PERFORMANCE PORTABILITY OF PARALLEL KERNELS ON SHARED-MEMORY SYSTEMS

BY

JOHN ANDREW STRATTON

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Professor Wen-mei W. Hwu, Chair
Associate Professor Deming Chen
Associate Professor Steven Lumetta
Professor David Padua

ABSTRACT

This work describes my solution to the performance portability problem: between CPUs and GPUs in particular, but laying the foundation for even broader performance portability support. I argue that the best approach is to use a language like OpenCL as a portable, low-level programming model with well-defined mechanisms for expressing multi-level parallelism and locality. That low-level program representation can be supported with architecture-specific compilers, runtimes, and libraries to target the application code to various platforms with high performance. High-level language designers or tool developers could then target this single, low-level programming and parallelism model as a portable, high-performance intermediate program representation.

To demonstrate the feasibility of this approach, I show how one would design a good CPU implementation of OpenCL given that the programs are written according to the current high-level GPU vendor optimization guidelines. Programs written in such a way already meet the criteria of good GPU performance, and in this work, I show that those same programs on a CPU platform implemented according to my proposals can out-perform an OpenMP implementation of the same algorithm on the same system.

To my wife, Jenny,
Thanks for everything

ACKNOWLEDGMENTS

I would like to thank my adviser, Wen-mei Hwu, for having faith in my potential when it was just that, and for giving me more public credit than a graduate student often gets. Thanks to the IMPACT research group's current, former, and ancillary members, who have been excellent mentors, colleagues, and supporters. Special thanks go to Shane Ryoo and Sain Ueng for helping me get started in research, and to Hee-Seok Kim for helping with many implementation details and carrying out several experiments. And thanks to Professors Steve Lumetta, Sanjay Patel, Deming Chen, David Padua, and Nacho Navarro, for welcoming me into the academic community, especially those who served on my dissertation committee.

Thanks to MulticoreWare Inc. for believing that this technology was worth a significant financial and development investment. I will be glad to help make it pay off.

Thanks to all the people and organizations that have supported me and my research materially and financially during my graduate studies. My salary has, at various times, been paid in part by the following: the Gigascale Research Center of the Semiconductor Research Corporation, the Microsoft-/Intel Universal Parallel Computing Research Center, NVIDIA Corporation, MulticoreWare Inc., and the University of Illinois at Urbana-Champaign through its Department of Electrical and Computer Engineering. Many of the named financial supporters also provided significant equipment donations for the experiments presented here. I also want to thank the donors and judges who have encouraged me through awards and honors, including Dan Vivoli and the Dan Vivoli Endowed fellowship committee, Nell W. Reid and the E. A. Reid Fellowship committee, and Dr. Frederic T. and Edith F. Mavis and the administrators of the Mavis Future Faculty Fellows program.

Any finally, greatest thanks to my ever-growing family for their love and support.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

BRAM	Block Random Access Memory
CGRA	Coarse-Grained Reconfigurable Array
CPU	Central Processing Unit
DMA	Direct Memory Access
DSP	Digital Signal Processor
ESL	Electronic System Level
FPGA	Field-Programmable Gate Array
FSB	Front-Side Bus
FSM	Finite State Machine
GPU	Graphics Processing Unit
HLS	High-Level Synthesis
I/O	Input/Output
ILP	Instruction-Level Parallelism
MPMD	Multiple-Program Multiple-Data
SIMT	Single-Instruction Multiple-Threads
OoO	Out-of-Order
RISC	Reduced Instruction Set Computing
RTL	Register Transfer Logic
SDC	System of Difference Constraints
SIMD	Single-Instruction Multiple-Data

SIMT	Single-Instruction Multiple-Threads
SPMD	Single-Program Multiple-Data
VLIW	Very Long Instruction Word

CHAPTER 1

INTRODUCTION

In years past, a system's inability to achieve reasonable performance for generic codes that performed well on other systems were typically attributed to architecture design flaws or compiler limitations. Once such expectations are established, they are self-fulfilling, because new system designs are often constrained by legacy programs.

With the advent of widespread parallel programming, the mindset of the field has changed drastically, primarily because legacy applications were no longer regarded as the sole or even primary constraint on system design. The fragmenting of the parallel programming realm meant that programmers could not choose any language or programming model that was supported by a wide variety of relevant architectures. Among the various proposals for an industry standard language for both CPU and accelerator architectures, the most widely supported and adopted is OpenCL [23], specifically the hierarchical SPMD programming model component it shares in common with the CUDA language [6]. Yet even after the standardization and implementation process, there is still a serious disagreement among industry vendors about what kind of OpenCL code people should write for good performance.

To illustrate the issue of performance portability, let us use the example of a typical, experienced software engineer as recently as ten years ago. Computer architectures have a wide variety of instruction sets and processor designs: Alpha, x86, Itanium, PowerPC, SPARC, MIPS, and ARM, just to name several. Yet the typical software developer did not have to assume a particular processor to write good programs in high-level languages like C or C++. That is to say, whatever made a generic C or C++ program "good" was independent of the specific architecture on which it would eventually be executed.

My research has operated under this definition.

A program exhibits performance portability for a particular

set of computer systems when that piece of software, executed on each system, achieves reasonable performance while producing valid results.

The performance portability problem has gotten a lot of attention, and for good reason. In this thesis, I define performance portability as a property exhibited by some piece of software for some collection of architectures, where that piece of software achieves a good level of performance on each architecture. The use of the word "good" in the definition means that performance portability is necessarily subjective. Essentially, a software developer decides that a particular piece of software does not have performance portability for the class of processors they care about when they decide to split their code into multiple versions for multiple processors. That decision is determined by context-specific weightings given to the cost of creating and maintaining multiple versions of code versus the potential loss of performance on certain platforms.

Performance portability as a general metric is applied differently in different software and system contexts. In the world of sequential computing, performance portability usually meant that a single piece of software was able to compile and execute on a variety of microprocessors from multiple vendors. Although x86 architectures dominated the consumer-level computing market, the instruction set and architectures have not remained static. Yet many software developers never consider which specific architecture generation with specific ISA extensions will be running their code. In parallel clusters, performance portability among cluster MPI programs usually means that the software package works well on a variety of clusters and interconnects. This dissertation focuses on the emerging parallel accelerator programming model of hierarchical, fine-grained SPMD kernel programming. This programming model is embodied in many widely used languages, such as OpenCL [23], CUDA [33], C++AMP [32], and OpenACC [37]. In practice, OpenCL is the only language with multiple mature implementations from multiple vendors, in large part because it was the first language proposed with an open specification and the explicit goal of portability among multiple devices.

Despite portability being a goal of the OpenCL language, a belief in the performance portability of applications written in OpenCL is noticeably ab-

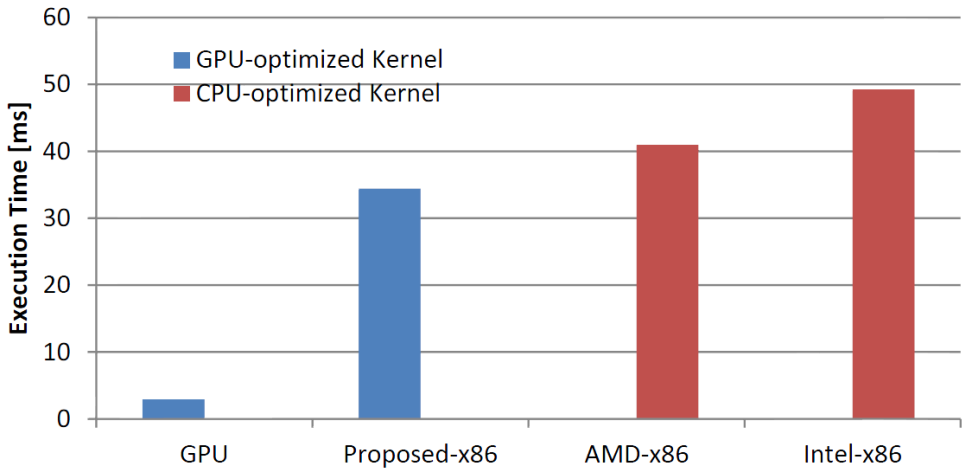


Figure 1.1: The EP NAS Parallel Benchmark in OpenCL executed on current and proposed OpenCL implementations. GPU results were run on an NVIDIA Tesla C2050. All platforms targeting x86 were executed on an Intel Core^{TM}i7-3770 CPU.

sent from today's accelerated code developers. To some extent, this is to be expected; software projects that chose an accelerator programming model typically place a higher-than-average priority on performance, and therefore a higher standard for performance portability. However, if publications are any indication, there is a widespread belief that optimizing OpenCL programs for a GPU architecture requires drastically different optimizations and coding practices than does optimizing an OpenCL application for a multi-core CPU architecture or non-GPU accelerator [46, 41]. For example, let us take a recent work studying the NAS Parallel benchmarks in OpenCL on a variety of architectures. Seo et al. saw that with the OpenCL language implementations available to them for GPUs and CPUs, they had to optimize the GPU-targeted and CPU-targeted OpenCL kernels very differently to get good performance on each [46]. When their results are reproduced for the "Embarrassingly Parallel" benchmark on updated hardware in Figure 1.1, we can confirm that the GPU-optimized kernel greatly outperforms the Intel and AMD OpenCL language implementations on x86 for the CPU-optimized kernel, largely because of the accelerator device's higher peak throughput.

In this dissertation, I argue that the perceived lack of performance portability of OpenCL applications for CPU and GPU architectures does not mean that the architectures lack features necessary for achieving high performance with a common coding style. Instead it seems that the performance portabil-

ity problem between modern GPUs and CPUs has more to do with the fact that certain implementations of the OpenCL language have differing implicit performance costs. For some applications, performance portability will never be truly achieved, if the chosen algorithms are clearly much more suited to one kind of architecture. For algorithms that can perform well on both CPUs and GPUs in theory, I demonstrate that there is nothing fundamentally deficient about programming for a CPU in OpenCL as opposed to OpenMP or other more traditional CPU parallel programming models.

As Figure 1.1 also shows, the work implemented in this dissertation achieves higher performance on the x86 processor than either existing industry OpenCL language implementation, but most importantly gets very high CPU performance *for the GPU-optimized kernel*. In this example, the language implementation methodology described here obviates the need to maintain the distinct "CPU-optimized" version of the code, achieving single-source performance portability.

1.1 Software and Hardware Industry Context

The context for this material is set by the current trends towards parallel and heterogeneous computing in large consumer device markets. Energy consumption limitations on practically every class of computing device led CPU vendors to abandon the single-core design methodology. Fully automatic exploitation of explicitly parallel platforms is largely seen as ineffective for most application code written under the assumption of sequential program execution. The software industry experienced a major disruption, because sequential legacy codes could no longer fully utilize the CPUs of their customers. The software industry has been effectively forced to invest a significant amount of development effort to migrate away from sequential programming models for performance-sensitive code regions.

Simultaneously, forces in various special-purpose architecture markets were leading architects in those markets to increase the flexibility and programmability. In particular, GPUs began as fixed-function units whose only output was four-channel pixel colors at eight unsigned bits per channel. Demand for increasing realism and artistic expression in rendered scenes led GPUs to increase in flexibility. The vertex and fragment units adopted more and

more programmability, moving from being fixed-function only, to adopting programmability with straight-line assembly, to supporting limited and finally arbitrary branching. Pioneers tracked the increasing capabilities of the GPU with increasingly varied applications, hijacking graphics programming interfaces to perform various linear algebra and scientific simulation calculations.

GPU vendors realized that with CPU software developers being forced to parallelize code anyway, those developers may be willing to parallelize their code for a GPU. GPU vendors released GPU-friendly programming languages and tools that eschewed explicit graphics terms or constructs to attract general developers. However, in 2007, when CUDA was released, it would still not have been fully feasible to make serious claims regarding performance portability between CPUs and GPUs. The GPU architectures of the time imposed stringent requirements on software to achieve a significant fraction of peak performance, which have since been significantly softened as described in Chapter 3. Today, the first-order performance guidelines offered by GPU vendors are mostly limited to fundamental performance issues shared in common with CPU architectures, namely good locality management, vector execution, and scalability.

1.2 Kernel Programming in the Software Abstraction Hierarchy

Figure 1.2 shows a few levels of abstraction and translation in the process of expressing and solving real-world problems on computing hardware, annotated with some typical performance-impacting decisions or transformations. From the programmer's standpoint, anything addressed by tools, libraries, and architecture after the application code has been written is part of the "system". The programmer does not care whether the CPU is a VLIW processor relying on the compiler to completely control dynamic instruction scheduling, or an OoO architecture that takes on much of the scheduling burden in hardware. Neither does the programmer care about the complexity of the code behind library interfaces, which could be implemented differently for every system. Those optimizations that differ most among architectures are typically assigned to the tools, libraries, and compilers for that architecture.

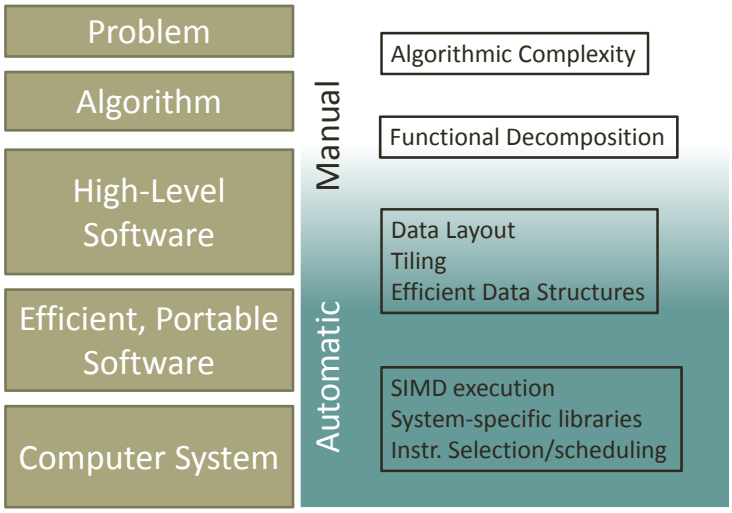


Figure 1.2: Levels of abstraction translating a high-level problem into a sequential, computational solution. On the right are tasks roughly associated with the level of abstraction at which they are typically addressed.

This is another way of describing performance portability, that the application programmer is not expected to perform those optimizations unique to one architecture.

A widespread sentiment today is that it is no longer possible to write one piece of parallel software and expect it to get reasonable performance on both CPUs and GPUs from various vendors [43, 46, 41]. This adds a heavy burden to software engineers, who must now consider additional development and perpetual maintenance costs for every additional platform they wish to target. Software developers also have no reasonable guarantees that their code will remain viable for future hardware platforms.

In this work I demonstrate that the lack of performance portability is far from inevitable. In sequential processors, certain architecture design principles were fundamental, whereas others were addressed by the compilers and libraries associated with that processor. Similarly, fundamental architecture design principles govern parallel architectures, and other design variations can typically be abstracted in a general way.

1.2.1 Generic, high-level translation and optimization

Figure 1.3 illustrates how the abstraction and optimization hierarchy should be adjusted in my proposal for performance portability among parallel archi-

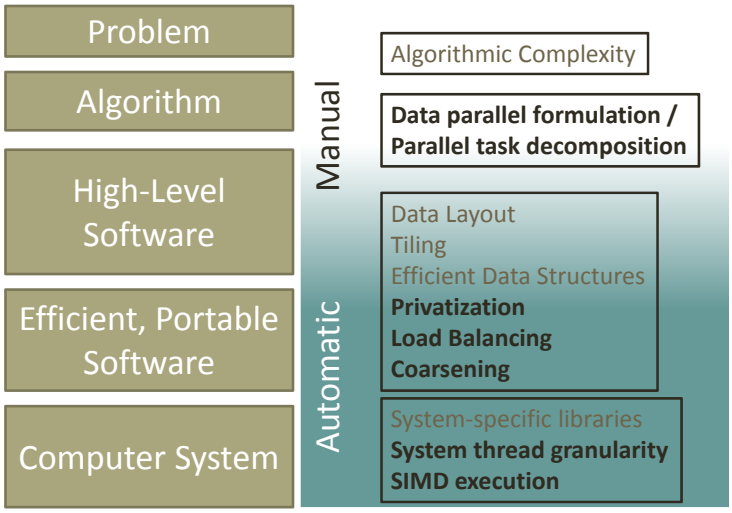


Figure 1.3: Levels of abstraction translating a high-level problem into a parallel computational solution. On the right are tasks roughly associated with the level of abstraction at which they are typically addressed. Tasks not typically necessary for a sequential execution context are shown in bold.

tectures. Specifically, it is a proposal for how we can categorize most optimizations as either generically applicable and expressible, or system-specific. Achieving performance portability means that the programmer or high-level programming tools can perform generic optimizations targeting a system-agnostic interface. It also requires that system-specific performance concerns are hidden behind portable abstractions with system-specific implementations.

The most important responsibility of the programmer is choosing a fine-grained parallel decomposition of their application. Fine-grained decomposition is most portable for parallel architectures, primarily because tools can more robustly aggregate work into coarser tasks than they can further decompose tasks into finer threads. Therefore, the initial parallel decomposition should create a very large number of parallel tasks, to fill a current GPU's tens of thousands of thread contexts many times over and still have room to scale for several more architecture generations. The parallel decomposition should also embody good locality principles, with tasks within a group assigned to nearby or overlapping input and output data.

Beyond decomposition, several other high-level optimizations are often necessary to get good performance, such as tiling and data layout [48, 52]. These optimizations are nearly universal, as they address fundamental architecture

design principles such as locality, line-oriented memory systems, and resource contention. Chapter 2 will summarize the most important high-level optimization patterns or techniques as they apply to the hierarchical SPMD programming model. Even so, it is critical for researchers now to consider how to design high-level parallel programming languages and compilers such that we can practically move responsibility for many of these optimizations away from the programmer and onto the system implementers. While the specific parameters for optimizations like tiling may need to be tuned to each specific system for best performance [44, 4], underestimating hardware resources is typically much preferable to overestimating those resources, and conservative parameter selections should often be reasonably portable.

1.2.2 Architecture-specific, low-level translation and optimization

Performance portability requires that each system implementing the language have at least non-conflicting requirements of what "good" source code should look like. The most pressing issues are related to implementing uniform abstractions for vectorization and multithreading, when the degrees of each vary widely from one architecture to another. A recent GPU will have several orders of magnitude more thread contexts than a recent CPU. Architectures like GPUs that implement very fine-grained, low-overhead threads in hardware can often map the fine-grained tasks of the hierarchical SPMD kernel directly to hardware thread contexts. For a CPU architecture, the tool and runtime components of the system must bridge the gap between the many fine-grained tasks of the input program and the few hardware thread contexts available on the system. Challenging as this may seem, it is in practice much easier to automate an increase in parallelism granularity than it is to automate a decrease in parallelism granularity, effectively autoparallelization.

1.3 Organization of this Document

Chapters 5 and 6 will cover the core proposals in this dissertation for how such low-level optimizations should be directed for CPU architectures. The ex-

perimental results supporting the feasibility of performance portability will be presented in Chapter 7. The work begun in this dissertation has already made a significant impact in the research and industry communities, as evidenced by the commercial support of the development of the Multicore cross-Platform Architecture and the numerous research projects that built on the work presented here. Chapter 8 will highlight collaborative projects targeting the fine-grained SPMD kernel programming model to other architectures, including FPGAs, Rigel, and others. Chapter 9 will summarize the conclusions of this dissertation, including recognition of areas for continued development.

CHAPTER 2

CURRENT PRACTICES

The modern field of GPU computing had a major inflection point approximately six years ago with the first support for C-based programming languages for general computation on GPUs. Very quickly, the community discovered and published what worked well on GPU platforms and what did not at first. As the years progressed, GPU architects and application researchers continually pushed at the boundaries of what GPUs could do effectively, significantly improving performance for many workloads. In this chapter, I focus on characterizing the broad optimization problems facing high-performance software development in general, and examples of specific mechanics in current GPU programming models used to address those challenges.

2.1 Characterizing Optimization Patterns for Massively Threaded Systems

A segment of the parallel programming community has long been interested in characterizing the programming patterns that are effective for parallel systems [31, 22]. However, in private conversations, some of authors in that field have confided that they sometimes struggle with the fact that once a parallel program is implemented, the optimization process involves software development practices completely outside the domain of their structural patterns. I would therefore like to begin the academic discussion of a set of patterns systematizing the *optimization* of parallel programs. The optimization patterns were drawn in particular from an informal survey of the GPU Computing

Much of this chapter has been adapted from portions of a previously published work, ©2012 IEEE, reprinted with permission [48]. The original paper was written in collaboration with the many others who contributed to the development and analysis of the Parboil benchmarks.

Gems contributions [17, 18], and from a focused and detailed analysis of the Parboil benchmarks [49].

Because accelerators are highly parallel devices, many of the techniques specifically address general performance issues that arise from programming a highly parallel shared-memory architecture, such as contention and load imbalance. Some techniques are not specific to highly parallel architectures, but avoid especially severe performance cliffs given the design of today's accelerator architectures, such as the especially software-driven approaches to effective bandwidth utilization and locality management. Still other techniques are specifically targeted towards leveraging the benefits of a hybrid system, using the versatility of the CPU to not only process necessarily sequential code regions but to also precondition GPU kernel inputs such that kernels can be further optimized more than would be possible for general input.

Finally, parallel architectures are fundamentally a collection of sequential processing units. When a parallel architecture is well used, the performance limitation of a program on that architecture is the efficiency of the sequential programs running on each execution unit. Therefore, sequential program performance optimization is still an area of interest for the SPMD code executed on the accelerator. I will not discuss those techniques here, as they are well studied and not unique to parallel programming systems, but acknowledge that immature compiler technology sometimes will necessitate direct programmer implementations of "trivial" code optimizations.

I firmly believe that every one of the patterns described here has been explained in previous work, but note that previous descriptions of these patterns as they apply to GPU workloads are typically embedded within implementations of specific workloads. While I do not take credit for being the first to discover any of these individual transformations, there is useful insight to be gained by consolidating summaries of all those that applied to the Parboil benchmarks in a way that highlights their generality to a variety of GPU computing workloads. By gathering the optimization patterns together, anchored by the real benchmarks using them, we can study how they interact with one another, their variations among different applications, and their individual and cumulative results on real hardware systems.

To demonstrate the impact of each individual pattern we will review, for benchmarks where that pattern was particularly relevant, performance im-

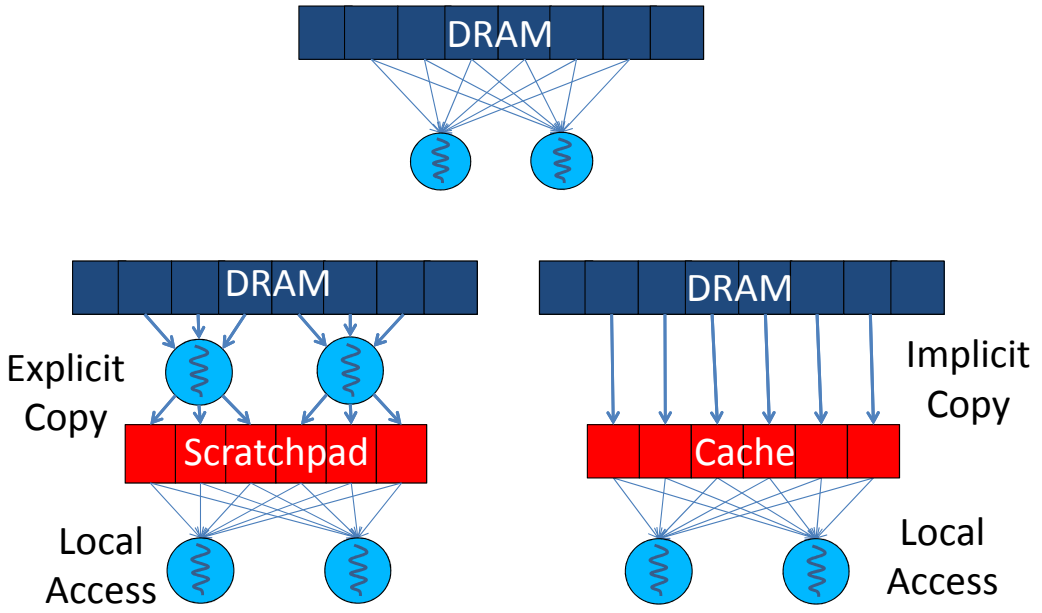


Figure 2.1: Tiling diagram for implicit storage (e.g. cache) and explicit storage (e.g. scratchpad)

provements from the highest-performing code we could write without that optimization to the highest-performing code we have. Except where noted otherwise, results in this section are collected from an NVIDIA Tesla S1070.

