GPU Compute Memory Systems

by

George Lai Yuan

B.A.Sc., The University of British Columbia, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

November, 2009

© George Lai Yuan 2009
Abstract

Modern Graphic Process Units (GPUs) offer orders of magnitude more raw computing power than contemporary CPUs by using many simpler in-order single-instruction, multiple-data (SIMD) cores optimized for multi-thread performance rather than single-thread performance. As such, GPUs operate much closer to the “Memory Wall”, thus requiring much more careful memory management.

This thesis proposes changes to the memory system of our detailed GPU performance simulator, GPGPU-Sim, to allow proper simulation of general-purpose applications written using NVIDIA’s Compute Unified Device Architecture (CUDA) framework. To test these changes, fourteen CUDA applications with varying degrees of memory intensity were collected. With these changes, we show that our simulator predicts performance of commodity GPU hardware with 86% correlation. Furthermore, we show that increasing chip resources to allow more threads to run concurrently does not necessarily increase performance due to increased contention for the shared memory system.

Moreover, this thesis proposes a hybrid analytical DRAM performance model that uses memory address traces to predict the efficiency of a DRAM system when using a conventional First-Ready First-Come First-Serve (FR-FCFS) memory scheduling policy. To stress the proposed model, a massively multithreaded architecture based upon contemporary high-end GPUs is simulated to generate the memory address trace needed. The results show that the hybrid analytical model predicts DRAM efficiency to within 11.2% absolute error when arithmetically averaged across a memory-intensive subset of the CUDA applications introduced in the first part of this thesis.

Finally, this thesis proposes a complexity-effective solution to memory scheduling that recovers most of the performance loss incurred by a naive in-order first-in first-out (FIFO) DRAM scheduler compared to an aggressive out-of-order FR-FCFS scheduler. While FR-FCFS scheduling re-orders memory requests to improve row access locality, we instead employ an interconnection network arbitration scheme that preserves the inherently high row access locality of memory request streams from individual “shader cores” and, in doing so, achieve DRAM efficiency and system performance close to that of FR-FCFS with a simpler design. We evaluate our interconnection network arbitration scheme using crossbar, ring, and mesh networks and show that, when coupled with a banked FIFO in-order scheduler, it obtains up to 91.0% of the performance obtainable with an out-of-order memory scheduler with eight-entry DRAM controller queues.
# Table of Contents

Abstract .................................................................................................................. ii
Table of Contents ..................................................................................................... iii
List of Tables ........................................................................................................ vii
List of Figures .......................................................................................................... vi
Glossary .................................................................................................................. x
Acknowledgements ................................................................................................ xi

1 Introduction ......................................................................................................... 1
1.1 Motivation ......................................................................................................... 2
1.2 Contributions ..................................................................................................... 3
1.3 Background ....................................................................................................... 5
   1.3.1 SIMD GPU architecture ......................................................................... 5
   1.3.2 Modern DRAM memory .......................................................................... 6
   1.3.3 Interconnection networks ....................................................................... 11
1.4 Organization ..................................................................................................... 12

2 Supporting CUDA Applications on GPGPU-Sim .................................................. 15
2.1 CUDA memories .............................................................................................. 15
   2.1.1 Local memory .......................................................................................... 17
   2.1.2 Texture memory ...................................................................................... 17
   2.1.3 Constant memory ................................................................................... 19
2.2 CUDA benchmarks ......................................................................................... 20

3 Hybrid Analytical Performance Modeling of DRAM ............................................ 25
3.1 DRAM efficiency ............................................................................................... 25
3.2 Hybrid analytical model ................................................................................... 26
3.3 Processing address traces ............................................................................... 30
# Table of Contents

4 Complexity-Effective Memory Access Scheduling ............................................. 36  
4.1 Motivation ........................................ 36  
4.2 Effect of interconnect on row locality .................................................. 40  
4.3 Grant holding interconnect arbitration ................................................... 41  
4.4 Interleaving due to multiple virtual channels ......................................... 44  
4.5 Complexity comparison ........................................................................... 45  

5 Methodology .................................................................................................... 49  
5.1 Supporting CUDA applications on GPGPU-Sim ....................................... 49  
5.2 Hybrid analytical modeling of DRAM ....................................................... 53  
5.3 Complexity-effective memory access scheduling .................................... 55  

6 Experimental Results ..................................................................................... 57  
6.1 GPGPU-Sim simulation .............................................................................. 57  
6.1.1 Baseline ................................................................................................. 57  
6.1.2 DRAM utilization and efficiency ........................................................... 59  
6.1.3 Are more threads better? ...................................................................... 61  
6.2 Hybrid analytical modeling of DRAM ....................................................... 64  
6.2.1 DRAM efficiency prediction accuracy ................................................... 64  
6.2.2 Modeling error analysis ........................................................................ 67  
6.2.3 Sensitivity analysis ................................................................................ 68  
6.3 Complexity-effective memory access scheduling .................................... 72  
6.3.1 Complexity-effective DRAM scheduling performance ......................... 72  
6.3.2 Row streak breakers ............................................................................. 74  
6.3.3 Static virtual channel assignment ......................................................... 76  
6.3.4 Sensitivity analysis ................................................................................. 76  

7 Related Work .................................................................................................... 79  
7.1 GPU architecture simulation ..................................................................... 79  
7.2 Analytical models ....................................................................................... 80  
7.3 DRAM controller designs ........................................................................... 81  

8 Conclusions and Future Work ....................................................................... 83  
8.1 Conclusions ............................................................................................... 83  
8.2 Future work ................................................................................................ 85  
8.2.1 Improving the analytical DRAM model .............................................. 85  
8.2.2 Combining the analytical DRAM model with analytical processor models 85  
8.2.3 Design space exploration of intelligent interconnects ......................... 86  

Bibliography ........................................................................................................ 87
List of Tables

1.1 DRAM parameters ...................................................... 7

2.1 Benchmark thread organization .................................. 20
2.2 Benchmark Properties ................................................. 21

4.1 Calculated complexity .................................................. 47

5.1 Microarchitectural parameters for testing GPGPU-Sim ................. 50
5.2 Example showing difference between IPC versus IPC weighted by cycle .... 53
5.3 Microarchitectural parameters for analytical DRAM model ................ 53
5.4 Benchmarks for analytical DRAM model ............................... 54
5.5 Microarchitectural parameters for complexity-effective memory access scheduling .... 55
5.6 Benchmarks for complexity-effective memory access scheduling ....... 55
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Average memory latency measured in performance simulation</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Memory system organization</td>
<td>8</td>
</tr>
<tr>
<td>1.3</td>
<td>Illustration of timing constraints</td>
<td>9</td>
</tr>
<tr>
<td>1.4</td>
<td>Typical interconnect router architecture</td>
<td>11</td>
</tr>
<tr>
<td>1.5</td>
<td>Block diagram of a Parallel Iterative Matching separable allocator</td>
<td>13</td>
</tr>
<tr>
<td>2.1</td>
<td>Simulated pipeline architecture</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>Timing diagram example</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>Sliding window profiling technique example</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Sliding window profiling algorithm for processing memory request address traces</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>Measured DRAM efficiency for random uniform memory traffic with varying row access locality</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>Conceptual example showing effect of request interleaving due to interconnect</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>Architecture</td>
<td>40</td>
</tr>
<tr>
<td>4.4</td>
<td>Measured row access locality before the interconnect and after</td>
<td>41</td>
</tr>
<tr>
<td>4.5</td>
<td>Grant-holding interconnect router architecture</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>Measured DRAM efficiency and utilization</td>
<td>54</td>
</tr>
<tr>
<td>6.1</td>
<td>Instruction Classification</td>
<td>58</td>
</tr>
<tr>
<td>6.2</td>
<td>Memory Instructions Breakdown</td>
<td>58</td>
</tr>
<tr>
<td>6.3</td>
<td>Baseline performance</td>
<td>59</td>
</tr>
<tr>
<td>6.4</td>
<td>Performance comparison with 8600GTS</td>
<td>59</td>
</tr>
<tr>
<td>6.5</td>
<td>Impact of DRAM memory controller optimizations</td>
<td>60</td>
</tr>
<tr>
<td>6.6</td>
<td>DRAM Efficiency</td>
<td>60</td>
</tr>
<tr>
<td>6.7</td>
<td>DRAM Utilization</td>
<td>60</td>
</tr>
<tr>
<td>6.8</td>
<td>Effects of varying number of CTAs</td>
<td>62</td>
</tr>
<tr>
<td>6.9</td>
<td>Comparison of measured DRAM efficiency to predicted DRAM efficiency</td>
<td>64</td>
</tr>
<tr>
<td>6.10</td>
<td>Arithmetic mean absolute error of predicted DRAM efficiency</td>
<td>65</td>
</tr>
<tr>
<td>6.11</td>
<td>Comparison of measured DRAM efficiency to predicted DRAM efficiency</td>
<td>66</td>
</tr>
</tbody>
</table>
List of Figures

6.12 Average row access locality ........................................... 68
6.13 Average DRAM controller queue occupancy ..................... 69
6.14 Arithmetic mean absolute error versus DRAM controller queue size ........................................... 69
6.15 Arithmetic mean absolute error versus number of DRAM chips per DRAM controller ........................................... 70
6.16 Error of predicted DRAM efficiency of “Most Pending” memory scheduling algorithm ........................................... 71
6.17 Baseline configuration results ........................................... 72
6.18 DRAM efficiency and memory latency .............................. 73
6.19 Row streak breakers .......................................................... 75
6.20 Harmonic mean IPC for different virtual channel configurations ........................................... 76
6.21 Harmonic mean IPC across all applications for different network topologies ........................................... 77
6.22 Harmonic mean IPC normalized to FR-FCFS for different DRAM controller queue sizes ........................................... 78
7.1 Average number of unique banks requested per warp memory operation ........................................... 81
## Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTA</td>
<td>Cooperative Thread Array, a collection of threads that run on a single shader core. Threads in a CTA share a common fast memory and can perform barrier synchronization [5], [86]</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture, a framework for running non-graphic parallel applications on GPUs, [86]</td>
</tr>
<tr>
<td>DRAM efficiency</td>
<td>Percentage of time that the DRAM chip is transferring data across its bus over only the period of time when the DRAM is active, [86]</td>
</tr>
<tr>
<td>DRAM utilization</td>
<td>Percentage of time that a DRAM chip is transferring data across its bus over an arbitrary length of time, [86]</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-In First-Out, [86]</td>
</tr>
</tbody>
</table>
**FR-FCFS**  First-Ready First-Come First-Serve, an out-of-order memory scheduling algorithm that reorders requests to maximize row access locality. [86]

**GPU**  Graphic Processing Unit. [86]

**HG**  Hold Grant, an interconnect allocation policy that prioritizes the allocation decisions made in the previous cycle. [86]

**HM**  Harmonic Mean. [86]

**HMHG**  Hash-Matching Hold Grant, an interconnect allocation policy that prioritizes the allocation decisions made in the previous cycle only if the hash of the destination row of the pending request matches the hash of the destination row of the previously granted request for the same input-output combination. [86]

**PIM**  Parallel Iterative Matching, a separable allocation policy based on random prioritization of inputs and outputs. [86]
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMHG</td>
<td>Row-Matching Hold Grant, an interconnect allocation policy that prioritizes the allocation decisions made in the previous cycle only if the destination row of the pending request matches the destination row of the previously granted request for the same input-output combination, <a href="#">86</a></td>
</tr>
<tr>
<td>Row access locality</td>
<td>Average number of back-to-back memory requests sent to the same row of the same bank of the same DRAM chip, <a href="#">86</a></td>
</tr>
<tr>
<td>SIMD</td>
<td>Single-Instruction Multiple-Data, <a href="#">86</a></td>
</tr>
<tr>
<td>Warp</td>
<td>A group of 32 threads that run in lock-step, at any time either running the same instruction or no instruction at all, <a href="#">86</a></td>
</tr>
</tbody>
</table>
Acknowledgements

I am very grateful to my supervisor, Professor Tor Aamodt, for his guidance during the last two years. Tor introduced me to the rich field of computer architecture and demonstrated first-hand the level of dedication and hard work required to be successful in this field. This thesis would not have been possible without his help and support. I would also like to thank Professor Guy Lemieux and Professor Konrad Walus for being the members of my thesis committee and providing insightful feedback.

I would like to thank my colleagues and friends Ali Bakhoda, Xi Chen, Wilson Fung, Johnny Kuan, Jason Kuo, Ivan Sham, Andrew Turner, and Henry Wong as well as the students in the Networked Systems Laboratory for the interesting discussions we had and for the valuable comments they provided for the work in this thesis. They made my experience at UBC unforgettable and I am very lucky to have been able to meet them.

I am grateful to my parents for their unconditional love and support throughout my life. I would not be where I am today without the sacrifices that they have made. They have done a wonderful job raising me and for that I am forever indebted to them. And above all, I wish to sincerely thank my girlfriend Hyunjung for her unyielding support, bearing with me through the tough times. I dedicate this work to them.

Finally I would like to express my thanks to Natural Sciences and Engineering Research Council (NSERC) of Canada that provided financial support for this work through a CGS-M award.

The work described in this thesis is based upon the following three published bodies of work.
The first is the paper titled “Analyzing CUDA Workloads Using a Detailed GPU Simulator”, which appeared in the 2009 Annual IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009) and was co-authored by Ali Bakhoda, George Yuan, Wilson Fung, Henry Wong, and Dr. Tor Aamodt. The study was designed collaboratively by Ali Bakhoda, George Yuan, Wilson Fung, Henry Wong, and Dr. Tor Aamodt. George Yuan collected the benchmarks used in this study and modified the simulator to provide the necessary support required by the benchmarks. Ali Bakhoda integrated the detailed interconnection network simulator necessary to perform the network topology study in this study. Wilson Fung, Ali Bakhoda, and Dr. Tor Aamodt were also responsible for developing much of the simulator’s baseline infrastructure.

The second is the paper titled “A Hybrid Analytical DRAM Performance Model”, which appeared in the Fifth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2009) held in conjunction with the 36th Annual IEEE/ACM International Symposium on Computer Architecture and was co-authored by George Yuan and Dr. Tor Aamodt. The study was designed collaboratively by George Yuan and Dr. Tor Aamodt. George Yuan conducted the research, analyzed the data, and drafted the manuscript under the guidance of Dr. Tor Aamodt.

The third is the paper titled “Complexity Effective Memory Access Scheduling”, which has been accepted to the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2009) and was co-authored by George Yuan, Ali Bakhoda and Dr. Tor Aamodt. The study was designed collaboratively by George Yuan and Ali Bakhoda and Dr. Tor Aamodt. George Yuan conducted the research, analyzed the data, and drafted the manuscript under the guidance of Dr. Tor Aamodt.
Chapter 1

Introduction

In the past, GPUs, commonly known as video cards, have primarily been used for video games, specifically for accelerating 3D graphics rendering. Since then, GPUs have evolved into massively parallel computing machines that now have theoretical throughputs tens to hundred times faster than any general computer processor (CPU) available in the market. This is due to the graphics processors’ inherently high amounts of compute power specifically designed for data-parallel execution. Furthermore, recent GPUs are programmable through languages that are very similar to traditional languages, such as the “C” programming language, taught in most universities. Researchers have already demonstrated that the compute power of GPUs can be harnessed for general-purpose applications and thereby provide a far more cost-effective solution compared to using CPUs. Yet at the same time, there are obstacles that prevent other applications from running efficiently on GPUs. Intensive control flow, dynamic program behavior, concurrency, and unpredictable memory access patterns all can limit the thread-level parallelism available and impose bottlenecks, thereby reducing performance.

This thesis focuses on the memory subsystem of GPUs running general purpose applications. First, it describes the changes to the memory system modeled by the performance simulator that we use to reflect what occurs in real GPU hardware. Next, it proposes a hybrid analytical model to predict performance of modern DRAM when used in conjunction with modern memory access scheduling policies. Finally, it proposes a complexity effective solution for memory access scheduling.
1.1 Motivation

As the gap between memory speed and microprocessor speed increases, efficient dynamic random access memory (DRAM) system design is becoming increasingly important as it becomes an increasingly limiting bottleneck. This is especially true in General Purpose Applications for Graphics Process Units (GPGPU) where applications do not necessarily make use of caches (which are both limited in size and capabilities relative to the processing throughput of GPUs).

Modern memory systems can no longer be treated as having a fixed long latency. Figure 1.1 shows the measured latency for different applications when run on our performance simulator, GPGPU-Sim, averaged across all memory requests. (We show later in Section 6.1.1 that GPGPU-Sim models contemporary GPUs fairly accurately, achieving 86% correlation to real hard-

---

1 As of NVIDIA’s CUDA Programming Framework version 2, all caches are read-only [39].
1.2 Contributions

As shown, average memory latency can vary greatly across different applications, depending on their memory access behavior. In the massively multithreaded GPU Compute architecture that we simulate, threads are scheduled to Single-Instruction Multiple-Data pipelines in groups of 32 called *warps*, each thread of which is capable of generating a memory request to any location in GPU main memory. (This can be done as long as the limit on the number of in-flight memory requests per GPU shader core has not been reached. In CPUs, this is determined by the number of Miss Status Holding Registers (MSHR) [25].) In this microarchitecture, a warp cannot proceed as long as any single thread in it is still blocked by a memory access, making DRAM an even bigger bottleneck. As such, the memory system of GPU architectures must be properly modeled to provide useful insight for microprocessor and memory designers alike. Two such modeling methods that designers use are analytical modeling and detailed performance simulation. Analytical modeling relies on developing mathematical expressions to relate performance metrics to microarchitecture and workload parameters. Detailed performance simulation requires cycle-by-cycle simulation of the behavior of a proposed hardware design. While performance simulation is more accurate, analytical modeling can be orders of magnitude faster in obtaining meaningful results. In this work, we explore the GPU memory system using both hybrid analytical modeling and detailed performance simulation.

