EVALUATION AND ENHANCEMENT OF MEMORY EFFICIENCY TARGETING GENERAL-PURPOSE COMPUTATIONS ON SCALABLE DATA-PARALLEL GPU ARCHITECTURES

A Dissertation Presented
by
Byunghyun Jang
to
The Department of Electrical and Computer Engineering
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in the field of
Computer Engineering
Northeastern University
Boston, Massachusetts
January 2011
© Copyright 2011 by Byunghyun Jang
All Rights Reserved
Abstract

This thesis addresses the memory efficiency of general-purpose applications running on massively multi-threaded, data-parallel GPU architectures. Although scalable, data-parallel GPU architectures and their associated general-purpose programming models offer impressive computational capability and attractive power budgets, the pace of migrating general-purpose applications to this emerging class of architectures is significantly hindered by the efficiency of memory subsystem present on these platforms. Programmers are forced to optimize the memory behavior of their code if they are interested in reaping the full benefits of these high performance, data-parallel architectures.

In this thesis, we present a comprehensive study of memory access behavior for data-parallel workloads targeting GPUs, and present an algorithmic methodology to address memory inefficiency issues. We establish a mathematical model to capture memory behavior that enables us optimize memory system performance. We present a comprehensive analysis of memory access patterns that fully incorporates the influence of thread mapping and explains the memory behavior of kernels running on GPU hardware—this modeling and analysis serves as a theoretical foundation throughout this thesis. We then show how this new model of memory system activity can be used to enhance the memory efficiency of kernels through a series of algorithmic memory efficiency enhancement techniques. The techniques explored in this thesis include: 1) vectorization via data transformations on vector-based GPU architectures, 2) appropriate memory space selection, and 3) search for an optimized thread mapping and
work group size. To demonstrate the power of our proposed algorithmic methodology, we develop a tool that implements this proposed approach and tests it on a diverse class of general-purpose benchmark applications. The experiments are conducted using the industry standard heterogeneous programming language, OpenCL, on two mainstream GPU platforms available in the market.
Acknowledgements

I am grateful to many people and was very fortunate to have them.

First of all, I’d like to express my sincere appreciation to my advisor, Prof. David Kaeli for his invaluable guidance, advice, and support. He taught me not only technical knowledge but a number of priceless treasures such as leadership and professionalism. It was my fortune to meet and have him as my PhD advisor.

I’d also like to thank my labmates who spent countless time with me at 225 and 263 Egan building. We together have discussed, shared, worked, taught, reviewed, and encouraged. Those precious moments will never be forgotten.

Special thanks go to my committee, Chris Reeve and Prof. Waleed Meleis. My internship with Chris and the Shader Compiler Team at AMD provided me with a number of research ideas which have resulted in several sections of this dissertation. Two semesters of working with Prof. Meleis as his TA gave me an opportunity to rethink about how important the role of a teacher is for the future of young minds.

Lastly but most specially, I am grateful to my lovely wife, Jihye, for her endless support and love. She and my two sons, Jun and Yoon, have always been a fundamental motivation and energy. I also must acknowledge my father, Moonwon Jang, who must have been watching me from the heaven, and my mother, Yoonjung Noh, for their extraordinary passion for my education. I am glad that their belief and efforts have become fruitful.
Contents

Abstract iii
Acknowledgements v

1 Introduction 1
  1.1 A Brief History of GPU Computing . . . . . . . . . . . . . . . . . . 3
  1.2 The Benefits of GPU Computing . . . . . . . . . . . . . . . . . . 4
  1.3 The Challenges in GPU Computing . . . . . . . . . . . . . . . . . 6
  1.4 Contributions of This Thesis . . . . . . . . . . . . . . . . . . . . 10
    1.4.1 A Mathematical Model for Memory Access Behavior . . . . . 10
    1.4.2 A Comprehensive Mathematical Analysis and Characterization
        of Memory Access Patterns . . . . . . . . . . . . . . . . 11
    1.4.3 Development of Memory Efficiency Enhancement Techniques
        Across Different Architectures . . . . . . . . . . . . . . 12
    1.4.4 Demonstration of the Effectiveness of the Proposed Methodol-
        ogy Through Extensive Benchmarks and Tool Development . . 13
  1.5 Organization of This Thesis . . . . . . . . . . . . . . . . . . . . 13

2 Background and Related Work 15
  2.1 GPU Hardware Architecture . . . . . . . . . . . . . . . . . . . . 15
    2.1.1 CPU vs. GPU . . . . . . . . . . . . . . . . . . . . . . . . . 16
    2.1.2 GPU Memory Subsystem . . . . . . . . . . . . . . . . . . . 20
2.1.3 Two Variants: Vector-based (AMD) and Scalar-based (NVIDIA) GPUs ............................................. 25
2.2 GPU Programming Model ................................................. 30
  2.2.1 OpenCL .............................................................. 32
2.3 GPU Execution Model .................................................. 35
2.4 Related Work ............................................................. 37
  2.4.1 Manual Mappings and Their Optimizations ................. 37
  2.4.2 Automatic Approach .............................................. 39
  2.4.3 Polyhedral Model ................................................. 41
  2.4.4 Other Models ...................................................... 42
  2.4.5 Other Research Trends ............................................ 44

3 Motivation: Memory Efficiency and Performance ................. 46
  3.1 Impact of Vectorization on Performance ......................... 46
  3.2 Impact of Memory Space Selection on Performance .......... 49
  3.3 Impact of Thread Mapping on Performance ..................... 52
  3.4 Impact of Work Group Sizing on Performance ................. 53

4 Memory Access Modeling ................................................... 57
  4.1 Serial Model without Thread Mapping ............................ 57
  4.2 Parallel Model Incorporating Thread Mapping ................. 60

5 Memory Access Analysis ................................................... 66
  5.1 Basic Pattern Classification ........................................ 66
  5.2 Extended Pattern Classification ................................... 68
    5.2.1 Coalesced Memory Accesses .................................. 71
    5.2.2 Accesses to the Same Memory Address ..................... 71
    5.2.3 Data Prefetch .................................................. 72
### 6 Vectorization via Data Transformations  
6.1 Background: Vectorization on GPUs ........................... 74  
6.1.1 Conventional Loop Vectorization ........................... 75  
6.1.2 Loop Vectorization on AMD GPUs ........................... 76  
6.2 Vectorization Test ................................................. 78  
6.3 Quantifying Vectorization .......................................... 79  
6.4 Generating Data Transformation Rules ............................ 80  
6.5 Vectorization Algorithms ........................................... 83  
6.6 Evaluating Vectorization on GPUs ................................. 85  

### 7 Memory Space Selection ........................................... 94  
7.1 Global Memory Space Selection ................................. 94  
7.2 Constant Memory Space Selection ............................... 95  
7.3 Texture Memory Space Selection ............................... 98  
7.4 Local Memory Space Selection ............................... 100  
7.5 Memory Selection Algorithm .................................. 100  
7.6 Experimental Results ........................................... 101  

### 8 Thread Mapping and Work Group Sizing ......................... 108  
8.1 Cost and Gain Calculation ...................................... 109  
8.2 Resource Constraints and Search Space .......................... 110  
8.3 Work Group Size on Commodity GPUs ........................ 111  
8.4 Evaluation of Thread Mapping and Work Group Sizing ........ 113  

### 9 Discussion and Future Work ...................................... 123  
9.1 Discussion of the current research ............................ 123  
9.2 Future Work ................................................... 126  

### 10 Conclusion ..................................................... 129  

Bibliography ......................................................... 131
List of Figures

2.1 *Comparison of CPU and GPU hardware layout and silicon utilization.* Modern multi-core CPUs devote a significant amount of transistors to control logic and caches to increase serial code performance, while GPUs devote more transistors to computational logic to achieve throughput computing. GPUs are also composed of independent multi-processors for hardware scalability, while cores in the CPU are tightly-coupled together, making it difficult to scale the number of cores.  

2.2 *Compute capability trend between the CPU and GPUs.* The compute capability of GPUs is two orders of magnitude higher than that of current CPUs and the gap continues to widen.

2.3 *Memory bandwidth trend between CPUs and GPUs.* The memory bandwidth of GPUs is much higher than that of CPUs and the gap continues to widen.

2.4 *Simplified diagram of a modern GPU memory subsystem.* GPU memory is composed of on-chip and off-chip memories. On-chip memory includes caches, software-managed local scratchpad memory, and registers. Off-chip memory includes global, texture, and constant memories. The detailed hardware implementation and the characteristics of each memory space varies by vendor. The visibility and accessibility of each memory space can vary across different GPU programming models.
2.5 *Thread groups and their mapping to GPU hardware.* A kernel composed of a number of thread groups is executed on a GPU hardware in a way that its thread groups are assigned onto one of the multiprocessors by hardware scheduler until all thread groups are completed. 31

2.6 *Scalable GPU programming model* [1]. The two-level thread hierarchy (i.e., thread grouping) maintains the same level of hardware utilization across both devices with different number of multiprocessors and different input problem sizes. Symbols, TG and MP, refer to thread group and multiprocessor, respectively. 33

2.7 *An OpenCL memory model.* OpenCL memory model is very similar to modern GPUs’. 35

2.8 *A GPU execution model.* When executed, the hardware groups the threads of a work group into a batch of threads (typically 16 to 64 threads) and schedules those threads together. The size of the thread batch varies across hardware vendors and GPU models. 36

3.1 *Performance impact of vectorization in matrix multiplication kernel on an AMD platform.* The performance impact of other optimizations can be found in our earlier work [2]. 48

3.2 *Performance impact of memory space selection in matrix multiplication kernel on an NVIDIA platform.* Efficient utilization of local memory boosts the performance by an order of magnitude over the default memory space (global memory). Appropriate memory space selection based on memory access pattern is a key to maximize effective memory bandwidth of GPUs. 50

3.3 *Performance impact of memory space selection in forward and backward projection routines on a NVIDIA Geforce 8800 Ultra GPU.* Utilizing memory spaces is shown to be the most effective optimization in these kernels. 51
3.4 Two possible thread mappings of matrix multiplication kernel. (a) The outermost loop iterated by \( i_1 \) maps to \( tx \) and the middle loop iterated by \( i_2 \) maps to \( ty \), (b) The outermost loop iterated by \( i_1 \) maps to \( ty \) and the middle loop iterated by \( i_2 \) maps to \( tx \).

3.5 Impact of thread mapping on performance on a NVIDIA GTX 285 GPU. The changes in memory access pattern between two thread mappings make huge performance difference.

3.6 Map view of the performance of matrix multiplication kernel across different combinations of work group size configurations. The work group sizing is a multi-variables optimization problem. The shade of gray indicates the execution time (the darker, the shorter execution time).

4.1 Memory access pattern partitioning of matrix multiplication kernel after thread mapping. The columns of the memory access pattern associated with thread indices represent inter-thread memory access patterns and the rest of the columns represent intra-thread memory access patterns.

4.2 Graphical representation of memory access pattern partitioning of matrix multiplication kernel after thread mapping. The two-dimensional thread map and physical memory layout are shown to help illustrate the inter-thread and intra-thread memory access patterns.

5.1 Basic memory access pattern classifications and their representations in the model [3]. Note that the element being accessed is shown in gray, \( C \) is a constant, and \( Z \) is a random number.

5.2 Two different linear patterns and their representation in our parallel model. “W” denotes the width of a work group. The false linear pattern does not lead to fully coalesced accesses while the true linear pattern does.
5.3 True shifted and false shifted patterns and their representation in our parallel model. “W” denotes the width of a work group.

5.4 A coalesced memory access pattern and its representation in the model. Adjacent threads whose IDs are consecutive access adjacent memory addresses. Note that this example matches true linear pattern.

5.5 A same address read and its representation in the model. There is no inter-thread pattern and there is a intra-thread pattern.

6.1 Memory mapping for GPU vectorization. The size of an array of float type is reduced by a factor of 4 when it maps to an array of float4 type during memory copy from host to GPU device.

6.2 Performance comparison between scalar and vector implementation of matrix multiplication kernel on an AMD platform. The performance impact of other optimizations can be found in our earlier work [2].

6.3 Performance comparison between scalar and vector implementation of Hydro Fragment kernel on an AMD platform.

6.4 Performance comparison between scalar and vector implementation of General Linear Recurrence Equations kernel on an AMD platform.

6.5 Performance comparison between scalar and vector implementation of Cholesky Decomposition/Substitution Subroutine kernel on an AMD platform.

6.6 Performance comparison between scalar and vector implementation of Vpenta kernel on an AMD platform.

7.1 Captured memory access pattern for the kernel in Listing 7.1. Array Y possesses an appropriate memory access pattern for constant memory.

7.2 Performance impact of placing array Y in constant memory in the example kernel shown in Listing 7.1. The performance gain associated with using constant memory is substantial.

7.3 Captured memory access pattern for the kernel in Listing 7.2. Array Y possesses a shifted access pattern.
7.4 Performance impact of placing array Y in texture memory in the example shown above at array size 2048. ........................................ 99

7.5 Performance comparison between using default global memory and using selected memory for the Parboil ComputeQ kernel as run on an NVIDIA platform. Note that we used the input data size provided by benchmark suites. ......................................................... 105

7.6 Performance comparison between using default global memory and using selected memory for the Parboil ComputeFH kernel as run on an NVIDIA platform. Note that we used the input data size provided by benchmark suites. ......................................................... 106

7.7 Performance comparison between using default global memory and using selected memory for the Parboil CP kernel as run on an NVIDIA platform. Note that we used the input data size provided by benchmark suites. ......................................................... 106

7.8 Performance comparison between using default global memory and using selected memory for the Parboil SAD kernel as run on an NVIDIA platform. Note that we used the input data size provided by benchmark suites. ......................................................... 107

7.9 Performance comparison between using default global memory and using selected memory in for the PhysBAM kernel as run on an NVIDIA platform. Note that we used the input data size provided by benchmark suites. ......................................................... 107

8.1 Cost of two thread mapping schemes for matrix multiplication kernel when the work group size is 32(tx) × 8(ty). “×” indicates uncoalesced memory accesses and SL denotes the loop strip length. ............ 110
List of Tables

1.1 GFLOPs/Watt and GFLOPs/$ comparison between commodity CPU and GPUs. The GFLOPs/Watt and GFLOPs/$ of the GPU are up to 37.1 and 40.8 times those of the CPU. † Maximum Power Consumption. ‡ As of Oct. 2010 at Amazon.com.

2.1 Hardware comparison between the modern CPU and GPUs. Notable differences include the number of cores, the number of parallel threads supported, die size, and the number of transistors. † Maximum number of active threads that hardware supports.

2.2 GPU memory spaces. † Per compute unit. ‡ With limitations per object.

2.3 GPU memory spaces (continued).

2.4 Comparison between AMD and NVIDIA GPU computing platforms. † Brook+ is deprecated and no longer officially supported. ‡ The size of wavefront varies (16 to 64 threads) depending on machine variants.

3.1 Machine code statistics of scalar and vector version of matrix multiplication kernel. The machine code statistics are obtained from AMD Stream Kernel Analyzer [4]. † The ratio between ALU and memory instruction count. ‡ General Purpose Registers.

6.1 Loop characteristics for the benchmark suites studied for the vectorization. A significant percentage of loops in high performance applications are vectorizable when applying data transformations.
6.2 Breakdown of the reasons why benchmark loops can not be vectorized even after data transformation. .................................................. 92
6.3 Summary of benchmark kernels for vectorization on an AMD platform. 92
6.4 Summary of benchmark kernels for vectorization on an AMD platform (continued). *The speedup shown includes data transformation overhead and is measured when the size of each dimension of the array is 3072. ................................................................. 93
7.1 Summary of benchmark kernels tested for memory space selection. . 103
7.2 Summary of benchmark kernels tested for memory space selection (continued). *The speedup shown is measured using the largest input size provided by each benchmark suite. ................................. 104
8.1 An example of algorithmic thread mapping and work group size search in matrix multiplication kernel. †ty×tx (total number of threads). . 114
8.2 An example of algorithmic thread mapping and work group size search in matrix multiplication kernel (continued). †ty×tx (total number of threads). * Execution time in milliseconds when input matrix size is 1024×1024. ................................................................. 115
8.3 Experimental results of work group search for vector addition kernel. The register count is 3. ................................................................. 116
8.4 Experimental results of work group search for vector addition kernel (continued). *The input size is 2^{25}. ................................. 116
8.5 Experimental results of work group search for matrix-vector multiplication. The register count is 17. ................................................................. 117
8.6 Experimental results of work group search for matrix-vector multiplication (continued). *The size of each dimension of the input arrays is 4096. †Speedup over default global memory space. ......................... 118
8.7 Experimental results for the radiological path calculation. The register count is 28. ................................................................. 120
8.8  Experimental results for the radiological path calculation (continued).

   The input image size is $256 \times 32$. ........................................ 120

8.9  Experimental results of NMS algorithm. The register count is 15. . . . 121

8.10 Experimental results of NMS algorithm (continued). The input image

   size is $256 \times 256$. ............................................................ 122
Chapter 1

Introduction

Driven by the demand for real-time 3D graphics rendering, Graphics Processing Units (GPUs) have evolved rapidly as very powerful, fully programmable, many-core, data-parallel architectures. With the introduction of improved development environments and support for general-purpose high-level programming languages, GPUs have recently become the co-processor of choice to accelerate a range of general-purpose data-parallel applications [5, 6, 7, 8, 9]. Compute-intensive, data-parallel portions of a given application, called kernels, are offloaded to a GPU, accelerating the application by several orders of magnitude, while the host CPU continues to execute non-kernel tasks. This evolution of GPUs into general-purpose, data-parallel computing platforms has opened up a new era of affordable heterogeneous supercomputing and has changed the landscape for the future of High Performance Computing (HPC).

As a result of these advances, a new research community has been spawned that has successfully mapped a broad range of computationally demanding, complex problems to GPUs. This movement is commonly referred to as GPU computing or General-Purpose computation on a GPU (GPGPU), and has been gaining momentum by reaching out to a broad class of application domains. Several commercial end-user products that harness the power of GPUs are already available in the market [10, 11] and the latest edition of the popular computer architecture textbook by Patterson
and Hennessy [12] has now introduced a chapter on GPU computing.

The tremendous computing benefits that GPUs offer, however, do not come for free. Fundamental differences in the underlying GPU hardware architectures from conventional general-purpose processors make it challenging to efficiently map general-purpose workloads to this emerging class of architectures [5, 6, 7, 13, 14, 15, 16, 17, 18]. Efficient utilization of the memory subsystem tends to be one of the most challenging issues [19, 20, 21, 22, 23, 24]. At any point in time on a GPU, hundreds or thousands of threads try to issue memory reads or writes simultaneously, and these accesses are serialized if the memory pattern is not optimized for underlying memory subsystem. Memory optimization is at the center of GPGPU programming and optimization.

In this thesis, we present an algorithmic approach to effectively address memory inefficiency issues when mapping serial, data-parallel loops onto massively multi-threaded, data-parallel GPU hardware. We present a memory access pattern modeling and analysis methodology that 1) helps programmers understand the behavior of memory operations on GPUs, 2) guides programmers on how to optimize their code quickly and efficiently, and 3) potentially serves as a theoretical foundation to an automatic parallelization framework. Our proposed model is simple yet powerful in the sense that it can be easily embedded into automatic parallelization or source-to-source compiler while capturing almost all information required to enable a number of performance-critical memory optimizations. To the best of our knowledge, this is the first work to deal with memory access patterns on massively-parallel GPUs in an algorithmic and systematic fashion, to additionally consider the impact of thread mapping on performance, and to present algorithmic techniques to apply memory optimizations. Our proposed methodology provides many benefits and opportunities on GPUs.

In this chapter, we present an introduction to GPUs for general-purpose, high-performance computing. We begin by presenting a brief history of GPU computing,
the unique benefits that it provides, and the current challenges facing the GPU computing community. Finally, we highlight the contributions of this thesis and provide an outline of the remainder of the thesis.

1.1 A Brief History of GPU Computing

The idea to run general-purpose applications on computer graphics hardware was introduced in the late 1990’s. At that time, the GPU was a fixed-function processor designed as a graphics rendering pipeline. During this early period, programming GPUs for general-purpose programs was done by using a graphics Application Programming Interface (API) such as OpenGL [25] or DirectX [26]. Algorithms had to be cast in terms of graphic functions, which is difficult and time-consuming. Developing GPU applications required deep knowledge of the underlying hardware and graphics programming interfaces, and application development was limited by the APIs. Despite all these obstacles, a large volume of research work obtained significant speedups and demonstrated that the GPU was a very promising platform for compute-dominated workloads [23, 27, 28, 29].

The introduction of programmable commodity graphics hardware in 2001 (AMD’s R200 family and NVIDIA’s G30 family) allowed for more flexible general purpose programming and helped to increase interest in using GPUs as general-purpose co-processors. As a result, the term General Purpose computation on GPUs (GPGPU) was coined for this new computing domain. At that time, general-purpose programming on GPUs was done using shader languages which were still designed specifically for graphics rendering. The shader languages used included C for graphics (Cg) [30], High-Level Shading Language (HLSL) [31], and OpenGL [25]. The use of these languages for GPGPU persisted through 2006.

Significant changes occurred in 2006 in the GPGPU research community. The new GPU architectures unified the vertex and pixel processors (R600 family from AMD and G80 family from NVIDIA) and new Software Development Kits (SDKs)
dedicated to general-purpose programming (CUDA [32] and Brook+ [33, 34]) were developed. Fully programmable hardware and associated programming languages firmly established the GPU as a powerful data-parallel many-core co-processor for accelerating general-purpose applications. Since then, GPGPU has received enormous attention from researchers in a wide range of scientific and engineering fields and a large number of general-purpose applications have been reported to be successfully mapped to GPUs.

A more detailed history of GPU computing can be found in [7, 14, 35].

1.2 The Benefits of GPU Computing

GPUs provide advantages over other high performance parallel platforms in terms of both performance and cost, as well as other a number of metrics. We highlight some of these advantages in this section.

Inexpensive GPUs are inexpensive in the sense that they provide unprecedented levels of GFLOPs per dollar cost-performance. Given their impressive computational capabilities, and combined with the recent advances in associated parallel programming frameworks, GPUs offer an attractive path to low-cost high-performance supercomputing. Table 1.1 shows a GFLOP per dollar comparison between commodity GPUs and CPUs available in 2010.

Low power GPUs offer very attractive power budgets for the High Performance Computing (HPC) community. They provide unprecedented GFLOPs per watt ratios which cannot be found in any other existing platforms. A platform’s power budget is increasingly becoming the biggest concern for large scale computing, especially as the amount of data to process and the computational demands increase exponentially. Therefore, the GPU’s low power requirements
CHAPTER 1. INTRODUCTION

Table 1.1: GFLOPs/Watt and GFLOPs/$ comparison between commodity CPU and GPUs. The GFLOPs/Watt and GFLOPs/$ of the GPU are up to 37.1 and 40.8 times those of the CPU. † Maximum Power Consumption. ‡ As of Oct. 2010 at Amazon.com.

are particularly attractive for large scale systems such as data centers and supercomputers. Table 1.1 shows the GFLOP per watt between commodity CPU and GPUs.