2.1.1 Tiling

Tiling is perhaps the most widely used and understood technique for best utilizing a tiered memory hierarchy. While the technique is fundamentally the same in sequential code optimization, the actual implementation can vary with the design of an architecture's memory hierarchy, as shown in Figure 2.1. Tiling in the context of a CPU architecture with a large-capacity, implicitly managed cache hierarchy typically means writing regions of code that operate intensively on smaller sections of memory. The regions could then be repeated many times for different sections, or tiles, of data. The application need not explicitly define the region of memory being operated on, as the hardware should automatically respond to the heavy usage of certain regions and retain those regions in the cache.

Table 2.1: Tiling results

Benchmark	Pattern performance impact
Stencil	3.15×
TPACF	1.12×
SGEMM	6.18×

One of the most obvious differences of current GPU architecture is explicitly managed on-chip memory, such as on the right side of Figure 2.1. To use the small-capacity, high-bandwidth scratchpad, software must explicitly move data into it before use. The threads themselves are mediators between DRAM and scratchpad, under the direction of source code written by application programmers.

Recent GPUs have also added small implicit caches to their general memory system, providing a hybrid of implicitly and explicitly managed locality mechanisms. What makes even cached GPUs significantly different from typical CPUs is that the ratio of cache capacity to the number of potentially active threads is incredibly small for GPUs. Indeed, the overall predicted trend for highly multithreaded processors is towards more limited resources per thread [26]. For instance, if all thread contexts were active in an NVIDIA Fermi GPU and cache space were partitioned among active threads, each thread would have a mere 32 bytes of L1 or L2 cache space.

Clearly, neither caches nor scratchpads in current GPUs were designed for CPU-style temporal locality and thread-private tiles, but for overlapping accesses among threads. A block of threads may collectively have 16kB of private cache or scratchpad space even when all thread contexts are active, which is often sufficient to hold worthwhile-sized tiles of data. Therefore, the software technique of tiling is still applicable for GPUs, but very often must take the form of cooperative tiling using the shared resources of several threads for sufficient impact.

The performance impacts of tiling are significant, as shown in Table 2.1. The 3× improvement in performance for the stencil benchmark corresponds to the fact that memory tiling reduces the number of bytes accessed from global memory per iteration from 5 words per thread to 1.25 words per thread on average. Performance does not increase by a full factor of 4× pri-

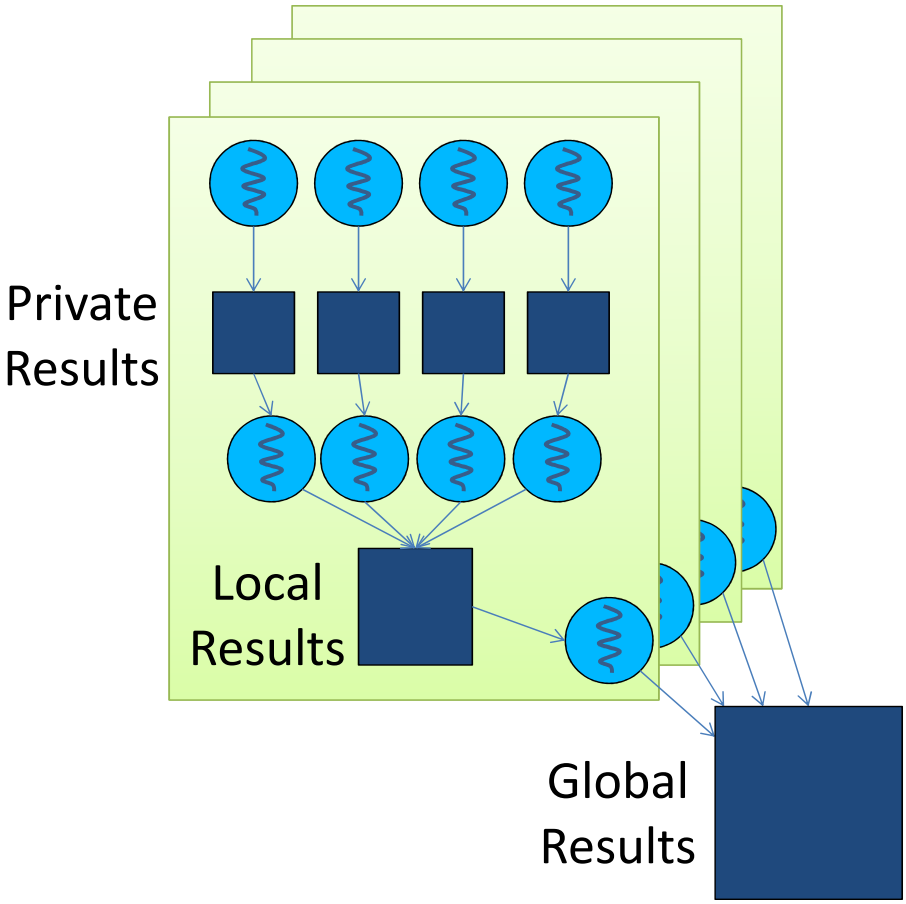


Figure 2.2: Example of the common hierarchical privatization pattern

marily because some accesses are still misaligned, not fully utilizing DRAM bandwidth. Although the results in Table 2.1 are for a cacheless GPU, our experiments in Section 3.3 verify, as any CPU high-performance programmer will assert, that software tiling is still critical for architectures with implicit caches.

2.1.2 Privatization

Privatization is the transformation of taking some data that was once common or shared among parallel tasks and duplicating it such that different parallel tasks have a private copy on which to operate. Parallel threads typically operate most efficiently when they can operate completely independently, avoiding coordination with other threads, but many parallel algorithms require threads to interact to obtain a final result. Privatization is

Table 2.2: Privatization results

Benchmark	Pattern performance impact
BFS	3.15×
Histo	2.26× (GTX 480)

applied to isolate regions of code where threads can operate independently and efficiently, before eventually combining results.

Figure 2.2 shows a common multi-level privatization pattern reflecting the hierarchical task decomposition common among highly parallel systems such as clusters or single-chip GPUs. Working up from the bottom of Figure 2.2, a global result is built from the partial results from many independent tasks (thread blocks in the case of a GPU). Those partial results are each in turn constructed from many more "private" results. This kind of privatization has applications in many different kinds of algorithms. Collective operations such as sorting or reductions will use this pattern, as will data structures such as histograms or queues.

One limitation of privatization is that the data footprint of the copies and the overhead of combining the copies scale with the amount of parallelism being exploited. Therefore, privatization is an extremely powerful technique for today's CMPs, with a relatively small number of threads, but somewhat limited for the levels of thread parallelism in highly multithreaded architectures. Often the "private" results are still shared by several GPU threads due to resource limitations, but are intended to be constructed with as little inter-thread cooperation as possible. As shown in Table 2.2, the BFS Parboil benchmark privatizes the output work queues, resulting in a 3× performance improvement over an unprivatized implementation. Privatization allows the BFS kernels to exchange more costly global memory atomic operations for shared memory atomic operations, and also collects irregular updates in shared memory before bulk-committing results to the global queue in a more regular pattern, improving bandwidth. For the histogram benchmark, the privatization transformation was ineffective for the S1070 due to shared memory capacity limitations; we therefore report GTX 480 speedups in Table 2.2 for that benchmark.

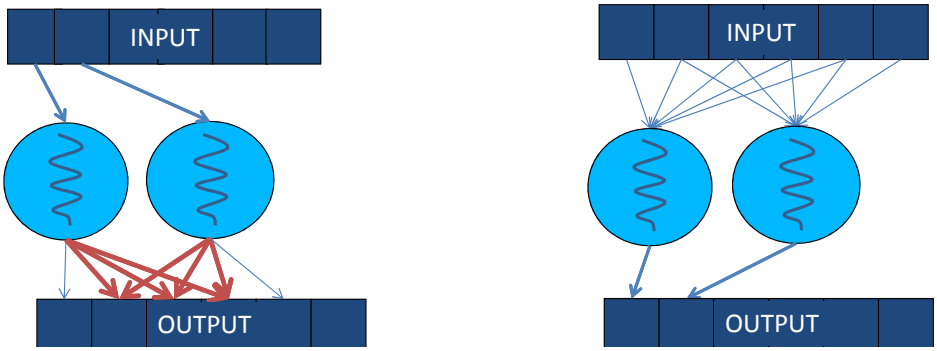


Figure 2.3: Depiction of a scatter-to-gather transformation

2.1.3 Scatter to gather transformation

A few Parboil applications demonstrate a computation pattern where an input datum would either contribute to many output elements, or contribute to one or more statically unknown output elements, such as shown in Figure 2.3. In either case, a common pattern for sequential implementation is to examine each input element, determine the output elements it affects, and update each one before moving on to the next input element.

This method works poorly as parallelism scales, because the output accesses are either contentious or random or both. Examining the previous techniques, we see that tiling is very effective on input data, and privatization is very effective on output data. However, a kernel implemented with a scattering approach has no input read sharing to tile, and no outputs with multiple updates from the same thread to privatize. In these situations, it is often very important to transform the code such that input elements are read-shared, but output elements are private to a parallel task. This is more palatable than the converse case because shared reads can be much more efficiently handled than conflicting writes, which typically require more costly atomic operations and coherence enforcement. A conversion to gather accesses means that privatization can be applied to output writes, reducing their cost, while techniques such as tiling can be applied to improve shared read efficiency.

Scatter-to-Gather transformation works exceptionally well when the range of inputs affecting an output can be found without direct examination of the input data contents. If this input-to-output mapping cannot be done statically, sometimes the transformation must be supplemented with a binning

Table 2.3: Scatter-to-Gather results

Benchmark	Pattern performance impact (GTX 480)
Histo	1.22×

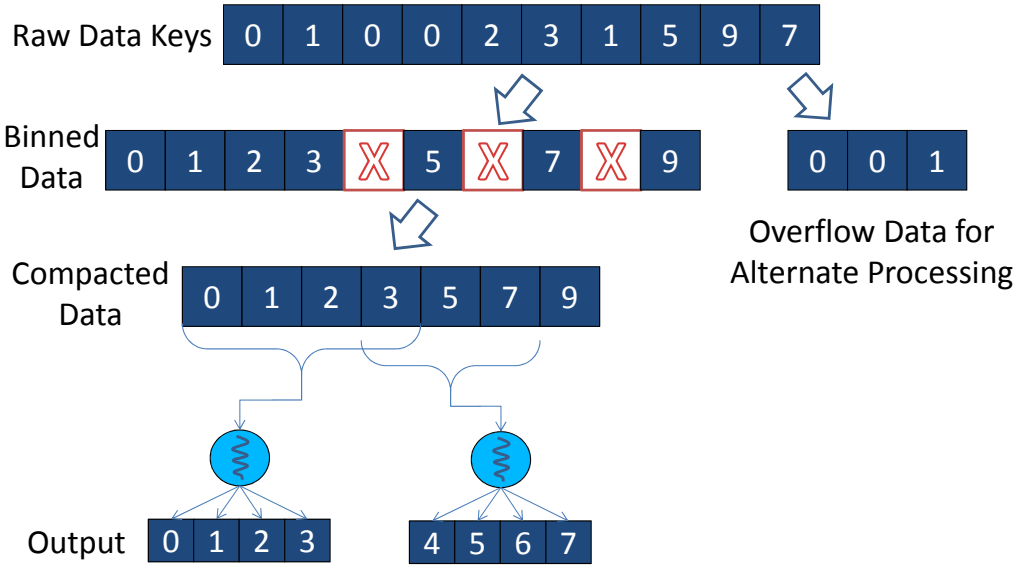


Figure 2.4: An example showing the binning, regularization, and compaction optimization patterns

operation. The Parboil Histogram benchmark gains about 20% performance on a Fermi architecture by using a gather-based approach instead of a scattering approach. Results are presented on the GTX 480 because the gather approach for the Parboil Histogram benchmark is only effective for GPUs with sufficient shared memory space to privatize a reasonable portion of the output histogram. The S1070 system does not have sufficient scratch-pad capacity for the scatter-to-gather transformation to improve histogram benchmark performance, and is therefore inapplicable as an optimization for that architecture.

2.1.4 Binning

A gather operation can be difficult to orchestrate without a method of determining, based on output location, which inputs contribute to that location.

Table 2.4: Binning results

Benchmark	Pattern performance impact
CutCP	12.0×

In the Parboil Histogram benchmark, for instance, a set of work-groups redundantly reads a section of the input data from off-chip DRAM, but each only processes the set falling within its own output range.[1] In general, the bandwidth cost of reexamining data scales with the amount of parallelism. Therefore, for some applications it is beneficial to first create a data structure creating a map from output locations to a small subset of the input data that may affect that output location, reducing the redundant reading of data. This data structure creation is called "binning" because it often reflects a sorting of input elements into bins representing a region of space containing those input elements. In the example of Figure 2.4, the unsorted data keys are examined and sorted into an array. If the input dataset were very regular, the sorting by key alone would likely create an efficiently accessible data structure. However, in the presence of irregularity, there will either be empty or overflowing bins for any fixed bin size, which should be addressed by some combination of the following two optimization patterns: regularization and compaction.

Binning can improve system performance in several ways. If the GPU is performing both the binning and the computation, the overhead of binning can be outweighed by the improved efficiency of the main compute kernels. Alternatively, the binning operation could be offloaded to the CPU, potentially making better use of all available system resources. Binning is applicable in particular for the CutCP benchmark, as shown in Table 2.4. The speedups from binning are often very high, because binning input data for a kernel's input changes the fundamental computational complexity of the kernel algorithm. A scatter-based kernel may not need binning to get comparable computational complexity, but even for scattering kernels, binning is important because it can facilitate privatization of tiles of output data.

[1] An example of a Scatter-to-Gather transformation without binning. Binning is not a useful technique for the Histogram benchmark because the actual histogram contribution is no more expensive in computation or bandwidth than the operation of sorting the data into bins would be itself.

Table 2.5: Regularization results

Benchmark	Pattern performance impact
SpMV	2.4×
MRI-Gridding	2.62×

2.1.5 Regularization

Load imbalance has been one of the banes of parallel processing throughout its history. Typically load imbalance is exacerbated when the level of parallelism being exploited increases. Architectures exploiting SIMD or SIMT vector processing suffer from low-level imbalance if the tasks assigned to different execution lanes process different amounts or kinds of work. GPU architectures are no exception. Furthermore, if threads co-executing in a thread block have imbalanced loads, the shared resources of the entire thread block may be occupied until the last thread completes, potentially reducing the real amount of thread-level parallelism available for the architecture to exploit.

Some applications that exhibit load imbalance can predict at run-time where and how the load imbalance will occur. In the example of Figure 2.4, we assume that the program can count the number of data elements for each key at much lower cost than actually calculating its contribution to its particular output. A preprocessing step can limit the amount of imbalance in work units executed on the GPU by identifying regions of load imbalance and proactively addressing them. In our example, during the binning process, elements that "overflow" a bin can be put in a separate data structure, which can be processed by some method less sensitive to load imbalance.

Regularization is the optimization pattern of preconditioning GPU kernel input to improve performance. Among the Parboil benchmarks, there are examples of processing work separately using a GPU kernel insensitive to imbalance, offloading irregular work for the CPU to process concurrently with the accelerated kernel. Other cases have no visible impact on the kernel code except that load imbalance and warp divergence are on average improved, resulting in higher performance. Regularization increases the efficiency of the primary accelerated kernels handling the majority of the processing, resulting in higher system performance overall, with impact as listed in Table 2.5.

Table 2.6: Compaction results

Benchmark	Memory buffer size reduction
SpMV	49%
MRI-Gridding	68%

2.1.6 Compaction

Compaction has been a technique within extremely parallel, shared-memory systems and programming models for quite some time as well. The fundamental issue is that when parallel work units produce a varying number of output elements into statically allocated output buffers, the buffer size must be overprovisioned. Because tasks determine output locations statically, unused holes or spaces in the output are the consequence of overprovision, such as those bins marked by X's in Figure 2.4. Output gaps interleaved with useful data cause bandwidth efficiency to drop for DRAM and cache architectures operating on transactions of larger, contiguous memory chunks. Compaction is a method of coordinating parallel tasks to dynamically determine output locations such that no holes are introduced.

If compaction were a separate processing step, as depicted in Figure 2.4, it would simply move all the useful data elements into contiguous addresses, filling in the holes, while keeping track of where each output section begins, as it will be data-dependent [5]. More often, and in the MRI-Gridding and BFS Parboil benchmarks where GPU computation produces compacted output, the compaction is integrated into the kernel producing output itself.

The benefits from compaction primarily stem from the reduced memory footprint of the compacted data format. Performance impacts are typically only meaningful for bandwidth-bound kernels, and even then only minimally if the overprovisioned regions of the buffers are mostly contiguous. Thus, we quote not performance results but memory capacity reduction effects in Table 2.6. Compaction is essential for the MRI-Gridding benchmark in particular, for which we cannot even run uncompacted versions of reasonable datasets on most GPUs due to insufficient global memory capacity.

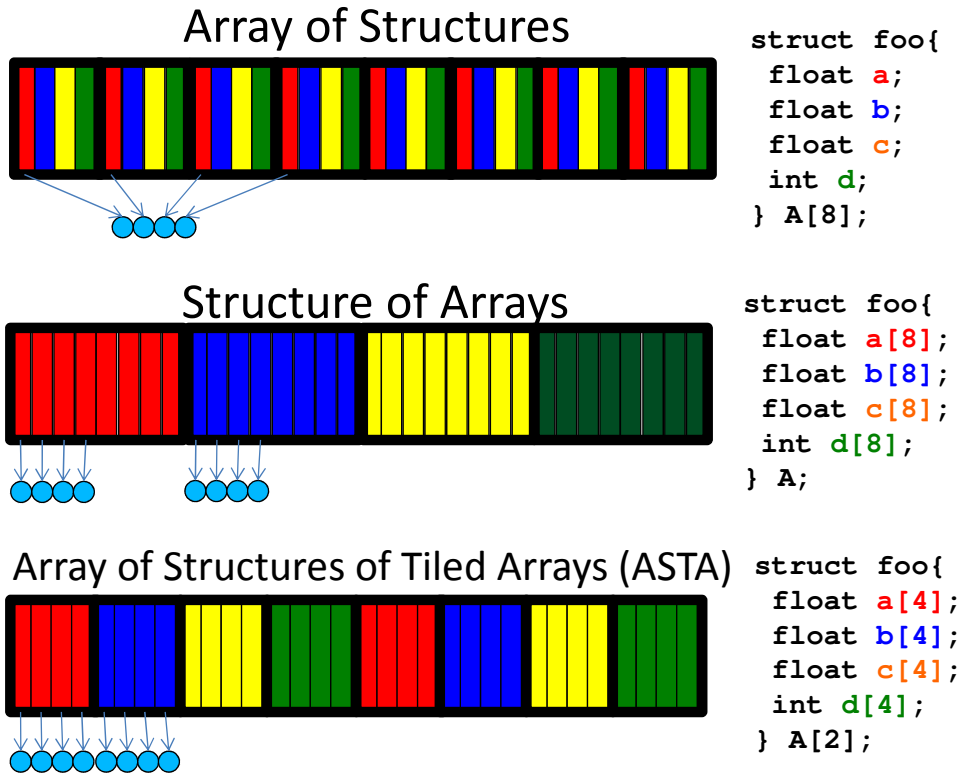


Figure 2.5: Data layout example for a collection of structures. Different layouts affect the coalescing of each warp access and the locality of multiple accesses.

Table 2.7: Data layout results

Benchmark	Pattern performance impact
LBM	11.0×
SpMV	1.21×

2.1.7 Data layout transformation

DRAM systems supporting both CPU and GPU architectures are designed to transfer data in large, contiguous lines or rows. Poor usage of CPU cache lines or GPU coalesced bursts will result in poor performance. However, GPU coalescing rules are somewhat harsher, because of the shorter time window over which the software could make use of a data burst from DRAM before any unused data is "dropped," requiring retransmission if needed at a future time. In some GPU architectures, the window is instantaneous, only exposed to a single SIMD instruction. More recent architectures introduce a small degree of caching extending this window, but because of the high degree of threading and the cache's low capacity, the window in practice is still very small. This is in contrast to CPU cache lines, which will typically sit in the cache for a longer period of time before being replaced.

Programmers work within the DRAM system design with well chosen data traversal orders or task index organization. If the elements in question are single-word data and closely associated with task indexes, a good choice of task index to element index mapping is typically sufficient to get good memory system performance. However, that pleasant situation is not always feasible. Sometimes, the data elements needed within a particular time window are not naturally adjacent to each other in the memory address space. Take, for example, the diagram in Figure 2.5, which shows a warp accessing fields from a set of cells for various layouts. In the top case, using C standard data structure layout, the warp access addresses with a large stride between them, requiring multiple memory lines of mostly unused data to fulfill the requests. The middle case of Figure 2.5 shows equivalent accesses with a structure-of-arrays layout, a common transformation. However, even the middle case results in a large distance between the addresses of two fields, which are likely to happen very close together in time.

Depending on the memory system design, performance can be improved

further by more complex layout transformation [53], perhaps resulting in a layout like that depicted at the bottom of Figure 2.5 where accesses to multiple fields will request adjacent, contiguous regions of memory. Specific examples of data layout transformation in the Parboil benchmarks include several instances of array-of-structure to structure-of-array transformations, a matrix transposition in SGEMM, and a transposed sparse matrix data storage format in SpMV. We specifically isolate the data layout transformation effect for the LBM benchmark, with an order-of-magnitude speedup as shown in Table 2.7. Overall, transformations for the purposes of achieving coalescing often achieve very high performance gains, such as the LBM, while layout transformations for improving memory level parallelism or avoiding moderate partition camping can effect a more modest improvement. Sung et al. report speedups ranging from 5% to 30% for already coalesced accesses in different benchmarks [53].

2.1.8 Granularity coarsening

Granularity coarsening has been anecdotally described in many application optimization papers, perhaps most rigorously by Volkov in regards to linear algebra kernels [54]. When a larger task is decomposed into a set of fine-grained work-items, there is almost invariably some amount of overhead introduced in the problem decomposition. The overhead may vary for different algorithms and kernels, but almost every kernel will exhibit some inefficiencies in recalculating values like address offsets or other seemingly "small" operations in many threads. The finer the decomposition, typically the larger the overhead incurred. In addition to innate implementation inefficiencies, most real systems incur some fixed costs creating or scheduling parallel tasks, and communication operations tend to become more costly as the number of communicating tasks grows.