1.2 Contributions

This thesis makes the following contributions:

1. It presents data characterizing the performance of a set of existing CUDA applications collected on a research GPU simulator (GPGPU-Sim).

---

The term “hybrid” was first used for modeling by Shanthikumar and Sargent [50] to describe mathematical models that used both simulation and analytic techniques.
2. It shows that, for certain applications, decreasing the number of threads running concurrently on the hardware can improve performance by reducing contention for on-chip resources.

3. It provides an analysis of application characteristics including the dynamic instruction mix, SIMD warp branch divergence properties, and DRAM locality characteristics.

4. It presents a novel DRAM hybrid analytical model to model the overlapping of DRAM timing constraint delays of one bank by servicing requests to other banks, given an aggressive First-Ready First-Come First-Serve (FR-FCFS) memory scheduling policy. This policy reorders a memory request stream to increase DRAM efficiency by increasing its row access locality, the number of memory requests per DRAM row access.

5. It presents a method to use this hybrid analytical model to predict DRAM efficiency over the runtime of an application by profiling a memory request address trace.

6. It presents two heuristics that, when used with the previously mentioned profiling method, provide bounds on the available bank-level parallelism achievable by the DRAM. Moreover, it presents a simple method of averaging the predictions of these two heuristics to reduce the arithmetic mean of the absolute error across all benchmarks to 11.2%.

7. It shows that GPU architectures do not benefit from previously proposed schedulers that emphasize fairness due to the different challenges that they present, having at least an order of magnitude more threads compared to CPU multiprocessors.

8. It shows that the row access locality inherent in the memory request stream from individual shader cores can be destroyed by the interconnection networks typically used for connecting shader cores to DRAM controllers. We introduce a novel interconnect arbitration scheme that

---

4In this thesis, we use arithmetic mean of the absolute value of error to validate the accuracy of our analytical model, which was argued by Chen et al. [11] to be the correct measure since it always reports the largest error numbers and is thus conservative in not understating said errors.
preserves this inherent row access locality, thus allowing for a much simpler DRAM controller design.

9. It presents a qualitative and quantitative analysis of the performance of our interconnect arbitration scheme and its various heuristics for both mesh and crossbar networks.

10. It also presents a simple solution to deal with the interleaving that can occur due to the interconnection network router having multiple virtual channels, which can improve interconnect throughput [12].

1.3 Background

This section provides an overview of the fundamental concepts of contemporary SIMD GPU architecture, modern DRAM memory, and interconnection networks for this thesis.

1.3.1 SIMD GPU architecture

Graphic applications that run on GPUs have plentiful parallelism due to independence of calculating each pixel’s color. Since there are millions of pixels for high-resolution displays, there is significant parallelism. NVIDIA’s latest GPU architectures exploit this fact by using multiple multi-processor cores, which we refer to as shader cores, each supporting up to 1024 threads. The shader cores operate in Single-Instruction Multiple-Data (SIMD) fashion with eight parallel processors per shader core where the same pipeline stage for each processor in a shader core will always be executing the same instruction on its own separate set of data. By issuing the same instruction to all processors, the overhead for instruction scheduling can be minimized, increasing GPU processing capabilities at the expense of requiring a minimum amount of SIMD thread-level parallelism to fully use it.

With the introduction of Compute Unified Device Architecture (CUDA), NVIDIA allows pro-
1.3. Background

grammers to exploit GPUs to run non-graphic, general-purpose applications as well. CUDA allows programmers to write and run multi-threaded C functions, or kernels, on GPUs. Each kernel is comprised of an organization of threads. These threads are issued to shader cores in groups called cooperative thread arrays (CTAs), where each kernel has one or more CTAs. Within a CTA, threads are grouped into batches of 32 called warps which always execute the same instruction at any time. With an eight-wide shader core, each warp is issued in groups of eight to the pipelines over four back-to-back cycles. The many supported threads per shader core allows for “barrel processing”, where the stalls of one warp due to a long-latency memory load can be hidden by processing other warps in the meantime.

1.3.2 Modern DRAM memory

Before explaining our hybrid analytical DRAM performance model, it is necessary to be familiar with the background of modern DRAM and the mechanics of the memory that we are modeling, GDDR3 [47], including the different timing constraints and various other terminology. A summary is provided in Table 1.1.

GDDR3-SDRAM is a modern graphics-specific DRAM architecture that improves upon older DRAM technology used in graphics processors, like DDR-SDRAM [38], by increasing clock speeds and lowering power requirements. Like DDR-SDRAM, it uses a double data rate (DDR) interface, meaning that it transfers data across its pins on both the positive and negative edge of the bus clock. More specific to the particular GDDR3 technology that we study, a 4n-prefetch architecture is used, meaning that a single read or write access consists of 4 bits transferred across each data pin over 2 cycles, or 4 half-cycles. (This is more commonly known as the burst length, BL.) For a 32-bit wide interface, this translates to a 16 byte transfer per access for a single GDDR3 chip.

Similar to other modern DRAM technology, GDDR3-SDRAM is composed of a number of memory banks sharing a common data bus in order to exploit bank-level parallelism in the memory
1.3. Background

Table 1.1: DRAM parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Banks</td>
<td>Number of Banks</td>
<td>4</td>
</tr>
<tr>
<td>Bus Width (in Bytes)</td>
<td>Bus Width (in Bytes)</td>
<td>4</td>
</tr>
<tr>
<td>BL</td>
<td>Burst Length (in Bytes)</td>
<td>16</td>
</tr>
<tr>
<td>$t_{CCD}$</td>
<td>Delay between successive reads or successive writes (in cycles)</td>
<td>2</td>
</tr>
<tr>
<td>$t_{RRD}$</td>
<td>Minimum time between successive ACT commands to different banks (in cycles)</td>
<td>8</td>
</tr>
<tr>
<td>$t_{RAS}$</td>
<td>Minimum time between opening (ACT) and closing (PRE) a row (in cycles)</td>
<td>21</td>
</tr>
<tr>
<td>$t_{RCD}$</td>
<td>Minimum time between issuing ACT and issuing a Read or Write (in cycles)</td>
<td>12</td>
</tr>
<tr>
<td>$t_{RC}$</td>
<td>Minimum time between successive ACTS to different rows in same bank (in cycles)</td>
<td>34</td>
</tr>
<tr>
<td>$t_{WTR}$</td>
<td>Write to Read command delay (in cycles)</td>
<td>5</td>
</tr>
<tr>
<td>$t_{RP}$</td>
<td>Minimum time between closing a row and opening a new row (in cycles)</td>
<td>13</td>
</tr>
<tr>
<td>$CL$</td>
<td>Column Address Strobe Latency (in cycles)</td>
<td>9</td>
</tr>
</tbody>
</table>

request stream. We define bank-level parallelism as the spreading of memory requests across multiple DRAM banks so that a ready DRAM bank can immediately transfer data across the data bus when another bank is unready. A DRAM bank may be unready if it is in the process of switching rows or refreshing its data cells. The bank-level parallelism of a memory request stream is crucial to performance because it maximizes usage of the data bus pins connecting the GPU to the DRAM chips, a limited and thus precious resource.

Each bank is composed of a 2D array that is addressed with a row address and a column address, both of which share the same address pins to reduce the pin count. In a typical memory access, a row address is first provided to a bank using an activate command that activates, or “opens”, all of the memory cells in the row, connecting them using long wires to the sense amplifiers, which are subsequently connected to the data pins. A key property of the sense amplifiers is that, once they detect the values of the memory cells in a row, they hold on to the values so that subsequent accesses to the row do not force the sense amplifiers to re-read the row. Intelligent memory controllers exploit this property, scheduling requests out of order and grouping together requests to the same
row to improve the row access locality of the memory requests. Doing so consequently improves
the efficiency of the memory system by reducing the number of times rows are opened and closed
4. The organization of such a memory system is shown in Figure 1.2.

Typically, the issuing of commands to DRAM is subject to various timing constraints, as out-
lined in Table ???. For clarification, these timing constraints are illustrated in Figure 1.3.

The process of “opening the row,” which must be done before a column can be requested, takes
a significant amount of time called the row address to column address delay, or $t_{RCD}$ [47]. After

---

4“Row access locality” is temporal in the sense that concurrent accesses to the same row are fast. It is spatial in
the sense that accessing any word in DRAM requires activating the whole row, allowing subsequent accesses to other
data in the row to be fast.
1.3. Background

Figure 1.3: Illustration of timing constraints

this delay, the column address can then be provided to the bank. The process of reading the column address and choosing the data in the specific column to be output across the data pins is not instantaneous, instead taking an amount of time called the column access strobe latency, or \( CL \), from when the column address is provided to when the first bits of data appear on the data pins. However, since all of the memory cells in the opened row are connected to sense amplifiers, multiple column addresses can be provided back-to-back, limited only by the column to column delay, \( t_{CCD} \), and the operations will be pipelined. Once the initial \( CL \) cost has been paid, a bank can provide a continuous stream of data as long as all accesses are to the opened row.

If we want to access data in one row and another row is opened in the same bank, we must close the opened row by disconnecting it from the long wires connecting it to the sense amplifiers. Before these long wires, which represent a significant capacitance, can be connected to the new row, they must first be precharged in order for the sense amplifiers to work properly. This process is defined as the row precharge delay, or \( t_{RP} \) [13]. To summarize, in order to service a request to a new row when a different row has already been opened, we must first close the row (by issuing a precharge
1.3. Background

command) and open the new row (by issuing an activate). This process takes an amount of time \( t_{RP} + t_{RCD} \).

Furthermore, in a single bank, there is a minimum time required between issuing an activate command and then issuing a precharge (in other words, between opening and closing the row). This minimum time is defined as the row access strobe latency, \( t_{RAS} \), and is greater than \( t_{RCD} \). This is because \( t_{RAS} \) also encapsulates the process of restoring the data from the sense amplifiers to the corresponding row cells, or “refreshing.” (Unlike static RAM, which retains the values in its memory cells as long as it is provided power, dynamic RAM must be periodically refreshed since the bit values are stored in capacitors which leak charge over time [13].) Opening a new row too quickly may not allow adequate time to perform the refresh [53]. Accordingly, the minimum time required between issuing two successive activate commands in a single bank is \( t_{RAS} + t_{RP} \), which is defined as the row cycle time, \( t_{RC} \).

In addition to these timing constraints, there are additional constraints on the minimum time needed between successive activate commands to different banks, \( t_{RRD} \), and the minimum time needed between a write and a read command or a read and a write command (\( t_{WTR} \) and \( t_{RTW} \)) since the data bus is bi-directional and requires time to switch modes.

In the specific GDDR3 technology that we study, each row must be refreshed every 32ms, taking 8192 DRAM cycles each time. With an 800MHz DRAM clock and 4096 rows per bank, this amounts to each bank having to perform refresh 4.2% of the time on average, assuming DRAM is idle and all rows contain data [47]. As mentioned previously, the process of accessing a row also refreshes the data in the row, meaning only rows of unaccessed data needs to be actively refreshed, possibly reducing the refreshing overhead from 4.2%. Due to this relatively small overhead, we model the effects of refresh on performance in neither our performance simulator nor our analytical model.
1.3. Background

![Router Architecture](image1.png)

(a) Router architecture

![Router Pipeline Stages](image2.png)

(b) Router pipeline stages

Figure 1.4: Typical interconnect router architecture

### 1.3.3 Interconnection networks

Out-of-order memory scheduling requires complex fully-associative comparison logic. In this thesis, we also explore simplifying the memory controller design by incorporating intelligence at the level above the memory controller: the interconnection network. In a multiprocessor architecture, interconnection networks are typically used to transfer memory writes and memory read requests from issuing processors to receiving memory controllers, returning memory reads from memory controllers back to the processors, as well as processor-to-processor communication. The network can be comprised of either a single crossbar router that connects all inputs to all outputs or several routers connected in a variety of configurations.

In this thesis, we evaluate the effect of crossbar, mesh, and ring networks on DRAM. No matter the configuration, all routers are required to perform the common task of sending packets (requests) from a set of input ports to a set of output ports. This task is called allocation. Since the network
1.4 Organization

has finite bandwidth, packets that cannot be transferred in one cycle are split up into flits and sent over multiple cycles. Figure ?? shows a typical router architecture along with the steps for performing pipelined routing of a packet [12]. Routing computation, the process of determining which output the packet should be directed to, and virtual channel allocation, the process of reserving buffer space at the output port, is performed on a per-packet level. Switch allocation, the process of reserving a time slot for transferring a flit, and switch traversal, the actual transfer of the flit across the crossbar switch, is performed on a per-flit level.

There are many different algorithms for performing allocation. In this thesis, we use parallel iterative matching (PIM), a simple algorithm easily realizable in hardware [4], which is important in high-speed interconnects. In PIM allocation, a request matrix of input rows by output columns is first generated to determine which inputs require which outputs. In the context of a crossbar, the number of inputs is the number of shader cores $C$ and the number of outputs is the number of DRAM controllers $M$. PIM is a type of separable allocator, where arbitration is done in two stages, which is advantageous where speed is important [12]. In our study, we perform input-first allocation. Figure 1.5 shows a simple example of an input-first allocator with three inputs and two outputs. First, the input arbiters select from a single request per input port. In other words, each input arbiter chooses to assert up to one output, depending on whether there are requests from the corresponding input to the corresponding output. The output of this first stage of input arbiters is then fed to the output arbiters, which choose from the different requests to each output. This guarantees that a single input port will not be sending to multiple output ports and a single output port will not be receiving from multiple input ports in the same cycle.

1.4 Organization

The rest of the thesis is organized as follows:
Chapter 2 describes the necessary changes to the memory system of GPGPU-Sim to support applications written using NVIDIA's CUDA framework. It then introduces the CUDA applications that were collected to perform the experiments done in this work.

Chapter 3 proposes techniques to predict the efficiency of a DRAM system when employing a hybrid analytical model that applies a memory request address trace analysis.

Chapter 4 presents two novel metrics to quantify the temporal locality of an application and proposes a novel cache contention model, based upon the above two metrics, to accurately predict the number of extra data cache misses due to cache contention when the number of threads sharing a cache approaches and/or exceeds the cache’s associativity.

Chapter 5 describes our simulation methodology, highlighting the applications used in our study and how the memory system in our simulation infrastructure was augmented to support...
1.4. Organization

applications coded using NVIDIA’s CUDA programming framework.

- Chapter 6 presents and analyzes our experimental results.

- Chapter 7 reviews related work.

- Chapter 8 concludes this thesis and suggests future work.

Although GPUs can achieve throughput (in terms of FLOPs) orders of magnitude higher than CPUs, they require applications with, among other things, high arithmetic intensity to do so. General purpose applications with irregular memory access behavior perform poorly. This work is a step towards research on how GPUs can be used for a broader class of applications.
Chapter 2

Supporting CUDA Applications on GPGPU-Sim

The version of GPGPU-Sim used by Wilson et al. [16] used the portable instruction set architecture (PISA). For benchmarks, applications from various benchmark suites including NVIDIA’s CUDA Software Development Kit [37], SPLASH2 [55], and SPEC CPU2006 [52] were manually modified from CUDA format to a format recognizable by the simulator. Due to the engineering effort required in this process, it was decided for the work of Bakhoda et al. [5] that a massively multithreaded performance simulator that could support native CUDA applications without any source code modifications would be the better approach. To this end, GPGPU-Sim was modified to support CUDA’s parallel thread execution (PTX) instruction set, allowing it to simulate CUDA applications without requiring any changes to the original CUDA source code. We first describe the microarchitectural changes made to the performance simulator to properly model CUDA. We then describe the CUDA applications that were collected and used as benchmarks.

2.1 CUDA memories

The CUDA programming framework supports five distinct logical memory spaces, each with its own unique properties. These are the global memory, local memory, shared memory, texture memory, and constant memory. The set of CUDA applications that we wished to support used all of these memory spaces.

Figure 2.1 shows the logical layout of the shader core pipeline that we simulate with the memory
2.1. CUDA memories

Instructions that pass through the memory stage of the pipeline are checked first whether they are memory instructions. If so, then they get routed to the appropriate substage within the memory stage. This thesis describes the simulator modifications necessary to support local memory, texture memory, and constant memory.