Accessibility GPUs are everywhere. All desktop and laptop computers have a GPU on board. This ubiquitous access to GPUs enables us to accelerate commonly-used desktop applications such as text processing and multimedia applications [6, 13, 35, 36]. Accessibility is also an important design point for developers because the cost of software development can be easily justified if there exists a significantly large customer population. This has been a major problem with traditional parallel computing systems that have negligible market presence compared to conventional general-purpose processors [14]. GPGPU-enabled GPUs are ubiquitous in hundreds of millions of PCs. GPU Computing marks the first time that high performance parallel computing has been feasible within a mass-market product.

Small form factor Small form factor is a very attractive benefit that GPUs uniquely offer. Before the introduction of the GPU computing, parallel software applications have been running on servers and clusters of room or house size and such space requirements limit the use of these applications. In fact, there have been a number of HPC systems that failed to be widely adopted due to those reasons. Kirk and Hwu’s note [14] that the National Institute of Health (NIH) refused
to fund parallel programming projects for some time because they realized that huge cluster-based machines would not work in a clinical setting.

**Scalability** Another attractive feature of the move to GPUs comes from their innate ability to scale with problem sizes [1, 37]. Both the hardware architecture and programming model are designed for scalable computing; a kernel is equally divided into multiple thread blocks and each block is assigned to a multiprocessor by the hardware scheduler. Therefore, a properly designed kernel can easily scale in performance with the number of multiprocessors available in hardware. This is particularly important given that each new generation of GPU provides a larger number of processing cores and the input size of data-parallel applications tends to vary significantly.

### 1.3 The Challenges in GPU Computing

Despite all of the attractive benefits mentioned in the previous section, there are a number of challenges to overcome for GPUs to become mainstream. We summarize some of these issues in this section.

**Program Optimization** If the application programmer does not care about performance, programming a GPU is very easy. He/she can literally write a parallel program in a matter of minutes. However, efficiently mapping a general-purpose algorithm onto a GPU platform is by no means an easy task. There are two root causes for this added difficulty: 1) remember that GPU hardware architectures have been specifically tuned for 3D graphics rendering, and 2) their lack of abstraction associated with close-to-the-metal programming models. GPU programmers must understand both the hardware architecture, and the proper programming constructs to access this hardware if they intend to reap the full benefits of this platform. Although GPUs are gradually adding more general-purpose features, the underlying microarchitecture of the GPU is fundamentally
different from conventional general-purpose processors (detailed in Section 2.1). Designed around the underlying hardware architecture, the programming model is very different from conventional ones and requires programmers to control low-level hardware resources. The programmer explicitly maps threads to each data point; careless thread mapping leaves code underperforming. These two aspects of GPU computing present a significant code optimization challenge to the programmer.

Another barrier to producing optimized code is that GPU hardware vendors keep a large portion of microarchitectural details confidential and inaccessible to the public. Programmers are left to attempt to apply trial and error methods to produce optimized codes.

Program optimization is probably the largest looming challenge in current GPU computing. We expect that hardware vendors and GPU researchers will begin to focus more on these issues. This makes a number of contributions in this direction.

**Memory Bottleneck** If we try to rank the causes of performance loss and bottlenecks in GPU computing, we would find that the memory subsystem is the primary cause [2, 19, 21, 38, 39, 40]. The theoretical peak performance shown in Figure 2.2 in Section 2.1.1 assumes no memory delays, but in reality, it is extremely difficult to hide all memory latencies.

The characteristics of GPU memory (high bandwidth, high latency, and non-adaptive) make the performance of a GPU kernel extremely sensitive to irregularity in memory access patterns. At any time, hundreds or thousands of threads issue read or write memory operations simultaneously, and these high-latency accesses are serialized if the threads generate access patterns that are not optimized for the underlying memory organization. Serialized off-chip memory accesses cause thread stalls, which in turn degrade performance significantly. Considering that many general-purpose applications possess irregular memory
access patterns, we need to be able to address irregularities to better utilize the underlying memory system and to reap the true benefits of GPU computing.

**Two-Level Thread Hierarchy** Another significant programming challenge is to develop a proper *two-level thread hierarchy*. The two-level thread hierarchy groups threads to utilize or share local hardware resources. It is a programming model that maximally utilizes the underlying scalable hardware architecture of GPUs, which are composed of multiple independent multiprocessors. Workloads must be divided into smaller sets that fully occupy the hardware resources of a multiprocessor (similar to the 0-1 knapsack problem) [1, 15, 41]. This hierarchical thread design needs to consider the limited local hardware resources (e.g., shared memories and registers) as well as other constraints (e.g., work group size and maximum number of threads per work group allowed on a single compute unit). This becomes a multi-variable optimization problem.

**On-Chip Hardware Resource Constraints** GPU kernels are continuing to grow in size as programmers attempt to map more complex algorithms to GPUs. Large kernels tend to possess higher *register pressure* [2, 42]. Registers are a limited hardware resource that restricts the number of active threads; a large number of threads are needed to hide memory latency. Register spilling is a technique that balances register pressure by saving/restoring a variable using slower global memory. From a compiler perspective, optimizing register pressure and register spilling is a very challenging task since their trade-offs can be unpredictable at compile time [43].

The size of local on-chip scratchpad memory may limit the performance of some applications when the applications exhibit a significant amount of data reuse that exceeds the size of local memory. The programmer should carefully select the most significant data to place in this memory to maximize performance.
Thread Divergence  When kernels are executed, the hardware batches a fixed number of threads and schedules them together \(^1\) (we will discuss this issue in Section 2.3). However, if threads within a batch take a different conditional branch path, the two paths are serialized—some threads must idle while others execute, reducing hardware utilization. Given this scenario, a kernel that is control-intensive may limit performance with the current GPU execution model. Fung et al. [44] propose a hardware approach that dynamically regroups threads into a new batch on the fly, to limit the impact of divergence in control flow.

Data Communication between Host and Device  Another challenge that needs to be addressed is the cost of data communication on the PCI Express bus between the host and the GPU device. If an application has multiple sections of CPU and GPU computations which are interleaved and data-dependent of each other, data transfer between host and device is inevitable. In this case, overall application performance may be limited by the overhead associated with these transfers. The heterogeneous computing community is well aware of this challenge and is trying to address the issue. One potential solution to this issue is to integrate the CPU and the GPU on the same die [6]. AMD’s Fusion [45] is one of the realizations of this approach.

Inter-thread Data Communication and Synchronization  GPUs lack the basic facilities required to support efficient, low-latency, inter-thread communication and synchronization. However, efficient transfer and synchronization of intermediate results is essential for many general-purpose algorithms. General-purpose multi-core CPU architectures use memory coherence and software locks to address this issue. In a GPU environment, memory coherency and synchronization generally require explicit barrier synchronization in the programming model. Kernels that require extensive use of inter-thread data communication

\(^1\) Note that this thread group is different from work group. The former is a subgrouping of work groups by hardware, while the latter is a grouping in high level programming by the programmer. The size of thread batches depends on the specific GPU model.
and synchronization should be rewritten to be more GPU friendly. Otherwise the computational resources on the GPU will remain idle and the performance will suffer.

1.4 Contributions of This Thesis

In this thesis, we address one of the most significant challenges that today’s GPU computing community is facing: memory efficiency. Our goals are twofold.

Our first goal is to provide programmers with an algorithmic methodology that guides them on enhancing the memory efficiency of their code. Programmers currently tune their applications manually by searching the design space exhaustively, without a full understanding of the performance characteristics of their applications. This approach is not only time consuming, but also frequently leaves the code unoptimized, underutilizing the available hardware resources.

Our second goal is to provide a theoretical foundation for automatic parallelization and for performance critical optimizations. The theoretical models that existing automatic GPU parallelization frameworks employ are not able to capture the concept of two-level hierarchical thread mapping, parallel hardware thread batching execution, and distinct characteristics of each memory space, thereby missing important optimization opportunities.

In the rest of this section, we summarize the contributions of this thesis, with a focus on the novelty that we uniquely introduce, develop, and analyze in this thesis.

1.4.1 A Mathematical Model for Memory Access Behavior

Built on a thorough study of memory access behavior of data-parallel workloads running on GPUs, we present a mathematical model that captures memory access patterns present in data-parallel workloads (Chapter 4). The motivation behind this
development is to provide the foundation for an algorithmic, structured optimization methodology, as well as for an automatic framework such as a source-to-source compiler.

The model we propose fully incorporates a two-level thread hierarchy and the parallel thread execution models that are currently employed in GPU computing. The model is simple enough to be easily embedded into automatic framework with minimal overhead, yet powerful enough to explain all memory access behavior taking place on GPUs. To our knowledge, our work is unique in this regard.

1.4.2 A Comprehensive Mathematical Analysis and Characterization of Memory Access Patterns

It occurs frequently that even experienced programmers do not fully understand the root cause of bottlenecks in their code. This phenomenon becomes more serious when it comes to memory efficiency. Memory optimization is the hardest part of GPGPU programming, and programmers need to have a good understanding of the memory behavior of their application in order to efficiently optimize their code and achieve high performance.

In this thesis, we present a mathematical model for the analysis and characterization of memory behavior of applications running on a massively multi-threaded, data-parallel GPUs using a mathematical representation (Chapter 5). We consider the differences between conventional multi-core CPUs and this emerging class of many-core architectures, to help the programmer identify the causes of memory bottlenecks in their applications, and devise a method to effectively optimize the memory behavior of their application.
CHAPTER 1. INTRODUCTION

1.4.3 Development of Memory Efficiency Enhancement Techniques Across Different Architectures

Using our proposed memory reference characterization model, we develop and present a series of algorithmic memory optimization techniques, each of which deals with key issues present on all GPU architectures.

First, we focus on vectorization optimization using data transformations targeting vector-based architectures (i.e., AMD GPUs) (Chapter 6). Using our model, we show that we can determine whether loop nests are vectorizable (vectorization test) and then arrive at the proper data transformation required to vectorize the original loop body. Our method increases the number of loops that can be vectorized and provides a significant increase in performance that easily amortizes the cost of implementing data transformations.

Second, we present an algorithmic memory space selection algorithm that maximizes the effective memory bandwidth using our model and analysis (Chapter 7). We consider the characteristics of each memory space considered by our model and present an algorithm that selects the best memory space for a given memory access pattern.

Third, we present an algorithmic search for optimized thread mappings and work group sizes using our model and analysis (Chapter 8). Using the right thread mapping and work group size are essential metrics when using current GPU programming models and can have an enormous impact on performance. Nonetheless, these topics have not been previously studied. Using our proposed metrics and other static analysis, we present an algorithm that searches for optimized thread mappings and work group sizes. This is perhaps the most significant contribution of this thesis.
1.4.4 Demonstration of the Effectiveness of the Proposed Methodology Through Extensive Benchmarks and Tool Development

To demonstrate the power of our proposed methodology, we build a tool that directly implements our modeling, analysis, and memory efficiency enhancement algorithms and then run it on a wide range of benchmark kernels. The benchmark applications we test include popular GPGPU benchmark kernels, well-known HPC benchmark suites, and real-world applications which cover a wide range of memory access patterns present in typical data-parallel workloads running on GPUs. We present experimental results using an industry standard heterogeneous programming language, OpenCL, run on mainstream GPUs from two hardware vendors, AMD and NVIDIA.

1.5 Organization of This Thesis

The remainder of this thesis is organized as follows.

Chapter 2 provides the reader with necessary background information and related work on the subject of this thesis. This chapter is divided into four sections. The first section summarizes different GPU hardware architectures. It begins with comparisons between conventional multi-core CPUs and GPUs. It includes a discussion of general issues present in most GPU memory subsystems, followed by a discussion of two hardware variants available in the market. The second and third sections highlight the salient features of GPU programming and execution models, respectively. The last section concludes this chapter by summarizing related work in the literature.

Chapter 3 presents the motivation for this thesis. The chapter presents the impact of several important memory optimizations on performance. Using actual performance results of benchmark experiments, each section demonstrates the impact of vectorization, memory space selection, thread mapping, and work group sizing on performance. The goal of this chapter is to give the reader an appreciation for the
importance of memory efficiency with respect to overall performance.

Chapters 4 and 5 form the theoretical basis of the thesis, upon which all following algorithmic methodologies are built. Chapter 4 presents a mathematical model that captures memory access patterns present in a data-parallel loop nest. This model is used to drive a number of the optimizations developed later in this thesis. Chapter 5 presents an analysis of memory access patterns using the model presented in Chapter 4. The chapter presents the classifications of memory access patterns and discusses how GPU memory access behavior is represented in the model.

Chapters 6 through 8 present the different uses of our proposed memory access modeling and analysis to enhance memory efficiency on GPUs. Chapter 6 presents vectorization using data transformations targeting vector-based GPU architectures. Chapter 7 shows an algorithmic memory space selection based on our static memory access pattern analysis. Finally, Chapter 8 presents an algorithmic search for optimized thread mappings and work group sizes. Note that Chapter 6 specifically targets AMD GPUs while Chapters 7 and 8 are applicable to any GPU available today—though we benchmarked on NVIDIA GPUs in this thesis.

Chapter 9 presents the limitations of our approach and discusses remaining future works.

Chapter 10 concludes the thesis.
Chapter 2

Background and Related Work

To place the contributions of this thesis in a proper context, this chapter introduces the basics of GPU computing and discussed related prior work. We start by reviewing the state-of-the-art in GPU hardware, emphasizing the differences from conventional CPUs and two different hardware implementations. Differences in the underlying memory system is discussed in detail. We then review current GPU programming and execution models, which are key concepts to fully appreciate the importance of memory performance when tuning GPU applications. Finally, we present a survey of the related research that considers the limitations of current GPU computing approaches. The chapter provides the reader a solid understanding of the topics required to completely understand the rest of the thesis.

2.1 GPU Hardware Architecture

To effectively reap the full benefit of GPU hardware, the programmer needs to understand the details of the underlying hardware architecture. As GPUs have evolved to provide real-time, high-definition 3D graphics, GPU hardware architecture, these devices have incorporated customized hardware architectures that have diverged significantly from modern multi-core CPUs. In this section, we highlight some of these
differences, emphasizing particular design features in their memory subsystem. This discussion includes a review of the two most popular commodity GPU platforms available in the market.

### 2.1.1 CPU vs. GPU

We begin our discussion of the hardware architecture of GPUs by contrasting their overall organization and memory subsystems with more familiar modern CPUs. The design philosophies between CPUs and GPUs are fundamentally different. Figure 2.1 illustrates the key differences in hardware configuration and silicon utilization between the two architectures. The design of the CPU is optimized for sequential code performance, where the code is control-intensive and possesses complex and irregular memory access patterns. The CPU devotes more transistors to control logic that enables threads to execute independently. The CPU memory system automatically caches data which reduces memory latencies and achieves high performance on sequential code. GPUs, on the other hand, are especially well-suited to solve problems that can be expressed as many independent computations in parallel with high arithmetic intensity (i.e., the ratio of the number of arithmetic operations divided by the number of memory operations). The GPU devotes more transistors to computational logic rather than data caching and control flow logic, enabling high-throughput data-parallel computing.

Another salient feature of GPU hardware lies in its scalability. Unlike conventional multi-core CPUs, where multiple ALU cores are integrated with common control logic and a memory subsystem, GPUs are composed of multiple independent multiprocessors (see Figure 2.1(b)), allowing hardware designers to easily expand the number of cores. This design trend will continue as these architectures have been shown to be highly-scalable, cost-effective solutions for large-scale high-performance computing [46, 47, 48]. However, this GPU hardware architecture presents challenges and produces a somewhat complex programming model (detailed in Section 2.2).
Modern multi-core CPUs devote a significant amount of transistors to control logic and caches to increase serial code performance, while GPUs devote more transistors to computational logic to achieve throughput computing. GPUs are also composed of independent multiprocessors for hardware scalability, while cores in the CPU are tightly-coupled together, making it difficult to scale the number of cores.

Table 2.1 compares a number of important hardware features between modern commodity multi-core CPU and GPUs. Along with Figure 2.1, this table shows that these two architectural styles are clearly targeting different classes of applications.

Figures 2.2 and 2.3 show the trend of compute capability and memory bandwidth between modern multi-core CPU and GPUs. The computational power of a GPU has well surpassed the performance of current multi-core CPUs; this performance gap is continually widening. Figure 2.2 presents data comparing the computational power of a CPU and a GPU as measured in billions of floating point operations per second.
Table 2.1: *Hardware comparison between the modern CPU and GPUs*. Notable differences include the number of cores, the number of parallel threads supported, die size, and the number of transistors. † Maximum number of active threads that hardware supports.

(GFLOPs).

GPU architectures offer higher memory bandwidth than CPUs (an order of magnitude higher, as can be seen in Figure 2.3). GPU memory subsystems are designed to deliver high bandwidth instead of low latency. High bandwidth is obtained through the use of a wide memory bus and specialized double data rate (DDR) memories that operate efficiently when the memory access granularity is large. The highest possible throughput is achieved when a large number of small memory accesses are buffered, reordered, and coalesced into a small number of large requests.

Another key difference is in how the two architectures address the issue of accessing off-chip memory, a very expensive operation incurring hundreds of clock cycles of latency. CPUs devote a significant number of transistors to on-chip caches in order to reduce stalls due to off-chip memory accesses. For applications with little data-level parallelism, this latency reduction approach is an effective strategy. For applications
Figure 2.2: Compute capability trend between the CPU and GPUs. The compute capability of GPUs is two orders of magnitude higher than that of current CPUs and the gap continues to widen.

possessing a large amount of parallelism, Simultaneous Multi-Threading (SMT) can be used. SMT is a powerful technique for hiding memory latency in applications, eliminating the need for more expensive caching mechanisms. This technique ensures that the processor remains busy by executing instructions from a different thread while one is waiting on memory accesses—latency is “hidden” rather than reduced. Latency hiding requires the ability to quickly switch from one thread to another. In contrast to CPU architectures, the overhead to perform thread switching is a single cycle on GPUs. Hence, GPU applications must launch kernels with a large number of threads or they can suffer from performance penalties. In the next chapter, we review the details of the GPU memory subsystem.
Figure 2.3: *Memory bandwidth trend between CPUs and GPUs.* The memory bandwidth of GPUs is much higher than that of CPUs and the gap continues to widen.

### 2.1.2 GPU Memory Subsystem

From the perspective of general-purpose computing, modern GPUs are a memory-bound architecture (versus compute-bound). Therefore, it is key to understand the underlying memory subsystem on a GPU. Our goal will be to optimize memory operations since this knowledge is crucial for the programmer to maximize the performance of many GPU applications [19, 21, 24, 28, 49].

Driven by the demand of real-time graphics rendering, GPUs are comprised of multiple memory spaces that have very different characteristics aimed at improving the performance of graphics applications. Threads on GPUs may access data from multiple memory spaces during their execution. For applications to obtain high performance on GPUs, the characteristics and requirements of these memory spaces must be well understood. Major factors to consider are whether the memory is physically located on-chip or off-chip (relative to the compute units), whether the memory is automatically cached, and its scope and access requirements.
The GPU memory system has also evolved toward high memory bandwidth (versus low latency) to achieve throughput computing. Therefore, maximizing effective memory bandwidth while avoiding high latency is a key to reap the full performance available on such a platform. Three techniques are at the core of achieving maximal effective memory bandwidth: efficient memory space utilization, coalesced memory access, and efficient thread mapping and work group sizing. GPU memory is characterized as high bandwidth (thanks to a wide memory bus), high latency (due to lack of caches), and non-versatile (due to the regular coalesced memory accesses associated with graphics processing). These characteristics make the memory performance of access patterns present in more general purpose kernels critical.

Memory spaces on the GPU include both off-chip and on-chip memories. Off-chip memory is comprised of constant memory and global memory; global memory is called texture memory when it is bound to a special hardware component called the texture fetch unit. On-chip memory includes some caches for constant and texture memory\(^1\), and local memory that is used for data reuse or local data sharing (i.e., scratchpad memory or a software-managed cache). On-chip memory of a compute unit is inaccessible by other compute units on the device. Two memory spaces, private memory and registers, are unique to individual threads and are allocated mostly at the compiler’s discretion. Private memory is used primarily for automatic array variables and to spill data when not enough registers are available, and is located off-chip.

Figure 2.4 shows a configuration of the various memory spaces of modern GPUs. Next, we will take a closer look at each memory space.

Global memory is the default memory space for input and output data. It is an off-chip memory and not cached, though it is the most flexible in terms of accessibility and size. Since accesses to global memory can incur a relatively long latency, it is imperative to completely utilize the full memory bus width to deliver as much data as possible in the smallest number of global memory transactions to the compute units.

\(^1\)The Fermi architecture [50] from NVIDIA incorporates a cache hierarchy for global-memory as well.
CHAPTER 2. BACKGROUND AND RELATED WORK

Figure 2.4: Simplified diagram of a modern GPU memory subsystem. GPU memory is composed of on-chip and off-chip memories. On-chip memory includes caches, software-managed local scratchpad memory, and registers. Off-chip memory includes global, texture, and constant memories. The detailed hardware implementation and the characteristics of each memory space varies by vendor. The visibility and accessibility of each memory space can vary across different GPU programming models.

As such, performance is very sensitive to the resulting data access pattern when working with global memory. Ideal access to global memory occurs when a scheduled group of threads request data in the same address range, allowing requests to be combined into a few accesses that fully utilize the memory bus—thread groups that are scheduled together are called a thread batch\(^2\) and combining memory requests is known as coalescing. When memory access patterns do not fully exploit the properties of global memory, another memory space is often more desirable to use [1, 51, 52].

**Constant memory** is an off-chip memory that is cached on-chip. As the name suggests, this memory space is read-only and can hold only a small amount of constant data (or non-modifiable input data) used by all threads across all compute units. The single-banked cache of constant memory has broadcast capability, and thus the

\(^2\)A thread batch is called *wavefront* or *wave* on AMD platforms and *warp* on NVIDIA platforms. The size of a thread batch is 64 threads on an AMD HD 5870 GPU and 32 threads on a NVIDIA GTX 285 GPU. Memory accesses are issued in units of half-thread batches. GPU programming languages have not yet included explicit representations of thread batches.
bandwidth of constant memory is maximized when all threads in a thread batch read
the same memory address. Once cached on-chip, data access latency is as fast as
a register-based access, though the throughput of the cache is decreased by a factor
equal to the number of different requests within a thread batch. Note that the inherent
properties of constant memory are not appropriate for every pattern associated with
read-only data; constant memory is best utilized when all threads of a thread batch
access the same data element simultaneously.

Texture memory is an abstraction where global memory is accessed through an
on-chip hardware texture unit. It is designed and optimized for graphics texture
mapping (naturally exploiting the 2D locality present in graphics images). Texture
memory offers a number of benefits over global memory: 1) it is cached on-chip, 2) it
provides better performance for uncoalesced accesses, and 3) it has hardware support
for address calculations, automatic boundary checking, and data interpolation. Ran-
dom memory access patterns are better served by texture memory than with global
memory \(^3\) [1].