The CUDA and OpenCL programming models lend themselves to an "elemental" style of decomposition, where the source code of the kernel is scalar, processing a single element, with as many threads created as there are elements to process. With this extreme level of decomposition, the level of redundancy and other inefficiencies can be surprisingly high, but difficult to address within the elemental-function methodology as the cost of commu-

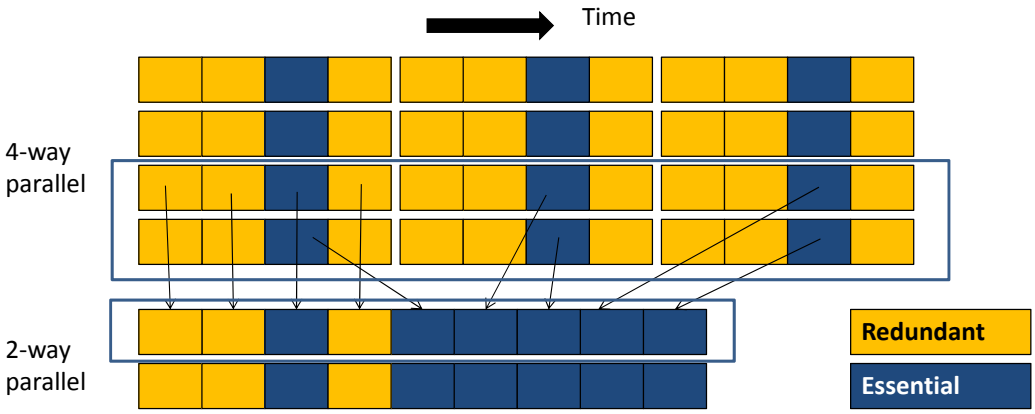


Figure 2.6: Granularity coarsening and resulting efficiency gains. Each shaded box represents an executed instruction or operation.

Table 2.8: Granularity coarsening results

Benchmark	Pattern performance impact
SGEMM	1.96×
CutCP	1.3×

nicating between different tasks is still higher than the cost of redundant computation.

Granularity coarsening is essentially a de-parallelization of a program. Instead of executing code where each thread processes one element, each thread processes several. Figure 2.6 shows a coarsening transformation by a factor of six. By putting several threads together, redundant operations that were previously executed once by each original thread have their redundant executions reduced by a factor of the degree of coarsening. Furthermore, what had been shared reads or conflicting writes to a variable in the untransformed code become private uses of data. In the example of Figure 2.6, although task parallelism was reduced by a factor of six, the total number of operations required to compute the full output was reduced by nearly two-thirds. The efficiency gains make incremental coarsening worthwhile so long as the amount of parallelism is still sufficient to occupy the parallel resources of the device. Examples of specific efficiency gains are shown in Table 2.8.

```
1 __kernel void MatMul( __global float *A,
2                        __global float *B, __global float *C) {
3   float result;
4   __local float A_tile[TILE_WIDTH];
5   __global float *A_line = A + get_group_id(1)*A_WIDTH;
6
7   result = 0.0f;
8   for (int i = 0; i < A_WIDTH; i+= TILE_WIDTH) {
9       A_tile[get_local_id(0)] = A_line[i + get_local_id(0)];
10
11    barrier(CLK_LOCAL_MEM_FENCE);
12
13      for (int ii = 0; ii < TILE_WIDTH; ii++)
14        result += A_tile[ii] *
15          B[(i+ii)*B_WIDTH + get_global_id(0)];
16
17    barrier(CLK_LOCAL_MEM_FENCE);
18  }
19  C[C_WIDTH * get_group_id(1) + get_global_id(0)] = result;
20 }
```

Figure 2.7: A simple, portable matrix multiplication kernel in OpenCL

2.2 Practical Details of Implementing High-Performance Accelerator Code

The previous sections primarily highlight high-level optimizations necessary for optimizing parallel code in general. In practice, there are some specific details about precisely *how* the high-level optimizations are expressed in a particular language such as CUDA or OpenCL. Clearly, if different vendors have seriously divergent expectations of how programmers should use the language, it is impossible for programmers to write portable code. OpenCL programmers have already adopted certain practices to get performance on a variety of GPU and CPU platforms. This section outlines those practices and explains how they match the major architecture principles of GPUs and CPUs today. Figure 2.7 shows a listing of an OpenCL kernel for multiplying two matrices in single precision, which we will use as an ongoing example in this paper. If these practices could be performance-portable in theory, then achieving the goal of a portable language will require that both GPU and CPU implementations of OpenCL deliver high performance for codes written with these programming practices.

2.2.1 Parallelism: Threads and SIMD

The OpenCL programming model includes a two-level decomposition of work. Although the decomposition is just a two-level hierarchy of parallel tasks, the two levels have very distinct performance implications. All of the work-items in a work-group are guaranteed to be scheduled together, allowing them to coordinate more closely. Work-groups cannot make any assumptions about scheduling or co-scheduling of other work groups, which means both that atomic operations must be used to guard critical sections, and that dataflow-based programming patterns that rely on multi-group barriers are disallowed.

The groups of co-scheduled tasks are a clear source of thread-level parallelism, and are exploited in that way by nearly all implementations. Less obviously, perhaps, the work-items within a group are exploited as a source of vector-level parallelism on all GPU implementations known to the authors. The reasoning is that OpenCL's single-program multiple-data programming model will often naturally lead programmers to write groups of work-items with nearly-identical paths through the program in many situations. Even if it were not fully natural, the GPU implementations made this programming pattern fastest on their architectures from the beginning, and every GPU programming guide discourages "divergence," or writing programs such that different work-items within a work-group take different paths through the program, making SIMD less effective.

To be performance-portable, the amount of parallelism in and among work-groups needs to be flexible. Work-groups must be at least as wide as the native SIMD width of the machine, but never larger than the machine's capacity to schedule locally and simultaneously. The number of work-groups should be several times larger than the number of processors on the device to enable load-balancing. Given the variety of architectures, it is unclear whether these constraints can be met for all platforms with a fixed-size work-group. At minimum, performance-portable programs have to query the device parameters at runtime and choose group sizes appropriately, or allow the platform itself to choose a group size suitable for itself.

Our example kernel program computes a single output element with each work-item, so the number of work-items for a reasonably large matrix would be substantial. In line 19, `get_group_id(1)`, the second element of the group index tuple, is used as the output row index, indicating that each work-

group should process some contiguous section of a particular output row. Therefore, every work-item in a work-group will need to access the same row of data from the A matrix. Also, there are no divergent branches in the kernel. Every condition is independent of the work-item index, so the entire path taken through the program execution is uniform across all work-items. Therefore, SIMD groups of work-items will be fully exercised, without the need to predicate any SIMD lanes at any time.

One interesting point about the OpenCL programming model is that by using groups of work-items as the basis for SIMD execution, the OpenCL implementations are providing the user a way to exploit SIMD without requiring them to program to a specific SIMD width. This is critical for portability, because different CPU and GPU architectures have widely varying SIMD widths. Work-groups that are significantly larger than an architecture's SIMD width enable additional thread- or instruction-level parallelism, as the group can be divided into multiple SIMD-width units. Subsequent proposed languages captured this insight particularly well also, such as the ISPC programming model that advocates SPMD programming as an easy and effective way of writing SIMD code for x86 CPUs [41].

2.2.2 Spatial and temporal memory locality

A large part of writing high-performance code is managing data locality well. In a recent survey article of seven broad GPU programming optimization techniques, only one was not directly related to memory locality management [50]. The OpenCL programming model makes explicit certain architectural realities that other languages try to keep abstracted away. Large, coherent memories are inherently more expensive to access than small, local data resources. As is typical, we will divide our discussion of locality into two major classes: spatial locality and temporal locality.

Spatial locality originally came from the observation that in sequential, stack-based programs, if a particular address was accessed, other addresses nearby were likely to be accessed soon in the future. Today, spatial locality is almost a performance requirement, because we build our entire memory systems out of large-line data transactions, such as cache lines and DRAM bursts. Furthermore, building hardware data structures with many ports for

independent simultaneous access is very expensive. In particular, this means that even if a wide SIMD unit has gather and scatter capabilities, spatial locality among the addresses accessed will significantly reduce the number of unique memory lines touched by the access, which means a much reduced overall throughput demand on the memory system. In fact, OpenCL programmers are specifically encouraged to assign work-items to data elements such that memory accesses are "coalesced" [36, 35, 1, 19]. Formally, an access is considered coalesced if it can be decomposed into the form:

`uniform_base_address + (get_local_id(0) % SIMD_WIDTH).`

A coalesced access causes all of the work-items in a particular SIMD execution bundle to access a set of contiguous elements in memory for the given instruction or expression. To be tolerant to varying SIMD widths across architectures, many programs assign the entire work-group to a contiguous set of elements, such that any contiguous subdivision of the group will have a coalesced vector access.

Temporal locality is somewhat more complex to manage portably. Given that inter-group scheduling is out of the programmer's control, her focus is on temporal locality within each group. Furthermore, it is primarily temporal locality in accesses from multiple work-items that needs explicit management; task-private temporal locality is usually handled through simple register promotion. There are two possible approaches to achieving temporal locality in OpenCL: explicitly managed local memory buffers or assuming an implicitly managed cache. Older GPUs prevalent during OpenCL's drafting had very limited caching support, forcing programmers to manage temporal locality through explicitly-managed scratchpads. More recent GPUs from NVIDIA, AMD, and Intel all include memory caches all the way down to the L1 level, which simplifies but does not eliminate the need to specifically consider how temporal locality is managed. Even if a cache is present, it can be exploited one of two ways on a GPU: explicitly controlling the execution order with work-group barriers, or relying on implicit, round-robin scheduling patterns on GPUs to keep all work-items in a work-group roughly in phase with each other. Either of these mechanisms will ensure that memory locations accessed repeatedly by different work-items in the group will likely still be in cache.

In the given example, every access to global memory is perfectly coalesced

across the entire work-group, because the index expressions on lines 9, 15, and 19 are all of the form `uniform_base + get_local_id(0)`. Therefore, this kernel achieves a high percentage of peak global memory bandwidth consumption, even though more advanced tiling algorithms could reduce the total number of global memory accesses significantly. The shown kernel also uses the OpenCL local memory as a software-managed cache. Work-items collectively copy a tile of data from global to local memory on line 9, and then iterate through all of the elements in the tile in lines 13-15. Barriers are necessary to separate each dynamic computation section from the preceding and succeeding local memory update regions.

Note that controlling locality on GPUs always relies on being able to switch between actively executing work-items frequently and with low overhead. Round-robin instruction scheduling is another way of saying that the hardware makes frequent implicit moves between actively executing work-items to balance their progress. Frequent barriers are effectively programmer commands to suspend currently executing work-items at the barrier so that other work-items can catch up. Therefore, we can say that a fundamental requirement of an OpenCL implementation that supports performance portability for current developer practices is a low-overhead mechanism for switching execution between work-items.

2.2.3 Summary

Table 2.9 shows a compact representation of which optimization patterns were relevant for each benchmark. Note that the table does not convey the relative importance of each optimization pattern to each benchmark. In my experience and that of my colleagues, a given benchmark's performance improvement due to optimization is typically dominated by one or two optimizations, with others making smaller contributions. Also, note that certain optimization patterns are widely applicable, such as granularity coarsening, while others are only applicable to applications with certain characteristics, such as binning. Finally, we would like to point out that some of the optimization patterns are clustered. Regularization and compaction, for instance, are typically combined rather than applied separately, because both are applicable for similar workload characteristics.

Table 2.9: Applicability of optimization patterns to each benchmark

Benchmark	Tiling	Privatization	Scatter to Gather	Binning	Regularization	Compaction	Data Layout Transformation	Granularity Coarsening
cutcp	X		X	X	X			X
mri-q	X						X	X
mri-gridding	X		X	X	X	X		X
sad	X							X
stencil	X							X
tpacf	X	X						X
lbm							X	
sgemm	X						X	X
spmv					X	X	X	X
bfs		X			X	X		
histo		X	X					X

We are not necessarily convinced that these optimization patterns are a comprehensive list, and would not be surprised to see other application domains introduce optimization patterns not represented in these studied benchmarks. However, given the generality of the patterns that we have seen so far, we do suggest that at least some of these optimization patterns will be applicable to almost any GPU application workload.

This section has described optimizations for resource utilization in binary terms: meeting or not meeting certain criteria for "well-behaved" code. Following the optimization guidelines in this section will generally result in high hardware utilization efficiency, i.e. reaching an achieved bandwidth or execution efficiency close to the architecture's peak. Unfortunately, the guidelines alone cannot predict whether bandwidth or execution throughput will be the ultimate limiting performance factor for a given architecture. Yet "well-behaved" code does not guarantee high performance. To ensure high performance, code must apply the optimizations from Section 2.1 for increasing parallelism, decreasing output contention, and increasing locality as much as possible. For instance, the example kernel program of Figure 2.7 is not as well tiled as it could be, and only captures locality among a group of outputs in the same output row. Better tiling techniques would further increase the amount of captured locality and improve performance [54].

Finally, many of the optimization patterns could be applied to any parallel system, and are still relevant for today's multicore CPUs after decades of research and experience with high-performance parallel systems. While innovation may still surprise us, it appears likely that manual program optimization according to the patterns presented in this section will continue to be relevant for parallel architectures in general, and GPUs specifically, for years to come. Current software developers for high-performance applications would do well to brush up on these optimization patterns, and to continue to publish optimization insights either applying these general patterns to specific contexts, or possibly describing new optimization patterns. And low-level languages intending to support portable, high-performance code development needs to support the expression of these optimizations in ways that can be portably implemented across architectures.

CHAPTER 3

PERFORMANCE IMPACT OF ARCHITECTURES

Having explored the workload and optimization characteristics demonstrated by the Parboil benchmarks, we can now discuss in more detail the impact of different GPU architecture features on workload performance. Such a study could be approached in multiple ways, with each method leading to particular insights. To address how these trends may affect performance portability, we would like to focus on three primary methods of inquiry. In the first, we assume that GPU workloads are in a state of perpetual hardware-software co-design and optimization. Our experimental results under this methodology will show how optimized software targets new features in each successive hardware generation, and how architecture changes amplify the benefit of particular optimization patterns. Secondly, we examine the other end of the optimization spectrum, to see how a simple, unoptimized implementation of each of the Parboil benchmarks improves with successive hardware generations. These results will tell us how well implicit or compiler-targeted hardware features are finding ways to improve performance without explicit software support. Finally, we would like to compare the performance gains of code optimization for each architecture generation, to understand how different architectures change or preserve the optimization process. Unfortunately, the Parboil data sets for the MRI-Gridding benchmarks were large enough that the GPU global memory capacity of both the 9800 GX2 and the S1070 were insufficient to collect baseline results. This both highlights the necessity of the compaction optimization for this benchmark, and prevents us from analyzing its performance any further.

Much of this chapter has been adapted from portions of a previously published work, ©2012 IEEE, reprinted with permission [48]. The original paper was written in collaboration with the many others who contributed to the development and analysis of the Parboil benchmarks.

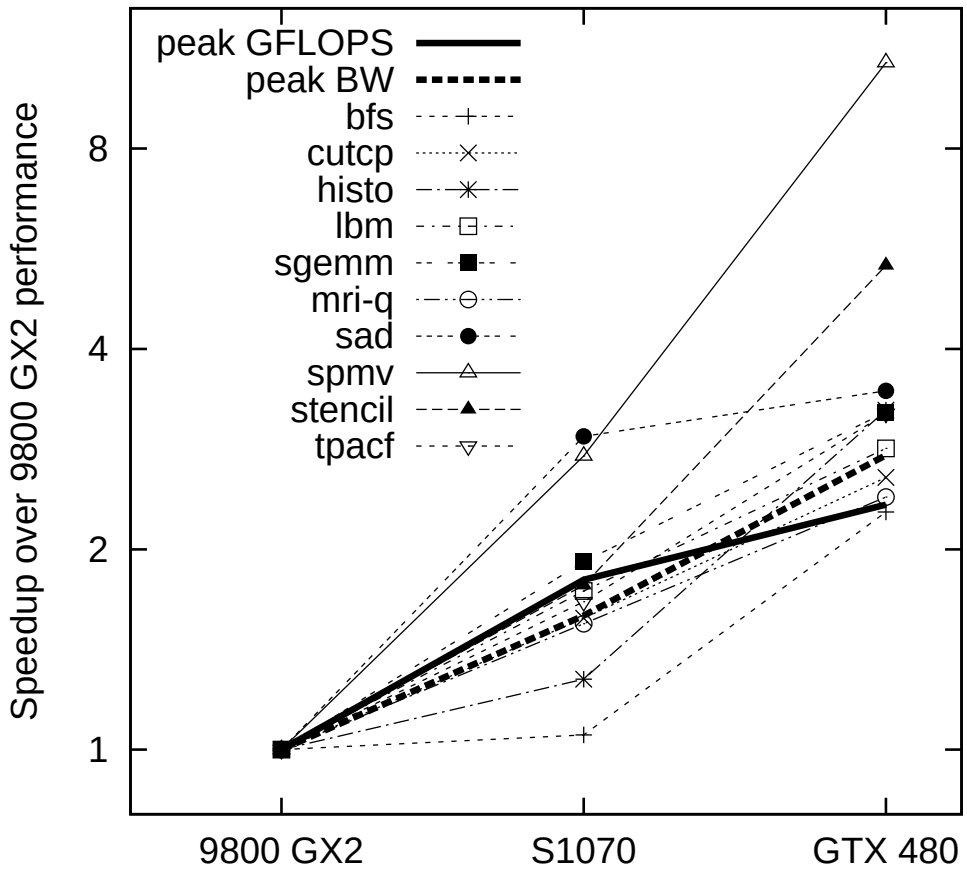


Figure 3.1: Performance of code optimized for each successive GPU generation, plotted against raw throughput and bandwidth scaling for comparison

3.1 Hardware-Software Co-Design Results

We optimized each of the benchmarks for each of the GPU architectures studied, and recorded the speedups achieved by successive architecture generations in Figure 3.1. Because the implementations for the earlier generations were already optimized around most of the performance cliffs of those architectures, the advances made by successive generations are typically near the increase in raw bandwidth or instruction throughput. Architecture feature improvements with a moderate impact on optimized workload performance include increased register file capacity, which boosted the performance of applications such as SGEMM and SAD in particular because of the extensive register tiling of those benchmarks. The performance improvement for register tiled benchmarks came less from the opportunity of additional register tiling, which reaches asymptotically low incremental benefits and had little impact in practice, but more from the architecture's ability to increase occupancy for the same degree of register tiling.

The single feature with the most performance impact overall was the global memory cache added in the GTX 480 generation. Even for optimized codes, scratchpad usage can introduce meaningful inefficiencies into the software. That overhead is typically overcome by the performance improvement due to captured locality otherwise unattainable in the absence of a cache, but does put scratchpad at a disadvantage to implicit caches for certain workloads. Furthermore, the GTX 480's cache captures what private scratchpads never can: shared locality among different thread blocks and access patterns with irregular locality. The spmv benchmark performance increases for the GTX 480 primarily from the caching of irregular accesses to the dense vector. The stencil benchmark benefits from caching because any memory tiling approach in that benchmark must address the fact that the size of input tile needed to compute an output tile does not match the output tile size. Thread block sizes must be chosen to fit either the input or output tile size, resulting in inefficiencies from idle threads or increased software complexity for explicitly copying input tiles, respectively. In addition the tile borders overlap with the working sets of other thread blocks, exposing locality that cannot be captured with private scratchpad memory. The version of the stencil benchmark optimized for the GTX 480 actually avoids memory tiling, improving the efficiency of the instruction stream by relying on the cache and thread block

scheduling policy to capture locality. The cache also significantly improves the BFS benchmark's performance by caching the end of the output queue while threads in a block incrementally add to its tail.

Surprisingly, atomic operations to shared memory had less performance impact than we expected. On further analysis, we found that privatization optimizations had reduced contention on shared memory locations requiring atomic updates to the point that the overhead of our software atomic updates, which scales with contention, was not so high as to make hardware assisted atomics indispensable. While our iterative atomic update methods were limited to certain situations, the versions optimized for the 9800 GX2 targeted those situations specifically, resulting in sufficient atomic update performance.

Finally, we note that the BFS benchmark in particular does not scale very well with regards to the number of SMs in the system. The BFS kernel that fills the GPU and performs device-wide barrier synchronizations in certain kernels does not perform as well on the S0170 as on the narrower 9800 GX2 and GTX 480 devices. As it is likely that machine widths will be increasing on average in the future, it seems reasonable to expect that using atomic operations for chip-wide synchronizations will become increasingly inefficient, and should perhaps be avoided if possible.

3.2 Baseline Performance Improvements

Figure 3.2 shows the performance improvement of a single, optimization-agnostic implementation across the different GPU generations, again plotted against the raw throughput and bandwidth improvements of the devices themselves. The definition of "unoptimized" is somewhat slippery, because it is always possible to write less efficient code by doing some kind of useless computation. Our philosophy while writing these baseline versions was to write the simplest functional code that seemed reasonable to us. We cannot claim that the baseline versions of all the benchmarks are equivalently unoptimized, but believe we can still learn some useful insights by paying attention to what "inefficiencies" are automatically mitigated or eliminated by particular architectures.

Generally, we can see that the performance trends are definitely positive,

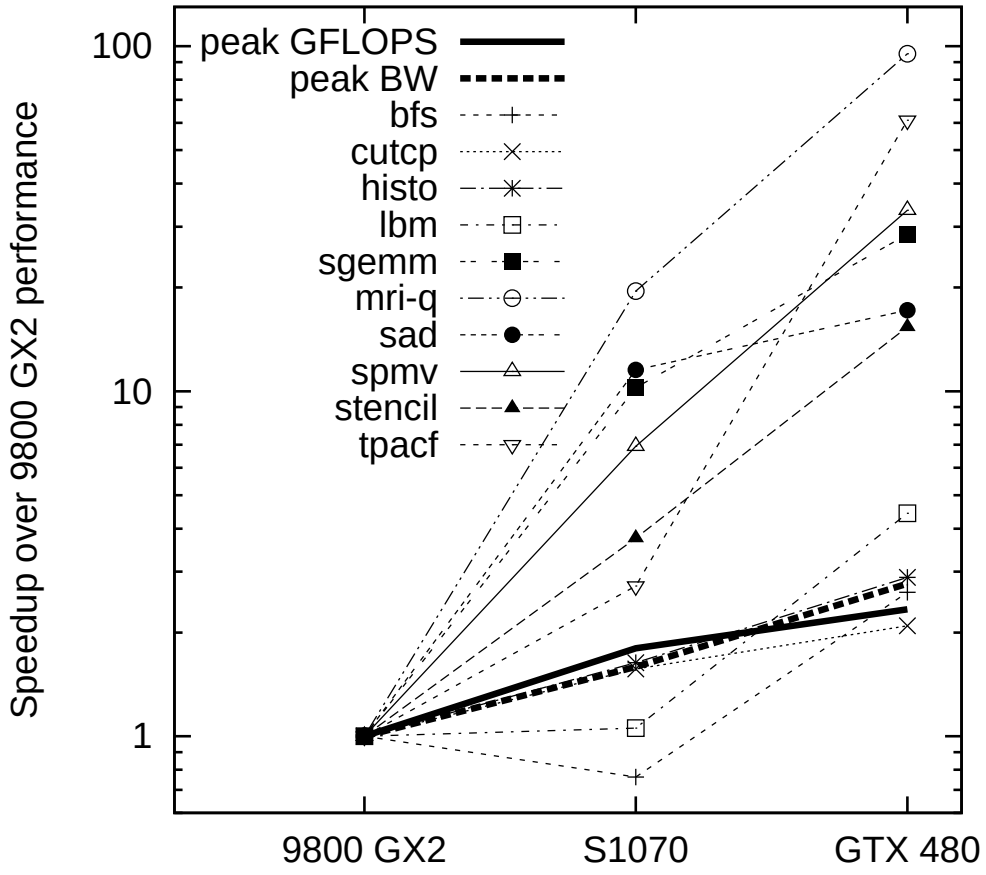


Figure 3.2: Performance of unoptimized code across GPU generations, plotted against raw throughput and bandwidth scaling for comparison

and significantly higher in magnitude than the improvement of optimized code versions. In several instances, one architecture generation brings order-of-magnitude speedups over the previous generation, mostly for benchmarks with artificially poor memory bandwidth performance for uniform or misaligned accesses on the 9800 GX2 surging in performance when those limitations were removed in the S1070. The Fermi generation improved global memory broadcast accesses further by automatically promoting them to use the constant memory cache. Broadcast accesses are those where each thread in a warp loads from exactly the same address in a particular instruction. The GPU's constant cache supports this access pattern with very high performance. The constant cache design of the GTX 480 architecture enables the CUDA compiler to automatically transform accesses to use it under certain conditions, which reduces pressure on the general global memory cache and results in significant speedups for unoptimized mri-q, tpacf, and sgemm implementations.