Figure 2.1: Simulated pipeline architecture
2.1. CUDA memories

2.1.1 Local memory

The use of local memory accesses is determined automatically by the compiler. They are commonly used for objects large enough that the compiler deems them unsuitable for storage in registers, such as large structures or arrays. Within GPGPU-Sim, we model local memory accesses in accordance to the CUDA programming guide \[39\] in that they are not cached, like global memory accesses, and are thus expensive in terms of the latency that they incur. However, since local memory is defined on a per-thread basis, we can make sure that their accesses are always coalesced by storing each local memory variable of threads in a half-warp in a contiguous block in memory. Consider an example where each thread declares a struct with four int variables in local memory. Instead of storing the variables of the struct of each thread in a contiguous space, we instead interleave the variables of each thread to ensure memory coalescing.

2.1.2 Texture memory

Texture memory accesses are backed by an L1 read-only texture cache. The use of texture memory must be specified explicitly by the programmer by calling functions included in the CUDA runtime API library. In our simulator, we leverage the CUDA runtime API library header files while replacing their library function implementations with our own which invoke simulation setup functions (such as memory allocation/deallocation and memory copy) and simulation sessions. Proper setup of textures for use in GPGPU-Sim requires the following steps: first, the memory needed by the texture must be allocated within the simulated memory space (e.g. GPU main memory). Second, the memory must be “bound” to a texture reference, which is an object declared by the programmer. Apart from having a pointer to the allocated memory, the reference also stores the texture name as well as other texture-specific properties. This is because texture memory load instructions differ from load instructions of other memory spaces in PTX in that it uses a unique addressing mode that relies on the name of the texture and the co-ordinates within the texture. Finally, the
data must then be copied from CPU memory to the simulated memory space.

CUDA allows for texture objects to be used in a wide variety of ways depending on how the programmer configured the texture reference, which, in GPGPU-Sim, is a struct containing the properties in variables that are set using functions from the CUDA runtime API. We added support for the following capabilities:

- Different data types: Texture data type can vary from single byte chars to four byte ints and floats to even doubles. Moreover, they can be either scalar or up to a vector of four values. As such, texture load implementation must properly account for the many different possible fetch sizes, from a single char (one byte) to a vector of four floats (sixteen bytes). During simulation, a texture fetch instruction provides the texture name, which we use to look up the texture reference that holds the pointer to the texture (e.g. address base) in the simulated memory space. The texture fetch instruction also provides the texture co-ordinates which we use with the data size to calculate the address offset which, when summed with the address base, provides us the address of the requested data in the simulated memory space.

- Multi-dimensionality: Textures can be either one-dimensional (indexed with one address value) or two-dimensional (indexed with two address values). In CUDA, texture cache blocks are fetched from GPU main memory in a way that preserves two dimensional spatial locality for two-dimensional textures. In other words, a texture fetch to a specific co-ordinate will also retrieve data close to the co-ordinate in both dimensions to the same cache block. In GPGPU-Sim, we implement a 4D blocking address scheme which essentially permutes the bits in requested addresses to promote spatial locality in a 2D space rather than in linear space.

- Different addressing modes: The provided index (or indices) into the texture can be either the absolute co-ordinates in integers or also normalized floats from 0.0 to 1.0 (useful for image
2.1. CUDA memories

processing). If normalized address mode is used, CUDA supports a variety of sampling modes if the normalized address does not scale to integer co-ordinates. For instance, the data of the nearest co-ordinate to the normalized address may be fetched or the values around the normalized address can be sampled and averaged (linear filtering). We find that none of the benchmarks we study use normalized addressing so we only implemented pick nearest. We leave the implementation of other sampling modes needed for imaging processing and other graphical applications to future work.

• Boundary condition handling: In accordance with CUDA, we added two methods to GPGPU-Sim for handling indices that are outside the range of the texture array: “Clamping” clamps index values below 0.0 and above 1.0 to the range [0.0,1.0) (for normalized addressing mode). “Wrapping” uses only the fractional part of the texture co-ordinate so that, for example, an index of 1.75 and -0.25 will both wrap to a texture co-ordinate of 0.75 (again for normalized addressing mode).

2.1.3 Constant memory

The purpose of constant memory is to store read-only “constant” variables shared among all threads. As such, constant memory accesses are backed by an L1 read-only constant cache. Unlike the texture memory cache, the constant cache allows single cycle access latency only if all threads in a half-warp request the same data. In other words, the texture cache can be thought of as having one read port for each shader processor in a shader core while the constant cache has only a single read port shared among all shader processors in a shader core. If there is port conflict (e.g. when shader processors require access to multiple cache blocks in the same cycle), the accesses are serialized and the pipeline stalls while the constant cache is being accessed. Since a warp cannot proceed as long as any thread is stalled waiting on memory, the order in which the constant cache services requests does not matter. In the benchmarks we study, we find that none of the constant memory accesses
2.2. CUDA benchmarks

Table 2.1: Benchmark thread organization

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Abr.</th>
<th>Grid Dimensions</th>
<th>CTA Dimensions</th>
<th>Concurrent CTAs/core</th>
<th>Total Threads</th>
<th>Instruction Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES Cryptography [27]</td>
<td>AES</td>
<td>(257,1,1)</td>
<td>(256,1,1)</td>
<td>2</td>
<td>65792</td>
<td>25M</td>
</tr>
<tr>
<td>Graph Algorithm:</td>
<td>BFS</td>
<td>(128,1,1)</td>
<td>(512,1,1)</td>
<td>4</td>
<td>65536</td>
<td>17M</td>
</tr>
<tr>
<td>Breadth First Search [20]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coulombic Potential [23][45]</td>
<td>CP</td>
<td>(8,32,1)</td>
<td>(16,8,1)</td>
<td>8</td>
<td>32768</td>
<td>126M</td>
</tr>
<tr>
<td>gpuDG [54]</td>
<td>DG</td>
<td>(268,1,1);</td>
<td>(84,1,1);</td>
<td>5</td>
<td>22512;</td>
<td>596M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(268,1,1);</td>
<td>(112,1,1);</td>
<td></td>
<td>30016;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(603,1,1)</td>
<td>(256,1,1)</td>
<td>4</td>
<td>154368</td>
<td></td>
</tr>
<tr>
<td>3D Laplace Solver [17]</td>
<td>LPS</td>
<td>(4,25,1)</td>
<td>(32,4,1)</td>
<td>6</td>
<td>12800</td>
<td>82M</td>
</tr>
<tr>
<td>LIBOR Monte Carlo [18]</td>
<td>LIB</td>
<td>(64,1,1)</td>
<td>(64,1,1)</td>
<td>8</td>
<td>4096</td>
<td>907M</td>
</tr>
<tr>
<td>Matrix Multiply [46]</td>
<td>MMC</td>
<td>(8,32,1)</td>
<td>(16,16,1)</td>
<td>3</td>
<td>65536</td>
<td>341M</td>
</tr>
<tr>
<td>MUMmerGPU [48]</td>
<td>MUM</td>
<td>(782,1,1)</td>
<td>(64,1,1)</td>
<td>3</td>
<td>50000</td>
<td>773M</td>
</tr>
<tr>
<td>Nearest Neighbor [9]</td>
<td>NN</td>
<td>(938,1,1)</td>
<td>(16,1,1)</td>
<td>8</td>
<td>15008</td>
<td>6.4M</td>
</tr>
<tr>
<td>Neural Network</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Digit Recognition [7]</td>
<td>NEU</td>
<td>(6,28,1);</td>
<td>(13,13,1);</td>
<td>5</td>
<td>28392;</td>
<td>40M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(50,28,1)</td>
<td>(5,5,1)</td>
<td>8</td>
<td>35000</td>
<td></td>
</tr>
<tr>
<td>N-Queens Solver [41]</td>
<td>NQ</td>
<td>(223,1,1)</td>
<td>(96,1,1)</td>
<td>1</td>
<td>21408</td>
<td>2M</td>
</tr>
<tr>
<td>Ray Tracing [28]</td>
<td>RAY</td>
<td>(16,32,1)</td>
<td>(16,8,1)</td>
<td>3</td>
<td>65536</td>
<td>71M</td>
</tr>
<tr>
<td>StoreGPU [3]</td>
<td>STO</td>
<td>(384,1,1)</td>
<td>(128,1,1)</td>
<td>1</td>
<td>49152</td>
<td>134M</td>
</tr>
<tr>
<td>Weather Prediction [29]</td>
<td>WP</td>
<td>(9,8,1)</td>
<td>(8,8,1)</td>
<td>3</td>
<td>4608</td>
<td>215M</td>
</tr>
<tr>
<td>Black-Scholes option pricing [37]</td>
<td>BLK</td>
<td>(256,1,1)</td>
<td>(256,1,1)</td>
<td>3</td>
<td>65536</td>
<td>236M</td>
</tr>
<tr>
<td>Fast Walsh Transform [37]</td>
<td>FWT</td>
<td>(512,1,1);</td>
<td>(256,1,1);</td>
<td>4</td>
<td>131072;</td>
<td>240M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(512,1,1)</td>
<td>(512,1,1)</td>
<td>2</td>
<td>131072</td>
<td></td>
</tr>
</tbody>
</table>

To evaluate GPGPU-Sim, we needed benchmarks that exhibited a wide variety of behaviors in terms of arithmetic intensity, warp divergence, and memory access patterns. Preliminary evaluation of the applications included in NVIDIA’s CUDA software development kit (SDK) showed that they tend to conform well to the CUDA Programming Framework, exhibiting relatively high IPC normalized to peak, little to no warp divergence, and moderate to low bandwidth utilization. To find other applications with other behaviors, we collected applications from other research communities and NVIDIA’s CUDA Zone that reported relatively low speedup versus running on CPUs. Below, we describe the CUDA applications not in the SDK that we use as benchmarks in our study. These applications were developed by the researchers cited below and run unmodified on our simulator.

request more than one address per warp.

2.2 CUDA benchmarks
2.2. CUDA benchmarks

Table 2.2: Benchmark Properties

<table>
<thead>
<tr>
<th>Abr.</th>
<th>Local Memory?</th>
<th>Shared Memory?</th>
<th>Constant Memory?</th>
<th>Texture Memory?</th>
<th>Barriers?</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>BFS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CP</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DG</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>LFS</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LIB</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MMC</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>MUM</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>2D</td>
<td>-</td>
</tr>
<tr>
<td>NN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NEU</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NQ</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>RAY</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>STO</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WP</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BLK</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FWT</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
</tr>
</tbody>
</table>

Tables 2.1 and 2.2 lists the benchmarks name along with the main application properties, such as the organization of threads into CTAs and grids as well as the different memory spaces on the GPU exploited by each application. Note that the instruction counts listed in Table 2.1 are the dynamic instruction counts run on the GPU kernels summed across all threads. These kernels are essentially function calls that have been parallelized to run on the GPU and, as such, some of the instruction counts may seem small relative to typical instruction counts of entire programs. Multiple entries separated by semi-colons in the grid and CTA dimensions indicate the application runs multiple kernels.

**AES Encryption (AES)** [27] This application, developed by Manavski [27], implements the Advanced Encryption Standard (AES) algorithm in CUDA to encrypt and decrypt files. The application has been optimized by the developer so that constants are stored in constant memory, the expanded key stored in texture memory, and the input data processed in shared memory. We encrypt a 256KB picture using 128-bit encryption.

**Graph Algorithm: Breadth-First Search (BFS)** [20] Developed by Harish and Narayanan [20], this application performs breadth-first search on a graph. As each node in the
2.2. **CUDA benchmarks**

graph is mapped to a different thread, the amount of parallelism in this applications scales with the size of the input graph. BFS suffers from performance loss due to heavy global memory traffic and branch divergence. We perform breadth-first search on a random graph with 65,536 nodes and an average of 6 edges per node.

**Coulombic Potential (CP)** [23, 45] CP is part of the Parboil Benchmark suite developed by the IMPACT research group at UIUC [23, 45]. CP is useful in the field of molecular dynamics. Loops are manually unrolled to reduce loop overheads and the point charge data is stored in constant memory to take advantage of caching. CP has been heavily optimized (it has been shown to achieve a $647\times$ speedup versus a CPU version [46]). We simulate 200 atoms on a grid size of $256\times256$.

**gpuDG (DG)** [54] gpuDG is a discontinuous Galerkin time-domain solver, used in the field of electromagnetics to calculate radar scattering from 3D objects and analyze wave guides, particle accelerators, and EM compatibility [54]. Data is loaded into shared memory from texture memory. The inner loop consists mainly of matrix-vector products. We use the 3D version with polynomial order of $N=6$ and reduce time steps to 2 to reduce simulation time.

**3D Laplace Solver (LPS)** [17] Laplace is a highly parallel finance application [17]. As well as using shared memory, care was taken by the application developer to ensure coalesced global memory accesses. We observe that this benchmark suffers some performance loss due to branch divergence. We run one iteration on a 100x100x100 grid.

**LIBOR Monte Carlo (LIB)** [18] LIBOR performs Monte Carlo simulations based on the London Interbank Offered Rate Market Model [18]. Each thread reads a large number of variables stored in constant memory. We find the working set for constant memory fits inside the 8KB constant cache per shader core that we model. However, we find memory bandwidth is still a bottleneck due to a large fraction of local memory accesses. We use the default inputs, simulating 4096 paths for 15 options.
2.2. CUDA benchmarks

**Matrix Multiply (MMC) [46]** This version of matrix multiply improves upon the optimized version of matrix multiply included in the CUDA SDK. In addition to using blocking to minimize the number of memory accesses, this version also exploits loop unrolling to reduce loop overheads. Furthermore, each CTA is carefully sized to maximize utilization of the 16kB shared memory per shader core.

**MUMmerGPU (MUM) [48]** MUMmerGPU is a parallel pairwise local sequence alignment program that matches query strings consisting of standard DNA nucleotides (A,C,T,G) to a reference string for purposes such as genotyping, genome resequencing, and metagenomics [48]. The reference string is stored as a suffix tree in texture memory and has been arranged to exploit the texture cache’s optimization for 2D locality. Nevertheless, the sheer size of the tree means high cache miss rates, causing MUM to be memory bandwidth-bound. Since each thread performs its own query, the nature of the search algorithm makes performance also susceptible to branch divergence. We use the first 140,000 characters of the Bacillus anthracis str. Ames genome as the reference string and 50,000 25-character queries generated randomly using the complete genome as the seed.

**Nearest Neighbor (NN) [9]** Nearest neighbor is a data mining application that uses dense linear algebra to find the k-nearest neighbors from an unstructured data set. The application is bandwidth bound and has a very regular memory access pattern, thus achieving close to the peak off-chip bandwidth. We find the 3 nearest neighbors from a set of 42764 records.

**Neural Network (NEU) [7]** Neural network uses a convolutional neural network to recognize handwritten digits [7]. Pre-determined neuron weights are loaded into global memory along with the input digits. We modified the original source code to allow recognition of multiple digits at once to increase parallelism. Nevertheless, the last two kernels utilize CTAs of only a single thread each, which results in severe under-utilization of the shader core pipelines. For this reason, we only run the first two kernels in our simulations in Chapter 6. We simulate recognition of 28 digits from
the Modified National Institute of Standards Technology database of handwritten digits.

**N-Queens Solver (NQ) [41]** The N-Queen solver tackles a classic puzzle of placing N queens on a NxN chess board such that no queen can capture another [41]. It uses a simple backtracking algorithm to try to determine all possible solutions. The search space implies that the execution time grows exponentially with N. Our analysis shows that most of the computation is performed by a single thread, which explains the low IPC. We simulate N=10.

**Ray Tracing (RAY) [28]** Ray-tracing is a method of rendering graphics with near photo-realism. In this implementation, each pixel rendered corresponds to a scalar thread in CUDA [28]. Up to 5 levels of reflections and shadows are taken into account, so thread behavior depends on what object the ray hits (if it hits any at all), making the kernel susceptible to branch divergence. We simulate rendering of a 256x256 image.

**StoreGPU (STO) [3]** StoreGPU is a library that accelerates hashing-based primitives designed for middleware [3]. We chose to use the sliding-window implementation of the MD5 algorithm on an input file of size 192KB. The developers minimize off-chip memory traffic by using the fast shared memory. We find STO performs relatively well.

**Weather Prediction (WP) [29]** Numerical weather prediction uses the GPU to accelerate a portion of the Weather Research and Forecast model (WRF), which can model and predict condensation, fallout of various precipitation and related thermodynamic effects of latent heat release [29]. The kernel has been optimized to reduce redundant memory transfer by storing the temporary results for each altitude level of a cell in registers. However, this requires a large amount of registers, thus limiting the maximum allowed number of threads per shader core to 192, which is not enough to cover global and local memory access latencies. We simulate the kernel using the default test sample for 10 timesteps.
Chapter 3

Hybrid Analytical Performance Modeling of DRAM

In this chapter, we describe our analytical model of GDDR3 memory. First, we present a more formal definition of DRAM efficiency in Section 3.1. Then, we present our baseline analytical model for predicting DRAM efficiency in Section 3.2. Finally, we show how to implement our model using a sliding window profiling technique on a memory request address trace in Section 3.3.

