Design of a Hybrid Memory System for General-Purpose Graphics Processing Units

by

Patrick Carpenter

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
August 4, 2012

Keywords: GPU, Non-volatile memory, Phase-change memory, Hybrid simulator

Copyright 2012 by Patrick Carpenter

Approved by

Weikuan Yu, Chair, Assistant Professor of Computer Science and Software Engineering
Xiao Qin, Associate Professor of Computer Science and Software Engineering
Soo-Young Lee, Professor of Electrical and Computer Engineering

Abstract

Addressing a limited power budget is a prerequisite for maintaining the growth of computer system performance into and beyond the exascale. Two technologies with the potential to help solve this problem include general-purpose programming on graphics processors and fast non-volatile memories. Combining these technologies could yield devices capable of extreme-scale computation at lower power.

The goal of this project is to design a simulator supporting a hybrid memory system, containing both dynamic random-access memory (DRAM) and phase-change random-access memory (PCRAM), to replace the graphics global memory. Because of the proprietary nature of graphics hardware and the relative immaturity of phase-change memory, it is necessary to develop an appropriate simulation framework to conduct further research. In this work, GPGPU-Sim and a modified version of DRAMSim2 are combined in the design of a hybrid simulator named GPUHM-Sim. The design, implementation and validation of GPUHM-Sim are the primary contributions of this work.

Acknowledgments

This author would very much like to acknowledge the steadfast support and encouragement provided by his advisor and mentor, Dr. Weikuan Yu, as well as the expertise and enthusiasm of Drs. Xiao Qin and Soo-Young Lee as vital members of the graduate committee. Without the combined experience and insights of these individuals, to whom this author is greatly indebted, this work would not have been possible.

This work is thanks in large part to a collaboration with Dr. Dong Li of Oak Ridge National Laboratory and Dr. Xipeng Shen of the College of William and Mary. The author is very grateful for these collaborators' passion for excellence in research and unceasing diligence.

The author is similarly grateful for the guidance and assistance provided by those fellow students (in no particular order: Cristian Cira, Adarsh Jain, Xuechao Li, Zhuo Liu, Xinyu Que, Yuan Tian, Bin Wang, Yandong Wang, Cong Xu) with whom, during his graduate studies in the Parallel Architecture and System Laboratory, he had the distinct privilege and honor of working, and without whom the quality of this work would be significantly diminished. Moreover, the assistance and recognition provided to the Laboratory by its academic and industrial partners and sponsors (in no particular order: Auburn University, Department of Energy, HPC Advisory Council, Mellanox Technologies, National Aeronautics and Space Administration, National Science Foundation, NVIDIA Corporation, Oak Ridge National Laboratory, TeraGrid) has done much to make possible the continued research activities of the Laboratory and is deeply appreciated.

Finally, the author would like to thank the Department of Computer Science and Software Engineering and Auburn University for furnishing exciting opportunities, excellence in instruction and the environment in which this work was carried out.

Table of Contents

Abstract ii

Acknowledgments iii

List of Figures vi

List of Tables vii

1 Introduction 1

2 Background 9

2.1 General-Purpose Computation on Graphics Processing Units 10

2.1.1 The Rise of Graphics Processors for High Performance Computing . . 10

2.1.2 NVIDIA's Compute Unified Device Architecture 11

2.1.3 Advantages and Disadvantages 14

2.2 Non-Volatile Memory Technologies 17

2.2.1 Varieties of Fast Non-Volatile Memory Technology 17

2.2.2 Advantages and Disadvantages 19

2.3 The General-Purpose Graphics Processor and Non-Volatile Memory 21

3 Phase-Change Random Access Memory in a Hybrid Graphics Global Memory . 25

3.1 Proposal, Plan and Goals 25

3.2 Simulation Frameworks 26

3.2.1 Simulating Graphics Devices with GPGPU-Sim 27

3.2.2 Simulating Phase-Change Memory with DRAMSim2 28

3.3 Integration, Verification and Validation 29

3.3.1 Integration of Simulation Frameworks 29

3.3.2 Verification Methodology 32

3.3.3 Verification Results 36

4 Related and Future Work 45
4.1 Related Work 45
4.2 Conclusions and Future Work 46
Bibliography 48

List of Figures

2.1 Price-performance comparison of various memory technologies. 21

3.1 GPGPU-Sim architecture model. 28

3.2 High-level integration plan 30

3.3 Intermediate integration plan 30

3.4 Low-level integration plan 31

3.5 Global memory read bandwidth from SHOC for several devices 36

3.6 Global memory write bandwidth from SHOC for several devices 37

3.7 Peak single-precision performance from SHOC for several devices 37

3.8 Peak double-precision performance from SHOC for several devices 38

3.9 Instructions per cycle achieved versus computational intensity 39

3.10 Instructions per cycle from hybrid simulator versus computational intensity . . 41

3.11 Memory fetch latency from hybrid simulator versus computational intensity . . 42

3.12 Comparison of global memory read bandwidth for original and hybrid simulators 43

List of Tables

2.1 Characteristics of non-volatile memory technologies 21

3.1 SHOC v.1.1.1 Benchmark Descriptions 33

3.2 System configuration overview for SHOC benchmark experiments. 34

Chapter 1

Introduction

The demand for computer systems with ever-increasing levels of performance has driven, and continues to drive, innovation at the forefront of high performance computing. Engineers and scientists demand such systems in order to solve some of the world's most challenging and exciting problems—problems such as climate change [4], nuclear fusion, the design of life-saving pharmaceutical drugs, etc.—both at larger scales (e.g., climate models at global rather than continental scales) and at finer granularities (e.g., ab initio rather than parametric models to predict protein structure). Feasibly solving such problems at large scales and fine granularities expands the scope of questions which can meaningfully be addressed via computational techniques, accelerating the rate of scientific and engineering achievement—and all that goes with it. By pioneering innovations in the performance of computer systems and computer applications, all computing professionals—and, in the context of this work, the high performance computing community—not only advance their own science and technology, but catalyze advances of the entire technical ecosystem.

Over the years, innovations targeting performance have taken many forms and relied on diverse insights. For instance, improvements have been achieved from advances in the underlying technology (e.g., vacuum tubes versus transistors and the integrated circuit, solid-state drives versus hard disk drives, the historical shrinking of feature size with advances in the manufacturing process), in the hardware architecture (e.g., superscalar and pipelined processor architectures, multi-channel main memory), in hardware targeted at specific applications domains and computing paradigms (e.g., application-specific integrate circuits and field-programmable gate arrays, various parallel architectures reflecting designations in Flynn's taxonomy), and in software and algorithms themselves. Indeed, architectural and

system innovations can target and have targeted components at virtually every level (e.g., processors, peripherals, memory, disk, network). To date, the story of performance improvements in computing systems and the applications which they service is one of essentially unbridled success.

In fact, innovations have consistently resulted in explosive growth in raw power of computing systems. Since the early 1970s, Moore's Law has accurately reflected increasing transistor counts of computer microprocessors. For many years, these increases came in the form of increased instruction-level parallelism, which provided performance benefits to a very general class of computer applications, and relied on shrinking feature size and adjusting voltage as necessary [15]. However, this process began to lead—somewhat recently, in the fairly short history of modern computing—to an increase in power dissipation, to the point that the community has begun to move towards increased thread-level parallelism (via multi- and many-core technologies) in order to maintain exponential growth of computing capacity. Unlike instruction-level parallelism, thread-level parallelism does not provide performance benefits to traditional sequential applications; rather, applications must typically be designed in such a way as to exploit thread-level parallelism for performance benefit (it is worth noting that the distinction is not entirely clear, since applications can be written in such a way as not to benefit from instruction-level parallelism, and since several techniques—including framework- and compiler-aided techniques—exist in order to ease the development of applications to exploit thread-level parallelism, but in general the distinction between instruction-level and thread-level parallelisms is a useful one for understanding challenges particularly relevant to the high-performance computing community). Despite this shortcoming of thread-level parallelism, it has allowed for the development of truly powerful computing systems which benefit applications of actual import; indeed, Los Alamos National Laboratory's Roadrunner—which supported massive thread-level parallelism through 122,400 processor cores, including cores from both Opteron processors and Cell graphics

processors—was [26] the first computer system to break the petaflop barrier in 2008, performing over 10^{15} floating-point instructions per second.

As the high performance computing community strives to deliver an exaflop computer system—that is, one which can perform 10^{18} floating-point operations per second—it must overcome several obstacles, not least among which is a severely limited power budget. When Roadrunner was delivered in 2008, it required a total power of approximately 2.35 megawatts and achieved a sustained efficiency of around 437.43 million floating-point operations per second per watt, making it not only the fastest computer system in the world, but also the third most power-efficient [26] [25]. In order to build and operate an exaflop computing system, the high performance computing community must satisfy a variety of constraints, including the two which are of particular interest in the context of this work: performance and power. By a straightforward argument, the performance—in terms of floating-point operations per second—of an exaflop computer must be roughly three orders of magnitude larger than that of Roadrunner ca. 2008. On the other hand, realistic power budgets for future exaflop systems are generally held to be around 20 megawatts [13], around one order of magnitude larger than the power consumption of Roadrunner ca. 2008. What this means, in practical terms, is that a realistic exaflop computing system must boast a power efficiency of around two orders of magnitude higher than that of Roadrunner ca. 2008, or in the neighborhood of over $40,000$ million floating-point operations per second per watt. As of November 2011, the most power-efficient petascale computing system—located at the GSIC Center at the Tokyo Institute of Technology—achieved a maximum sustained power efficiency of around 958.35 million floating-point instructions per second per watt—slightly more than twice the efficiency of Roadrunner ca. 2008, and still more than around 40 times less than will be required in order to break the exaflop computing barrier. Novel solutions will be required to meet power and performance budgets.

Several avenues exist along which the high performance computing community has been pursuing solutions to the problem of reducing power consumption of the world's most powerful computer systems. At least three possibilities include increasing the flops per watt of computer processors (for instance, by using special-purpose accelerators such as graphics processing units or, more generally, adopting designs which employ massive thread- level parallelism on microprocessors of simpler design), making fundamental changes to the memory of computer systems in order to promote power savings (such as the use of 3D stacked memory or emerging fast non-volatile memory technologies to reduce dynamic power consumption), and improving software which can reduce power from under-utilized computer hardware (e.g., techniques such as fine-grained dynamic voltage and frequency scaling might be successfully applied to many hardware components in future exascale systems). To satisfy the constraint on total power consumption, it is likely that a combination of such technologies will be useful.

Of these technologies, two have emerged which have enjoyed significant attention in recent years, which promise to aid in the pursuit of the exascale, and which form the basis of this work: the use of graphics processing units for general purpose computation, and fast non-volatile memory technologies—in particular, phase-change random access memory—as replacements for the volatile memory technologies which are today commonplace. The former is thanks to a combination of several factors: the evolution of the graphics processor into a device excelling at computations involving high computational intensity as well as massive thread- and data-parallelism, initial interest, culminating in the release by NVIDIA of its Compute Unified Device Architecture (CUDA) in 2007, in the use of graphics devices for applications in scientific computing, and the impressive power efficiency of these devices, which can help compute nodes reach power efficiencies which are greater than those of traditional compute nodes by a factor of 4 or more, to name only a few. Meanwhile, the latter offers a non-volatile alternative to traditional volatile memory technologies such as dynamic

and static random-access memory, while providing orders of magnitude of performance improvement over other non-volatile memory technologies (e.g., flash and hard disk). The non-volatility of these emerging technologies refers to the fact that power is not required to maintain the integrity of data they store (whereas dynamic random access memory must be periodically refreshed due to leakage), meaning that the proper use of these technologies can be used to reduce total power consumption of the memory system by addressing static power (that is, power which is consumed when the memory system is not being used to read or write data). Of course, one of the key benefits of these technologies is the non-volatility itself; however, this aspect is not treated in this work. In summary, both of these technologies appear promising in terms of leading to the exascale, which leads to the fundamental idea of this project.