Local memory is a small, on-chip scratchpad memory on each compute unit—it is
referred to as Local Data Share (LDS) on AMD platforms and as shared memory on
NVIDIA platforms. Utilizing local memory is key to harnessing the full computing
power of the GPU [1, 19, 21, 52]. Given the large number of processing elements within
a GPU, the long-latency accesses to off-chip memory are commonly the bottleneck
of a compute kernel. To achieve high performance, it is necessary to load data into
low-latency local memory as much as possible (we use the term prefetch to refer to
this usage throughout this thesis). Since memory space is very limited in size and
partitioned by active work groups simultaneously running on a compute unit, the
amount of local memory used is one of the key factors that determine the number
of work groups that can reside on a compute unit simultaneously. Nevertheless,
the use of local memory significantly increases software development complexity. It
requires programmers to recognize the potential benefit from data prefetching and

\(^3\)This property changes in devices supporting caching for global memory access.
then manually partition the data. Note that local memory is only beneficial when there is significant data reuse among threads within a work group to compensate for the extra cost of explicitly loading data from off-chip memories.

Registers and Private Memory are used to store automatic variables within a kernel. Compute units have a large number of registers that must be statically allocated to a thread for its entire lifetime. Local arrays and any data that does not fit into registers are spilled into private memory\(^4\), which is located off-chip as a part of global memory. Both registers and private memory are only visible to individual threads. Just as local memory size is a constraint on performance, register allocation is a major limiting factor on the number of threads (or work groups) that can be active on a compute unit. Excessive use of registers (high register pressure) in a kernel limits the number of threads that can run simultaneously, which in turn exposes memory latency. The compiler plays a critical role in register usage [42]. Since neither of these memory spaces are explicitly programmable by the programmer, they are not considered during our memory space selection in this thesis. However, determining an appropriate work group size is highly dependent on register constraints, and so we still need to carefully consider register usage when designing our work group sizing algorithm in Chapter 8.

With the exception of on-chip memories, all off-chip memory spaces are persistent across kernel calls and are accessible (without consistency) by all threads executing a kernel. All memory spaces that are on-chip and any data that is cached are not visible to threads executing on different compute units. Table 2.2 and 2.3 summarizes the characteristics of the different GPU memory spaces just described.

\(^4\)Private memory can be assigned explicitly to reduce the impact of register pressure. However, careful attention should be given to this decision since the tradeoff between expensive off-chip memory access and memory latency hiding through lowering register pressure is difficult to predict.
CHAPTER 2. BACKGROUND AND RELATED WORK

<table>
<thead>
<tr>
<th>Memory space</th>
<th>Location</th>
<th>Typical Size</th>
<th>Cached (size(^\dagger))</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>512 MB to 4 GB</td>
<td>No</td>
<td>R/W</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>64 KB</td>
<td>Yes (8 KB)</td>
<td>R</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Up to global memory size(^\dagger)</td>
<td>Yes (6 to 8 KB)</td>
<td>R/W</td>
</tr>
<tr>
<td>Local</td>
<td>On-chip</td>
<td>16 KB to 64 KB(^\dagger)</td>
<td>N/A</td>
<td>R/W</td>
</tr>
<tr>
<td>Private</td>
<td>Off-chip</td>
<td>Up to global memory</td>
<td>No</td>
<td>R/W</td>
</tr>
<tr>
<td>Register</td>
<td>On-chip</td>
<td>16K to 32K</td>
<td>N/A</td>
<td>R/W</td>
</tr>
</tbody>
</table>

Table 2.2: GPU memory spaces. \(^\dagger\)Per compute unit. \(^\dagger\)With limitations per object.

<table>
<thead>
<tr>
<th>Memory space</th>
<th>Scope</th>
<th>Persistent</th>
<th>Managed by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>Kernel</td>
<td>Yes</td>
<td>Programmer</td>
</tr>
<tr>
<td>Constant</td>
<td>Kernel</td>
<td>Yes</td>
<td>Programmer</td>
</tr>
<tr>
<td>Texture</td>
<td>Kernel</td>
<td>Yes</td>
<td>Programmer</td>
</tr>
<tr>
<td>Local</td>
<td>Work group</td>
<td>No</td>
<td>Programmer</td>
</tr>
<tr>
<td>Private</td>
<td>Work item</td>
<td>No</td>
<td>Compiler</td>
</tr>
<tr>
<td>Register</td>
<td>Work item</td>
<td>No</td>
<td>Compiler</td>
</tr>
</tbody>
</table>

Table 2.3: GPU memory spaces (continued).

2.1.3 Two Variants: Vector-based (AMD) and Scalar-based (NVIDIA) GPUs

Given the requirement to satisfy the same graphics API specifications (e.g., DirectX [26] and OpenGL [25]), it is not surprising that the hardware architectures of AMD and NVIDIA GPUs are similar in nature. Both provide an array of multiprocessors\(^5\) that are designed for scalable data-parallel computation, and each multiprocessor is independent and has its own limited amount of hardware resources which are shared by active threads.

\(^5\)AMD and NVIDIA respectively call those SIMD engines and Stream Multiprocessors (SM).
The memory subsystems of both platforms are very similar as described in Section 2.1.2. Memory accesses on both platforms are satisfied in parallel using parallel threads. If the access pattern is well structured, the accesses result in only one or a few memory transactions for all threads in a thread batch. This grouping of memory accesses is called *coalesced memory access* and is crucial to achieve the maximum effective bandwidth for servicing memory operations. Therefore, it is usually the first optimization to consider on both GPU platforms.

In order to overcome memory bottlenecks, hardware vendors recommend that programmers follow a prescribed set of memory optimization techniques. *Vectorization* is a technique that assures higher memory utilization on vector-based architectures. *Memory coalescing* is a technique to avoid thread stalls due to serialized memory access on both vector-based and scalar-based architectures. To achieve high performance on GPUs, programmers adapt their kernels to follow these recommendations, though it may be impossible for many general-purpose applications to be rewritten in this way.

Next, we turn our attention to the differences between two commercial platforms, highlighting the distinct features of each platform.

**AMD Platform**: AMD GPUs employ a true SIMD vector architecture where both computation and memory operations are performed on up to four data elements at a time [37, 53, 54]. Modern data-parallel programming languages including OpenCL and Brook+ ⁶ [33] provide built-in short vector types that allow code to be explicitly tuned for the architecture. Short vector types include *float*, *double*, or *int* types of 2 to 4 elements long, with the size appended as a suffix (e.g., *float4*, *int2*, etc.).

Vectorization is a key memory optimization available on AMD platforms that can be exploited to maximize the utilization of the available memory bandwidth [2, 37, 54]. Memory lanes are based on 4-element vectors, and bandwidth may be wasted if this

⁶Brook+ was AMD’s past high-level general purpose GPU programming language before the introduction of OpenCL.
vector type is not used. Even though the compiler will attempt to map multiple scalar memory operations into a vector format, using larger vector types explicitly in high-level code will more directly aid the compiler to maximize the utilization of the available execution units and memory bandwidth. Furthermore, since each iteration of a loop is mapped to a thread, vectorization also decreases the total number of threads to be processed, resulting in a smaller number of threads to carry out the same amount of work. As such, vectorization has a profound effect on the performance of AMD GPUs.

Although AMD now only supports using the industry standard OpenCL (we will detail the OpenCL in Section 2.2.1) to harness both their CPU and GPU, the reader might be interested in their previous programming language, Brook+ [33]. Originating out of the BrookGPU [33] programming language, Brook+ offers C/C++-like programming style with a minimal set of extensions to expedite the adoption of their programming platform. It adopts a stream processing model which simplifies parallel hardware and software by restricting the kinds of parallel computation that can be performed. In fact, the Brook+ programming environment adopts uniform streaming, which implements a flat thread hierarchy. There is no concept of thread grouping, which reduces programming complexity significantly but lacks a way of expressing local hardware resources.

With respect to programmable memory spaces, AMD’s Brook+ programming model abstracts away much of the underlying memory subsystem, making the GPU memory spaces transparent to the programmer. Memory space mapping is primarily determined by the compiler, and thus any optimization of memory accesses on an AMD GPU platform should focus on effective use of vectorization with the Brook+ programming.

AMD also provides a low-level programming environment using AMD’s Intermediate Language (IL) [55] at the Compute Abstraction Layer (CAL) [34] to give more control over the underlying hardware architecture. Writing code in IL can lead to higher performance, but programming tends to be more tedious and time-consuming
since it requires a more detailed understanding of the ISA and the hardware architecture.

**NVIDIA Platform:** NVIDIA GPUs employ a scalar architecture in the sense that computation and memory operations are not necessarily executed in a vector fashion. The NVIDIA architecture is well-matched to parallel programs that are not vectorizable. Despite differences in NVIDIA’s and AMD’s underlying hardware architectures and execution models, the same memory utilization challenges exist on both architectures. For NVIDIA GPUs, exploiting memory coalescing and customized memory space mapping are key optimizations to apply in order to achieve high memory bandwidth [1, 56].

Although NVIDIA supports the OpenCL programming language, it is worthwhile taking a look at NVIDIA’s own general-purpose programming language and framework called C for CUDA [32] since this language was the first successful dedicated SDK that made GPU computing popular and contributed significantly to establishment of a vibrant general purpose GPU computing community.

From the beginning, C for CUDA (we will refer to this language as CUDA in this thesis) took a different approach than Brook+. At its core, CUDA presents a two-level thread hierarchy and full memory space mapping. This flexibility comes at a price—the programmer needs to carefully consider the specific thread configuration, along with appropriate hardware resource use. The programmer also needs to map each data structure into the most appropriate memory address space based on knowledge of the distinct characteristics of each memory space and the memory access pattern associated with each data entity. OpenCL later adopted this two-level thread hierarchy programming model, and it has become the de facto model for GPU computing. We will discuss this programming model and OpenCL in more detail in Section 2.2. Table 2.4 summarizes the salient features just discussed on the AMD and NVIDIA GPU platforms.

To reflect on these differences, the microarchitectural differences between the two platforms (i.e., vector-based vs. scalar based) certainly contributes to the fact that
one architecture can outperform the other for specific classes of applications. Each architecture possesses distinct strengths and weaknesses over the other, depending on the class of application that the architecture is intended to run. Vector architectures are simple yet powerful, but generally there is a limit on the effectiveness of the vector length; using a long vector length creates inefficiencies for algorithms that inherently use shorter or unaligned vectors. In contrast, scalar architectures provide programmers with increased programmability resulting in consistently high performance in a wider spectrum of general-purpose algorithms but typically exhibit lower performance than vector architectures when applications can be fully vectorized. From an application developer’s perspective, it is very important to know how well an application will map to the underlying hardware before making any decision on which platform to choose.

<table>
<thead>
<tr>
<th></th>
<th>AMD</th>
<th>NVIDIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base microarchitecture</td>
<td>Vector VLIW</td>
<td>Scalar</td>
</tr>
<tr>
<td>Programming language</td>
<td>Brook+†, CAL, OpenCL</td>
<td>CUDA, OpenCL</td>
</tr>
<tr>
<td>Execution model</td>
<td>Stream (Brook+), SIMT</td>
<td>SIMT</td>
</tr>
<tr>
<td>Programmable memory space</td>
<td>Uniform (Brook+), Multiple</td>
<td>Multiple</td>
</tr>
<tr>
<td>Thread hierarchy</td>
<td>Flat (Brook+), Two-level</td>
<td>Two-level</td>
</tr>
<tr>
<td>Hardware thread batch</td>
<td>Wavefront (64 threads‡)</td>
<td>Warp (32 threads)</td>
</tr>
<tr>
<td>Memory efficiency assurance</td>
<td>Vectorization, Coalescing</td>
<td>Coalescing</td>
</tr>
</tbody>
</table>

Table 2.4: Comparison between AMD and NVIDIA GPU computing platforms. † Brook+ is deprecated and no longer officially supported. ‡ The size of wavefront varies (16 to 64 threads) depending on machine variants.
CHAPTER 2. BACKGROUND AND RELATED WORK

2.2 GPU Programming Model

GPU computing adopts a heterogeneous computing model, where data-parallel, compute-intensive code is off-loaded onto the GPU (accelerator), and executed asynchronously with respect to execution on the CPU (host). For asynchronous and independent execution, all input data needs to be explicitly copied to the GPU prior to kernel execution, and results need to be copied back to the CPU for post processing. To get maximum benefit from this heterogeneous environment, loops containing data parallel computation need to be carefully identified and reprogrammed to target a data-parallel architecture [5, 6, 13, 35, 57].

High performance GPU computing also adopts a massively multi-threaded, data-parallel programming model. Each kernel is mapped to a data element and becomes a thread (also known as Single Program Multiple Data, SPMD, model). In other words, when mapping serial data-parallel loop nests onto GPUs, each iteration of the loop nest is mapped to a thread. Loop nests of more than one level are typically mapped to multi-dimensional thread configurations whose dimensions are less than or equal to the depth of the loops. The organization of multi-dimensional thread configurations is akin to multi-dimensional arrays in that threads in the lowest (finest-grained) dimension are adjacent to each other, while threads at higher dimensions are a fixed stride apart (the stride size depends on the extent of the lower dimensions).

A key feature and abstraction of the current scalable GPU programming model is a thread grouping or a two-level thread hierarchy. This abstraction not only provides a way of expressing fine-grained data and thread parallelism nested within coarse-grained data and task parallelism, but also provides the means to efficiently utilize per-multiprocessor resources such as local scratchpad memories and registers while maintaining the same level of hardware resource utilization.

Using the two-level thread hierarchy model (the model used in OpenCL and CUDA), programmers are required to chunk their workloads into equally-sized pieces

---

7A data-parallel loop nest is a set of for loops wherein a set of arrays are referenced using the associated loop iteration variables.
called thread groups (called work groups in OpenCL and thread blocks in CUDA) while judiciously considering the amount of local hardware resources that each work group consumes and the available hardware resources per compute unit. As a result, the kernel ends up consisting of multiple thread groups; e.g., a single loop iteration is equally divided into multiple smaller sub-iterations (similar to loop strip mining [58]). When executed, each thread group is assigned to a multiprocessor (called compute unit in OpenCL) by the hardware thread scheduler. Figure 2.5 describes how a kernel that is programmed with multiple thread groups of equal size is executed on GPU hardware. The thread groups are assigned onto one of the multiprocessors by hardware, until all thread groups are completed.

![Diagram of thread groups and their mapping to GPU hardware](image)

Figure 2.5: Thread groups and their mapping to GPU hardware. A kernel composed of a number of thread groups is executed on a GPU hardware in a way that its thread groups are assigned onto one of the multiprocessors by hardware scheduler until all thread groups are completed.

The number of thread groups that can be active is dependent on the availability of hardware resources. Since hardware resources in a multiprocessor are shared by multiple thread groups, the amount of hardware resources (e.g., local scratchpad...
memory and registers) that the thread group consumes can limit the number of active threads that are run simultaneously. This hardware contention produces a small number of active threads, resulting in low hardware occupancy and inefficient hardware utilization. Therefore, proper thread grouping is a key decision parameter in scalable GPGPU programming and at the same time one of the factors that make programming and optimization more difficult.

Figure 2.6 illustrates how thread grouping is performed and their mapping to multiprocessors. The same Figure also shows how the kernel scales as the number of cores changes; we see that it takes twice as long for a GPU with two multiprocessors to complete the entire kernel versus a GPU with four multiprocessors, though the same per-multiprocessor utilization is maintained. This scalable programming model provides true portability of applications across hardware that can present a different number of cores. A thread group is the basic building block of a scalable kernel design, and a multiprocessor is the basic building block of a scalable hardware architecture.

One of the key ideas behind the thread grouping concept is to enable the programmer to map data to local scratchpad memory resources. Achieving high performance on GPUs relies heavily on the effective utilization of local software-managed scratchpad memories [1, 2, 19, 38, 41] which is available on GPU architectures. Each multiprocessor block has local resources and acts as the basic building block for scalable hardware design. Therefore arriving at a thread grouping and designing an effective two-level thread hierarchy will continue to play a critical role in the efficient mapping of general-purpose applications on to GPU architectures.

### 2.2.1 OpenCL

Vendor specific GPU programming languages, such as Brook+ and CUDA, are limited in that they only execute on the vendor’s own hardware. This becomes an obstacle that prevents GPUs from becoming more widely accepted platforms. The GPU community had been looking forward to a more universal software development
Figure 2.6: Scalable GPU programming model [1]. The two-level thread hierarchy (i.e., thread grouping) maintains the same level of hardware utilization across both devices with different number of multiprocessors and different input problem sizes. Symbols, TG and MP, refer to thread group and multiprocessor, respectively.

programming environment.

OpenCL (short for Open Computing Language) [41] was introduced in 2008 by the Khronos group as an open industry standard for heterogeneous computing which is targeted to harness the full computing power of systems consisting of multi-core CPUs and other accelerators such as GPUs, the Cell processor, and DSPs. The OpenCL is, by design, a cross-vendor, non-proprietary solution for heterogeneous computing.

The OpenCL specification is made up of three main parts [41]: the language specification, the platform layer API, and the runtime API. The language specification describes the syntax and programming interface which is based on a subset of ISO
C99 with language extensions. The platform layer API gives the developer access to routines that query for the number and types of devices in the system. The developer can then select and initialize the necessary compute devices to properly run their workload. This layer takes care of context management, work-queues for job submission, and data transfer requests. Finally, the runtime API allows the developer to queue up compute kernels for execution and is responsible for managing the compute and memory resources in the system.

Just as in AMD’s and NVIDIA’s custom platforms, in OpenCL a kernel is a basic unit of executable code. Execution of a kernel can proceed either in-order or out-of-order, depending on a parameter passed to the system in OpenCL platform. The execution domain of a kernel is defined by an N-dimensional index space, each element in the space is called a work item, which is in turn grouped together into a work group, which is the work grain used for synchronization and communication.

OpenCL pursues a multi-level, multi-space memory model that is very similar to NVIDIA’s. It defines four memory spaces as we detailed in Section 2.1.2: 1) private, 2) local, 3) constant, and 4) global memories. Figure 2.7 shows a diagram of the memory hierarchy. Private memory is a memory space that can only be used by a single work-item and local memory is a memory that can be used by the work-items in a work-group. Local memory is similar to the Local Data Share (LDS) of AMD GPUs and the shared memory on NVIDIA GPUs. Constant memory is a memory space that can be used to store constant data for read-only access by all of the compute units in the device during the execution of a kernel. The host processor is responsible for allocating and initializing memory objects that reside in this memory space. Finally, global memory is a memory that can be used by all the compute units on the device. This is the off-chip GPU device memory that is available on modern GPUs. As the reader may already notice, the OpenCL memory model reflects the memory organization of modern GPUs.

A noteworthy feature of OpenCL is its ability to perform compilation: a developer
Figure 2.7: An OpenCL memory model. OpenCL memory model is very similar to modern GPUs'.

has the option of pre-compiling their OpenCL compute kernel (off-line compilation) versus having the OpenCL runtime compile their kernels on demand (on-line compilation). There is a trade-off of portability and compile time between these two options.

2.3 GPU Execution Model

When the kernel is executed on GPUs, thousands or millions of threads are generated. The number of total threads is usually dictated by the input size, though it depends on how the programmer maps threads to data, e.g., which data or which dimension of data is mapped to a thread. From a high-level programming point of view, it is a Single Program Multiple Data (SPMD) model—an instance of a kernel (i.e., a set of threads) run on different data points.

The threads to be executed enter into a queue and wait there until one of the compute units becomes available. When a compute unit is available, a work group
is assigned to it and executes on that compute unit until all threads in the work group finish executing. When a work group is executed, threads are grouped into hardware scheduling units that we refer to as *thread batches* (called *wavefront* or *wave* on AMD platforms and *warp* on NVIDIA platforms). The scheduled threads are ordered with consecutively increasing thread IDs (performed after linearization in the case of multi-dimensional thread configurations). All threads in a warp execute the same instruction—following the Single Instruction Multiple Thread (SIMT) execution model. The first and second halves of the thread batch (half wave or warp) interleave execution and issue memory instructions, thus a *half wave or warp* is the key thread unit dimension to consider in terms of memory access efficiency. The size of a wave or warp varies from 16 to 64, depending on underlying hardware architecture, but 64 threads on AMD and 32 threads on NVIDIA are used these two mainstream platforms. Figure 2.8 illustrates the hierarchy of threads.

![GPU execution model](image)

**Figure 2.8: A GPU execution model.** When executed, the hardware groups the threads of a work group into a batch of threads (typically 16 to 64 threads) and schedules those threads together. The size of the thread batch varies across hardware vendors and GPU models.
2.4 Related Work

In this section, we highlight some prior work in general purpose application mapping to GPUs. We begin by summarizing prior work that maps applications taken from a number of problem domains. We also discuss associated optimization techniques explored in these projects. We also discuss previous work on automatic optimization approaches.

2.4.1 Manual Mappings and Their Optimizations

Since the introduction of fully programmable GPU hardware and dedicated SDKs in 2006, a number of researchers from different domains have been reporting optimization techniques along with great speedups. The applications or algorithms span a range of scientific and engineering fields, include basic numerical routines such as the BLAS library [59], real world applications such as medical imaging [60, 61], physics and scientific simulations [62, 63, 64], and information technologies [65, 66, 67].

Volkov et al. [68] report impressive performance speedups for dense linear algebra using CUDA on NVIDIA GPUs. The resulting implementations outperformed vendor libraries and approach the peak throughput possible on the target hardware platform. The authors present a couple of optimization techniques aimed at increasing parallelism. Their report includes detailed benchmarking of the GPU memory system that reveals sizes and latencies of on-chip caches and the translation lookaside buffer (TLB). This work is a representative example that demonstrates how to exploit the full computational power that a GPU provides by performing tuning at a very low level.

Silberstein et al. [19] obtained a 2700× speedup over a CPU for computation of a sum-products kernel when run on an NVIDIA GPU. This impressive speedup is achieved through efficient use of the software-managed scratchpad memory present on the NVIDIA GeForce 8800 GTX hardware. They also present an analytical model
for performance analysis of the algorithm. They demonstrated that the use of on-
chip scratchpad memory is key for maximizing effective memory bandwidth and is
especially crucial in memory-bound applications.

Jiang et al. [20] automate the tuning process of a matrix multiplication kernel
by selecting the best implementation from multiple versions according to empirical
evaluation. The tuning strategy presented is similar to approach discussed in AT-
LAS [69], which employs parametrized code generators that can generate multiple
codes according to input tuning parameter values. Ino et al. [70] also utilize a similar
technique for generating an optimized LU decomposition kernel routine.

While much of early optimization-related work targets an NVIDIA platform, our
prior work [2] presents an optimization methodology that targets an AMD platform.
In this thesis we explore optimization spaces targeting three different computing re-
sources: ALU, memory, and threads. One of our main observations is that vector-
ization (i.e., using vector type variables) is the most effective memory optimization
on vector-based AMD GPU architecture. We present a statistics-guided optimization
techniques in this thesis.

In his thesis and associated papers, Ryoo discusses a number of strategies for
optimizing programs on a highly data-parallel architecture that exploits fine-grained
resource sharing [15, 16, 71]. He proposes a method where optimization spaces are se-
lected by examining the static code and prunes configurations that do not necessarily
lead to desirable qualities in isolation or in combination.

More recently, researchers reported the use of GPUs to increase processing through-
put of complex real-time systems. Vasiliadis et al. [67] present an intrusion detection
system that exploits the computational power of GPUs to off-load costly pattern
matching operations from the CPU. They present how all system resources can be
orchestrated to meet the requirements of complex real-time applications.
2.4.2 Automatic Approach

Motivated by the fact that GPU programming complexity poses a significant challenge for developers and hinders its wider adoption to general-purpose mainstream, researchers have been making substantial efforts on developing more automated programming techniques and frameworks.

