Architecture-Aware Mapping and Optimization on Heterogeneous Computing Systems

Mayank Daga

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Wu-chun Feng, Chair
Alexey V. Onufriev
Yong Cao

April 27, 2011
Blacksburg, Virginia

Keywords: Multicore CPU, GPU, CUDA, OpenCL, Optimizations,
Performance Evaluation, Molecular Modeling

Architecture-Aware Mapping and Optimization on Heterogeneous Computing Systems

Mayank Daga

ABSTRACT

The emergence of scientific applications embedded with multiple modes of parallelism has made heterogeneous computing systems indispensable in high performance computing. The popularity of such systems is evident from the fact that three out of the top five fastest supercomputers in the world employ heterogeneous computing, i.e., they use dissimilar computational units. A closer look at the performance of these supercomputers reveals that they achieve only around 50% of their theoretical peak performance. This suggests that applications that were tuned for *erstwhile* homogeneous computing may not be efficient for today's heterogeneous computing and hence, novel optimization strategies are required to be exercised. However, optimizing an application for heterogeneous computing systems is extremely challenging, primarily due to the architectural differences in computational units in such systems.

This thesis intends to act as a *cookbook* for optimizing applications on heterogeneous computing systems that employ graphics processing units (GPUs) as the preferred mode of accelerators. We discuss optimization strategies for multicore CPUs as well as for the two popular GPU platforms, i.e., GPUs from AMD and NVIDIA. Optimization strategies for NVIDIA GPUs have been well studied but when applied on AMD GPUs, they fail to measurably improve performance because of the differences in underlying architecture. To the best of our knowledge, this research is the *first* to propose optimization strategies for AMD GPUs. Even on NVIDIA GPUs, there exists a lesser known but an extremely severe performance pitfall called partition camping, which can affect application performance by up to *seven-fold*. To facilitate the detection of this phenomenon, we have developed a performance prediction model that analyzes and characterizes the effect of partition camping in GPU applications. We have used a large-scale, molecular modeling application to validate and verify all the optimization strategies. Our results illustrate that if appropriately optimized, AMD and NVIDIA GPUs can provide *371-fold* and *328-fold* improvement, respectively, over a *hand-tuned*, SSE-optimized serial implementation.

To my Parents and Grandparents...

मम्मी–पापा, दादा–दादीमा

Acknowledgements

I express my heart felt gratitude to people who transformed my graduate life into a successful and memorable voyage.

First and foremost, I am indebted to my advisor Dr. Wu-chun Feng for not only providing me with academic guidance but also inspiring me by manifesting the facets of a remarkable personality. His disposition to push himself and produce excellent work has many a times stimulated me to work diligently. Contrary to the popular belief, he was around whenever I wanted to discuss anything with him, be it directions for my thesis or career options. I thank Dr. Feng for bringing the best out of me, which apprised me of my capabilities and instilled a greater confidence in me. I can indubitably say that these two years of working with Dr. Feng have been the most formative years of my life thus far.

I am grateful to Dr. Alexey Onufriev for suggesting me to work with Dr. Feng for my M.S. degree. It has been a pleasure collaborating with Dr. Onufriev on the *NAB* project for the last three years. I also thank him for being on my thesis committee.

I am thankful to Dr. Yong Cao for being on my thesis committee.

Ramu Anandakrishnan has been my oldest acquaintance at Virginia Tech. I thank Ramu for accepting my candidature for the summer internship, which initiated my association with this University. It has been a rewarding experience to collaborate with Ramu on the *HCP* project, since the days when it was known as *MLCA*. Working with Ramu led to my very first publication, which will always remain special.

Coming to United States for post-graduate studies is quite a leap from baccalaureate studies in India and it is not uncommon for one to feel lost, especially during the initial days. I was fortunate enough to have Ashwin Aji and Tom Scogland to resort to at such times. They generously clarified all my doubts, no matter how naive they were. I earnestly thank them both.

John Gordon and Andrew Fenley have unknowingly done a great favor by developing a *GEM* of an application.

I thank the SyNeRGy members for providing me with constructive feedback throughout these two years, which helped me improve my work.

I thank Shankha Banerjee for helping me fix compilation errors on countless occasions. Shankha Da has also been a great room mate and an amazing friend. I thoroughly enjoyed the companionship of Vishnu Narayanan and would always cherish the cooking sessions as well as after-dinner discussions that I had with him.

Anup Mandlekar is credited with making *Programming Languages* the most entertaining course that I have ever taken. We experienced the zenith of enjoyment during that course.

I am greatly thankful to Michael Stewart for his unconditional help on numerous occasions in the last two years. *Great* Michael redefines the adage, "a friend in need is a friend indeed".

The camaraderie I shared with Ryan Braithwaite would be memorable in the sense that even though we were not familiar with each other's work, we always had good albeit *infrequent* conversations, depending upon Ryan's presence at the CRC.

I am thankful to Vaibhav Bhutoria who insisted that I should pursue Masters.

Sony Vijay has been a confidante and the pillar of my strength. There is sheer magic in her company which makes me forget all the worldly tensions and worries.

Last and certainly not the least, I cannot overstate my gratitude for my parents who have always supported me in my endeavors and more importantly, have believed in my abilities. I could never be where I am today without all they have done.

My respects to Almighty!

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Related Work 6

1.3 Contributions 9

1.4 Document Overview 12

2 Heterogeneous Computing **13**

2.1 Systems 13

2.1.1 Multicore CPUs 14

2.1.2 Graphics Processing Units 16

2.2 Programming Environments 20

2.2.1 OpenCL 21

2.2.2 CUDA 22

2.3 Application 23

3 Mapping and Optimization on Multicore CPUs **27**

3.1 Optimizations 27

3.1.1 Optimizations for Sequential Program 28

3.1.2 Parallelization 31

3.2 Results 32

3.2.1 Optimizations for Sequential Program 33

3.2.2 Parallelization 34

4 Mapping and Optimization on AMD GPUs **37**

4.1 Optimizations . . . 37

4.1.1 Kernel Splitting . . . 39

4.1.2 Local Staging . . . 40

4.1.3 Vector Types . . . 41

4.1.4 Image Memory . . . 42

4.1.5 Optimizations in Combination . . . 43

4.2 Results . . . 44

4.2.1 Kernel Splitting . . . 45

4.2.2 Local Staging . . . 46

4.2.3 Vector Types . . . 47

4.2.4 Image Memory . . . 48

4.2.5 Optimizations in Combination . . . 48

5 Mapping and Optimization on NVIDIA GPUs **51**

5.1 Optimizations . . . 51

5.1.1 Cookbook Optimizations . . . 52

5.1.2 Partition Camping . . . 54

5.2 Results . . . 63

5.2.1 Cookbook Optimizations . . . 64

5.2.2 Partition Camping . . . 66

6 Summary and Future Work **69**

6.1 Summary . . . 69

6.2 Future Work . . . 71

Bibliography **74**

A Micro-Benchmarks for Detection of Partition Camping **82**

List of Figures

1.1 Execution of a program on heterogeneous computing system 2

1.2 Need for *architecture-aware* optimizations. Baseline: *Hand-tuned* SSE optimized serial implementation 3

1.3 Peak performance on AMD and NVIDIA GPUs 5

2.1 Rise of multicore CPUs [25] 14

2.2 Block diagram of a modern multicore CPU 15

2.3 Block diagram of an AMD stream processor and thread scheduler 17

2.4 Overview of NVIDIA GPU architectures 20

2.5 Programming environments: {OpenCL and CUDA} 21

2.6 GEM: Output 23

2.7 GEM: High-level overview 24

2.8 GEM: Algorithm for CPU implementation 24

2.9 GEM: Algorithm for GPU implementation 25

2.10 GEM: Memory bounds checking 26

3.1 SIMDization using *hand-tuned* vector intrinsics 30

3.2 Speedup due to optimizing the sequential program. Baseline: Basic sequential implementation on CPU 33

3.3 Scalability. Baseline: CPU sequential implementation optimized by `(-O3)` and vector intrinsics 36

4.1 Kernel splitting 38

4.2 Register accumulator 40

4.3 Vector loads 42

4.4 Image memory loads 43

4.5 Speedup due to optimizations in isolation. Baseline: Basic OpenCL GPU implementation. MT: Max. Threads, KS: Kernel Splitting, RA: Register Accumulator, RP: Register Preloading, LM: Local Memory, IM: Image Memory, LU{2,4}: Loop Unrolling{2x,4x}, VASM{2,4}: Vectorized Access & Scalar Math{float2, float4}, VAVM{2,4}: Vectorized Access & Vector Math{float2, float4} 45

4.6 Speedup with optimizations in combination. Baseline: Basic OpenCL GPU implementation. MT: Max. Threads, KS: Kernel Splitting, LM: Local Memory, RP: Register Preloading, IM: Image Memory, VAVM: Vectorized Access & Vector Math, VASM: Vectorized Access & Scalar Math . . . 50

5.1 The adverse effect of partition camping in GPU kernels. PC: Partition Camping . . 55

5.2 Partition camping effect in the 200- and 10-series NVIDIA GPUs. Column P_i denotes the i^{th} partition. All memory requests under the same column (partition) are serialized. 56

5.3 Comparison of CampProf with existing profiling tools. `gmem`: global memory; `smem`: shared memory. 57

5.4 Screenshot of the CampProf tool 61

5.5 Speedup due to cookbook optimizations. Baseline: Basic {CUDA,OpenCL} GPU implementation. MT: Max. Threads, NDB: Non Divergent Branches, RA: Register Accumulator, CM: Constant Memory, SM: Shared Memory, LU: Loop Unrolling 65

5.6 GEM: Memory access pattern 66

5.7 GEM: CampProf output 67

6.1 Speedup when optimized for each architecture 70

A.1 Code snapshot of the 'read' micro-benchmark for the NVIDIA 200- and 10-series GPUs (Without Partition Camping). Note: `ITERATIONS` is a fixed and known number. 83

A.2 Code snapshot of the 'read' micro-benchmark for the NVIDIA 200- and 10-series GPUs (With Partition Camping). Note: `ITERATIONS` is a fixed and known number. 83

List of Tables

3.1 Amdahl's Law: Speedup vs #Cores 35

6.1 Impact of optimization strategies on AMD and NVIDIA GPUs. Greater the number of +s, greater is the positive impact. The ones in red are architecture-aware optimizations 72

Chapter 1

Introduction

1.1 Motivation

Homogeneous computing systems, i.e., systems that use one or more similar computational units, have provided adequate performance for many applications in the past. Over the years, numerous scientific applications embedded with multiple modes of parallelism (like SIMD, MIMD, vector processing) have emerged and hence, homogeneous computing has been rendered less potent than before. According to corollary to the Amdahl's Law, the decrease in effectiveness of homogeneous computing systems is because they are being used for applications which are ill-suited to them [6]. Therefore, the need arises to adopt heterogeneous computing systems, as exemplified in figure 1.1.

Heterogeneous computing is the well-orchestrated and coordinated effective use of a suite

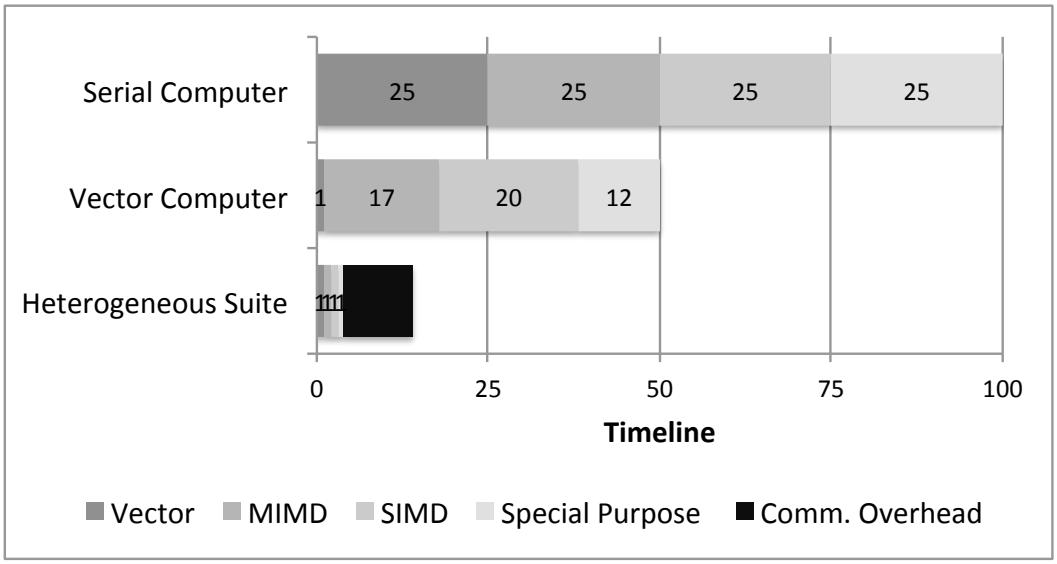


Figure 1.1: Execution of a program on heterogeneous computing system

of dissimilar high-performance computational units, such as multicore CPUs in conjunction with accelerators like CellBE, GPUs, FPGAs and ASICs. It provides superior processing capability for computationally demanding tasks with diverse computing needs [39]. The popularity of heterogeneous computing systems is evident from the fact that three out of the five fastest supercomputers in the world employ heterogeneous computing [1]. These supercomputers use traditional multicore CPU cores in conjunction with hundreds of GPU-accelerated cores, thereby making GPUs the preferred accelerator platform for high performance computing. The increased popularity of GPUs has been assisted by their (i) sheer computing power, (ii) superior performance/dollar ratio and (iii) compelling performance/watt ratio. A wide range of applications in image and video processing, financial modeling and scientific computing have been shown to benefit from the use of heterogeneous computing systems [37]. Thus, heterogeneous computing systems have begot a new era in supercomputing.

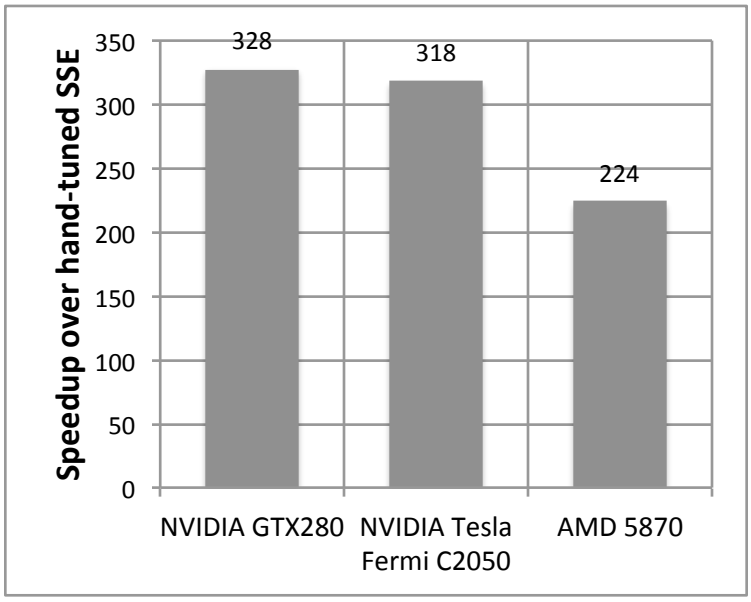


Figure 1.2: Need for *architecture-aware* optimizations. Baseline: *Hand-tuned* SSE optimized serial implementation

However, a closer look at the performance of the GPU-accelerated supercomputers reveal that they achieve only around 50% of their theoretical peak performance [1]. This is due to the fact that adaptation to heterogeneous computing requires one to overcome a significant number of challenges like (i) identification of embedded heterogeneous parallelism in an application, (ii) reduction of cross-over overhead that occurs when different computational units of a heterogeneous computing system interact, (iii) adaptation to new programming environments which enable programming of diverse computational units, and (iv) development of novel parallel algorithms for existing applications, to name a few [28]. Even if one adapts to these challenges, one has to deal with the fact that even the same computational units of a heterogeneous system are *not* created equal. For example, there exist uniform and non-uniform memory access (UMA/NUMA) CPUs as well as GPUs from various ven-

dors, which have different inherent architectures. On NUMA CPUs, memory is physically distributed between cores and hence, the access times depend on the physical location of memory relative to the core. With respect to GPUs, the difference in architectural subtleties has to be taken into consideration. In [8], the authors present that programs *do not* exhibit consistent performance across NVIDIA GPUs from different generations. Therefore, to expect a program to yield equal performance on GPUs from other vendors would be improper. To illustrate further, we plot figure 1.2, which portrays the speedup obtained for a molecular modeling application when run on three different GPUs; two NVIDIA GPUs, each from a different generation, and an AMD GPU. The molecular modeling application is described in detail in section 2.3. From the figure, we note that speedup on two NVIDIA GPUs is more or less similar, though not exactly equal, but speedup on the AMD GPU is materially lower than its NVIDIA counterparts.

It is intriguing that despite the materially higher peak performance of AMD GPUs, they exhibit less performance improvement, as shown in figure 1.3. The GPUs we used are near the far right end, where the difference increases to nearly 2-fold. While some loss from peak performance is expected, a device with a higher peak, especially higher by almost double, is expected to produce at least comparable if not greater performance. The fact that the AMD GPU does not, leads us to believe that *architecture-aware* optimization strategies are necessary. However, optimizations require deep technical knowledge of inherent architectures and hence, it is far from trivial to optimize one's program and extract optimum performance from heterogeneous computing systems.