Despite the raw bandwidth improvements of the S1070 over the 9800 GX2, the strided access pattern of the unoptimized lbm benchmark saw practically no performance improvement. It was not until the cache of the GTX 480 that its performance meaningfully improved. The GTX 480 cache also had significant impact on the performance of codes with had shared locality in the accesses among thread blocks that was not exploited by explicit memory tiling, in particular the stencil benchmark.

3.3 Optimization and Architecture Interactions

Finally, we examine the performance improvements of optimization for each benchmark and GPU generation, with results presented in Figure 3.3. The overall trend is significantly downward, implying that optimizations in general are becoming less critical over time. Conversely, we can say that many of the performance cliffs avoided by optimization are becoming less steep with successive architecture generations. However, there are some exceptions. The binning optimization pattern, exemplified by the cutcp benchmark in particular, results in consistently high speedups due to the change in fundamental algorithmic complexity, as should be expected. Also, while architectures are becoming slightly less sensitive to imperfect access patterns, good data lay-

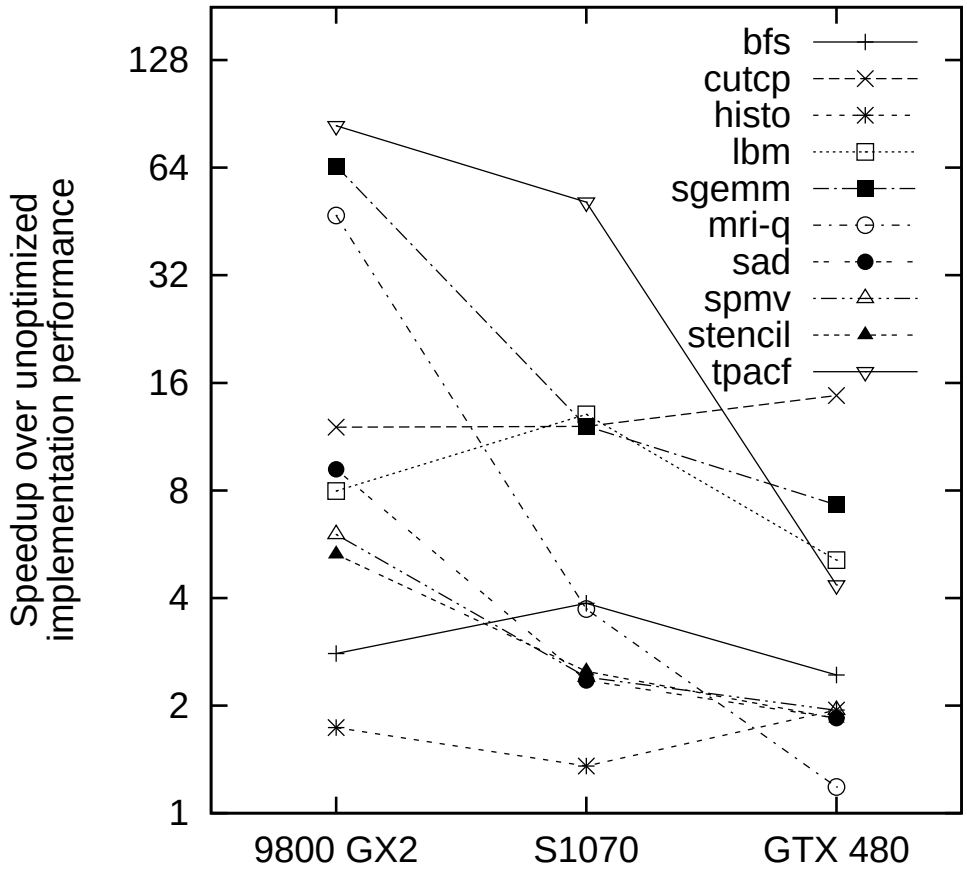


Figure 3.3: Speedup of optimizations for each GPU generation

out remains extremely important, as exemplified by the lbm benchmark's 5× performance improvement from layout transformation, even on the Fermi architecture. For the sgemm benchmark, register tiling results in consistently high speedups. For such "simple" codes, the primary bottleneck is instruction stream efficiency: how many instructions compute necessary floating-point operations relative to how many instructions calculate addresses or move memory around. Even when artificial bandwidth inefficiencies are addressed by the Fermi architecture, a significant speedup can be expected from good register tiling.

3.4 Summary and Conclusions

Hundreds of articles have been published on optimizing applications for GPUs, and for good reason. Especially when the general computing languages and workloads were new to GPU architectures, it was not enough for the application to simply have good locality, but the locality needed to be in very specific forms for the system to recognize and support. But some of the "worst" days of GPU computing are now behind us. Although legacy GPUs will still linger in the marketplace for several years, NVIDIA and other vendors seem to be getting on track with the design philosophy that the system cannot put too many constraints on software before good performance can be reached.

In summary, experiments show that most core optimization patterns continue to be relevant, but that architecture-specific constraints on how those optimizations are expressed are reduced. It would be difficult to make the case for performance portability for the earliest GPU generations supporting general computing because of all the ancillary constraints on high-performance source code not directly related to locality or divergence in the abstract. However, as new architectures become more flexible in terms of their support for slight deviations from the architecture's ideal expression of locality or regularity, there is more room in the intersection of various architectures' constraints, widening the opportunity for performance portability of code that can fit within that intersection.

CHAPTER 4

MEMORY SPACE MAPPING AND COMPILER DETAILS

This chapter describes some implementation details common to the topics discussed in the remainder of this document. An implementation of an accelerator kernel-programming language on a non-GPU platform generally includes three primary components: the source-to-source compiler from the kernel programming model to multithreaded C code of some kind, a back-end compiler for generated object code, and a runtime library for invoking such kernels. Chapters 5 and 6 will discuss different methods for translating computation and thread-private variables from a general perspective. The methodology of both sections must be supplemented with some additional handling of the unique SPMD kernel programming memory model, described here.

Parts of the work in this dissertation have been implemented for both CUDA and OpenCL. The CUDA implementation was released as the MCUDA framework, where the source-to-source compiler was developed within the Cetus source-to-source compilation framework [29], with slight modifications to the IR and preprocessor to accept ANSI C with the language extensions of CUDA version. Our OpenCL-to-C compiler was implemented as an automatic source-to-source translator in the Clang frontend [10] of the LLVM compiler infrastructure [28], and is being released as the Multicore cross-Platform Architecture (MxPA).

4.1 Implementing the Memory Spaces

4.1.1 Globally shared memories

Given that the C language does not have a memory model supporting read-only memory spaces allowing multiple initializations, all of the globally shared device memory spaces will be treated similarly: global, constant, and texture objects. Global and constant variables are already required to be either statically allocated at file scope or potentially dynamically allocated in host code (global memory only). In the C memory model, both of these situations are congruous with the implementation of the execution model, and need not be further modified.

It is useful to note that although CUDA and OpenCL define separate memory spaces targeting scratchpads, global DRAM, and private registers of GPU architectures, all data reside in the same shared memory system in C. A typical CPU system provides a single, cached memory space that implements the features of all of the CUDA/OpenCL memory spaces, at least to a reasonable degree.

4.1.2 Block-shared memory

Objects in the shared memory space of CUDA (OpenCL local memory) are defined as private to each thread block. Because such objects are shared from the perspective of the serialized threads, they are treated as shared objects by the compiler serializing the thread blocks, and are not targeted for any kind of scalar expansion. C scope semantics specify that objects declared within the kernel function are created on the program stack when the function is called. This is exactly the effect desired: an instance created when the thread block begins computation on function invocation, not visible to any other thread blocks, and freed as soon as the thread block completes the function. Thus the shared memory variables need only their language-specific specifier removed.

CUDA allows a special case for statically unsized shared memory arrays. The size of these arrays is determined in an additional parameter to the kernel invocation, and could be any size supported by GPU hardware. To handle this case, the compiler must remove the static declaration of the

array, and instead replace it with a dynamically allocated memory object. OpenCL does not have language mechanisms for declaring local memory of undefined size, and instead allows the user through API functions to request a dynamic amount of local memory, passing a pointer to that allocated space as a parameter to the kernel.

In addition, it is possible, though rare, for programmers to specify shared memory objects at file scope. This may be done to allow multiple device functions to access the same array without having to worry about CUDA's restrictions on passing pointers to shared memory. If the programmer uses this feature, the compiler ought to address it specially, as C semantics would allocate only a single object at file scope. Without privatization, this effectively requires the execution of thread blocks to be serialized. The solution to this issue could potentially dovetail nicely with the method of addressing statically unsized arrays, by removing the static declaration and introducing a pointer parameter to a privatized object.

Neither of these special cases is currently handled, as their use is rare, but the compiler could be extended to address them both. The key attribute required is that, similarly to the statically sized arrays within a kernel function, the program semantics allow each actively executing block access to a private scratchpad space. Under C semantics, dynamic sizing must either be implemented as heap-allocated memory, or allocated memory of some fixed maximum size, which the program is free to use as much as required up to that maximum. Performance tuning experiments have already shown that better performance can be obtained on some architectures if the official CUDA capacity limits are surpassed. This would undoubtedly be true again if another arbitrary limit were enforced on our system, even though the OpenCL API forces us to report such a limit.

4.2 Work Distribution and Runtime Framework

The compilation stages for the device code generate kernel functions that can be invoked with a single blockIdx value to complete the computation for that thread block. Somewhere between the host code kernel invocation and the block-level function, there needs to be a system that will take the specification for the kernel launch, enumerate the work units defined by the thread

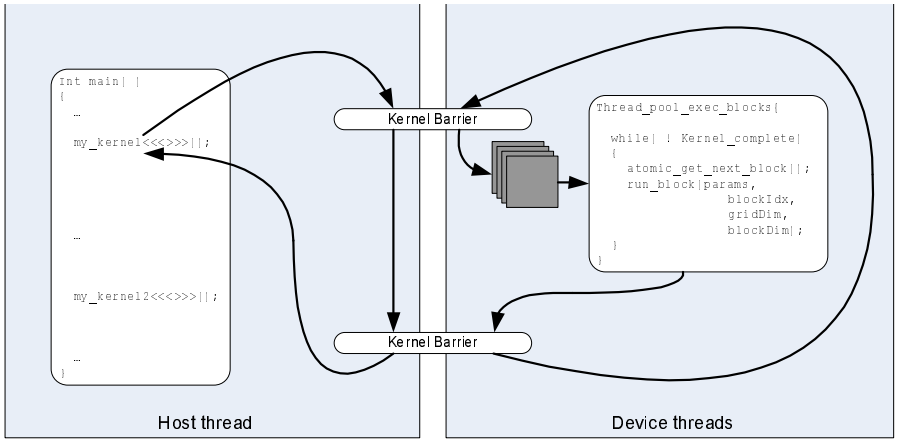


Figure 4.1: Parallel kernel runtime framework using dynamic block assignment

blocks, and prepare and assign those work units to threads. For a CPU that gains no benefits from multithreading, an efficient way of executing the kernel computation is to simply introduce an intermediary function between the host code and the device code that includes a nested loop over block indexes. However, CPU architectures that do gain performance benefits from multithreading will likely not achieve full efficiency with this method. Because these blocks can execute independently according to the programming model, the set of block indexes may be partitioned arbitrarily among concurrently executing threads, and can at least partially execute asynchronously with host code, if the host is also encapsulated in another thread. This allows the kernels to exploit the full parallelism expressed in the programming model.

For both the CUDA and OpenCL implementations, we implement a runtime library for the symbols of the host API of the language. The currently available MCUDA distribution includes parallel kernel runtime implementations using OpenMP or Pthreads. MxPA currently only supports OpenMP, but work is underway to support other threading models as well. Figure 4.1 illustrates the runtime framework described in the remainder of this section as implemented in Pthreads as an example implementation.

For CUDA, the kernel launch is designated by an annotated function call directly to the symbol for the kernel name. In MCUDA, the kernel launch

statement is translated into a function call to the runtime kernel launch routine. The function call specifies a reference to the kernel function to be invoked, the kernel configuration parameters, and the parameters to the kernel function itself. In the runtime library kernel launch routine, the host thread stores the kernel launch information into global variables and enters a barrier synchronization point. A statically created pool of worker threads, representing the device in the CUDA model, also enters the barrier. On exiting, each worker thread reads the kernel launch data and begins executing blocks. The host thread then enters a second barrier to wait for kernel completion before returning to the host code.

The Pthreads runtime in MCUDA includes support for static and dynamic methods of assigning computation to CPU threads. The static method distributes a contiguous set of blocks to each worker thread. Any thread is assigned at most one additional block compared to any other thread. Each thread then executes independently until completing its set. Under the dynamic method, each worker thread iteratively acquires and executes blocks until all blocks in the kernel have been issued. Each thread, when requesting a block to execute, atomically loads the current block index, represented by a global variable. If it is within the range specified by the kernel launch configuration parameters, it executes that block and increments the current block index to mark that the block is being processed. Otherwise, all blocks in the kernel have been claimed by some worker thread.

In both methods, when each worker thread completes processing, it enters the barrier at which the host thread is waiting. When all worker threads reach the barrier, the kernel execution has completed, and the host thread is allowed to leave the barrier and return to the host code.

In OpenCL, kernels are not invoked directly, but are encapsulated in objects which can be created with `clCreateKernel` with the kernel's function name passed as a string. That object is called through the OpenCL `clEnqueueNDRangeKernel` API function with the kernel name passed as a string. In that scenario, the OpenCL kernel program, instead of being compiled and linked directly with the host code, is compiled into a shared library object. When the kernel is created, its name is used in a dynamic symbol lookup to get the correct function handle, which is then invoked behind the `clEnqueueNDRangeKenel` interface.

A large body of work explores scheduling policies of parallel work units

in OpenMP and other frameworks [3, 30, 16, 7]. These results are just as applicable here as they are in any other similar system, and could potentially increase the performance by better managing load balance or synchronization costs from dynamic partitioning. Although the specific implementation described here uses the Pthreads approach, OpenMP compilers targeting Pthreads would have many similar components: a single thread executing the serial host code, and a group of threads sharing the work of the parallel device code with a CUDA thread block as the smallest work unit assigned. Initial experiments have suggested that for the applications tested, the overhead of parallel execution was negligible compared to the total program execution time.

CHAPTER 5

SERIALIZING SPMD PROGRAMS

The cornerstone of the implementation of CUDA for a CPU is reducing the number of hardware thread contexts required. At minimum, all threads within a single thread block must be simultaneously active, with all their state. Currently, there are roughly two orders of magnitude difference between the typical number of logical threads within a thread block and the typical number of hardware thread contexts available in a CPU. Implementing a static SPMD model on a hardware substrate with significantly fewer thread contexts than the SPMD application requests leads to many performance issues arising from scheduling and context switching. This chapter describes a structured, sequential implementation of a constrained static SPMD programming model that can be mapped to a single CPU thread context. The constraints are solely on the use of barrier synchronizations: namely that they fit the static-instance, textually aligned barrier model used in languages like CUDA and Titanium.

There are several challenges in effectively serializing an SPMD program with synchronization. First, without modifying the operating system or architecture, the compiler or runtime library must somehow manage the execution of logical threads explicitly. Second, the SIMD-like nature of the logical threads in many CUDA applications should be clearly exposed to the code generator. However, this goal is in conflict with supporting arbitrary control flow among logical threads. Finally, in a typical load-store architecture, private storage space for every thread requires extra instructions to move data in and out of the register file. Reducing this overhead requires identifying storage that can be safely reused for each thread.

The translation component of MCUDA that addresses these goals is com-

This chapter has been adapted from portions of a previously published work, ©2009 Springer, used with permission [51]. The original workshop paper was written in collaboration with Sam S. Stone and Wen-mei W. Hwu.

posed of three transformation stages: iterative wrapping, synchronization enforcement, and data buffering. For purposes of clarity, we consider only the case of a single kernel function with no function calls to other procedures, possibly through exhaustive inlining. This is always legal for the CUDA programming model, which does not allow recursion. It is possible to extend the framework to handle function calls with an interprocedural analysis, but this is left for future work. In addition, without loss of generality, assume that the code does not contain `goto` or `switch` statements, possibly through prior transformation [2]. All transformations presented are performed on the program's AST. To avoid confusion, keep in mind through this discussion that the final MCUDA implementation applies these transformations only to logical threads within a thread block, as explained in the following chapter.

5.1 Simple Serialization

The first step in the transformation simply serializes all the threads within the SPMD function. This changes the nature of the function from an SPMD specification to a sequential specification, temporarily ignoring any potential synchronization between logical threads. Figure 5.1 shows an example kernel function before and after this transformation. The loops enumerate the values of the previously implicit threadIdx variable and perform a logical thread's execution of the enclosed statements on each iteration. For the remainder of this document, we will refer to this introduced iterative structure as a *thread loop*. Local variables are reused on each iteration because only a single logical thread is active at any time. Shared variables still exist and persist across loop iterations, visible to all logical threads. The other implicit variables must be provided to the function at runtime, and are therefore added to the parameter list of the function.

By introducing a thread loop around a set of statements, we are making several explicit assumptions about that set of statements. The first is that the program allows each logical thread to execute those statements without any barrier synchronization between threads. Mutual exclusion synchronization, such as a lock or atomic operation, is provided implicitly by the serialization. The second is that there can be no side entries into or side exits out of the thread loop body. If the programmer has not specified any synchroniza-

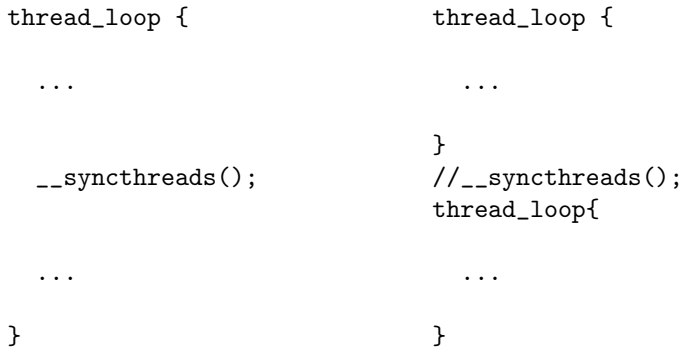
```
1 void cenergy(numatoms, gridspacing, energygrid[] )
2 {
3     int x = get_global_index(0);
4     int y = get_global_index(1);
5     int outIdx = get_global_size(0)*y + x;
6
7     float energy = 0.0f;
8     int atomid = 0;
9     while (atomid < numatoms) {
10            ...
11    }
12    energygrid[outIdx] = energy;
13 }
```

(a) OpenCL kernel

```
1 void cenergy(numatoms, gridspacing, energygrid[],
2                local_size, group_id, num_groups, work_dim)
3 {
4     // Thread Loop
5     for (int __y__ = 0; __y__ < local_size[0]; __y__++)
6         for (int __x__ = 0; __x__ < local_size[1]; __x__++)
7     {
8         int x = group_id[0]*local_size[0] + __x__;
9         int y = group_id[1]*local_size[1] + __y__;
10        int outIdx = num_groups[0]*local_size[0]*y + x;
11
12        float energy = 0.0f;
13        int atomid = 0;
14        while (atomid < numatoms) {
15                ...
16        }
17        energygrid[outIdx] = energy;
18    }
19    // end Thread Loop;
20 }
```

(b) Serialized OpenCL Kernel

Figure 5.1: Introducing a thread loop to serialize logical threads in a kernel computing Coulombic potential.



(a) Unenforced syncthreads

(b) Syncthreads enforced with loop fission

Figure 5.2: Simple synchronization enforcement with loop fission

tion point and the function contains no explicit return statement, no further transformation is required, as a function cannot have side entry points, and full inlining has removed all side-exits. In the more general case, where using a single thread loop is insufficient for maintaining program semantics, we must partition the function into sets of statements that do satisfy these properties.

5.2 Enforcing Synchronization

A thread loop implies a barrier synchronization among logical threads at its boundaries. Each logical thread executes to the end of the thread loop, and then "suspends" until every other logical thread (iteration) completes the same set of statements. For a simple synchronization within a serialized kernel function, we can enforce the synchronization by applying a loop fission (or splitting) operation, as shown in Figure 5.2. In the context of a serialized kernel function, a loop fission operation essentially partitions the statements of a thread loop into two sets of statements with an implicit barrier synchronization between them.

Although a loop fission operation applied to the thread loop enforces a barrier synchronization at that point, a fission operation cannot be applied when the barrier is within some other control structure. For example, consider the case of Figure 5.3. Splitting the thread loop exactly at the point of synchronization is not coherent with the bracket-nesting requirements of the C language. Therefore, a refinement of the loop fission operation is required

```
thread_loop {
  ...

  if(condition) {

    ...

    __syncthreads();


    ...

  }

  ...
}
```

(a) Unenforced syncthreads

```
thread_loop {
  ...

  if(condition) {

    ...
  }
  //  __syncthreads();

thread_loop {
    ...

  }

  ...
}
```

(b) Syncthreads incorrectly enforced

```
thread_loop {
    ...
}
  if(condition) {
    thread_loop{
      ...
    }
  //  __syncthreads();

    thread_loop {
      ...
    }
  }
thread_loop {
    ...
}
```

(c) Syncthreads correctly enforced by multiple thread loops

Figure 5.3: A case where simple loop fission fails to correctly enforce synchronization within control flow, and an example of how multiple thread loops could correctly enforce it

to address the general case. Essentially, all control statements or structures on which a barrier is control-dependent must be moved outside of thread loops for correct program functioning.

5.2.1 Deep fission

In describing the algorithm for enforcing synchronization points, I first assume that all control structures have no side effects in their evaluation or header. This is because the side effects could potentially modify thread-private variables, and thus must be enclosed in a thread loop. However, deep fission could move these conditionals out of any thread loop, necessitating some transformations prior to deep fission. In particular, `for` loops must be transformed into `while` loops in the AST, moving the initialization and update expressions to appropriate statements. In addition, all conditional evaluations with side effects must be removed from the control structure's declaration and assigned to a temporary variable, which then replaces the original condition in the control structure.