By carefully combining these technologies so as to bolster their strengths and diminish their weaknesses, it is hoped that a hybrid system—one capable of massive computational capacity at lower power—can be developed. Several questions must be answered in order to design an effective and efficient hybrid system of this kind: What kind of non-volatile memory ought to be used? How can memory controllers effectively and efficiently manage data placement, access, and migration? More generally, how can proposed architectures and designs be compared quantitatively, in the absence of actual hardware implementations? The design and implementation of a suitable evaluation framework for hybrid graphics global memory is of the utmost importance, and as detailed later, is the main focus and contribution of this work.

Many factors must be taken into consideration in order to properly analyze and understand the performance of programs executing on graphics devices; these include relevant hardware and architectural details, execution configurations of the graphics programs, as well as inherent and emergent computational aspects of the program itself. Understanding hardware and architectural influences is complicated by at least two considerations: first, although NVIDIA hardware and its associated CUDA have, to date, been the predominant driver of

the use of graphics processors for general-purpose computation, challengers ATI (now owned by AMD) and OpenCL (among others) are gaining momentum; second, NVIDIA's hardware and low-level driver interfaces are entirely proprietary, and the physical instruction-set architecture varies from device to device (in practice, these issues do not represent huge obstacles to understanding device performance characteristics, effectively and efficiently leveraging NVIDIA hardware, or to enjoying easy application portability across hardware devices, but they are worth mentioning). The performance characteristics of fundamental algorithms and data structures is equally as important for computing on graphics devices as it is for computing on traditional processors; it is worth noting that these characteristics may need to be reinterpreted and re-evaluated in light of changed assumptions regarding hardware properties. Such complications can render infeasible first-principles analyses of specific scenarios involving graphics devices and the applications they service. In many cases, other methods are appropriate.

A useful way to gain some understanding of performance behavior of computer hardware, including graphics hardware, is to empirically evaluate the performance of many programs with differing computational qualities on one or more hardware systems with differing performance characteristics. Benchmarking—that is, evaluating the performance of standard applications while varying the hardware system on which these applications are executed—is a tried and tested technique. Several benchmark suites exist for evaluating the performance of systems using graphics devices, including the Scalable HeterOgeneous Computing (SHOC) suite [8] developed at Oak Ridge National Laboratory, Rodinia, Parboil, etc. As part of this work, experiments are conducted to evaluate the performance of the SHOC benchmark suite. Performance data are collected and used in order to evaluate the performance of the proposed simulation testbed.

The goal of this project is to design a hybrid simulation framework, called the Graphics Processing Unit Hybrid Memory Simulator (GPUHM-Sim), to enable flexible and reconfigurable simulation of hybrid global memory systems inside graphics devices. For a variety

of reasons (including the proprietary nature of graphics hardware, the relative immaturity of phase-change memory technology, etc.), it has been convenient to adopt the use of appropriate simulation frameworks, which have been integrated to produce GPUHM-Sim: GPGPU-Sim [3] for the graphics device, and a modified version of DRAMSim2 [23] for the phase-change memory. Each of these simulators represents a complex software artifact the functioning of which must be at least minimally understood to be of use in evaluating competing experimental designs. First, GPGPU-Sim and DRAMSim2 are carefully analyzed in order to understand the functioning of these two systems. Then, a detailed integration plan is formulated which seeks to create an extensible interface between these two simulators which will preserve the flow of information. The design emphasizes strict separation of interface and implementation, and supports hybridization at several levels of the memory request pipeline (across memory partition units, or within a device itself via changes to the external memory simulation framework). An initial integration of these simulators is then carried out and significant effort is spent in an attempt to evaluate the quality of the integration in terms of the resiliency, performance and output quality of the hybrid simulator. Performance results are collected from simulations using the original version of GPGPU-Sim as well as the initial version of GPUHM-Sim, and results are compared. Results from these initial efforts have been good, with GPUHM-Sim deviating from GPGPU-Sim by at most a few percent for similar memory configurations. The integration of these two simulation frameworks, constituting the design of GPUHM-Sim, and the validation and verification of the resulting hybrid simulator, as well as the definition of a novel architectural simulation framework for hybrid graphics global memory inside graphics processing units, are the primary contributions of this work.

The rest of this thesis is arranged as described in the remainder of this paragraph. Chapter 2 expands on the information presented in the introduction to provide additional background, context, and motivation for the project which is the subject of this work. Furthermore, it is the hope of this author that this chapter can serve as a useful starting point,

or guide, for readers wishing to somewhat better acquaint themselves with the topics of modern high performance computing, general-purpose computation on graphics processing units, or fundamental notions of emerging non-volatile random access memory technologies. Chapter 3 contains a complete report of the goals, work, and results of the project which is the main subject of this work. First, the project motivation, goals and scope are restated in a clear and concise fashion. Then, the simulators GPGPU-Sim and DRAMSim2 are described and analyzed. Finally, the integration plan is presented in detail, and the results of the initial integration are described and evaluated. Chapter 4 consists of a review of more-or-less closely related work, conclusions that can be drawn from the work and the project, and an outline of future goals and directions.

Chapter 2

Background

This chapter expands on the information presented in Chapter 1 in order to provide additional background, context, and motivation for the project which is the subject of this work. Furthermore, it is the hope of this author that this chapter can serve as a useful starting point, or guide, for readers wishing to somewhat better acquaint themselves with the topics of modern high performance computing, general-purpose computation on graphics processing units, or fundamental notions of emerging non-volatile random access memory technologies. Interested readers are encouraged to consult referenced sources directly.

Section 2.1 introduces the graphics processor as general-purpose computational device, and recounts some of the key developments leading to its current widespread adoption in the high performance computing community. Specifically, parallelism and how it applies to graphics devices, an overview of NVIDIA's CUDA, advantages and disadvantages of graphics devices for general-purpose computation, and some representative graphics devices and examples of systems which use them, are discussed.

Section 2.2 attempts to provide a minimal introduction to the landscape of fast non-volatile memory technologies which are of primary interest to this work, and to compare these to slower non-volatile as well as volatile devices. The section concludes with a brief review of some actual products which are beginning to enter the market or which are slated for increased production in the near future.

Section 2.3 examines the graphics memory hierarchy and candidate non-volatile memory technologies and attempts to provide an adequate answer to the following question, which was posed in Chapter 1 as a question of fundamental importance in beginning to carry

out the intended project of research: which graphics storage target(s), if any, are well-suited to integration with non-volatile memory technologies, and which non-volatile memory technology(ies) are appropriate for with graphics storage target(s)? By evaluating hardware characteristics and their relation to power and performance, a tentative answer is provided which establishes a clear direction for the remaining work.

2.1 General-Purpose Computation on Graphics Processing Units

This section provides background information related to the use of graphics processing units for general-purpose computations. In particular, issues related to programmability and performance are described. These and other issues inform decisions related to the design GPUHM-Sim.

2.1.1 The Rise of Graphics Processors for High Performance Computing

Over the last 20 to 30 years, graphics processors have evolved from fixed-function, special-purpose computational devices, into devices programmable through a restricted graphics shader application programming interface, and recently into fully programmable computational coprocessors. [21] Because of their humble origins as devices used for graphics rendering, graphics processors have developed with a different set of performance characteristics to those of traditional processors, corresponding to the differences in engineering requirements. By way of comparison, graphics processors are very high-throughput, high-latency devices. This is due to the fact that graphics cards were designed primarily for graphics, and humans are more sensitive to lower throughput (e.g., lower screen resolutions or smaller monitors) than they are to longer latency (e.g., viewing graphics many milliseconds after the GPU began processing the scene). For instance, modern NVIDIA Fermi GPUs can achieve over one teraflops of single-precision floating-point performance and over 100 gigabytes per second of sustained on-device memory bandwidth. Additionally, graphics processors dedicate much more hardware real estate to arithmetic-logic and floating-point units than to control

or cache units, meaning that they excel at numerically-intensive applications but not at ones which exhibit complicated control logic. Again by way of comparison to traditional processors, graphics hardware exhibits incredible amounts of thread- and memory-level parallelism. For instance, modern NVIDIA Fermi graphics cards contain in the neighborhood of 450 processor cores, many of which can simultaneously access a word of global memory over a wide memory bus. However, this hardware performance comes at the cost of programmability; achieving good performance on graphics processors takes effort and expertise. In many respects, graphics devices evolved into systems not dissimilar from those constituting an object of study in the fields of high performance and scientific computing.

2.1.2 NVIDIA's Compute Unified Device Architecture

NVIDIA's CUDA programming model and toolkit, released in late 2007, allow developers to write general-purpose programs to be executed, in part, on NVIDIA hardware [17] [16]. CUDA represents a significant step forward in terms of programmability and has led to drastically increased adoption of general-purpose computing on graphics devices by programmers in several areas of application. It provides a minimal set of extensions to the standard C++ programming language, in addition to several application programming interfaces, for efficiently mapping general-purpose computation to their graphics devices. More specifically, CUDA allows programmers explicit access to the hierarchies of processors and memories on GPUs through similarly hierarchical software abstractions. Within a traditional C++ program, application programmers define a special function, called a kernel, to be executed by the GPU. Then, from within main (host) code, the application programmer may copy data from host memory to (graphics) device global memory and back again, and in addition, invoke, or launch, the kernel, which triggers execution on the GPU over the system I/O bus. Memory transfers and kernel executions happen (or can be made to happen through appropriate flags to appropriate functions) asynchronously to execution of host code, allowing for increased performance via overlapping of communication and computation.

It is at this point useful to introduce, or revisit, terminology related to the hardware architecture of NVIDIA's graphics devices which will be employed in explaining details of NVIDIA's CUDA. NVIDIA Fermi devices are organized hierarchically in terms of both processing and memory models. At the highest level, the GPU represents the entire computing device, and contains a device-global memory area referred to as global memory. Modern graphics devices typically include on the order of gigabytes of such memory, which provides high-throughput, high-latency, device-wide read/write access. At the next level, groups of processor cores, referred to as streaming multiprocessors, of which NVIDIA Fermi devices typically have around 15, contain multiprocessor-local shared memory, which is treated as an explicitly software-managed cache. The size of shared memory varies, but is typically around 64 kilobytes per multiprocessor; regarding performance, shared memory is typically much lower-latency than global memory. At the lowest level, processor cores, also referred to as streaming processors, of which NVIDIA Fermi devices have 32 per multiprocessor, can access large numbers of processor-local registers, which represent the fastest available memory on GPUs.

Having briefly outlined some of the key architectural features of representative CUDA-enabled hardware devices, the focus returns to CUDA. When a kernel is launched on a device, many threads (the number and arrangement of which are specified as part of the kernels invocation) are created and distributed to hardware scheduling units corresponding to the streaming multiprocessors. The set of all threads created in response to a kernel launch is referred to as a grid. Grids are further subdivided into thread blocks, such that all the threads belonging to a given thread block are scheduled on the same multiprocessor. Groups of threads within blocks referred to as warps are scheduled in a single-instruction, multiple-thread manner across the cores belonging to a multiprocessor. Using data provided automatically by the CUDA framework, threads may access their position in the grid (which thread block they belong to, as well as their position within their thread block) and issue instructions accordingly. Note that for the threads of a warp to make efficient use of

hardware, they must abide by the hardware limitation already noted and execute the same instructions; however, they may operate on independent data elements.

The introduction of CUDA 4.0 between February and April 2011 gives a strong indication of the direction in which the community can expect graphics technology to be heading. NVIDIA [18] cites three main areas of improvement in CUDA 4.0: usability, multi-device programming capabilities, and developer tools. Each of these areas bears discussion in terms of its impact on general purpose programming on graphics processing units.

Usability and programmability: For the purposes of this discussion, the most important innovations in this category include how host threads share graphics devices and no-copy pinning of system memory. With CUDA 4.0, multiple host threads can share a single graphics device, and a single host thread can use multiple graphics devices; this represents a significant paradigm shift in terms of application structuring to leverage varying graphics resources. No-copy pinning of system memory improves further upon the benefits of pinned host memory.