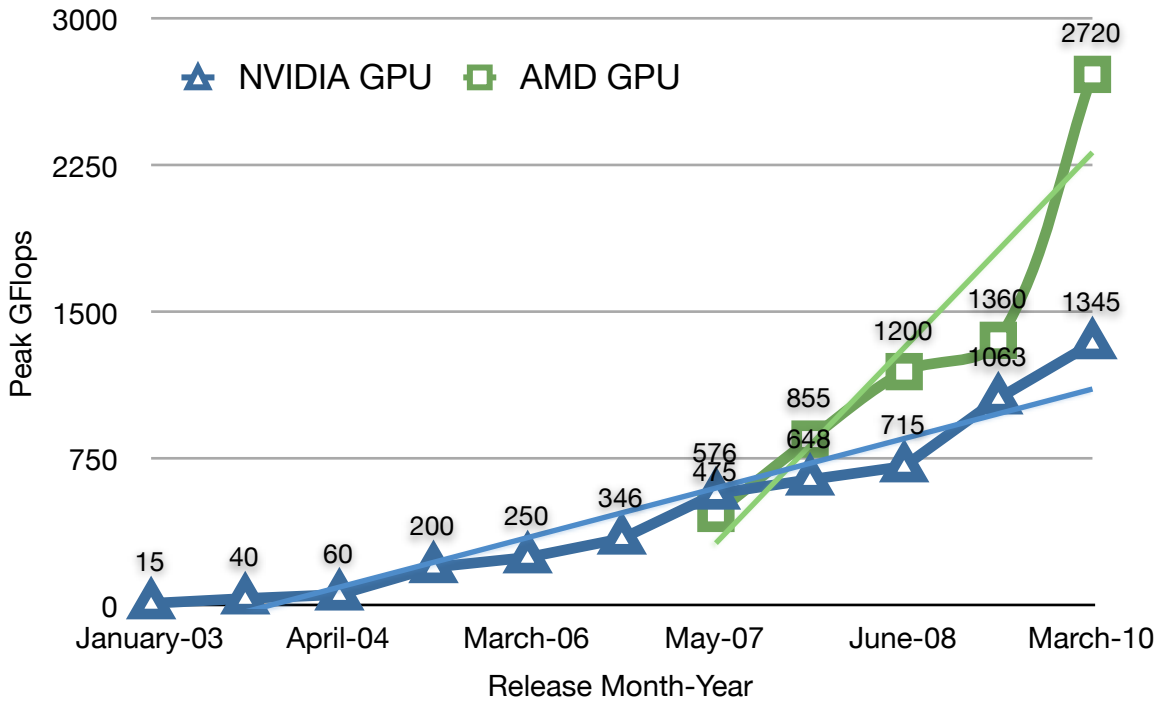


Figure 1.3: Peak performance on AMD and NVIDIA GPUs

Therefore, even with platform agnostic programming environments like OpenCL that enable execution of programs on various heterogeneous computing systems like, multicore CPUs, CellBE, AMD GPUs, NVIDIA GPUs and FPGAs, it behooves one to perform *architecture-aware* optimizations to realize greater performance improvement.

1.2 Related Work

With the advent of NVIDIA's CUDA architecture, considerable research has been done to determine optimization strategies for accelerating programs on heterogeneous computing systems with NVIDIA GPUs [31, 32]. The NVIDIA CUDA Programming Guide lists many optimization strategies useful for extracting peak performance on NVIDIA GPUs [36]. In [40–43], Ryoo et al. present optimization principles for a NVIDIA GPU. They conclude that though optimizations assist in improving performance, the optimization space is large and tedious to explore by hand. In [46], Volkov et al. argue that the GPU should be viewed as a composition of multi-threaded vector units and infer that one should make explicit use of registers as primary on-chip memory as well as use short vectors to conserve bandwidth. In [48], the authors expose the optimization strategies of the GPU subsystem with the help of micro-benchmarking.

Analytical models have also been developed to help the programmer understand bottlenecks and achieve better performance on NVIDIA GPUs. In [44], Baghsorkhi et al. have developed a compiler front end, which analyses the kernel source code and translates it into a Program Dependence Graph (PDG) that is useful for performance evaluation. The PDG allows them to identify computationally related operations, which are the dominant factors affecting the kernel execution time. With the use of symbolic evaluation, they estimate the effects of branching, coalescing and bank conflicts in the shared memory.

In [21], Hong et al. propose an analytical model, which dwells upon the idea that the major

bottleneck of any kernel is the latency of memory instructions and that multiple memory instructions are executed to successfully hide this latency. Hence, calculating the number of parallel memory operations (memory warp parallelism) enables them to accurately predict performance. Their model relies upon the analysis of the intermediate PTX code generated by the CUDA compiler. However, the PTX is just an intermediate representation, which is further optimized to run on the GPU and is not a good representation of the actual machine instructions, thereby introducing some error in their prediction model [34].

Boyer et al. present an automated analysis technique to detect race conditions and bank conflicts in a CUDA program. They analyze the PTX to instrument the program to track the memory locations accessed [11]. Schaa et al. focus on the prediction of execution time for a multi-GPU system, given that execution time on a single GPU is known [15]. They do so by introducing models for each component of the multi-GPU system; the GPU execution, PCI-Express, the memory (RAM), and disk.

In [22], Hong et al. propose an integrated power and performance model for GPUs, where they use the intuition that once an application reaches the optimal memory bandwidth, increasing the number of cores would not help the application performance and hence, power can be saved by switching off the additional cores of the GPU. Nagasaka et al. make use of statistical tools like regression analysis and CudaProf counters for power modeling on the GPU [20]. Our work also relies on regression techniques, but we chose very different parameters for our performance model. Bader et al. have developed automated libraries for data re-arrangement to explicitly reduce partition camping problem in the kernels [29].

However, none of the work referred to so far addresses the severe performance impact caused due to partition camping on NVIDIA GPUs. Thus, we have developed a performance model that analyzes and characterizes the effect of partition camping in GPU kernels. Also, a tool has been developed that visually depicts the degree of partition camping by which a CUDA application suffers. While developing micro-benchmarks and using statistical analysis tools is a common practice to understand the architectural details of a system, we have used them to create a more realistic performance model than those discussed. We also deviate from the existing literature and predict a *performance range* to understand the extent of partition camping in a GPU kernel.

To challenge NVIDIA's dominance in the GPU computing realm, AMD supports OpenCL on its GPUs. OpenCL is anticipated to play an important role in deciding how heterogenous computing systems will be programmed in the future. This is primarily due to fact that it is hardware agnostic, unlike CUDA. Programs written in OpenCL can execute on GPUs as well as CPUs, irrespective of the vendor. NVIDIA supports OpenCL by compiling the OpenCL program to the same binary format as that of CUDA and hence, the *well-researched* CUDA optimizations also work for OpenCL on NVIDIA GPUs.

On the other hand, all the published work regarding optimization strategies on AMD GPUs has been Brook+ centric, which has now been deprecated by AMD [10]. In [26, 49], the authors propose optimization strategies for Brook+ and evaluate them by implementing a matrix multiplication kernel and a multi-grid application for solving PDEs respectively. In [7], authors accelerate the computation of electrostatic surface potential for molecular

modeling by using Brook+ on AMD GPUs. In [19], the authors present a software platform which tunes the OpenCL program written for heterogeneous architectures to perform efficiently on CPU-only systems. The only work related to OpenCL GPU optimizations that we came across was a case study discussing an auto-tuning framework for designing kernels [27].

The present work is hence, the *first* attempt to propose OpenCL optimization strategies for AMD GPUs. We believe that one needs to exploit the causal relationship between programming techniques and the underlying GPU architecture to extract peak performance and hence, proclaim the need for *architecture-aware* optimization strategies.

1.3 Contributions

The goal of this thesis is to serve as a *cookbook* for optimizing applications on the most popular form of heterogeneous computing systems, i.e., one having multicore CPUs and GPUs as the computational units. In section 1.1, we discuss that it is imperative to perform architecture-aware optimizations to achieve substantial performance gains on heterogeneous computing systems. To this effect, we have devised optimization strategies for the following three architectures: (i) multicore CPUs, (ii) AMD GPUs, and (iii) NVIDIA GPUs. Below is a succinct description of our contributions with respect to each platform.

Multicore CPUs: We discuss optimization strategies that improve the instruction throughput of an application, such as (a) compiler optimizations, (b) vector intrinsics, and (c) cache blocking. We also portray the benefits of parallelizing an application on modern

multicore CPUs. Using Amdahl's Law, we project speedups that can be obtained with the increase in CPU cores. However, the financial burden imposed by increasing the number of cores paves the way for the adoption of accelerator platforms such as GPUs.

AMD GPUs: To the best of our knowledge, this work is the *first* to present and propose optimization strategies for AMD GPUs, based on the knowledge of its underlying architecture. Architectural subtleties like the presence of vector cores rather than scalar cores, the presence of only one branch execution unit for 80 processing cores, and the presence of a rasterizer on the GPU influence our proposal of the following optimization strategies: (a) use of vector types, (b) removal of branches, and (c) use of image memory.

NVIDIA GPUs: Over the last three years, optimization strategies for NVIDIA GPUs have been discussed in the literature and they have been well studied and evaluated [40, 41]. However, there exists a lesser known but an extremely severe performance pitfall for NVIDIA GPUs called *partition camping*, which can affect the performance of an application by as much as *seven-fold*. To facilitate the detection of partition camping effect, we have developed a performance model which analyzes and characterizes the effect of partition camping in GPU applications.

We have applied these optimizations to a large-scale, production-level, molecular modeling application called GEM and present a comparison of the efficacy of each optimization strategy [17]. Recent literature leads one to believe that applications written for NVIDIA GPUs

should perform best. However, we illustrate that when optimized appropriately, performance on a present generation AMD GPU can be better than an equally optimized implementation on the competing NVIDIA GPU. As an example, we managed to improve the performance of GEM on AMD GPUs from being 31% worse to 12% better as compared to NVIDIA GPUs. Overall for the AMD GPU, we have achieved a *371-fold* speedup over a *hand-tuned* SSE serial version and *36-fold* when compared to the SSE version parallelized across 16 CPU cores.

Fundamental contributions: Predicting the future to foretell the existence of GPUs is difficult. However, in case the GPUs become extinct, the ideology of architecture-aware optimizations can be extended to future heterogeneous architectures and platforms. Though the in-depth understanding of underlying architecture to obtain optimal performance seems obvious, it is seldom exercised. One can also learn from the optimization strategies that we have proposed, especially, how we amortize the adverse branching effects on GPUs. Also, the idea of predicting performance range rather than predicting the exact application performance would be more realistic on next generation architectures, if they have large performance variations as the GPUs, which have them due to partition camping effects.

1.4 Document Overview

The rest of this thesis is organized as follows. In chapter 2, we discuss background information about (i) computational units of heterogeneous computing systems, like multicore CPUs and GPUs from AMD and NVIDIA, (ii) programming environments used to program these systems, like OpenCL and CUDA, and (iii) the molecular modeling application, GEM, which serves as our case-study to test all the optimization strategies that have been devised. In chapter 3, we discuss the optimization strategies for multicore CPUs as well as portray the performance benefits obtained by the use of these optimizations. In chapters 4 and 5, we describe the optimization strategies for AMD and NVIDIA GPUs respectively and also demonstrate the results. In chapter 6, we present a summary of all the optimization strategies discussed on various platforms followed by a discussion of some future work that can be built upon this thesis.

Chapter 2

Heterogeneous Computing

In this chapter, we present an overview of the systems and the programming environments that we used, to propose architecture-aware optimization strategies. Lastly, we describe the GEM application, which acts as a case study to verify and validate all our findings.

2.1 Systems

This section presents a background of the chip microprocessors and the graphical processing units, for which we have proposed optimization strategies.

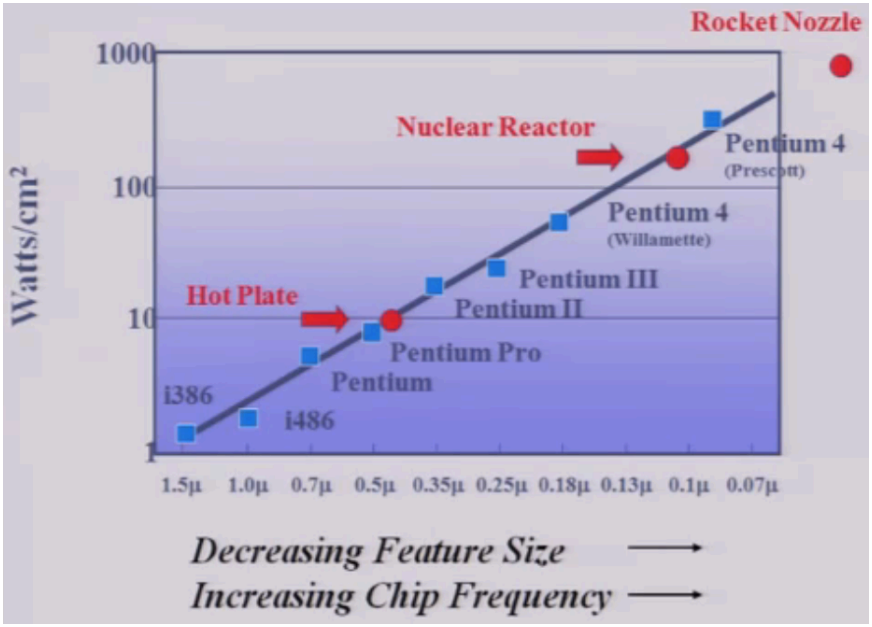


Figure 2.1: Rise of multicore CPUs [25]

2.1.1 Multicore CPUs

Development of CPUs has been governed by Moore's Law since the days of unicore microprocessors [30]. However, as shown in figure 2.1, leveraging Moore's Law to increase the CPU clock frequency resulted in processors that generate as much power per square centimeter as in a nuclear reactor and thus, hitting a *power wall*. If the philosophy of increasing CPU frequency would have persisted, then the amount of power generated per square centimeter by a microprocessor could have been similar to that generated by a rocket nozzle. To mitigate this disaster, chip designers leveraged Moore's Law to increase the number of cores rather than increasing the frequency on the CPU and hence, multicore CPUs came into existence.

Over the years, the multicore CPU has evolved into an efficient, albeit complex, architecture. A modern multicore CPU consists of multiple levels of hierarchy; it is a multi-socket, multi-

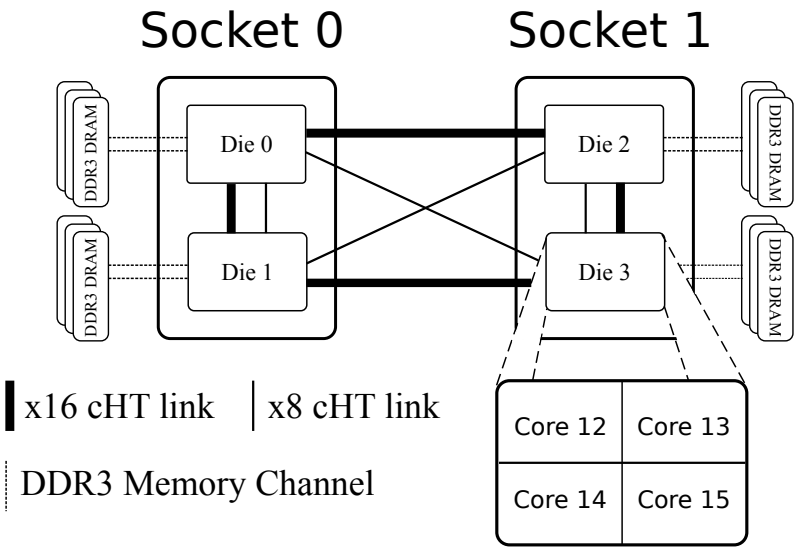


Figure 2.2: Block diagram of a modern multicore CPU

die, multicore microprocessor, as shown in figure 2.2. In addition, the presence of multiple on-die memory controllers results in a non-uniform memory access architecture (NUMA). Each scalar core consists of four-wide SIMD units that support a wide range of vector instructions [24]. In addition to multiple cores, there exist three levels of caches to improve the performance of the memory subsystem. Each core contains L1 and L2 levels of cache, while the L3 cache is shared among the cores of a die.

To improve single-thread performance, the CPU employs an out-of-order, superscalar architecture, thereby increasing the instruction-level parallelism. It also consists of a sophisticated branch prediction unit to reduce the performance impact of branch misprediction. CPUs provide for fast synchronization operations as well as efficient, in-register, cross-lane SIMD operations. However, they lack the presence of scatter/gather operations for non-contiguous memory accesses.

2.1.2 Graphics Processing Units

The widespread adoption of compute-capable graphics processing units (GPUs) in desktops and workstations has made them attractive as accelerators for high-performance parallel programs [12]. The increased popularity has been assisted by (i) the sheer computational power, (ii) the superior price-performance ratio and, (iii) the compelling performance-power ratio.

GPUs have more transistors devoted to performing computations than for caching and managing control flow. This means that on a GPU, computations are essentially free but memory accesses and divergent branching instructions are not. Thus, the key aspect of GPU programming is to successfully hide the latency of memory accesses with computations. The GPU does this by performing massive multithreading, thereby allowing us to initiate thousands of threads such that when one of the threads is waiting on a memory access, other threads can perform meaningful work. Multiple threads on a GPU can execute the same instruction. This type of architecture is known as Single Instruction Multiple Thread (SIMT), and it makes the GPU very suitable for applications which exhibit data parallelism, i.e., the operation on one data element is independent of the operations on other data elements. GPUs are primarily developed by two vendors: AMD and NVIDIA. The GPUs from each vendor is significantly different from the other, in terms of architecture. A description of each is presented in following sections.