Then, for each synchronization statement S, we apply Algorithm 1 to

Input: Function F in AST representation
Input: Deep Fission Target Statement S
Output: Function F with Thread Loops Fissed Around S
while *Scope C containing S is not a thread loop* **do**
    Create new thread loop L;
    Make all children of C preceding S children of L;
    Add L as child of C preceding S;
    Create new thread loop $L2$;
    Make all children of C following S children of $L2$;
    Add $L2$ as child of C following S;
    if *C is an if-else construct* **then**
        Create new thread loop $L3$;
        Make children of C in the branch not containing S children of $L3$;
        Add $L3$ as child of C in place of the statements it now contains;
    end
    $S = C$;
end
apply loop fission to C around S;
Return F;

Algorithm 1: Deep fission around a synchronization statement S

Figure 5.4: Applying deep fission to enforce synchronization

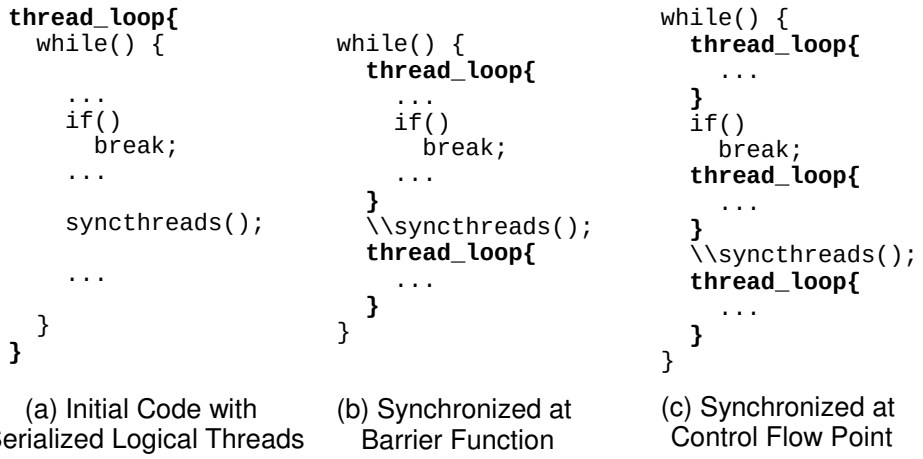


Figure 5.5: Addressing unstructured control flow. The break statement is treated as an additional synchronization statement for correctness.

the AST with S as the input parameter. Figure 5.4 shows the steps of the algorithm as they are applied to a matrix multiplication kernel. Figure 5.4(b) shows the preprocessing step of transforming the `for` loop into a `while` loop, with initialization and update statements adjusted appropriately. The thread loop cannot be split at the barrier currently, because the barrier is contained within the `while` loop. Therefore, the internal statements of the `while` loop are partitioned into two new thread loops straddling the original local of the barrier. Finally, the `while` loop structure as a whole becomes the statement around which the outermost thread loop must be split, accomplished with a simple loop fission operation around the `while` loop.

After this algorithm has been applied with each of the programmer-specified synchronization points as input, the code may still have some control flow for which the algorithm has not properly accounted. Recall that thread loops assume that there are no side entries or side exits within the thread loop body. Control flow statements such as `continue`, `break`, or `return` may not be handled correctly when the target of the control flow is not also within the thread loop. Figure 5.5(b) shows a case where irregular control flow would result in incorrect execution. In some iterations of the outer loop, all logical threads may avoid the break and synchronize correctly. In another iteration, all logical threads may take the break, avoiding synchronization. However, in the second case, control flow would leave the first thread loop before all logical threads had finished the first thread loop, inconsistent with the program's specification. This is because the structure of the code resulting from

the deep fission operation changed the target of the break from the statement immediately after the `while` to the statement immediately after the thread loop. Again, because the synchronization point is control-dependent on the execution of the break statement, the break statement itself must be reached or not reached uniformly by all threads on each particular `while`-loop iteration. In essence, these are the same restrictions of a barrier, allowing the compiler to essentially treat the break itself as another "synchronization" point.

Essentially, this is just a further extension of the previously stated principle that all control statements on which a true barrier is control-dependent must be moved outside of thread loops. Therefore, the compiler must pass through the AST at least once more to identify these violating control flow statements. Upon the identification of a control flow statement S whose target is outside its containing thread loop, Algorithm 1 is once again applied, treating S as a synchronization statement, which will move statement S outside of any thread loop. For the example of Figure 5.5, this results in the code shown in Figure 5.5(c). Because these transformations more finely divide thread loops, they could reveal additional control flow structures that violate the thread loop properties. Therefore, this irregular control flow identification and synchronization step is applied iteratively until no additional violating control flow is identified.

The key insight is to not support arbitrary control flow among logical threads, but leverage the restrictions in the language to define a sequential ordering of the instructions of multiple threads that satisfies the partial ordering enforced by the synchronization points. This "over-synchronizing" allows a complete implementation of "threaded" control flow using only iterative constructs within the code itself. The explicit synchronization primitives may now be removed from the code, as they are guaranteed to be bounded by thread loops on either side, and contain no other computation. Because only barrier synchronization primitives are provided in the CUDA programming model, no further control-flow transformations to the kernel function are needed to ensure proper ordering of logical threads. Figure 5.6(a) shows the matrix multiplication kernel after this hierarchical synchronization procedure has been applied.

```
__kernel void
small_mm_list(float* A_list,
   float* B_list, const int size)
{
  float sum;
  int mat_start, col, row, out_idx, i;
  mat_start = group_id(0) * size * size;
  col = mat_start + tid(0);
  row = mat_start + (tid(1) * size);

  sum = 0.0;

  for(i = 0; i < size; i++)
    sum += A_list[row + i] *
               B_list[col + (i*size)];

  //  Barrier before overwriting input
  __syncthreads();

  out_idx = matrix_start + tid(0) +
      (tid(1) * size);
  A_list[out_idx] = sum;
}
```

(a) OpenCL kernel

```
__kernel void small_mm_list(float* A_list,
    float* B_list, const int size)
{
  float sum[];
  int mat_start, col[], row[],
        out_idx[], i[];
  for( each tid ) {

    mat_start = group_id(0) * size * size;
    col[tid] = mat_start + tid(0);
    row[tid] = mat_start + (tid(0) * size);

    sum[tid] = 0.0;

    for(i[tid] = 0; i[tid] < size; i[tid]++)
      sum[tid] +=
          A_list[row[tid] + i[tid]] *
            B_list[col[tid] + (i[tid]*size)];
  }

  for( each tid ) {
    out_idx[tid] = mat_start +
        (tid(1) * size) + tid(0);
    A_list[out_idx[tid]] = sum[tid];
  }
}
```

(b) Universal Private Variable Replication

```
__kernel void
small_mm_list(float* A_list,
    float* B_list, const int size)
{
  float sum[];
  int matrix_start, col[], row[], out_index, i;

  matrix_start = blockIdx.x * size * size;
  for(tid.x = 0; tid.x < blockDim.x; tid.x++) {
    col[tid] = matrix_start + tid.x;

    for(tid.y = 0; tid.y < blockDim.y; tid.y++) {
      row[tid] = matrix_start + (tid.y * size);
      sum[tid] = 0.0;

      for(i = 0; i < size; i++)
        sum[tid] += A_list[row[tid] + i] *
                      B_list[col[tid] + (i*size)];
    }
  }

  for(tid.x = 0; tid.x < blockDim.x; tid.x++)
    for(tid.y = 0; tid.y < blockDim.y; tid.y++) {
      out_index = matrix_start +
                          (tid.y * size) + tid.x;
      A_list[out_index] = sum[tid];
  }
}
```

(c) Selective Scalar Expansion and Loop Invariant Code Motion

Figure 5.6: Data replication in an example kernel multiplying many small matrices. Some array sizes omitted and ID-queries abbreviated.

5.3 Replicating Thread-Local Data

Once the control flow has been restructured, the final task remaining is to buffer the declared variables as needed. Each logical thread should have a local store for variables, independent of all other logical threads. Because these logical threads no longer exist in separate thread contexts, the translated program must emulate private storage for logical threads. The simplest implementation creates private storage for each thread's instance of the variable, analogous to scalar expansion [21]. This technique, which I will refer to as *universal replication*, fully emulates the local store of each logical thread by creating an array of values for each local variable, as shown in Figure 5.6(b). Statements within thread loops access these arrays by thread index to emulate the logical thread's local store.

However, universal replication is often unnecessary and inefficient. In functions with no synchronization, thread loops can completely serialize the execution of logical threads, reusing the same memory locations for local variables. Even in the presence of synchronization, some local variables may have live ranges completely contained within a thread loop. In this case, logical threads can still reuse the storage locations of those variables because a value of that variable is never referenced outside the thread loop in which it is defined. For example, in the case of Figure 5.6(b), the local variable `k` can be safely reused because the live range of its value begins and ends within one iteration of the third thread loop.

Therefore, to use less memory space, the source-to-source compiler should only create arrays for local variables when necessary. A live-variable analysis determines which variables have a live value at the end of a thread loop, and creates arrays for those values only. This technique, called *selective replication*, results in the code shown in Figure 5.6(c), which allows all logical threads to use the same memory location for the local variable `k`. However, `a` and `b` are defined and used across thread loop boundaries, and must be stored into arrays.

References to a variable outside of the context of a thread loop can only exist in the conditional evaluations of control flow structures. Control structures must affect synchronization points to be outside a thread loop, and therefore must be uniform across the logical threads. Because all logical threads should have the same logical value for conditional evaluation, we sim-

ply reference element zero as a representative, as exemplified by the `while` loop in Figure 5.6(b-c).

5.4 Comparison with Industry Implementations

Much previous work has addressed the challenge of implementing OpenCL on x86 processors, both published academically and implemented industrially. Here, we cover those related works most directly related to our methodology and those that are most popularly used today. We will study each implementation as it relates to the running example in Figure 2.7.

The AMD CPU OpenCL language implementation is based on the Twin Peaks technology [13]. The primary insight of the implementation is that modern, multicore, superscalar, x86 CPUs support a relatively low level of thread-level parallelism, but a very high degree of instruction-level parallelism. Therefore, it makes most sense to combine all of the work-items in a group into a single CPU thread. The AMD CPU stack accomplishes this with user-level threading techniques, using irregular control flow to "simulate" multiple parallel work-items with a single user thread.

Figure 5.7 shows a pseudocode example of how this user-level threading is accomplished. First, the implementation declares a data structure suitable for holding all the data private to a single work-item, and then initializes a collection of such data structures to hold the state of all work-items in the group (details not shown). The CPU thread calls the `MatMul` function with a particular work-group index, which the compiler has modified such that it will complete the execution of all work-items in the specified work-group. It initializes the local state of the work-group on line 23, and selects the work-item with index 0 to be the first *active* work-item. At any given time, the active work-item is the one being advanced through the program. To support multiple work-items with the same kernel code, a level of indirection is added, with `active_wi` pointing to the private data of the active work-item.

The program execution follows the original OpenCL kernel's operations, referring to the active work-item's private storage through `active_wi` for references to private variables. Execution of the first work-item continues until the barrier statement on line 34. At the barrier, the framework saves the program location where the current work-item should be restarted, in this

```
1 struct wi_state {
2   int local_id[3], group_id[3], global_id[3];
3   float result;
4   float *A_line;
5   int i;
6   int ii;
7   void *restart_point;
8 }
9
10 struct wi_state group_state[WORKGROUP_SIZE];
11 struct wi_state *active_wi;
12
13 void barrier(int fence, void* restart) {
14   (active_wi++)->restart_point = restart;
15   if (active_wi = group_state + WORKGROUP_SIZE)
16     active_wi = group_state;
17   goto active_wi->restart_point;
18 }
19
20 void MatMul(float *A, float *B, float *C,
21             int g_id[3], g_size[3]) {
22   float A_tile[TILE_WIDTH];
23   setup_wi_contexts(group_state);
24   active_wi = group_state;
25 kernel_start:
26   active_wi->result = 0.0f;
27   active_wi->A_line = A + active_wi->group_id[1]*A_WIDTH;
28
29   for (active_wi->i = 0; active_wi->i < A_WIDTH;
30        active_wi->i+= TILE_WIDTH) {
31       A_tile[active_wi->local_id[0]] =
32           A_line[active_wi->i + active_wi->local_id[0]];
33
34     barrier(CLK_LOCAL_MEM_FENCE, &&restart_0);
35     restart_0:
36       for (active_wi->ii = 0;
37            active_wi->ii < TILE_WIDTH; active_wi->ii++)
38         active_wi->result +=
39             A_tile[active_wi->ii] *
40                 B[(active_wi->i+active_wi->ii)*B_WIDTH +
41                     active_wi->global_id[0]];
42     barrier(CLK_LOCAL_MEM_FENCE, &&restart_1);
43     restart_1:
44   }
45   C[C_WIDTH*active_wi->group_id[1] + active_wi->global_id[0]] =
46       active_wi->result;
47   barrier(0, &&kernel_finish);
48 }
```

Figure 5.7: C-like pseudocode representing AMD's OpenCL implementation

case the label `restart_0` on line 35. It then updates the `active_wi` pointer to the next work-item's private state, and performs an indirect jump to the location at which the newly activated work-item should be restarted. At initialization, all work-items have their restart points set to the `kernel_start` label, so in this case, the indirect jump takes the second work-item to the beginning of the program, as it should. The second work-item will then follow the same program path to the same barrier, at which point the framework will make the third work-item the active work-item, and so on, until all work-items in the group have reached the first barrier.

When the last work-item executes the first barrier, the condition on line 15 will evaluate to true for the first time, and restart the first work-item at `restart_0`, allowing it to continue execution where it left off. It will do so until the second barrier at line 42, where it will again save its current restart point and switch to the second work-item. The constant switching of active work-items continues until work-items begin to reach the kernel's end. At that point, the work-items save their restart point as some sentinel value that will lead the framework to the cleanup code to finish the current work-group, and prepare to execute another work-group if available.

The Twin Peaks methodology has several aspects that make it ill-suited to support the programming practices outlined in Chapter 2. First, the overhead of changing the active work-item is significant. The example shown uses illegal label-passing to illustrate the concepts, but the real implementation is based on `setjmp` and `longjmp`. Even after significant optimization of those low-level routines for this context, the Twin Peaks authors claim an overhead of 10ns or 30 clock cycles per work-item change. Additionally, the micro-threading approach makes no effort to capture vector-level parallelism across work-items. Each work-item is executed in isolation, and any vectorization is limited to opportunities within the code of a single work-item.

Finally, the Twin Peaks implementation does not capture spatial locality as expected by the developer. Figure 5.8 shows a graphical representation of a single work-group's accesses to the input matrix B over the course of one tile. In a GPU implementation, with wide SIMD vectors and round-robin scheduling, large collections of contiguous addresses are accessed and consumed together. However, a serialization of work-items with the Twin Peaks methodology effectively executes all of a single work-item's accesses first, before the accesses of any other work-items. But the kernel follows the

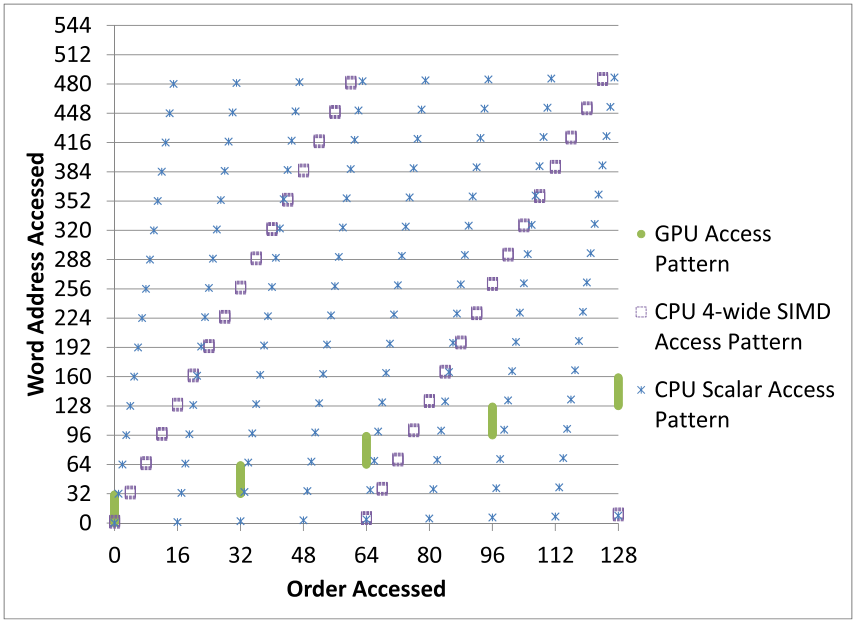


Figure 5.8: Access patterns to the matrix B in our example OpenCL program with various implementations. Cache lines are assumed to be 32 words wide, with boundaries marked by the major Y axis lines.

```
1 void MatMul( float *A, float *B, float *C,
2        int g_id[3], int g_size[3]) { // work-group ID and size
3   simd_float result[WORKGROUP_SIZE/SIMD_W];
4   float A_tile[TILE_WIDTH];
5   float *A_line = A + g_id[1]*A_WIDTH;
6   for (__x__ = 0; __x__ < g_size[0]; __x__+=SIMD_W) {
7     simd_store(result[__x__], simd_expand(0.0f));
8   }
9   for (int i = 0; i < A_WIDTH;
10       i+= TILE_WIDTH) {
11    for (__x__ = 0; __x__ < g_size[0]; __x__+=SIMD_W) {
12        simd_store(&A_tile[__x__] , simd_load(&A_line[i + __x__]));
13    }
14    //barrier(CLK_LOCAL_MEM_FENCE);
15    for (__x__ = 0; __x__ < g_size[0]; __x__+=SIMD_W) {
16      for (int ii = 0;
17           ii < TILE_WIDTH; ii++)
18        simd_accumulate(&result[__x__] , A_tile[ii] *
19            simd_load(&B[(i+ii)*B_WIDTH +
20                         g_size[0]*g_id[0]+__x__]));
21    }
22    //barrier(CLK_LOCAL_MEM_FENCE);
23  }
24  for (__x__ = 0; __x__ < g_size[0]; __x__+=SIMD_W) {
25    simd_store(&C[C_WIDTH*g_id[1] + g_size[0]*g_id[0]+__x__],
26        result[__x__]);
27  }
28 }
```

Figure 5.9: C-like pseudocode representing Intel's vectorizing OpenCL implementation

guidelines to support SIMD across work-items, leading to interleaved accesses among work-items but strided accesses in the access stream of a single work-item. Figure 5.8 shows how the serialized implementation accesses a wide range of addresses in a short amount of time for the first work-item, followed by another set of strided accesses from the second work-item, and so on. If the tile size or the memory footprint of the work-group's total state gets large enough, this kind of access pattern will cause significant cache thrashing, and result in very poor spatial locality usage.

Intel's implementation of OpenCL for x86 is both the most recent and the least explicitly disclosed or studied. Our best understanding is that the Intel implementation would behave somewhat like the pseudocode in Figure 5.9. The figure assumes that the implementation uses region-based serialization for simplicity, but this is not necessarily clear. What is more clear, and noteworthy, is the implementation's focus on explicitly combining multiple work-items into vectorized execution bundles. Instead of creating private, scalar data elements for every work-item, it will create vector data elements for each SIMD bundle, as the declaration of the variable `result` on line 3

shows. All serialization loops are effectively unrolled by a factor of `SIMD_W`, the width of the SIMD units, with each iteration performing operations on vector values.

In practice, for recent CPUs, Intel's methodology works very well compared to the other techniques already described. It does map multiple work-items to the SIMD units of the architecture, mirroring the expected behavior as described in Chapter 2. The barrier overhead of the implementation is not clear from the disclosed materials, but experimentally seems to be somewhere between the region-based methods and the Twin Peaks method. The explicit combining of work-items into SIMD units does assist in the capturing of spatial locality, but still does not use the caches as effectively as they could be used. The CPU 4-wide SIMD Access Pattern in Figure 5.8 shows why. For GPUs, the effective SIMD width of the processor is very wide, and the cache line size is closely matched to the SIMD data vector width for 32-bit words. In CPUs, while the SIMD widths have increased recently, the cache lines are still significantly larger than the SIMD data vector width. Therefore, a single SIMD access will utilize a smaller portion of the cache line by itself. In a kernel written according to the OpenCL programming guidelines, other work-items in adjacent SIMD bundles would be consuming the rest of that data. However, the overall control flow of the compute region on lines 15-20 of Figure 5.9 still executes all of the accesses for one SIMD group before any accesses from the next SIMD group. The final result is an access pattern that looks like the *** points of Figure 5.8, somewhere between the completely serialized and completely vectorized access patterns.

For comparison, Figure 5.10 shows pseudocode for the result of the region-based serialization methodology of this section. Some private variables such as `result` are expanded into an array of values, with one element for each work-item. However, analysis detected cases where private variables always store values uniform across the entire work-group, such as the variable `A_line`, and avoid creating separate memory locations to store redundant information.

Instead of adding functionality to the barrier function, the compiler uses the very presence of the barrier function to inform analysis of the kernel code. According to the previously described algorithms, the kernel code is split up into contiguous regions that contain no barriers. Each region is then serialized with an inserted counted loop over the work-item indexes.

```
1 void MatMul( float *A, float *B, float *C,
2        int g_id[3], int g_size[3]) { // work-group ID and size
3   float result[WORKGROUP_SIZE];
4   float A_tile[TILE_WIDTH];
5   float *A_line = A + g_id[1]*A_WIDTH;
6   for (__x__ = 0; __x__ < g_size[0]; ++__x__) {
7     result[__x__] = 0.0f;
8   }
9   for (int i = 0; i < A_WIDTH; i+= TILE_WIDTH) {
10    for (__x__ = 0; __x__ < g_size[0]; ++__x__) {
11      A_tile[__x__] = A_line[i + __x__];
12    }
13    //barrier(CLK_LOCAL_MEM_FENCE);
14    for (__x__ = 0; __x__ < g_size[0]; ++__x__) {
15      for (int ii = 0; ii < TILE_WIDTH; ii++)
16        result[__x__] += A_tile[ii] *
17          B[(i+ii)*B_WIDTH + g_size[0]*g_id[0]+__x__];
18    }
19    //barrier(CLK_LOCAL_MEM_FENCE);
20  }
21  for (__x__ = 0; __x__ < g_size[0]; ++__x__) {
22    C[C_WIDTH*g_id[1] + g_size[0]*g_id[0]+__x__] = result[__x__];
23  }
24 }
```

Figure 5.10: C-like pseudocode representing region-based loop serialization

In Figure 5.10, one region occupies lines 6-8, initializing the private variable `result` for all work-items. A second region on lines 10-12 copies a tile of data from global to local memory. The main computational region on lines 14-18 consumes the copied tile, accumulating inner products for each column of B. The final region on lines 21-23 copies the final results to the correct region of the output space. The regions themselves constitute nodes in a dynamic control flow graph independent of work-item index, with each dynamic region executed for all work-items. In the example kernel, there is a loop over the second and third regions, executing both for each tile of the input data, while the first and last regions are executed only once each.

The inserted serialization loops themselves then maintain the semantics of the original barriers, not letting any operations following the barrier in the dynamic execution completing before any operation before that barrier. Therefore, the barrier itself can be removed from the final code, as it adds no information or constraint not already represented by the serialized code.

From a portability standpoint, the region-based serialization methodology has several advantages over the Twin Peaks technique. First, the overhead of executing a barrier is significantly reduced. In this methodology, a barrier only adds a cost of a loop branch and loop counter increment in the worst case. In practice, the overhead is even smaller, because optimizing compilers

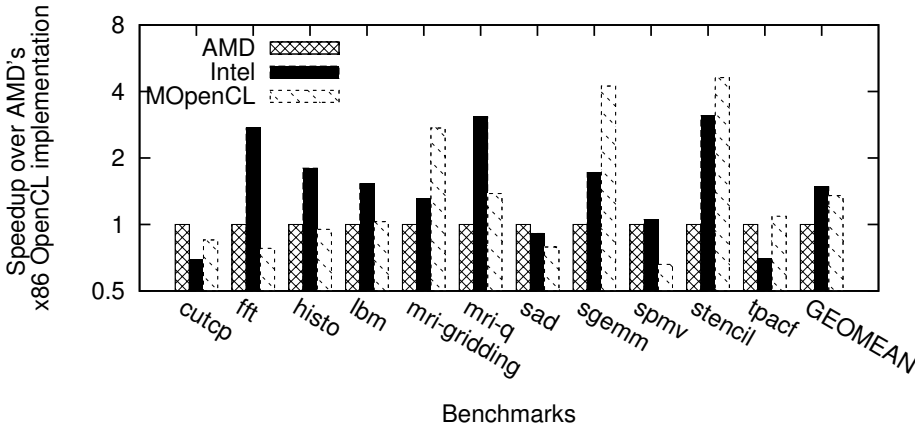


Figure 5.11: Evaluating the performance of region-based serialization compared to the Intel and AMD OpenCL stacks

apply optimizing transformations such as loop unrolling to the serialization loops. Such optimizing loop transformations are practically prohibited by the indirect jumps of the Twin Peaks methodology. Second, this implementation could indirectly result in SIMD vectorization across work-items, if the inserted serialization loops happen to be innermost loops, and a vectorizing compiler is able to conservatively prove the vectorizability of those loops. And finally, the implementation does not fundamentally solve the spatial locality expectation mismatch, as the access patterns remain largely unchanged. The CPU scalar access pattern in Figure 5.8 still accurately describes the serialized access pattern of the main computation region: strided accesses along a column of the B matrix, followed by more strided accesses along subsequent columns.