3.1 DRAM efficiency

We first define *DRAM utilization* as the percentage of time that a DRAM chip is transferring data across its bus over an arbitrary length of time, $T$. We also define *DRAM efficiency* as the percentage of time that the DRAM chip is transferring data across its bus over only the period of time when the DRAM is active ($T_{active}$). We say that a DRAM chip is active when it is either actively transferring data across its bus or when there are memory requests waiting in the memory controller queue for this DRAM and it is waiting to service the requests but cannot due to any of the timing constraints mentioned in Section 1.3.2. If the DRAM is always active, the DRAM utilization and DRAM efficiency metric will evaluate to the same value for the same period of time (e.g. $T = T_{active}$). While 100% DRAM efficiency and utilization is theoretically achievable, less-than-maximum throughput can occur for a variety of different reasons: there may be poor row access locality among the requests waiting in the memory controller queue, resulting in DRAM having to pay the overhead cost of constantly closing and reopening different rows too frequently,
and therefore the few requests per open row are not enough to hide the aforementioned overheads; the memory controller may also be causing the DRAM to frequently alternate between servicing reads and writes, thus having to pay the overhead cost of $t_{WTR}$ and $t_{RTW}$ each time; and, specific only to DRAM utilization, the memory controller queue may simply be empty, forcing the DRAM to sit idle.

### 3.2 Hybrid analytical model

As described in the previous section, both our DRAM utilization and DRAM efficiency metrics are essentially fractions. We use these definitions as the basis of our analytical model, replacing the numerators and denominators with an expression of variables that can be derived by analyzing a collected memory request address trace.

We develop our analytical model by first considering requests to a single bank $j$ for a time period $T$, which starts from when a request to bank $j$ must open a new row and ends when the last request to the new row in bank $j$ present in the memory controller queue at the start of the period has been serviced. We use $i$ to denote the requests to any bank of $B$ banks, not just those to bank $j$. The DRAM efficiency of bank $j$ is first expressed as:

$$ Eff_j = \frac{MIN\left[\sum_{i=0}^{B-1} t_i \right]}{MAX\left(t_{RC}, t_{RP} + t_{RCD} + t_j\right)} $$

(3.1)

where

$$ t_i = \text{number of requests to bank } i \times T_{src\_req} $$

(3.2)

and

$$ T_{src\_req} = \frac{\text{request size}}{\text{DRAMs\_per\_MC} \times \text{busW} \times \text{data\_rate}} $$

(3.3)

In essence, the numerator of Equation 3.1 represents the time spent transferring data divided by the period of time represented in the denominator. To aid our description, we provide the example
3.2. Hybrid analytical model

Figure 3.1: Timing diagram example. Here, Bank 0 has a different row opened so precharge and activate must first be issued before read commands can be issued. Banks 1 through 3 have already been initialized to the correct rows.

shown in Figure 3.1 which will be referred to as we elaborate our model.

The principle underlying Equation 3.1 is that a bank must first wait at least $t_{RC}$ before switching to a new row. This overhead can be hidden if there are requests to other banks that are to opened rows. In Figure 3.1, this is the case. In this example, $j$ is 0 and the requests to banks 1, 2, and 3 help hide the precharge and activate delays.

The numerator in Equation 3.1 is the sum of all $t_i$, where $t_i$ is defined as the product of two terms: the number of requests that can be serviced by bank $i$ over the set period of time defined in the denominator multiplied by the time it takes to service each request ($T_{srvc,req}$). Note that the sum of all $t_i$ also includes $t_j$. We assume that all memory requests are for the same amount of data. The formula for calculating how much a single request to bank $i$ contributes to $t_i$ is shown in
3.2. Hybrid analytical model

Equation 3.3. Here, request_size is the size of the memory request in bytes, DRAMs_per_MC is the number of DRAM chips controlled by each memory controller (e.g., having two DRAM chips connected in parallel doubles the effective bandwidth per memory controller, meaning each single chip only needs to transfer half the data per request), busW is the bus width in bytes, and data_rate is the number of transfers per cycle, 2 for GDDR3’s double data rate interface [47]. Moreover, the minimum granularity of a memory access, ReqGranularity which determines how many read or write commands need to be issued per request, is defined in Equation 3.4. For a fixed memory request size, the minimum granularity determines the maximum number of parallel DRAM chips that can be used efficiently by a single memory controller. (With our simulation parameters, the maximum number of parallel DRAM chips is four.)

\[
\text{ReqGranularity} = \text{DRAMs}_{\text{per}_{-}\text{MC}} \times \text{busW} \times BL
\]  

Equation 3.4 depends on a new parameter unused in the previous equations called the burst length, or BL. Our example corresponds to the baseline configuration described in Section 5.2 where each request is comprised of 2 read commands to a DRAM chip (2 read commands * 2 DRAM chips per memory controller * 4B bus width * burst length 4 = 64B request size in our baseline configuration).

Basing our model on our defined metrics, we set the denominator of Equation 3.1 as the period of time starting from when a particular bank of a DRAM chip begins servicing a request to a new row (by first precharging, assuming a different row is opened, and then issuing activate to open the new row) and ending at when the bank issues a precharge to begin servicing a new request to a different row. To simplify our definition of T in this case, our model assumes that a precharge command is not issued until its completion can be immediately followed by an activate to the same bank. Under this assumption, the delay between successive precharge commands is then constrained by the same delay as the delay between successive activate commands, t_{RC}. This is shown as T in Figure 3.1. The denominator is controlled by two different sets of factors, depending
3.2. Hybrid analytical model

on how many requests need to be serviced in this new row. This is because switching to a new row is primarily constrained by the three timing parameters described in Section 1.3.2 \( t_{RC} \) (row cycle time), \( t_{RP} \) (row precharge delay), and \( t_{RCD} \) (row address to column address delay). In the GDDR3 specification that we use, \( t_{RC} \) is greater than the sum of \( t_{RP} \) and \( t_{RCD} \), so the denominator must always be greater than or equal to \( t_{RC} \). This is expressed in our model using the MAX() function.

In our example, since there is only one request (2 read commands) to bank 0, the denominator is dominated by \( t_{RC} \) so \( T \) equals to \( t_{RC} \).

Given our microarchitectural parameters outlined in Table 5.3 and Table 1.1 our read and write requests take four “DRAM clock” cycles to complete, where each request is comprised of two commands. To obtain number of requests to bank \( i \), we count the number of requests waiting in the memory controller queue that are to the row currently open for corresponding bank \( i \). This is equivalent to doing so when the memory controller first chooses the request to the new row of bank \( j \) (in other words, right before it needs to precharge and activate). Requests to unopened rows are not counted because they must wait at least \( t_{RC} \) before they can begin to be serviced, by which time they can no longer be used to hide the timing constraint overhead for bank \( j \). Since requests to the new row in bank \( j \) itself can not be used to hide the latency of precharging and activating the new row, it appears in both the denominator and in the numerator as one of the terms of \( \sum_i t_i \) (when \( i = j \)). In our example, there are 2 read commands to bank 0 and 10 read commands issued to banks 1 through 3 so here \( t_j=2*2=4 \) and \( \sum_i t_i = 2*(2+10) = 24 \). Therefore, for the time period \( T \), \( Eff_0 = 24/34 = 70.6\% \).

As previously described, \( t_i \) is used to determine how much of the row switching overhead can be hidden by row-hit requests to other banks. With good DRAM row access locality, there may be more such requests than necessary, causing \( \sum_i t_i \) to be greater than the denominator in Equation 3.1. Using the MIN() function to find the minimum between \( \sum_i t_i \) and the expression identical to the denominator is simply done to impose a ceiling on our predicted result at unity, which we argue
3.3. Processing address traces

is accurate since \( \sum_i t_i > \text{MAX}(t_{RC}, t_{RP} + t_{RCD} + t_j) \) means that there are more than enough requests to completely hide the timing constraint delays imposed by opening the new row in bank \( j \).

As stated before, this equation assumes \( T = T_{\text{active}} \). In other words, it does not take into account the amount of time when the DRAM is inactive; therefore, it is more suitable for predicting DRAM efficiency rather than DRAM utilization. We expect that available memory request address traces will not necessarily provide any timing information on when DRAM is inactive, in which case we are limited to only being able to predict the DRAM efficiency. In order to find the DRAM efficiency over the entire runtime length of a program, we sum the numerators obtained using Equation 3.1 of the time periods that make up the runtime length of the program and divide by the sum of the corresponding denominators.

\[
Ef_f_j = \frac{\sum_{n=1}^{N} \text{MIN} \left[ \text{MAX}(t_{RC}, t_{RP} + t_{RCD} + t_{j,n}), \sum_{i=0}^{B-1} t_{i,n} \right]}{\sum_{n=1}^{N} \text{MAX}(t_{RC}, t_{RP} + t_{RCD} + t_{j,n})}
\]  

(3.5)

Equation 3.5 shows the DRAM efficiency calculation of a program whose entire runtime length is comprised of \( N \) periods, where the DRAM efficiency of each period can be calculated using Equation 3.1. How we determine which time periods to use is explained next in Section 3.3, where we detail how we use Equation 3.1 with a memory address trace to determine the DRAM efficiency.

3.3 Processing address traces

In order to use the equations shown in the previous section, the variables must be obtained by processing memory request address traces. We assume that the memory request address trace is captured at the point where the interconnection network from the processor feeds the requests into the memory controller queue, allowing the order in which the requests are inserted into the memory controller to be preserved. We assume that the trace is otherwise devoid of any sort of
3.3. Processing address traces

timing information, such as the time when requests enter the queue.

Consider a simple case of a memory controller with a queue size of 4 and a DRAM with only two banks, Bank 0 and Bank 1, where each bank has only two rows (Row A and Row B for Bank 0 and Row X and Row Y for Bank 1). Now assume the memory address trace shown in Figure 3.2(a), where the top request (Bank 0 Row A) is the oldest.

To process the trace, we use the algorithm in Figure 3.3 to determine the $t_i$ values needed for Equation 3.1. Figure 3.3 shows two different heuristics for which we quantify the accuracy in Section 6.2. The first heuristic, which we call profiling + no activate overlap, is guarded by the “if(mode == no_activate_overlap)” statement and the second heuristic, which we call profiling + full activate overlap, is guarded by the “if(mode == full_activate_overlap)” statement. While this algorithm applies only to First-Ready First-Come First-Serve (FR-FCFS), it can be modified to handle other memory scheduling algorithms. In Section 6.3.4, we modify our algorithm to predict the DRAM efficiency of another out-of-order scheduling algorithm, “Most-Pending” [43]. Our approach is based on a sliding window profiling implementation where the window is the size of the memory controller queue.

We start with the assumption that initially Row A of Bank 0 and Row X of Bank 1 are open (Figure 3.2(a)). Following our algorithm, the first request that we encounter is to Bank 0 Row A, which is an opened row, so we can remove that request from the sliding window and increment $t_0$ by $T_{srvc\_req}$ (in Section 3.2), which is calculated to be 4 cycles using the DRAM parameters shown later in Table 5.3. The next request that we read in is to the same bank but this time to Row B. We leave this request in the sliding window, e.g. our memory controller queue, in the hopes that, by looking ahead to newer requests, we can find ones that are to the opened row. Moving on, it can be seen that the next request is again to Bank 0 Row A, allowing us to service it right away.

In this manner, we process all the requests up to Request 8, removing (servicing) requests that
3.3. Processing address traces

are to the opened rows (shown as shaded entries in Figure 3.2(b)), at which point our window size is 4, meaning the memory controller queue is now full. The \( t_i \) values can now be used to calculate the efficiency for this period using Equation 3.1. The \( t_i \) values are then reset to signify the start of a new prediction period and repeat our algorithm from the start. For no activate overlap, we then open the row of only the next oldest request, which is to Bank 0 Row B (Figure 3.2(c)). The DRAM efficiency over the entire length of the trace can also be computed using the method described at the end of Section 3.2.

While no activate overlap focuses on the prediction of efficiency by accounting for the requests in the memory controller queue that can help reduce the overhead of timing constraint delays, it does not account for the timing constraints of separate banks that can also be overlapped. More specifically, the process of precharging and activating rows in different banks of the same chip is only constrained by \( t_{RRD} \). With a \( t_{RC} \) value of 34 and a \( t_{RRD} \) of 8 as per our DRAM parameters[1,1], we can completely overlap the activation of rows in all 4 banks \((34/8 = 4.25 \text{ activate commands issuable per row cycle})\). Performance-wise, this means that even for a memory access pattern with minimal row access locality, the access pattern that finely interleaves requests to all 4 banks can achieve 4 times the bandwidth of an access pattern that has requests to only a single bank. As such, it is crucial to also take into account the overlapping of precharge and activates for applications with poor row access locality. Otherwise, the analytical model will significantly underestimate the DRAM efficiency. We model this using our second heuristic, full activate overlap, which instead opens the rows of the oldest requests to all banks, assuming that the delays associated with switching rows can be overlapped in all banks. This is shown in Figure 3.2(d) as Bank1 also switching from RowX to RowY in accordance with the code guarded by “mode = full activate overlap” in Figure 3.3. Leaving the opened row for Bank1 as RowX essentially forces the switching overhead of this bank to be paid separately from Bank0’s switching overhead from RowA to RowB (profiling + no activate overlap), potentially causing underestimation of DRAM efficiency. Conversely, switching
3.3. Processing address traces

the rows of all banks at the same time completely overlaps the switching overheads of all banks (full activate overlap), potentially causing overestimation of DRAM efficiency.

In essence, the two heuristics described provide upper and lower bounds on the amount of bank-level parallelism, either fully overlapping the row switching overheads for full activate overlap or completely serializing the row switching operations of the DRAM banks for no activate overlap. Since the actual amount of bank-level parallelism will be somewhere within the bounds, we also show results for a third heuristic, averaged, which averages the DRAM efficiency predictions of the first two heuristics. In Section 6.2, we will show the accuracy of our three heuristics described in this section in comparison to the DRAM efficiency measured in performance simulation. We will show that our averaged heuristic results in highest accuracy.
3.3. Processing address traces

![Sliding window profiling technique example](image)

Figure 3.2: Sliding window profiling technique example
3.3. Processing address traces

**Start:** Reset window\_size to 0

while (window\_size < memory controller queue size) {
  //while memory controller queue is not full
  Read in newest request, req, to sliding window
  if (req.row == opened\_row[req.bank]) {
    //check if request is to opened row, if so, service it
    // (first-ready first-come first-serve policy)
    remove req from window and from trace
    t\_req\_bank += T\_srv\_req //update t
  } else {
    // request is to different row, so store in queue and
    // check if later requests can be serviced first
    window\_size += 1 //add this req into window
  }
}

Calculate\_efficiency(t\_t) //uses t\_t to calculate efficiency

Reset(t\_t)

if (mode == no\_activate\_overlap) {
  //find oldest request, oldest\_req
  opened\_row[oldest\_req.bank] = oldest\_req.row
}

if (mode == full\_activate\_overlap) {
  //find oldest request of all banks, oldest\_req[]
  for (all i in banks) {
    opened\_row[oldest\_req[i].bank] =
      oldest\_req[i].row;
  }
}

Go back to **Start**

Figure 3.3: Sliding window profiling algorithm for processing memory request address traces
Chapter 4

Complexity-Effective Memory Access Scheduling

In this chapter, we first describe how the interconnection network that passes requests between the shader cores and memory controllers affects the memory request address streams. Next, we show how these effects can drastically reduce the efficiency of a memory controller that employs in-order request scheduling. Finally, we propose an elegant interconnection arbitration scheme that increases the efficiency of in-order request scheduling, allowing for a more complexity-effective design.

4.1 Motivation

The scaling of process technology has allowed processor throughput to increase exponentially. However, the properties of packaging prevent the pin count from increasing at the same rate. To alleviate this, chip designers maximize the percentage of pins on a chip used for transferring data. Increasing the amount of off-chip memory supported by the chip will likely require increasing the number of memory channels. Each additional memory controller requires its own set of address pins for communicating the requested memory addresses and control pins for relaying the commands to control the DRAM chips.

One approach to reduce the number of memory channels is to have the memory controller for each channel control multiple DRAM chips, thereby amortizing the address and control pin overhead across multiple DRAM chips [15]. However, there is a limit to how much of this “chip-
4.1. Motivation

level parallelism” is available. Consider a case where the typical memory request size is 64 bytes. In the CUDA Programming Model, the requests made by a half-warp of 16 threads accessing contiguous 32-bit words can be coalesced into a single 64-byte memory request [39]. If we assume that a single DRAM chip has a burst length of 4 and a bus width of 4 bytes (typical of modern graphics double-data rate (GDDR3) memory technology), then the maximum number of DRAM chips that a memory controller can potentially control without wasting data pin bandwidth is four (4 DRAM chips per memory controller × 4B bus width × 4 burst length = 64B). Furthermore, increasing the number of DRAM chips per memory controller reduces the number of read/write commands per activated row for each chip. If the memory access pattern of a particular application exhibits low row access locality, then DRAM efficiency can reduce greatly. This occurs when there is a lack of requests to service to the activated rows of other banks when one bank is constrained by the DRAM timing overhead needed to switch rows.

