR ACCURACY

Figure 6.6: CUDA 6.5 Benchmark Error

The absolute difference between modeled and measured is shown for these benchmarks in Figure 6.6. The chart plots the absolute value of the timing difference as a percentage, normalized with respect to the actual K20c runtime. The absolute error is reduced for the CppIntegration and VectorAdd benchmarks.
Chapter 7

Conclusion

This thesis explored reverse engineering a number of microarchitectural attributes of the caching behavior and overall memory hierarchy of the NVIDIA Kepler microarchitecture. Our methodology involved using a pointer chasing (pchase) microbenchmark to measure access latencies across the different cache levels. The goal was to discover some of the key cache parameters of each cache level.

These timings and cache parameters were then used to improve the Nvidia Kepler detailed timing simulation within the Multi2Sim simulator. The revised memory hierarchy has been configured to closely match the memory hierarchy of the Nvidia Tesla K20c. For the benchmarks studied, we found that the accuracy of the memory system has been significantly improved. We leveraged the following memory-intensive CUDA SDK kernels: CppIntegration, MatrixMul16, and VectorAdd. Performance accuracy was improved by an average of 10.8% as compared to the Nvidia K20c physical hardware.

To improve the Kepler memory model further, there needs to be more focus on the L2 cache. The exploration should include the impact of L2 banking, and the on-chip memory interconnect that tie the SMXs to the L2 banks. We anticipate that the Multi2sim will continue to grow in importance as new benchmark suites are supported. It will be critical to demonstrate the fidelity of these workloads, which will be more susceptible to second-order effects.
Bibliography


BIBLIOGRAPHY


Appendix A

Mathematics

Arithmetic Mean
\[
\bar{x}_a = \frac{x_1 + x_2 + \ldots + x_n}{n} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{A.1}
\]

Geometric Mean
\[
\bar{x}_g = \sqrt[n]{x_1 \cdot x_2 \cdot \ldots \cdot x_n} = \left( \prod_{i=1}^{n} x_i \right)^{1/n} \tag{A.2}
\]

Harmonic Mean
\[
\bar{x}_h = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}} = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}} \tag{A.3}
\]