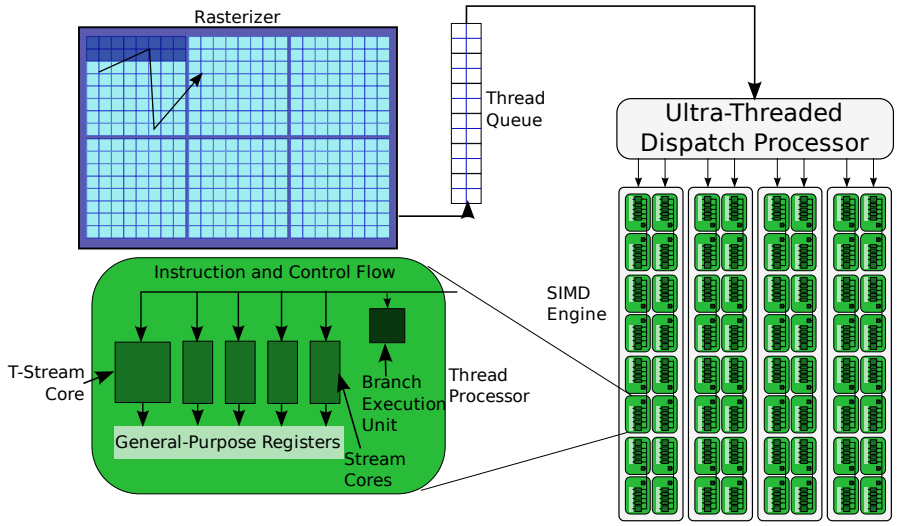


Figure 2.3: Block diagram of an AMD stream processor and thread scheduler

AMD GPUs

AMD GPUs follow a classic graphics design which makes them highly tuned for two-dimensional and image data as well as common image operations and single- precision, floating point math. A block diagram of an AMD GPU can be found in figure 2.3.

In this case, the compute unit is known as a *SIMD Engine* and contains several thread processors, each containing four standard processing cores, along with a special-purpose core called a T-stream core and *one* branch execution unit. The T-Stream core executes certain mathematical functions in hardware, such as transcendentals like $sin()$, $cos()$, and $tan()$. As shown in figure 2.3, there is only one branch execution unit for every five processing cores, thus any branch, divergent or not, incurs some amount of serialization to determine which path each thread will take. The execution of divergent branches for all the cores in a compute unit is performed in a *lock-step* manner, and hence, the penalty for divergent branches can

be as high as *80 cycles per instruction* for compute units with 80 cores. In addition, the processing cores are vector processors, as a result of which, using vector types can produce material speedup on AMD GPUs.

Recent GPUs from AMD are made up of a large number of processing cores, ranging from 800 to 1600 cores. As a result, humongous numbers of threads are required to be launched to keep all the GPU cores fully occupied. However, to effectively run many threads, one needs to keep a check on the amount of registers used per thread. The total number of threads that can be scheduled at a time cannot exceed the number of threads that can fit in the registers in the register file. Hence, to support the execution of many threads, AMD GPUs have a considerably large register file, e.g., 256 KB, on the latest generation of GPUs.

Another unique architectural feature of AMD GPUs is the presence of a *rasterizer*, which makes them more suitable for working with two-dimensional matrices of threads and data. Hence, accessing scalar elements stored contiguously in memory is not the most efficient access pattern. However, the presence of vector cores makes accessing the scalar elements in chunks of 128 bits slightly more efficient. Loading these chunks from image memory, which uses the memory layout best matched to the memory hardware on AMD GPUs, also results in large improvement in performance.

NVIDIA GPUs

NVIDIA GPUs consist of a large number of compute units known as Streaming Multiprocessors (SMs), and each SM consists of a certain number of *scalar* Streaming Processor (SP) cores. On each SM, up to a thousand threads can be run, thus enabling it to be a massively parallel architecture. The minimum unit of execution on a NVIDIA GPU is called a warp, which is a group of 32 threads. If threads of a warp diverge and follow different execution paths, then the execution of these threads would be serialized, thereby slowing down the execution. Therefore, optimal performance is obtained when all the threads in a warp execute the same instructions.

On the NVIDIA GT200 architecture, each SM has on-chip shared memory. The shared memory enables extensive reuse of on-chip data, thereby greatly reducing off-chip traffic and improving application performance. On the latest NVIDIA Fermi architecture, each SM owns a configurable on-chip memory that can act as either shared memory or as a L1 cache. The device memory consists of thread local and global memory, both of which reside off-chip. On the GT200 architecture, the global memory has been divided into 8 partitions of 256-byte width. If all active warps on the GPU try to access the same global memory partition then, their accesses are serialized, which in turn, adversely affects performance. This phenomenon is known as the Partition Camping problem [4]. NVIDIA Fermi GPUs also consist of a L2 cache also which was missing on the previous architectures. Figure 2.4 lays out the architectural difference between NVIDIA's Fermi and the GT200 architectures.

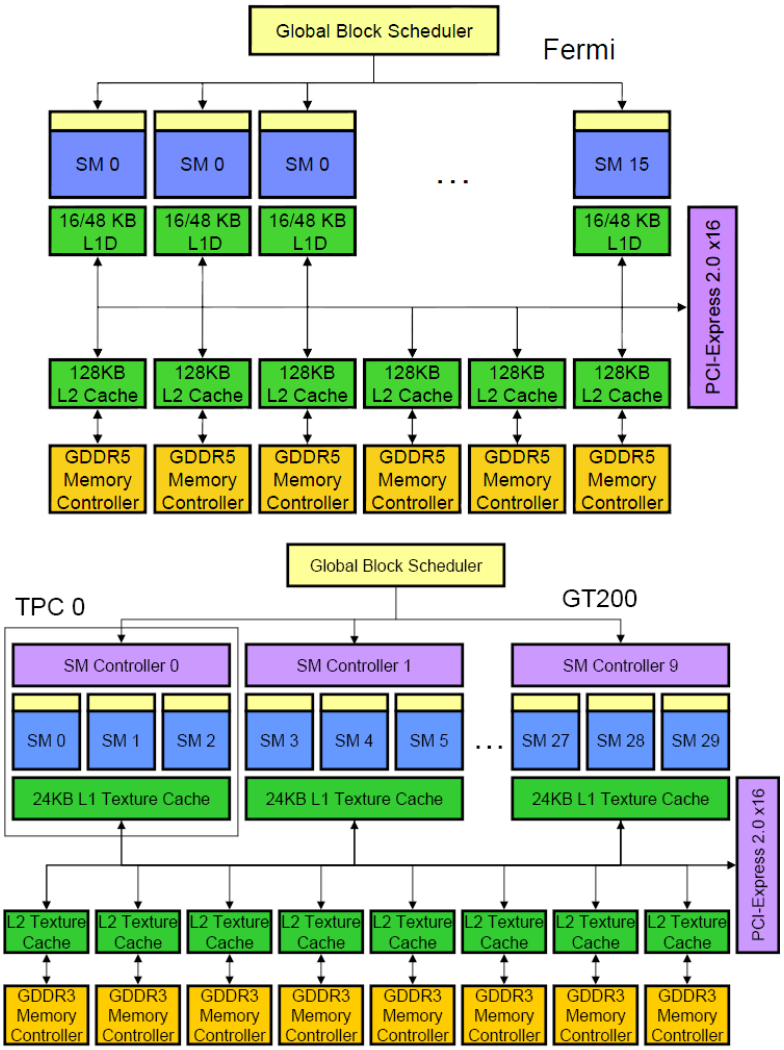


Figure 2.4: Overview of NVIDIA GPU architectures

2.2 Programming Environments

Various programming languages exist that enable the implementation of applications on multicore CPUs. We chose to use the C language along with *hand-tuned* SSE instructions due to high performance efficiency of C.

Implementing applications on the GPUs has been assisted by the development of frameworks

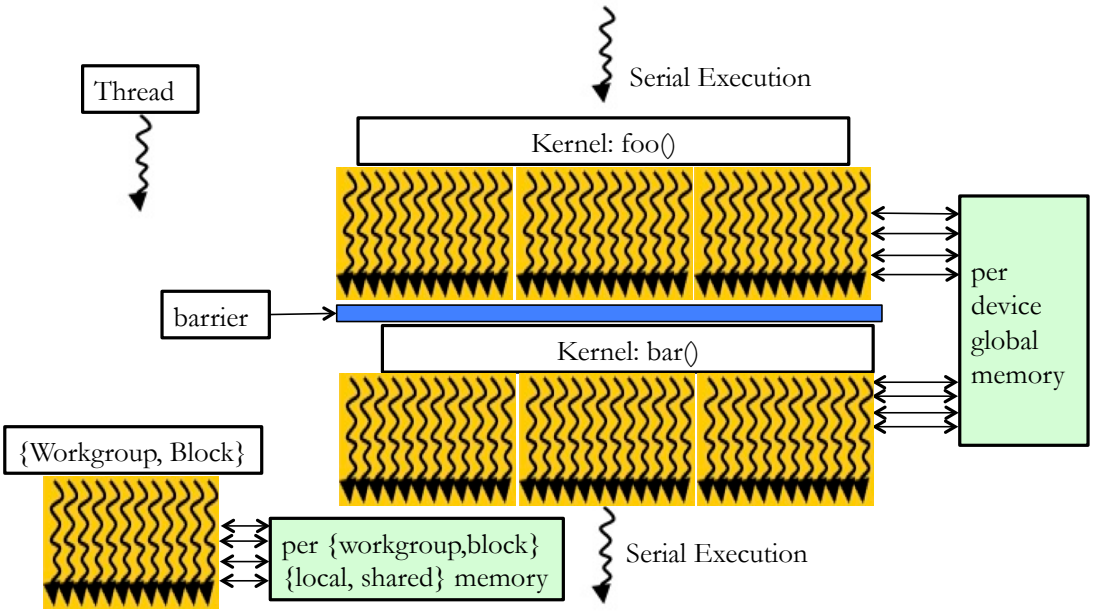


Figure 2.5: Programming environments: {OpenCL and CUDA}

like OpenCL and CUDA [18, 32]. Programs written in OpenCL can execute on a multitude of platforms like multicore CPUs, GPUs and even the Cell Broadband Engine, whereas programs written in CUDA can execute currently only on NVIDIA GPUs. OpenCL and CUDA are very similar; the main difference is in the terminology. Figure 2.5 portrays the differences, and the following two sections describe the OpenCL and CUDA programming environments in detail.

2.2.1 OpenCL

OpenCL is currently the only open standard language for programming GPUs and is supported by all major manufacturers of GPUs and some manufacturers of CPUs, including AMD, Intel, and most recently ARM. An OpenCL application is made up of two parts,

C/C++ code that is run on the CPU and OpenCL code in a C-like language on the GPU. The CPU code is used to allocate memory, compile the OpenCL code, stage it, and run it on the GPU. The OpenCL code is made up of kernels, which are essentially functions designated to be run on GPU when invoked by the CPU. Each kernel is in turn made up of a one- to three-dimensional matrix of work groups, which consist of one- to three-dimensional matrices of threads. The kernel is the atomic unit of execution as far as the CPU is concerned, but other levels become necessary on the GPU. While a kernel is a cohesive entity, the work groups within a kernel are not designed to communicate with each other safely. Only threads within a work group are capable of synchronizing with one another.

2.2.2 CUDA

CUDA is a framework developed by NVIDIA, which facilitates the implementation of general-purpose applications on NVIDIA GPUs. It provides a C-like language with API's to be used for various purposes. CUDA programs are executed by a *kernel*, which is essentially a function defined on the GPU and launched from the CPU. CUDA logically arranges the threads in up to three-dimensional blocks, which are further grouped into two-dimensional grids. Each thread on the GPU has its own ID, which provides for a one-to-one mapping. Each block of threads is executed on one SM, and these threads can share data via shared memory. This makes synchronization within a thread block possible, but not across thread blocks.

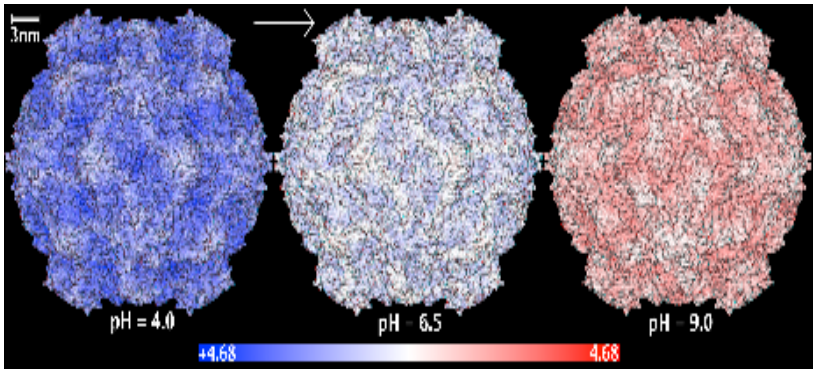


Figure 2.6: GEM: Output

2.3 Application

To illustrate the efficacy of proposed optimization strategies, we applied them against a production-level, molecular modeling application called *GEM* [17]. GEM allows the visualization of electrostatic potential along the surface of a macromolecule, as shown in figure 2.6. The working of GEM (figure 2.7) is as follows. The electrostatic potential, ϕ_i, is computed based on a single charge located inside an arbitrary biomolecule, and $\mathbf{d_i}$ is the distance from the source charge, $\mathbf{q_i}$, to the point of observation. The molecular surface is projected by a small distance, $\mathbf{p}$, and the distance between the point of observation and the center is defined as $\mathbf{r} = \mathbf{A} + \mathbf{p}$, where $\mathbf{A}$ is the electrostatic size of the molecule. GEM belongs to the N-Body class of applications (or dwarfs[1]), but while most N-Body applications are all-pairs computations given a single list of points, GEM is an all-pairs computation between two lists. The input to it is a list of all atoms in the molecule along with their charges and a list of pre-computed surface points or *vertices* for which the potential is desired. The algorithm is presented in figure 2.8.

[1]A dwarf or a motif is a common computation or communication pattern in parallel algorithms [9].

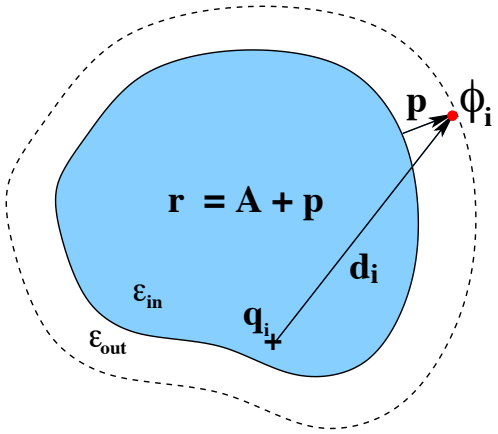


Figure 2.7: GEM: High-level overview

From the algorithm, we note that GEM is an inherently data-parallel application, i.e., the potential at one surface point can be computed independently of the computation of potential at other surface points. Taking advantage of this artifact, we assigned each GPU thread with the task of computing the electrostatic potential at one surface point in our GPU implementation of GEM. Once all the threads finish with computing the potential at their respective surface point, a reduce (sum) operation is performed to calculate the total surface potential of the molecule. The algorithm for the GPU implementation is presented in figure 2.9. From the algorithm, for each thread, the coordinates of the atoms as well

```
GEM-CPU(){
    /* Iterate over all vertices */
    for v = 0 to #SurfacePoints do {
        /* Iterate over all atoms */
        for a = 0 to #Atoms do {
            /* Compute potential at each vertex
            due to every atom */
            potential += calcPotential( v, a )
        }
    }
}
```

Figure 2.8: GEM: Algorithm for CPU implementation

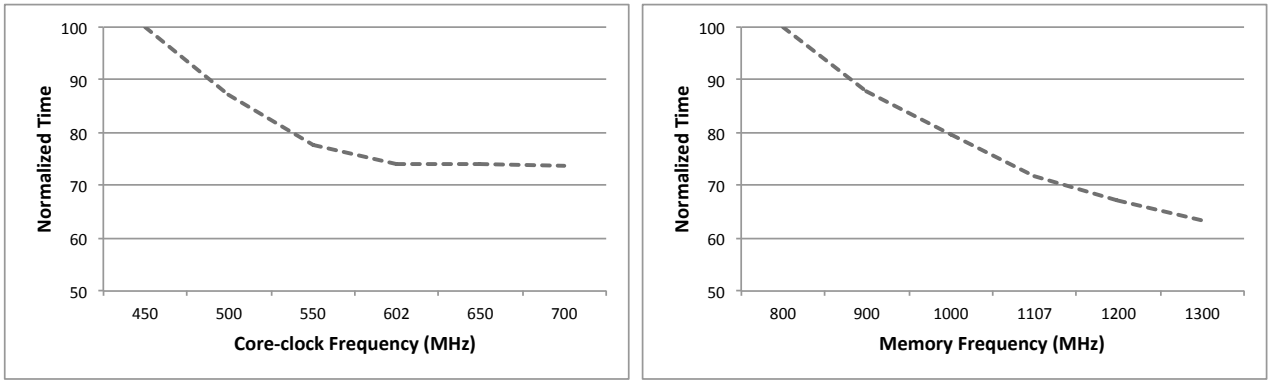
```
// Host function on the CPU */
GEM-GPU(){
    int tot_potential, int potential[#SurfacePoints]
    /* launch GPU kernel with as many
    threads as #SurfacePoints */
    GPU_kernel( atoms, potential )
    /* Read back potential computed on the GPU */
    /* Iterate over the potentials read back
    from the GPU to compute total potential*/
    for v = 0 to #SurfacePoints do {
        tot_potential = potential[v]
    }
}
// GPU Kernel
GPU_kernel( atoms, potential ) {
    tid = thread_id;
    /* Iterate over all atoms */
    for a = 0 to #Atoms do {
        /* Compute potential at each vertex
        due to every atom */
        potential[tid] += calcPotential( v, a )
    }
}
```

Figure 2.9: GEM: Algorithm for GPU implementation

as the vertex need to be accessed #atoms times from the memory. Since for a thread, the vertex coordinates do not change, caching them should improve application performance as it would reduce the number of slow memory accesses. Also, the potential at each vertex is updated for every atom and is stored in an accumulator variable. This variable can be cached or stored in a register to provide further performance improvement.