5.5 Performance Analysis

Figure 5.11 shows the performance results of the region-based serialization methodology for the OpenCL versions of the Parboil benchmarks optimized for an NVIDIA GPU. The results vary quite a lot reflecting the drastically different implementation approaches, but show that on average, the Intel OpenCL stack outperforms both the AMD OpenCL stack and the region-based serialization.

The `cutcp` and `sad` benchmarks suit the AMD implementation well, because they conveniently use short-vector types which the AMD implemen-

tation can directly convert into SIMD instructions, but somewhat complex control flow preventing the Intel and region-based serialization methods from extracting quite so much SIMD utilization. Kernels with regular control flow and little or no barrier synchronization favor the Intel implementation methodology, such as `histo`, `mri-q`, `spmv`, and `lbm`. But those kernels that make extensive use of shared memory and barrier synchronization, such as `tpacf`, `stencil`, `mri-gridding`, and `sgemm`, favor the region-based serialization method because of the significantly reduced overhead of handling those dynamic barriers.

Overall, there is no clear, consistent best implementation among these three. Those kernels which rely most heavily on low-overhead barriers favor region-based serialization, while those favoring implicit SIMD execution of multiple work-items favor Intel's implementation, and those that fortuitously fall into the pattern that the AMD implementation can vectorize favor it. This is one reason why performance portability seems elusive today: the CPU implementations themselves have drastic performance differences between them. However, we also have to recognize that such differences are not unheard of simply among various compilers for a single-core codebase as well, depending on the strengths and methodologies of each compiler.

CHAPTER 6

A VECTOR MACHINE MODEL OF ACCELERATED KERNEL EXECUTION

The previous implementation methodologies show some common performance insights and common portability oversights. It is clear that the work-items in a single work-group should be combined into a single, sequential CPU thread. Work-items within a group are primarily a source of vector- and instruction-level parallelism, both of which CPU architectures exploit from within a single CPU thread. The CPU implementations vary widely in their approach to serializing work-items and capturing SIMD parallelism from the work-items, with the Twin Peaks method vectorizing only explicit vector operations within a work-item, region serialization relying on autovectorization technology, and Intel's methodology directly targeting SIMD instructions. And finally, no current CPU implementation does an excellent job of handling spatial locality given the most common OpenCL programming practices. Instead, they each result in some kind of strided access pattern by executing one or more work-items as long as possible instead of interleaving the accesses of the work-items that would consume the elements of a particular cache line.

We propose a vector-based serialization of the work-group. Even though the physical SIMD width of a machine is of a fixed and limited value, the programming model's usages would benefit from executing work-groups in a way that emulates a work-group-wide vector machine. Instead of advancing only a small number of work items until they are forced to yield, an implementation could execute each dynamic statement for all work-items in the group before moving on to the next statement. The C Extensions for Array Notation (CEAN) programming model provides an excellent mechanism for describing just such execution semantics.

Array Declaration	`int array[ARRAY_SIZE]`
Full array slice	`array[:]`
Bounded array slice	`array[100:100]`
Indirect gather or scatter	`array[indexes[:]]`

Figure 6.1: CEAN array slice notation examples

6.1 C Extensions for Array Notation

Intel introduced CEAN as part of their production compiler in 2010. It has also been implemented in gcc, although not integrated into the trunk, and proposed to the C++ standards committee as an industry-standard extension of C and C++. It is very similar to, and likely inspired by, FORTRAN-style array operations. The basic syntax is shown in Figure 6.1. An *array slice expression* is an array subscript expression (C99 6.5.2.1) that uses an *array slice operator*. The two most relevant array slice operator types are the *full slice* and *bounded slice* operators. A full slice operator, syntactically expressed with a single semicolon as the subscript expression, can only be used on arrays with a known size, and evaluates to the entire contents of the array. A bounded slice operator can be used on any array or pointer, and is a subscript expression of the form:

array_ptr [*base_index* `semicolon` *extent*].

The base index determines the offset of the first element of the slice, and the extent value determines the number of contiguous elements that should be extracted in the slice. The example bounded array slice in Figure 6.1 accesses a 100-element slice from the array, beginning with index 100 and ranging to index 199. A bounded slice will always result in an array value with a number of elements equal to the extent. Multi-dimensional slices are permitted, but we will restrict ourselves to single-dimensional array operations for this chapter. Array expressions can also be used as array subscript expressions into other arrays, which is useful for defining indirect gather and scatter accesses. Indirect accesses tend to be significantly slower than full or bounded accesses in practice.

Operations on multiple array expressions must operate per-element across the extent of all involved array expressions. For instance, adding a scalar to an array expression will result in a new array expression with the scalar addition applied to every element of the array. Adding a pair of array ex-

```
1 void MatMul( float *A, float *B, float *C,
2        int g_id[3], int g_size[3]) { // work-group ID and size
3   float result[WORKGROUP_SIZE];
4   float A_tile[TILE_WIDTH];
5   float *A_line = A + g_id[1]*A_WIDTH;
6
7   result[:] = 0.0f;
8   for (int i = 0; i < A_WIDTH; i+= TILE_WIDTH) {
9       A_tile[0:g_size[0]] = A_line[i:g_size(0)];
10    //barrier(CLK_LOCAL_MEM_FENCE);
11      for (int ii = 0; ii < TILE_WIDTH; ii++)
12        result[:] += A_tile[ii] *
13          B[(i+ii)*B_WIDTH + g_id[0]*g_size[0]:g_size[0]];
14    //barrier(CLK_LOCAL_MEM_FENCE);
15  }
16  C[C_WIDTH*g_id[1] + g_id[0]*g_size[0]:g_size[0]] = result[:];
17 }
```

Figure 6.2: CEAN-based result of our proposed OpenCL implementation

pressions means an element-wise addition, and requires that the two array expressions have the same number of elements.

6.2 Implementing OpenCL with CEAN

Figure 6.2 shows how we can apply CEAN-style transformations similar to the way previous work applies loop-based serialization. As with previous work, we expand the `result` private variable into an array, because its value depends on work-item index. However, instead of introducing loops over the kernel code, we simply replace the scalar expressions in the code with array slices where appropriate. Accesses to the `result` local variable on lines 7, 12, and 16 use a full slice expression over the array. Accesses to the global memory could use the indirect array access expression syntax in the general case. However, in the example code, all global memory accesses are provably coalesced across the entire work-group. They can therefore be converted into the faster bounded slice operations by decomposing the index operation into the form *base_index + get_local_id(0)*. Once the base index expression has been identified, the compiler can generate a bounded array slice beginning at that base index and with an extent equal to the work-group size. The global memory access transformation is applied to `A_line` global memory pointer access on line 9, with `i` as the base index. Accesses to matrix `B` on line 13 and matrix `C` on line 16 are also converted into array slice accesses by first applying the following equivalence: `get_global_id(0)`

`== get_group_id(0)*get_group_size[0] + get_local_id(0)`. The result of all these transformations is a program that expresses the execution of the work-group as a sequence of vector operations over local variables.

CEAN has two important properties that make it very well suited to describing OpenCL work-group execution. First, array expression operations were specifically introduced to support SIMD execution on CPUs. Operations over array expressions are explicitly independent across all elements, and therefore directly targeted as vectorization opportunities. Second, the execution semantics are such that each statement using array slice expressions is evaluated in its entirety before the next statement executes, just as it would if all operations were only scalar. This creates the kind of access pattern that actually achieves the spatial locality the developer intended, matching the GPU memory access pattern highlighted in Figure 5.8 in Chapter 5. And like in the prior region-based serialization approach, barriers are rendered irrelevant in the final code. The array slice ordering constraints essentially provide the same ordering as if there were a barrier after every statement.

Note that this choice of scheduling has strong implications for the local layout of data within the work-group. The Twin Peaks authors specifically defend their choice of storing all the private data for a single work-item contiguously in memory in a data structure. Their claim is that such a layout will get the best spatial locality [13] (although they admit that more research is needed on the topic). This makes sense given their execute-until-yield serialization model. If one work-item is going to be executed for a long time, it makes most sense that all its private data would be close together, and not interleaved with the data from other work-items. However, it makes vectorization across work-items inefficient. In order to efficiently combine multiple work-items into a SIMD bundle, all instances of the local variables for work items in that bundle should be contiguously stored.

Whether due to shortcomings on the compiler analysis and transformation capability or restrictions on how CEAN can be used, there are cases where CEAN-based translation of a particular piece of code is not possible. One case is loops where the loop iteration count cannot be determined to be a work-item invariant value, because Intel's compiler (as of version 13.1) does not permit CEAN notation in the condition checks of loops. To get the fullest benefit of CEAN possible, when our compiler intends to target CEAN it will still perform region formation as normal, so that the choice to use CEAN

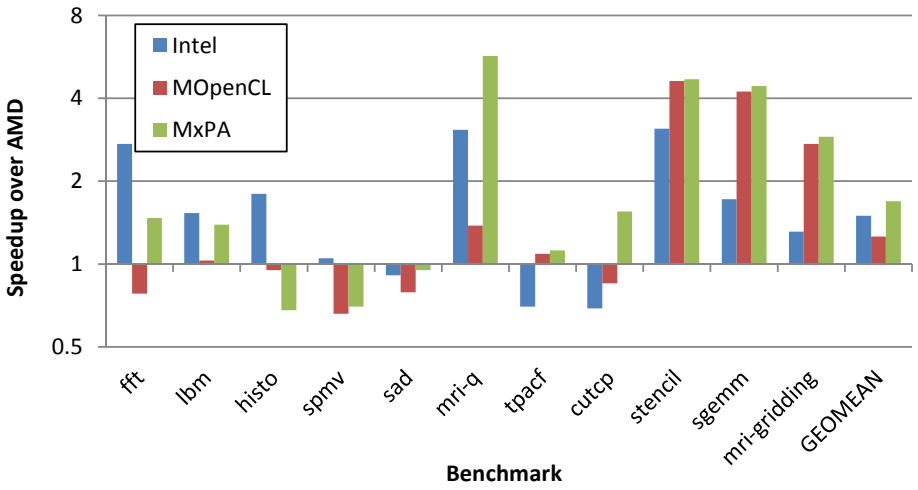
notation or region-based loop serialization can be made on a per-region basis. This allows vector-serializable regions to utilize CEAN notation, without loss of generality.

6.3 Performance Analysis

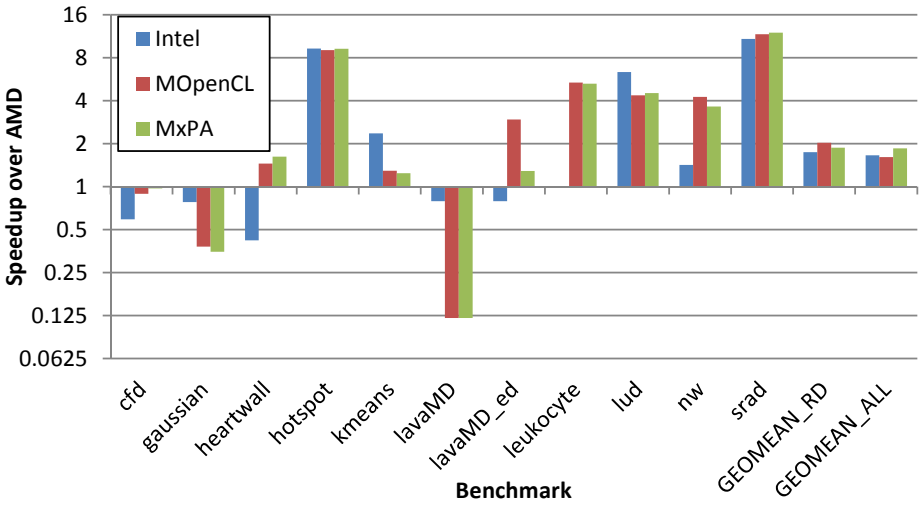
Figure 6.3 shows the performance results of the vector-based serialization methodology as embodied in the MxPA product, compared with the AMD and Intel OpenCL implementations as well as the region-based serialization methodology, labeled "MOpenCL" in the graph. Again, the results vary quite a lot reflecting the drastically different implementation approaches, but show that on average, the vector-based serialization outperforms all previous work by a significant margin for the workload represented by these benchmarks.

In most cases where the region-based serialization outperformed AMD and Intel, the vector-based serialization performed even more strongly, including every case from the Parboil benchmarks. This shows that the methodology is equally efficient at delivering low-overhead barrier semantics as the region-based serialization was, but that the more explicit vectorization notation was making additional performance contributions. When those contributions are small, it is because the compiler was already able to autovectorize the original region-serializing loops for those benchmarks. In the `mri-q` case, the region-based serialization was not able to perform adequate vectorization, whereas the vector-based serialization enabled it. It outperformed even Intel's vectorizing implementation because of better locality management that only streamed through the shared input data set once per work-group, instead of one per SIMD-bundle. In some cases, the compiler was unable to make very effective use of CEAN, such as for the thread-dependent innermost loops in `spmv` and `sad`, and the gain from targeting other regions was minimal.

Some limitations of the current implementations of MxPA and MOpenCL are revealed by the `lavaMD` Rodinia benchmark, where the programmers wrote extra control flow code to deal with the fact that they were creating work-groups larger than their chosen tile size. The result labeled `lavaMD_ed` is excluded from the GEOMEAN calculations, but shows the performance achievable simply by changing the work-group size to match the tile size, and



(a) Parboil benchmarks



(b) Rodinia benchmarks

Figure 6.3: Evaluating the performance of vector-based serialization (MxPA) compared to the Intel and AMD OpenCL stacks, as well as region-based serialization (MOpenCL).

deleting all the control flow made irrelevant by that change.

The two cases where vector-based serialization is significantly outperformed by region-based serialization are `histo` and `lavaMD_ed`. In `histo`, most memory accesses are either perfectly regular, in-place updates to each data element, or the actual scattered histogram accesses. Forcing SIMD execution proved to be detrimental, because the ISA targeted does not support scatter accesses directly from the vector registers. The `lavaMD_ed` result shows a limitation of the C compiler used for the CEAN output of MxPA, where opportunities to avoid storing temporary array slices on the program stack were not exploited.

In other cases, vector-based serialization did improve performance over region-based serialization, but was not able to surpass the performance of the best industry OpenCL implementation for that benchmark. `sad` remained too irregular for vectorization across multiple work-items, such that AMD's direct vectorization of short vector types was able to get the best usage of the CPU hardware execution units. As previously noted, the innermost loop of `spmv` was not suitable for CEAN transformation, and the region-based serialization was outperformed by both industry platforms. `lbm` has no reuse of data among its work-items and no barriers, and a group working set somewhat larger than the L1 cache size, so the policy of executing each SIMD bundle to completion gave Intel's platform a slight advantage.

Although there is no clear, consistent best implementation among these four, the average performance clearly favors the vector-based serialization, even with its current limitations. In successful cases, it combines the benefit for low-overhead barriers with the throughput of vector execution, and more closely matches the spatial locality profile expected by a programmer trained according to GPU performance guidelines.

CHAPTER 7

PORTABILITY

The goal of this dissertation is to demonstrate the feasibility of performance portability. From the outset, we must admit that because a software developer could have arbitrarily high standards for performance portability, we can never satisfy everyone. Some software developers, particularly high-performance library developers, will go to great lengths to tune their library for a specific architecture, with portability achieved for the end-user through the many platform-specific implementations underneath the library interface. On the other extreme, some software developers care only for functional portability, and consider any software development effort specifically dedicated to performance to be misplaced. But many application developers choosing accelerated kernel languages, and OpenCL in particular, fall somewhere in the middle. They are specifically choosing a language and programming model believed to increase software development costs because they hope to get performance gains from using it. At the same time, the presence of a particular kind of accelerator is not guaranteed, and most software developers would prefer to have as wide a set of system targets as possible from the single codebase. For these developers, they will not accept performance they see as being "bad," but may be happy with leaving some performance on the table on some architectures if it means that they can be more productive writing code with "good enough" performance overall.

For such performance-minded programmers, it is not enough to simply show that one language implementation gets X% better than some other implementation of that language. It is not even helpful to know whether a particular piece of code gets Y% of the system's peak computational throughput or memory bandwidth, because we do not know what percentage of the theoretical peak we should be expecting for a particular application. Portability is primarily defined by the amount of performance you are giving up by not writing multiple versions of code customized to each architecture.

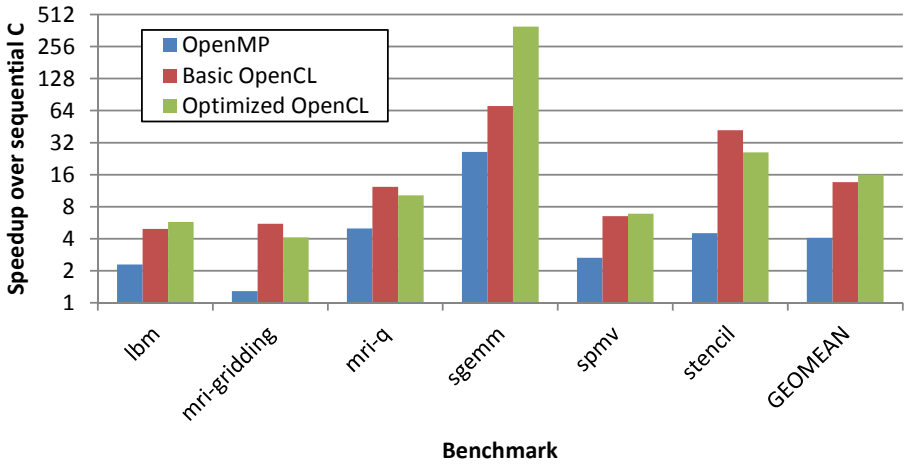


Figure 7.1: A study of the performance of various parallel implementations compared to a baseline, single-threaded CPU implementation

The question then becomes, what alternative code versions should we be comparing against?

For many software developers, the code to beat is the code they already have, and that code may not have been particularly well optimized. When it comes time to turn attention to an existing piece of code and improve its performance, there are natural questions of how much return on development effort can be achieved through optimization. Here, we assume that acceleration is important, and that an accelerator implementation is therefore imperative. While this dissertation does not make any particular claims about the ease of adopting an accelerator programming model, assuming that one must, we can measure the performance return on that investment.

7.1 CPU Performance Effects of Programming in OpenCL

Figure 7.1 shows the clear benefit of targeting OpenCL as a portable language that captures high-level optimization patterns. The OpenMP results show that the benchmarks do benefit from parallel scaling. However, the OpenMP compiler was typically not able to perform any more advanced optimization transformations than the C compiler. Even the baseline OpenCL implementations, due to the constraints of the programming model, had somewhat more natural tiling units in the OpenCL work-groups, and typ-

ically got better performance than their basic OpenMP counterparts. The GPU-optimized OpenCL kernels generally were comparable to the baseline OpenCL implementations on the CPU architecture, in part because the programming model forced even the baseline implementation to accommodate some form of locality simply by enforcing a two-level hierarchical task decomposition. Only one benchmark showed a performance change of more than 2× due to GPU-specific optimization, with one notable exception. The drastic performance change in `sgemm` highlights just how much that benchmark is dependent on manual tiling, despite it being the most easily transformed and most studied kernel in the field. The OpenMP code performs both input and tiling on the output, setting each parallel OpenMP task to compute a tile of the output matrix, and processing sections of input in sequence. Yet despite the programmer obviously taking efforts to improve the OpenMP code performance, the choices made were suboptimal compared to the choices made simply to map the programming model to the OpenCL language's hierarchical tiling scheme, even without explicit input tiling. By enforcing a vector-based serialization pattern, the MxPA tool was forcing the task-tile of the naive OpenCL kernel to compute on the input data in a tiled fashion. Manual input tiling was able to speed up the kernel by an additional 7×. On average, the OpenMP implementations were approximately scaling performance with the number of cores on the test system (four), while the OpenCL implementations were approximately another factor of four higher, due to improved tiling and vectorization enforced by the programming model for even the "baseline" implementations.

The Rodinia benchmarks were excluded from Figure 7.1 because they have neither sequential baseline code to normalize for nor multiple OpenCL versions embodying different optimization levels. However, we can examine the benchmarks together by directly comparing the relative performance of the OpenMP and OpenCL implementations in both suites. For this purpose, we regard the Rodinia benchmarks as most similar in optimization level to the Parboil basic OpenCL implementations. Like the Parboil OpenCL base versions, these kernels were mostly developed by students in accelerator programming courses, applying their optimizations skills to some new kernel as a project. This means that software development institutions hiring new graduates and assigning them to accelerator kernel programming would likely get these kinds of results.

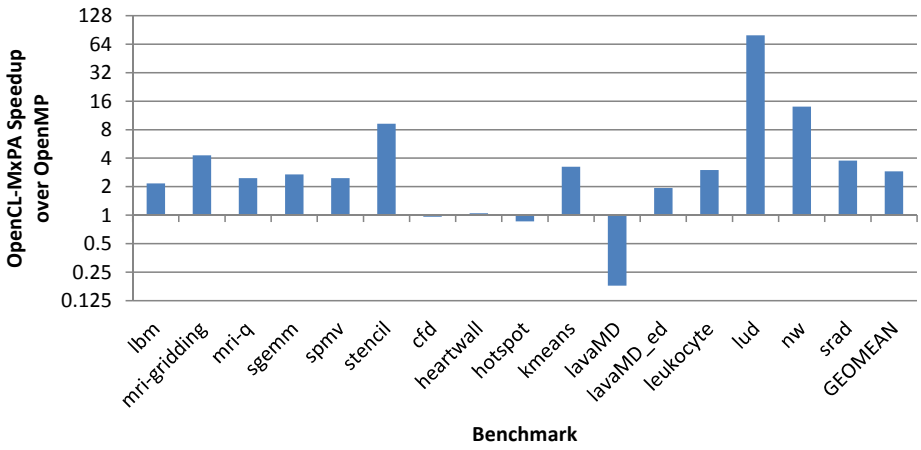


Figure 7.2: Comparing OpenCL and OpenMP implementations of the Parboil and Rodinia benchmarks

7.2 Comparing OpenMP and OpenCL as Parallel Programming Models for a CPU

Figure 7.2 shows those results, with strong evidence that the OpenCL performance was getting a significantly better final performance than the comparable OpenMP implementations. Comparing multiple source versions is always difficult, because we do not have a firm understanding of how much effort was spent on each kernel, whether the same programmer or a programmer of the same skill was being applied to each, and therefore what software development environments are most comparable to the results shown. However, we do know that the software is real, considered publishable by their respective developer groups, and developed independently of this dissertation research.

The results show that in all but one case, the OpenCL implementation either beat or nearly matched the performance of the OpenMP implementation. On one extreme, the three benchmarks where MxPA gained the most, `lud`, `nw` and `stencil`, were all cases where individual kernels completed in a very short amount of time, on the order of milliseconds, although the application as a whole would still run for several seconds. This shows that despite extensive research and studies of reducing the overhead of OpenMP work distribution automatically, the direct coarsening of the OpenMP tasks accomplished through the serialization of OpenCL work groups proved to be much more efficient for short-running kernels. (The `lud` benchmark addi-

tionally performs no direct input tiling on the OpenMP version of the code, resulting in even more exceptional speedups. The GEOMEAN performance improvement of MxPA over OpenMP is reduced to only 2.1× if that data point is excluded.)