One approach is to use directive-based high-level abstractions. This approach simplifies the task of porting sequential programs to GPUs. One of the earliest works that adopts this approach is hiCUDA [17] by Han et al. HiCUDA is a high-level directive-based language for CUDA programming for efficiently parallelizing sequential code. HiCUDA can reduce the amount of CUDA programming required, but still leaves the code-optimization burden as a task for the programmer.

Another successful work using this approach is PyCUDA [18] by Klöckner et al. They developed a runtime system and a wrapper that uses a widely-used high-level scripting language, Python. Thanks in part to the popularity of Python, this work received a lot of attention and has a large number of users.

Another promising approach is to generate source-to-source compilers. Baskaran et al. [22] developed a compile time transformation scheme which finds program transformations that can lead to efficient global memory access patterns. The proposed compiler framework optimizes affine loop nests using a polyhedral model. In [21], they develop an approach for automatic data management for on-chip shared memories, including creation of buffers in on-chip local memories using the same model. The polyhedral model has been used for shared memory parallel machines to improve locality and minimize communication cost [72]. While this popular model can be used to optimize per-multiprocessor scratchpad memory on a GPU, it lacks the ability to handle data-level optimizations (e.g., pattern-aware optimizations) and does not address how best to map data to different GPU memory spaces.

CUDA-lite [73] is another translator which generates code for optimal tiling of global memory data. CUDA-lite relies on information that a programmer provides
via annotations to perform transformations. However, automated transformation approaches typically suffer from low performance due to their inability to efficiently handle differences in memory architectures.

Another practical approach is to apply cross compilation. Lee et al. [74] present a compiler framework for automatic source-to-source translation of standard OpenMP applications targeting CUDA-based GPGPU applications. The goal is to make existing OpenMP applications targeted for shared memory parallel computers amenable to execution on GPUs. Lee’s work finds several key transformations needed for this translation. MCUDA [75] maps the CUDA programming model onto a conventional shared memory multi-core CPU architecture. MCUDA provides the ability to apply the CUDA programming model for developing data-parallel applications running on traditional shared-memory parallel systems. Diamos et al. [76] present Ocelot, a binary translation framework designed to translate Parallel Thread Execution ISA (PTX [77]) used on the NVIDIA platform to SPU assembly used by the IBM Cell Processor. Specifically, they show that the PTX thread hierarchy can be mapped to many-core architectures, explore translation techniques that can hide memory latency, and discuss data structures that can be efficiently emulated.

The cited previous work on cross-platform compilation frameworks tends to be fairly straightforward to implement, and a good option as compared to hand-coded translation from serial code to parallelized GPU code. This is because the original code is already parallelized and many of the associated parallelization tasks have been addressed. Another difficult issue unsolved with previously cited automatic approaches is their inability to find the best memory space mapping using an algorithmic approach. One good example is that tiling loops into smaller blocks is required to effectively use per-multiprocessor resources (e.g., scratchpad memory and registers) on GPU architectures, though automating this process is a non-trivial task.

Unlike manual optimization approaches, the success of any automated method relies heavily on underlying model used for optimization. Most existing automated approaches do not employ a standard model, nor are they directly applicable to GPU
architectures. In the following two sections, we briefly review the polyhedral model which has been used for GPU memory optimizations, and several other representative models.

2.4.3 Polyhedral Model

The polyhedral model is a powerful mathematical framework which is based on parametric linear algebra and integer linear programming. It provides an abstraction to represent nested loop computations and their data dependencies using integer points in a polyhedra. The polyhedral model is perhaps the most advanced model that has been developed for loop transformations and optimizations to date [72, 78]. It was developed and used originally to improve locality on a single core processor, and later extended to minimize communication costs on a multi-core processor.

The polyhedral model makes it possible to reason about the correctness of complex loop transformations in a completely mathematical setting, and by introducing cost functions, makes it possible to quantify the degree of data reuse and communication bandwidth required among processors. A number of previous research studies on loop locality optimization and automatic loop parallelization, targeting single or multi-core processors, have adopted this model [72, 78, 79, 80, 81].

Baskaran et al. describe how to use the polyhedral model for automatic parallelization and optimization targeting many-core GPUs [21]. They employ the polyhedral model to find efficient utilization of explicitly software managed local memory [21].

Although the polyhedral model can provide a comprehensive analysis of data access information on single and multi-core CPUs (where only a few simultaneous threads need to be considered), it lacks the ability to account for important memory access behavior taking place on massively multi-threaded data-parallel GPUs where a group of threads access memory simultaneously. (Section 2.1.1 details the fundamental differences between these platforms).

There are two reasons why we chose not to adopt the polyhedral model in this
thesis. First, the polyhedral model is not presently capable of capturing the impact of thread mapping. As we will see in Section 3.3 and 3.4, multi-dimensional thread mapping plays a critical role in determining the efficiency of memory operations in kernels and consequently affects performance significantly. Thread mapping is a key concept in GPU programming, and any optimization approach needs to consider the impact of the final thread mapping.

Second, the polyhedral model does not presently account for the characteristics of different memory spaces available on a system. The model is very good at expressing the degree of data reuse and communication volume required among data partitions. But GPUs have a number of memory spaces (the current polyhedral model would only make sense for considering local memory on GPUs). As we will see in Section 3.2, each GPU memory space has distinct characteristics, and effective utilization of these memory spaces by new optimization techniques can have a huge impact on performance.

Lastly and fundamentally, the polyhedral model is built upon inter-statement analysis as opposed to inter-data analysis. Today’s GPU programming and execution model is data-oriented—given that each data point is mapped to a thread, inter-data analysis can provide more information required for memory optimizations. We will present our own inter-data analysis (we refer to this as inter-thread behavior) in Section 4 and 5.

2.4.4 Other Models

A significant volume of research has been conducted on loop optimization in the 1990’s. Wolf and Lam [82, 83] proposed algorithms that improved the locality and parallel execution of a loop nest by applying loop transformations based on distance and direction vectors. Their model serves today as the theoretical basis for many traditional loop transformations such as loop interchange, skewing, reversal, and tiling. Wolf and Lam’s work also addressed the following compilation questions: which loop
transformations should be applied, and in what order, considering maximizing parallelism versus improving data locality. Their work considered various loop transformations as \textit{unimodular} transformations. They also introduced dependence vectors that capture both distance and direction information. The loop transformation problem can be formulated as directly solving for the unimodular transformation that maximizes some objective function, while satisfying a set of constraints.

In a later paper, Lim and Lam [84] proposed affine transforms to maximize parallelism and minimize synchronization overhead, specifically targeting multi-processors. Their algorithm is based on the concept of \textit{affine partitioning}, where instances of each instruction are identified by the loop index values of their surrounding loops, and affine expressions are used to map these loop indices to partition numbers which are used for two different purposes: space and time partition. A space partition number indicates the processor number on which operations with the same space numbers are executed, while the time partition number dictates the execution sequence on a single space partition.

Feautrier [85, 86] proposed a piece-wise affine scheduler that minimizes the overall execution time of a program and tries to minimize communication after loop parallelization. For each statement in a loop body, the model finds a piece-wise affine mapping that maps its loop indices to a multi-dimensional time index. The model then uses a heuristic based on parametric integer programming to minimize the dimensionality of time. Minimizing the dimensionality of time corresponds to maximizing the degree of parallelism.

The models mentioned above were developed around very similar goals (i.e., improving locality and maximizing parallelism with minimal communication overhead). Most of the other loop-optimization approaches are superseded by the polyhedral model in terms of the degree of their optimization capability. Just like the polyhedral model (i.e., absence of thread mapping, accounting for the characteristics of different memory spaces, and data-level analysis), the alternative models could be considered improper to address memory inefficiency issues present on massively data-parallel
many-core GPUs.

2.4.5 Other Research Trends

**GPU Virtualization:** Hardware virtualization is a mainstream technology issue for the data center. By hiding the physical characteristics of the underlying hardware from users, a number of features can be provided that include resource sharing and increased hardware utilization. Although virtualization technologies have not been widely adopted in high performance GPU computing due to their proprietary programming models, heterogeneity, and the overhead associated with indirect access to physical resources, it is a potential research path being studied in order to better utilize GPU hardware.

Shi et al. [87] describe a GPGPU solution in VMs called *vCUDA* which allows applications running within VMs to leverage GPU acceleration. The key idea behind their implementation is an API call interception and redirection. Their results show that GPU acceleration for HPC applications in VMs is feasible and competitive with those running in a native, non-virtualized environment. Gupta et al. [88] propose a very similar approach and present their implementation on a Xen-based hypervisor called *GViM*.

**Architectural Simulations:** As GPU computing continues to grow in popularity, researchers have also turned their attention to exploring architectural enhancements and simulation tools. However, due to the closed nature of most GPU architectures, there are a limited number of simulators and associated tools, especially as compared to the number developed for CPUs.

Fung et al. [44, 89] present a simulation infrastructure called GPGPU-Sim, which models a modern GPU architecture and is capable of running applications written in CUDA and OpenCL. Using their simulator, they explore dynamic warp formation and scheduling for efficient control flow handling. They also present a new memory interconnection network arbitration scheme, and characterize the performance impact of
several microarchitectural design choices using general-purpose applications written
in CUDA [90]. They report that performance is more sensitive to interconnect bi-
sectional bandwidth (rather than latency) for the class of applications studied. They
found if they ran fewer thread groups concurrently, they could improve performance
by reducing contention in the memory system.

Diamos et al. [91] extend their original PTX translator, Ocelot, into a functional
PTX emulator and use it to characterize PTX kernels. They report on an analysis
of a wide range of CUDA-based kernels possessing different workload characteristics
that impact control flow, data flow, degree of parallelism, and memory behavior.
Their results highlight the importance of different optimization issues such as branch
reconvergence and inter-thread sharing, and highlight opportunities for additional
parallelism.
Chapter 3

Motivation: Memory Efficiency and Performance

In the previous two chapters, we discussed how memory operations have become the most significant bottleneck on GPUs and that programmers should carefully optimize the memory access behavior of their code in order to exploit the full computational potential from GPUs. In this chapter we show, using example benchmark kernels and actual performance numbers, that memory efficiency can significantly impact the performance of general-purpose applications running on massively parallel GPUs. Addressing memory inefficiencies is the main focus and motivation of this thesis work. In each of the following sections we present the impact of different memory optimizations on performance. The benchmark applications we use in this chapter include conventional GPGPU benchmarks and real-world applications. We also present results for applications run on both AMD and NVIDIA commodity GPUs.

3.1 Impact of Vectorization on Performance

Vectorization has always been a very important optimization technique in high performance computing. This programming technique modifies the executable program
to better utilize all computational units and associated memory subsystem, resulting in greatly improved parallel hardware utilization and performance.

The impact of vectorization on performance is even more significant on vector-based GPUs (e.g., AMD GPUs) than on vector-based CPUs. On GPUs, vectorization not only reduces the number of expensive off-chip memory accesses by a factor of up to the vector length (typically by a factor of 4), but also decreases the total number of threads to be completed by up to the same factor when the threads map to vector data. Next, we show the impact that vectorization can have on the performance of AMD GPUs.

The benchmark kernel we use in this section is *matrix multiplication*. Matrix multiplication is one of the most commonly-used subroutines in numerical analysis and scientific applications. It is also the most popular benchmark kernel used in GPGPU research as it exhibits many interesting characteristics that can illustrate the true power of GPU programming and optimizations.

The original serial version of the matrix multiplication kernel is shown in Listing 3.1. We implemented both scalar and vector versions of this kernel to compare the impact of vectorization on performance.

```plaintext
for(i1=0; i1<M; i1++)
  for(i2=0; i2<N; i2++)
    for(i3=0; i3<P; i3++)
      C[i1][i2] += A[i1][i3] * B[i3][i2];
```

Listing 3.1: *Serial matrix multiplication loop nest*. Loop indices of the loop nest of depth 3 access three two dimensional arrays. We use this conventional GPGPU benchmark kernel in a number of places throughout this thesis.

Figure 3.1 compares the performance of the matrix multiplication kernel with and without vectorization. Although there is an overhead associated with performing data transformations to vectorize this kernel, we clearly see the performance benefits from
the vectorization—the performance increase becomes more apparent as the input size increases. All experiments are conducted on an 2.66GHz Intel Core 2 Duo running 64-bit Linux, with 2 GB main memory and an AMD HD 5870 GPU.

![Graph](image)

**Figure 3.1:** Performance impact of vectorization in matrix multiplication kernel on an AMD platform. The performance impact of other optimizations can be found in our earlier work [2].

Table 3.1 lists the machine code statistics of both the scalar and vector versions. These results show that without applying any memory optimizations, we only obtain a quarter of the available bandwidth per memory fetch. Vectorization increased the utilization of the texture unit to 100%. Even though there is added overhead on the CPU when transposing the second matrix, this overhead is easily amortized by the speedup obtained on the GPU. This benefit becomes more significant as we increase the size of the matrix. Vectorization is a very effective memory optimization technique for vector based AMD GPUs. We will present a more generalized approach to vectorization that is developed in this thesis in Chapter 6.
Table 3.1: Machine code statistics of scalar and vector version of matrix multiplication kernel. The machine code statistics are obtained from AMD Stream Kernel Analyzer [4]. † The ratio between ALU and memory instruction count. ‡ General Purpose Registers.

### 3.2 Impact of Memory Space Selection on Performance

Next, we look at the impact of memory space selection on performance. As we introduced in Section 2.1.2, kernels running on GPUs can access data from multiple memory spaces. Since each memory space possess different characteristics, it is important for programmers to map data to the right space based on each data structure’s associated memory access patterns. Careful mapping of data to the most appropriate memory space can have a huge impact on overall performance.

In this section, we again use the matrix multiplication kernel shown in Listings 3.1 to show the impact of memory space on performance. We experimented on an NVIDIA GTX 285 GPU. The matrix multiplication kernel exhibits a significant amount of data reuse (rows of the first matrix and columns of the second matrix are reused a number of times to compute one element in the output matrix) and so local scratchpad memory is the best memory space for storing the matrix. Figure 3.2 shows the performance in milliseconds with and without using on-chip local scratchpad memory. The baseline implementation uses the default global memory space. As we clearly see from the graph, utilizing appropriate memory space can make a huge difference in overall performance by maximizing the effective memory bandwidth (an order of magnitude better performance for most input sizes in this case). Utilizing
CHAPTER 3. MOTIVATION: MEMORY EFFICIENCY AND PERFORMANCE

these memory spaces properly is a key memory optimization technique.

![Graph showing performance impact of memory space selection in matrix multiplication kernel on an NVIDIA platform.](image)

**Figure 3.2:** Performance impact of memory space selection in matrix multiplication kernel on an NVIDIA platform. Efficient utilization of local memory boosts the performance by an order of magnitude over the default memory space (global memory). Appropriate memory space selection based on memory access pattern is a key to maximize effective memory bandwidth of GPUs.

Next, we look at the performance of a real application in order to further motivate the need for improved memory space mapping/layout. In work with Massachusetts General Hospital (MGH), we implemented forward and backward projection routines in CUDA for NVIDIA GPUs [2, 38]. Forward and backward projections are two fundamental building blocks in Iterative Reconstruction Techniques (IRT) for medical imaging [92, 93]. IRTs compute an iterative sequence of forward projections and corrective back projections until a satisfactory image is obtained. This approach is very computationally demanding but IRT produces superior images compared to traditional reconstruction approaches such as Filtered Back Projection (FBP) [94]. Reducing the image reconstruction time of IRT is key for making this technique
adoptable in practice.

Figure 3.3 shows the performance of the forward and backward projection kernels in terms of execution time across different optimization techniques explored. The leftmost bar is the execution time of the baseline GPU implementation and the second bar shows the execution time when appropriate memory spaces are selected. The rightmost bar shows the result when data restructuring is applied on top of memory space optimization. Here, we again clearly see the large impact of memory space selection on performance in both cases. Appropriate memory selection is one of the most effective memory optimization techniques on GPU platforms, but it is sometimes a challenging task for programmers to make the right selection.

![Figure 3.3](image)

(a) Backward projection. (b) Forward projection.

Figure 3.3: Performance impact of memory space selection in forward and backward projection routines on a NVIDIA GeForce 8800 Ultra GPU. Utilizing memory spaces is shown to be the most effective optimization in these kernels.

Although the characteristics of each memory space can be found in the programming guide [1, 51, 52] provided by each hardware vendor, it is still challenging for programmers to know when and how to use each memory space. In Chapter 7, we will show how to approach this challenge and present an algorithmic technique that
3.3 Impact of Thread Mapping on Performance

When evaluating how best to map data on to memory spaces on a GPU, we need to understand the associated memory access pattern. Each pattern produced by a kernel is determined by two factors: the original loop structure (algorithm) and the thread mapping. Thread mapping, which is determined by the programmer, plays a central role in determining the memory access behavior and thus overall performance. In this section we show the impact of proper thread mapping on performance. Chapter 8 presents our implementation to thread mapping.

Consider again the serial matrix multiplication data-parallel loop nest shown in Listing 3.1. The loops iterate over three two-dimensional arrays (i.e., $A$, $B$, and $C$). Intuitively, we would like each thread in the GPU kernel to compute a single value in the output array. To accomplish this mapping, we map the two outer loop iterators ($i_1$ and $i_2$) to a two-dimensional thread configuration. We have two choices to consider here:

- Mapping $\alpha$: map $i_1$ to the lower dimension of the thread configuration (labeled $tx$ throughout this thesis) and $i_2$ to the higher dimension of the thread configuration (labeled $ty$ throughout this thesis), or

- Mapping $\beta$: map $i_1$ to $ty$ and $i_2$ to $tx$.

These two mapped kernels are shown in Listing 3.2 and Listing 3.3 respectively. Figure 3.4 gives a graphical view of these two thread mappings.

As Listings 3.2 and 3.3 show, the thread mapping changes the memory access pattern (e.g., $C[tx][ty]$ in mapping $\alpha$, versus $C[ty][tx]$ in mapping $\beta$) and this can have a huge impact on overall performance. Figure 3.5 compares the performance of the two mappings on an NVIDIA GT200 architecture (GeForce GTX 285) for a range of input sizes; we see that performance differs by an order of magnitude. We
CHAPTER 3. MOTIVATION: MEMORY EFFICIENCY AND PERFORMANCE

will show how this performance sensitive thread mapping is represented in our model in Chapter 4 and how the model is used to derive optimized thread mappings in Chapter 8.

Listing 3.2: Thread mapping $\alpha$. The outermost loop iterated by $i_1$ maps to $tx$ and the middle loop iterated by $i_2$ maps to $ty$.

```c
int tx = get_global_id(0);
int ty = get_global_id(1);
for(i3=0; i3<P; i3++)
    C[tx][ty] += A[tx][i3]*B[i3][ty];
```

Listing 3.3: Thread mapping $\beta$. The outermost loop iterated by $i_1$ maps to $ty$ and the middle loop iterated by $i_2$ maps to $tx$.

```c
int tx = get_global_id(0);
int ty = get_global_id(1);
for(i3=0; i3<P; i3++)
    C[ty][tx] += A[ty][i3]*B[i3][tx];
```

3.4 Impact of Work Group Sizing on Performance

As described in detail in Section 2.2, in order to enable scaling of the number of compute units while maintaining the same level of hardware utilization, GPUs impose the requirement that compute units are autonomous execution units (i.e., no direct communication is possible between compute units). Each compute unit has the same amount of local hardware resources on which only threads (or work groups) assigned to it can operate. Synonymous with compute units, the programming model divides threads into independent groups (called work groups in OpenCL), each of which is identical in the number of threads and resource usage, so as a problem scales in
size, additional work groups are created, but no other changes to the algorithm are required. These two concepts, work groups and compute units, are the enablers of scalability in GPU computing on the software side and hardware side, respectively.

Optimization challenges arise due to the fact that per-compute-unit hardware resources determine the number of work groups and threads that can run simultaneously. Programmers need to carefully consider the work group size while factoring in these hardware resources to maximize hardware utilization.

To understand the impact of work group size on performance, we again consider the matrix multiplication kernel taken from the NVIDIA SDK and use local memory to test a range of work group size dimensions [95]. Figure 3.6 illustrates the

---

Figure 3.4: *Two possible thread mappings of matrix multiplication kernel.* (a) The outermost loop iterated by \( i_1 \) maps to \( tx \) and the middle loop iterated by \( i_2 \) maps to \( ty \), (b) The outermost loop iterated by \( i_1 \) maps to \( ty \) and the middle loop iterated by \( i_2 \) maps to \( tx \).
Figure 3.5: Impact of thread mapping on performance on a NVIDIA GTX 285 GPU. The changes in memory access pattern between two thread mappings make huge performance difference.

The changes in memory access pattern between two thread mappings make huge performance difference. The performance of the kernel when the work group size is 512 ($32 \times 16$ (ty) \times 16 (tx), upper-left corner in the graph) is approximately 5.9\times better than when it is 32 ($2 \times 16$ (tx), bottom-left corner). Even given the same work group size, 512 for example, the performance differs by 2.6\times between two different configuration ($32 \times 16$ (tx), upper-left corner and ($2 \times 256$ (tx), bottom-right corner).

Finding a good work group size is not intuitive, and is very complex when programming with local memory. Static constraints such as the maximum number of active threads and work groups allowed on a single compute unit, and the maximum number of threads per work group must be considered along with the register usage per thread and the local memory usage per work group. Changing the size or dimensions of a work group may end up resulting in very different performance. Therefore, in this thesis, we attempt to assist the programmer by suggesting a thread
Figure 3.6: Map view of the performance of matrix multiplication kernel across different combinations of work group size configurations. The work group sizing is a multi-variables optimization problem. The shade of gray indicates the execution time (the darker, the shorter execution time).

mapping and work group size that best utilizes the underlying hardware by means of static memory access pattern analysis. In Chapter 8, we present an algorithm that recommends sizes and dimensions that are likely to perform well for a given work group.
Chapter 4

Memory Access Modeling

In this chapter, we present a mathematical model that captures the memory access pattern information present in data-parallel loop nests. This proposed modeling, along with the static analysis in the following chapter, form the theoretical backbone of this thesis, and are used for driving a series of algorithmic memory enhancements and optimization techniques.

The memory access patterns associated with applications running on massively multi-threaded, data-parallel GPU architectures are a result of a combination of the inherent structure of each loop nest and the thread mapping. The loop structure is generally determined by the underlying algorithm, while the thread mapping is selected by the programmer. Given a particular thread mapping, the memory access pattern for each array present in a serial data-parallel loop nest can be captured into a mathematical model that can be used to drive a range of algorithmic optimizations.

4.1 Serial Model without Thread Mapping

The model we use is a vector affine form, which is a well-known representation in linear algebra and finds a number of applications in engineering mathematics [96]. The vector affine form is simple but powerful enough for us to capture all necessary
information related to memory accesses present in loop nests (including the thread mapping, which is key in modern data-parallel GPU programming models). One of the applications of the affine transformation has been loop transformation optimizations, where it was used extensively in the late 80s and 90s [82, 83, 84]. At that time, most work targeted improving locality on a single core [97], and later minimizing data synchronization on multi-processor systems [83, 84].