Figure 4.1: Measured DRAM efficiency for random uniform memory traffic with varying row access locality

Figure 4.1 shows the DRAM efficiency achieved using FR-FCFS scheduling of uniform random memory access patterns with varying degrees of row access locality measured using a stand-alone DRAM simulator based upon GPGPU-Sim [5]. In figure 4.1, rand1 is an artificially generated uniform random memory access pattern. Since the row designation of each memory access is random, the high number of rows per chip (4096 for the particular GDDR3 memory that we simulate [47]) imply that the chance of two back-to-back requests going to the same row in a particular bank is near-zero, meaning that a new row must be opened after the servicing of each
4.1. Motivation

memory request. In rand2, we replicate each random memory access once so that there are always two requests back-to-back to the same row. Similarly, in rand3, we replicate each random memory access so that there are always three requests back-to-back to the same row. As can be seen, fewer DRAM chips per memory controller effectively means more DRAM read and write commands to the rows of each chip to transfer the same total amount of data, thus increasing the DRAM efficiency.

The example above assumes a memory access pattern that is uniform random. Such a memory access pattern will tend to generate requests to all banks in a chip, thus maximizing “bank-level parallelism”. As such, the DRAM efficiency values presented in the above example represent the near-maximum efficiencies obtainable for the given row access locality parameters. On the other hand, a non-uniform memory access pattern may generate (a disproportionate amount of) requests to only a subset of the banks of a DRAM chip, which can cause DRAM efficiency to be drastically lower. For example, consider again the previous example where the row access locality is two and the number of DRAM chips per memory controller is two. If all of the requests go to a single bank, the DRAM efficiency plummets from 80.7% to 23.6%.

In order to maximize DRAM efficiency, modern DRAM controllers will schedule DRAM requests out of order in an effort to maximize the row access locality. Without doing so, the performance of in-order scheduling can be drastically worse when simple regular request streams (that would ordinarily schedule very efficiently in DRAM) from multiple sources become interleaved. Figure 4.2 shows a simplified example that illustrates this. Consider first a single shader core interacting with a single DRAM controller (Figure 4.2(a)). The request stream of this single shader core consists of two requests to the opened row, RowA. Assuming that Row A has already been loaded into the DRAM row-buffer, both requests can be serviced immediately and no row-switching latency is incurred. Now consider two shader cores interacting with the same DRAM controller (Figure 4.2(b)). The first shader core issues two requests to RowA while the second shader core issues two requests to
4.1. Motivation

Figure 4.2: Conceptual example showing effect of request interleaving due to interconnect (all requests to a single bank)

RowB. In trying to uphold fairness, a conventional interconnection network router arbiter may use round-robin arbitration, resulting in the two input request streams becoming finely interleaved when they pass through the router. In this case, an out-of-order scheduler would group together the requests to Row A and service them first before switching to Row B and servicing the remaining requests, thereby only paying the row-switching latency once. On the other hand, an in-order DRAM scheduler would service the requests as it receives them, therefore having to pay the row-switching latency three times, resulting in drastically lower DRAM throughput. (Note that the DRAM timing in Figure 4.2(b) is not shown to scale. In our experiments, servicing a single 64B request takes 4 cycles while the row switching delay is at least 25 cycles so the effect of request interleaving is actually much worse than illustrated.) If an intelligent network router recognized that, in this scenario, it should transfer all the requests of one shader core first before transferring the requests of the second shader core, then the performance obtainable using out-of-order DRAM
4.2. Effect of interconnect on row locality

Figure 4.3: Architecture (DRAM chips are off-chip)

scheduling could be achieved with a much simpler in-order FIFO DRAM scheduler.

In this thesis, we leverage this key observation to design an intelligent interconnection network coupled with a simple DRAM controller that can achieve the performance of a much more complex DRAM controller. Such a solution saves area and results in a shorter design cycle and less verification complexity.

4.2 Effect of interconnect on row locality

To determine the effects of memory request stream interleaving due to multiple shader core inputs, we augmented our simulator, which models the architectures shown in Figure 4.3, to measure the pre-interconnect row access locality, the row access locality of the memory request sent from the shader cores to the interconnect, versus the post-interconnect row access locality, the row access locality of the memory requests seen at the DRAM controllers after they traverse the interconnect, for a set of CUDA applications.

Figure 4.4 presents the measured post-interconnect row access locality divided by the pre-interconnect row access locality (Figure 4.4(a)) for crossbar, mesh, and ring networks as well as the
4.3 Grant holding interconnect arbitration

One of the fundamental pillars of conventional interconnection networks and their many arbitration policies is the concept of fairness, such that all input-output combinations receive equal service \[12\]. Without some degree of fairness in arbitration, nodes in the network may experience starvation.

Figure 4.4: Row access locality measured in number of requests per row activate command before the interconnect (Pre-Icnt) and after (Post-Icnt). HM = Harmonic Mean calculated ratio (Figure 4.4(b)). As can be seen, the row access locality decreases by almost 47% across all network topologies after the interconnect when arithmetically averaging the ratios of the pre-interconnect locality versus the post-interconnect locality across all applications.

Our configuration for Figure 4.4 has 28 shader cores and 8 memory channels (one DRAM controller per memory channel), which corresponds roughly to NVIDIA’s GT200 architecture of 30 shader cores and 8 memory channels \[6\].

4.3 Grant holding interconnect arbitration

One of the fundamental pillars of conventional interconnection networks and their many arbitration policies is the concept of fairness, such that all input-output combinations receive equal service \[12\]. Without some degree of fairness in arbitration, nodes in the network may experience starvation.
4.3. Grant holding interconnect arbitration

One common method of achieving fairness is to perform round-robin arbitration so that the most recent node that gets serviced becomes the lowest-priority in the next arbitration cycle to allow other nodes to get serviced. As seen in the previous section, such a policy can significantly reduce the amount of row access locality in the memory request stream seen by the DRAM controller. With an in-order memory request scheduler, this can lead to lower DRAM throughput. We therefore propose an interconnect arbitration scheme that attempts to preserve the row access locality exhibited by memory request streams from individual shader cores. To this end, we introduce two simple, alternative modifications that can be applied to any arbitration policy:

“Hold Grant” (HG): If input $I$ was granted output $O$ in the previous arbitration cycle, then input $I$ again gets the highest priority to output $O$ in the current cycle.

“Row-Matching Hold Grant” (RMHG): If input $I$ was granted output $O$ in the previous arbitration cycle, then input $I$ again gets the highest priority to output $O$ in the current cycle as long as the requested row matches the previously requested row match.

With HG, as long as a shader core has a continuous stream of requests to send to a DRAM controller, the interconnect will grant it uninterrupted access. RMHG has more constraints on when to preserve grant in order be more fair, but may not perform as well since it will not preserve any inherent bank-level parallelism found in the memory request stream, found to be important in reducing average thread stall times in chip multiprocessors by Mutlu et al. [33]. To maximize the benefit of using our proposed solution, the network topology and routing algorithm should be such that there is no path diversity. Common examples are a crossbar network or a mesh with non-adaptive routing. (We leave the development of interconnect arbitration schemes that preserve row access locality in network topologies with path diversity to future work.) This forces requests sent to the interconnect to leave in the same order. If requests were otherwise allowed to arrive at the DRAM controllers out of order, the row access locality in the memory request stream sent from shader cores may also be disrupted in this way.
4.3. **Grant holding interconnect arbitration**

In the standard interconnection network router pipeline, there are two stages where our hold-grant policies can be enforced: virtual channel allocation, which happens first, and switch allocation. In our experiments, we enforce our hold-grant policies during virtual channel allocation, as illustrated in Figure 4.5. If this is not done, virtual channel allocation will arbitrate packets randomly. In this case, enforcing hold-grant during switch allocation would be ineffective since the packets arbitrated during virtual channel allocation would already be interleaved. Implementation of our scheme requires a small amount of additional memory and additional comparators to keep track of and determine which inputs to hold grant on. This overhead is quantified in more detail in Section 4.5.
4.4 Interleaving due to multiple virtual channels

In interconnection networks, having multiple virtual channels can reduce network latency by reducing router pipeline stalls \[12\]. In certain network topologies and with certain routing algorithms, it can also be necessary to prevent deadlock.

In the context of memory request stream interleaving, multiple virtual channels can effectively cause a stream of memory requests sent from one source to one destination to arrive out of order. This occurs since a request that enters a network router gets assigned the first virtual channel that becomes available “on-the-fly”. (We refer to this as dynamic virtual channel assignment, or \textit{DVCA}.) When back-to-back requests from one source shader core are sent to one destination DRAM controller, they may get assigned to different virtual channels, where each virtual channel is essentially a different path. This is akin to having path diversity, which was explained in Section 4.3 to be detrimental in preserving row access locality. With requests taking multiple paths to the destination, order cannot be enforced.

To handle the case when multiple virtual channels are used in the interconnect, we propose a static, destination-based virtual channel assignment (\textit{SVCA}). While requests from one input node can still use multiple virtual channels in this assignment, all requests from one shader core (input node) to one specific DRAM controller (output destination node) are always assigned the same virtual channel in each router. In the case where there are virtual channels $VC_0$ to $VC_{v-1}$ for each router and $M$ DRAM controllers, then all requests to DRAM controller $M$ must use $VC_n$ and only $VC_n$, where $n = M \mod v$. Doing so allows for a fair virtual channel allotment across all input-output combinations. We evaluate the performance impact of using SVCA versus DVCA in Section 6.3.3 and show that applications obtain a speedup of up to 17.7% in a crossbar network with four virtual channels per port when using SVCA over DVCA.
4.5 Complexity comparison

Since our proposed changes are not limited to the DRAM controller circuitry, but also apply to the interconnection network, we attempt to define the complexity with a general-enough method that can be applied to any network topology. To this end, we quantify the complexity of our design based on the differences in two metrics, the amount of storage required in bits, and the number of bit-comparisons.

With FR-FCFS, whenever a new command is issued, every request in the DRAM controller queue must be checked to determine the best candidate for issuing [15][44]. For a queue size $Q$, this translates to $Q$ times the number of row bits + the number of bank bits. In the GDDR3 technology that we study, there are 4096 rows ($n_R$) and 4 banks ($n_B$), which is equal to 12 row bits $\lceil \log_2(n_R) \rceil$ and 2 bank bits $\lceil \log_2(n_B) \rceil$. For a GPU system with $M$ DRAM controllers, this means a maximum number of bit comparisons of $M \times Q \times (\lceil \log_2(n_R) \rceil + \lceil \log_2(n_B) \rceil)$ every DRAM command clock. This number represents an upper bound and can be much lower in certain scenarios. First, the average occupancy of the DRAM controller queue may be very low, especially if the application is not memory-intensive. Second, the DRAM request search may be performed in a two step process where only requests to available banks are selected to check for row buffer hits. Available banks include those that have pending requests and are not in the process of precharging (closing) a row or activating (opening) a row.

Since a DRAM controller that implements FR-FCFS needs to perform a fully-associative comparison, it does not need to be banked based on the number of DRAM banks. This allows the controller to maximize its request storage capability if, for instance, all requests go to a single bank. For our complexity-effective solution, we use a banked FIFO queue with one bank per DRAM bank. Doing so allows us to consider requests to all banks simultaneously, thus promoting bank-level parallelism. With this configuration, the DRAM controller considers up to $n_B$ requests.
4.5. Complexity comparison

each cycle, checking whether the row of the request matches the opened row for each bank to determine whether the DRAM controller can issue the request immediately or whether it needs to first open a new row. Furthermore, when requests are first inserted into the DRAM controller queue, the DRAM controller must check to which of the banks to insert the request, requiring an additional $[\log_2(n_B)]$ comparisons. This adds up to $M \times (n_B \times \lceil \log_2(n_R) \rceil + \lceil \log_2(n_B) \rceil)$ bit comparisons summed across all DRAM controllers.

Our interconnect augmentations to the interconnect, HG and RMHG, also require additional bit comparisons. We implement our grant-holding policy at the virtual channel allocator. If we are using HG, then we check whether input-output combinations granted in the previous arbitration cycle are also pending in this cycle. If so, we force allocation to choose the same input-output combinations again. This requires knowing which output was chosen in the last arbitration cycle for each input arbiter and, if there is a request to that output again this cycle, then it is chosen again. To do this, we store one bit for each input-output combination and set the bits of the combinations that were previously granted. We then compare these bits to the ones in the request matrix and, if there is a match, we clear the other request bits to the other inputs for the corresponding output and the other outputs for the corresponding inputs. This essentially forces the allocation algorithm to choose the same input-output combination for the next cycle because no other options are available. For a crossbar with one router, this totals to $C \times M$ bit comparisons and $C \times M$ additional bits of storage required, where $C$ is the number of shader cores. In the context of a mesh, the number of routers is equal to $C+M$ and an upper bound of nine total input and output ports for a two dimensional mesh. This is because we do not care about the output port in a shader core node to itself (since the shader core cannot sink memory requests) and the input port in a memory node from itself (since the memory controller cannot generate memory requests). (Some nodes on the edge or corners of the mesh will also have fewer ports.) This sums up to $(C+M) \times (20)$ bits compared and stored since each router will either have 4 inputs and 5 outputs or 5 inputs and 4 outputs for

46
## 4.5. Complexity comparison

Table 4.1: Calculated complexity (HMHG4 = Hash-Matching Hold Grant using 4-bit hashes)

<table>
<thead>
<tr>
<th></th>
<th>Bits Stored in ICNT</th>
<th>Bits Compared in ICNT</th>
<th>DRAM Scheduler</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRFCFS</td>
<td>0</td>
<td>0</td>
<td>M^2Q^2([log2(n_R)] + [log2(n_B)])</td>
<td>3584 bits compared</td>
</tr>
<tr>
<td>BFIFO+HG</td>
<td>C*M</td>
<td>C*M</td>
<td>0</td>
<td>224 bits stored</td>
</tr>
<tr>
<td>BFIFO+HG</td>
<td>20*(C+M)</td>
<td>20*(C+M)</td>
<td>0</td>
<td>720 bits stored</td>
</tr>
<tr>
<td>(MESH)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFIFO+RMHG</td>
<td>C*M + [log2(n_R)]<em>n_B</em>M</td>
<td>C*M + [log2(n_R)]*M</td>
<td>0</td>
<td>608 bits stored</td>
</tr>
<tr>
<td>(XBAR)</td>
<td></td>
<td></td>
<td></td>
<td>320 bits compared</td>
</tr>
<tr>
<td>BFIFO+RMHG</td>
<td>20*(C+M) + (C + M)*[log2(n_R)]<em>n_B</em>M</td>
<td>20*(C+M) + (C + M)*[log2(n_R)]*5</td>
<td>0</td>
<td>14,880 bits compared</td>
</tr>
<tr>
<td>(MESH)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFIFO+HMHG4</td>
<td>C<em>M + 4</em>M</td>
<td>C<em>M + 4</em>M</td>
<td>0</td>
<td>320 bits stored</td>
</tr>
<tr>
<td>(XBAR)</td>
<td></td>
<td></td>
<td></td>
<td>320 bits compared</td>
</tr>
<tr>
<td>BFIFO+HMHG4</td>
<td>20*(C+M) + (C+M)<em>4</em>5</td>
<td>20*(C+M) + (C+M)<em>4</em>5</td>
<td>0</td>
<td>1440 bits stored</td>
</tr>
<tr>
<td>(MESH)</td>
<td></td>
<td></td>
<td></td>
<td>1440 bits compared</td>
</tr>
</tbody>
</table>

a total of 20 input-output combinations.

An interconnect allocation policy that uses RMHG requires more bit comparisons since, in addition to the bit comparison and storage overheads incurred using HG, it must also check that back-to-back requests are indeed to the same row, thus requiring \([\log_2(n_R)]\) comparisons for each entry in the request matrix. Compounding this problem for the crossbar and, to a much greater extent, the mesh is that requests sent to the same output may be to ultimately different destinations, such as to different banks or, in the case of mesh, to different chips. One solution to account for this is to have each router keep track of the rows of the most recently granted request to the different possible destinations for each output. Therefore for a single router, which is the case of the crossbar, this would require a total of \([\log_2(n_R)]n_B*M\) additional bits of storage for the interconnect and \([\log_2(n_R)]*M\) bit comparisons. Accordingly, each router of a mesh would require \([\log_2(n_R)]*n_B*M\) additional bits of storage and \([\log_2(n_R)]*5\) bit comparisons, \([\log_2(n_R)]\) for each output. Summed across all routers in a mesh, the total requirements is \((C + M)*[\log_2(n_R)]*n_B*M\) additional bits of storage and \((C + M)*[\log_2(n_R)]*5\) bit comparisons.

Compared to using HG, a mesh network using RMHG can incur significant bit-comparison and
area overheads, even more than FR-FCFS. However, it is possible to reduce both bit comparison and bit storage overheads by matching a hash of the row instead of finding exact matches and storing fewer row entries than the maximum required of $n_B \times M$. We call this heuristic *Hash-Matching Hold Grant (HMHG)*. In our experiments, we match and store only a 4-bit hash of the row bits for the last granted request for each output for each router (HMHG4) instead of using RMHG. We found that using more bits did not improve performance. HMHG4 has an overhead of $(C+M) \times 4 \times M$ bit comparisons and $(C+M) \times 4 \times 5$ bits stored for the mesh and $4 \times M$ bit comparisons and $4 \times M$ bits stored for the crossbar. Table 4.1 summarizes our findings. The right-most column shows the calculated complexity for our baseline configuration with 28 shader cores, 8 memory controllers with 32-entry queues each, and 4096 rows and 4 banks per DRAM chip.