Multi-device support: Unified Virtual Addressing (UVA) and GPUDirect 2.0 bring improvements in both programmability (UVA brings the host/device domain closer to a truly global address space) and in performance (by reducing graphics devices' dependence on the host to perform communication and I/O). The importance of these advances to increased adoption of general purpose programming on graphics devices should not be understated.

Developer tools: With the release of CUDA 4.0, NVIDIA has added multi-device support to the cuda-gdb debugger. In summary, we see a trend towards increased use of and reliance on the use GPUs in high-performance scientific computing applications.

The most current version of CUDA—CUDA 4.1 [19]—was released in late 2011, and offers improvements in application development features (notably, an LLVM-based compiler which results in faster code for many applications), library support (e.g., over a thousand new image processing functions were added to the NVIDIA Performance Primitives library), and developer tools (more features in cuda-gdb and cuda-memcheck for debugging). Although representing a modest incremental increase in the overall framework, these changes—as well

as those made for the release of CUDA 4.0—are useful in evaluating the current and near-future needs of the community, as perceived by NVIDIA: more efficient and effective application development through more advanced language features and generally better tool and development and environment support.

2.1.3 Advantages and Disadvantages

Several advantages, as well as important disadvantages, are associated with the use of graphics processing units for accelerating applications in high-performance scientific computing. Some of the most important advantages motivating their use have already been mentioned; briefly, they include the following: high peak floating point operation rates; high peak energy- and cost- efficiency; and large peak memory bandwidth. Together, these advantages make graphics devices potentially very attractive to solve the problems of extreme-scale computing.

The chief disadvantages associated with the use of graphics devices in the aforementioned capacity are related to the difficulty associated with achieving near-peak performance for real-world applications. Broadly speaking, graphics devices suffer from three fundamental limiters of graphics performance: the I/O bottleneck across the PCIe bus; the single-instruction, multiple-thread model of parallelism; and the sensitivity of various memory areas to access patterns. Each of these limiters deserves some discussion.

Applications which employ graphics accelerators to perform computations must take into account the cost of transferring data to and from the graphics device. Currently, graphics devices are connected to the host system via the PCI-e bus, which provides relatively low-bandwidth, high-latency communication between the host and available devices. Typically, this problem can be overcome using a variety of techniques, including, but not necessarily limited to, the following: ensuring that computations to be performed on the device are such that the benefit from using the device outweighs the cost of communication; overlapping computation on the host with computation on the device (some recent interesting work in this

direction can be found in [5]); overlapping computation on the device with communication to and/or from the device (a feature introduced in recent versions of CUDA). Other efforts seek to eliminate the the potential for bottleneck entirely by integrating traditional processors and graphics processors; this is the basis for the notion of the arithmetic processing unit [9].

The single-instruction, multiple-thread model of parallelism required by graphics devices presents other challenges to achieving excellent performance for on-device computations. As described earlier in this section, threads belonging to a block which has been scheduled to a streaming multiprocessor are grouped into warps, which are then scheduled to the multiprocessor's array of streaming processors. These streaming processors share a common instruction cache and operate in lockstep. Therefore, to process an instruction from each thread in a warp in a single cycle, all threads must be executing the same instruction (i.e., have a common instruction pointer). A performance problem arises when threads within a warp exhibit divergent behavior; that is, when threads within a warp take different control-flow paths through the kernel function, the hardware is incapable of processing threads in lockstep. CUDA guarantees correctness of kernels exhibiting divergence by effectively splitting divergent warps into multiple warps containing dummy threads, which essentially represent no-op instructions (the actual mechanism underlying the treatment of this situation is more complex, but unimportant for the purposes of this discussion). Control-flow divergence can easily result in performance degradation of a factor of two (for if/else type divergence) and could result in much more if nested branching is present. Fortunately, the hardware is able to rejoin split warps eventually (at any post-dominator, usually the immediate post-dominator) in order to avoid excessive loss of performance. Warp divergence can be addressed at the application level by writing code which does not exhibit much divergence; at the hardware architecture level, warp divergence can be mitigated by techniques which exploit the existence of many warps processing the same instruction stream [11] [10]. At the compiler level, kernels can be statically analyzed and instructions reordered to reduce divergence [7]. Also, interesting research has investigated the benefits to be had by exploiting

inter-warp divergence (as opposed to intra-warp divergence, to which the above discussion applies), referred to as warp specialization [2].

Several memory areas are available inside graphics processors, each of which offers unique opportunities and challenges. At the lowest level in the memory hierarchy, each streaming multiprocessor contains a register file providing thread-local, low-latency access for data. However, the number of registers is limited, and so in practice not all data can be stored there. At the next level, streaming multiprocessor-local shared memory serves as an explicitly-managed cache, and provides low-latency accesses, when accessed according to an appropriate pattern. Words in shared memory are stored in a number of banks, each of which can be accessed simultaneously and with low latency. However, when multiple requests for words in shared memory correspond to the same shared memory bank—referred to as a shared memory bank collision—requests are serialized. The number of simultaneous requests for distinct words of shared memory is referred to as the degree of the shared memory bank conflict; performance of the shared memory system is inversely proportional to the maximum degree of any bank conflict occurring during a round of simultaneous access requests (note, however, that in the case of all requests accessing the same word of shared memory, no performance penalty is incurred). Furthermore, the amount of shared memory available to threads scheduled onto a given multiprocessor is limited, necessitating other options.

At the next level in the graphics memory hierarchy, there exist global memory, constant memory, and texture memory areas (note that each of these is realized by the same physical hardware, and differs from the others only in how it is treated by the architecture). Constant memory provides read-only access (on the device; it is written by the host) to all threads and all blocks. It is cached on each streaming multiprocessor, and on cache hits, provides low-latency access when all simultaneous requests are for the same word of constant memory; other access patterns result in degrees of serialization similar to shared memory. Texture memory, like constant memory, provides read-only access to all threads and is cached on streaming multiprocessors. However, texture fetches can provide low-latency access for

rounds of simultaneous requests exhibiting spatial locality; in other words, for rounds of requests accessing words of texture memory which are ”near” each other according to the texture geometry. Global memory provides both read and write access to all threads and all blocks, and (along with constant and texture memories) represents the memory area through which host applications may communicate data to and from the device. Global memory is not cached (typically; for more details on the capabilities of modern graphics devices, see [17] [16]) and provides high-latency access. Several important ways exist in which long latencies of global memory accesses are addressed by CUDA developers. First and foremost, by ensuring that memory accesses are coalesced (the requirements for coalescing vary with the graphics hardware, but coalescing can be guaranteed by having threads access linearly contiguous and aligned chunks of global memory; see [16] for more information), high levels of memory throughput can be achieved. In order to avoid performance degradation from inevitably long latencies, it is important to have both a large number of warps ready to execute instructions and a large enough number of instructions not accessing global areas of memory. Together, these ensure that processor cores have enough work to do to effectively mask memory access latency.

2.2 Non-Volatile Memory Technologies

This section provides a brief introduction to the fast non-volatile memory technologies which are candidates for integration with the graphics processor main memory. These memory technologies vary in terms of power, performance, and access characteristics, and as such may be more or less suitable for use in a variety of contexts. By covering characteristics of various alternatives, the decision made for this project can be given strong support.

2.2.1 Varieties of Fast Non-Volatile Memory Technology

Fast non-volatile modern technologies, which have received some attention in the recent research literature, are different from both other non-volatile memory technologies such as

flash and volatile memory technologies such as dynamic and static varieties random access memories. They differ from the former in that they generally offer faster, lower-density storage; compared to the latter, they can provide a variety of benefits to real systems, not least of which is persistent storage of critical data. These storage technologies have been, and continue to be, enabled by innovative hardware designs. This section introduces the reader to three of these technologies: spin-transfer torque random access memory, a variety of magnetoresistive memory; random access memory based on memristors, a recently developed circuit element complementary to the resistor, capacitor, and inductor; and phase-change random access memory.

The memristor is a recently introduced basic circuit element—like the resistor, capacitor, and inductor—capable of maintaining internal state, making them suitable for storing data, among other things [6]. Each of the four basic circuit elements can be defined in terms of relationships among fundamental circuit variables, which include current, voltage, charge, and flux. The resistor relates current and voltage, according to Ohm's law; the capacitor relates charge and voltage; and the inductor relates flux and current. Note also that charge is related to current, and flux to voltage, by definition. The memristor relates the flux and charge; together with the other relationships, it completes the understanding of all relationships between fundamental circuit variables. By relating charge and flux, memristors are able to act as variable resistance elements, whose resistance is based on currents to which they have been subjected; that is, they maintain measurable state based on applied signals.

Spin-transfer torque memory is a kind of magnetoresistive memory and differs from conventional dynamic memory technologies in that it stores information using magnetic tunnel junctions instead of electrical charges, which are comprised of three layers: one reference ferromagnetic layer, one free ferromagnetic layer, and a tunnel barrier layer [28]. While the magnetic direction of the reference layer remains fixed, the magnetic direction of the free layer can be changed; if the directions of the free and reference layers align, resistance is

high (indicating logically set, or one), and if the directions are different, resistance is low (indicating logically reset, or zero).

Phase-change memory is based on a unique property of a certain kind of alloy (chalogenide alloys) with two stable states—crystalline and amorphous—to store information [28]. The low-resistance crystalline state can be achieved by heating such an alloy to a temperature between crystallization and melting temperatures, and a bit of information whose alloy is in this state is interpreted as logically set, or equal to one. The high-resistance amorphous state can be achieved by heating the alloy to a temperature above the melting temperature, after which the temperature is quickly reduced; a bit of information whose alloy is in this state is interpreted as logically reset, or equal to zero. Because the difference in electrical resistance between these two states is so large, it is possible to use intermediate states to encode several bits of information instead of one; devices exploiting this are referred to as multiple level devices, others being referred to as single level cell.

2.2.2 Advantages and Disadvantages

All non-volatile memory technologies share a common property which makes them desirable, compared to volatile memory technologies such as static and dynamic random access memory technologies: the ability to store data in a persistent fashion without needing to be periodically refreshed. Devices with this property have at least two advantages: first, they can be used to protect data from interruptions in the supply of power, to avoid loss of important data; second, they can use less power than their volatile counterparts for storing data which are accessed according to appropriate patterns (appropriateness of access patterns for such technologies is discussed later, and represents a key consideration in much of the ongoing work), since their is no need to refresh stored data. It is the second of these advantages which the greater project, of which the work described in this thesis represents only one (albeit significant) aspect, seeks to exploit. It is worth noting, however, that the first advantage mentioned above—the suitability of non-volatile memory technologies for

use as a resilient or fast persistent data store—may also prove useful in achieving present and future goals of high-performance computing; indeed, such an advantage might even be exploited by graphics devices, although this possibility is not explored in this work.

Where non-volatile memories pay the price for these advantages, however, is in performance (in terms of both access latency and dynamic power), or, more accurately, in price-performance (that is, performance compared to cost, which can be discussed in terms of monetary cost or memory density). In other words, non-volatile memory technologies typically suffer from lower levels of performance compared to volatile counterparts of similar cost. Figure 2.1, courtesy Motoyuki Ooishi, compares write latency and memory capacity of several memory technologies, both volatile and non-volatile. Memory technologies with higher write latencies and memory capacities are typically more well-suited to applications requiring large amounts of persistent storage (e.g., disk), whereas technologies with lower write latencies and memory capacities are better suited to applications requiring small amounts of fast storage (e.g., caches).

Traditional non-volatile memory devices such as NOR and NAND flash require that cells be erased before being written, which significantly increases the latencies associated with writing to these devices; these latencies, along with large memory capacities, make them good candidates for applications requiring persistent storage. Of the memory technologies which are more suitable for working memory, magnetic random access memory technologies offer a memory capacity comparable to that of static random access memory; this implies that they are, perhaps, better suited to situations where static random access memory proves useful (e.g., in caches—indeed, a recent research effort in this direction, which is closely aligned with the goals of the project motivated here, is touched upon later). Phase-change random access memory has a memory capacity, hence a cost, more comparable to that of dynamic random access memory; this implies that it might be used in much the same way as dynamic random access memory, if the order of magnitude write latency barrier can be overcome.