Consider a data-parallel loop nest of depth $D$ that accesses an $M$-dimensional array. The memory access pattern of the array can be represented as a memory access vector, $\vec{m}$, which is a column vector of size $M$ that describes how each dimension of the array is traversed. For example, in the matrix multiplication kernel in Listing 4.1, matrix $A$ is a 2-dimensional array where the first dimension is traversed by $i1$ and the second dimension is traversed by $i3$; the corresponding memory access vector is represented as

$$\vec{m}_A = \begin{bmatrix} i1 \\ i3 \end{bmatrix}$$

Although the concept is simple, significant insight can be gained by decomposing the memory access vector to its affine form as follows.

$$\vec{m} = M\vec{i} + \vec{o}$$

where $M$ is a memory access matrix whose size is $M \times D$, $\vec{i}$ is an iteration vector of size $D$ iterating from the outermost to innermost loop, and $\vec{o}$ is a column vector of
size $M$ that denotes the starting offset in each dimension of the array. Note that we only consider loops with array accesses that can be represented as affine functions of loop indices and symbolic variables (e.g., height, width, etc.). We have found that this restriction does not limit us since most scientific applications involve loops possessing affine access patterns [98].

The number of rows in the memory access matrix, $M$, is equal to the total number of dimensions in the array, and the number of columns is equal to the total depth of the loop nest. For example, in the matrix multiplication example, matrix $A$ is a two-dimensional array, so $M_A$ has two rows. Since the matrix multiplication algorithm has three nested loops, $M_A$ has three columns. The memory access matrix for array $A$ is represented as follows:

$$M_A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Each column in the memory access matrix, $M$, represents the memory access pattern of the corresponding loop level, and the number of columns is dictated by the depth of the loop nest. Each row in $M$ and $\vec{a}$ represents the memory access pattern of a dimension of the array, and the number of rows is dictated by the number of dimensions in the array. Non-zero entries in the memory access matrix specify which loop levels are iterating over the dimensions of an array.

The iteration vector is a column vector containing the iterators at each level of the loop nest. In the matrix multiplication example the iterators are $i1$, $i2$, and $i3$, going from the outermost loop to the innermost loop. So, the iteration vector of array $A$ is represented as:

$$\vec{i}_A = \begin{bmatrix} i1 \\ i2 \\ i3 \end{bmatrix}$$

Finally, the offset vector has one row for each dimension of the array. If any access of a dimension has an offset it will appear at the corresponding row of the
offset vector. Accesses to array A from the matrix multiplication example always begin at zero, so the offset vector is a vector of zeroes (zero vector).

$$\vec{o}_A = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The memory access patterns of all arrays in the serial matrix multiplication loop nest shown in Listing 4.1 are captured as follows:

$$\vec{m}_A = \begin{bmatrix} i_1 \\ i_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\vec{m}_B = \begin{bmatrix} i_3 \\ i_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\vec{m}_C = \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Assuming that arrays are stored in row-major order, the physically contiguous row elements of array A are accessed linearly as inner loop $i_3$ iterates, and the column elements of array A (physically separated by a non-unit stride) are accessed by the outer loop $i_1$. Similar representations are provided for arrays B and C. In all cases, memory accesses of all array dimensions start from the first element so the resulting offset vectors are zero vectors.

### 4.2 Parallel Model Incorporating Thread Mapping

In the current data-parallel programming model that was introduced in Section 2.2, one or more of loop iterations in a serial loop nest are mapped to each dimension of
the thread configuration when parallelized. In this section we show how this thread mapping is represented in our proposed model.

Suppose that we decide to map the two outer loops \((i1 \text{ and } i2)\) of the matrix multiplication kernel shown in Listing 4.1 to a two-dimensional thread configuration (i.e., each iteration of the outer two loops maps to each dimension of 2D thread map). This means that the first two columns in the memory access matrix represent the memory access pattern among the threads (referred to as \textit{inter-thread memory access pattern}) and the third column represents the memory access pattern within each thread (referred to as \textit{intra-thread memory access pattern}). Figure 4.1 illustrates this memory access pattern partitioning and it is of particular importance to understanding the memory access behavior in terms of thread mapping and work group configuration. This information is essential to appropriate memory selection and thread configuration optimization as we will show in Chapter 8.

In our representation, thread mapping simply replaces iteration indices with thread indices. We use \(tx\), \(ty\), and \(tz\) to denote each thread dimension, starting from the lowest dimension.\(^1\) In the two-dimensional thread mapping of the matrix multiplication kernel, for example, the two outer loop iterations out of three are mapped to \(tx\) and \(ty\). There are two mapping choices in this case. After thread mapping, the memory access patterns for arrays A, B, and C are captured as follows (we call these two thread mappings \(\alpha\) and \(\beta\) and their respective kernel code can be found in Listing 3.2 and Listing 3.3). Note that the parentheses are used to denote thread mapping \(\beta\).

\[
\vec{m}_A = \begin{bmatrix} tx (ty) \\ i3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} tx (ty) \\ ty (tx) \\ i3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}
\]

\[
\vec{m}_B = \begin{bmatrix} i3 \\ ty (tx) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} tx (ty) \\ ty (tx) \\ i3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}
\]

\(^1\)The current GPGPU programming models allow up to three-dimensional thread mapping.
\[
\bar{m}_C = \begin{bmatrix} tx (ty) \\ ty (tx) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} tx (ty) \\ ty (tx) \\ i3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}
\]

Figure 4.1: Memory access pattern partitioning of matrix multiplication kernel after thread mapping. The columns of the memory access pattern associated with thread indices represent inter-thread memory access patterns and the rest of the columns represent intra-thread memory access patterns.

The thread mapping classifies memory access patterns into inter-thread and intra-thread patterns. The inter-thread memory access patterns tell us how thread dimensions are mapped to array dimensions. We collect this information by extracting the columns of the memory access matrix that are accessed by thread indices. For example, the inter-thread memory access patterns associated with arrays A, B, C in the matrix multiplication example are composed of the first two columns of their memory access matrices, respectively:

\[
M_A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad M_B = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad M_C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}
\]
In thread mapping $\alpha$, the “1” in left upper corner of the inter-thread memory access matrix for array A indicates that the highest dimension of the array (the row dimension) is accessed by threads in the lowest dimension of the thread map. In other words, the first column of the memory access matrix corresponds to $tx$ when performing the matrix multiplication.

In thread mapping $\beta$, the same “1” indicates that the highest dimension of the array is accessed by threads in the highest dimension of the thread map, $ty$. Later we will show how the inter-thread memory access pattern can be used to optimize memory accesses on the GPU.

The intra-thread memory access patterns are the columns of the memory access matrix that correspond to iteration indices that are not mapped to threads. In the matrix multiplication example, the intra-thread access patterns are represented by the third column of each memory access matrix:

\[
\begin{bmatrix}
0 \\
1
\end{bmatrix}
\begin{bmatrix}
i3
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 \\
0
\end{bmatrix}
\begin{bmatrix}
i3
\end{bmatrix}
\]

\[
\begin{bmatrix}
0 \\
0
\end{bmatrix}
\begin{bmatrix}
i3
\end{bmatrix}
\]

for arrays A, B, and C, respectively. These patterns indicate how the threads, that are mapped by the inter-thread memory access pattern, actually access the data. For example, the lowest dimension of array A (represented in the second row of its memory access matrix), is accessed by $i3$, meaning that consecutive data elements will be accessed by each thread. array B, on the other hand, is accessed by $i3$ in it’s highest dimension, so individual threads will access elements that are a stride apart.
Finally, array C does not use $i^3$ as an iterator, so the data that array C is accessing will not change within the third loop.

Figure 4.1 summarizes the memory access pattern partitioning of the matrix multiplication kernel after thread mapping and Figure 4.2 illustrates a more detailed graphical representation. In Figure 4.2, threads ranging from $(0, K)$ to $(M-1, K)$ where $0 \leq K < N$ access elements of array A ranging from $(L, 0)$ to $(L, P-1)$ where $0 \leq L < N$ in the same linear pattern. Similarly, threads ranging from $(Q, 0)$ to $(Q, N-1)$ where $0 \leq Q < M$ access elements in array B ranging from $(0, R)$ to $(P-1, R)$ where $0 \leq R < N$ in the same non-unit stride pattern. Array C exhibits a slightly different access pattern. Since the inter-thread representation indicates a one-to-one mapping to an element of the array and there is no intra-thread access pattern, all threads have the same memory access pattern.
Figure 4.2: Graphical representation of memory access pattern partitioning of matrix multiplication kernel after thread mapping. The two-dimensional thread map and physical memory layout are shown to help illustrate the inter-thread and intra-thread memory access patterns.
Chapter 5

Memory Access Analysis

Having presented a serial and parallel mathematical model that captures memory access information in the previous chapter, we demonstrate the utility of our model in this chapter through a comprehensive analysis of different memory access patterns and their representations in our model. We first classify the memory access patterns and how they are represented in the model. We then present how important GPU memory access characteristics (e.g., coalesced accesses, accesses to the same memory address, and data reuse) are represented in the model.

5.1 Basic Pattern Classification

We begin with a basic classification of memory access patterns that does not consider the effect of thread mapping. This basic classification is still valid in post-thread-mapping analysis, and, in particular, plays an important role in our vectorization work presented in Chapter 6. Figure 5.1 presents the summary of our basic memory access pattern classification, along with illustrative examples.

Linear and reverse linear memory access patterns refer to accesses in which a dimension of an array is contiguously accessed with respect to the iteration space...
CHAPTER 5. MEMORY ACCESS ANALYSIS

Figure 5.1: Basic memory access pattern classifications and their representations in the model [3]. Note that the element being accessed is shown in gray, $C$ is a constant, and $Z$ is a random number.

(first and second diagrams in Figure 5.1, respectively). In the memory access matrix, accesses in the same direction as the loop iteration space (linear pattern) are represented by “1”, and accesses in the opposite direction (reverse linear pattern) are represented by “−1”. The offset vector is a zero vector in both cases. Most level 1 and level 2 Basic Linear Algebra Subprograms (BLAS) [59] possess one of these two memory access patterns.

A shifted memory access pattern refers to an access in which a dimension of an array is contiguously accessed with respect to the iteration space (as in the linear pattern), but this contiguous access is shifted by some constant. The memory access matrix for this pattern is the same as the linear pattern, but now the offset vector is a non-zero vector. This is shown in the third diagram in Figure 5.1. This pattern is commonly found in multimedia applications where multiple data streams need to be
merged or transformed.

An *overlapping* memory access pattern (the fourth diagram in Figure 5.1) refers to an access in which a dimension of the array is accessed by more than one iteration space. This pattern is represented as more than one non-zero value in a row in the memory access pattern. This pattern is common in linear equation solver applications using LU decomposition where triangular matrices need to be traversed.

A *non-unit stride* memory access pattern refers to a pattern in which a dimension of an array is accessed with a non-unit stride (shown in the fifth diagram in Figure 5.1). In the memory access matrix, this pattern is represented by a non-unit constant element.

A *random* memory access pattern refers to a pattern in which a dimension of an array is accessed in a random fashion. Memory accesses that are indexed by another array usually fall into this category.

Note that these patterns can be combined, resulting in compound patterns. For example, the pattern that combines reverse linear and shifted accesses is represented as $M = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$, $\bar{o} = \begin{bmatrix} 0 \\ C \end{bmatrix}$ in the model.

### 5.2 Extended Pattern Classification

In previous section, we classified memory access patterns into a number of categories: *linear*, *reverse linear*, *shifted*, *overlapping*, *non-unit stride*, and *random* patterns. In this section we extend our analysis to capture the effect of thread mapping. This extension plays a critical role in finding optimized thread mappings and work group sizes.

We explained in Section 2.3 that threads in a thread group are automatically grouped into a thread batch by hardware when actually executed on the GPU. The choice of thread mapping changes this thread batch grouping and changes the inter-thread memory access pattern. In order to represent the impact of thread mapping
on memory access patterns, we subdivide each of our memory access pattern classifications into true and false subpatterns. For example, the linear pattern is divided into true linear and false linear patterns.

Consider array C in the matrix multiplication kernel shown in Listing 4.1. As described in Section 4.2, the memory access pattern of the array changes (i.e., C[tx][ty] or C[ty][tx]) depending on the two thread mappings (α and β). These two memory access matrices are represented using our parallel model as follows:

$$M_C(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} tx \\ ty \\ i3 \end{bmatrix}$$

$$M_C(\beta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} ty \\ tx \\ i3 \end{bmatrix}$$

These two different memory access patterns are illustrated in Figure 5.2. In thread mapping α, which is shown in Figure 5.2(b), adjacent memory addresses are accessed by threads of a certain stride apart. Since adjacent threads (consecutive thread IDs) form a hardware thread batch and access the memory simultaneously, this mapping does not produce coalesced memory accesses. In the thread mapping β, which is shown in Figure 5.2, adjacent memory addresses are accessed by adjacent threads with consecutive thread IDs and these memory accesses are fully coalesced. Based on this observation, we formally extend our memory access pattern classification as follows.

True linear pattern refers to accesses where threads with consecutive thread IDs access contiguous and increasing memory addresses. This pattern is represented by a “1” in the last row of the column corresponding to tx in the inter-thread memory access matrix (or a “-1” in the reverse linear case). Intra-thread memory access patterns do not play a role in this classification.
False linear pattern is similar to the true linear pattern except that consecutive memory addresses are accessed by threads with non-consecutive threads IDs. This pattern is represented by “1” or “-1” in any row aside from the last row of the column that corresponds to $tx$ in the inter-thread memory access matrix. Figure 5.2 shows a graphical view of the true and false linear patterns, along with their representations in our model.

All other patterns are similarly subdivided into true and false patterns. Figure 5.3 shows the true and false shifted patterns as an additional example.

Figure 5.3: True shifted and false shifted patterns and their representation in our parallel model. “W” denotes the width of a work group.
5.2.1 Coalesced Memory Accesses

When threads of a thread batch execute a memory instruction, the GPU hardware attempts to convert these separate accesses into one or a few coalesced accesses whenever possible—the exact number of accesses required depends on the size of data type and the access pattern. Note that the extent to which GPUs can coalesce memory accesses is highly dependent on the underlying hardware.

In our memory access pattern analysis model, a true linear pattern and a true reverse linear pattern are considered as fully coalesced memory accesses. Our approach is conservative in the sense that all other patterns could result in various types of coalesced access patterns, but we consider them as uncoalesced patterns for simplicity in this work. Figure 5.4 illustrates an example of the coalesced memory access.

![Figure 5.4: A coalesced memory access pattern and its representation in the model. Adjacent threads whose IDs are consecutive access adjacent memory addresses. Note that this example matches true linear pattern.](image)

5.2.2 Accesses to the Same Memory Address

Identifying when different threads read from the same memory address is crucial for the utilization of constant memory. Using our model, we can recognize when threads
will read from the same address based on two requirements: 1) there are no inter-thread memory access patterns (there are all zeros in the inter-thread entries of the memory access matrix), and 2) there exists an intra-thread memory access pattern (at least one non-zero element) where the non-zero element does not correspond to a loop variable of the input problem size. If the loop variable value is a function of the problem size, then the array could potentially be a candidate for data prefetching. Note though that if the size of the array is guaranteed to be smaller than the size of constant memory it is possible to place the array to constant memory even though the loop variable is a function of the problem. Figure 5.5 illustrates an example of the same memory address read.

![Figure 5.5: A same address read and its representation in the model. There is no inter-thread pattern and there is a intra-thread pattern.](image)

### 5.2.3 Data Prefetch

Data prefetch is a key mechanism to remove memory bottlenecks and is an essential technique on GPUs if we want to achieve high performance. Data prefetching is implemented by using local memory after loop strip mining is performed. Efficient

---

Footnote:

1Loop strip mining is a technique to split a single loop into a nested loop. The resulting inner loop iterates over a strip of the original loop - the number of iterations of the inner loop is known as the *strip length*. 

---
use of local memory can significantly boost performance [1, 19, 52].

In our representation, the potential benefits of data prefetch can be explored by characterizing any intra-thread memory access patterns present. If we detect an intra-thread pattern, this is typically due to loops that are not explicitly mapped to threads (e.g., loop bodies that remain inside a kernel body, even after thread mapping is performed). There are potential benefits to prefetching this data to local memory and then accessing those elements locally. In this thesis we only consider linear intra-thread patterns as candidates for prefetch, since other patterns tend to be less deterministic and makes it difficult to predict the potential benefits of prefetching. Note that there is no benefit from prefetching data to local memory in a kernel that does not have an intra-thread access pattern (e.g., vector addition kernel).
Chapter 6

Vectorization via Data Transformations

AMD GPUs employ a vector architecture where memory operations and computations are performed on a vector granularity. The full computational power of these parallel chipsets can only be achieved when software utilizes the entire vector length via loop vectorization. However, loop vectorization is significantly hindered by irregular memory access patterns present in arrays. In this chapter, we present a data transformation technique that allows us to vectorize loops that are not vectorizable by the compiler alone. The data transformation rules needed to guide loop vectorization are generated using our algorithm based on the proposed model and analysis of data access patterns presented in Chapter 4 and 5.

6.1 Background: Vectorization on GPUs

Before we present our proposed methodology for vectorization through data transformations on GPUs, we provide the readers with a brief background on conventional loop vectorization and differences between vectorization on general vector processors and on GPUs.
6.1.1 Conventional Loop Vectorization

Vectorization is the process of substituting scalar operations with their vector counterparts, where a single operation is performed on each element of the vector operand. Loop vectorization differs significantly depending on the implementation of the underlying hardware and programming language. As an example, when the vector length is unbounded (e.g., Fortran 90), an entire loop iteration is replaced with a single vector statement (e.g. for (i=0; i<N; i++) X[i] = Y[i]; is replaced with $X[0:N-1] = Y[0:N-1]$) [99]. On the other hand, when the vector length is a constant (e.g., 4 elements as in short SIMD architectures such as the vector extensions in a commodity x86 microprocessors), strip-mining is required by a programmer if $N$ is larger than the vector length $VL$ [100, 101]. These two different approaches to vectorization entail applying different dependence analysis and transformation methods when vectorizing loops.

Typical vector operations work as follows. Data is loaded from memory into vector registers and the same operation is applied to each element of vector registers. The output is then written back to memory from the vector registers. Vector operations require that data is placed in contiguous locations in memory with proper alignment with respect to the vector length. This requirement presents challenges when faced with irregular array accesses in the loop body. There has been a large volume of prior work devoted to addressing this issue. Nuzman et al. proposed a technique for dealing with interleaved data accesses in which data is reorganized after being loading into vector registers from memory [101]. However, this approach requires that the underlying vector machine has the capability to shuffle data within vector registers. Algorithm 1 shows the logical view of vector programming on a short SIMD vector machine.

Nested loops provide more opportunities and challenges in vectorization. Early
Algorithm 1: Short SIMD vector programming

1. for \( i=0; i < N; i+=VL \) do
2. vector read from memory to register;
3. reorganize vector registers (optional);
4. perform vector operations on vector registers;
5. reorganize vector registers back to original (optional);
6. vector write to memory;
7. end

Vectorization research focused on innermost loop vectorization (ILV) due to its simplicity, but recently outer loop vectorization (OLV) has received attention from researchers as well because of the likelihood of coarser-grained data parallelism [100]. There has been work that uses data reordering during compilation to automatically vectorize for short SIMD vector extensions [101]. Their technique, however, is limited to applications where the memory access patterns are a constant stride of powers of 2, and platforms that support data reordering in hardware and the Instruction Set Architecture (ISA).

Since short SIMD vector units have been introduced in CPUs (e.g., Altivec on PowerPC, SSE and 3DNow on x86), loop vectorization at compile time has also been studied [100, 101, 102]. Automatic loop vectorization is available on some compilers but is not generally available due to issues related to data alignment and irregular memory access patterns. For these reasons, many programmers still do hand-coded assembly to vectorize their programs [103]. There are also prior works from the hardware community that provide a set of more versatile memory operations to minimize the overhead of irregular memory operations and relax the restrictions on the programming model [46, 47, 48].

### 6.1.2 Loop Vectorization on AMD GPUs

AMD GPUs are based on a SIMD vector machine that operates on up to four floating point or integer vector operands at a time (this vector size originated from the RGBA
color representations in computer graphics). However, unlike typical short SIMD vector machines where vector mapping occurs when loading/storing between memory and vector registers (as in lines 2 and 6 in Algorithm 1), vector mapping on the AMD GPU takes place when copying data from the CPU to the GPU\(^1\). Algorithm 2 shows the logical view of vector programming on a GPU. Note that \textit{arrayCPU} and \textit{arrayGPU} in line 1 refer to the array variables on the CPU and GPU, respectively.

\begin{algorithm}
\caption{GPU vector programming}
\begin{algorithmic}
\State \texttt{memcpy(float arrayCPU to float4 arrayGPU)};
\State /* kernel starts */
\For {\text{\(i=0; i < N; i+=VL\)}}
\State use float4 here;
\EndFor
\State /* kernel ends */
\State \texttt{memcpy(float4 arrayGPU to float arrayCPU)};
\end{algorithmic}
\end{algorithm}

![Diagram](float arrayCPU[2][8] \downarrow \text{memcpy (float arrayCPU to float4 arrayGPU)} \downarrow \text{float4 arrayGPU[2][2]})

Figure 6.1: \textit{Memory mapping for GPU vectorization}. The size of an array of float type is reduced by a factor of 4 when it maps to an array of float4 type during memory copy from host to GPU device.

In the AMD GPU programming models, both OpenCL and Brook+ provide built-in vector types to allow code to be explicitly programmed in vector form. Vector types refer to float, double, or int types of 2 to 4 elements long, where the size is appended as a suffix (e.g., float4, int2, etc.). Line 1 in Algorithm 2 shows the process

\(^1\)The back-end GPU compiler tries to use all vector register components (fine-grained vectorization) as much as possible using extensive data dependence analysis, but its ability is largely limited by high-level code structure.
of copying data from an array of a float type on a CPU to an array of float4 type on a GPU. During this copy, implicit vectorization occurs where float data on the CPU is converted into float4 on the GPU. Consequently, the number of elements in the array on the GPU are reduced by a factor of the vector length, and the size of each element is increased by a factor of the vector length. This memory mapping changes are illustrated in Figure 6.1. The vectorized array is then used in the kernel body (line 3). Note that the number of loop iterations in the kernel body is also reduced by a factor of the vector length. Remember that in order for vector operations to be used on AMD hardware, data operands must be located sequentially in memory; this is an important issue discussed in the following sections.

Vectorization can have a profound effect on performance, especially on vector-based GPUs. For instance, using vector (versus scalar) types helps maximize the utilization of the available memory bandwidth. Also, when the vectorized dimensions of arrays are mapped to threads, loop vectorization on a GPU also can decrease the number of threads to be completed. This means we will need fewer threads to complete the same amount of work. To obtain these performance benefits, we focus on data transformations as a path to map otherwise non-vectorizable loops onto vector GPU hardwares.

6.2 Vectorization Test

As introduced previously, implicit vectorization on the AMD platform occurs by mapping scalar type data to vector type data when copying data from the CPU to the GPU. In order to vectorize a loop on an AMD platform, the data being accessed must be located contiguously in memory. Therefore, our primary focus is on the last dimension of the array access (i.e., the last row dimension in a row-major memory layout).