The one case where the OpenMP application significantly outperformed the OpenCL application on MxPA was `lavaMD` from the Rodinia benchmark suite, where the kernel code structure causes the MxPA compiler to fall back to region-based serialization specifically on regions where the threads are block-copying data from global to local memory. The region-based serialization causes the large-strided accesses, as highlighted in Chapter 5, with demonstrated performance loss. Stronger MxPA compiler analyses should be able to improve the performance of that kernel significantly.

In summary, the Parboil and Rodinia benchmarks show that not only is OpenCL an applicable programming model for parallel CPU architectures, but that with the appropriate implementation methodology, it is more than comparable with the OpenMP implementations developed by the same teams that developed the OpenCL benchmarks. While we can find instances where the OpenMP benchmarks could be further optimized, the same could be said about the OpenCL versions, especially considering our knowledge of the limitations of the MxPA compiler, which were clearly demonstrated in some cases.

Finally, the results overall most clearly demonstrate that in practice, one of the most highly regarded C/OpenMP compilers of our time does not perform the optimizations that compiler research has promised, such as automatic loop tiling or reducing the overhead of distributing very small OpenMP tasks. As a programmer, this is somewhat disheartening, because it means that most of what makes accelerator programming hard is necessary even for good CPU performance, particularly explicit tiling and parallelization. However, the results of this section show that when such transformations are applied for the benefit of accelerator architectures in a language like OpenCL, a well-designed compiler and runtime for that same language can preserve the benefits of those transformations for a CPU architecture as well.

CHAPTER 8

EXTENDING PERFORMANCE PORTABILITY TO BROADER ARCHITECTURE CLASSES

Performance portability may not be limited to only CPU and GPU architectures, although those are the only clearly demonstrated platforms of this thesis. The fundamental translation mechanism of coarsening fine-grained parallel programming models into larger tasks makes the technology a very versatile methodology for targeting other kinds of architectures as well. This chapter summarizes some of the ongoing, collaborative efforts adapting the core technology of this dissertation to other architectures.

8.1 Rigel

The Rigel architecture is a 1024-core MIMD research architecture developed by colleagues at the University of Illinois [20], targeting task- and data-parallel visual computing workloads that scale up to thousands of concurrent tasks. The design objective of Rigel is to provide high compute density while enabling an easily targeted, conventional programming model.

The Rigel architecture is summarized in Figure 8.1. The fundamental processing element of Rigel is an area-optimized, dual-issue, in-order core with a RISC-like ISA, single-precision FPU, and independent fetch unit. Eight cores and a shared cache comprise a single Rigel cluster. Clusters are grouped logically into a tile using a bi-directional tree-structured interconnect. Eight tiles of 16 clusters each are distributed across the chip, attached to 32 global cache banks via a multistage interconnect. The last-level global cache provides buffering for 8 high-bandwidth GDDR memory controllers.

Applications are developed for Rigel using a task-based API, where a task is mapped to one Rigel core. Tasks can vary in length and do not execute

Parts of Section 8.1 have been adapted from portions of a previously published work, ©2012 Springer-Verlag, used with permission [25]. The original work was written in collaboration with S. Kofsky, D. Johnson, W. Hwu, S. Patel, and S. Lumetta.

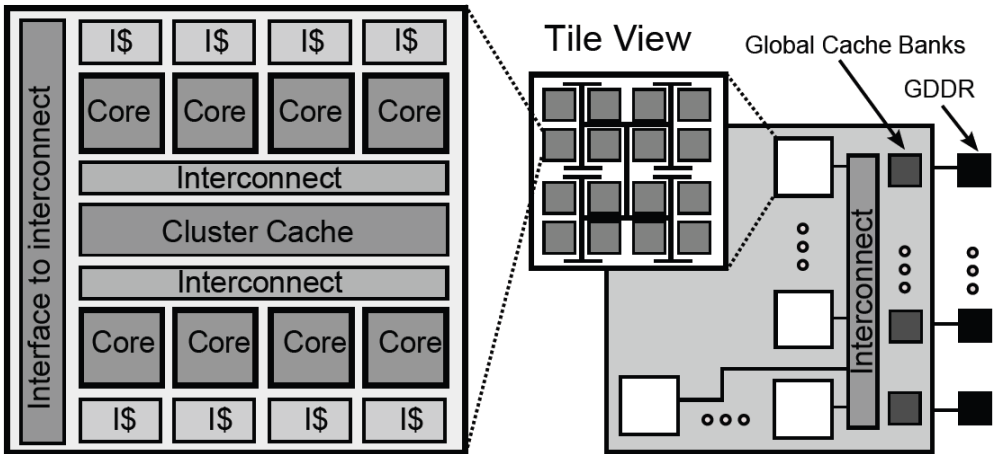


Figure 8.1: Rigel architecture diagram

in lock-step. Task generation and distribution is dynamic and handled by software; the hardware only implements global and cluster level atomic operations.

8.1.1 Mapping CUDA to the Rigel architecture

Rigel's fundamental design rejects wide SIMD as inapplicable for certain workloads, but still pursues energy-efficient, throughput-oriented computing. It would be possible to run the fully coarsened code used for large-core CPUs on each of the thousand cores of the architecture, but that is not the ideal scenario. In the Rigel architecture, the closest analog to a CPU core or a GPU processor is not a single core, but a cluster, which holds a private cache roughly equivalent to a CPU core's L1 cache or a GPU's processor cache. The cores within a cluster are also capable of executing local barriers very efficiently, making the distribution of tasks within a group among the cores of a cluster more palatable.

To target a CUDA kernel to a Rigel cluster, we extended the MCUDA tool to transform the source code to be amenable to Rigel's MIMD execution model. For each serialization region, instead of completely serializing the region, the region is parallelized into small tasks for each thread index within

the group, which are dynamically distributed over the Rigel cores in a cluster using the architecture's hardware-assisted task-management system. At the end of a region, the thread queue on the cluster is reset so that the cluster can iterate over each thread again. Shared variables are stored as a per-cluster data structure. Each core can read and write to the shared data through the cluster cache. Further, local variables are stored in a cluster level data structure since we allow CUDA threads to migrate between cores within a cluster across regions. However, local CUDA thread variables that are produced and consumed between synchronization points do not have to be replicated since they are not used when a CUDA thread moves to another core.

8.1.2 Performance considerations

Unlike GPUs, Rigel uses software to handle the work distribution from CUDA kernels, which to incur some potentially avoidable overhead for task assignment and distribution. The RCUDA runtime supports load balancing at the cluster level by allowing individual cores to fetch CUDA threads on demand. Dynamic fetching can be expensive for short regions, which could result from either very simple kernels, or kernels with many synchronization points. An alternative is to statically assign work to each Rigel core such that each core executes a fixed portion of CUDA threads within a thread block. The static assignment significantly reduces the number of Rigel tasks a core must fetch in a region, at the cost of reduced dynamic load balancing ability. Therefore, for static work assignment to perform optimally, the CUDA threads must perform similar amounts of work and the number of CUDA threads should be divisible by eight so that each Rigel core does the same amount of work.

Sometimes such a static schedule opens up new opportunities for optimization. Just as in-code, region-based serialization opens up more opportunities for CPU compilers, some degree of in-code serialization enables optimization across multiple logical threads on Rigel as well. Thread fusing is a source level transformation that merges threads into a group so they can execute in parallel through software pipelining. For some kernels it is advantageous to enforce an execution order as a way to optimize memory accesses.

In CUDA code with a two-dimensional thread block, it is common to see

an indexing function based on the thread index. For example:

```
(threadIdx.y * BLOCK SIZE) + threadIdx.x
```

.

The Y dimension is multiplied by a constant factor, usually the block size or some other constant such as the width of an input matrix. On the other hand, the X dimension of the thread index is used to direct threads with contiguous X indexes to contiguous memory addresses. On Rigel, it is beneficial to concurrently execute CUDA threads with the same Y value so that the cores effectively share the same cache lines in the shared cluster cache. In addition to enforcing an execution order, fusing threads is also advantageous since it allows the compiler to optimize across a group of threads. The code in CUDA threads is the same, except for the thread index values, and with thread fusing the compiler is able remove redundant computation, creating faster, more efficient code.

8.1.3 Initial evaluation

All performance results for the Rigel accelerator design are produced using a cycle-accurate execution driven simulator that models cores, caches, interconnects, and memory controllers [20]. We use GDDR5 memory timings for the DRAM model. Benchmark and library codes are run in the simulator and are compiled with LLVM 2.5 using a custom backend. Inline assembly was used for global and cluster level atomic operations. Optimizations have yet to be fully implemented in our compiler, and thus were applied by hand editing translated CUDA kernels. Results for CUDA on a GPU were gathered on a Tesla T10 4-GPU server using one GPU.

With the exception of MRI and SAXPY, all benchmark codes were taken from external sources and were originally written to be executed on a GPU. Our benchmarks include a 2D image filter with 5x5 kernel (Convolve), dense-matrix multiply (DMM), 256-bin histogram (Histogram), fractal generation (Mandelbrot), medical image construction (MRI), SAXPY from BLAS (SAXPY) and matrix transpose (Transpose). MRI uses two kernels: the first to initialize data structures, and the second to perform the actual computation. Histogram also uses two kernels: the first calculates many partial histograms from subsets of the input array, and the second merges the partial histograms. Table 8.1 lists data sizes and characteristics for all benchmarks.

Table 8.1: RCUDA benchmarks

Name	**Data Set**	**# Kernels**	**Thread Block Dimensions**	**Shared Memory Usage?**
Convolve	1024×1024	1	(16, 16)	Yes
DMM	1024×1024	1	(16, 16)	Yes
Histogram	2M	2	(192,1) (256,1)	Yes
Mandelbrot	512×512	1	(16,16)	Yes
MRI	8192,8192	2	(512,1),(256,1)	No
SAXPY	2M	1	(512,1)	No
Transpose	1024×1024	1	(16,16)	Yes

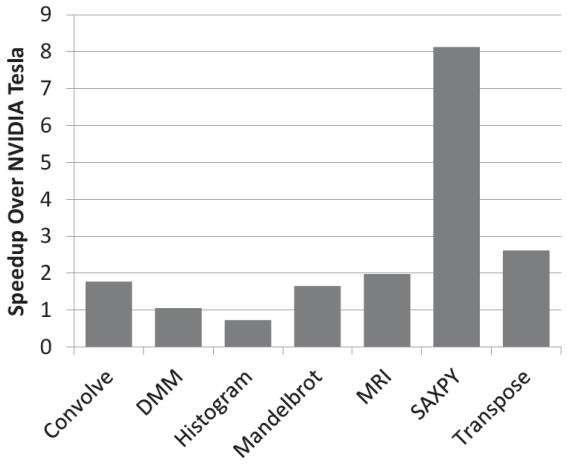


Figure 8.2: RCUDA baseline results

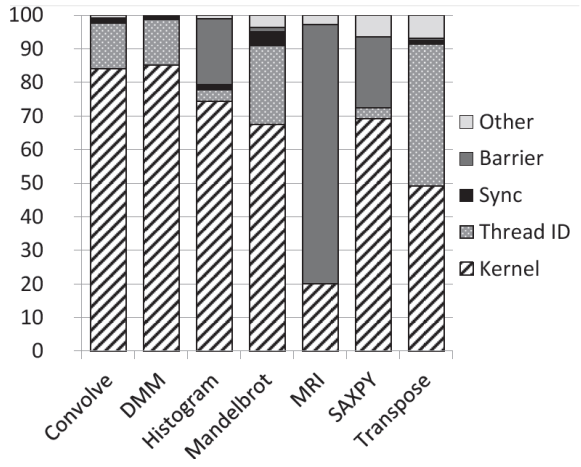


Figure 8.3: RCUDA baseline performance breakdown

In Figure 8.2 we show the normalized speedup of the naive translation on Rigel over NVIDIA's Tesla. Given that Rigel's peak throughput is about 1.1 times that of the tested GPU, these results show that the GPU-optimized code is also well suited to Rigel with the RCUDA methodology of implementation, achieving speedups even beyond 10% improvement in most cases for the baseline, fully-dynamic scheduling methodology.

We analyze the runtime overhead of our RCUDA framework on Rigel, shown in Figure 8.3. We break down runtime into five categories: (1) Kernel, which is the measurement of the time spent executing the CUDA kernel code, (2) Thread ID, the overhead of generating the CUDA thread indexes when dynamic load balancing is used, (3) Sync, the time spent in the thread block barrier call, (4) Barrier, measuring the time cores spend waiting for kernel execution to complete, which represents load imbalance, and (5) Other, which includes all other overheads including runtime initialization, thread block fetch and host code.

We see that thread index generation is quite expensive, particularly for kernels with two-dimensional thread blocks. For two-dimensional thread blocks, the CUDA thread indexes are generated from a count of remaining threads. The conversion from a one-dimensional count to a two-dimensional index requires a significant amount of computation that can be comparable to the total work of smaller CUDA kernels such as Transpose and Convolve. Additionally, thread indexes are generated twice in Transpose and Convolve due

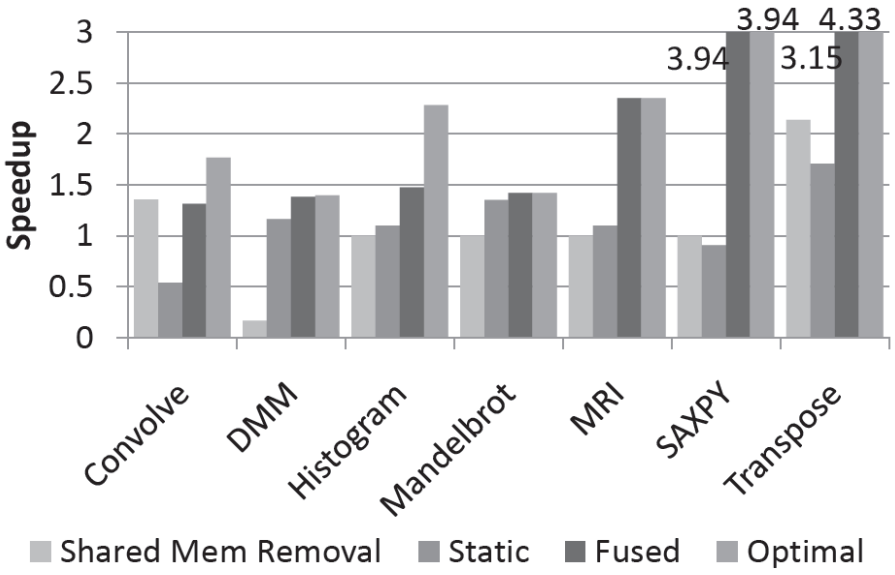


Figure 8.4: RCUDA optimization performance effects measured as speedup over baseline

to a single synchronization point in each kernel. We find that the time spent in the thread block barriers is low, even though it is implemented in software. We see that in Histogram and SAXPY the barrier constitutes roughly 20runtime. The Histogram code does not generate a large enough grid to utilize the entire chip, so some cores only wait in the barrier without executing any kernel code and SAXPY has a very short kernel, so load imbalance contributes to the high barrier cost. The barrier makes up the majority of MRI's runtime due to insufficient parallelism that leaves most clusters idle. The first kernel utilizes only 16 clusters while the second kernel only uses 32 of the 128 available clusters. The amount of parallelism would increase with larger datasets, which would take significantly longer to simulate fully.

We apply optimizations individually to each benchmark and then combine the beneficial optimizations to create an optimal version of each benchmark as shown in Figure 8.4. Shared memory removal was applied to the Convolve, DMM and Transpose benchmarks. Removing the shared memory accesses also allowed for all the synchronization points to be removed. DMM was the only benchmark where the optimization did not improve the runtime because the mapping function generated for DMM is complex, requiring costly multiplication instructions. All benchmarks except Convolve and SAXPY showed

an improvement when using static scheduling of threads. Convolve is the only benchmark where the amount of work varies greatly between threads because not all CUDA threads compute an output value. SAXPY has very short kernels, so the overhead of statically dividing the workload is significant, and the runtime increases by 10%. Thread fusing improves the performance of all benchmarks; in every case, at least some amount of redundant calculation could be removed. The optimal version of each benchmark uses the combination of optimizations that results in the fastest runtime. Convolve uses shared memory accesses removal along with thread fusing. DMM and Histogram use static work partitioning and thread fusing. Mandelbrot, MRI, Transpose and SAXPY only use thread fusing.

8.1.4 Conclusions

Clearly, the core MCUDA infrastructure was sufficiently powerful to enable a very different execution model. The changes to the execution model were primarily focused on mapping regions of the kernel code, which are completely free of overing constraints, onto system resources appropriate for the working set typically assigned to a thread block. The optimizations for improving performance typically fell into two categories. One was platform-specific scheduling optimizations, more intelligently balancing static versus dynamic scheduling for the closely-collaborating cores in the cluster, which can be done completely automatically. The other highlights that the GPU-encouraged model of using a software-managed cache was poorly matched to the Rigel architecture's locality management and task-communication mechanisms. Taken alone, this observation could lead us to conclude that most kernels should eschew explicitly managed caches, given that modern GPUs, CPUs, and Rigel all provide implicitly managed caches.

8.2 FPGAs

With increasing transistor densities, the computational capabilities of commercial FPGAs provided by companies like Xilinx and Altera have greatly increased. Modern FPGAs are technologically in sync with the rest of the IC industry by employing the latest manufacturing process technologies and supporting high-bandwidth I/O interfaces such as PCIe. By embedding fast DSP macros, memory blocks and 32-bit microprocessor cores into the reconfigurable fabric, a complete SoC platform is available for applications which require high-throughput computation at a low power footprint. The flexibility of the reconfigurable fabric provides a versatile platform for leveraging different types of application-specific parallelism, whether coarse- or fine-grained, data- or task-level, or pipeline parallelism of various configurations. Reconfigurability, though, has an impact in the clock frequency achievable on the FPGA platform. Synthesis-generated wire-based communication between parallel modules may limit the throughput of designs with wider parallelism compared to smaller but faster-clocked architectures.

FPGA devices reportedly offer a significant advantage (4×-12×) in power consumption over GPUs. J. Williams et al. [55] showed that the computational density per watt in FPGAs is much higher than in GPUs. This is even true for 32-bit integer and floating-point arithmetic (6× and 2× respectively), which maximize the raw computational density of GPUs.

Programming FPGAs often requires hardware design expertise, as it involves interfacing with the hardware at the RTL level. However, the advent of several academic and commercial ESL design tools for HLS [11, 12, 56, 15, 14] has raised the level of abstraction in FPGA design. Most of these tools use high-level languages as their programming interface. Some of the earlier HLS tools could only extract fine-grained parallelism at the operation level by using data dependence analysis techniques. Extraction of coarse-grained parallelism is usually much harder in high-level languages that are designed to express sequential execution. To overcome this obstacle, some HLS tools have resorted to employing language extensions for allowing the programmers to explicitly annotate coarse-grained parallelism in the form of parallel

streams, tasks [56] or object-oriented structures [15]. In a different approach, special high-level languages that model parallelism with streaming dataflows have been employed in HLS tools [14].

8.2.1 AutoPilot C for FPGAs

AutoPilot's programming model conforms to a subset of C which may be annotated with pragmas that convey information on different implementation details. Synthesis is performed at the function level, producing corresponding RTL descriptions for each function. The RTL description of each function corresponds to an FPGA core which consists of private datapaths and FSM-based control logic. Attached to each core's FSM are start and done signals that enable cross-function synchronization (including function calls and returns).

The front-end engine of AutoPilot (based on the LLVM compiler [28]) uses dependence analysis techniques to extract ILP within basic blocks. Coarser parallelism, such as loop iteration parallelism, can also be exploited by injecting *AUTOPILOT UNROLL* pragmas in the code (assuming there are no loop-carried dependencies). Note that unrolling and executing loop iterations in parallel impacts FPGA resource allocation proportionally to the unroll factor. Concurrency at the function level is specified by the *AUTOPILOT PARALLEL* pragma. The affected functions are launched concurrently by the parent function, which stalls executing until every child function has returned. Thus it is possible to implement an MPMD execution model with a configuration of heterogeneous FPGA cores (i.e. parallel cores corresponding to different functions). Note that AutoPilot will schedule two functions (cores) to execute in parallel only when they cause no hazards. A hazard arises when two functions access the same memory block (resource hazard) or pass data from one function to another (data hazard).

With regard to memory spaces, AutoPilot may map variables onto local (on-chip) or external (off-chip) memories. By default all arrays get mapped onto local BRAMs while scalar variables are mapped on configurable fabric logic. C pointers may also be used (with some limitations) in the input code and, combined with the *AUTOPILOT INTERFACE* pragma, they can infer off-chip memory accesses.

8.2.2 FCUDA design methodology

The work of this dissertation seeds one potential pathway for synthesizing FPGA configurations by leveraging an existing HLS tool called AutoPilot, which takes annotated C code as its input. The advantages offered by the CUDA programming model in an FPGA design flow are multifold. Even though CUDA incorporates more memory spaces than AutoPilot, they both distinguish between on-chip and off-chip memory spaces, and leverage programmer-specified data transfers between off- and on-chip memory storage.

Coarse-grained parallelism in CUDA is expressed in the form of thread-blocks that execute independently on the independent processing units within the GPU. Moreover, the number of thread-blocks in CUDA kernels is typically on the order of hundreds or thousands. Thus, thread blocks constitute an excellent candidate in terms of lack of synchronization requirements and workload granularity for FPGA core implementation. Mapping thread-blocks onto parallel cores on the FPGA minimizes inter-core communication without limiting parallelism extraction. Low inter-core communication helps achieve higher execution frequencies and eliminate synchronization overhead. As a final point, CUDA provides a very concise programming model for expressing coarse-grained parallelism through the single-thread kernel model. AutoPilot (as most existing HLS tools), on the other hand, employs a programming model that expresses coarse-grained parallelism explicitly in the form of multiple function calls annotated with appropriate pragmas. FCUDA automates the extraction of the inferred parallelism in CUDA code into explicit parallelism in AutoPilot input code while handling data partitioning and FPGA core synchronization. Thus, it eliminates the tedious and error-prone task of directly expressing the coarse-grained parallelism in C for AutoPilot. Our FPGA design flow allows the programmer to describe the parallelism in a more compact and efficient way through the CUDA programming model regardless of the implemented number of FPGA cores.

Our CUDA-to-FPGA flow (Figure 8.5) is based on a code transformation process, FCUDA (currently targeting the AutoPilot HLS tool), which is guided by preprocessor directives (FCUDA pragmas) inserted by the FPGA programmer into the CUDA kernel. These directives control the FCUDA translation of the expressed parallelism in CUDA code into explicitly-expressed

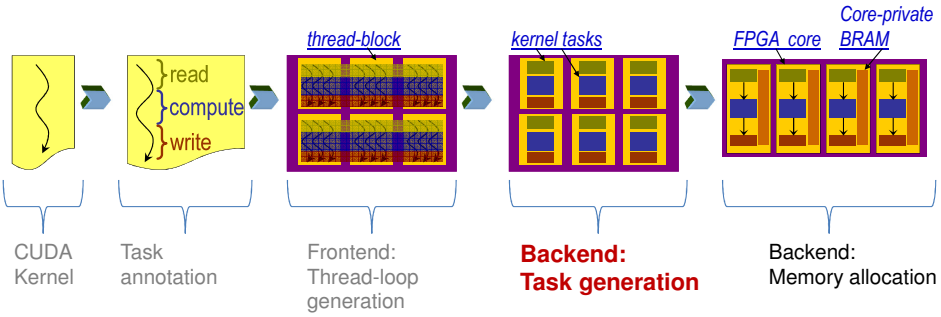


Figure 8.5: FCUDA toolchain flow

coarse-grained parallelism in the generated AutoPilot code. The FCUDA pragmas describe various FPGA implementation dimensions which include the number, type and granularity of tasks, the type of task synchronization and scheduling, and the data storage within on- and off-chip memories. AutoPilot subsequently maps the FCUDA specified tasks onto concurrent cores and generates the corresponding RTL description. Moreover, AutoPilot uses LLVM's dependence analysis techniques and its own SDC-based scheduling engine [56] to extract fine-grained, instruction-level parallelism within each task. Finally Xilinx FPGA synthesis tools are leveraged to map the generated RTL onto reconfigurable fabric. We demonstrated that the FPGA accelerators generated by our FPGA design flow can efficiently exploit the computational resources of top-tier FPGAs in a customized fashion and provide better performance compared to the GPU implementation for a range of applications.