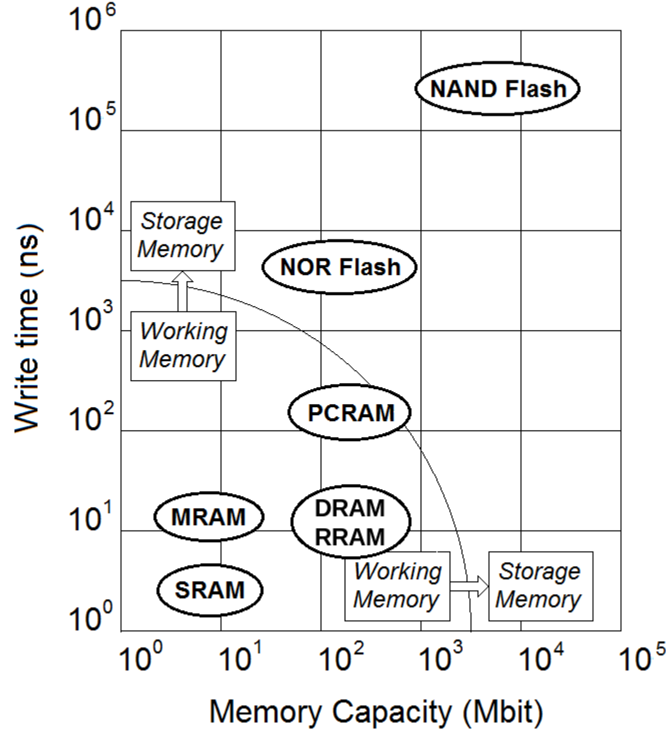

Figure 2.1: Price-performance comparison of various memory technologies.

Table 2.1 provides an incomplete and, in parts, partially outdated summary of memory technology characteristics, and is based—in part—on data compiled by Perez and De Rose [22].

-	**PCRAM**	**RRAM**	**MRAM**
Cost	$8 - 16F^2$	$> 5F^2$	$37F^2$
Read	48 ns	$<$ 10 ns	$<$ 10 ns
Write	$40 - 150$ ns	10 ns	12.5 ns
Energy	100 pJ	2 pJ	0.02 pJ
Endurance	10^8	10^5	$> 10^{15}$

Table 2.1: Characteristics of non-volatile memory technologies

2.3 The General-Purpose Graphics Processor and Non-Volatile Memory

In this chapter, two technologies with significant potential to be of great benefit to efforts in the high-performance computing community—the use of graphics processing units

to accelerate general-purpose computations of scientific interest, and the use of fast non-volatile memory technologies to improve either aspects of traditional memory hierarchies—have been briefly introduced. In this section, the choice of phase-change memory as the non-volatile technology, and of the graphics global memory as the storage target, is motivated, and other directions which might have been selected are mentioned. In particular, available benefits—possible drawbacks—of the project are defined.

As has already been discussed, non-volatile memory technologies differ in terms of both access latency and storage density, as well as in many other respects. Depending on such characteristics, a particular technology may be more-or-less well-suited to various applications (e.g., cache, main memory, permanent storage). There are at least two graphics memory areas—the multiprocessor-local shared memory, and the global memory, including global storage areas for constant and texture memory—where non-volatile memory might be fruitfully introduced, and these areas differ fundamentally in terms of their performance properties. These properties must inform the selection of the non-volatile memory technology.

As an explicitly managed cache, shared memory offers potentially low-latency access for a relatively small amount of on-chip storage. As such, non-volatile memory technologies such as magnetoresistive (e.g., spin-transfer torque) memories and ferroelectric memories—which offer relatively lower latencies and storage densities—might make for appropriate replacements for the static random access memory currently employed for that purpose. Indeed, that possibility [24] has already received investigation, and is reviewed later.

As a sort of graphics device main memory area, global memory uses as its underlying technology GDDR dynamic random access memory, which is similar in most respects to DDR dynamic random access memory, and offers a relatively large amount of high-latency access. As such, non-volatile memory technologies such as phase-change random access memories—which offer somewhat longer latencies and storage densities similar to those of dynamic random access memory—might make for appropriate replacements of the dynamic random

access memory currently employed for that purpose. It is this possibility that provides the basis for the collaborative project, in support of which the primary contributions described in this thesis are made and later described.

Thus far, only the possibility of using phase-change memory in place of dynamic random-access memory in the graphics global memory area has been advanced as a motivation for its use; however, advantages and, potentially, disadvantages are associated with its use, and understanding these is key to fully understandin the design decisions directly affecting the work which is the primary focus of this thesis. Among the benefits available from using phase-change memory in the proposed way include lower static power and the opportunity to exploit non-volatility for permanent or semi-permanent storage of data. The former is due to the lack of any need to refresh memory contents, owing to to the latter. At the present time, power savings alone (rather than the usefulness, per se, of having non-volatile storage within graphics devices) has been the primary motivator underlying development of a hybrid simulator; an example of exploiting non-volatility for graphics devices [14] is reviewed in a later chapter.

Although memristor-based memories are being actively developed and researched, at the present time available technologies are too immature—especially in terms of manufacturing processes and chip yield—to warrant further consideration here; their relatively lower write endurance, compared to the alternatives, is another factor weighing against their use in the intended capacity. However, storage devices based on memristors should be expected to play an important and exciting role in the future of memory technologies.

A few downsides to using phase-change memory ,which make using this technology within the graphics global memory area an interesting research challenge, are worth mentioning. First off, while phase-change memory has a similar read latency and storage density as compared to dynamic random-access memory, its write latency is several times higher. Similarly, while static power consumption is much lower for phase-change memory, power

consumed during writes is typically much higher (although there appear to be design trade-offs involved in producing phase-change memory devices such that lower write energies could be had from decreasing the device's non-volatility [29], this is not a possibility given further consideration here). Overcoming higher active power consumption and access latencies for writes involves developing effective data placement and migration strategies; additionally, increased latencies from writes can be addressed by the usual mechanism inside graphics devices: high bandwidth from massive memory parallelism and regular access patterns (i.e., coalesced global memory accesses) along with sufficient amounts of computation to mask latency. A hybrid global memory system, using both dynamic and phase-change random access memories, could use phase-change memory for mostly read-friendly memory objects, avoiding these disadvantages.

Chapter 3

Phase-Change Random Access Memory in a Hybrid Graphics Global Memory

This chapter contains a complete report of the goals, work, and results of the project which is the main subject of this work. First, the project motivation, goals and scope are restated in a clear and consise fashion. Then, the simulators GPGPU-Sim and DRAMSim2 are described and analyzed. Finally, the integration plan is presented in detail, and the results of the initial integration are described and evaluated.

Section 3.1 attempts to more clearly define this project in terms of both planned work and desired outcomes, and to situate this work in terms of other ongoing work which is being carried out as part of the larger collaboration (of which this project represents only one aspect).

Section 3.2 provides background on the two simulation frameworks used in carrying out this work, GPGPU-Sim and DRAMSim2, and exposes those features of each simulator which are most relevant in the context of this project.

Section 3.3 reports on the subsequent analysis, design and initial implementation of a hybrid simulator supporting further research goals, and the efforts spent in order to assess and improve quality and accuracy of the result.

3.1 Proposal, Plan and Goals

This project seeks to design an architectural simulator for hybrid global memory inside graphics processing units. Such a framework is needed to carry out research involving the use of phase-change random-access memory to reduce power consumption of the graphics global memory system. This should be possible thanks to non-volatile memory's not requiring static power to retain data. However, at least two limitations of phase-change random

access memory must be overcome in order that it be viable as a part of the graphics global memory system: a large energy requirement for writes, as well as high latencies for writes [27]. Overcoming these limitations, however, is not the focus of the work described in this thesis.

Given the proprietary nature of NVIDIA graphics hardware, the relative immaturity of change memory technology, and the difficulty inherent in constructing an actual hybrid device and associated memory controller logic, direct experimentation is deemed inappropriate. On the other hand, detailed, first-principles analysis could prove unwieldy for so complex a system as the one being proposed.

Simulation is, however, appealing, given the existence of simulation frameworks for both graphics devices and phase-change memory devices. Since the architectural changes being investigated in this work are new, currently no simulation framework exists, insofar as this author is aware, suitable for simulating a general-purpose graphics device with a customized, hybrid global-memory system. The initial efforts which have been aimed at developing such a capability are the main contribution of this work and are described in the following sections.

3.2 Simulation Frameworks

In this section, the two simulation frameworks on which the described hybrid simulator is based are introduced. Given that much of the work involved in carrying out this initial integration relies on some (indeed, in some aspects, significant) understanding of the purpose and functioning of these constituents, the material presented here is of fundamental importance in the following sections.

3.2.1 Simulating Graphics Devices with GPGPU-Sim

GPGPU-Sim [3] is a cycle-accurate simulation framework for graphics devices and the general-purpose compute kernels which they execute. At a high level, GPGPU-Sim is organized into three modules—cuda-sim, gpgpu-sim and intersim—which perform unique roles necessary to the simulation of real CUDA programs on NVIDIA hardware.

In GPGPU-Sim, cuda-sim is responsible for functional simulation of CUDA programs; that is, its purpose is to ensure that the execution of real CUDA programs behaves in a manner which is logically consistent with that in which it should be expected to behave on correctly-functioning NVIDIA hardware. Despite its being a vital component of the GPGPU-Sim architecture, it has been necessary, for the purposes of this work, to make alterations to this module; indeed, it is not within the scope of this project—nor, perhaps, should it ever be—to change the logical behavior of CUDA programs executing on devices with hybrid memory (i.e., causing the result to be different from that obtained using a device without a hybrid global memory system). As such, this module does not receive a great deal of attention in this work.

The gpgpu-sim and intersim modules are responsible for the timing simulation of CUDA programs in GPGPU-Sim; in other words, they use models of NVIDIA hardware to estimate device performance for the simulated kernel. As illustrated in 3.1 [1] , gpgpu-sim models performance of processor, chips and global memory, while intersim is used to model the complex interconnection network connecting processors and off-chip global memory. Since the motivation for performing this study is to evaluate tradeoffs in power and performance, these modules are of great interest. In particular, it is useful for the purposes of discussion in section 3.3 to describe in some detail the memory request lifecycle modeled by GPGPU-Sim.

At a high level, shader cores interpret CUDA instructions, some of which require global memory accesses to be performed. Shader cores generate memory fetch requests accordingly, forwarding these to the interconnection network. From there, memory fetch requests are routed to the appropriate memory partition unit, which consists of a memory controller, a

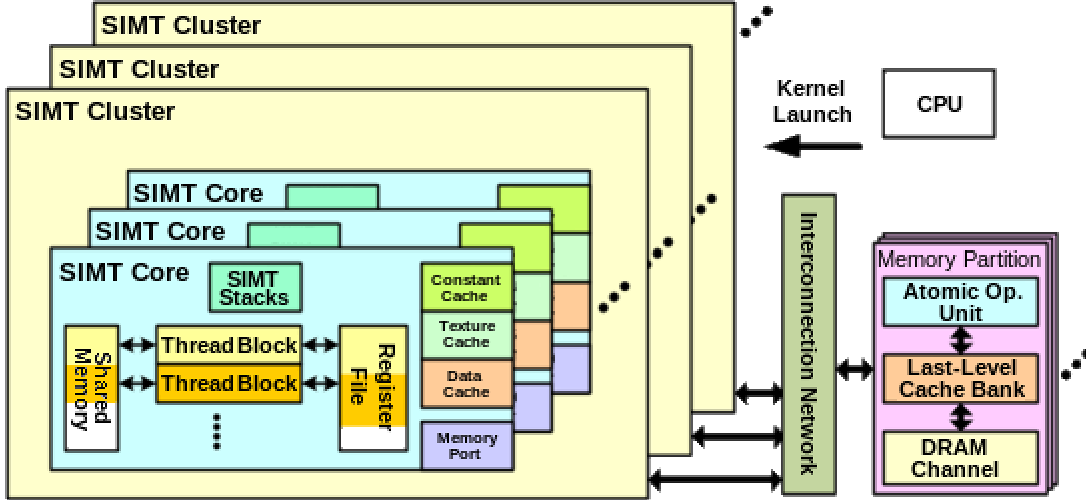

Figure 3.1: GPGPU-Sim architecture model.

dedicated L2 cache for texture fetches, and a memory device. If a request is a texture fetch, then the L2 cache is checked before scheduling the request to the memory device. Memory fetches which must access the device incur some additional latency, which is modeled by GPGPU-Sim. When a request has been satisfied, notification is pushed back up the memory system, first to the interconnection network from the memory partition unit, and ultimately to the shader cores.