In our mathematical model, vectorizing multiple arrays simultaneously in a loop nest requires that each array possesses the same memory access pattern with respect
to its last dimension. This condition can be checked by inspecting the last row of the memory access matrices for the selected arrays.

Consider the memory access matrices $M_A^i$ and $M_B^i$ for the two arrays $A$ and $B$ taken from the matrix multiplication shown in Section 4.1. The last rows, $[0, 0, 1]$ and $[0, 1, 0]$ respectively, differ, indicating that the row dimension of array $A$ is accessed by $i_3$ (index of innermost loop) and the row dimension of array $B$ is accessed by $i_2$ (index of middle loop). These two access patterns differ and thus cannot be vectorized in a single vectorized computation. Therefore, we define multiple arrays as vectorizable if the last rows in both their memory access matrices and their offset vectors are the same.

6.3 Quantifying Vectorization

Since it frequently occurs that not all statements in a loop nest can be vectorized, traditional vectorizing compilers categorize vectorization into completely and partially vectorizable loops [104]. Completely vectorizable loops refer to those in which every statement is vectorizable, while partially vectorizable loops refer to those in which some, but not all, statements are vectorizable.

In our case we need a somewhat different metric for quantifying vectorization as our vectorization methods are different. We define loops as intrinsically vectorizable if all arrays can be directly mapped to vector types without any changes to the loop body. We also define fractionally vectorizable if some, but not all, arrays can be mapped to vector types. In the fractionally vectorizable case, we use the equation below to further quantify the degree of vectorization:

$$Q_V(\%) = \frac{\# \text{ of Vectorizable Array Instances}}{\text{Total} \# \text{ of Array Instances}} \times 100$$

Note that $Q_V$ value of an intrinsically vectorizable loop is 100%.
6.4 Generating Data Transformation Rules

Given our ability to test for and quantify vectorization, we develop an algorithmic method to generate data transformation rules for non-vectorizable loops that pass our vectorization test. The data transformation rule for each array is composed of a transformation matrix $T_m$ of size $D \times D$ and a shift vector $\vec{t}_v$ of size $D$, where $D$ is the size of the iteration vector and is denoted as the pair, $(T_m, \vec{t}_v)$.

The application of $(T_m, \vec{t}_v)$ entails two steps. First, the actual data (i.e., the array layout) is transformed on the CPU using our data transformation routine, which takes as input the original data and transformation rules, and produces restructured data. Second, a new memory access vector, $\vec{m}'$ is generated as follows:

$$
\vec{m}' = M'\vec{i} + \vec{o}' = T_m M \vec{i} + (T_m \vec{o} + \vec{t}_v)
$$

It is common that the same array will be referenced multiple times with different memory access patterns within a loop body; we handle these instances individually if their $\vec{m}$ vectors are different. Next, we will present the required data transformations for each class of memory access pattern.

**Linear and Reverse Linear Access Patterns:** the data transformation required for the linear pattern (including the reverse linear pattern) is to reverse the order of the data. Consider the loop body shown in Listing 6.1.

```c
for( i1=0; i1<=N; i1++)
    for( i2=0; i2<=M; i2++)
        X[i1][i2] += Y[i1][M-i2] + 1;
```

Listing 6.1: Vectorization sample for linear and reverse linear access patterns. Array Y exhibits a reverse linear pattern while array X exhibits linear pattern.

The memory access matrices for array X and Y are $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$. 
Both dimensions of array X and the first dimension of array Y are linear patterns, while the second dimension of array Y is a reverse linear pattern. If we choose to transform Y to match the pattern in X, the data transformation matrix, $\mathbf{T}_m^Y$, would be $
abla \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ and $\vec{t}_v^Y$ would be a zero vector.

**Shifted Access Pattern**: a shifted access pattern can cause alignment problems during vectorization. The data transformation required to correct for a shifted access pattern is to shift all array elements by a constant amount. This can be done by pointer manipulation and involves little overhead in C/C++. As a result, $\vec{t}_v$ will be a non-zero vector, whereas $\mathbf{T}_m$ is the same as in the linear pattern case. A code snippet taken from a Livermore Loops kernel (Hydro Fragment) which exhibits this pattern is shown in Listing 6.2, along with its data transformation rules. Note that there are two instances of array Z, each of which has a different shift vector. Therefore we generate individual data transformation rules for each case. This results in generating two (versus one) new arrays, which come with added storage overhead.

```plaintext
for(i=0; i<N; i++)
    X[i] = q + Y[i] * (r * Z[i+10] + t * Z[i+11]);
```

Listing 6.2: Vectorization sample for shifted access pattern. Array Z exhibits two different shifted patterns while array Y exhibits a linear pattern.

$$
\mathbf{T}_{m}^{Z_1,Z_2} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \vec{t}_{v}^{Z_1} = \begin{bmatrix} -10 \end{bmatrix}, \quad \vec{t}_{v}^{Z_2} = \begin{bmatrix} -11 \end{bmatrix}
$$

**Overlapping Access Pattern**: an overlapping access pattern can occur when multiple loop indices are involved in the computation of a single array dimension. Consider the following sample code in Listing 6.3.
for(i=0; i<N; i++)
    for(j=0; j<M; j++)
        X[i][j] = Y[i][i+j] + 1;

Listing 6.3: Vectorization sample for overlapping access pattern. Array Y exhibits a overlapping pattern while array X exhibits a linear pattern.

The memory access matrix of array Y, $M_Y$, is $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$. Since multiple elements in the last row are nonzero ([1, 1] in this case), this indicates that multiple loop indices are involved in the access of the last dimension of the array. In other words, the last dimension of array Y is accessed jointly by two loop levels (indexed by $i$ and $j$). In this example, the elements of array Y accessed by inner loop $j$ are shifted by one as outer loop $i$ iterates. The data transformation matrix, $T_m^Y$, is $\begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$ and shift vector, $\vec{t}^Y$, is a zero vector.

Non-Unit Stride Access Pattern: the data transformation needed for a non-unit stride pattern is to gather elements being accessed into a big chunk of linear addressable memory. The overhead associated with this data transformation may be relatively large since it requires fine-grained array restructuring. A loop possessing this pattern is shown in Listing 6.4 below.

for(i=0; i<N; i++)
    for(j=0; j<M; j++)
        X[i][j] = Y[i][2j] + 1;

Listing 6.4: Vectorization sample for non-unit stride access pattern. Array Y exhibits a non-unit stride pattern while array X exhibits a linear pattern.
CHAPTER 6. VECTORIZATION VIA DATA TRANSFORMATIONS

The memory access matrix of array \( Y \), \( M_Y \), is \[
\begin{bmatrix}
1 & 0 \\
0 & 2
\end{bmatrix}
\]
and the data transformation matrix, \( T_{m}^{Y} \), becomes \[
\begin{bmatrix}
1 & 0 \\
0 & \frac{1}{2}
\end{bmatrix}
\].

Random and Compound Access Patterns: linearization of a random memory access pattern remains an open problem for loop vectorization—random patterns make loop vectorization complicated, hindering wider adoption of vector processors for general purpose computation. Even though there are custom techniques for enabling vectorization of semi-random patterns which possess some degree of regularity [46], it is impossible (though very expensive) to vectorize completely random patterns. For these reasons we do not attempt to transform loops possessing random patterns in this thesis.

The data transformation required for compound access patterns (arrays that exhibit more than one patterns) is straightforward. We first decompose the pattern into one of the patterns above, and then apply secondary transformations for each additional pattern. The order of data transformation selection does not matter and the generation of the data transformation matrix is straightforward as long as the memory access matrix is a non-singular matrix.

6.5 Vectorization Algorithms

Algorithm 3 illustrates how vectorizability is tested for and how \( Q_V \) is computed. The algorithm takes as input the memory access matrices of all array instances in the targeted loop nest and outputs the largest vectorizable group of array instances and \( Q_V \). The algorithm scans all input memory access matrices and compares each row to find the largest group of array instances that are vectorizable together. Two rows are vectorizable (as in line 7 in Algorithm 3) if they have a relationship described by one of the patterns described in the previous section. Note that the offset vector is not
CHAPTER 6. VECTORIZATION VIA DATA TRANSFORMATIONS

considered in the algorithm because it does not impact vectorizability (any shifted pattern can be easily vectorized by shifting data). $Q_V$ is simply computed at the end of the algorithm using the number of elements present in the largest vectorizable group.

Algorithm 4 presents the process of generating data transformation rules. The input is the group of vectorizable array instances discovered by Algorithm 3, and the output is a set of data transformation rules for each array. The goal of the algorithm is to find the position of a vectorizable row in the memory access matrix and the relationship between the vectorizable row and the given target row that we are trying to vectorize. The algorithm takes each memory access matrix and offset vector in an input vectorizable group one at a time and compares each row of $M$ and $\vec{\sigma}$ with a given target row. Memory access pattern relationships are identified by the sign (e.g., negative for a reverse linear pattern), the magnitude (e.g., a non-unit stride pattern), the number of non-zero components (e.g., an overlapping pattern) of a given row in $M$ and the value of $\vec{\sigma}$ (e.g., present in a shifted pattern). One of the data transformation rules, array permutation (e.g., switching dimensions in multidimensional arrays), can be used when the vectorizable row is not the last row (line 8). Note that the data transformation rule determination for compound memory access patterns is handled by identifying each pattern and adding it to the set.

Our heuristics used in this algorithm is based on a simple case matching method (see as an example multiple independent if-conditionals from line 16 to 27). While the order of if-conditionals certainly affects the performance of the algorithm, its optimality depends on captured memory access patterns (i.e., contents of memory access matrix and offset vector). Since the execution time of this algorithm is very small in practice, though, we believe that the performance tradeoff between different heuristics is not a big issue.

The complexity of the vectorization test algorithm (Algorithm 3) is $O(n^4)$. This is based on the assumption that the dimension of arrays in most scientific and engineering applications does not exceed 4. However, the contributions of the second and fourth loop iterations to overall complexity are negligible. Therefore, the effective
Algorithm 3: Vectorization test and $Q_V$ computation.

```
input : M for all instances of arrays in a loop nest (SM)
output: a vectorizable group (VG) and $Q_V$
1 VG ← ∅; G ← ∅;
2 for each $M \in SM$ do
3     Put M into G;
4     for each row $\vec{r} \in M$ do
5         for each $N \in SM$ do
6             for each row $\vec{l} \in N$ do
7                 if $\vec{l}$ is vectorizable with $\vec{r}$ then
8                     G ← G + N; break;
9                 end
10             end
11         end
12     if (size of G) > (size of VG) then
13         VG ← G;
14     end
15     G ← ∅;
16 end
17
18 $Q_V \leftarrow (\text{size of VG}) / (\text{size of SM}) \times 100$;
```

complexity of the algorithm can be reduced to $O(n^2)$. Furthermore, the number of arrays found in data-parallel workloads generally remains small—the maximum number of arrays involved in the kernels we evaluated in this thesis was 12, therefore we believe that the complexity of the vectorization test algorithm is not a big issue in practice. The complexity of Algorithm 4 is $O(n^2)$ but for the same reasons as stated for Algorithm 3 above, the effective complexity is $O(n)$.

### 6.6 Evaluating Vectorization on GPUs

We have evaluated the effectiveness of our loop vectorization strategy on an AMD GPU using a number of different benchmark kernels. All benchmarks are executed on an Intel Core 2 Duo at 2.66 GHz running 64-bit Linux, with 2 GB main memory and
with a single AMD HD 5870 GPU. The stream computing SDK version 2.1 is used to program and compile the benchmark kernels. For all benchmarks selected, two GPU versions (scalar and vector) are implemented using the OpenCL programming language, and actual data transformation are performed on the host CPU.

To illustrate the power of the presented optimizations, we investigated three high performance computing benchmark suites: Linpack, Livermore Loops, and NAS kernels. These suites implement scientific kernels that are prime candidates for acceleration on data parallel architectures. Table 6.1 provides some important characteristics for these three benchmarks suites. Within these benchmarks are a number of loops that initialize arrays and wrap functions for test purposes (in the NAS kernels and Linpack). These are considered to be non-kernel loops and are excluded from the number of total loops reported. The remainder of the loops are then divided into categories that identify the number of loops that are trivially vectorizable, intrinsically vectorizable (only after data transformation), and non-vectorizable (such as those with loop-carried dependencies). The statistics show that a significant percentage of loops in high performance applications are vectorizable after applying data transformations. Table 6.2 classifies the reasons why loops could not be vectorized.

We focus our evaluation on five loops: the first is a commonly used GPGPU benchmark, matrix multiplication, and the remaining four loops are taken from the three benchmark suites mentioned above. Our selections are based on the following criteria: 1) The loop is initially non-vectorizable but vectorizable after data transformation, and 2) The loop possesses a wide range of memory access patterns and loop structures to demonstrate the effectiveness of our proposed methods.

All benchmarks were run 5 times while varying the problem size, and the arithmetic mean of the execution time is plotted in from Figure 6.2 to 6.6. Table 6.3 and 6.4 also summarizes the detailed results of the benchmark experiments.

The matrix multiplication implementation used in our study of vectorization is shown in Listing 4.1. Our algorithm informs us that there are three possible choices for vectorization ($\{A,B\},\{A,C\},\{B,C\}$) via a dimension switch (2D matrix transpose)
and since $Q_V$ is the same in all three cases, we arbitrarily choose \{A,B\} for our implementation. Figure 6.2 shows a performance comparison between the scalar and vector implementations. It is clear that the vectorized kernel outperforms its scalar counterpart and that the overhead associated with data transformation is relatively small. Additional information is summarized in Table 6.3 and 6.4.

![Figure 6.2: Performance comparison between scalar and vector implementation of matrix multiplication kernel on an AMD platform.](image)

Next, we select two of the Livermore loops: kernel 1 (Hydro Fragment) and kernel 19 General Linear Recurrence Equations. Kernel 1 is fairly small in size and there are three arrays involved. The first two arrays are accessed with the same linear pattern, and the last array is accessed twice using a shifted address pattern with different offsets. Our algorithm tells us to duplicate the third array and shift each element by a constant offset. The resulting transformed arrays enable us to vectorize the loop. The effect on performance is shown in Figure 6.3. Note that the overhead associated with the data transformation (i.e., shifting in this case) is almost zero thanks to pointer manipulation, though there is storage overhead associated with
data duplication.

Kernel 19 operates on four one-dimensional arrays. Two of these arrays possess linear and reverse linear access patterns, and the other two have shifted and reverse shifted patterns. Each array is either transformed or transformed after duplication by our algorithm. The effect on performance is plotted in Figure 6.4. The vectorization effect on performance is substantial (11.5× speedup over the scalar version at an input size of 3072).

![Figure 6.3: Performance comparison between scalar and vector implementation of Hydro Fragment kernel on an AMD platform.](image)

We also studied two loops from the NAS benchmark suite: one is the Cholesky decomposition/substitution subroutine and the other is from Vpenta. The Cholesky loop accesses two 3-dimensional arrays in the loop nest, where one array has a linear memory access pattern and the other has a non-unit stride pattern. Our algorithm recognizes that the two arrays can be vectorized if the dimensions of one of the arrays are switched. The second loop chosen from the NAS benchmark exhibits the same memory access pattern and data transformation rules, though the array dimension is
Figure 6.4: Performance comparison between scalar and vector implementation of General Linear Recurrence Equations kernel on an AMD platform.

different. The performance effects of these two kernels are plotted in Figure 6.5 and Figure 6.6, respectively.
Figure 6.5: Performance comparison between scalar and vector implementation of Cholesky Decomposition/Substitution Subroutine kernel on an AMD platform.

Figure 6.6: Performance comparison between scalar and vector implementation of Vpenta kernel on an AMD platform.
Algorithm 4: Generating data transformation rules.

**input**: all $M$ and $\vec{o}$ pairs for vectorizable arrays ($VG$) and a target row ($\vec{c}$)

**output**: a set of data transformation rules ($T$ and $\vec{t}$ pair) ($DT$)

1. $DT \leftarrow \emptyset$; $T \leftarrow \text{null}$; $\vec{t} \leftarrow \vec{0}$;
2. for each $(M, \vec{o})$ pair $\in VG$ do
   3. for each row $(\vec{r}, o) \in M$ do
   4.   if $\vec{r} \equiv \vec{c}$ then
      5.     if $(\vec{r}$ is last row of $M)$ and ($o \equiv 0$) then
         6.       $T \leftarrow$ (no transformation);
         7.     end
      8.     else if $o \equiv 0$ then
         9.       $T \leftarrow$ (permutation, row number);
        10.    end
      11.    if $o \neq 0$ then
         12.       $T \leftarrow$ (shifted); $\vec{t} \leftarrow -\vec{o}$;
        13.    end
    14. else
        15.     if sign is different then
           16.       $T \leftarrow$ (reversing);
        17.    end
        18.     if magnitude is different then
           19.       $T \leftarrow$ (non-unit stride, magnitude);
        20.    end
        21.     if (number of non-zero components in $\vec{r}$) $>$ (number of non-zero components in $\vec{c}$) then
           22.       $T \leftarrow$ (overlap, non-zero components);
        23.    end
        24.    if $o \neq 0$ then
           25.       $T \leftarrow$ (shifted); $\vec{t} \leftarrow -\vec{o}$;
        26.    end
    27. end
   28. if ($T \equiv \text{null}$) and ($\vec{t} \equiv \vec{0}$) then continue;
   29. else $DT \leftarrow DT + (T, \vec{t})$;
30. end
### Benchmarking Results

#### Table 6.1: Loop characteristics for the benchmark suites studied for the vectorization.

A significant percentage of loops in high performance applications are vectorizable when applying data transformations.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Candidate Loops</th>
<th>Trivially Vectorizable</th>
<th>Intrinsically Vectorizable after Data Transform</th>
<th>Non Vectorizable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linpack</td>
<td>7</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Livermore Loops</td>
<td>24</td>
<td>3</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>NAS Kernel</td>
<td>32</td>
<td>13</td>
<td>12</td>
<td>7</td>
</tr>
</tbody>
</table>

#### Table 6.2: Breakdown of the reasons why benchmark loops can not be vectorized even after data transformation.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Non Vectorizable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>Linpack</td>
<td>2</td>
</tr>
<tr>
<td>Livermore Loops</td>
<td>13</td>
</tr>
<tr>
<td>NAS Kernel</td>
<td>7</td>
</tr>
</tbody>
</table>

#### Table 6.3: Summary of benchmark kernels for vectorization on an AMD platform.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Memory Access Patterns</th>
<th>Data Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication</td>
<td>Linear, Non-unit Stride</td>
<td>Dimension Switch (2D)</td>
</tr>
<tr>
<td>Livermore 1</td>
<td>Linear, Shifting</td>
<td>Duplicate, Shifting</td>
</tr>
<tr>
<td>Livermore 19</td>
<td>Linear, Shifting, Reverse Linear</td>
<td>Duplicate, Shifting, Reversing Col.</td>
</tr>
<tr>
<td>NAS Cholesky</td>
<td>Linear, Non-unit Stride</td>
<td>Dimension Switch (3D)</td>
</tr>
<tr>
<td>NAS Vpenta</td>
<td>Linear, Non-unit Stride</td>
<td>Dimension Switch (2D)</td>
</tr>
</tbody>
</table>

Table 6.1: *Loop characteristics for the benchmark suites studied for the vectorization.*

Table 6.2: *Breakdown of the reasons why benchmark loops can not be vectorized even after data transformation.*

Table 6.3: *Summary of benchmark kernels for vectorization on an AMD platform.*
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$Q_V$</th>
<th>Speedup over Scalar*</th>
<th>Data Trans. Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication</td>
<td>66.6%</td>
<td>2.02×</td>
<td>25.3%</td>
</tr>
<tr>
<td>Livermore 1</td>
<td>100%</td>
<td>2.29×</td>
<td>negligible</td>
</tr>
<tr>
<td>Livermore 19</td>
<td>100%</td>
<td>11.5×</td>
<td>5.65%</td>
</tr>
<tr>
<td>NAS Cholesky</td>
<td>100%</td>
<td>1.37×</td>
<td>2.27 %</td>
</tr>
<tr>
<td>NAS Vpenta</td>
<td>100%</td>
<td>2.34×</td>
<td>37.6 %</td>
</tr>
</tbody>
</table>

Table 6.4: Summary of benchmark kernels for vectorization on an AMD platform (continued). *The speedup shown includes data transformation overhead and is measured when the size of each dimension of the array is 3072.
Chapter 7

Memory Space Selection

As discussed in Section 2.1.2, multiple memory spaces are present on GPUs and they are exposed to the programmer during GPU programming. Because the characteristics and best usage of each memory space are distinct, the programmer should place each data in the most appropriate space based on its associated memory access pattern. Proper memory selection can have a huge impact on performance as we have shown in Section 3.2, but remains a challenging optimization problem [1, 38, 105].

The following subsections summarize how the characteristics and best usage of each memory space are represented in our model, followed by our proposed algorithm that selects the best memory space using the captured memory access patterns. Note that we use NVIDIA GPUs in this chapter.

7.1 Global Memory Space Selection

Global memory is an off-chip memory space that is not cached. Even though its usage is the most flexible (it can be used for any part of the program), its performance varies substantially depending on the associated memory access pattern. The use of this memory should be restricted to fully-coalesced access cases. Serialized access to global memory is very expensive given the high cost of off-chip memory accesses (the
latency to global memory is two orders of magnitude higher than accesses to on-chip memory). As we presented in Section 5.2.1, only true linear and true reverse linear patterns are guaranteed to be fully coalesced and we place arrays with these access patterns into global memory space.

Given the characteristics of global memory, we have developed the following rules for the best use of global memory:

- for output data, or
- for read-only or read-write input data that has true linear or true reverse linear patterns.

Note that our definition is conservative (a sufficient condition rather than a necessary one). For example, an overlapped pattern can be coalesced when the shift amount in later iterations becomes a multiple of the half size of a hardware thread batch.

7.2 Constant Memory Space Selection

Constant memory is a small off-chip space (64KB on the current generation of commodity mainstream GPUs) that is cached and read-only. Its scope is global in the sense that it is visible to all threads. The latency of constant memory is similar to reading from a register when all threads of a half thread batch read the same address from the cached data. Reads to different addresses by threads within a half batch are serialized, resulting in a cost that increases linearly with the number of different addresses read by all threads within a half batch [1]. We recall the definition of a same address read using our model presented in Section 5.2.2 as follows:

- no inter-thread memory access patterns (all zero elements) and any intra-thread memory access pattern (at least one non-zero element), implying that memory access to this array is independent across threads.
Note that this is also a sufficient condition in that only threads of a half warp are required to read the same memory address in order for access to remain nonserialized. This detailed investigation depends on thread configuration (e.g., work group size and shape (dimensions)). We do not consider that details for the sake of simplicity.

The following summarizes the cases in which constant memory should be selected:

- read-only data that is small enough to fit in the available constant memory space, and
- accesses that satisfy the “same address read” requirement.

Consider the following simple loop nest and associated memory access patterns for arrays X and Y, which shows the case where constant memory is best utilized.

```c
for(i=0; i<M; i++) {
    float tmp = 0;
    for(j=0; j<=N; j++) {
        tmp += Y[j];
    }
    X[i] = tmp;
}
```

Listing 7.1: Example kernel for constant memory space selection. All threads read the same address of array Y.