We performed memory bounds checking of the GPU implementation to better understand the effectiveness of GPU optimization strategies. Specifically, we studied the change in execution time of the application with varying GPU core-clock and memory frequencies.

In figure 2.10a, the GPU memory frequency is kept constant at the default value of 1107 MHz while the core-clock frequency is varied. We note that the execution time decreases steadily with the increase in core-clock frequency. However, this decrease levels off around



(a) Time vs GPU Core-Clock Frequency (b) Time vs GPU Mermory Frequency

Figure 2.10: GEM: Memory bounds checking

550 MHz; increasing the clock frequency further, even over-clocking, has no effect on the execution time. Therefore, from the figure, we infer that the application is *compute bound* until the clock frequency of 550 MHz, and after that it starts to become *memory bound*. We also note that at the default core-clock frequency of 602 MHz, there is a possibility of the application being *memory bound*.

To corroborate the claim made, we plot figure 2.10b, where the core-clock frequency is kept constant at the default value of 602 MHz while the memory frequency is varied. From the figure, we note that execution time decreases steadily with the increase in memory frequency. In this case, even over-clocking the memory of the GPU helps in the reduction of execution time. Therefore, we infer that the application is *memory bound* at the default core-clock and memory frequencies of the GPU.

Subsequent chapters present optimization strategies for various heterogeneous computing systems.

Chapter 3

Mapping and Optimization on Multicore CPUs

In this chapter, we first discuss the optimization strategies for multicore CPUs and then present the performance benefits that were obtained by applying strategies, both in isolation and in combination, to GEM.

3.1 Optimizations

There used to be era in computer science when software developers could bank on chip designers to make faster CPUs, which in turn made software programs run faster. However, this *free lunch* is now over [45]. Although the CPUs are no longer getting faster, they employ complex hardware to execute powerful instructions, perform pipelining, branch-prediction,

execute multiple instructions in the same clock cycle and even reorder the instruction stream for out-of-order execution. All these assist in making programs run faster, though, only if programs are appropriately optimized. Today's CPU also consist of larger and multiple caches which aid in reducing the number of accesses to the slow main memory. However, efficient utilization of caches is imperative for superior performance. Optimization strategies for a sequential program are presented in section 3.1.1.

Nowadays, CPUs consist of multiple cores instead of just a single core, but programs need to be parallelized to efficiently use the multiple cores. However, designing parallel programs is a daunting task, as parallel programs may suffer from load balancing, sequential dependencies and synchronization issues [38]. A programmer has to understand these bottlenecks and go through the arduous procedure of optimizing one's program to reap supreme performance. We discuss parallelization across multiple cores in section 3.1.2.

3.1.1 Optimizations for Sequential Program

In this section, we discuss optimization strategies that lead to improvement in single-thread performance of an application.

Compiler Optimizations

With the increase in complexity of CPU hardware, compilers have become more intelligent. Compilers improvise on the features of modern CPUs and optimize programs, though only

when the programmer explicitly asks the compiler to do so. For example, the GNU/GCC C compiler consists of flags like i) `-O1`, ii) `-O2`, iii) `-O3`, iv) `-march`, which auto-optimize the program during the compilation phase. Each `-O{1,2,3}` flag performs different types of optimizations like speculative branch-prediction, loop-unrolling, use of in-line functions, efficient register utilization, etc. The `-O3` flag generates the most optimized program. Almost all the optimizations performed by the compiler are targeted towards better instruction throughput.

Most modern CPUs consist of SIMD units which can execute the same instruction on multiple data at the same time. If the compiler is made aware of underlying CPU architecture using `-march` compiler flag, it takes advantage of these SIMD units and implicitly vectorizes the program, thereby improving performance.

SIMDization using Vector Intrinsics

Modern CPUs provide powerful instructions or vector intrinsics which enable short-vector data parallelism (aka SIMD) on a sequential program. These instructions are known by different names for different architectures, e.g., Intel calls them SSE instructions while on Power7 processors, these are known as VSX instructions. Most CPUs contain four-wide SIMD units and hence, use of vector intrinsics can theoretically provide *4-fold* performance improvement.

SIMDizing a program by the use of vector intrinsics is an arduous task on the part of

```
// A function to compute the sum of two arrays, arr1 and arr2
// It is assumed that length of these arrays, i.e., num_elem_arr, is a multiple of 4
int hand-tuned-SIMD( int *arr1, int *arr2, int num_elem_arr ) {
    int sum;
    /* declaring vector type variables */
    typedef int v4sf __attribute__ ( mode ( V4SF ) );
    v4sf acc_sum, data1, data2;
    /* flushing the accumulator */
    acc_sum = __builtin__ia32__xorps( acc_sum, acc_sum );

    for( i = 0; i < num_elem_arr; i += 4 ) {
        /* loading data from arrays into vectors */
        data1 = __builtin__ia32__loadups( &arr1[i] );
        data2 = __builtin__ia32__loadups( &arr2[i] );
        /* adding two vectors and storing result in the accumulator */
        acc_sum += __builtin__ia32__addps ( data1, data2 );
    }
    /* storing the final sum in a scalar varible */
    sum = acc_sum[0] +  acc_sum[1] + acc_sum[2] + acc_sum[3] ;
    return sum;
}
```

Figure 3.1: SIMDization using *hand-tuned* vector intrinsics

the programmer, as one has to explicitly manage the vectors and also write programs in a construct that is not very lucid to understand. Figure 3.1 depicts a function that computes the sum of two integer arrays and has been *SIMDized* using Intel's SSE instructions. It can be noted that the sum of four integers from each array is computed by a *single* instruction, rather than four instructions as in usual case. The final result is then stored into a scalar variable by accessing computing the sum of each padded scalar in the vector accumulator.

Cache Blocking

CPUs rely heavily on caches to hide the main memory latency. However, minimization of *cache misses* is necessary to reap maximum benefits from the cache. Cache blocking is one technique that enables better cache utilization. Cache blocking improves application performance by enhancing the temporal and spatial locality of data accesses in the cache.

Cache blocking turns out be most beneficial for programs that have random memory access and high operation-count-to-memory-access ratio. Random memory access pattern is the primary requirement for cache blocking. Contiguous memory accesses increase the spatial locality and hence, result in greater number of cache hits. Greater operation-count-to-memory-access ratio provides the scope of using a technique for greater data re-use.

An application where cache blocking comes in handy is matrix-matrix multiplication. For matrix-matrix multiplication, matrices have orthogonal access patterns and also require memory to be accessed in strides, thereby satisfying the primary requirement for cache blocking. Also, in matrix-matrix multiplication, arithmetic operation count scales as $O(N^3)$ while, memory access count scales as $O(N^2)$. Therefore, the operation-count-to-memory-access ratio is $O(N)$ and hence, satisfies the second requirement for cache blocking. To implement cache blocking, the loop nests are restructured such that the computation proceeds in contiguous blocks chosen to fit in the cache. Choosing the appropriate block size determines how much beneficial cache blocking would be for a particular application.

3.1.2 Parallelization

Nowadays, it is difficult to find a CPU which does not have multiple cores. Hence, for continued improvement in application performance, it is imperative that applications should be parallelized. Programming environments like Pthreads, OpenMP, and OpenCL, assist in implementing a parallel version of an existing sequential program [2, 3]. When parallelizing

an application, the programmer has to deal with issues like load balancing, data dependencies and synchronization. These issues are dealt in different ways in each of the above mentioned programming environments. In Pthreads, locks and semaphores are explicitly used to avoid race conditions and deadlocks. On the contrary, OpenMP is pragma based and provides an abstraction to the programmer for dealing with the above mentioned issues.

3.2 Results

In this section, we discuss the performance benefits that were realized by implementing the optimization strategies in GEM.

To test the efficacy of the optimizations proposed for sequential programs, we used an Intel E5405 Xeon CPU running at 2.0 GHz with 8 GB DDR2 SDRAM. The operating system on the host is a 64-bit version of Ubuntu running the 2.6.28-19 generic Linux kernel. To compile our program we used the GNU/GCC compiler version 4.3.3. For the purpose of demonstrating the scalability of parallel version of GEM, we used a 16-core AMD Magny Cours CPU with 24 GB DDR3 SDRAM and a 2.6.35-11 Linux kernel. We implemented a parallel version of GEM using the OpenMP programming environment, which is discussed in section 3.2.2.

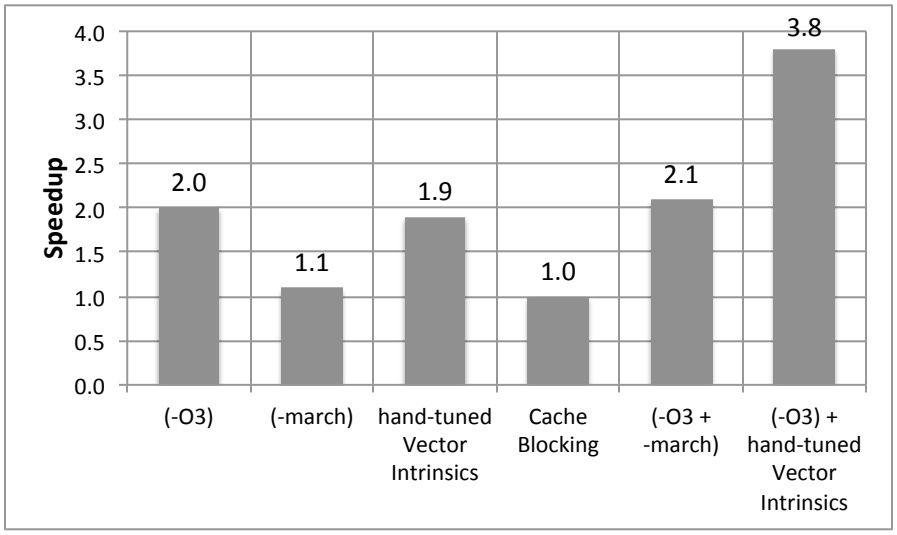


Figure 3.2: Speedup due to optimizing the sequential program. Baseline: Basic sequential implementation on CPU

3.2.1 Optimizations for Sequential Program

Figure 3.2 depicts the speedup obtained due to various optimizations, over a basic sequential implementation, on the CPU. It can be noted that the usage of `(-O3)` optimization flag of the GNU/GCC compiler results in a *2-fold* performance improvement over the basic implementation. This is because the `(-O3)` flag improves the instruction throughput of the application. Using compiler optimization flags like `(-march)` that lead to implicit vectorization of the program did not provide much benefit.

As implicit vectorization did not achieve significant improvement, we optimized GEM with *hand-tuned* vector intrinsics to implement an explicitly vectorized version. Specifically, we used Intel's SSE instructions and achieved a *1.9-fold* speedup over the basic implementation. Theoretically, SSE instructions are expected to provide 4-fold benefit due to the padding of four scalars into a vector. However, the process of vectorization an application incurs a

hidden cost which is not always amortized. It is due to this reason that our achieved speedup is not 4-fold.

GEM algorithm inherently makes an efficient use of the cache and hence, nothing more can be achieved via cache blocking. From the algorithm in figure 2.8, it can be noted that there are no random memory accesses and, the spatial and temporal locality is extremely high. Also, each surface point accesses the atomic coordinates in a sequential fashion, thereby, resulting in a large number of cache hits. Therefore, in figure 3.2, speedup due to cache blocking is *1-fold* over the basic implementation.

Using the above mentioned optimizations in combination resulted in multiplicative speedup, as shown in the figure. Using `(-O3)` and hand-tuned vector intrinsics in conjunction provided us with an improvement of *3.8-fold*, and it also resulted in being the most optimized CPU implementation of GEM.

3.2.2 Parallelization

Parallelization of an application is imperative to reap optimum performance on the modern multicore CPU. We parallelized our most effective CPU implementation, i.e., one optimized with `(-O3)` compiler flag & vector intrinsics.

Amdahl's Law dictates the scalability of a parallelized application and hence, gives an idea as to what would be the maximum expected improvement due to parallelization [6]. Primary requirement of using Amdahl's Law is to figure out the part of application that can

be implemented in parallel, and also how much proportion of the total execution time is constituted by the parallel part. This information can then be plugged in equation 3.1 to compute the speedup 'S', that can be achieved by parallelizing an application.

$$\begin{aligned} S &= \frac{1}{(1-P) + P/N} \qquad (3.1) \\ \text{where, } P &= \text{proportion of program that can parallelized} \\ \text{and, } N &= \text{number of cores} \end{aligned}$$

Therefore, we profiled GEM and figured out that the portion of the application that can be parallelized, i.e., the computational kernel, constitutes around *97%* of the total execution time. Using this information, we computed table 3.1, which depicts the amount of speedup that can be obtained by increasing the number of cores in the CPU. As noted in the table, the impact of increasing the number of cores is not significant after a certain threshold. This is due to the fact that after certain number of cores, sequential part of the application becomes the bottleneck and hence, parallelization further would not improve performance.

Figure 3.3 illustrates the scalability of GEM. It depicts the amount of speedup obtained with the increase in number of cores. From the figure, we note that the achieved speedup is very close to that predicted by Amdahl's Law. The figure also portrays the *ideal* speedup that

Table 3.1: Amdahl's Law: Speedup vs #Cores

	Number of Cores			
	10	100	1,000	10,000
Speedup	7.8	25.2	32.3	33.2

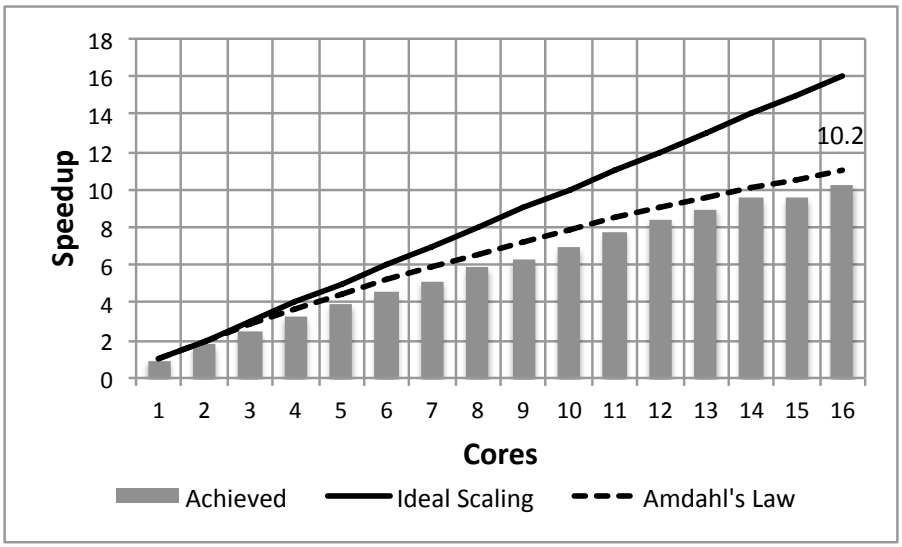


Figure 3.3: Scalability. Baseline: CPU sequential implementation optimized by `(-O3)` and vector intrinsics

would have been obtained if 100% of the application could be parallelized. Parallelization of GEM across 16 cores of the CPU resulted in an additional *10.2-fold* speedup.

Based upon the analysis presented thus far, parallelization of GEM across *10,000* CPU cores would result in a speedup of *126.2-fold* over the basic CPU implementation. However, procurement of 10,000 CPU cores would entail a huge investment and also getting access to 10,000 cores at the same time is extremely difficult. Therefore, to improvise further we implemented GEM on the GPU platform. GPUs provide similar computational benefits as that of a massive supercomputer albeit with a fairly low financial burden. For example, a 8-GPU cluster costing a few thousand dollars, can simulate 52 ns/day of the JAC Benchmark as compared to 46 ns/day on the Kraken supercomputer, housed at Oak Ridge National Lab and which costs *millions* of dollars [23].

In subsequent chapters, we discuss optimization strategies for the graphics processing units.

Chapter 4

Mapping and Optimization on AMD GPUs

In this chapter, we first discuss the optimization strategies for the AMD GPUs and then present the performance benefits that were obtained by applying strategies, both in isolation and in combination, to GEM.