3.2.2 Simulating Phase-Change Memory with DRAMSim2

DRAMSim2 [23] is a cycle-accurate memory simulator, developed at the University of Maryland, for modeling DDR2/3 systems. DRAMSim2 can be used either as a standalone simulator for trace-based modeling of memory systems, or—through a library interface—as the memory model in other simulation frameworks. It is in this latter capacity that DRAMSim2 is used in this work.

Use of the DRAMSim2 library interface consists of instantiating an instance of the MemorySystem class, and subsequently, adding memory transactions and cycling the simulator. Cycling of the simulator causes pending transactions to advance through the DRAMSim2 simulation model to completion. Upon completion of a transaction, DRAMSim2 invokes a

callback routine, provided upon initialization of the MemorySystem object, alerting the host application to the completion of the request.

The reason for which it was deemed necessary to incorporate DRAMSim2 into GPGPU-Sim is the insufficiency of the GPGPU-Sim memory model to correctly model either power of the memory system or performance characteristics of phase-change memory technology. The choice of DRAMSim2 was motivated by collaborators' experience using this memory simulation framework, and the ability of DRAMSim2 to model phase-change (and potentially other non-volatile) memory power and performance characteristics.

3.3 Integration, Verification and Validation

This section describes the design and initial implementation of a hybrid simulator enabling research into the use of novel memory technologies inside graphics processing units, as well as some of the early efforts to ensure and assess quality.

3.3.1 Integration of Simulation Frameworks

Figure 3.2 shows the general strategy for integration at a high level of abstraction (more specifically, at the level of the gpgpu-sim performance simulator). The idea is a very simple one: to use the DRAMSim2 memory simulator—via its library interface—instead of the memory simulation included in the gpgpu-sim performance simulator. Issues arising at this level of abstraction include the nature of the replacement (i.e., should DRAMSim2 be used exclusively, or should it only be used for phase-change memory), as well as the general compatibility between GPGPU-Sim and DRAMSim2, among others.

Figure 3.3 shows the integration plan at the level of the memory request flow inside GPGPU-Sim. Here, DRAMSim2's library interface is observed to provide similar functionality to the component responsible for modeling a memory controller and controlled device

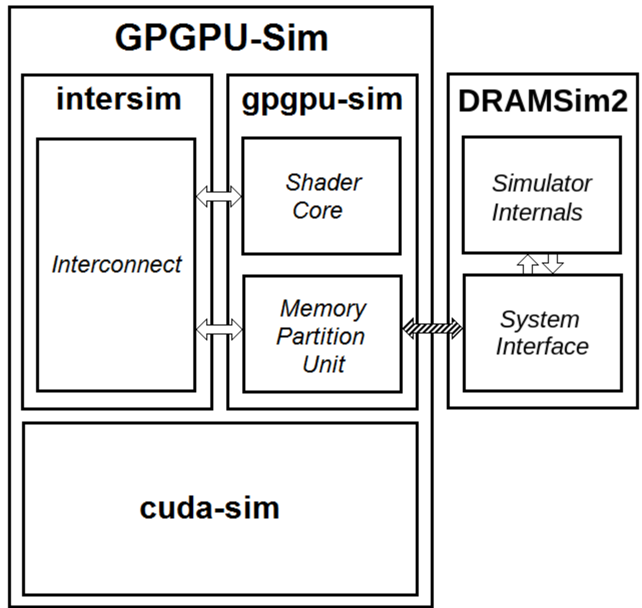

Figure 3.2: High-level integration plan

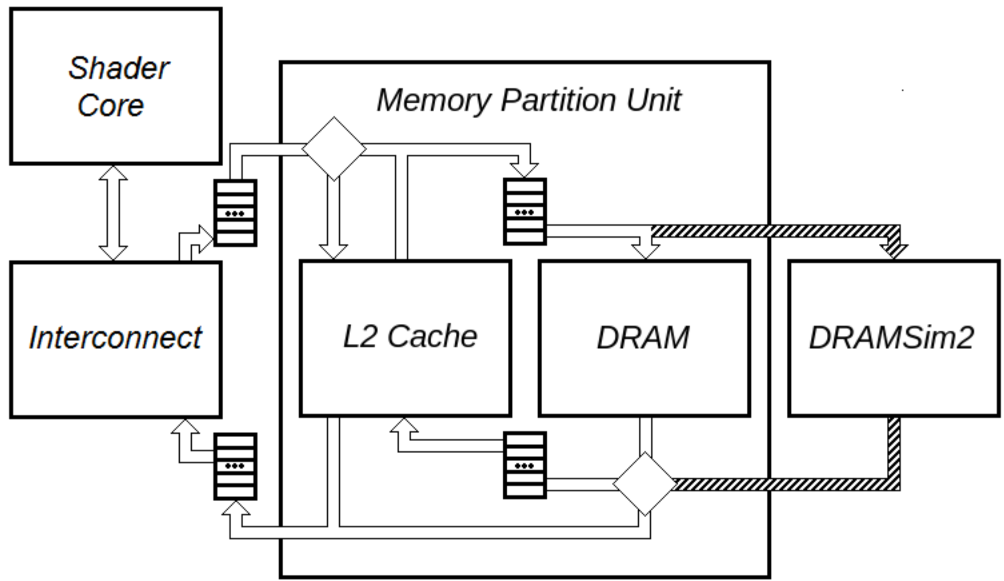

Figure 3.3: Intermediate integration plan

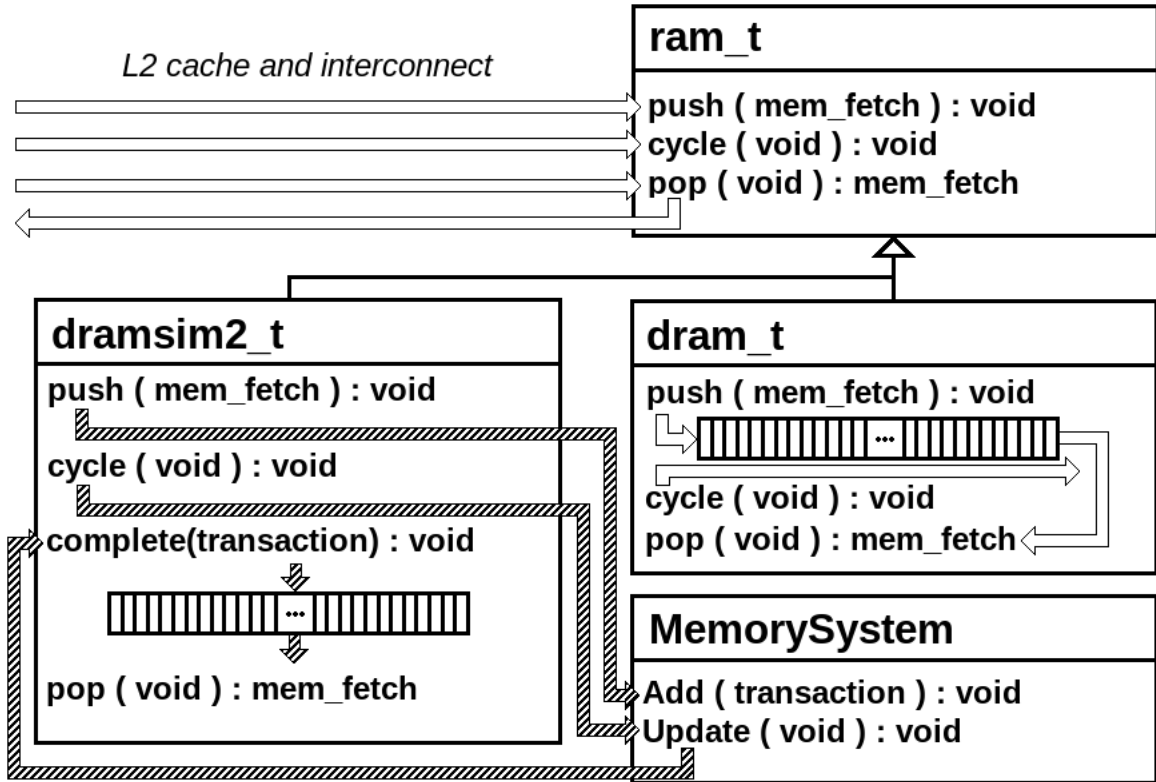

Figure 3.4: Low-level integration plan

within the gpgpu-sim performance simulation model. Given this, the strategy is to use multiple instances of DRAMSim2's library interface (one per memory controller and associated device), while leaving other stages of the memory request processing unaffected.

Figure 3.4 shows the integration plan at the level of the memory controller and device. As mentioned previously, the DRAMSim2 library interface provides an interface quite similar to that of the component in gpgpu-sim which is responsible for modeling a memory controller and controlled device; within the original gpgpu-sim module, this component is implemented as a class named dram_t. In order to provide gpgpu-sim a means of using DRAMSim2 to model the memory controller and associated device, the interface common to both DRAMSim2's library and the dram_t class is used to define an abstract parent class, named ram_t, which replaces dram_t in the integrated simulation framework. To provide access to DRAMSim2, a new child class, named dramsim2_t, is implemented in such a way as to forward memory requests coming from the interconnect or L2 cache to DRAMSim2,

and forward requests completed by DRAMSim2 to the interconnect or L2 cache. In other words, the dramsim2_t class is responsible for initializing an instance of the DRAMSim2 library interface, and acting as an intermediary between the gpgpu-sim memory model and this created instance of the DRAMSim2 library.

3.3.2 Verification Methodology

In order to perform a baseline evaluation of the correctness of the initial implementation of the design described in the previous subsection, several experiments were performed and results compared. First, empirical data was collected using the Scalalable HeterOgeneous Computing (SHOC) benchmark suite [8] using locally available graphics hardware. These provide some context with which to evaluate later results. Then, results of a proof-of-concept investigation performed using the original version of GPGPU-Sim with varying underlying memory latencies are used to illustrate the relationship between instructions per cycle, arithmetic intensity, and memory latency. Finally, a comparison is performed between the results output by the original GPGPU-Sim and those output by the hybrid GPGPU-Sim/DRAMSim2 framework.

The SHOC benchmark suite, developed at Oak Ridge National Laboratory [20], is comprised of a variety of programs with differing computational properties which exercise various areas of graphics hardware performance. Benchmarks in the suite are characterized according to the degree to which they are representative of synthetic microbenchmarks on the one hand (corresponding to SHOC level 0 benchmarks), or production applications on the other (corresponding to SHOC level 2 benchmarks). Table 3.1 describes the benchmarks which constitute version 1.1.1 of the SHOC suite.

Benchmark	**Level**	**Description**
BusSpeedDownload	0	Measures the bandwidth of the PCIe bus for transfers from the host to the device.
BusSpeedReadback	0	Measures the bandwidth of the PCIe bus for transfers from the device to the host.
DeviceMemory	0	Measures read and write bandwidths of device global, texture and shared memories.
MaxFlops	0	Measures the maximum floating-point computation rate of the device, for both single- and double-precision computations.
FFT	1	Measures performance of a 1D Fast Fourier Transform kernel.
MD	1	Measures performance of a pairwise nbody kernel from a Lennard-Jones potential calculation application.
Reduction	1	Measures performance for a large global sum reduction kernel.
Scan	1	Measures performance of a parallel prefix sum kernel.
SGEMM	1	Measures performance for device versions of the SGEMM BLAS routine on square matrices.
Sort	1	Measures performance of a radix sort kernel over unsigned integer key-value pairs.
Spmv	1	Measures performance of a kernel which computes products of sparse matrices with dense vectors.
Stencil2D	1	Measures performance for a 2D, 9-point stencil kernel.
Triad	1	Measures bandwidth for a large vector dot product kernel.
S3S	2	Measures bandwidth for S3D's getrates kernel.