Given that the outer loop is mapped to threads (the thread vector is \([i]\)), the first column of each memory access matrix represents the inter-thread memory access pattern for each array and the second column represents intra-thread memory access pattern (see Figure 7.1). Therefore, each element of array X is accessed by a corresponding thread once (denoted as 1 in an inter-thread and 0 in an intra-thread memory access pattern). Alternatively, each element of array Y is accessed by all threads with the same linear pattern (denoted by 0 in the inter-thread and 1 in the


\[
\tilde{m}_x = \begin{bmatrix} 1 & 0 \\ i & j \end{bmatrix} + 0
\]

\[
\tilde{m}_y = \begin{bmatrix} 0 & 1 \\ i & j \end{bmatrix} + 0
\]

Figure 7.1: Captured memory access pattern for the kernel in Listing 7.1. Array Y possesses an appropriate memory access pattern for constant memory.

intra-thread memory access pattern cases). Therefore, array Y meets the requirements of constant memory. Figure 7.2 illustrates the performance impact of placing array Y in constant memory. As shown, the performance gain associated with using constant memory is substantial (approximately \(1.5 \times\) speedup across all array sizes).

Figure 7.2: Performance impact of placing array Y in constant memory in the example kernel shown in Listing 7.1. The performance gain associated with using constant memory is substantial.
7.3 Texture Memory Space Selection

Similar to constant memory, texture memory is an off-chip space that is cached and is read-only. But its size and usage differ from those of constant memory. First, texture memory can be much larger in size than constant memory. Theoretically, the entire global memory space can be bound to the texture unit. Second, the texture cache is optimized for 2D spatial locality meaning that the best performance is achieved when all threads of a half thread batch access the addresses that are close to each other in 2D [1, 56, 105]. This 2D locality can be described as storing data through the Z-curve [56, 106]. Placing data in texture memory also has other benefits such as hardware interpolation, data type conversion, and hardware address calculation.

These characteristics of texture memory guide us to use it in the following cases:

- read-only data that is uncoalesced (see the definition of coalesced accesses in our model in Section 5.2.1), including random patterns, or
- read-only data that does not meet the requirements of global, constant, and local memory.

Consider the following simple loop extracted from [105] and the memory access vector for array Y. The loop simply copies the data from input array Y to array X, shifted by shift amount; array Y exhibits a shifted memory access pattern and is uncoalesced when the shift is not a multiple of the size of a half-warp.

```c
for(i=0; i<M; i++) {
    X[i] = Y[i + shift];
}
```

Listing 7.2: Example kernel for texture memory space selection. The memory access pattern of array Y changes between coalesced and uncoalesced depending on the value of shift.
\[ \tilde{m}_Y = \mathbb{1}[i] + [\text{shift}] \]

Figure 7.3: Captured memory access pattern for the kernel in Listing 7.2. Array Y possesses a shifted access pattern.

Figure 7.4 shows a performance comparison between using global and using texture memory for array Y in the kernel shown in Listing 7.2 on a NVIDIA GTX 285 GPU. Placing data in texture memory improves memory performance by approximately 30% over placing data in global memory when uncoalesced (the value of shift is not a multiple of 16 (half warp)). Uncoalesced access to global memory serializes memory access, introducing a significant memory bottleneck. Note that if memory accesses are fully coalesced, global memory slightly outperforms texture memory (as shown when the shift amount is “0” or “16” in Figure 7.4). This is because the overhead associated with accessing the texture unit access does not exist.

Figure 7.4: Performance impact of placing array Y in texture memory in the example shown above at array size 2048.
7.4 Local Memory Space Selection

Local memory is a small on-chip software-managed scratchpad memory that resides on each compute unit. Local memory differs from global, constant, or texture memory in that it is local to the compute unit and visible to only active threads running on that compute unit. The latency of local memory access (assuming no bank conflicts) is similar to register access (two orders of magnitude faster than global memory [1, 52]). Effective use of this low-latency on-chip shared memory is crucial to avoid memory bottlenecks and to approach the peak computational rates available on a GPU.

Data that has any degree of reuse should be a candidate to be placed into local memory [1, 19, 52, 105]. This memory is used as a local cache for data placed in other memory spaces on the device. Explicit prefetching is required by the programmer to utilize this memory space. Given that: 1) the limited local memory space is shared equally by multiple work groups running on a compute unit, 2) the size of a work group directly affects both the amount of local memory used and the total number of registers required, finding the best usage for this local memory becomes a challenging multi-dimensional optimization problem. For this reason, hardware vendors provide a spreadsheet calculator that guides the programmer to increase hardware utilization and arrive at a good configuration [107]. We will revisit this optimization problem and present our algorithmic solution to this challenge in Chapter 8.

The following summarizes the characteristics of the data that should place in local memory:

- read-write data that can benefit from prefetching (see the definition of data prefetching in our model in Section 5.2.3).

7.5 Memory Selection Algorithm

Having described the characteristics and best usage of each memory space, next we present an algorithm to identify the best memory space to place data based on its
captured memory access pattern. The algorithm takes as input the memory access matrix and offset vector pairs of all instances of an array, the size, and the read-write characteristics, and the thread index vector. It then scans the input data to select the most appropriate memory space for the array. The following criteria are used for completing a selection: 1) read-only vs. read-write, 2) size of data, 3) memory access classification, 4) inter-thread memory access pattern, and 5) intra-thread memory access pattern.

Pseudocode for this algorithm is shown in Algorithm 5. Note that the small in line 4 refers to the case where the data set size is smaller than the available constant memory.

After running the algorithm, multiple memory spaces may be selected for a single array depending on the number of instances of the array. In this case, we make a final selection using the following priorities: for read-only data, 1) texture, 2) global, 3) local, 4) constant, and for read-write data, 1) global, and 2) local. For example, if local memory space is selected for the first access (exhibits the potential for data prefetching) and texture memory space is selected for the second access (exhibits random access) for the same array, then we select the texture memory for this mapping because it is not chunkable and so can not be placed in local memory.

7.6 Experimental Results

We evaluate the effectiveness of our memory selection methodology using NVIDIA’s SDK 3.0 on an NVIDIA GeForce GTX 285 (G200 family) GPU. For all benchmarks selected, two GPU versions are implemented: one using only the default (global) memory and one using algorithmically selected memory spaces.

The serial loops evaluated here are selected from the 
Parboil benchmark suite [108] and PhysBAM [62, 109]. The Parboil is a set of benchmarks comprised of real world applications to measure the performance of general-purpose programs on GPU platforms, and PhysBAM is an object-oriented C++ library capable of solving a variety of
Algorithm 5: Memory selection.

\begin{algorithm}
\begin{algorithmic}
\State input: $M$ and $\vec{\omega}$ pairs of all instances of an array ($SMA$), the size, read-write, and a thread index vector ($\vec{t}$)
\State output: memory space ($MS$)
\State $MS \leftarrow 0$;
\If {read-only}
\For {each $M \in SMA$}
\If {Small \&\& Same address read}
\State $MS \leftarrow MS + \text{Constant}$;
\EndIf
\ElsIf {Data prefetch}
\State $MS \leftarrow MS + \text{Local}$;
\EndIf
\ElsIf {Coalesced}
\State $MS \leftarrow MS + \text{Global}$;
\Else
\State $MS \leftarrow MS + \text{Texture}$;
\EndIf
\EndFor
\Else
\For {each $M \in SMA$}
\If {Data prefetch}
\State $MS \leftarrow MS + \text{Local}$;
\Else
\State $MS \leftarrow MS + \text{Global}$;
\EndIf
\EndFor
\EndIf
\end{algorithmic}
\end{algorithm}

problems in computational fluid dynamics, computational mechanics, and computer graphics.

We investigated seven kernels from four Parboil benchmark applications and one kernel from the PhysBAM library. For each serial kernel, the memory access profiles
Table 7.1: Summary of benchmark kernels tested for memory space selection.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Kernel</th>
<th>Memory Access Patterns and Additional Memory Access Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRI-Q</td>
<td>ComputePhiMag</td>
<td>Linear read, Linear write, No temporal reuse</td>
</tr>
<tr>
<td>MRI-Q</td>
<td>ComputeQ</td>
<td>Linear read, Linear write, Same address read</td>
</tr>
<tr>
<td>MRI-FHD</td>
<td>ComputeRhoPhi</td>
<td>Linear read, Linear write, Not enough temporal reuse</td>
</tr>
<tr>
<td>MRI-FHD</td>
<td>ComputeFH</td>
<td>Linear read, Linear write, Same address read</td>
</tr>
<tr>
<td>CP</td>
<td>cpuenergy</td>
<td>Non-unit stride write, Same address read</td>
</tr>
<tr>
<td>SAD</td>
<td>sad4_cpu</td>
<td>Linear read, Linear write, Random read, Temporal reuse</td>
</tr>
<tr>
<td>SAD</td>
<td>larger_sads</td>
<td>Random read-write</td>
</tr>
<tr>
<td>PhysBAM</td>
<td>ClampParticle</td>
<td>Linear write, Random read, Temporal reuse</td>
</tr>
</tbody>
</table>

are built using our model and the memory spaces are selected using Algorithm 5. Table 7.1 summarizes the benchmark kernels used for the evaluation of our proposed memory selection methodology on the NVIDIA platform.

The kernels, *ComputePhiMag* and *ComputeRhoPhi*, are small kernels and all data arrays are accessed with linear reads, linear writes, and no data prefetching potential. These three characteristics guide our algorithm to select global memory space for all arrays. The kernel *larger_sads* has a large read-write array whose memory access pattern is random for both read and write. Due to this random write access, the only possible selection is global memory. For these three kernels (whose memory selection is only global), a more formal performance comparison is not available because our baseline implementation is global memory.

The data memory access patterns in the *ComputeQ* and *ComputeFH* kernels can be characterized by a linear write, linear read, and same address read in our model. The read-only input array in both kernels is accessed by all threads with the same access pattern, regardless of the thread configuration (denoted as the ”same address read” in our model) and placed in constant memory by our algorithm after
Checking the size requirement. Figure 7.5 and 7.6 show the performance impact of these selections on each kernel, respectively.

The kernel \textit{cpuenergy} exhibits similar characteristics found in the two kernels above, except that it has a non-unit stride. The memory selected for this kernel is the same and the results showing the impact of this choice are shown in Figure 7.7. The kernel, \textit{sad4\_cpu}, exhibits various kinds of memory access patterns. There are multiple arrays used in this kernel and our model characterizes those accesses as linear reads, linear writes, random reads, and temporal reuse. Our algorithm selects global, texture, and shared memory spaces for those arrays to be placed. Figure 7.8 shows that the resulting performance improvement is significant when using all three memory spaces.

The kernel \textit{ClampParticle} from the PhysBAM benchmark is a representative function that frequently arises in physics simulations. It determines the new position for a subset of particles using a weighted sum of the positions of its neighbors. Using our model, its memory access pattern is characterized as a linear write, a random

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Kernel</th>
<th>Memory Selections</th>
<th>Speedup over Default Space*</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRI-Q</td>
<td>ComputePhiMag</td>
<td>Global</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>ComputeQ</td>
<td>Global, Constant</td>
<td>3.03×</td>
</tr>
<tr>
<td>MRI-FHD</td>
<td>ComputeRhoPhi</td>
<td>Global</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>ComputeFH</td>
<td>Global, Constant</td>
<td>5.34×</td>
</tr>
<tr>
<td>CP</td>
<td>cpuenergy</td>
<td>Global, Constant</td>
<td>1.3×</td>
</tr>
<tr>
<td>SAD</td>
<td>sad4_cpu</td>
<td>Global, Texture, Shared</td>
<td>13.5×</td>
</tr>
<tr>
<td></td>
<td>larger_sads</td>
<td>Global</td>
<td>N/A</td>
</tr>
<tr>
<td>PhysBAM</td>
<td>ClampParticle</td>
<td>Global, Texture, Shared</td>
<td>8.89×</td>
</tr>
</tbody>
</table>

Table 7.2: \textit{Summary of benchmark kernels tested for memory space selection (continued).} *The speedup shown is measured using the largest input size provided by each benchmark suite.
Figure 7.5: Performance comparison between using default global memory and using selected memory for the Parboil ComputeQ kernel as run on an NVIDIA platform. Note that we used the input data size provided by benchmark suites.

read, and a temporal reuse. Our algorithm selects global, texture, and shared memory spaces for the arrays used in the kernel. Figure 7.9 illustrates the performance improvement of using the selected memory spaces. For a large input size, the speedup achieved is $8.89 \times$. Table 7.1 summarizes our experimental results.
CHAPTER 7. MEMORY SPACE SELECTION

Figure 7.6: Performance comparison between using default global memory and using selected memory for the Parboil ComputeFH kernel as run on an NVIDIA platform. Note that we used the input data size provided by benchmark suites.

Figure 7.7: Performance comparison between using default global memory and using selected memory for the Parboil CP kernel as run on an NVIDIA platform. Note that we used the input data size provided by benchmark suites.
CHAPTER 7. MEMORY SPACE SELECTION

Figure 7.8: Performance comparison between using default global memory and using selected memory for the Parboil SAD kernel as run on an NVIDIA platform. Note that we used the input data size provided by benchmark suites.

Figure 7.9: Performance comparison between using default global memory and using selected memory in for the PhysBAM kernel as run on an NVIDIA platform. Note that we used the input data size provided by benchmark suites.
Chapter 8

Thread Mapping and Work Group Sizing

As we have discussed previously in Section 3.4, thread mapping and work group size can have an enormous impact on the performance of GPGPU applications. On GPUs where a group of threads access memory simultaneously, uncoalesced memory accesses are the biggest detriment to memory bandwidth, and choosing thread mapping and work group size that are inappropriate for the memory access patterns of the kernel will significantly degrade performance. Selecting a proper thread mapping and an appropriately-sized work group can produce better memory access behavior and result in more efficient use of local memory. This has the effect of prefetching and caching data, alleviating any memory bottlenecks. Our methodology uses the estimated number of uncoalesced memory accesses and the estimated size of the data that can be stored in local memory to compute cost and gain metrics, respectively. Using these metrics, we can evaluate the efficiency of a range of combinations of thread mappings and work group sizes.
8.1 Cost and Gain Calculation

Our cost and gain calculations are performed on a per-work-group basis, since all work groups have the same memory access behavior and hardware resource usage. Our proposed cost estimate is computed as follows. Fully coalesced memory patterns (i.e., true linear or true reverse linear) have no cost. False linear and false reverse linear patterns, on the other hand, have a cost of either “H”, “W”, “D”, or the product of a combination of these, where “H”, “W”, “D” are the height, width, and depth of the work group. The values that are chosen for the cost depend on the mapping of thread dimensions to array access dimensions. For example, in the case of the mapping $\alpha$ of the matrix multiplication kernel, array A exhibits a false linear pattern because adjacent threads are accessing different rows of the array. Since the “1” in the inter-thread pattern corresponds to $tx$, and $tx$ is the width of the work group, the cost becomes “W” in this case. The cost associated with array B is zero because $tx$ is not involved and we assume that higher dimensions ($ty$ in this case) of the thread map are always a multiple of a half warp size. The cost for the array C is “H*W” due to its false linear pattern present in both $tx$ and $ty$. Therefore, when the work group size is $8 \times 32$, the cost associated with arrays A, B, and C are $32$, $0$, and $8 \times 32$, respectively. Figure 8.1 shows a graphical view of our cost calculation for the two mappings of the matrix multiplication kernel.

The estimated gain is based on the number of elements in local memory, while taking loop strip mining into account. Note that data prefetching always entails loop strip mining. The strip length cannot exceed the size of the smallest dimension of thread map, otherwise we could not specify the entire size of the prefetched data simultaneously using local thread IDs. The gain is calculated by multiplying the size of smaller dimensions and size of the strip length. For example, for the $\alpha$ mapping matrix multiplication kernel, when the work group size is $8 \times 32$, the local memory accesses for arrays A and B are $32 \times 8$ (strip length) and $8 \times 8$ (strip length), respectively and the gains become $8 \times 8$ (smaller of 32 and 8) × 8 (strip length) and
Figure 8.1: Cost of two thread mapping schemes for matrix multiplication kernel when the work group size is $32(tx) \times 8(ty)$. “×” indicates uncoalesced memory accesses and SL denotes the loop strip length.

$8 \times 8$ (smaller of 8 and 8) $\times 8$ (strip length). Choosing the smaller dimension takes into account the fact that selecting a smaller strip dictates a higher inner loop trip count. The final cost and gain are the total of the individual costs and gains for each array.

8.2 Resource Constraints and Search Space

There are two per-compute-unit hardware resources to consider when searching for a good work group size: registers and local memory. Registers are a hardware resource that are shared among all active threads on a single compute unit (directly related
to the number of active threads). Local memory is shared by active work groups (directly related to the number of active work groups). The following two equations show how the two hardware resources are related to each other in terms of work group size.

\[
\frac{\#\text{Registers}}{\text{Thread}} \times \frac{\#\text{Threads}}{\text{Work Group}} \times \frac{\#\text{Work Groups}}{\text{Compute Unit}} < \frac{\#\text{Available Registers}}{\text{Compute Unit}} \tag{8.1}
\]

\[
\frac{\#\text{Local Memory}}{\text{Work Group}} \times \frac{\#\text{Work Groups}}{\text{Compute Unit}} < \frac{\#\text{Available Local Memory}}{\text{Compute Unit}} \tag{8.2}
\]

Equation 8.1 implies that given a particular register pressure ratio (denoted as \#\text{Registers} / \text{Thread} in the equation) computed on a per-kernel-body basis, the number of active work groups (denoted as \#\text{Work Groups} / \text{Compute Unit}) must decrease as the work group size (denoted as \#\text{Threads} / \text{Work Group}) increases. Equation 8.2 indicates that if the number of active work groups decreases, the amount of local memory that each work group can use (denoted as \#\text{Local Memory} / \text{Work Group}) increases. When a kernel does not use local memory, Equation 8.2 can be ignored. When multi-dimensional thread mapping is involved, there are a number of choices available when selecting the dimension sizes of a work group. In this case, we consider all possible choices where the lowest dimension size (\(tx\)) is multiple of a half warp size (the key unit for memory accesses). For example, the work group size of 256 in a two-dimensional thread mapping has 2 \((ty) \times 128 (tx)\), 4 \((ty) \times 64 (tx)\), 8 \((ty) \times 32 (tx)\), and 16 \((ty) \times 16 (tx)\) as possible combinations.

### 8.3 Work Group Size on Commodity GPUs

Hardware vendors usually provide ideal work group sizes for their GPUs [1, 52, 54, 56]. Since hardware resources almost always constrain the work group size, our

\^{1}The ideal work group size can be different across GPU architectures. NVIDIA recommends that GPUs with compute capability 1.3 (e.g., GTX 285) should have between 256 and 512 threads per
proposed search space starts from the largest ideal size and progresses down toward the minimum acceptable size until we find enough good candidates (specified by the programmer). For NVIDIA GPUs, the ideal work group size is not the maximum possible number of active threads per compute unit, as creating multiple work groups will allow for better hardware utilization.

Our search methodology for selecting the best thread mapping and work group size is summarized in Algorithm 6. Given the memory space selections provided by Algorithm 5, we start with the ideal work group size that allows for at least two work groups to be active per compute unit after accounting for register pressure (Equation 8.1 is used). From there, we decrease the work group size using predefined ordered sets of work group configurations, calculating the cost and gain of each configuration. Finally, the selection of candidates that produce optimized thread mappings and work group sizes is based on the following priority. When local memory is used, (i.e., there exists positive gain), we use 1) high gain, 2) low cost, 3) large work group size, otherwise we use, 1) low cost, 2) high occupancy, 3) high number of active work groups. Note that when data is prefetched into local memory, the number of device memory accesses tends to decrease significantly, implying that the associated cost is reduced. Note also that if local memory is not used, the work group size does not play a role.

Consider the mapping $\beta$ in the matrix multiplication kernel. According to Algorithm 5, arrays A and B possess access patterns that can benefit from data prefetching using local memory, and the output array C does not, so it is placed in global memory. The search begins with the largest ideal work group size, 512. Since the register pressure allows for 2 active work groups on a single compute unit, it is a valid starting size. Because the algorithm uses a two-dimensional thread mapping, we search all predefined configurations of a two-dimensional work group (attempts #1 through #12 in Table 8.1). Note that the lowest dimension ($tx$) is a multiple of the warp size. For each work group configuration that contains the the same total number of work group.
Algorithm 6: Searching for beneficial thread mappings and work group sizes.

| input: Memory access patterns and memory space selections for arrays, a thread index vector, the number of registers used, the desired number of candidates for good work group sizes (C) |
| output: Candidates for a good thread mapping and work group size |

1. Compute the largest starting work group size from the predefined work group configuration table (WGT) that allows at least 2 active work groups;
2. while WGT is not empty do
3.   Compute the number of active work groups (AWG) based on the register count;
4.   Compute the amount of local memory required (ALM);
5.   if ALM ≤ local memory size then
6.     Compute the cost and gain;
7.     Push it into the priority queue (PQ) based on our priority;
8.   end
9.   Get the next smaller work group size configuration from WGT;
10. end
11. Output the first C number of elements from PQ;

threads, we compute the cost and gain and store it in a priority queue data structure. The algorithm continues this process until it visits all entries of the predefined work group size configuration table. Finally, the algorithms output the specified number of elements (denoted as C in the algorithm) in the priority queue. Table 8.1 and 8.2 summarizes the work group search process, along with metrics, our rank, and the actual measured performance in the order of search.

8.4 Evaluation of Thread Mapping and Work Group Sizing

To evaluate our proposed thread mapping/work group sizing methodology, we developed a tool that directly implements the memory selection algorithm (Algorithm 5), and the thread mapping and work group size selection algorithms (Algorithm 6). As input, the tool takes a serial data-parallel loop nest, the loop iterators that are going
to be mapped to threads, and the desired number of candidate work group sizes. Our tool then outputs the selected memory space for each array, along with the potential candidates for optimized thread mappings and work group sizes. The tool first scans the loop nest and captures the memory access patterns present in the arrays into an affine model. It then analyzes access patterns and runs Algorithm 5 and 6.

We evaluate the utility of our framework using four different applications: two common numerical kernels and two complex real world kernels. These four benchmarks contain a wide range of memory access patterns, thread configurations, and resource requirements. We use our framework to generate suggested thread mappings and work group sizes, and we compare the performance of our suggested configurations against actual execution times.

This set of experiments were conducted on an NVIDIA GeForce GTX 285 GPU using the OpenCL programming language and SDK version 3.1. The host system is

Table 8.1: An example of algorithmic thread mapping and work group size search in matrix multiplication kernel. $ty \times tx$ (total number of threads).