4.1 Optimizations

In this section, we discuss optimization strategies that improve application performance on AMD GPUs. These strategies have been devised by a careful analyses of the underlying GPU architecture. The most intuitive optimization for any GPU platform is making sure that none of the computational resources go under-utilized. On a GPU, this is done by

```
//before kernel splitting
int main () {
    ...
    int a = 5;
    clEnqueueNDRangeKernel(..., kernel, ...);
    ...
}

__kernel void work_kernel ( int a ) {
    /* conditional inside the kernel */
    if ( a >= 5 ) {
        /* do work 1 */
    } else {
        /* do work 2 */
    }
}

//after kernel splitting
int main() {
    ...
    int a = 5;
    /* splitting the kernel into two */
    if ( a >= 5 ) {
        /* calling kernel #1 */
        clEnqueueNDRangeKernel(..., kernel1, ...);
    } else {
        /* calling kernel #2 */
        clEnqueueNDRangeKernel(..., kernel2, ...);
    }
    ...
}

/* kernel #1 */
__kernel void work_kernel1 ( int a ) {
    /* do work1 */
}
/* kernel #2 */
__kernel void work_kernel2 ( int a ) {
    /* do work2 */
}
```

Figure 4.1: Kernel splitting

executing enough threads such that none of the GPU cores are ever idle and the *occupancy* is always maximum. We call this optimization *Max. Threads*. Other optimization strategies are discussed in subsequent sections.

4.1.1 Kernel Splitting

In section 2.1.2, it is mentioned that AMD GPUs consist of only one branch execution unit for every five processing cores. Further it mentions that as a result, even non-divergent branches can cause a significant performance impact on AMD GPUs. Oftentimes, even when the outcome of a conditional can be determined before a GPU kernel is called, the conditional is frequently pushed into the GPU kernel anyway in order to simplify the understanding of the code. Figure 4.1 shows a simple example of this phenomenon, which we denote *kernel splitting*. On CPUs, the performance lost to branching is minimal due to speculative execution and branch prediction. On NVIDIA GPUs the cost is higher, but not as much as on the AMD GPUs which can be reasoned as follows. On NVIDIA GPUs, threads in a warp (i.e. 32 threads) are required to follow the same execution path whereas on AMD GPUs, threads in a wavefront (i.e. 64 threads) are required to follow the same execution path. This requirement doubles the probability of the occurrence of divergence on AMD GPUs and hence, the cost of branching is greater on AMD GPUs. Presence of branching can lead to a difference of up to 30% on AMD architectures, as showed in [7].

Kernel splitting is implemented by moving the conditionals, result of which are predetermined at the beginning of a kernel, to the CPU. The kernel is then *split* into two parts, each part executes a different branch of that conditional. Despite the simplicity of the optimization, implementing it in practice can be complicated. For example, kernel splitting results in 16 different kernels in GEM.

```
//before using register accumulator
__kernel void kernel( int count, __global int* global_arr ) {
    for ( int i = 0; i < count; i++ ) {
        /* incrementing gmem for every iteration */
        global_arr[threadIdx.x] ++;
    }
}

//after using register accumulator
__kernel void kernel( int count, __global int* global_arr ) {
    /* preloading gmem location in a register, 'g' */
    int g = global_arr[threadIdx.x];
    for ( int i = 0; i < count; i++ ) {
        /* incrementing 'g' */
        g++;
    }
    /* updating gmem location with value of 'g' */
    global_arr[threadIdx.x] = g;
}
```

Figure 4.2: Register accumulator

4.1.2 Local Staging

Local staging alludes to the use of *on-chip* memory present on the GPU. Subsequent accesses to the on-chip memory are more efficient than accessing data from its original location in the global memory. Conventional wisdom states that prefetching and reusing data in constant and local memories is extremely efficient. On AMD GPUs this is true when large amounts of data is seldom reused or when data loaded into local or constant memory is reused by more than one thread in the active work group. In general local memory and constant memory are faster than global memory, allowing for speedups, but they are not always the best option.

Local and constant memory on AMD GPUs are significantly slower to access than the registers. Hence, for small amounts of data and for data that is not shared between threads, registers are much faster. Using too many registers in a thread can produce *register pressure* or a case where less threads can be active on a compute unit at a time due to lack of registers.

Reduction in the number of active threads can degrade performance, but since AMD GPUs have a large register file, i.e., 256k, it is frequently worth using extra registers to increase the memory performance. One case where this is especially true is for accumulator variables, as shown in figure 4.2. If an algorithm includes a main loop in each thread, which updates a value once each time through the loop, moving that accumulator into a register can make a significant difference in performance, as will be discussed in section 4.2. It is worth noting that there is a wall with register usage on AMD GPUs. Beyond 124 registers per thread, the system starts putting data into *spill* space, which is actually global memory, and hence, can degrade performance.

4.1.3 Vector Types

Vector types in OpenCL are designed to represent the vectors used by SIMD execution units like SSE or AltiVec. Vector is a single large word of 64 to 512 bits in size containing smaller scalars. Generally the most used type of this class is the `float4`, as it matches the size of the registers on an AMD GPU as well as the size of an SSE word. AMD GPUs are optimized for memory accesses of 128 bits as well as computation on vectors of 2-4 floats. However, some math is not optimized, specifically the transcendental functions provided by the T-Stream core. The overhead of unpacking the vector and doing the transcendentals is higher than just doing all the math with scalars in some cases.

Even when scalar math is faster, loading the memory in float2 or float4 is more efficient than

```
//before vector loads
__kernel void kernel( __global float* global_x, __global float* global_y,
                      __global float* global_z, __global float* global_c ) {
    float local_x, local_y, local_z, local_c;
    int i;
    /* loading scalars from gmem */
    local_x = global_x[i];
    local_y = global_y[i];
    local_z = global_z[i];
    local_c = global_c[i];
}

//after vector loads
__kernel void kernel ( __global float* global_arr ) {
    float4 local4;
    int i;
    /* loading vectors from gmem */
    local4 =  vload4( i, global_arr );
}
```

Figure 4.3: Vector loads

loading scalars, as shown in figure 4.3. Prefetching with vector loads followed by unrolling a loop to do the math in scalars, and then storing the vector can be a significant improvement.

4.1.4 Image Memory

On AMD GPUs, image memory is an efficient way to increase performance of memory accesses, as it acts as a cache when there is high data reuse in the program. Image memory offers many transformations meant to speedup the access of images stored within it and also is equally capable of reading simple quads, or `float4` vectors. As mentioned above, loading larger vector types from memory is more efficient, adding to it, the benefits of caching and more efficient memory access patterns offered by image memory, make these two a potent combination. In addition, the changes necessary to use image memory for read-only input data is minimal in the kernel, as shown in figure 4.4. It only requires modification of the initial load of the `float4`.

```
//before image memory loads
__kernel void kernel ( __global float* global_arr ) {
    float4 local4;
    /* loading from gmem */
    local4 =  vload4( 0, global_arr );
}

//after image memory loads
__kernel void kernel ( __read_only image2d_t image4 ) {
    int i;
    /* initial setup */
    const sampler_t smp = CLK_NORMALIZED_COORDS_FALSE |
                          CLK_ADDRESS_NONE |
                          CLK_FILTER_NEAREST;
    /* setting up coordinates for image memory loads */
    int2 coord1 = (int2)((i % 8192), (i / 8192));
    float4 local4;
    /* loading from image memory */
    local4 =   read_imagef( image4, smp, coord1 );
}
```

Figure 4.4: Image memory loads

4.1.5 Optimizations in Combination

When applying optimizations, what happens when the optimizations are combined? If two optimizations can improve performance of a base application, it is very alluring to assume that they will "stack" or combine to create an even faster end result when applied together. All the optimizations presented to this point, produce some amount of benefit when applied to basic code but not multiplicative benefit as one would expect. Given the fact that they all benefit the basic implementation, it would stand to reason that using all of them together would produce the fastest application, however, this is not the case.

In the auto-tuning work of [16], Datta et al. had many optimizations to tune simultaneously, as we do here, and decided on an approach which was later referred to as *hill climbing* [47]. Essentially, hill climbing consists of optimizing along one axis to find the best performance, then finding the best parameter for the next axis after fixing the first, and so-on. This implies

that all the parameters are cumulative, or at least that order does not matter. While this is a popular, and at least marginally effective approach, we find that the inherent assumptions about optimization combination are not reasonable, at least when it comes to optimizing for GPUs. Further discussion of optimization stacking will be presented in section 4.2.

4.2 Results

In this section, we demonstrate the effectiveness of each optimization technique, in isolation as well as when combined with other optimizations. To accomplish this, we have used a AMD Radeon HD 5870 GPU. The 'Host Machine' consists of an E5405 Intel Xeon CPU running at 2.0 GHz with 8 GB DDR2 SDRAM. The operating system on the host is a 64-bit version of Ubuntu running the 2.6.28-19 generic Linux kernel. Programming and access to the GPU was provided using OpenCL version 1.1. For the accuracy of results, all the processes, which required graphical user interface, were disabled to limit resource sharing of the GPU.

Figure 4.5 portrays that there is around 30% performance improvement over the basic implementation, when maximum number of threads are launched on the GPU. This infers that higher *occupancy* on the GPU leads to better performance.

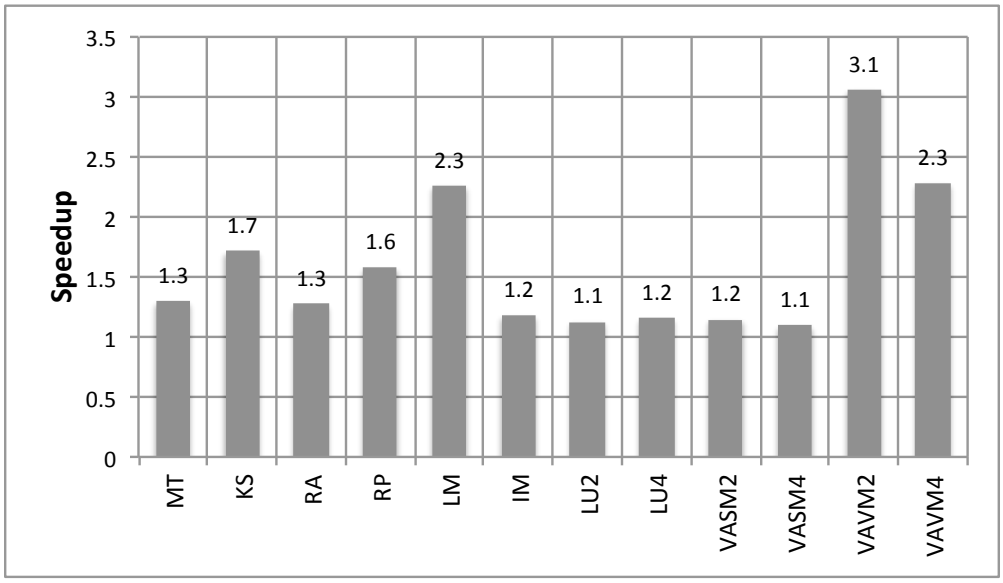


Figure 4.5: Speedup due to optimizations in isolation. Baseline: Basic OpenCL GPU implementation. MT: Max. Threads, KS: Kernel Splitting, RA: Register Accumulator, RP: Register Preloading, LM: Local Memory, IM: Image Memory, LU{2,4}: Loop Unrolling{2x,4x}, VASM{2,4}: Vectorized Access & Scalar Math{float2, float4}, VAVM{2,4}: Vectorized Access & Vector Math{float2, float4}

4.2.1 Kernel Splitting

In figure 4.5, we compare the performance results between the basic OpenCL implementation and the one optimized with *kernel splitting*. We find that kernel splitting provides *1.7-fold* performance benefit. This can be reasoned as follows. The AMD GPU architecture consists of only one branch execution unit for five processing elements, as discussed in section 2.1.2. Hence, branches incur a huge performance loss as the computation of the branch itself takes five times as long as computation of branches on similar architectures. One should strive to avoid branching on the GPU, and kernel splitting is one effective way of ensuring that.

4.2.2 Local Staging

In order to obtain optimum performance on GPUs, thread utilization should be improved. This can be achieved by reducing the number of registers utilized per thread, since more registers mean fewer threads in the kernel. However, register file size of present generation AMD GPUs is quite large and hence, one should not strictly inhibit the use of registers. We achieved superior performance by making explicit use of registers in our computational kernel. GEM involves accumulating the atomic potential at every vertex of the molecule. Rather than updating the intermediate result in global memory, we used a *register accumulator*. This approach provided us with a *1.3-fold* speedup over the basic implementation, as shown in figure 4.5. Using registers to preload data from global memory is also deemed to be favorable. Preloading provides up to *1.6-fold* performance benefit over the basic implementation. The kernel incorporates high data reuse as same data is loaded from within a loop and hence, preloading this data in a register and using it within the loop provides substantial performance benefit.

Improvement due to the use of *local memory* is almost *2.3-fold* over the basic implementation. Local memory is an on-chip scratch pad memory present on each SIMD unit of the GPU. It is appropriate to use local memory when there is high data re-use in the kernel which as mentioned, is true for GEM. Performance benefit obtained due to the use of local memory is *1.4 times* more than that obtained by register preloading. This behavior of the GPU is aberrant, as one would expect register preloading to be more beneficial than using local memory, given the fact that register file is the fastest on-chip memory.

4.2.3 Vector Types

Loop unrolling reduces the number of dynamic instructions in a loop, such as pointer arithmetic and "end of loop" tests. It also reduces branch penalties and hence, provides better performance. Figure 4.5 presents the performance benefit obtained by *explicit* two-way and four-way loop unrolling. As four-way unrolling reduces the dynamic instruction count by a factor of two more than two-way unrolling, it results in better performance.

Accessing global memory as *vectors* proves to be more beneficial than scalar memory accesses, as shown in figure 4.5. However, the length of vector, either `float4` or `float2`, which culminates in the fastest kernel performance may depend upon the problem size. From the figure, we note that `float2` is better than `float4` for GEM. Use of vectorized memory accesses pack in up to four scalar accesses into one vector access, thus conserving memory bandwidth as accesses which would have taken four memory accesses can now be completed with one access. Vectorized accesses also improve the arithmetic intensity of the program.

Use of vector math proves to be highly beneficial on AMD GPUs as, corroborated by figure 4.5. Vector math provides up to *3-fold* speedup in case of `float2`. AMD GPUs are capable of issuing five floating point scalar operations in a VLIW, and for most efficient performance, utilization of all VLIW slots is imperative. It is almost always the responsibility of the compiler to make sure that instructions are appropriately assigned to each slot. However, there might be instances when due to the programming constructs used, compiler may not do so. Use of vectorized math assists the compiler in ensuring that the ALU units

are completely utilized. It improves the mapping of computations on five-way VLIW and 128-bit registers of the AMD architecture. The dynamic instruction count is also reduced by a factor of the length of vector, since, multiple scalar instructions can now be executed in parallel.

4.2.4 Image Memory

Presence of L1 and L2 texture caches assists the image memory to provide additional memory bandwidth when data is accessed from the GPU memory. Using image memory in read-only mode results in the utilization of *FastPath* on AMD GPUs which leverages the presence of L2 cache [5]. However, if image memory is used in read-write mode, GPU sacrifices the L2 cache in order to perform atomic operations on global objects. Hence, one should be judicious in using read-write image memory only when necessary. We have used read-only image memory to store data that is heavily reused in the kernel. An improvement of up to *1.2-fold* over the basic implementation was obtained, as shown in figure 4.5.

4.2.5 Optimizations in Combination

In this section, we present performance benefits obtained when various optimization strategies are combined. We portray that optimizations when combined do not provide multiplicative performance benefit as one would expect. In addition, we show that optimizations which performed better in isolation may perform worse when used in conjunction with another op-

timization strategy. We also discuss the methodology that we followed in order to implement the most optimized implementation of GEM. We would like to state that the choice of optimizations is specific for GEM and may vary with other applications. Figure 4.6 depicts that speedup obtained due to a combination of Max. Threads and Kernel Splitting is *1.8-fold* over the basic implementation. However, if individual speedups are taken into consideration, then the multiplicative speedup should have been *2.2-fold* which is greater than what we actually achieved. Also from the figure, we note that a combination of Kernel Splitting and Register Preloading tends to be better than that of Kernel Splitting and Local Memory, though in isolation, local memory performs better than register preloading. This proves that a certain optimization strategy which performs better in isolation is not guaranteed to perform well in combination also. Similarly, when Vector Math is used in conjunction with other optimization techniques like Kernel Splitting, performance obtained is not as one would expect. Scalar Math tends to be better off with Kernel Splitting, though, in isolation the results are otherwise. In our ordeal to find out the best performing on-chip memory in combination, Kernel Splitting and Image Memory were used together in an another implementation. Our results indicate that Register Preloading is the most effective on-chip memory, which is in accordance to the fact that registers are fastest among the three.