Table 3.1: SHOC v.1.1.1 Benchmark Descriptions

Furthermore, benchmarks can be executed in one of three configurations—Serial, Embarrassingly Parallel and True Parallel—which differ in terms of how many graphics processors and compute nodes are evaluated. All results in this section are for the Serial configuration, and as such, provide information most relevant at the level of a single graphics device within a single compute node.

The SHOC benchmark suite has been used to evaluate performance of several graphics devices. Table 3.2 presents some of the relevant information related to the configuration of systems on which experiments were conducted.

The Heterogeneous Auburn Working Cluster (hawc) was constructed in Fall 2011, with generous hardware donations by NVIDIA and with Laboratory funds and equipment, to support research and teaching objectives related to the awarding to Auburn of the status of

GPU	Node	OS	CPU	RAM
GTX 280	hawc1	64-bit CentOS 6.0	Intel Q8200	8GB DDR2
GTX 470	hawc1	64-bit CentOS 6.0	Intel Q8200	8GB DDR2
GTX 480	hawc4	64-bit CentOS 6.0	Intel i7-950	8GB DDR3
Tesla M2050	eagles-0-0	64-bit CentOS 5.5	Intel Xeon 5650	24GB DDR3
Tesla C2070 hawc4	64-bit CentOS	Intel i7-950	8GB DDR3	

Table 3.2: System configuration overview for SHOC benchmark experiments.

NVIDIA CUDA teaching center. It consists of four compute nodes—hawc1, hawc2, hawc3, and hawc4—publicly accessible via the head node at hawc.cse.eng.auburn.edu. A 64-bit version of CentOS 6 has been installed on each node, as have basic parallel development packages and tools to facilitate parallel programming using OpenMP, MPI and NVIDIA's CUDA version 4.0. Each node is equipped with a multi-core processor—an Intel Q8200 on hawc1 and Intel i7-950s on each of nodes hawc2, hawc3 and hawc4—in addition to two graphics devices—currently, two GTX 470s on hawc1, two GTX 480s on each of hawc2 and hawc3, as well as a GTX 480 and a Tesla C2070 on hawc4—and eight gigabytes of main memory. These nodes are connected by a gigabit ethernet connection through hawc.cse.eng.auburn.edu.

In order to gain some rough understanding—in addition to that which comes from knowledge and analysis of graphics architecture—of the relationship between instructions per cycle, memory latency and arithmetic intensity, an experiment was performed using GPGPU-Sim to simulate a CUDA kernel of tunable computational intensity.

The kernel designed for this evaluation allows for varying arithmetic intensity by computing, with 2^{20} threads linearly arranged in 2^{10} thread blocks (hence, with 2^{10} threads per block), an iterative map of 2^{20} single-precision floating point numbers for numbers of iterations $[1, 2, 4, 8, 16, 32, 64]$. The number of global memory accesses per thread is fixed at two, and accesses are performed in a completely coalesced manner. Computational intensities corresponding to each of the numbers of iterations are estimated, based on a simple static

analysis, to be proportional to $[6, 9, 15, 27, 51, 99, 195]$, where computational intensity is defined as the number of arithmetic operations over the number of global memory reads and writes (fixed at two for this kernel).

To simulate varying memory latencies using GPGPU-Sim, initialization parameters for the default GPGPU-Sim memory system were changed to scale underlying memory latency parameters (specifically, tCCD, tRRD, tRCD, tRAS, tRP, tRC, CL, WL, tCDLR, and tWR) according to the desired increase in global memory latency; in other words, to increase the end-to-end memory latencies by a factor of n, each of these parameters was scaled by n. This simplification was motivated by the assumed approximate linearity of end-to-end latency of memory systems as a function of underlying device latencies. Experiments used v.3.0.1 of the GPGPU-Sim tool, and aside from the indicated changes to the initialization configuration, the initialization used was identical to the initialization files provided as part of the GPGPU-Sim distribution for the Quadro FX 5800 graphics processor.

Finally, simulation experiments were carried out using the SHOC benchmark suite to evaluate the performance of the hybrid simulator compared to the original simulator. In particular, results for two benchmarks—the MaxFlops and DeviceMemory level-zero benchmarks—are reported. The selection of these benchmarks was motivated by the fact that these benchmarks have clear behavior and performance characteristics, and differences in performance variation can be used to extract meaningful information. Specifically, variations in the DeviceMemory benchmarks should indicate sensitivity to memory system characteristics and variations in the MaxFlops benchmarks should be indicative of potential impacts on performance of compute-bound kernels. Reported results are for the implementations found in the Serial suite of CUDA benchmarks belonging to SHOC v.1.1.1. The initialization configuration files for GPGPU-Sim were those for the Fermi configuration, provided as part of the GPGPU-Sim v3.0.2 standard distribution.

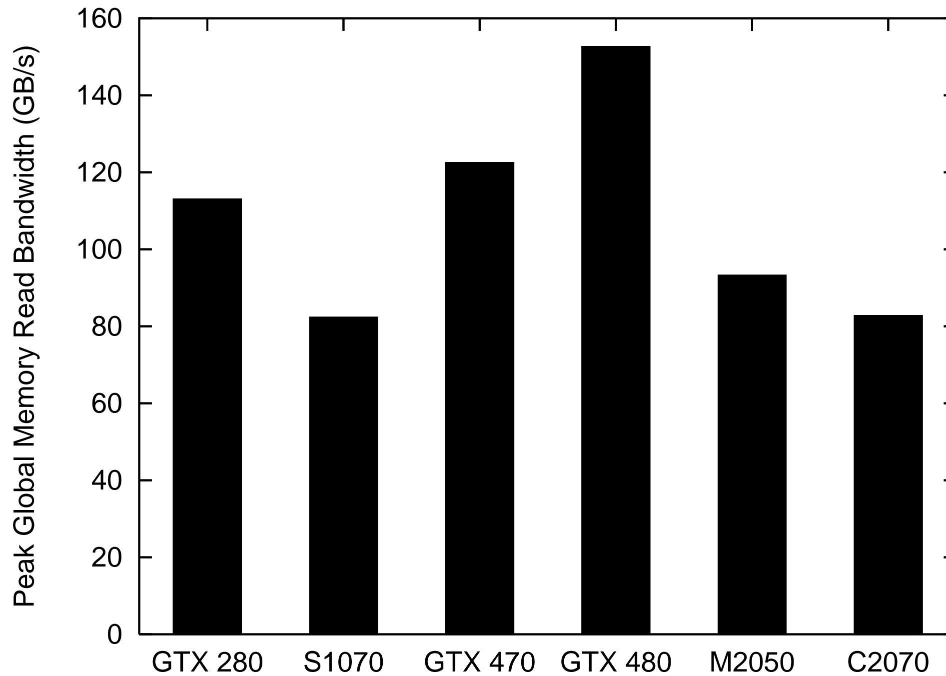

Figure 3.5: Global memory read bandwidth from SHOC for several devices

3.3.3 Verification Results

Figures 3.5, 3.6, 3.7 and 3.8 show the global memory read and write bandwidth and single- and double-precision peak processing rates, respectively, for several graphics processors belonging to the systems previously discussed. These figures provide a baseline against which simulation results are assessed.

At least two interesting trends can be observed from the global memory peak bandwidth results: first, the relatively older graphics processors (the GTX 280 and S1070) have comparably higher global memory read bandwidths than global memory write bandwidths, whereas the opposite is true for relatively newer devices (the GTX 470, GTX 480, M2050a nd C2070); second, the devices designed exclusively for general-purpose computation (the M2050 and C2070) compare less favorably in terms of both peak read and peak write bandwidth to CUDA-enabled graphics devices designed to accommodate graphics applications' performance demands.