<table>
<thead>
<tr>
<th># Attempt</th>
<th>WG size $^\dagger$</th>
<th># Active WG</th>
<th>Cost</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2×256 (512)</td>
<td>2</td>
<td>0</td>
<td>2×2 + 2×2 (8)</td>
</tr>
<tr>
<td>2</td>
<td>4×128 (512)</td>
<td>2</td>
<td>0</td>
<td>4×4 + 4×4 (32)</td>
</tr>
<tr>
<td>3</td>
<td>8×64 (512)</td>
<td>2</td>
<td>0</td>
<td>8×8 + 8×8 (128)</td>
</tr>
<tr>
<td>4</td>
<td>16×32 (512)</td>
<td>2</td>
<td>0</td>
<td>16×16 + 16×16 (512)</td>
</tr>
<tr>
<td>5</td>
<td>32×16 (512)</td>
<td>2</td>
<td>0</td>
<td>16×16 + 16×16 (512)</td>
</tr>
<tr>
<td>6</td>
<td>2×128 (256)</td>
<td>4</td>
<td>0</td>
<td>2×2 + 2×2 (8)</td>
</tr>
<tr>
<td>7</td>
<td>4×64 (256)</td>
<td>4</td>
<td>0</td>
<td>4×4 + 4×4 (32)</td>
</tr>
<tr>
<td>8</td>
<td>8×32 (256)</td>
<td>4</td>
<td>0</td>
<td>8×8 + 8×8 (128)</td>
</tr>
<tr>
<td>9</td>
<td>16×16 (256)</td>
<td>4</td>
<td>0</td>
<td>16×16 + 16×16 (512)</td>
</tr>
<tr>
<td>10</td>
<td>2×64 (128)</td>
<td>8</td>
<td>0</td>
<td>2×2 + 2×2 (8)</td>
</tr>
<tr>
<td>11</td>
<td>4×32 (128)</td>
<td>8</td>
<td>0</td>
<td>4×4 + 4×4 (32)</td>
</tr>
<tr>
<td>12</td>
<td>8×16 (128)</td>
<td>8</td>
<td>0</td>
<td>8×8 + 8×8 (128)</td>
</tr>
</tbody>
</table>
configured with an 2.66 GHz Intel Core 2 Duo running 64-bit Linux, with 2GB of main memory.

The first kernel evaluated is vector addition. This is a very simple kernel, and is usually the first kernel that a programmer learns for GPU programming. In vector addition, there is a single loop that iterates over three one-dimensional arrays, and each array exhibits the same access pattern: a true linear inter-thread pattern and no intra-thread pattern. Since the kernel has only one loop, the only possible thread mapping is a one-dimensional thread configuration. The number of registers required per thread is 3.

According to Algorithm 5, global memory is selected for all arrays. Since the number of registers used does not restrict the recommended work group size, we start with the largest work group size that hardware vendor recommends (512). For any
configuration, the cost and the gain are always zero since all memory access patterns are coalesced and no local memory is utilized. Algorithm 6 repeats the cost and gain calculations on the ordered, predefined set of work group sizes and pushes each result into the priority queue. The work group configurations that remain at the frontend of the priority queue become the candidates for selection. Table 8.3 and 8.4 summarizes the experimental results. We see that our ranking of work group sizes closely aligns with the actual measured performance.

<table>
<thead>
<tr>
<th># Attempt</th>
<th>WG size†</th>
<th># Active WG</th>
<th>Cost</th>
<th>Gain</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>512</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>2</td>
<td>256</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>3</td>
<td>128</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>50%</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>25%</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>25%</td>
</tr>
</tbody>
</table>

Table 8.3: *Experimental results of work group search for vector addition kernel*. The register count is 3.

<table>
<thead>
<tr>
<th># Attempt</th>
<th>WG size†</th>
<th>Our rank</th>
<th>Measured execution time*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>512</td>
<td>3</td>
<td>3.39 ms</td>
</tr>
<tr>
<td>2</td>
<td>256</td>
<td>2</td>
<td>3.38 ms</td>
</tr>
<tr>
<td>3</td>
<td>128</td>
<td>1</td>
<td>3.39 ms</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>4</td>
<td>3.60 ms</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>5</td>
<td>5.89 ms</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>5</td>
<td>11.61 ms</td>
</tr>
</tbody>
</table>

Table 8.4: *Experimental results of work group search for vector addition kernel (continued)*. *The input size is $2^{25}$. 
The second benchmark kernel is *matrix-vector multiplication*. This kernel has a two-level nested loop that iterates over arrays that differ in their number of dimensions. There are two one-dimensional arrays and one two-dimensional array. Again, there is only one choice for thread mapping: the memory access pattern of the one-dimensional output (write-only) array is $[1 \ 0] \begin{bmatrix} tx \\ i2 \end{bmatrix}$ and it exhibits a true linear inter-thread pattern with no intra-thread access. The memory access pattern of the one-dimensional input (read-only) array is $[0 \ 1] \begin{bmatrix} tx \\ i2 \end{bmatrix}$ and possesses no inter-thread accesses and is a true linear intra-thread pattern. Finally, the memory access pattern of the two-dimensional input (read-only) array is captured as $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} tx \\ i2 \end{bmatrix}$ and exhibits a false linear inter-thread pattern and a true linear intra-thread pattern. Given the memory access patterns, Algorithm 5 selects global memory for the first array. The second and third arrays exhibit the potential of benefiting from data prefetching, so for the second array texture memory is selected with prefetching to local memory, and for the third array global memory is selected, with prefetching to local memory.

In this kernel, the amount of local memory used is a major limiting factor that influences the work group size. Due to the local memory usage, the first valid work group size we can consider is 32. Table 8.5 and 8.6 summarizes the search process, and includes the cost and gain, measured execution time, and resulting speedup when compared to using only global memory.

<table>
<thead>
<tr>
<th># Attempt</th>
<th>WG size</th>
<th># Active WG</th>
<th>Cost</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>4</td>
<td>32</td>
<td>$32 \times 32 + 32$ (1056)</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>4</td>
<td>16</td>
<td>$16 \times 16 + 16$ (272)</td>
</tr>
</tbody>
</table>

Table 8.5: *Experimental results of work group search for matrix-vector multiplication.* The register count is 17.
The next kernel studied is a real world application that computes radiological paths—the most computationally expensive step in medical image reconstruction [38]. The particular algorithm is called the improved Siddon algorithm [92] and is the most widely used in its domain. The algorithm computes the contribution of the various parts of the body to the radiation received by a number of X-ray detector cells. For each radiological path from the radiation source to a detector cell, the local intensities of object cells (i.e., cells in the body) that the rays hit are integrated along the path. Given the number of detector cells and the radiation emission, radiological paths are calculated more than a million times. The kernel involves four arrays: one three-dimensional read-only array, one two-dimensional write-only array, and two one-dimensional read-only arrays. Given that we decide to map this loop to a two-dimensional thread configuration, there are two potential mappings. Here we show only the better of the mappings, which has the following memory access pattern (the order of the arrays is the same as the order when they were listed above).

Table 8.6: Experimental results of work group search for matrix-vector multiplication (continued). *The size of each dimension of the input arrays is 4096. † Speedup over default global memory space.
Following the order shown above, memory access patterns are classified as: no inter-thread and random intra-thread pattern, true linear inter-thread and no intra-thread pattern, true linear inter-thread and no intra-thread pattern, and true linear inter-thread and no intra-thread pattern. Given these pattern classifications, our memory selection algorithm selects texture memory for the first array and global memory for the rest of the arrays. The algorithm then searches to find good candidates for thread mappings and work group sizes, as shown in Table 8.7 and 8.8.

The final example is also taken from a real world application called Speeded Up Robust Features (SURF) [110]. The SURF algorithm is used to detect features in images for applications like stabilization and panorama stitching. The particular kernel we tested is called non-max suppression (NMS), which plays a critical role in the SURF algorithm. The kernel looks for the largest value in a stack of images which have been subjected to convolutions with different filters. The kernel involves four arrays: three two-dimensional read-only arrays which possess the same memory access pattern, and one two-dimensional write-only array. We map this loop to a two-dimensional thread configuration and again only show the better of the two possible mapping choices. The memory access patterns are as follows.

\[
\begin{bmatrix}
0 & 0 & Z \\
0 & 0 & Z \\
\end{bmatrix}
\begin{bmatrix}
\begin{bmatrix}
tx \\
ty \\
i3 \\
\end{bmatrix}
\end{bmatrix},
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
\end{bmatrix}
\begin{bmatrix}
\begin{bmatrix}
tx \\
ty \\
i3 \\
\end{bmatrix}
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
\end{bmatrix}
\begin{bmatrix}
\begin{bmatrix}
tx \\
ty \\
i3 \\
\end{bmatrix}
\end{bmatrix}
\]
<table>
<thead>
<tr>
<th># Attempt</th>
<th>WG size</th>
<th># Active WG</th>
<th>Cost</th>
<th>Gain</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2×128 (256)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>50%</td>
</tr>
<tr>
<td>2</td>
<td>4×64 (256)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>50%</td>
</tr>
<tr>
<td>3</td>
<td>8×32 (256)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>50%</td>
</tr>
<tr>
<td>4</td>
<td>16×16 (256)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>50%</td>
</tr>
<tr>
<td>5</td>
<td>2×64 (128)</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>50%</td>
</tr>
<tr>
<td>6</td>
<td>4×32 (128)</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>50%</td>
</tr>
<tr>
<td>7</td>
<td>8×16 (128)</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>50%</td>
</tr>
<tr>
<td>8</td>
<td>2×32 (64)</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>50%</td>
</tr>
<tr>
<td>9</td>
<td>4×16 (64)</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>50%</td>
</tr>
<tr>
<td>10</td>
<td>2×16 (32)</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>25%</td>
</tr>
</tbody>
</table>

Table 8.7: *Experimental results for the radiological path calculation.* The register count is 28.

<table>
<thead>
<tr>
<th># Attempt</th>
<th>WG size</th>
<th>Our rank</th>
<th>Measured execution time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2×128 (256)</td>
<td>3</td>
<td>176.90 ms</td>
<td>1.61×</td>
</tr>
<tr>
<td>2</td>
<td>4×64 (256)</td>
<td>3</td>
<td>176.50 ms</td>
<td>1.61×</td>
</tr>
<tr>
<td>3</td>
<td>8×32 (256)</td>
<td>3</td>
<td>174.17 ms</td>
<td>1.60×</td>
</tr>
<tr>
<td>4</td>
<td>16×16 (256)</td>
<td>3</td>
<td>164.92 ms</td>
<td>1.62×</td>
</tr>
<tr>
<td>5</td>
<td>2×64 (128)</td>
<td>2</td>
<td>169.85 ms</td>
<td>1.58×</td>
</tr>
<tr>
<td>6</td>
<td>4×32 (128)</td>
<td>2</td>
<td>170.16 ms</td>
<td>1.58×</td>
</tr>
<tr>
<td>7</td>
<td>8×16 (128)</td>
<td>2</td>
<td>164.75 ms</td>
<td>1.63×</td>
</tr>
<tr>
<td>8</td>
<td>2×32 (64)</td>
<td>1</td>
<td>166.09 ms</td>
<td>1.59×</td>
</tr>
<tr>
<td>9</td>
<td>4×16 (64)</td>
<td>1</td>
<td>163.01 ms</td>
<td>1.63×</td>
</tr>
<tr>
<td>10</td>
<td>2×16 (32)</td>
<td>4</td>
<td>162.35 ms</td>
<td>1.63×</td>
</tr>
</tbody>
</table>

Table 8.8: *Experimental results for the radiological path calculation (continued).* The input image size is 256 × 32.
The patterns are classified as no inter-thread and random intra-thread pattern, and true linear inter-thread and no intra-thread pattern, respectively. Texture memory is selected for the first three arrays and global memory is selected for the last array. The results of the work group size search are summarized in Table 8.9 and 8.10.

<table>
<thead>
<tr>
<th># Attempt</th>
<th>WG size</th>
<th># Active WG</th>
<th>Cost</th>
<th>Gain</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2×256 (512)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>2</td>
<td>4×128 (512)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>3</td>
<td>8×64 (512)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>4</td>
<td>16×32 (512)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>5</td>
<td>32×16 (512)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>6</td>
<td>2×128 (256)</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>7</td>
<td>4×64 (256)</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>8</td>
<td>8×32 (256)</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>9</td>
<td>16×16 (256)</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>10</td>
<td>2×64 (128)</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>11</td>
<td>4×32 (128)</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>12</td>
<td>8×16 (128)</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>12</td>
<td>2×32 (64)</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>50%</td>
</tr>
<tr>
<td>12</td>
<td>4×16 (64)</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>50%</td>
</tr>
<tr>
<td>12</td>
<td>2×16 (32)</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>25%</td>
</tr>
</tbody>
</table>

Table 8.9: Experimental results of NMS algorithm. The register count is 15.
<table>
<thead>
<tr>
<th># Attempt</th>
<th>WG size</th>
<th>Our rank</th>
<th>Measured execution time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2×256 (512)</td>
<td>3</td>
<td>16.8 ms</td>
<td>5.11×</td>
</tr>
<tr>
<td>2</td>
<td>4×128 (512)</td>
<td>3</td>
<td>16.8 ms</td>
<td>5.12×</td>
</tr>
<tr>
<td>3</td>
<td>8×64 (512)</td>
<td>3</td>
<td>16.8 ms</td>
<td>5.12×</td>
</tr>
<tr>
<td>4</td>
<td>16×32 (512)</td>
<td>3</td>
<td>16.8 ms</td>
<td>5.13×</td>
</tr>
<tr>
<td>5</td>
<td>32×16 (512)</td>
<td>3</td>
<td>16.9 ms</td>
<td>5.12×</td>
</tr>
<tr>
<td>6</td>
<td>2×128 (256)</td>
<td>2</td>
<td>16.8 ms</td>
<td>5.13×</td>
</tr>
<tr>
<td>7</td>
<td>4×64 (256)</td>
<td>2</td>
<td>16.8 ms</td>
<td>5.12×</td>
</tr>
<tr>
<td>8</td>
<td>8×32 (256)</td>
<td>2</td>
<td>16.8 ms</td>
<td>5.12×</td>
</tr>
<tr>
<td>9</td>
<td>16×16 (256)</td>
<td>2</td>
<td>16.9 ms</td>
<td>5.11×</td>
</tr>
<tr>
<td>10</td>
<td>2×64 (128)</td>
<td>1</td>
<td>16.8 ms</td>
<td>5.11×</td>
</tr>
<tr>
<td>11</td>
<td>4×32 (128)</td>
<td>1</td>
<td>16.8 ms</td>
<td>5.12×</td>
</tr>
<tr>
<td>12</td>
<td>8×16 (128)</td>
<td>1</td>
<td>16.9 ms</td>
<td>5.13×</td>
</tr>
<tr>
<td>12</td>
<td>2×32 (64)</td>
<td>4</td>
<td>16.9 ms</td>
<td>5.12×</td>
</tr>
<tr>
<td>12</td>
<td>4×16 (64)</td>
<td>4</td>
<td>16.9 ms</td>
<td>5.13×</td>
</tr>
<tr>
<td>12</td>
<td>2×16 (32)</td>
<td>5</td>
<td>17.1 ms</td>
<td>5.08×</td>
</tr>
</tbody>
</table>

Table 8.10: Experimental results of NMS algorithm (continued). The input image size is 256 × 256.
Chapter 9

Discussion and Future Work

In this chapter, we discuss the limitations of our approach as well as the potential for further development, which remains as our future work. We hope that this chapter gives the reader some perspective for the potential of continue work down this research path.

9.1 Discussion of the current research

Our analysis models and optimization techniques presented in this thesis are best employed when considering memory-intensive kernels (i.e., where the ratio of ALU to memory operations is low)\(^1\). In general, when arithmetic intensity is high or memory access patterns are well optimized, the GPU hardware can perform efficient memory latency hiding. If hardware can hide most memory latencies, then memory optimizations will have a limited performance impact. Although this is a limitation, in reality the majority of GPGPU kernels are memory-bound. Below we list the limitations of each memory enhancement technique that was proposed earlier in this thesis.

\(^1\)Note that most general-purpose applications become memory bound when running on GPUs because of the parallel thread execution model, and because of the inability of the underlying memory architecture to handle irregular memory accesses (as we described in Sections 2.1 and 2.3).
Data transformation overhead in vectorization Although our experimental results demonstrate that the proposed data transformations can significantly increase the number of loops that can be vectorized, and that the overhead associated with our data transformations on massively multi-threaded vector GPU architectures can be easily amortized, the overall overhead (including the data transformation time and the storage requirements) can become an issue since the overhead of data duplication is heavily dependent on the particular memory access pattern present. For example, a multi-point stencil code (e.g., 9-point stencil) would require multiple duplications of an array using our approach. Given the overhead of data duplication and the potential impact on cache utilization, data transformations on more complex reference patterns remains an open problem. An exact analysis of the tradeoffs between the benefits of vectorization and the associated transformation overhead is challenging.

Another issue with our data duplication approach arises when the data array is both read and written in a single loop body. Consider a loop nest in which an array is read twice, each time with a different access pattern, and with an intervening write between the two reads. If the data is duplicated based on an analysis of the two different read access patterns, data consistency issues can arise due to the intervening write. Although these cases can be easily detected, special care should be taken when duplicating arrays that involve read/write access.

Accuracy of our cost/gain metrics Since our cost metrics do not assign weights to different arrays based on their potential contribution to overall memory contention, our approach may not find the best possible mappings as the number of memory access patterns and associated arrays in a kernel increase. For most kernels, however, our methodology has been shown to work very well, and our simple, highly-automated, algorithms accurately represent the complex interplay between GPU hardware and software.
It is also well known that static performance prediction of GPU workloads is a challenging task and can produce mispredictions due to the complex interplay between GPU software and hardware [111]. In these cases, our proposed methodology can be used to provide guidance for optimizations by the programmer.

**Conservative approach** While our work has already achieved significant performance benefits, our algorithms have room for further optimization in some places by applying more expensive analysis. For example, in our cost calculation, we treat a non-unit stride pattern as an uncoalesced pattern and classify it the same as a random pattern. However, in reality, a non-unit stride pattern possessing a short stride results in significantly fewer memory transactions than a truly random pattern. Addressing a number of side cases in our model remains as future work.

Presently, our framework only handles standard uses of the memory spaces. Programmers frequently utilize memory spaces differently. A good example is local memory on the GPU. Instead of using local memory for data prefetching, it can be used to avoid uncoalesced accesses. Texture memory can also be effectively used to exploit hardware-based interpolation and filtering in selected application domains even if the memory access pattern is better suited for another memory space.

**Simplifications and assumptions** We made several simplifications in this work as well. Our thread mapping assumes that the problem size is a multiple of a half warp. Real-world applications are unlikely to always meet this requirement, so techniques such as data padding will be required before applying our methodology. We also assume that the register size per thread of a kernel is fixed and therefore it always has priority to determine the work group size over local memory. However, the programmer can perform explicit actions to reduce the
number of registers if a certain number of active threads is desired. The programmer can also give priority to local memory usage to increase performance. Another assumption made is that the data we are working with is naturally-aligned (e.g., int or float). Though this assumption is true in most cases, our methodology would require some modifications to work for unaligned data.

**Device dependency** Finally, our analysis model is device dependent. Should the characteristics of memory spaces change, our analysis needs to be updated accordingly. For example, NVIDIA’s latest (Fermi) GPUs have an on-chip cache for global memory, so the sensitivity of performance to uncoalesced memory accesses should be reduced substantially. In this case our cost metrics need to be updated. Also, interestingly, AMD GPUs have a very different memory organization and associated characteristics [52]. Our analysis model requires updates to be applied to AMD GPUs.

### 9.2 Future Work

In this section, we list potential further development and applications of our work that remain as a future work.

**Developing cost and gain metrics for AMD GPUs** An imperative task that remains is to develop more accurate cost and gain metrics for memory space selection, thread mapping and work group sizing on AMD platforms. AMD only recently announced the detailed characteristics of their memory system and optimization guidelines. The characteristics of AMD’s GPU memory system are quite different from NVIDIA’s [52]. Therefore, our metrics presented in Chapter 7 and 8 need to be updated accordingly so they can apply to AMD platforms. Accurate cost and gain calculations are keys for searching for the optimized thread mapping and work group size.
More accurate analysis As described in Section 9.1, our analysis model is somewhat conservative and has made simplifications in several places. There is certainly room for further development and improvement. One good place for more accurate analysis is to fully incorporate half thread batch size (i.e., half wavefront and warp) into our analysis model. Although this will increase analysis complexity, it will provide more accurate analysis and metrics for fine-grained memory tuning.

Fully automated source-to-source compiler This thesis has already handled most of the necessary steps required for building a C/C++ to OpenCL source-to-source compiler. Since OpenCL and other GPU programming languages such as CUDA and Brook+ are an extension of standard C/C++ language, writing source-to-source compiler is not particularly complex. The challenging part is finding efficient two-level thread mappings and optimizations, which we proposed in this thesis. As we discussed in Section 2.4.2, existing source-to-source compilers do not provide a solution for this. We cannot expect high performance without properly handling thread mapping and memory optimizations. We believe the methodology presented in this thesis is powerful enough to differentiate it from other automatic frameworks once implemented.

Evaluation of mapping efficiency An evaluation of thread mapping efficiency across a range of application, programming models, and hardware architectures remain as future work items. Even though GPU computing continues to grow more popular every day, this environment lacks a proper set of metrics for evaluating the efficiency of different application mappings onto the programming model. Developing a new set of metrics to evaluate the mapping efficiency of a wide class of general-purpose workloads onto a scalable GPGPU programming model (paying attention to memory access patterns and efficiency) would be very interesting and appreciated by the GPGPU community. To this end, we need to find a new metrics to measure the mapping efficiency that can be used to
develop this taxonomy. We believe that this quantitative evaluation might be of particular use in the evaluation of current GPGPU programming models (in terms of the classes of applications which map efficiently or inefficiently), and for the design of tomorrow’s programming model for many-core architectures.
Chapter 10

Conclusion

The pace of adoption of GPUs to the mainstream of general-purpose computing is phenomenal. GPU computing satisfies an insatiable market demand for increased computing power with lower power budgets to accommodate the ever-increasing data processing needs both on large scale and desktop end-user computing. Although today’s GPU hardware architectures and programming models have already been shown to provide a viable path to high performance for many classes of data-parallel workloads, memory operations remain as the primary limiting factor for performance in many general-purpose applications. Programmers must first understand the memory access patterns of their applications and the characteristics of GPU memory hardware, and then optimize the mapping between them to obtain the full computational benefits that GPUs can provide.

With that motivation, this thesis has been devoted to evaluating and enhancing the memory efficiency of general-purpose applications running on massively data-parallel GPUs. We have shown the value of clearly characterizing memory behavior of applications through a mathematical form, accompanied by detailed classification and analysis. Our methodology is comprehensive enough to explain all necessary memory behavior present in data-parallel execution (most importantly, the effect of thread mapping) that the programmer needs to consider. Based on this theoretical base, we
have presented our approach to address a wide range of important memory efficiency issues in today’s data-parallel GPU computing environment. Our proposed memory efficiency enhancement methodology includes vectorization via data transformation targeting vector-based GPUs, appropriate memory space selection, and search for optimized thread mappings and work group sizes.

Our goal was twofold. One was to provide an algorithmic methodology that can ease the burden of performing memory optimizations from programmers and the other was to provide a powerful theoretical foundation for automatic optimization and parallelization.

Heterogeneous computing is the future of high performance computing as homogeneous computing (i.e., multi-core CPU alone) hits computational and power walls. GPUs are the main driver of this transition and have become an increasingly important component of heterogeneous computing. Researchers in academia and industry have recognized this and are making efforts to eliminate the obstacles that hinder the wider adoption of GPU computing. We believe that this thesis contributes to this effort and helps accelerate the promising ecosystem of data-parallel GPU computing.
Bibliography