To come up with the best combination of optimizations, we used an elimination policy, eliminating those optimizations that have been shown to perform worse. In this way, we could minimize the optimization search space. As Register Preloading and Scalar Math are known to perform better than the rest, we combined these two with Kernel Splitting and achieved

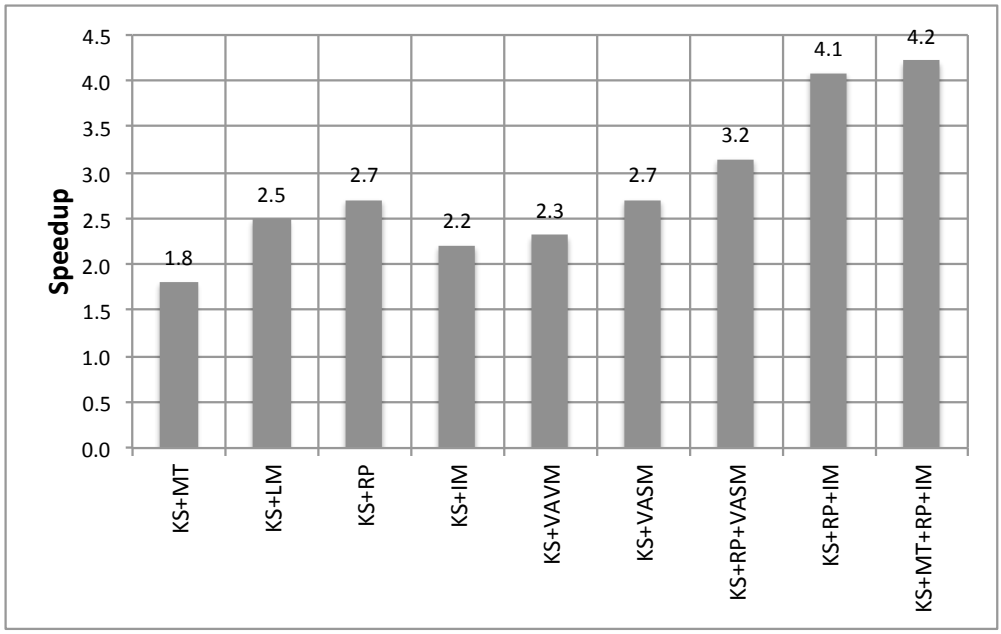


Figure 4.6: Speedup with optimizations in combination. Baseline: Basic OpenCL GPU implementation. MT: Max. Threads, KS: Kernel Splitting, LM: Local Memory, RP: Register Preloading, IM: Image Memory, VAVM: Vectorized Access & Vector Math, VASM: Vectorized Access & Scalar Math

a *3.2-fold* speedup over the basic implementation. Vectorized accesses have proved to be beneficial and hence, we used image memory as it internally loads vectors from memory and achieved a greater speedup, i.e, *4.1-fold* over the basic implementation. We then combined Kernel Splitting, Register Preloading, Image Memory and Max. Threads to achieve even greater speedup and also the most optimized implementation of GEM.

Due to GPUs being a *black-box*, it is not clear why these optimizations behave the way they do. This makes a strong case to urge the vendors to divulge more architectural details of GPUs. Figuring out these discrepancies in the results forms the basis of our future work.

In the following chapter, we discuss how to reap superior performance on another GPU architecture, i.e., NVIDIA GPUs.

Chapter 5

Mapping and Optimization on NVIDIA GPUs

In this chapter, we first discuss the well-known optimization strategies for NVIDIA GPUs and present the performance difference obtained when these optimizations are applied using either CUDA or OpenCL. We then discuss the partition camping problem and describe the performance prediction model that we developed to detect partition camping in GPU kernels.

5.1 Optimizations

In this section, we discuss optimization strategies that improve application performance on NVIDIA GPUs. Use of NVIDIA GPUs has been greatly facilitated by the evolution of CUDA programming model since early 2006 and hence, optimizing CUDA codes has been studied

extensively and many optimization strategies have become widely known [13,36,40–43,46,48]. We call these optimization strategies as 'cookbook' optimizations and do not delve in to lot of details about them.

However, there exists a *lesser known* but an extremely severe performance pitfall for NVIDIA GPUs called *partition camping* which can affect the application performance by as much as *seven-fold*. Unlike other pitfalls which can be detected using tools like CudaProf and CUDA Occupancy Calculator, there is no existing tool that can detect the extent to which an application suffers from partition camping. In order to facilitate the detection of the partition camping effect, we have developed a performance model which analyzes and characterizes the effect of partition camping. CampProf, a visual analysis tool has also been developed. It visually depicts the partition camping effect in any GPU application.

5.1.1 Cookbook Optimizations

In this section, we briefly describe the *fairly well-known* CUDA optimizations as well as elucidate the reasons why these optimizations help in attaining optimum performance.

1) Run numerous threads. An NVIDIA GPU has up to 512 cores, each of which requires a thread to be able to do work. If there are less threads than cores, then potential computation is wasted, thereby reducing the *occupancy*. Beyond having enough threads to fill the cores, global memory accesses are slow to the tune of hundreds of cycles blocking. To successfully amortize the cost of global memory accesses, there has to

be enough threads in flight to take over when one or more threads are stalled on memory accesses. Since CUDA threads are lightweight, launching thousands of threads does not incur materially more cost than hundreds. However, to be able to achieve high occupancy, the amount of registers used per thread has to be kept minimum. Therefore, there is a trade-off between the number of threads that can be launched, and the number of registers used per thread on NVIDIA GPUs.

2) Use on-chip memory. NVIDIA GPUs provide 2 types of low latency on-chip memory, in addition to registers, namely (i) shared memory and (ii) constant memory. Shared memory is shared among the threads of a thread-block and thus, enables data reuse between threads within a thread-block. In addition, shared memory can also be used as a small software managed cache thanks to its low latency and low contention cost. Constant memory on the other hand, is read-only to kernels and is beneficial for storing frequently used constants and unchanged parameters which are shared among all GPU threads. However, both of these memory types are of limited capacity, thereby necessitating judicious use of space. In both cases, they help reduce global memory waiting times by reducing the global memory accesses without increasing register usage.

3) Organize data in memory. Likely the most well-known optimization on NVIDIA GPUs is ensuring that reads from global memory are *coalesced*. This means that threads in the same warp should access contiguous memory elements concurrently. In addition to coalescing, one should also ensure that threads access data from different banks of the shared memory to avoid bank conflicts, or else these accesses would be serialized.

4) Minimize divergent threads. Threads within a warp should follow identical execution paths. If the threads diverge due to conditionals and follow different paths, then the execution of said paths becomes serialized. For example, if there are four possible branches and all are taken by some thread(s) in the same warp then the block will take 4 times as long to execute as compared to when they took the same branch, with the assumption that all branches execute same number of instructions. In extreme cases this could become as high as a *32-fold* slowdown.

5) Reduce dynamic instruction count. Execution time of a kernel is directly proportional to the number of dynamic instructions executed by it. The onus of reducing the number of instructions lies upon the programmer. Reduction in the number of instructions can be done using traditional compiler optimizations like common subexpression elimination, loop unrolling, and explicit pre-fetching of data from the memory. However, these optimizations result in increased register usage which in turn limits the number of threads that can be launched, thus reducing the occupancy of the kernel.

5.1.2 Partition Camping

In this section, we describe what is partition camping as well as illustrate the performance prediction model that we developed, using an indigenous suite of micro-benchmarks, to analyze and characterize the effect of partition camping.

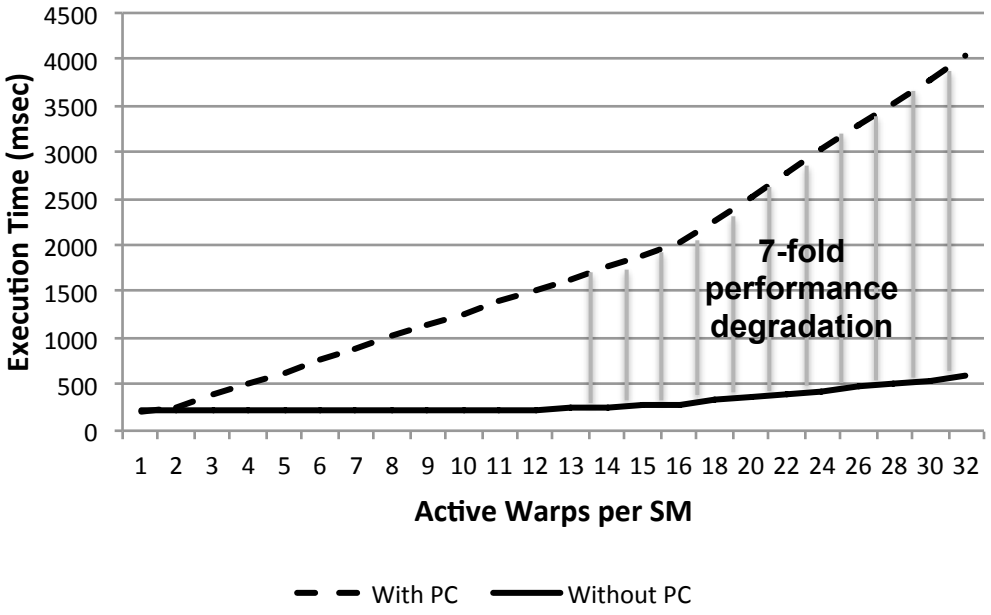


Figure 5.1: The adverse effect of partition camping in GPU kernels. PC: Partition Camping

The Problem

Partition camping arises when memory requests across blocks get serialized by fewer memory controllers on the graphics card (figure 5.2). Just as shared memory is divided into multiple banks, global memory is divided into either 6 partitions (on 8- and 9-series GPUs) or 8 partitions (on 200- and 10-series GPUs) of 256-byte width. The partition camping problem is similar to shared memory bank conflicts but experienced at a macro-level where concurrent global memory accesses by all the active warps in the kernel occur at a subset of partitions, causing requests to queue up at some partitions while other partitions go unused [35].

We developed a suite of micro-benchmarks to study the effect of various memory access patterns combined with the different memory transaction types and sizes. Our benchmarks

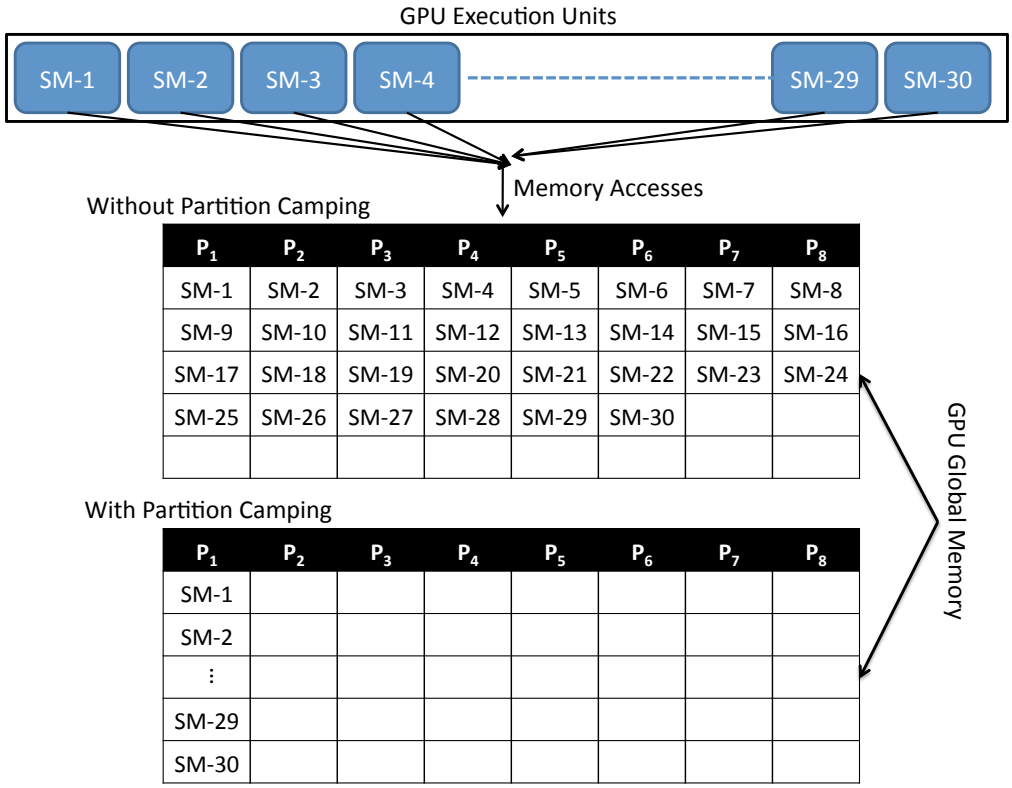


Figure 5.2: Partition camping effect in the 200- and 10-series NVIDIA GPUs. Column P_i denotes the i^{th} partition. All memory requests under the same column (partition) are serialized.

show that partition camping can degrade the performance of some kernels by up to *seven-fold* and hence, it is important to detect and analyze the effects of this problem. Details about our benchmarks can be found in appendix A

Discovery of the partition camping problem in GPU kernels is a difficult problem. There is existing literature on static code analysis for detecting bank conflicts in shared memory [11], but the same logic *cannot* be extended to detecting the partition camping problem. Bank conflicts in shared memory occur among threads in a warp, where all the threads share the same clock and hence, an analysis of the accessed address alone is sufficient to detect

Tool	GPU Kernel Characteristics					
	Occupancy	Coalesced Accesses (gmem)	Bank Conflicts (smem)	Arithmetic Intensity	Divergent Branches	Partition Camping
CUDA Visual Profiler	✓	✓	✓	✓	✓	✗
CUDA Occupancy Calculator	✓	✗	✗	✗	✗	✗
CampProf	✗	✗	✗	✗	✗	✓

Figure 5.3: Comparison of CampProf with existing profiling tools. `gmem`: global memory; `smem`: shared memory.

conflicts. However, the partition camping problem occurs when multiple active warps queue up behind the same partition *and at the same time*. This means that a static analysis of just the partition number of each memory transaction is not sufficient and hence, its timing information should also be analyzed. Each SM has its own private clock, which makes the discovery of this problem much more difficult and error prone. To the best of our knowledge, there are no existing tools that can diagnose the partition camping problem in GPU kernels. To this extent, we have developed a new tool, *CampProf*, which aims to detect the partition camping problem in GPU kernels. Figure 5.3 depicts how does CampProf compare to other existing tools like CudaProf and CUDA Occupancy Calculator. Note that the impact of partition camping is severe particularly in memory bound kernels. If the kernel is not memory bound, the effect of memory transactions will not even be significant when compared to the total execution time of the kernel and we need not worry about the partition camping problem for those cases.

Performance Prediction Model

We perform rigorous statistical analysis techniques to model the impact of partition camping in any memory-bound GPU kernel. We model the effect of memory reads separately from the memory writes, and also model the case with partition camping separately from the case without partition camping. So, we will be designing *four* model equations, one for each of the following cases: (1) Reads, Without partition camping, (2) Writes, Without partition camping, (3) Reads, With partition camping, and (4) Writes, With partition camping. We follow this approach because we believe that modeling at this fine level of detail gives us better accuracy. Specifically, we perform multiple linear regression analysis to fully understand the relationship between the execution time of the different types of our micro-benchmarks and their parameters. The independent variables (predictors) that we chose are: (1) the active warps per SM (w), and (2) the word-lengths that are read or written per thread. The dependent variable (response) is the execution time (t). The word-length predictor takes only three values (2-, 4- or 8-bytes) [1] corresponding to the three memory transaction sizes, and so we treat it as a group variable (b). This means, we first split the data-type variable into two binary variables (b_2 and b_4), where their coefficients can be either 0 or 1. If the co-efficient of b_2 is set, it indicates that the word-length is 2-bytes. Likewise, setting the co-efficient of b_4 indicates a 4-byte word-length, and if coefficients of both b_2 and b_4 are not set, it indicates the 8-byte word-length. We have now identified the performance model parameters and the performance model can be represented as shown in equation 5.1, where

[1] 1- and 2-byte word lengths will both result in 32-byte global memory transactions.

α_i denotes the contribution of the different predictor variables to our model, and β is the constant intercept.

$$t = \alpha_1 w + \alpha_2 b_2 + \alpha_3 b_4 + \beta \tag{5.1}$$

Next, we use SAS, a popular statistical analysis tool, to perform multiple linear regression analysis on our model and the data from our benchmarks. The output of SAS will provide the co-efficient values of the performance model.

Significance Test: The output of SAS also shows us the results of some statistical tests which describe the significance of our model parameters, and how well our chosen model parameters are contributing to the overall model. In particular, $R^{2[2]}$ ranges from 0.953 to 0.976 and $RMSE$ (Root Mean Square Error) ranges from 0.83 to 5.29. Moreover, we also used parameter selection techniques in SAS to remove any non-significant variable, and choose the best model. This step did not deem any of our variables as insignificant. These results mean that the response variable (execution time) is strongly dependent on the predictor variables (active warps per SM, data-types), and each of the predictors are significantly contributing to the response which proves the strength of our performance model. Informally speaking, this means that if we know the number of active warps per SM, and the size of the accessed word (corresponding to the memory transaction size), we can accurately and independently predict the execution times for reads and writes, with and without partition camping, by using the corresponding version of equation 5.1. We then

[2] R^2 is a descriptive statistic for measuring the strength of the dependency of the response variable on the predictors.

aggregate the predicted execution times for reads and writes without partition camping to generate the lower bound (predicted best case time) for the GPU kernel. Similarly, the predicted execution times for reads and writes with partition camping are added to generate the upper bound (predicted worst case time) for the GPU kernel. We validate the accuracy of our prediction model by analyzing memory access patterns in GEM in section 5.2.