In the next chapter, we will describe our experimental methodology. In Section 6.3, we will quantify the effects of using our complexity-effective DRAM controller design in comparison to conventional in-order and out-of-order scheduling memory controllers.
Chapter 5

Methodology

5.1 Supporting CUDA applications on GPGPU-Sim

Table 5.1 shows the simulator’s configuration used to test the applications described in Chapter 2. Rows with multiple entries show the different configurations that we have simulated. Bold values show our baseline. We simulate all benchmarks to completion to capture all the distinct phases of each kernel in the benchmarks, especially the behavior at the tail end of the kernels, which can vary drastically compared to the beginning. If the kernels are relatively short and are frequently launched, the difference in performance when not simulating the benchmark to completion can be significant.

This thesis performs three main studies using the benchmarks described in Section 2.2. The first study validates our simulator against real hardware while the other two explore the performance effects of varying two key microarchitecture parameters: the DRAM controller queue size and the shader resources that affect the limit on the maximum number of CTAs that can execute on a shader core concurrently.

We issue CTAs in a breadth-first manner across shader cores, selecting a shader core that has a minimum number of CTAs running on it, so as to spread the workload as evenly as possible among all cores. We note that this breadth-first CTA distribution heuristic can occasionally lead to counter-intuitive performance results due to a phenomena we will refer to as CTA load imbalance. This CTA load imbalance can occur when the number of CTAs in a grid exceeds the number that can run concurrently on the GPU. For example, consider six CTAs on a GPU with two shader cores
5.1. Supporting CUDA applications on GPGPU-Sim

Table 5.1: Microarchitectural parameters for testing GPGPU-Sim (bolded values show baseline configuration)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Shader Cores</td>
<td>32</td>
</tr>
<tr>
<td>Warp Size</td>
<td>32</td>
</tr>
<tr>
<td>SIMD Pipeline Width</td>
<td>8</td>
</tr>
<tr>
<td>Number of Threads / Core</td>
<td>256 / 512 / 1024 / 1536 / 2048</td>
</tr>
<tr>
<td>Number of CTAs / Core</td>
<td>2 / 4 / 8 / 12 / 16</td>
</tr>
<tr>
<td>Number of Registers / Core</td>
<td>4096 / 8192 / 16384 / 24576 / 32768</td>
</tr>
<tr>
<td>Shared Memory / Core (KB)</td>
<td>4 / 8 / 16 / 24 / 32 (16 banks, 1 access/cycle/bank)</td>
</tr>
<tr>
<td>Constant Cache Size / Core</td>
<td>8KB (2-way set assoc. 64B lines LRU)</td>
</tr>
<tr>
<td>Texture Cache Size / Core</td>
<td>64KB (2-way set assoc. 64B lines LRU)</td>
</tr>
<tr>
<td>Interconnection Network</td>
<td>Full Crossbar</td>
</tr>
<tr>
<td>Number of Memory Channels</td>
<td>8</td>
</tr>
<tr>
<td>GDDR3 Memory Timing</td>
<td>$t_{CL}=9$, $t_{RP}=13$, $t_{RC}=34$</td>
</tr>
<tr>
<td></td>
<td>$t_{RAS}=21$, $t_{RCD}=12$, $t_{RRD}=8$</td>
</tr>
<tr>
<td>Bandwidth per Memory Channel</td>
<td>16 (Bytes/Cycle)</td>
</tr>
<tr>
<td>DRAM request queue capacity</td>
<td>8/16/32 / 64</td>
</tr>
<tr>
<td>Memory Controller</td>
<td>out of order (FR-FCFS) / in order (FIFO)</td>
</tr>
<tr>
<td>Branch Divergence Method</td>
<td>Immediate Post Dominator [16]</td>
</tr>
<tr>
<td>Warp Scheduling Policy</td>
<td>Round Robin among ready warps</td>
</tr>
<tr>
<td>Maximum Supported In-flight</td>
<td>64 / Unlimited</td>
</tr>
<tr>
<td>Requests per Shader Core</td>
<td></td>
</tr>
</tbody>
</table>

where at most two CTAs can run concurrently on a shader core. Assume running one CTA on one core takes time $T$ and running two CTAs on one core takes time $2T$ (e.g., no off-chip accesses and six or more warps per CTA—enough for one CTA to keep our 24 stage pipeline full). If each CTA in isolation takes equal time $T$, total time is $3T$ ($2T$ for the first round of four CTAs plus $T$ for the last two CTAs which run on separate shader cores). Suppose we introduce an enhancement that causes CTAs to run in time $0.90T$ to $0.91T$ when run alone (i.e., faster). If both CTAs on the first core now finish ahead of those on the other core at time $1.80T$ versus $1.82T$, then our CTA distributor will issue the remaining 2 CTAs onto the first core, causing the load imbalance. With the enhancement, this actually causes an overall slowdown since now 4 CTAs need to be completed by the first core, requiring a total time of at least $3.6T$. We carefully verified whether this behavior occurs by plotting the distribution of CTAs to shader cores versus time for both configurations being compared. This effect would be less significant in a real system with larger data sets and
therefore grids with a larger number of CTAs.

In previous work, rather than attempting to eliminate the effect, we simply noted where it occurred. In this work, we introduce a new metric for measuring processor throughput. Our conventional calculation of IPC for our GPU architecture is performed by dividing the sum of all instructions committed in all cores by the number of cycles it takes for the last core to finish execution of its CTAs. If the finish times of the cores differ significantly, then the final calculated IPC may not be an accurate measure of the true performance capabilities of the architecture. Cores that finish early may be assigned CTAs from other kernels to maximize utilization.

In Section 6.1.3 we measure the impact of running greater or fewer numbers of threads. We model this by varying the number of concurrent CTAs permitted by the shader cores, which is possible by scaling the amount of on-chip resources available to each shader core. There are four such resources: number of concurrent threads, number of registers, amount of shared memory, and number of CTAs. The values we use are shown in Table 5.1. The amount of resources available per shader core is a configurable simulation option, while the amount of resources required by each kernel is extracted using ptxas.

Varying the number of concurrent CTAs can cause certain benchmarks to suffer from severe CTA load imbalance. To address this, we use a new performance metric, \textit{IPC weighted by cycle} (IPCwbc), to present our data in and only in Section 6.1.3. With this metric, we first determine the IPC values of the individual shader cores first, using the finish time of each core as the cycle time instead of using the finish time of the slowest core. We then take a weighted average of the individual IPC values, where the weights are the individual cycle times. Finally, this weighted average is multiplied by the number of cores to obtain our performance metric for the aggregate IPC of a multicore architecture. Consider the conceptual example shown in Table 5.2. There are two types of CTAs: CTA A symbolizes an arithmetically-intensive workload while CTA B symbolizes a memory-intensive workload. Core 1, which is assigned CTA A, finishes much faster than Core
5.1. Supporting CUDA applications on GPGPU-Sim

2 while also executing more instructions. Our conventional IPC metric calculates the system IPC to be only 1.1 whereas, if we use the IPC weighted by cycle method described above, we obtain an IPC that is almost double, 2.0. Which one is more accurate? Assume if, after Core 1 finishes execution, we assign it another CTA of type B and, after Core 2 finishes execution, we assign it another CTA of type A. Now both cores must run both types CTAs, and should thus finish at the same time. In this case, each core executes 110 instructions in 110 cycles, for an aggregate system IPC of 2.0 regardless of which IPC metric we use. Table 5.2 clearly shows that IPCwbc was able to “predict” this.

Note that our metric is optimistic in assuming that there is no destructive (or constructive) interaction between the two cores. In the scenario shown in Table 5.2 Core 2 may only be able to finish execution of CTA B in 100 cycles because Core 1 is idle for most of the time and thus does not contend for shared resources, such as DRAM. In reality, Core 2 may take longer if Core 1 was assigned another CTA to run immediately after finishing its first CTA. Nevertheless, this metric is sufficient in quantifying performance when CTA load imbalance occurs. Consider the case where a microarchitecture modification results in an increase in IPC compared to a baseline. There are three possible causes: one, the modification actually improves some part of the system; two, the CTA load imbalance was more severe in the baseline than in the modified configuration; and three, there was more constructive interference in the modified configuration than in the baseline. If we then compare IPCwbc and see that the modified architecture achieved a higher IPCwbc than the baseline, then we can rule out CTA load imbalance as a possible cause. This is because IPCwbc becomes more optimistic as the severity of the CTA load imbalance increases. If the IPCwbc of the modified architecture (which has less severe CTA load imbalance) is still higher than the baseline in spite of this, then we can say with confidence that the performance improvement was not due to CTA load imbalance. On the contrary, if the IPCwbc of the baseline is higher, then we know that the IPC of the baseline is only lower due to CTA load imbalance.
5.2 Hybrid analytical modeling of DRAM

To evaluate our hybrid analytical model, we modified GPGPU-Sim [5] to collect the memory request address streams that are fed to the memory controllers for use in our analytical model. Table 5.1 shows the microarchitectural parameters used in this study. The advantage of studying a massively multi-threaded architecture that GPGPU-Sim is capable of simulating is that it allows us to stress the DRAM memory system due to the sheer number of memory requests to DRAM that can be in-flight simultaneously at any given time. In our configuration, we allow for 64 in-flight requests per shader core. This amounts to a maximum of 2048 simultaneous in-flight memory requests to DRAM. In comparison, Prescott has only eight MSHRs [8] and Willamette has only four MSHRs [21]. With an eight memory controller configuration, we use our model on 8 separate...
5.2. Hybrid analytical modeling of DRAM

Table 5.4: Benchmarks for analytical DRAM model

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Label</th>
<th>Suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black-Scholes option pricing</td>
<td>BS</td>
<td>CUDA SDK</td>
</tr>
<tr>
<td>Fast Walsh Transform</td>
<td>FWT</td>
<td>CUDA SDK</td>
</tr>
<tr>
<td>gpuDG</td>
<td>DG</td>
<td>3rd Party</td>
</tr>
<tr>
<td>3D Laplace Solver</td>
<td>LPS</td>
<td>3rd Party</td>
</tr>
<tr>
<td>LIBOR Monte Carlo</td>
<td>LIB</td>
<td>3rd Party</td>
</tr>
<tr>
<td>Matrix Multiply</td>
<td>MMC</td>
<td>3rd Party</td>
</tr>
<tr>
<td>MUMmerGPU</td>
<td>MUM</td>
<td>3rd Party</td>
</tr>
<tr>
<td>Neural Network Digit Recognition</td>
<td>NN</td>
<td>3rd Party</td>
</tr>
<tr>
<td>Ray Tracing</td>
<td>RAY</td>
<td>3rd Party</td>
</tr>
<tr>
<td>Reduction</td>
<td>RED</td>
<td>CUDA SDK</td>
</tr>
<tr>
<td>Scalar Product</td>
<td>SP</td>
<td>CUDA SDK</td>
</tr>
<tr>
<td>Scan Large Array</td>
<td>SLA</td>
<td>CUDA SDK</td>
</tr>
<tr>
<td>Matrix Transpose</td>
<td>TRA</td>
<td>CUDA SDK</td>
</tr>
<tr>
<td>Weather Prediction</td>
<td>WP</td>
<td>3rd Party</td>
</tr>
</tbody>
</table>

Figure 5.1: Measured DRAM efficiency and measured DRAM utilization averaged across all DRAM chips for each benchmark

memory request address streams per application. We test our model on the 14 different applications shown in Table 5.4, some of which are from the set described in Section 2.2 and some of which are from Nvidia’s CUDA software development kit [37]. Our only criterion for selecting applications from the two suites is that they must show greater than 10% DRAM utilization averaged across all DRAM chips for our given baseline microarchitecture configuration.

To illustrate the diversity of our application set, Figure 5.1 shows the DRAM efficiency and utilization measured by GPGPU-Sim in performance simulation averaged across all DRAM chips for each application. We simulate each application to completion in order to collect the full memory
5.3 Complexity-effective memory access scheduling

Table 5.5: Microarchitectural parameters for complexity-effective memory access scheduling (bolded values show baseline configuration)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shader Cores</td>
<td>28</td>
</tr>
<tr>
<td>Threads per Shader Core</td>
<td>1024</td>
</tr>
<tr>
<td>Interconnection Network</td>
<td>Ring, Crossbar, Mesh</td>
</tr>
<tr>
<td>Maximum Supported In-flight Requests per Multiprocessor</td>
<td>64</td>
</tr>
<tr>
<td>Memory Request Size (Bytes)</td>
<td>64</td>
</tr>
<tr>
<td>Interconnect Flit Size (Bytes)</td>
<td>32</td>
</tr>
<tr>
<td>DRAM Chips</td>
<td>16</td>
</tr>
<tr>
<td>DRAM Controllers</td>
<td>8</td>
</tr>
<tr>
<td>DRAM Chips per Controller</td>
<td>2</td>
</tr>
<tr>
<td>DRAM Controller Queue Size</td>
<td>8,16,32,64</td>
</tr>
<tr>
<td>DRAM Controller Scheduling Policy</td>
<td>First-Ready First-Come First-Serve (FR-FCFS), Naive FIFO, Banked FIFO</td>
</tr>
</tbody>
</table>

Table 5.6: Benchmarks for complexity-effective memory access scheduling

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Label</th>
<th>Suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast Walsh Transform</td>
<td>fwt</td>
<td>CUDA SDK</td>
</tr>
<tr>
<td>LIBOR Monte Carlo</td>
<td>lib</td>
<td>3rd Party</td>
</tr>
<tr>
<td>MUMmerGPU</td>
<td>mum</td>
<td>3rd Party</td>
</tr>
<tr>
<td>Neural Network Digit Recognition</td>
<td>neu</td>
<td>3rd Party</td>
</tr>
<tr>
<td>Ray Tracing</td>
<td>ray</td>
<td>3rd Party</td>
</tr>
<tr>
<td>Reduction</td>
<td>red</td>
<td>CUDA SDK</td>
</tr>
<tr>
<td>Scalar Product</td>
<td>sp</td>
<td>CUDA SDK</td>
</tr>
<tr>
<td>Weather Prediction</td>
<td>wp</td>
<td>3rd Party</td>
</tr>
<tr>
<td>Nearest Neighbor</td>
<td>nn</td>
<td>Rodinia</td>
</tr>
</tbody>
</table>

request address trace.

5.3 Complexity-effective memory access scheduling

To evaluate our design, we again used GPGPU-Sim [5]. We implemented our changes to the interconnection network simulated using the versatile interconnection network simulator introduced by Dally et al. [12].