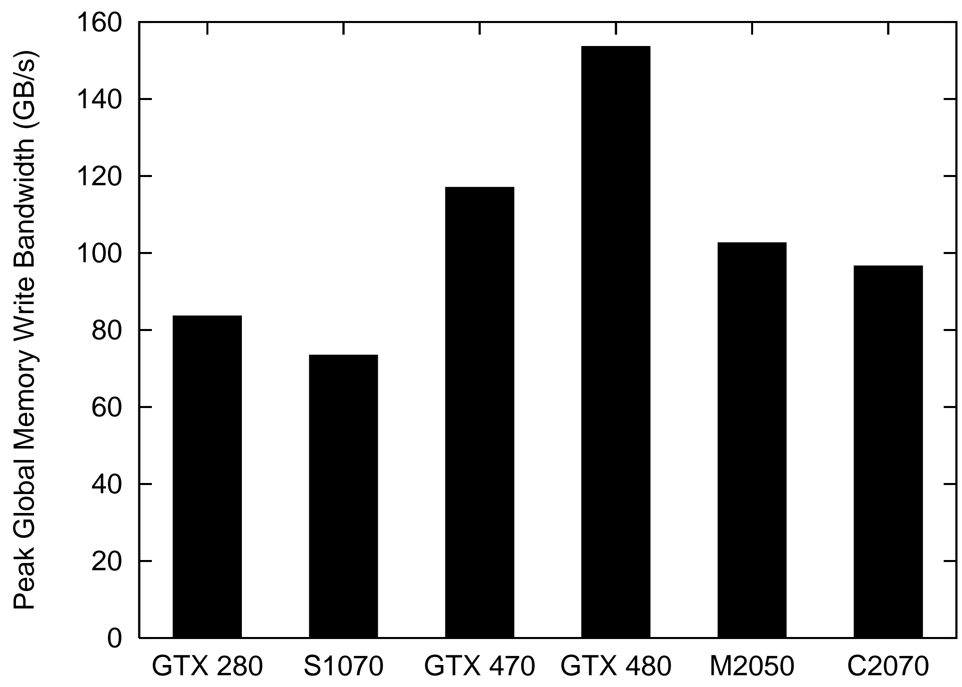

Figure 3.6: Global memory write bandwidth from SHOC for several devices

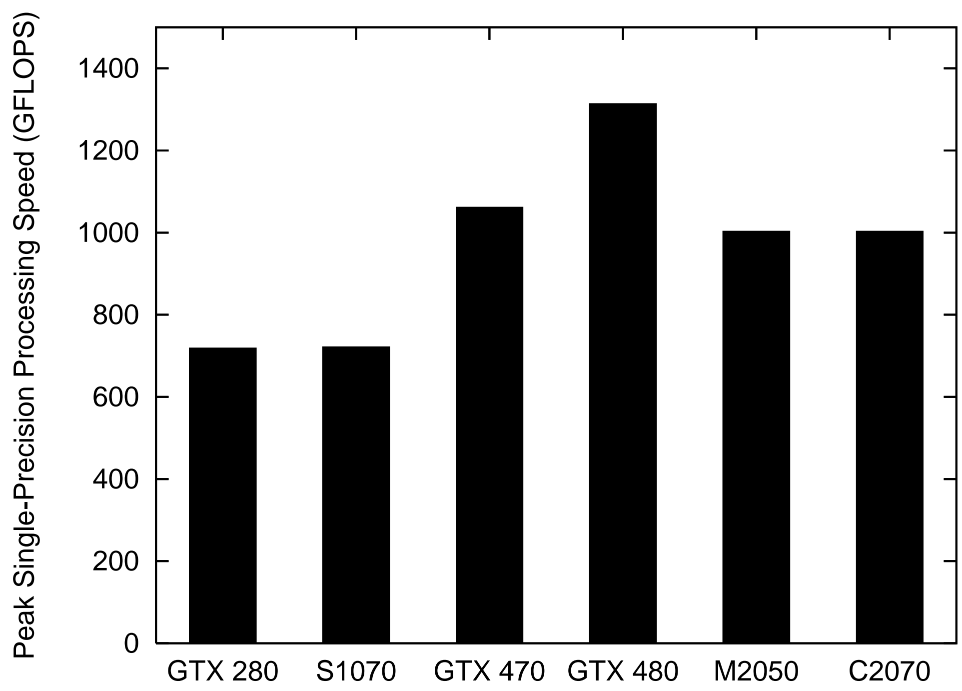

Figure 3.7: Peak single-precision performance from SHOC for several devices

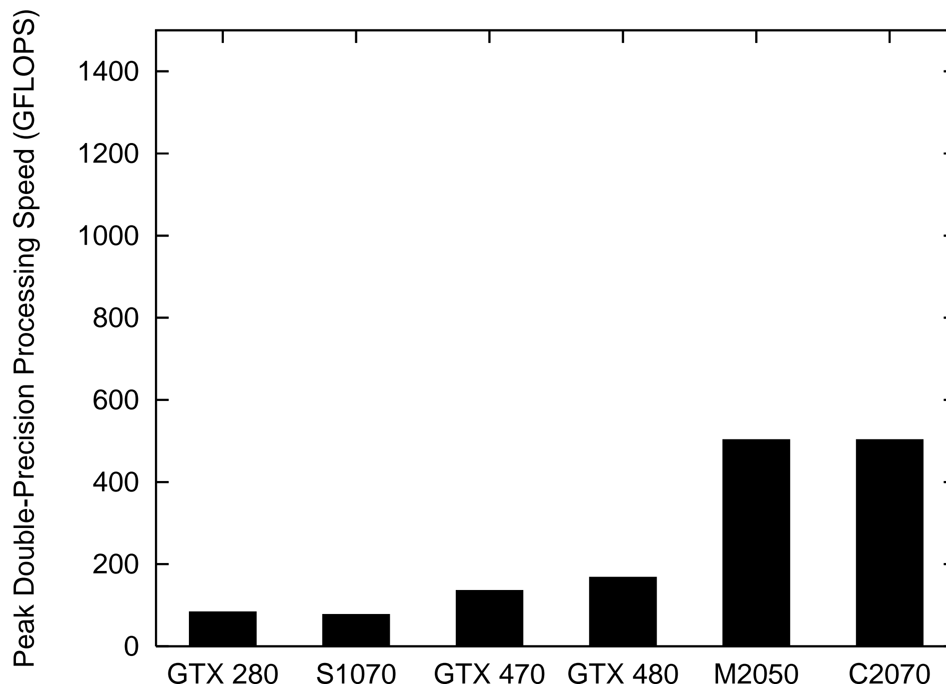

Figure 3.8: Peak double-precision performance from SHOC for several devices

Moreover, it is useful to identify at least two trends in the results for peak single- and double-precision floating point processing rates: first, single-precision processing is faster by a factor of at least two for all tested devices; second, the factor by which devices' peak single-precision processing rates exceed those devices' peak double-precision processing rates is a function of whether the device was designed specifically for general-purpose (scientific) computing (the M2050 and the C2070), in which case the factor is approximately two, or not (the GTX 280, S1070, GTX470 and GTX 480), in which case the factor is approximately ten. The first observation is owed to the fact that, at the hardware level, any device capable of performing a double-precision computation in a certain time should be capable of performing two single-precision computations (since double-precision arithmetic units are typically comprised of at least two single-precision units). The second observation underscores one of the major design differences between older-generation devices from NVIDIA and new Fermi

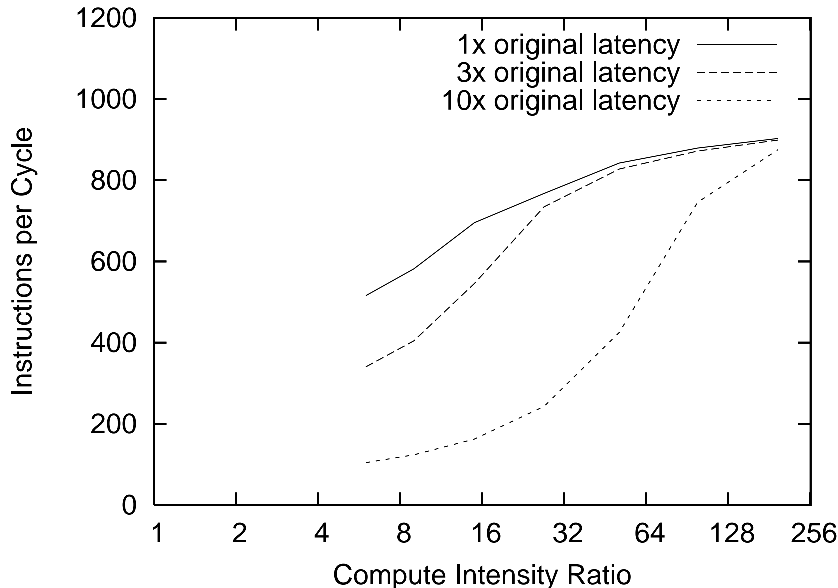

Figure 3.9: Instructions per cycle achieved versus computational intensity

devices: tailor-made devices for general-purpose (scientific) computing often rely on higher-precision computations than are typically required for traditional graphics applications, and as such, hardware support for fast double-precision arithmetic is a valuable feature.

Figure 3.9 shows the results from the proof-of-concept simulation work described previously. The performance in terms of the instructions per cycle is evaluated for three different scenarios: the original GPGPU-Sim characteristic device latencies, denoted by the trend labeled 1x; a configuration corresponding to device access latencies increased by approximately one-half an order of magnitude (a factor of 3), denoted by the trend labeled 3x; and a configuration corresponding to device access latencies increased by approximately one order of magnitude (a factor of 10), denoted by the trend labeled 10x. The required changes were made to the gpgpusim.config initialization file; see [1] for details.

At low computational intensities, the impact of uniformly longer access latencies is significant; increasing latencies by a factor of ten causes a factor of five decrease in the achieved instructions per cycle. As computational intensity increases, processing efficiency increases for two very closely-related reasons: first, a greater fraction of the time spent on the kernel is spent on computation, so that the effects of a constant number of memory accesses are

diminished by comparison; second, the effect on total time required of increasing computational load depends on the amount of communication (i.e., memory access latency) which can be hidden: at lower computational intensities, more latency is available to be hidden, and additional computation will overlap more effectively. For relatively (and modestly, compared to some real-world application kernels, such as those present in the SHOC suite) large intensities, the memory access latencies become much less of a factor, as the performance in each of the three cases asymptotically tends towards the device maximum processing rate (the maximum processing rate of the Quadro FX 5800, the configuration file for which was used to generate these trends, is 960 instructions per cycle).

These results have at least two implications for future research work aimed at performance tuning of hybrid memory system designs: first, that performance may be expected to degrade, for realistic graphics devices and computational kernels, by no more than about five percent, in the worst case and if phase-change memory completely replaces dynamic random access memory in the hybrid design (this represents a worst case since these experiments deal with changes to performance while increasing both read and write access latencies; however, only the write latency, and not the read latency, of phase-change memory is high compared to dynamic random access memory); second, that for application kernels with modest to intense computational requirements, the performance penalty, in the worst case, of using phase-change memory is reduced and, asymptotically, becomes insignificant. This second point is especially promising, since it provides a clear point of reference to which candidate application kernels can be compared, and could prove useful both in the development of the hybrid memory controller and in determining the applicability of the proposed architecture to real-world applications.

Figure 3.10 shows the results of executing the variable-intensity kernel used above on the original and hybrid graphics simulators. Note that the hybrid simulator utilizes a version of DRAMSim2 configured to simulate dynamic (not phase-change) random-access memory (the initialization is the default one and is comparable to GPGPU-Sim's default configuration for

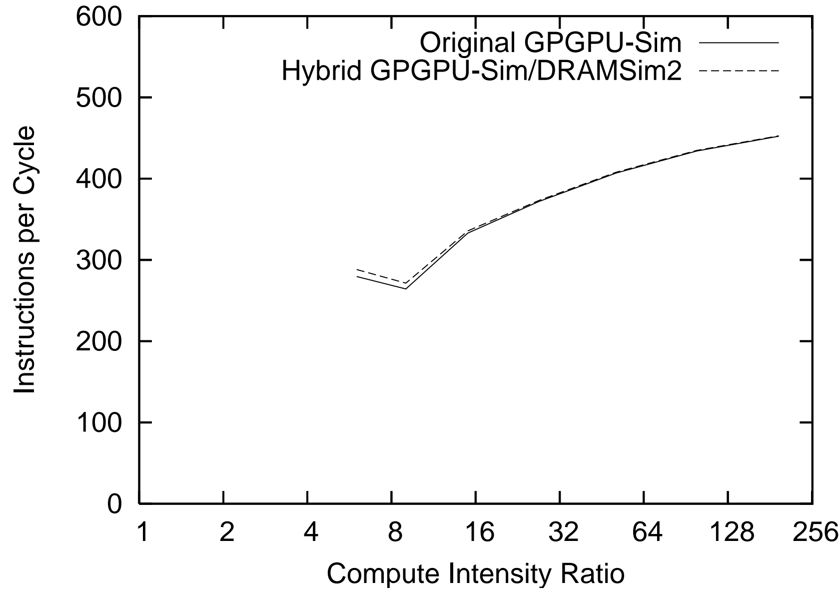

Figure 3.10: Instructions per cycle from hybrid simulator versus computational intensity

global memory devices); this has been done in order to allow for a meaningful comparison between the output of the hybrid and original simulators.

As can be seen in the figure, the performance of the hybrid simulator closely matches (to within a maximum percent error of three percent, at the lowest computational intensity) the output of the original version of GPGPU-Sim. Given the startlingly distinct nature of the simulation frameworks underlying the two simulators, this close correspondence is taken as a very positive sign that the initial implementation is correct in its essential details. Note also that these measurements closely correspond to measurements of single-precision floating-point processing rates for the GTX 480, which has both the same number of cores and core frequency as the device represented by the supplied configuration file (480 cores at 700 Mhz, with dual single-precision issue, yields around 1340 GFLOPS).

Despite the close correspondence between hybrid and original simulator results, it has been more satisfying to this author to attempt an explanation of the existing discrepancy. Figure 3.11 better shows the difference in memory-device-level performance which can be used to help explain this discrepancy.