The CampProf Tool

User-Interface Design and Features

CampProf is an extremely easy-to-use spreadsheet based tool similar to the CUDA Occupancy Calculator [33] and its screenshot is shown in figure 5.4. The spreadsheet consists of some input fields on the left and an output chart on the right, which can be analyzed to understand the partition camping effects in the GPU kernel. The inputs to CampProf are the following values: `gld 32b/64b/128b, gst 32b/64b/128b`, grid and block sizes, and active warps per SM. These values can easily be obtained from the CudaProf and the CUDA Occupancy Calculator tools. Note that the inputs from just a single kernel execution configuration are enough for CampProf to predict the kernel's performance range for any other execution configuration. CampProf passes the input values to our performance model which predicts and generates the upper and lower performance bounds for all the kernel execution configurations. CampProf plots these two sets of predicted execution times as two lines in the output chart of the tool. The best case and the worst case execution times form a band between which the actual execution time lies. In effect, the user provides the inputs for a

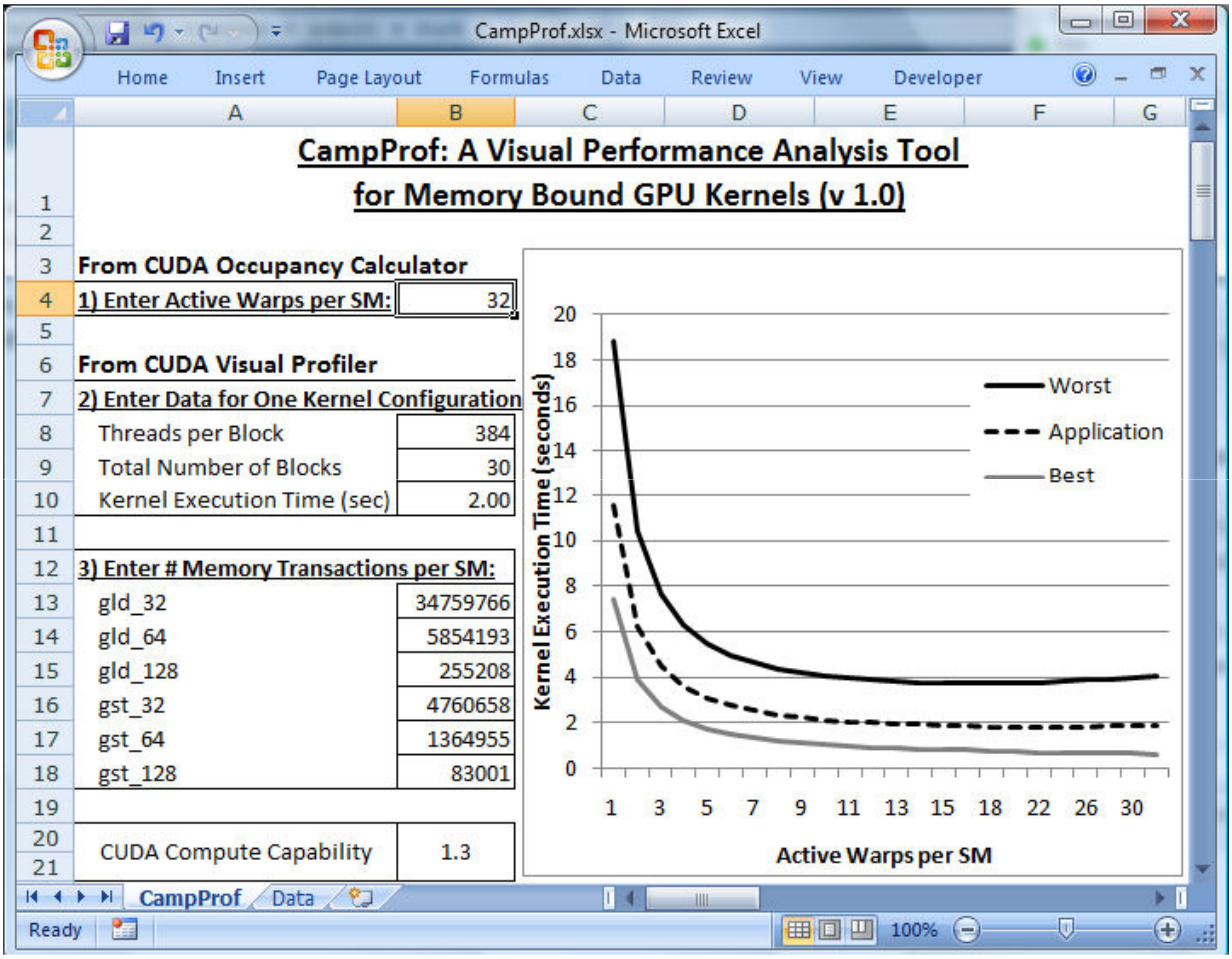


Figure 5.4: Screenshot of the CampProf tool

single kernel configuration, and CampProf displays the execution band for *all* the execution configurations.

In addition, if the actual kernel time for the given execution configuration is provided as input (`GPU Time` counter value from CudaProf), CampProf predicts and plots the kernel execution time at all the other execution configurations and is denoted by the 'Application' line in the output chart. We predict the application line by simply extrapolating the kernel

time from the given execution configuration, in a constant proportion with the execution band. Our performance model is therefore indirectly used to generate this line.

Visualizing the Effects of Partition Camping

To detect the partition camping problem in a GPU kernel, the user can simply use CampProf and inspect the position of the 'Application' line with respect to the upper and lower bounds (execution band) in the output chart. If the application line is almost touching the upper bound, it implies the worst case scenario, where all the memory transactions of the kernel (reads and writes of all sizes) suffer from partition camping. Similarly, the kernel is considered to be optimized with respect to partition camping if the application line is very close to the lower bound, implying the best case scenario. If the application line lies somewhere in the middle of the two lines, it means that performance can be potentially improved and there is a subset of memory transactions (reads or writes) that is queuing up behind the same partition. The relative position of the application line with respect to the execution band will show the *degree* to which the partition camping problem exists in the kernel. For example, while processing two matrices, the kernel might read one matrix in the row major format (without partition camping) and the other matrix might be read or written into in the column major format (with partition camping). This means that only a part of the kernel suffers from camping and the actual execution time will lie somewhere between the two extremities of CampProf's execution band. The only remedy to the partition camping problem is careful analysis of the CUDA code and re-mapping the thread blocks to the data, as explained in 'TransposeNew' example of the CUDA SDK [35].

Our approach of predicting a performance range is in contrast to the other existing performance models, which predict just a single kernel execution time. But, our method is more accurate because our model captures the large performance variation due to partition camping.

5.2 Results

In this section, we discuss performance improvements that we achieved using the optimization strategies mentioned in the previous section. We performed our experiments on two NVIDIA GPUs, belonging to two different generations; NVIDIA GTX280 and NVIDIA Tesla C2050 (Fermi). The 'Host Machine' consists of an AMD Opteron 6134 Magny Cours CPU running at 2.0 GHz with 8 GB DDR2 SDRAM. The operating system on the host is a 64 bit version of Ubuntu running the 2.6.28-19 generic Linux kernel. Programming and access to the GPU was provided using CUDA 3.2 toolkit as well as OpenCL version 1.0. For the sake of accuracy of results, all the processes which required graphical user interface were disabled to limit resource sharing of the GPU. NVIDIA GPUs have the benefit of being able to execute programs written in both CUDA and OpenCL. However, do these programming models reap equal performance? Our results portray that same optimizations when implemented in CUDA and OpenCL on the same GPU, amount to different performance improvements. Speedup across the two GPUs also varies due to the inherent architectural differences.

We also validate the performance model for partition camping by using the knowledge of

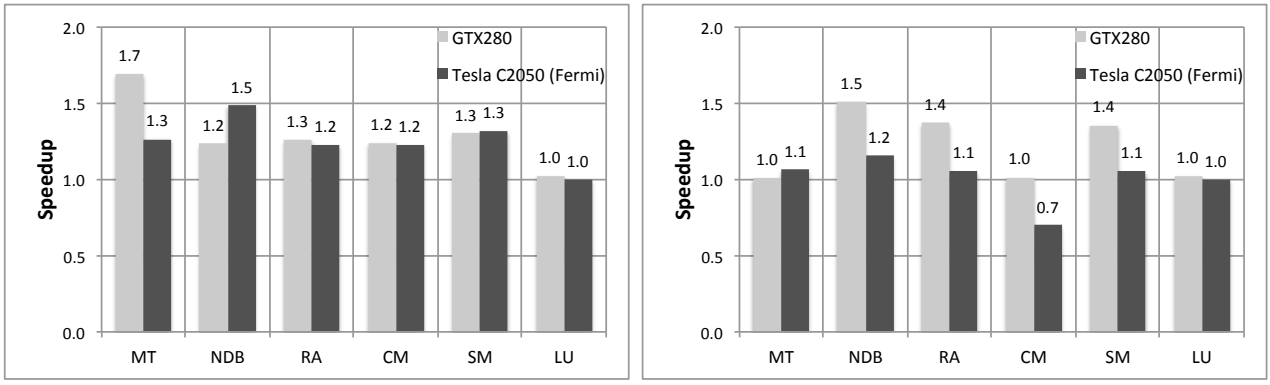
memory access pattern of GEM and the CampProf output. To accomplish this, we have used NVIDIA GTX280.

5.2.1 Cookbook Optimizations

In this section, we discuss performance benefits obtained due to implementation of *well-known* NVIDIA GPU optimization strategies. In [14], we have presented how these optimizations affect the performance of GEM when implemented in CUDA. The idea that we would like to put forth here is not how much beneficial each optimization strategy is, but how these optimizations perform when implemented in both CUDA and OpenCL and executed on the same GPU. OpenCL programs are internally converted to CUDA binaries and executed on NVIDIA GPUs and hence, are expected to achieve similar performance benefits. However, as shown in figure 5.5, this is not the case.

Figure 5.5a depicts the speedup obtained due to CUDA optimizations on NVIDIA GTX280 and NVIDIA Tesla C2050 (Fermi), over the basic CUDA implementations. We observe that speedups due to various optimization strategies vary across the two GPUs. This is quite as expected as, the underlying architecture of both GPUs is different as they belong to two different generations/families. Similar is the trend when OpenCL is used on the two GPU platforms, as shown in figure 5.5b. The variation among the two GPUs is more prominent in case of OpenCL.

However, when the performance of CUDA and OpenCL implementations are compared, we



(a) CUDA (b) OpenCL

Figure 5.5: Speedup due to cookbook optimizations. Baseline: Basic {CUDA,OpenCL} GPU implementation. MT: Max. Threads, NDB: Non Divergent Branches, RA: Register Accumulator, CM: Constant Memory, SM: Shared Memory, LU: Loop Unrolling

find that only in case of Loop Unrolling is the performance same, i.e., Loop Unrolling does not yield any benefit. This is due to the memory bound nature of GEM, as shown in section 2.3. Hence, what is required is the reduction in memory accesses but Loop Unrolling results in more efficient computation, thereby making no impact on the execution time. Other optimizations do result in performance improvement, though not equally for both programming models. Max. Threads on GTX280, results in *1.7-fold* improvement when implemented using CUDA but none when implemented using OpenCL. Similarly, removal of divergent branches results in *1.2-fold* improvement on the GTX280 using CUDA but when implemented using OpenCL, the benefit is *1.5-fold*, whereas on Tesla C2050 (Fermi), the performance improvements are flipped. Performance benefits are unequal for other optimization strategies also like using on-chip memory. When Constant Memory is implemented in OpenCL on Tesla C2050 (Fermi), a *slowdown* is observed but there is a performance improvement when CUDA is used.

It is not exactly clear why this disparity exists in performance benefits across programming

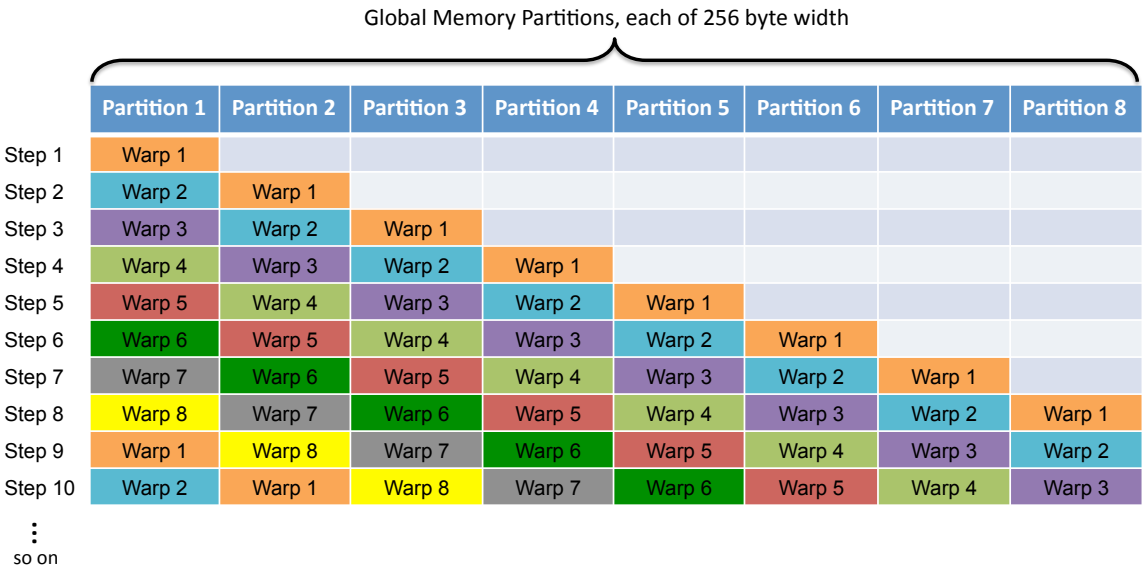


Figure 5.6: GEM: Memory access pattern

models, more so when a CUDA binary is executed in both cases. It is not even the case that either of CUDA or OpenCL optimizations always perform better than the other. These results pave way for future work which would deal with investigating this inconsistency.

5.2.2 Partition Camping

In section 5.1.2, we showed how our performance model is used to predict the upper and lower bounds of the GPU kernel performance. In this section, we validate the accuracy of our performance model by comparing the predicted bounds (best and worst case) with the execution time of GEM.

As mentioned in section 2.3, all GPU threads in GEM, access coordinates of molecule constituents stored in global memory in the following order: from the coordinate of first compo-

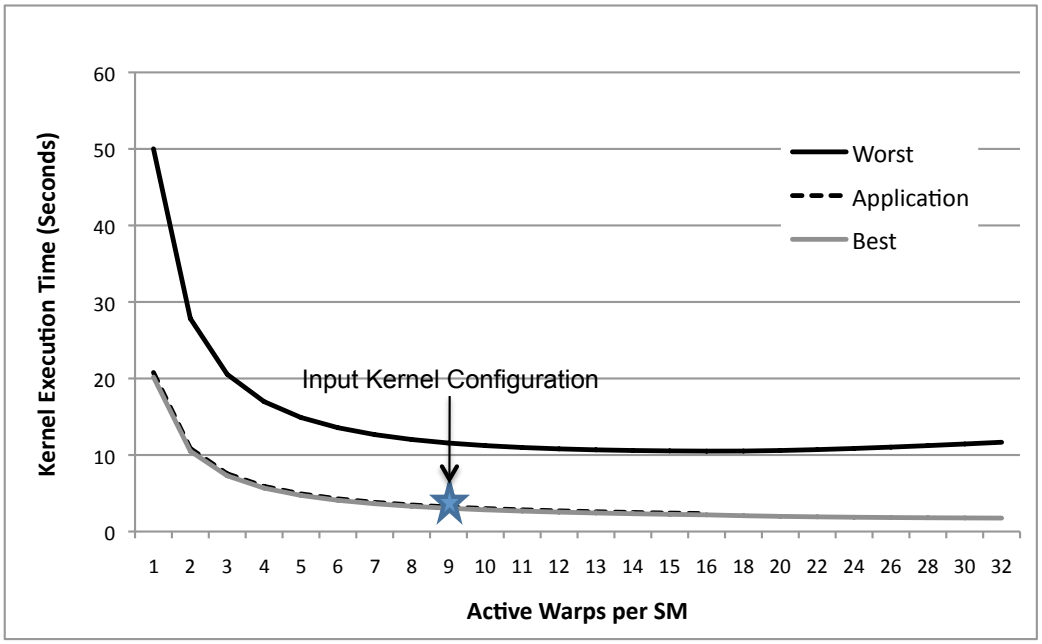


Figure 5.7: GEM: CampProf output

nent to the last, thereby implying that all active warps would be queued up behind the same memory partition at the beginning of the algorithm. However, only one warp can access that global memory partition, stalling all other warps. Once the first warp finishes accessing the elements in the first partition, it would move on the next partition, thus, the first partition can now be accessed by the next warp in the queue. Partition access will be pipelined, as shown in figure 5.6. Once this memory partition pipeline is filled up (i.e. after eight such iterations on a device with compute capability 1.2 or 1.3), memory accesses will be uniformly distributed across all available memory partitions. The pipelined nature of memory accesses can be assumed to not result in further stalls because the workload for all the warps is same. This illustrates that in theory, GEM does not suffer from partition camping.

To corroborate this fact, we use CampProf to analyze the partition camping behavior in

GEM. Figure 5.7 shows the CampProf output chart depicting the partition camping effect in GEM. The input to the tool is the number of memory transactions and the kernel execution time for one execution configuration (denoted by the $\star$). It shows the worst and best case times for all the execution configurations, as predicted by our model. The 'Application' line in the graphs is extrapolated from the actual execution time that was provided as input. The predicted 'Application' line is presented for only up to 16 active warps, beyond which it was not possible to execute due to shared memory and register usage constraints imposed by the GPU architecture. From the figure, it can be noted that the predicted application performance is extremely close to the 'Best' line of CampProf, which agrees with our discussions about GEM not suffering from partition camping. This is in conjunction to our expectation and thereby proves the efficacy of our performance prediction model as well as demonstrates the utility of the CampProf tool.