Table 5.5 shows the microarchitectural parameters used in this study. We evaluate our design on the 9 different applications shown in Table 5.6 across three network topologies: ring, mesh, and full crossbar. To occupy all nodes in a square mesh, we keep the number of memory controllers constant.
5.3. Complexity-effective memory access scheduling

at eight and slightly reduce the number of shader cores to 28, resulting in a 6x6 mesh. We found that certain applications that exhibit low off-chip traffic are not sensitive to varying memory controller configurations. In order to have results that are more meaningful, we focused our study on only those applications that meet all three of our selection criteria: first, the total processor throughput of the application must be less than 75% of the peak IPC for the baseline configuration; second, the DRAM utilization, percentage time spent of a memory channel transferring data over all time, averaged across all DRAM controllers must be over 20%. Finally, the DRAM efficiency averaged across all DRAM controllers must be less than 90%. Following these three criteria ensures that we study only applications that are truly memory-sensitive. We use 3 applications from NVIDIA’s CUDA software development kit (SDK) [37], 5 applications from the set used by Bakhoda et. al [5], and one application from the Rodinia benchmark suite introduced by Che et al. [9]. We simulate each application to completion.
Chapter 6

Experimental Results

6.1 GPGPU-Sim simulation

6.1.1 Baseline

We first simulated our baseline GPU configuration with the bolded parameters shown in Table 5.1. Figures ?? and ?? show the instruction classification and memory instruction breakdown of the CUDA benchmarks introduced in Section 2.2. As shown, these benchmarks exhibit a diverse range of computation, control flow, and memory intensity. The memory access patterns in terms of the types of memory spaces accessed also vary greatly from application to application. Figure 6.3 shows the performance of our baseline configuration (for the GPU only) measured in terms of scalar instructions per cycle (IPC). In the right-most set of bars, SDK represents the harmonic mean for all the applications from Nvidia’s CUDA Software Development Kit that we simulated, while HM represents the harmonic mean of the new applications that we introduced in Section 2.2. Evidently, there is a large gap in performance between the CUDA SDK applications and real applications developed by other authors. For comparison, we also show the performance assuming a perfect memory system with zero memory latency. Note that the maximum achievable IPC for our configuration is 224 (28 shader cores x 8-wide pipelines). We also validated our simulator against an Nvidia GeForce 8600GTS, which is top of the mid-range lineup for the G80 series, by configuring our simulator to use four shaders and two memory controllers. The IPC of the GPU hardware, as shown in Figure 6.4(a), was estimated by dividing the dynamic instruction count measured (in PTX instructions) in simulation by the product of the measured runtime on hardware and the shader
6.1. GPGPU-Sim simulation

![Instruction Classification](image1)

**Figure 6.1: Instruction Classification**

![Memory Instructions Breakdown](image2)

**Figure 6.2: Memory Instructions Breakdown**

clock frequency [36]. Figure 6.4(b) shows the scatter plot of IPC obtained with our simulations mimicking the 8600GTS normalized to the theoretical peak IPC versus the normalized IPC data measured using the 8600GTS. The correlation coefficient was calculated to be 0.899. One source of difference, as highlighted by the data for CP which actually achieves a normalized IPC over 1, is due to compiler optimizations in *ptxas*, the assembler that generates the final binary run on the GPU, which reduce the instruction count on real hardware[5]. Overall, the data shows that applications that perform well in real GPU hardware perform well in our simulator and applications that perform poorly in real GPU hardware also perform poorly in our simulator. In the following sections, we explore reasons why some benchmarks do not achieve peak performance.
6.1. GPGPU-Sim simulation

![Baseline performance (HM=Harmonic Mean)](image)

Figure 6.3: Baseline performance (HM=Harmonic Mean)

![Performance comparison with 8600GTS](image)

(a) (b)

Figure 6.4: Performance comparison with 8600GTS

6.1.2 DRAM utilization and efficiency

In this section we explore the impact that memory controller design has on performance. Our baseline configuration uses an *Out-of-Order* (OoO) *First-Ready First-Come First-Serve* (FR-FCFS) memory controller with a capacity of 32 memory requests. Each cycle, the OoO memory controller prioritizes memory requests that hit an open row in the DRAM over requests that require a precharge and activate to open a new row. Against this baseline, we compare a simple *First-In First-Out* (FIFO) memory controller that services memory requests in the order that they are received, as well as a FR-FCFS OoO controller with other varying input buffer capacity sizes of 8 (OoO8), 16 (OoO16), and 64(OoO64). We measure two metrics besides performance (based on

---

5We only simulate the input PTX code which, in CUDA, *ptxas* then assembles into a proprietary binary format that we are unable to simulate.
6.1. GPGPU-Sim simulation

Figure 6.5: Impact of DRAM memory controller optimizations

![Speedup Graph](image)

Figure 6.6: DRAM Efficiency

![DRAM Efficiency Graph](image)

Figure 6.7: DRAM Utilization

![DRAM Utilization Graph](image)

IPC): the first is *DRAM efficiency*, which is the percentage of time spent sending data across the pins of DRAM over the time when there are any memory requests being serviced or pending in the memory controller input buffer; the second is *DRAM utilization*, which is the percentage of time spent sending data across the DRAM data pins over the entire kernel execution time. These two measures can differ if an application contains GPU computation phases during which it does not
6.1. GPGPU-Sim simulation

access DRAM (e.g., if it has been heavily optimized to use “shared memory”).

Figure 6.5 compares the performance of our baseline to other memory controller configurations. We observe that AES, CP, LPS, MMC, NQ, and STO exhibit almost no slowdown for FIFO. Figure 6.6 shows AES and STO obtain over 75% DRAM efficiency. Close examination reveals that at any point in time all threads access at most two rows in each bank of each DRAM, meaning that a simple DRAM controller policy suffices. Furthermore, Figure 6.7 shows that AES and STO have low DRAM utilization despite the fact that they are data-processing applications. Both these applications make extensive use of shared memory (see Figure 6.2). NQ and CP have very low DRAM utilization, making them insensitive to memory controller optimizations. Performance is reduced by over 40% when using FIFO for BFS, LIB, MUM, RAY, and WP. These benchmarks all show drastically reduced DRAM efficiency and utilization with this simple controller.

6.1.3 Are more threads better?

GPUs can use the abundance of parallelism in data-parallel applications to tolerate memory access latency by interleaving the execution of warps. These warps may either be from the same CTA or from different CTAs running on the same shader core. One advantage of running multiple smaller CTAs on a shader core rather than using a single larger CTA relates to the use of barrier synchronization points within a CTA [46]. Threads from one CTA can make progress while threads from another CTA are waiting at a barrier. For a given number of threads per CTA, allowing more CTAs to run on a shader core provides additional memory latency tolerance, though it may imply increasing register and shared memory resource use. However, even if sufficient on-chip resources exist to allow more CTAs per core, if a compute kernel is memory-intensive, completely filling up all CTA slots may reduce performance by increasing contention in the interconnection network and DRAM controllers.

We explore the effects of varying the resources that limit the number of threads and hence CTAs
that can run concurrently on a shader core, without modifying the source code for the benchmarks. We vary the amount of registers, shared memory, threads, and CTAs between 25\% to 200\% of those available to the baseline. The results are shown in Figure 6.8 where we report speedup using both IPC and IPCwbc, explained in Section 5.1. For the baseline configuration, some benchmarks are already resource-constrained to only 1 or 2 CTAs per shader core, making them unable to run using a configuration with less resources. We do not show bars for configurations that for this reason are unable to run. Furthermore, CPS, LPS, LIB, and WP do not have enough CTAs to make use of increased shader resources, so performance remains constant as the CTA limit is increased beyond 100\%. For these benchmarks, all CTAs can run simultaneously for the baseline configuration. We first note that analyzing the speedup using IPCwbc (Figure 6.8(b)) results in speedups that are either monotonic or contain a single distinct peak as the CTA limit is scaled, removing the multiple

Figure 6.8: Effects of varying number of CTAs
peaks in speedup that occur for some benchmarks (BFS, DG, MMC, and MUM) when instead using IPC (Figure 6.8(a)). A closer analysis at the output logs showed that it is indeed the CTA load imbalance issue that causes these erratic results in Figure 6.8(a). For this reason, the rest of this subsection will focus on using IPCwbc (Figure 6.8(b)) to analyze the results. NQ shows little change when varying the number of CTAs since it has very few memory operations. For AES, DG, LPS, MUM, NQ, STO, and WP, performance increases monotonically as more CTAs per core are used. Each CTA in NQ and STO uses all of the shared memory in the baseline configuration, therefore increasing shared memory by half for the 150% configuration results in no increase in the number of concurrently running CTAs while doubling the amount of shared memory for the 200% configuration yields a large performance improvement. While AES and DG are capable of using the additional shader core resources by increasing the number of concurrently running CTAs, the increase in performance beyond the baseline configuration is minuscule (1.3% increase in IPCwbc arithmetically averaged across the two benchmarks when increasing shader resources from 100% to 200%). When the number of concurrently running CTAs increase, there is greater contention in the shared DRAM system, resulting in increased memory latency of 58.3% (AES) and 72.7% (DG). For MUM, the speedup is higher (6.0%) because the memory latency increase is much smaller at only 4.9%. BFS, NN, NEU, and RAY show distinct optima in performance when the CTA limit is at 150%, 50%, 150%, and 100% of the baseline configuration, respectively. Above these limits, we observe DRAM efficiencies decrease and memory latencies increase, again suggesting increased contention in the memory system. For configuration with limits below the optima, the lack of warps to hide memory latencies reduces performance.

Given the widely-varying workload-dependent behavior, always scheduling the maximal number of CTAs supported by a shader core is not always the best scheduling policy. We leave for future work the design of dynamic scheduling algorithms to adapt to the workload behavior.
6.2 Hybrid analytical modeling of DRAM

In Section 6.2.1, we will first show our results for the three heuristics that we described in Section 3.3. We show that our model obtain a correlation of 72.9% with an arithmetic mean absolute error of 11.2%. In Section 6.2.2, we provide an in-depth analysis on the cause of our errors. In Section 6.3.4, we perform a sensitivity analysis of our analytical DRAM model across different microarchitecture configurations.

6.2.1 DRAM efficiency prediction accuracy

Figure 6.9 compares the DRAM efficiency measured in performance simulation to the modeled DRAM efficiency when using the profiling technique described in Section 3.3. For clarity, we show only the arithmetic average of the efficiency across all DRAM controllers of each application. The arithmetic average of these absolute errors is 18.5% for no activate overlap and 17.0% for full activate overlap. The corresponding correlation is 72.2% and 65.9% respectively. Of the 15 applications studied, nine are predicted more accurately by full activate overlap, indicating that accounting for bank-level parallelism, even naively, is important in predicting DRAM efficiency.
6.2. Hybrid analytical modeling of DRAM

When we average the predictions of these two heuristics for \textit{averaged}, the error is reduced to 11.2\% and the correlation increases to 72.9\%. We described in Section 3.3 that, without accounting for overlap of row activates, we expect our predictions to underestimate the measured efficiency (\textit{profiling + no activate overlap}). (We explain why the results of some applications do not match these expectations in Section 6.2.2.)

In order to confirm our expectations, we propose a new metric, \textit{polarity}, which is defined as the arithmetic average of the \textit{non-absolute} errors divided by the arithmetic average of the \textit{absolute} errors. The value of polarity can range between -1 and 1, where -1 means that the test value (predicted efficiency) is always less than the reference value (measured efficiency), 1 means that the test value is always greater than the reference value, and 0 means that on average, the error is not skewed positively or negatively in anyway. We calculated the arithmetic mean of the non-absolute errors for \textit{profiling + no activate overlap} to be -14.1\%, meaning a polarity of -0.81. This implies a strong tendency to underestimate the DRAM efficiency, confirming our expectations. On the other hand, the polarity of average arithmetic error of \textit{profiling + full activate overlap} is 0.98, meaning it almost always overestimates the DRAM efficiency. We expect this to occur since the frequency
6.2. Hybrid analytical modeling of DRAM

Figure 6.11: Comparison of measured DRAM efficiency to predicted DRAM efficiency (1 data point per memory controller per benchmark)

of issuing activates is constrained by $t_{RRD}$. Moreover, a row activate to a bank cannot be issued as long as there are still requests to the current row of that bank, further constraining the frequency of issuing activates. Our profiling + full activate overlap heuristic captures neither of these, essentially meaning that assuming full overlap of the row activates will always be optimistic in predicting the DRAM efficiency. Figure 6.11 shows the scatter plots of predicted DRAM efficiency using our three heuristics to the measured DRAM efficiency. Each dot represents the measured DRAM efficiency versus predicted for a single DRAM controller. Several facts become immediately apparent. First, most of the data points for no activate overlap tend to be under the X=Y line, meaning that this modeling heuristic tends to underestimate the DRAM efficiency, while full activate overlap behaves the opposite. Second, the data points in Figure 6.11(c) are closest to the X=Y line, further illustrating the improved accuracy of averaging the predictions obtained using our first two heuristics.
6.2.2 Modeling error analysis

Applications where one technique generates significantly poor predictions tend to be predicted much more accurately by the other technique (BS, LPS, LIB, MMC, MUM, NN, RED, TRA). It is encouraging to see this because it implies that a heuristic such as averaged that combines no activate overlap and full activate overlap can reduce the mean absolute error.

We postulate that for applications with poor row access locality, the overlapping of row activates has more significant impact. Consequently full activate overlap should better predict DRAM efficiency than no activate overlap, which serializes the precharge and activate delays and potentially underestimates the DRAM efficiency. Conversely, we postulate that applications with good row access locality will not benefit as much from row activate overlapping, which will make full activate overlap overestimate the efficiency.

To verify this, we obtain the average row access locality by dividing the total number of read and write commands by the number of activate commands per memory controller and arithmetically averaging these values across all memory controllers for each application. Figure 6.12 shows this average row access locality. The three applications with the lowest row access locality, MUM, TRA, and WP, follow our hypothesis and are predicted more accurately by full activate overlap than no activate overlap.

A closer look at Figure 6.9 reveals that, for BS, LPS, MMC, and NEU, the measured DRAM efficiency is actually lower than that predicted by no activate overlap which we expected to be a lower bound on the DRAM efficiency. This result for NEU can be explained by its extremely high row access locality, essentially causing our model to predict close to 100%, regardless of the heuristic. In this case, the effects of neglecting other timing constraints, such as the read-to-write or write-to-read latency becomes exposed. For BS, LPS, and MMC, we measure the average DRAM controller queue occupancy when there is at least one request in the queue, shown in Figure 6.13.
We find that the average DRAM controller queue occupancy for these three applications is the lowest among all applications, all below 20%. This essentially means that our profiling technique is too aggressive in that the sliding window (whose length is equal to the modeled memory controller queue size) that it uses is too large relative to the average memory controller queue occupancy in simulation. As such, the profiling technique is finding row access locality that does not exist in simulations since the majority of the requests have not arrived to the memory controller yet. (Note that the profiling technique used does not take timing information into account so it assumes that the memory controller (sliding window) can always look ahead in the memory request address trace by an amount equal to the memory controller queue size.)

6.2.3 Sensitivity analysis

In this section, we present the accuracy of our results while sweeping across different key parameters: the DRAM controller queue size, the number of parallel DRAM chips per DRAM controller, and the DRAM controller scheduling policy.

We first present Figure 6.14, which shows the arithmetic mean absolute error of our three
6.2. Hybrid analytical modeling of DRAM

Figure 6.13: Average DRAM controller queue occupancy averaged across all DRAM chips for each benchmark (AM = arithmetic mean across all benchmarks)

Figure 6.14: Arithmetic mean absolute error versus DRAM controller queue size averaged across all DRAM controllers and across all benchmarks

profiling heuristics, full activate overlap, no activate overlap, and averaged, across four different DRAM controller queue sizes of 8, 16, 32 and 64. For clarity, we only present the values averaged across all DRAM controllers and all benchmarks. In general, the error tends to increase as the queue size increases. We attribute this to the fact that, since the queue occupancy is not always 100% for some benchmarks for our baseline DRAM controller queue size of 32 anyways (as explained in Section 6.2.2), the occupancy will be even lower when the queue size is increased to 64. This means that the size of the profiling window that we use is too large compared to what is occurring in simulation. Moreover, the efficiency of all DRAM controllers decreases as the queue size is reduced,
6.2. Hybrid analytical modeling of DRAM

![Hybrid analytical modeling of DRAM](image)

Figure 6.15: Arithmetic mean absolute error versus number of DRAM chips per DRAM controller

meaning the absolute value of the difference will decrease as well.

Figure 6.15 shows the modeling error as the number of DRAM chips connected in parallel (to increase bandwidth) to the DRAM controller is varied. Increasing the number of DRAM chips per controller is an effective way of increasing the total off-chip bandwidth to the chip without incurring much additional circuit logic on the chip (although it will increase the chip package size due to an increased number of pins needed for data transfer). Since our memory request data sizes are fixed at 64B and each of our DRAM chips is capable of transferring 16 bytes of data per command, the limit in this amount of parallelism is no more than 4 DRAM chips per controller. Furthermore, as this parallelism is increased, the memory access pattern becomes much more important in determining the DRAM efficiency. This is because more DRAM chips per controller means less read and write commands per request, thus reducing $T_{srvc,req}$. The number of requests per row (e.g., the row access locality) thus needs to be higher to maintain the DRAM efficiency as shown in Equation 3.1. Moreover, reducing $T_{srvc,req}$ can reduce the value of the denominator of Equation 3.1, $\text{MAX}(t_{RC}, t_{RP} + t_{RCD} + t_j)$. This means that there is less time to overlap the row activate commands of different banks, implying that no activate overlap should be more accurate than full activate overlap. As we see in Figure 6.15, this is indeed the case.

Finally, we also try implementing our hybrid analytical model for another memory scheduling
6.2. Hybrid analytical modeling of DRAM

Figure 6.16: Arithmetic mean absolute error of predicted DRAM efficiency of “Most Pending” memory scheduling algorithm averaged across all DRAM chips for each benchmark (AM = arithmetic mean across all benchmarks)

policy, “Most Pending” [43]. After all requests to a row have been serviced, our default memory scheduling policy, First-Ready First-Come First-Serve (FR-FCFS) [43], will then open the row of the oldest request. “Most Pending,” conversely, will open the row that has the most number of requests pending to it in the queue (e.g., essentially a greedy bandwidth maximization scheme). The algorithm to implement this in our analytical model is virtually the same as the one shown in Figure 3.3 except, instead of finding the oldest request whenever we switch rows, we now find the oldest request corresponding to the row that has the most number of requests. Figure 6.16 shows the arithmetic mean absolute error for our baseline configuration. We see that for this scheduling policy, the error is comparable to that of FR-FCFS.

In conclusion, we see that our model is quite robust and the error stays comparable when varying across key microarchitectural parameters and even for a different memory scheduling policy. With smarter heuristics, such as the simple one described in Section 6.2.2, this error can be reduced even further.
6.3 Complexity-effective memory access scheduling

In Section 6.3.1, we will first show how the interconnect modifications of our proposed solution as described in Section 4.3 performs in comparison to naive FIFO and FR-FCFS. Where trends across different network topologies are similar, we only present results for the crossbar network. Section 6.3.2 provides an in-depth analysis on how our interconnect modifications effect the memory request address streams seen at the DRAM controllers. In Section 6.3.3, we compare static destination-based virtual channel assignment to dynamic virtual channel assignment. In Section 6.3.4, we perform a sensitivity analysis of our design across different microarchitectural configurations.