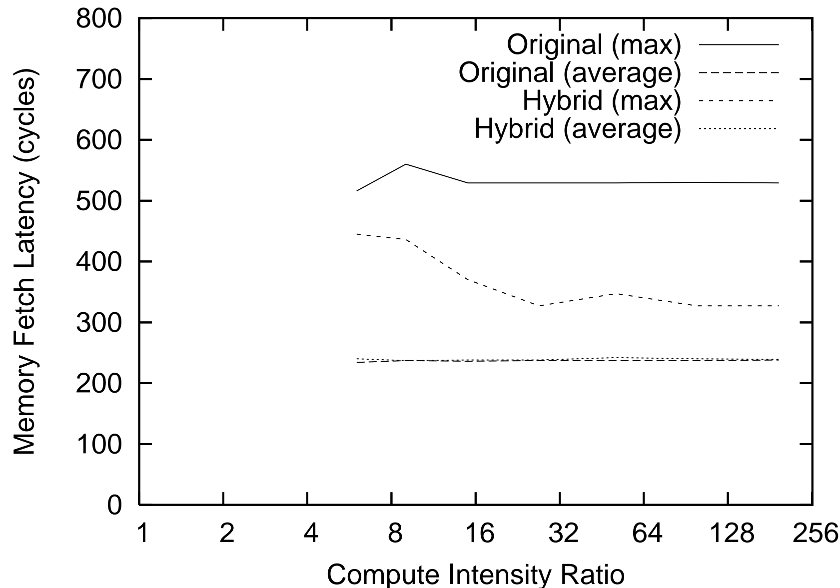

Figure 3.11: Memory fetch latency from hybrid simulator versus computational intensity

Although, on the average, both simulators require essentially the same number of cycles in order to complete a memory fetch request, the maximum memory fetch latency recorded by the original simulator is considerably larger than the maximum memory fetch latency recorded by the hybrid simulator. This could indicate a comparatively greater variability in the original memory simulator than is found in DRAMSim2. Greater variability in memory fetch request latency could easily translate into longer running times when parallelism is involved. To see that this is a possibility, consider the following two scenarios. Let X be the random variable for the following experiment: a fair die is rolled, and the result multiplied by two. Let Y be the random variable for the following experiment: a fair die is rolled twice, and the results added together. By elementary probability, $E[X] = 7$ and $Var[X] = 35/3 \sim 11.67$; similarly, $E[Y] = 7$ and $Var[Y] = 35/6 \sim 5.83$. Suppose that two parallel processors must each process a single task, and that the amount of time required to complete the task is modeled by X. In this case, the makespan—or the time it takes from the initial assignment of both tasks to when the last processor to finish finishes—is a random variable M_X with $E[M_X] = 161/18 \sim 8.94$. If the time to complete the task were instead modeled by Y, the makespan would be a random variable M_Y with $E[M_Y] = 5425/648 \sim$

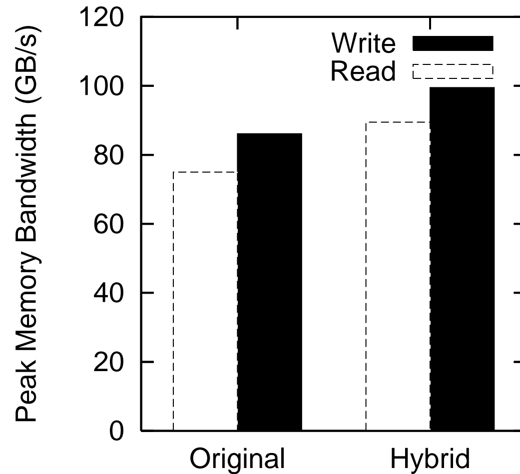

Figure 3.12: Comparison of global memory read bandwidth for original and hybrid simulators

8.37. Therefore, despite having the same average, the process with less variability can result in overall reduction in the total time to complete tasks in parallel, and it is this author's suggestion that the small discrepancy in observed instructions per cycle could be attributable, in whole or in part, to this phenomenon, particularly in light of the marked variance in the memory fetch request latency results. Whether this variability might become a source of concern later is another question, and one which may merit further consideration.

Figure 3.12 shows the global memory read bandwidth for the original and hybrid graphics simulators. As in the other experiment, the hybrid simulator is observed to achieve a slightly higher memory bandwidth than was possible in the original GPGPU-Sim, possibly for the reasons already hypothesized; in any event, the discrepancy is minor enough to be attributable to a variety of factors, including slight variations in the device timing parameters or in simulation internals or scheduling policies. Note also that the difference between read and write achieved bandwidth is consistent with experimental results from real hardware, presented earlier. The following modifications had to be made to the SHOC DeviceMemory benchmark in order to collect these results: first, automatic tuning of iterations inside device kernels based on measured runtime had to be disabled, since the simulator is not real-time and excessively long initial runs can cause the benchmark to abort; second, and as a result of

the first modification, device iterations had to be limited manually, and as such, the reported memory bandwidths should not be mistaken as peak bandwidths (indeed, only between half and a third of total peak memory bandwidth is achieved as a result of simulation, compared to experimental results for the GTX 480).

Chapter 4

Related and Future Work

This chapter contains a brief summary of some notable research efforts with goals which are closely aligned with the goals of this project, as well as some closing remarks on the work described herein and future work to be carried out to complete the project's ultimate goals. Specifically, section 4.1 describes research endeavors involving the application of novel memory technologies to general purpose graphics processing, including attempts to introduce both hardware transactional memory and spin-transfer torque memory, a kind of magnetoresistive memory technology. Section 4.2 briefly summarizes the key motivations and contributions of the project and, in particular, of the work discussed in detail in the previous chapter.Also, future work and possibilities for further research are identified.

4.1 Related Work

Recently, research carried out at the University of Virgina [24] investigated the potential to use spin-transfer torque random access memory, a kind of magnetoresistive memory technology, as a replacement for static random access memory for shared memory in graphics processors. One of the motivations for that project—power savings—is the driving factor behind the present project. Other motivations for using spin-transfer torque memory as a replacement for shared memory include both area savings and increased capacity (since this variety of memory technology has a smaller cell size than static random access memory; see relevant sections of chapter 2 for more details). The goal of the study was to evaluate the potential of this non-volatile memory technology for use in graphics devices; longer write latencies were justified there, as here, by the observation that massive thread parallelism can effectively reduce the negative impact of high-latency transfers. GPGPU-Sim was used to

simulate performance, while CACTI was used for power and area estimation. The results were quite positive: it was found that the use of non-volatile memory technology resulted in appreciable power (and, in their case, area and capacity) improvements. The affirmative finding in this study adds considerably to the confidence in the project currently being carried out. The present work differs from this work in that the target for replacement is the graphics global memory, rather than the graphics shared memory.

W. Fung et al. [12] studied potential benefits from using hardware transactional memory to implement an inter-multiprocessor synchronization primitive free from data races and deadlocks. Using GPGPU-Sim, they demonstrated that it was possible to capture much of the benefit of fine-grained locking mechanisms, enabling significantly improved performance compared to equivalent sequences of serial transactions. Although the methods employed to evaluate this novel hardware architecture are in many ways similar to those supporting this project, the motivations and goals are starkly dissimilar. A team of researchers from International Business Machines have filed a patent [14] on the use of a dedicated non-volatile memory storage device for graphics devices, in order to mitigate performance degradation due to the I/O bottleneck, and particularly, to the disk I/O bandwidth. This work differs from the present work in that it seeks only to augment graphics devices to avoid performance bottlenecks, instead of exploiting non-volatile memory technologies' power advantages.

4.2 Conclusions and Future Work

The notion of utilizing non-volatile memory technology, specifically phase-change random access memory, as a replacement for graphics global memory has been introduced and motivated with considerations of graphics and memory performance characteristics. The need for an appropriate means of evaluating competing design decisions was then explained as a key challenge in enabling future work in this direction. Appropriate simulation frameworks—GPGPU-Sim for graphics devices and DRAMSim2 for phase-change memory—were identified and justified, and the integration of these frameworks into a hybrid simulation

framework, GPUHM-Sim, has been described in detail. The design and initial implementation of this integrated simulation framework are the main contributions of this work, and support future research goals. Ultimately, the project will help provide useful insight into the application of non-volatile memory technologies to general-purpose graphics processing devices.

The work described in this thesis represents only a part (albeit an important one) of a larger research effort aimed at evaluating benefits of a proposed novel architecture. A great deal of additional work must be performed in support of these objectives. Additionally, while the design and architecture of the integrated simulation framework is essentially complete in its major features, there remain both design and implementation challenges which, when overcome, will greatly facilitate use of the tool. In any event, GPUHM-Sim should provide a much-needed capability to simulate novel hybrid architectures, enabling further research work in this direction.

Bibliography

[1] T. Aamodt et al., "GPGPU-Sim 3.x Manual," http://gpgpu-sim.ece.ubc.ca/GPGPU-Sim_3.x_Manual, 2012.

[2] M. Bauer, H. Cook, B. Khailany, "CudaDMA: optimizing GPU memory bandwidth via warp specialization," In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011.

[3] A. Bakhoda, G. Yuan, W. Fung, H. Wong, T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 163–174, 2009.

[4] P. Carpenter, "Performance in Scientific Computing with a Performance Characterization of the Parallel Ocean Program on Ranger," Bachelor's Thesis, Auburn University, 2010.

[5] S. Che, J.W. Sheaffer, K. Skadron, "Dymaxion: optimizing memory access patterns for heterogeneous systems," In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011.

[6] L. Chua, "Memristor - The missing circuit element," IEEE Transactions on Circuit Theory, vol. 18, iss. 5, pp. 507–519, 1971.

[7] B.R. Coutinho, D.N. Sampaio, F.M.Q. Pereira, W. Meira, "Divergence Analysis and Optimizations," In Proceedings of 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 320–329, 2011.

[8] A. Danalis, G. Marin, C. McCurdy, J. Meredith, P. Roth, K. Spafford, V. Tipparaju, J. Vetter, "The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite," In Proceedings of the Third Workshop on General-Purpose Computation on Graphics Processors (GPGPU 2010), 2010.

[9] D. Foley, "A Low-Power Integrated x86-64 and Graphics Processor for Mobile Computing Devices," IEEE Journal of Solid-State Circuits, vol. 47, iss. 1, pp. 220–231, 2012.

[10] W. Fung, T. Aamodt, "Thread Block Compaction for Efficient SIMT Control Flow," In Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA-17), pp. 25–36, 2011.

[11] W. Fung, I. Sham, G. Yuan, T. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," In Proceedings of the 40th IEEE/ACM International Symposium on Microarchitecture (MICRO40), 2007.

[12] W. Fung, I. Singh, A. Brownsword, T. Aamodt, "Hardware Transactional Memory for GPU Architectures," In Proceedings of the 44th IEEE/ACM International Symposium on Microarchitecture (MICRO44), 2011.

[13] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R.S. Williams, K. Yelick, "Exascale Computing Study: Technology Challenges in Achieving Exascale Systems," Information Processing Techniques Office (IPTO) of the Defense Advanced Research Projects Agency (DARPA) of the United States of America, 2008.

[14] A.W. Herr, A.T. Lake, R.T. Tabrah, "Non-volatile storage for graphics hardware," United States Patent Application Publication, Pub. No. US 2011/0292058 A1, 2011.

[15] N.S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J.S. Hu, M.J. Irwin, M. Kandemir, V. Narayanan, "Leakage current: Moore's law meets static power," IEEE Computer, vol. 36, iss. 12, pp. 68–75, 2003.

[16] NVIDIA Corporation, "CUDA Best Practices Guide," developer.nvidia.com, 2011.

[17] NVIDIA Corporation, "CUDA Programming Guide," developer.nvidia.com, 2011.

[18] NVIDIA Corporation, "CUDA Toolkit 4.0," developer.nvidia.com, 2011.

[19] NVIDIA Corporation, "CUDA Toolkit 4.1," developer.nvidia.com, 2011.

[20] Oak Ridge National Laboratory, "Scalable HeterOgeneous Computing (SHOC) Benchmark Suite," http://ft.ornl.gov/doku/shoc/start, 2012.

[21] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips, "GPU Computing," In Proceedings of the IEEE, vol. 96, no. 5, pp. 879–899, 2008.

[22] T. Perez, C.A.F. De Rose, "Non-Volatile Memory: Emerging Technologies And Their Impacts on Memory Systems," Technical Report No. 060, Pontificia Universidade Catolica do Rio Grande do Sul, 2010.

[23] P. Rosenfeld, E. Cooper-Balis, B. Jacob, "DRAMSim2: A Cycle Accurate Memory Simulator," Computer Architecture Letters, vol. 10, iss. 1, pp. 16–19, 2011.

[24] P. Satyamoorthy, "STT-RAM for Shared Memory in GPUs," Master's Thesis, The University of Virginia, 2011.

[25] The Green500, "The Green500 List :: Environmentally Responsible Supercomputing," http://www.green500.org/, 2011.

[26] Top500.org, "TOP500 Supercomputing Sites," http://www.top500.org/, 2011.

[27] H. Wong, S. Raoux, S. Kim, J. Liang, J. Reifenberg, B. Rajendran, M. Asheghi, K. Goodson, "Phase Change Memory," In Proceedings of the IEEE, vol. 98, iss. 12, pp. 2201–2227, 2010.

[28] C.J. Xue, Y. Zhang, Y. Chen, G. Sun, J.J. Yang, H. Li, "Emerging non-volatile memories: opportunities and challenges," In proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis (CODES+ISSS '11), 2011.

[29] G. Yue-Feng, S. Zhi-Tang, L. Yun, L. Yan, "Programming voltage reduction in phase change cells with conventional structure," In Proceedings of the International Conference on Electric Information and Control Engineering (ICEICE 2011), pp. 2469–2471, 2011.