Chapter 6

Summary and Future Work

In this chapter, we present a summary of the optimizations discussed and also present some future work that can be built upon this thesis.

6.1 Summary

In this section, we summarize our findings of architecture-aware optimization strategies for heterogeneous computing systems.

Heterogeneous computing has proved to be indispensable in high performance computing. The primary reason for the unprecedented success of heterogeneous computing has been the widespread adoption of compute capable graphics processing units (GPUs). GPUs would continue to enjoy popularity in future, as it is difficult to envision the efficient use of hundreds

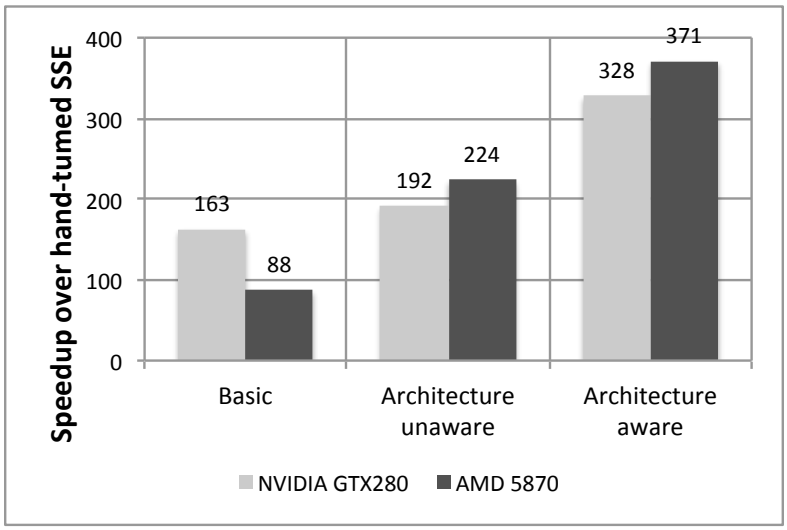


Figure 6.1: Speedup when optimized for each architecture

of traditional CPU cores. However, the use of hundreds of accelerator cores in conjunction with a handful of traditional CPU cores appears to be a sustainable roadmap.

GPUs come in various flavors and with different underlying architectures, thus, do not adhere to the *one-size-fits-all* philosophy. Hence to reap supreme performance on GPUs, one has to perform architecture-aware optimizations and mappings. To corroborate this fact, we present figure 6.1, which depicts the speedup obtained when three different implementations of the same application are run on GPUs from NVIDIA and AMD. The importance of performing architecture-aware optimizations is manifested by the fact that even with architecture-unaware optimizations, there is a performance improvement over the basic implementation. However, optimum performance is obtained only when architecture aware optimizations are performed. After performing architecture-aware optimizations, the AMD GPU is realized to its full potential and hence, performs *12%* better than the NVIDIA GPU.

Table 6.1 presents a summary of optimization strategies discussed so far along with their degree of effectiveness on AMD and NVIDIA GPUs.

6.2 Future Work

In this section, we discuss some future work that can be built upon this thesis.

Auto-optimization framework for OpenCL applications: The major contribution of this thesis is to acquaint the research community with the benefits of architecture-aware optimization strategies. With the evolution of OpenCL, application developers are enticed to run the same application on various architectures, but only to find out that dissimilar performance is obtained on each architecture. Therefore, a framework could be developed which when made aware of the underlying architecture, automatically optimizes the application for that architecture. Hence, achieving optimum performance on various heterogeneous computing systems would be guaranteed.

Auto-tuning framework to find the best combination of optimizations: This thesis demonstrates that there is no concrete way to figure out the best combination of various optimization strategies for an application, on a particular architecture. Brute force is the *only* method in which one can find the best combination. Therefore, a framework could be developed, which auto-tunes an application along various dimensions and identifies a set of optimization strategies that would help achieve optimum performance on any given heterogeneous computing system.

Table 6.1: Impact of optimization strategies on AMD and NVIDIA GPUs. Greater the number of +s, greater is the positive impact. The ones in red are architecture-aware optimizations

Optimization Strategy	**AMD**	**NVIDIA**
Algorithm Design:		
Recomputing instead of Transferring	+ + + + +	+ + + + +
Using Host Asynchronously	+	+
Execution Configuration:		
Running Numerous Threads	+ + + + +	+ + + + +
Ensuring #Threads to be a multiple of Wavefront/Warp Size	+ + + + +	+ + + + +
Ensuring #Workgroups/Blocks to be a multiple of #Cores	+ + + + +	+ + + + +
Reducing Per-Thread Register Usage	+ + +	+ + + + +
Control Flow:		
Removing Branches	+ + +	+
Removing Divergent Branches	+ + + + +	+ + + + +
Memory Types:		
Using Registers	+ + + + +	+ + +
Using Local/Shared Memory	+ + + + +	+ + + + +
Using Constant Memory	+ + +	+ + +
Using Image/Texture Memory	+ + + + +	+
Memory Access Pattern:		
Coalescing Global Memory Accesses	+ + + + +	+ + + + +
Avoiding Partition Camping	+ + +	+ + + + +
Avoiding Bank Conflicts	+ + + + +	+ + + + +
Instruction Count:		
Using Vector Types and Operations	+ + + + +	+
Prefetching of Data from Global Memory	+ + + + +	+ + + + +
Loop Unrolling	+ + + + +	+ + + + +

Performance modeling tool for compute bound applications: In our efforts to characterize and analyze the severe effect of partition camping in GPU kernels, we developed a performance prediction model for memory bound applications. Therefore, a similar model could be developed for compute bound applications. Also, the model which we have developed is amenable only to previous NVIDA GPU architecture (GT200) and hence, one can extend the model to the present generation architecture of NVIDIA GPUs, i.e., Fermi.

Efficacy of optimization strategies on novel architectures: The efficacy of the discussed optimization strategies on novel heterogeneous architectures like AMD Fusion and Intel Knights Ferry would be interesting. These architectures are different from current architectures in the sense that they combine the general purpose x86 CPU cores and the programmable accelerator cores on the same silicon. Even if these optimization strategies do not perform as expected, one could at least use a similar methodology of analyzing the underlying architecture and devise new optimization strategies.

Bibliography

[1] The Top500 Supercomputer Sites. http://www.top500.org.

[2] The OpenMP API Specification for Parallel Programming, 2010. http://openmp.org/wp/openmp-specifications/.

[3] Aaftab Munshi. The OpenCL Specification, 2008. http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf.

[4] Ashwin Aji, Mayank Daga, and Wuchun Feng. Bounding the Effect of Partition Camping in GPU Kernels. In *ACM International Conference on Computing Frontiers (To appear)*, 2011.

[5] AMD. AMD Stream Computing OpenCL Programming Guide. http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf.

[6] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, N.J., Apr. 18), AFIPS Press, Reston, Va., 1967, pp. 483;485, when Dr.

Amdahl was at International Business Machines Corporation, Sunnyvale, California. *Solid-State Circuits Newsletter, IEEE*, 12(3):19 –20, 2007.

[7] Ramu Anandakrishnan, Tom R.W. Scogland, Andrew T. Fenley, John C. Gordon, Wu chun Feng, and Alexey V. Onufriev. Accelerating Electrostatic Surface Potential Calculation with Multiscale Approximation on Graphics Processing Units. *Journal of Molecular Graphics and Modelling*, 28(8):904 – 910, 2010.

[8] Jeremy Archuleta, Yong Cao, Tom Scogland, and Wu-chun Feng. Multi-Dimensional Characterization of Temporal Data Mining on Graphics Processors, May 2009.

[9] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[10] ATI. ATI Stream Computing User Guide. *ATI, March*, 2009.

[11] Michael Boyer, Kevin Skadron, and Westley Weimer. Automated Dynamic Analysis of CUDA Programs. In *Proceedings of 3rd Workshop on Software Tools for MultiCore Systems*, 2010.

[12] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM TRANSACTIONS ON GRAPHICS*, 23:777–786, 2004.

[13] Daniel. Cederman and Philippas Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 57–64. Eurographics Association Aire-la-Ville, Switzerland, Switzerland, 2008.

[14] Mayank Daga, Wuchun Feng, and Thomas Scogland. Towards Accelerating Molecular Modeling via Multiscale Approximation on a GPU. In *1st IEEE International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, pages 75 –80, 2011.

[15] Dana Schaa and David Kaeli. Exploring the Multi-GPU Design Space. In *IPDPS '09: Proc. of the IEEE International Symposium on Parallel and Distributed Computing*, 2009.

[16] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil Computation Optimization and Auto-Tuning on Sate-of-the-Art Multicore Architectures. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12. IEEE, 2009.

[17] John C. Gordon, Andrew T. Fenley, and A. Onufriev. An Analytical Approach to Computing Biomolecular Electrostatic Potential, II: Validation and Applications. *Journal of Chemical Physics*, 2008.

[18] Khronos Group. OpenCL. `http://www.khronos.org/opencl/`.

[19] Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, and Bixia Zheng. Twin peaks: a Software Platform for Heterogeneous Computing on General-Purpose and Graphics Processors. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 205–216, New York, NY, USA, 2010. ACM.

[20] Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo, and Satoshi Matsuoka. Statistical Power Modeling of GPU Kernels Using Performance Counters. In *IGCC '10: Proceedings of International Green Computing Conference*, 2010.

[21] Sunpyo Hong and Hyesoon Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *ISCA '10: Proceedings of the 37th International Symposium of Computer Architecture*, 2009.

[22] Sunpyo Hong and Hyesoon Kim. An Integrated GPU Power and Performance Model. In *ISCA '10: Proceedings of the 37th International Symposium of Computer Architecture*, 2010.

[23] Jen-Hsun Huang. Opening Keynote, NVIDIA GPU Technology Conference, 2010. `http://livesmooth.istreamplanet.com/nvidia100921/`.

[24] Intel. Intel SSE Documentation. `http://www.intel80386.com/simd/mmx2-doc.html`.

[25] Intel, 1994.

[26] Byunghyun Jang, Synho Do, Homer Pien, and David Kaeli. Architecture-Aware Optimization Targeting Multithreaded Stream Computing. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 62–70, New York, NY, USA, 2009. ACM.

[27] Chris Jang. OpenCL Optimization Case Study: GATLAS - Designing Kernels with Auto-Tuning. `http://golem5.org/gatlas/CaseStudyGATLAS.htm`.

[28] Ashfaq A. Khokhar, Viktor K. Prasanna, Muhammad E. Shaaban, and Cho-Li Wang. Heterogeneous Computing: Challenges and Opportunities. *Computer*, 26:18–27, June 1993.

[29] Michael Bader, Hans-Joachim Bungartz, Dheevatsa Mudigere, Srihari Narasimhan and Babu Narayanan. Fast GPGPU Data Rearrangement Kernels using CUDA. In *HIPC '09: Proceedings of High Performance Computing Conference*, 2009.

[30] George E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998.

[31] Jack Nickolls and Ian Buck. NVIDIA CUDA Software and GPU Parallel Computing Architecture. In *Microprocessor Forum, May*, 2007.

[32] NVIDIA. CUDA. `http://www.nvidia.com/object/cuda_home_new.html`.

[33] NVIDIA. CUDA Occupancy Calculator, 2008. `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`.

[34] NVIDIA. The CUDA Compiler Driver NVCC, 2008. `http://www.nvidia.com/object/io_1213955090354.html`.

[35] NVIDIA. Optimizing Matrix Transpose in CUDA, 2009. `NVIDIA_CUDA_SDK/C/src/transposeNew/doc/MatrixTranspose.pdf`.

[36] NVIDIA. NVIDIA CUDA Programming Guide-3.2, 2010. `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf`.

[37] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[38] David Patterson. The Trouble With Multicore, 2010. `http://spectrum.ieee.org/computing/software/the-trouble-with-multicore`.

[39] Freund Richard and Conwell D. Superconcurrency: A Form of Distributed Heterogeneous Supercomputing. *Supercomputing Review*, pages 47–50, 1991.

[40] Shane Ryoo, Christopher Rodrigues, Sam Stone, Sara Baghsorkhi, Sain-Zee Ueng, and Wen mei Hwu. Program Optimization Study on a 128-Core GPU. In *Workshop on General Purpose Processing on Graphics Processing*, 2008.

[41] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and W. W. Hwu. Optimization Principles and Application Performance Evaluation of

a Multithreaded GPU Using CUDA. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 73–82, February 2008.

[42] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu. Program optimization space pruning for a multithreaded gpu. In *CGO '08: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, New York, NY, USA, 2008. ACM.

[43] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, John A. Stratton, Sain-Zee Ueng, Sara S. Baghsorkhi, and Wen mei W. Hwu. Program optimization carving for gpu computing. *Journal of Parallel and Distributed Computing*, 68(10):1389 – 1401, 2008. General-Purpose Processing using Graphics Processing Units.

[44] Sara S. Baghsorkhi and Matthieu Delahaye and Sanjay J. Patel and William D. Gropp and Wen-mei W. Hwu. An adaptive performance modeling tool for GPU architectures. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2008.

[45] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, 2009. `http://www.gotw.ca/publications/concurrency-ddj.htm`.

[46] Vasily Volkov and James Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proc. of the 2008 ACM/IEEE Conference on Supercomputing*, November 2008.

[47] Samuel Webb Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[48] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. In *IS-PASS '10: Proceedings of the 37th IEEE International Symposium on Performance Analysis of Systems and Software*, 2010.

[49] Fang Xudong, Tang Yuhua, Wang Guibin, Tang Tao, and Zhang Ying. Optimizing Stencil Application on Multi-thread GPU Architecture Using Stream Programming Model. In *Architecture of Computing Systems - ARCS 2010*, volume 5974 of *Lecture Notes in Computer Science*, pages 234–245. Springer Berlin / Heidelberg, 2010.

Appendix A

Micro-Benchmarks for Detection of Partition Camping

While partition camping truly means that any subset of memory partitions are being accessed concurrently, we choose the extreme cases for our study, i.e. all the available partitions are accessed uniformly (Without Partition Camping), or only one memory partition is accessed all the time (With Partition Camping). Although this method does not exhaustively test the difference degrees of partition camping, our study acts as a realistic first-order approximation to characterize its effect in GPU kernels. Thus, we developed two sets of benchmarks and analyzed the memory effects with and with- out partition camping. Each set of benchmarks tested the different memory transaction types (reads and writes) and different memory transaction sizes (32-, 64- and 128-bytes), which made it a total of 12 benchmarks for analysis.

```
// TYPE can be a 2-, 4- or an 8-byte word
__global__ void readBenchmark(TYPE *d_arr) {
    // assign unique partitions to blocks,
    int numOfPartitions = 8;
    int curPartition = blockIdx.x % numOfPartitions;
    int partitionSize = 256; // 256 bytes
    int elemsInPartition = partitionSize/sizeof(TYPE);
    // jump to unique partition
    int startIndex = elemsInPartition
                           * curPartition;
    TYPE readVal = 0;

    // Loop counter 'x' ensures coalescing.
    for(int x = 0; x < ITERATIONS; x += 16) {
        /* offset guarantees to restrict the
           index to the same partition */
        int offset = ((threadIdx.x + x)
                        % elemsInPartition);
        int index = startIndex + offset;
        // Read from global memory location
        readVal = d_arr[index];
    }
    /* Write once to memory to prevent the above
       code from being optimized out */
    d_arr[0] = readVal;
}
```

Figure A.1: Code snapshot of the 'read' micro-benchmark for the NVIDIA 200- and 10-series GPUs (Without Partition Camping). Note: `ITERATIONS` is a fixed and known number.

```
// TYPE can be a 2-, 4- or an 8-byte word
__global__ void readBenchmark(TYPE *d_arr) {
    int partitionSize = 256; // 256 bytes
    int elemsInPartition = partitionSize/sizeof(TYPE);
    TYPE readVal = 0;

    // Loop counter 'x' ensures coalescing.
    for(int x = 0; x < ITERATIONS; x += 16) {
    /* all blocks read from a single partition
        to simulate Partition Camping */
        int index = ((threadIdx.x + x)
                        % elemsInPartition);
        // Read from global memory location
        readVal = d_arr[index];
    }
    /* Write once to memory to prevent the above
       code from being optimized out */
    d_arr[0] = readVal;
}
```

Figure A.2: Code snapshot of the 'read' micro-benchmark for the NVIDIA 200- and 10-series GPUs (With Partition Camping). Note: `ITERATIONS` is a fixed and known number.

Figures A.1 and A.2 show the kernel of the micro-benchmarks for memory reads, without and with partition camping respectively. The benchmarks that simulate the partition camping effect (figure A.2) carefully access memory from only a single partition. The micro-benchmarks for memory writes are very similar to the memory reads, except that `readVal` is written to the memory location inside the for-loop (line numbers 21 and 14 in the respective code snapshots). We modify the `TYPE` data-type in the benchmarks to one of 2-, 4- or 8-byte words in order to trigger 32-, 64- or 128-byte memory transactions respectively to the global memory. Although our benchmarks have a high ratio of compute instructions to memory instructions, we prove that they are indeed memory bound, i.e. the memory instructions dominate the overall execution time. We validate this fact by using the methods discussed in [4]. Our suite of benchmarks is therefore a good representation of real memory-bound kernels.