6.3.1 Complexity-effective DRAM scheduling performance

Since RMHG was found in Section 4.5 to be relatively area intensive, we only show results for the Hold-Grant (HG) modification and Hash-Matching Hold-Grant modification using 4 bit hashes (HMHG4).
6.3. Complexity-effective memory access scheduling

![Graph of DRAM efficiency](image)

(a) DRAM efficiency

![Graph of Average memory latency](image)

(b) Average memory latency

Figure 6.18: DRAM efficiency and memory latency (HM = Harmonic mean, Queue size = 32)

We present the per-application IPC normalized to FR-FCFS for the crossbar network in Figure 6.17(a). The banked FIFO controller outperforms the unbanked controller across the board, indicating that the banked FIFO controller’s better proficiency at exploiting bank-level parallelism drastically outweighs the additional potential queueing capacity of the unbanked controller. The row access locality preservation in Figure 6.17(b) is calculated by dividing the post-interconnect locality of the various configurations by the pre-interconnect locality for FR-FCFS. Our interconnect modifications help preserve over 70% of the row access locality while FR-FCFS scheduling results in only 56.2% preservation.

We show the effect of our HG and HMHG4 interconnect modifications on the DRAM efficiency and memory latency, shown in Figure 6.18 for the crossbar network. Figure 6.18 shows that naive scheduling of memory requests in a system that supports thousands of in-flight memory requests can cause the memory latency to increase exponentially. Our HM and HMHG4 is able to reduce the latency of an in-order, banked DRAM controller (BFIFO) by 33.9% and 35.3% respectively. Fur-
thermore, both of our interconnect modifications, HM and HMHG4, improve the DRAM efficiency of BFIFO from 55% to 63%.

6.3.2 Row streak breakers

In order to gain a better understanding of what causes the performance gap between our complexity-effective design and FR-FCFS, we perform an in-depth analysis of the scheduled memory requests in relation to the DRAM controller queue contents for the banked FIFO scheduler. First, we define a sequence of row-buffer hits to any single bank as a single row streak. Whenever a DRAM bank switches from an old Row X to a new Row Y (i.e., when a row streak ends), we search backwards through the FIFO queue corresponding to this DRAM bank starting from the pending request to Row Y, looking for any other requests to Row X. We define these requests that we are looking for as stranded requests since, in a FR-FCFS scheduler, these stranded requests would also be serviced before the row switches. If we find a stranded request, we look at the requests between the stranded request and the pending request to Row Y. In an ideal interconnection network, these requests, which we call row streak breakers, would be prioritized to arrive at the DRAM controller after the stranded request, thus maximizing row access locality. Furthermore, a FR-FCFS scheduler would prioritize these row streak breakers lower than the stranded request.

Figure 6.19 characterizes the source of the row streak breakers for the banked FIFO configuration with and without our interconnect augmentations. Each time a stranded request is found using our method described in the previous paragraph, we increment the counter for each category once if any requests in the corresponding category are found in the set of row streak breakers. In this way, for any set of row streak breakers, more than one category may be credited. We categorize row streak breakers into three possibilities, those originating from a different shader core, those from the same core but from a different CTA, and those from the same CTA. It can be immediately seen that row streak breakers originating from different shader cores dominate the distribution.
6.3. Complexity-effective memory access scheduling

Furthermore, our interconnect augmentations significantly reduce the number of these particular row streak breakers. Note that the bars in Figure 6.19 do not add up to 100% because sometimes there are no stranded requests. In these situations, FR-FCFS would make the same scheduling decision as our banked FIFO scheduler, oldest-first. In other words, shorter bars indicate that the performance gap between the corresponding configuration and FR-FCFS is also smaller.

In virtually all cases, our interconnect augmentations reduce the number of row streak breakers due to different shader cores to less than 10% of the total row streaks. The one obvious exception is neu, where our interconnect augmentations can reduce the row streak breakers to no less than 30% of the total number of row streaks. Measuring the average number of shader cores that results in a row streak originate showed that this measure was close to 1 for most other applications, explaining why an input-based hold-grant mechanism works so well in this architecture. The largest exception is neu, where the requests of every row streak when using the FR-FCFS scheduling policy are from more than four different shader cores on average (4.18). In this case, FR-FCFS will be able to capture row access locality that our interconnection augmentations cannot.
6.3. Complexity-effective memory access scheduling

In this section, we compare the performance results of using static destination-based virtual channel assignment (SVCA) to more conventional dynamic virtual channel assignment (DVCA) in the case when there are multiple virtual channels per router port.

Figure 6.20 shows the IPC harmonically averaged across all applications for different virtual channel configurations: one virtual channel per port (vc1), two virtual channels per port with SVCA (svc2), two VCs with DVCA (dvc2), four VCs with SVCA (svc4), and four VCs with DVCA (dvc4). Here, we increase the number of virtual channels while keeping the amount of buffering space per virtual channel constant. When using our complexity-effective DRAM scheduling solution, BFIFO+HG and BFIFO+HMHG4, performance decreases by 21.3% and 21.7% respectively as the number of virtual channels is increased from one to four while using DVCA. Using SVCA (svc4) recovers most of this performance loss, improving upon dvc4 by 18.5% and 17.7%. In general, we find that adding virtual channels does not improve IPC, which matches one of the findings of Bakhoda et al. [5] that CUDA workloads are generally insensitive to interconnect latency.

6.3.4 Sensitivity analysis

To show the versatility of our interconnect modifications, we test them across two major microarchitectural parameters: the network topology and the DRAM controller queue size.

As Figure 6.21 shows, our interconnect modifications perform fairly similarly for two very differ-
6.3. Complexity-effective memory access scheduling

Figure 6.21: Harmonic mean IPC across all applications for different network topologies (Queue size = 32)

ent network topologies: a crossbar and a mesh. For these topologies, HMHG4 achieves 86.0% and 84.7% of the performance of FR-FCFS for memory-bound applications while requiring significantly less bit comparisons than FR-FCFS. For comparison, we also present the results for a ring network, which requires a minimum of two virtual channels for deadlock avoidance [12]. The deadlock avoidance algorithm for the ring imposes restrictions on which virtual channels can be used by any input-output combination to remove cyclic dependencies. To provide adequate bandwidth, we provision the ring network with a 512-bit wide datapath similar to the one in Larrabee [49]. While a ring network has the same number of nodes as a mesh, each router has only three input and output ports, one to the left, one to the right, and one to the shader core or DRAM controller that it is connected to. In comparison, each mesh router has 5 input and output ports, so the complexity as detailed in Section 4.5 of the ring will be 60% (or 3/5ths) of that of the mesh. As shown, our interconnect modifications do not work as well for the ring network due to the interleaving of memory request streams from having multiple virtual channels. In-depth analysis showed that the ring network had more than six times as many row streak breakers as the mesh and crossbar networks, thus achieving only 6.3% speedup over a banked FIFO memory controller with no interconnect modifications. We leave the development of an interconnect modification that works better for this case to future work.

Figure 6.22 shows the performance of our interconnect modifications harmonically averaged
6.3. Complexity-effective memory access scheduling

Figure 6.22: Harmonic mean IPC normalized to FR-FCFS for different DRAM controller queue sizes (Crossbar network)

across all applications in a crossbar network and with different DRAM controller queue sizes. For the smallest configuration of DRAM controller queue size 8, BFIFO+HMHG4 achieves 91.0% of the IPC of FR-FCFS. While the performance relative to FR-FCFS decreases as the DRAM controller queue size increases, the complexity of an increasing FR-FCFS DRAM controller queue also increases since the number of comparisons per cycle scales for FR-FCFS \cite{15, 44} but remains constant for a banked FIFO controller with our interconnect modifications.
Chapter 7

Related Work

This chapter describes related work. First we summarize prior work in GPU architecture simulation and benchmarking. Then we introduce other analytical models presented in the field of computer architecture, some specific to DRAM. Finally we describe recent work in DRAM controller design.

7.1 GPU architecture simulation

Existing graphics-oriented GPU simulators include Qsilver [51], which does not model programmable shaders, and ATTLILA [14], which focuses on graphics specific features. Ryoo et al. [45] use CUDA to speedup a variety of relatively easily parallelizable scientific applications. They explore the use of conventional code optimization techniques and take advantage of the different memory types available on NVIDIA’s 8800GTX to obtain speedup. While their analysis is performed by writing and optimizing applications to run on actual CUDA hardware, we use our novel performance simulator to observe the detailed behavior of CUDA applications upon varying architectural parameters.

We quantified the effects of varying the DRAM controller queue configuration and shader core resource limits which, to our knowledge, has not been published previously. While the authors of the CUDA applications which we use as benchmarks have published work, the emphasis of their papers was not on how changes in the GPU architecture can affect their applications [3, 7, 17, 18, 20, 27–29, 41, 45, 48, 54]. To our knowledge, GPGPU-Sim is the first published detailed performance simulator that supports running CUDA applications.
As of July 2009, there are several application suites suitable for GPU architecture study. Mahesri et al. introduce a class of applications for visualization, interaction, and simulation [26]. They propose using an accelerator architecture (xPU) separate from the GPU to improve performance of their benchmark suite. Ryoo et al.’s Parboil benchmark suite [45] has been introduced specifically to target Nvidia GPUs, being written in CUDA. Parboil provides a central script that provides a common interface for execution of all applications on both the GPU and CPU. Introduced most recently, the Rodinia benchmark suite [9] targets multicore CPUs and Nvidia GPUs by providing application source code written in both OpenMP and CUDA.

7.2 Analytical models

Ahn et al. [2] explore the design space of DRAM systems. They study the effects on throughput of various memory controller policies, memory controller queue sizes, and DRAM timing parameters such as the Write to Read delay and the Burst Length. More relevant to our work, they also present an analytical model for expected throughput assuming a “random indexed” access pattern and a fixed “record length”. This essentially means that they assume a constant fixed number of requests accessed per row and a continuous stream of requests whereas our model handles any memory access pattern. Furthermore, they only show the results of their analytical model for two micro-benchmarks while we test a suite of applications with various memory access patterns. We leave a quantitative comparison between their model and ours to future work.

In his PhD thesis, Wang [53] presents an equation for calculating DRAM efficiency by accounting for the idle cycles in which the bus is not used. His model assumes that the request stream from the memory controller to DRAM is known while we also account for optimizations to the request stream made by the memory controller to help hide timing constraint delays. He also considers only single-threaded workloads, for which he observes high degrees of access locality. Our massively multi-threaded workloads have a wide range of access localities.
7.3 DRAM controller designs

There exist many analytical models proposed for a variety of microprocessors, from out-of-order execution superscalar to in-order execution fine-grained multi-threaded processors to even GPU architectures. Agarwal et al. also present an analytical cache model estimating cache miss rate when given cache parameters such as cache size, cache line size, associativity, etc., and their model also requires an address trace of the program being analyzed. Of these microprocessor analytical models, only Karkhanis and Smith, Chen and Aamodt, and Hong and Kim model long, albeit fixed, memory latency.

7.3 DRAM controller designs

There exist many DRAM scheduler designs proposed for multi-core systems. The primary focus of these designs revolve around the principles of providing Quality-of-Service (QoS) or fairness for different threads and cores competing for shared off-chip bandwidth. When multiple applications run simultaneously on a system, the memory access traffic of one application can cause a naive DRAM controller to starve out the requests of another. To the best of our knowledge, this is the first work that addresses the problem of efficient DRAM access scheduling in a massively multithreaded GPU architecture with tens of thousands of threads.

Figure 7.1: Average number of unique banks requested per warp memory operation

Mutlu et al. present a parallelism-aware batching scheduler (PAR-BS) that coordinates the servicing of multiple requests from a single thread, particularly those to different banks in a DRAM chip, to reduce the average memory latency experienced across all threads. We anticipate
the design of such a scheduler to support requests from tens of thousands of threads, which is possible in GPU architectures, to be highly complex and area-intensive. Furthermore, threads stall at the warp-level, such that all threads in a warp must be ready before the warp can be issued to the shader core pipeline. We measure the average number of unique banks to which a warp sends requests, shown in Figure 7.1. Five out of the nine applications always send requests to only a single bank per warp. In our hold-grant interconnect arbiter, the requests in a warp will be kept together as they traverse the interconnect and they will all arrive at the same bank, so batching is already done. In \textit{mum}, each warp sends requests to at least three different memory controllers per warp memory operation (since each DRAM chip has only 4 banks). For PAR-BS to perform well in this case, there must be coordinated scheduling across multiple memory controllers.
Chapter 8

Conclusions and Future Work

This chapter concludes the thesis and discusses some future work.

8.1 Conclusions

In this thesis, we detailed the changes made to the memory system of GPGPU-Sim, a detailed performance simulator, to support CUDA applications. Specifically, support was added for memory accesses to local, constant, and texture memories. We studied the performance of twelve contemporary CUDA applications by running them on GPGPU-Sim, first validating our simulator against real hardware and then studying the performance impact of varying the DRAM controller queue size and varying the shader core resources that limit the maximum number of CTAs that can concurrently run on a shader core. We observed that sometimes running fewer CTAs concurrently than the limit imposed by on-chip resources can improve performance by reducing contention in the memory system.

In this thesis we have also proposed a novel hybrid analytical DRAM model which takes into account the effects of Out-of-Order memory scheduling and hiding of DRAM timing delays. We showed that this model can be used with a sliding window profiling technique to predict the DRAM efficiency over the entire runtime of an application, given we have its full memory request address trace. We chose a massively multi-threaded architecture connected to a set of high-bandwidth GDDR3-SDRAM graphics memory as our simulation framework and evaluated the accuracy of our model on a set of real CUDA applications with diverse dynamic memory access patterns. We
8.1. Conclusions

introduce two different heuristics to use in conjunction with the sliding window profiling approach that serve as upper and lower bounds on the achieved bank-level parallelism. By averaging the predictions of these two contrasting heuristics, we were able to predict the DRAM efficiency of a set of real applications to within an arithmetic mean of 11.2% absolute error.

Finally, we introduced a novel, complexity-effective DRAM access scheduling solution that achieves system throughput within up to 91.0% of that achievable with aggressive out-of-order scheduling for a set of memory-limited applications. Our solution relies on modifications to the interconnect that relays memory requests from shader cores to DRAM controllers. These modifications leverage the key observation that the DRAM row buffer access locality of the memory request streams seen at the DRAM controller after they pass through the interconnect is much worse than that access locality of the individual memory request streams from the shader core into the interconnect. Three such modifications are possible: either holding grant for a router input port as long as there are pending requests to the same destination (HG), holding grant for a router input port as long as there are pending requests to the same destination and the requested row matches the requested row of the previous arbitrated request (RMHG), or holding grant for a router input port as long as there are pending requests to the same destination and the hash of the requested row matches the hash of the requested row of the previous arbitrated request (HMHG). These modifications work to preserve the inherent DRAM row buffer access locality of memory request streams from individual shader cores which would otherwise be destroyed due to the interleaving of memory request streams from multiple shader cores. In doing so, it allows for a simple in-order memory scheduler at the DRAM controller to achieve much higher performance than if the interconnect did not have such modifications across different network topologies and microarchitectural configurations.
8.2 Future work

This section discusses the new areas of research to where this thesis can be extended.

8.2.1 Improving the analytical DRAM model

Our hybrid analytical model relies on running applications a single time in performance simulation to collect memory request address traces. Relying on performance simulation for this process may be undesirable, reducing the usefulness of our model. We anticipate that this process can be sped up by instead running the applications in functional simulation and then performing some sort of simple interleaving to obtain a representative memory request address trace. To address the average error of our model compared to performance simulation, we showed that applications that are predicted poorly by one heuristic tend to be predicted much better with the other and that a simple combination of the two heuristics resulted in significant improvement in prediction accuracy. This implies the possibility for further improvement by combining the two heuristics in more intelligent ways.

8.2.2 Combining the analytical DRAM model with analytical processor models

To our knowledge, all previous work that propose analytical modeling of processors [10, 11, 22, 24, 30, 31, 35, 40] model DRAM latency as a constant fixed latency. As this thesis has shown, DRAM latency varies dramatically from one application to the next, especially if DRAM is heavily utilized. Intuitively, the accuracy of such analytical models for processors can be improved by incorporating a more comprehensive analytical DRAM model that can predict more accurate memory latencies. The first step would be extending the analytical DRAM model described in this thesis to predict memory latency.
8.2. Future work

8.2.3 Design space exploration of intelligent interconnects

Our proposed “hold-grant” interconnect arbitration modifications has been shown to preserve the row access locality of memory request streams of general purpose applications running on GPUs. However, the majority of GPUs has and will always be targeted to the gaming community. As such, the experiments in Section 6.3 should be tried on graphics applications as well. Moreover, our modifications should be tried on other interconnect allocation policies, such as wave-front and iSLIP allocation [12]. If the targeted interconnect architecture has multiple virtual channels to deal with deadlock, such as ring and torus architectures, more work must also be done to improve system performance when using our interconnect arbitration scheme.
Bibliography


Chapter 8. Bibliography


[16] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation


Chapter 8. Bibliography


[44] Hemant G. Rotithor, Randy B. Osborne, and Nagi Aboulenein. Method and Apparatus for
Chapter 8. Bibliography