8.2.3 FCUDA design details

Concurrency in CUDA is inferred through a single-thread kernel with built-in variables that are used to distinguish the tasks of each thread. Application parallelism is expressed in the form of fine-granularity threads that are further bunched into coarse-granularity thread-blocks. Even though thread-level parallelism can improve performance, thread blocks offer higher potential for an efficient multi-core implementation on FPGAs. As discussed previously, CUDA thread-blocks comprise autonomous tasks that operate on indepen-

dent data sets and do not need synchronization. Conversely, CUDA threads within a thread-block usually reference shared data which often results in synchronization overhead and/or shared memory access conflicts.

Parallelism in C code for FPGA synthesis by AutoPilot is explicitly expressed through parallel function calls. A single callee function with a different set of arguments in each call may be used to infer a homogeneous multi-core configuration similar to the GPU organization, whereas different callee functions may model a heterogeneous multicore configuration on FPGA. Therefore, the core task of the FCUDA source-to-source translation can be simply described as converting thread-blocks into C functions and invoking parallel calls of the generated functions with appropriate argument sets. Having extracted the coarse-granularity parallelism at the thread-block level, fine-granularity parallelism at the thread level may also be extracted, provided that non-allocated resources exist on the FPGA. This disparity in the thread parallelism extraction scheme between GPU and FCUDA may lead to different combinations of concurrently executing threads in the two devices. Nevertheless, the degree of parallelism will not differ in typical CUDA kernels that comprise hundreds of threads per thread-block and thousands thread-blocks per grid.

Another important feature of the FCUDA philosophy consists of decoupling off-chip data transfers from the rest of the thread-block operations. The main goal is to prevent long latency references from impacting the efficiency of the multicore execution. This is particularly important in the absence of GPU-like fine-grained multi-threading support in FPGAs. Moreover, by aggregating all of the off-chip accesses into DMA burst transfers from/to on-chip BRAMs, the off-chip memory bandwidth can be utilized more efficiently. FCUDA also leverages synchronization of data transfer and computation tasks based on the FCUDA annotation injected by the FPGA programmer. The selection of the synchronization scheme often incurs a tradeoff between performance and resource requirements. The FPGA programmer needs to consider the characteristics of the accelerated kernel in order to make an educated decision. A simple and resource-efficient scheme is the simple DMA synchronization which serializes data communication and computation tasks. This scheme is memory-overhead-free and it can also be a good fit for kernels that are compute-intensive and incur low data communication traffic. At the opposite end, the ping-pong synchronization scheme overlaps data

communication with computation by doubling the number of BRAM blocks. The interconnection logic interchangeably connects each BRAM block to the compute logic and the DMA controller, ensuring that each BRAM block is actively connected to only one of the two modules in each cycle. However, this scheme may result in BRAM utilization overhead, impacting the number of cores that can be instantiated on the FPGA.

The MCUDA tool, with minor modifications, was used as the front end of the FCUDA source-to-source compiler, performing region-based serialization and scalar expansion as necessary. The back-end engine of the FCUDA source-to-source compiler leverages the implementation information annotated in the FCUDA pragma directives to guide the translation of the kernel coarse-grained parallelism into the function-level type of parallelism supported by AutoPilot. Tasks annotated by FCUDA COMPUTE and TRANSFER pragmas are extracted into task functions to perform the specific computation or data transfer specified by that region, leaving calls to the extracted functions in the original kernel program locations, hereafter called the *parent function*. Multiple calls of the task functions wrapped within AUTOPILOT REGION and PARALLEL directives in the parent function drive the synthesis tool to instantiate parallel processing cores on the configurable fabric. The degree of parallelism is specified by the parameter information included in the COMPUTE and TRANSFER pragmas. Apart from the type and number of cores for each subtask, the FPGA programmer can also extract thread parallelism (provided available resources exist) by injecting AUTOPILOT UNROLL and PIPELINE pragmas within the FCUDA COMPUTE annotated tasks, to specify thread-loop unrolling and pipelining, respectively.

FCUDA TRANSFER pragmas are used to annotate data communication tasks to off-chip addresses. According to the FCUDA philosophy, off-chip data communication usually infers DMA burst transfers of data between off-chip memory storage and on-chip BRAM arrays. The FCUDA back-end engine is also responsible for instantiating array variables which will infer BRAM block allocation during synthesis by AutoPilot. BRAM associated arrays are instantiated at the parent function and their number is determined by the degree of parallelism annotated in the compute tasks that reference them. BRAM associated arrays may be passed as arguments to compute and transfer functions similarly to the rest of the variables. More details on the leveraging of different CUDA memory spaces within FCUDA are discussed

Table 8.2: FCUDA benchmarks

Kernel	Configuration	Description
Matrix Multiply (matmul)	1024x1024	Common kernel in many imaging, simulation, and scientific applications
Coulombic Potential (cp)	4000 atoms, 512x512 grid	Computation of electric potential in a volume containing charged atoms
RSA Encryption (rc5-72)	4 Billion Keys	Brute force encryption key generation and matching

in the paper by Papakonstantinou et al. [39].

8.2.4 Initial evaluation

For the evaluation of our FPGA design flow we targeted Xilinx Virtex5 FPGA devices. Virtex5 FPGAs are fabricated in 65nm CMOS technology and can be clocked at frequencies of up to 550MHz. These features render them good candidates for making meaningful comparisons with most of the GPU devices used at the time the work was done. Moreover, the Virtex5 family included some of the biggest and most advanced FPGAs available that have the capacity to efficiently host high-concurrency multi-core accelerators. For our comparison experiments we chose the XC5VFX200T Virtex5 device, which has more than 100K LUTs, 16Mbits of on-chip BlockRAM memory and 384 DSP units. The GPU device used for the comparisons was NVIDIA's G80 (90nm fabrication technology) with 16 SM units and 128 cores.

The CUDA kernels we used in these experiments are described in Table 8.2. Two of them (`matmul` and `cp`) were based on GPU-optimized versions that were tailored into different integer bitwidth versions. The third kernel was implemented without any device-specific optimizations. In these experiments we focused on integer performance. Figure 8.6 compares the FPGA and GPU performance for all versions of the 3 kernels. The `rc5-72` kernel is intrinsically based on modulo-shift operations within 32-bit integers, and thus it was not transformed into smaller integer bitwidth implementation. The FPGA performance results are based on the assumption that the off-

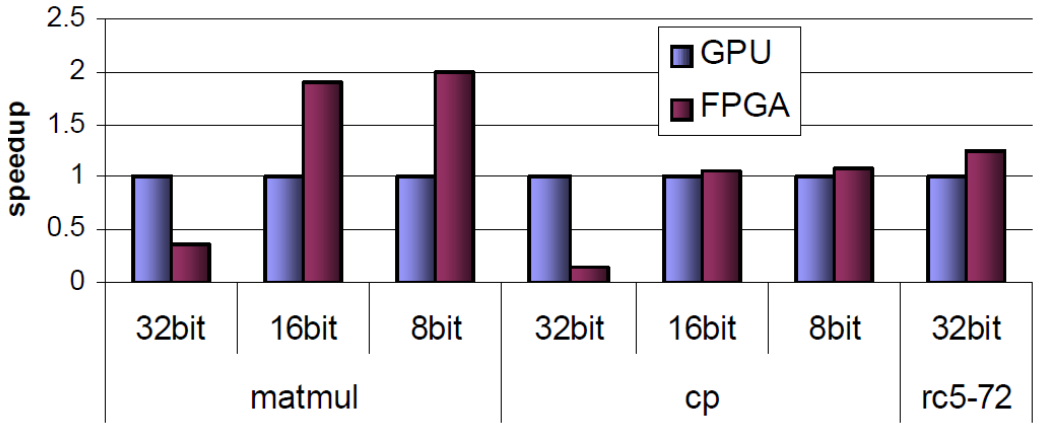


Figure 8.6: FCUDA results: speedup over GPU implementation

chip transfers are implemented by means of a high-bandwidth bus, such as the FSB (8.5GB/s) [19]. The computation task latencies are measured on the FPGA. Ping-pong synchronization is used between compute and data communication tasks and the latency of a single invocation is measured for all kernels. The GPU latencies do not include the data communication from/to the CPU.

As can be seen, the generated multi-core accelerators can outperform the GPU, especially in the case of smaller bitwidths, where application-specific customization can adapt the datapath of the cores and put the freed resources toward instantiating more cores. Moreover, the narrower datapaths allow faster operation execution. For example, the datapath FSM of the 16bit matmul has fewer states and also uses fewer DSP resources than the 32-bit one. In the case of 32bit CP, the compute intensive nature of the kernel results in high DSP utilization per core and low number of cores.

The number of implemented cores was determined by the resource (LUTs, BRAMs and DSPs) that was most restrictive. However, in the case that BRAM or DSP blocks are the limiting resource it may be possible to extract more parallelism without increasing the number of cores. For example, in the case of 16bit and 8bit matmul kernels where BRAM constrains the number

of cores, a 2X performance increase was achieved by exploiting thread-level parallelism within thread blocks. This is enabled by using AUTOPILOT UNROLL pragmas in the generated thread-loops.

8.2.5 Conclusions

The collaborative FCUDA work established once again that the hierarchical, fine-grained SPMD programming model popularized by GPU architectures has rich opportunities for exploiting parallelism at multiple levels and granularities. Although much of the transformation was initially driven by programmer annotation, current work is attempting to remove those restrictions. Given that the GPU programs used in these experiments were only modified by the added pragmas in most situations, the results show the potential for FPGAs to directly compete with GPUs as accelerators with existing GPU kernels representing certain kinds of workloads.

8.3 Coarse-Grained Reconfigurable Arrays

CGRAs are a configurable processor architecture design well suited to being targeted by software-pipelined loops, and are typically treated as accelerators for such loops. The Samsung Reconfigurable Processor, for instance, couples a VLIW processor with a CGRA accelerator [45]. Typically, such accelerators are targeted with sequential code, analyzed and transformed by advanced compilers. However, with the technology in this dissertation, we were able to implement the SPMD kernel accelerator programming model on the same processor.

8.3.1 Samsung Reconfigurable Processor

SRP, shown in Figure 8.7, is a traditional DSP processor architecture for various multimedia applications without the support of GPU-style multithreading. In order to exploit the instruction-level and loop-level parallelism em-

Parts of Section 8.3 have been adapted from portions of a previously published work, ©2012 IEEE, used with permission [24]. The original work was written in collaboration with H. Kim, M. Ahn and W. Hwu.

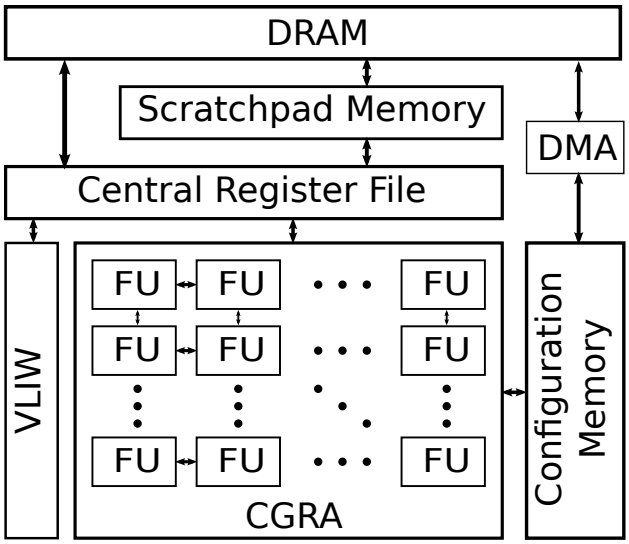


Figure 8.7: Block diagram of SRP architecture

bedding inside multimedia application, SRP has an accelerator called Coarse Grained Reconfigurable Architecture (CGRA). The CGRA is composed of an array of processing elements (PEs) such as functional units (FUs) and register files (RFs). These PEs are connected to each other by dedicated connection wires. The CGRA also has a dedicated memory for reconfiguring itself called *configuration memory*. By changing the content of the configuration memory, the CGRA can reconfigure itself for different kernels in multimedia applications. The configuration memory can host multiple loops simultaneously as long as their overall size is smaller than the capacity of the configuration memory. The kernels executed on the CGRA must be loops that can be modulo-scheduled [42]. All parts of the SRP code except the accelerated loops are executed on a separate VLIW processor. These code parts contain the instructions for the application control such as branch and jump and prepare the data necessary for the execution of the loops in the CGRA. In order to avoid the data copy for delivering program context from the VLIW to the CGRA or vice versa, the central register file is shared by the VLIW and the CGRA. Due to this, the execution handover from the VLIW to the CGRA or vice versa takes only a few cycles.

SRP has a simple but efficient two-level memory hierarchy. Instead of data cache, SRP has a scratchpad memory composed of multiple banks. As the access latency in the scratchpad memory is much shorter than in DRAM, it is usual to preload the necessary data in off-chip DRAM into the scratchpad

memory by using DMA.

Programming for the CGRA is implicit. The only thing for the application programmers to do is to select the loops that they want to accelerate on the CGRA. They can do it by adding a pragma right before the loops. Then the compiler for SRP automatically builds a configuration of the CGRA for the loop by the modulo-scheduling algorithm [40]. In the modulo-scheduling algorithm for the CGRA, the compiler tries to use not only instruction-level parallelism but also loop-level parallelism by overlapping several loop iterations at the same time. Generally an innermost natural loop is a good candidate for modulo-scheduling. Even though the compiler can find the configuration of good performance in many cases, it is advisable to fine-tune the loop for modulo-scheduling. The performance of the modulo-scheduled loop is mainly influenced by two factors: resource and recurrence constraints [42]. In order to increase the performance of the loop on the CGRA, the application programmers have to make their loop with the minimum recurrence if possible. They also should fit the maximum instantaneous resource usage in the loop to the available resources of the CGRA, even though if-conversion by the compiler can transform the control flow inside the loop.

8.3.2 OpenCL Compiler Framework for SRP

Figure 8.8: OpenCL compiler framework for CGRA

The proposed design of the OpenCL compiler framework is composed of a serializer and post-optimizer, in addition to the standard C compiler. Figure 8.8 shows the block diagram of the framework. The serializer is simply the MxPA OpenCL-to-C translator using region-based loop serialization, as described in Chapter 5.

The post-optimizer takes over the serialized kernel and performs specific optimizations for CGRAs. We have identified that additional optimizations must be developed in pursuit of maximum performance after serialization. In particular, the strength of CGRAs is the loop acceleration where it ex-

ploits wide instruction-level parallelism from an aggressive software pipelining. Since OpenCL kernels are generally regarded as the most performance demanding part of a program, program execution must remain in a CGRA as long as possible when it runs OpenCL kernels. Therefore, optimizations maximizing the coverage of software pipelinable loops in the serialized kernels should be followed.

The exposed serialization loop is often the innermost loop of the transformed kernel code, making it an excellent target for mapping to a CGRA. Serialization loops bring useful properties that the compiler can take advantage of. First, the serialization loops are canonical loops. Second, they are natural loops in that they have single entry and single exit. Third, execution of the serialization loops does not carry data dependence over its iterations by assertion, because the loop iterations were originally expressed as parallel tasks. Such properties are extremely valuable for the post-optimizer to further optimize the code with loop-level transformations, which could at some future point be integrated into Samsung's C compiler for SRP.

Resource Utilization Optimization

A high degree of instruction-level parallelism can be achieved from a successful software pipelining of a loop on a CGRA. In software pipelining [27], the total execution cycle of a loop, denoted as T, can be calculated from Eq. 8.1 as shown below:

$$T = (N - 1 + S) \times II, \tag{8.1}$$

where N is the trip count of the loop, II is initiate interval and S is stage count. II and S dictate the performance of the loop execution. Both resource and recurrence constraints play a key role for compiler in determining II. Among them, the compiler can ignore recurrence constraints according to a property of serialization loops. Therefore, the performance depends on the resource constraint.

While smaller II implies better performance in general, such low II can be caused by too few operations to schedule, resulting in many unused FUs. Resource utilization, denoted as R, is a metric to measure the efficiency of hardware for a given task as defined in Eq. 8.2:

Table 8.3: II, resource utilization and performance over different unrolling factors

Unrolling factor	II	Resource utilization	Performance (cycles)
1	5	0.11	5156
2	5	0.17	2608
4	5	0.27	1348
8	8	0.29	1088
16	14	0.30	964

$$R = \frac{M}{II \times W}, \tag{8.2}$$

where M is the number of operations of the loop, and W is the number of FUs of the reconfigurable grid. The importance of realizing smaller II is stressed here again in pursuit of better resource utilization. It implies a compiler should schedule as many operations as possible under the same II envelop.

Unrolling a loop with low resource utilization is a valuable optimization along with serialization. As previously mentioned, unrolling does not change recurrence constraints due to the inherent properties of OpenCL kernels. Therefore, a compiler can safely unroll a loop until full resource utilization is obtained.

Table 8.3 shows trends of II, resource utilization and performance for a serialized OpenCL vector addition kernel by changing the unrolling factor from two to sixteen. The loop is software pipelined over a CGRA of 4x4 FUs. Performance is measured on a cycle-accurate simulator assuming a perfect memory system. The latency of load operation is set to four cycles. The rate of increase of II is far lower than that of the unrolling factor, and it manifests multi-factor speedup. Beyond a point where the performance saturates, eight in this particular case, larger unrolling factor saturates resource utilization and begins to increase II proportionally.

Serialization Loop Flattening

Work-items in OpenCL have indexes with up to three dimensions. As such, serialization loops are formed as triply nested loops, as illustrated in Fig-

```
void kernel_func_3D(...) {
  for (z = 0; z < nz; z++) {
    for (y = 0; y < ny; y++) {
      for (x = 0; x < nx; x++) {
        ...
} } } }
```

(a) Serialization loops in the form of triply nested loops.

```
void kernel_func_3D(...) {
  for (zyx = 0; zyx < nz * ny * nx; zyx++) {
    z = zyx / (ny * nx);
    y = (zyx / nx) % ny;
    x = zyx % nx;
    ...
} }
```

(b) Flattened serialization loops

```
// ny = (1 << ly), nx = (1 << lx)
void kernel_func_3D(...) {
  for (zyx = 0; zyx < nz * ny * nx; zyx++) {
    z = zyx >> (ly + lx);
    y = (zyx >> lx) & (ny - 1);
    x = zyx & (nx - 1);
    ...
} }
```

(c) Flattened serialization loops using bit operations for index calculations.

Figure 8.9: Serialization loop and flattening optimization examples.

ure 8.9 (a). Therefore, software pipelining cannot be done for the outer loops, deferring processing of them to the control processor. As a consequence, branch and arithmetic operations from the outer loops will contribute to the execution cycles. It will also have to tolerate overheads due to switching execution mode between the control processor and the CGRA.

The nested serialization loops can be transformed into a single flattened loop [21]. A single loop reduces branch overhead from nested loops. In a case where the flattened loop is the innermost loop, the resulting code can run on a CGRA for many more cycles from the extended loop trip count. It also removes execution mode switching overhead.

Flattening the serialization loops is straightforward as they are canonical loops and natural loops at the same time. Index calculation from the flattened loop is implemented in a simple arithmetic. Figure 8.9 (b) shows the transformation example.

SRP's CGRA does not support integer division and modular operations by hardware. The compiler instead replaces them with equivalent software implementations, which in turn disqualifies the flattened loop for software pipelining.

In OpenCL, configuring a power of two number of work-items per work-group is a common practice [35]. This is because GPU hardware is designed to allocate resources in power-of-two units. If the programmer leaves the group size unspecified, then the OpenCL driver can choose a number of work-items per work-group. For the SRP architecture, a heuristic choosing

a power-of-two group size would be beneficial.

Under the condition where the loop trip counts are a power of two, flattening becomes available with efficient bit operations, as shown in Figure 8.9 (c). In this particular case, the compiler can substitute the integer division and modular operations with equivalent bit operations for the index calculation. This method can be extended further to address arbitrary trip counts by increasing the trip count to the next power of two that is equal to or larger than the actual trip count. The loop body is guarded with a predicate from comparing the loop index to the actual trip count. The conditional statement should be successfully if-converted by the compiler for software pipelining.

Serialization Loop Fission

When the original kernel code is imperfectly nested by the serialization loops, it effectively prevents the resulting code from being mapped to a CGRA. This is caused when the kernel code itself contains loops, which we call *kernel loops*, and they have sets of statements to execute either before and after them, such as Figure 8.10 (a). Because the kernel loop is the innermost loop, it alone will be considered for mapping onto a CGRA, forcing the leading and trailing blocks to execute on the slower control processor. By breaking the kernel code at the boundaries of kernel loops, the leading or trailing code blocks of a kernel loop become loop bodies of serialization loops, and available for mapping on a CGRA. Breaking the serialization loops is safe because the original OpenCL work-items have no execution dependencies by assertion.

Figure 8.10 (c) shows an example of loop fission for serialization loops. It contains initialization, an innermost loop and a termination part. After the fission, the initialization and termination can run on a CGRA as they are identified as software pipelinable. Thus, the transformation enlarges the coverage of a CGRA execution of the OpenCL kernel. Note that the post-optimizer could further apply additional transformations for kernel loops containing serialization loops. For instance, loop interchange or flattening could be applied to the nested loop in Figure 8.10 (c), transforming into a perfectly nested loop.

```
__kernel void fn(...) {
  ...   // init
  for (i = 0; i < N; i++) {
    ...
  }
  ...   // finish
}
```

(a) OpenCL kernel

```
void serial_fn(...) {
  for (x = 0; x < nx; x++) {
    ...   // init
    for (i = 0; i < N; i++) {
      ...
    }
    ...   // finish
} }
```

(b) Serialized kernel

```
void serial_fn(...) {
  for (x = 0; x < nx; x++) { ... // init }
  for (i = 0; i < N; i++) {
    for (x = 0; x < nx; x++) {
      ...
  } }
  for (x = 0; x < nx; x++) { ... // finish }
}
```

(c) Serialized kernel

Figure 8.10: Example of serialization loop fission.

```
// OpenCL code
__kernel void fn(__global uchar4* c,
    __global uchar4* a, __global uchar4* b) {
  c[idx] = a[idx] + b[idx];
}

// Serialized kernel code
void serial_fn(srp_uchar4* c,
    srp_uchar4* a, srp_uchar4* b) {
  for (x = 0; x < wgs; x++) {
    c[idx] = _I_intr003_rg_addb(a[idx], b[idx]);
} }
```

(a) Lowering OpenCL vector type

```
// OpenCL code
__kernel void fn(__global uchar* c,
    __global uchar* a, __global uchar* b) {
  c[idx] = a[idx] + b[idx];
}

// Serialized kernel code
void serial_fn(uchar* c, uchar* a, uchar* b) {
  for (x = 0; x < wgs; x+=4) {
    c[idx] = _I_intr003_rg_addb(a[idx], b[idx]);
} }
```

(b) Vectorization at the serialization loop-level

Figure 8.11: Two examples of SIMDization for SRP

SIMDization

SRP's CGRA supports subword parallelism via SIMD intrinsic instructions for a selected set of operations. For SIMD instructions, a 32-bit register can be divided into 2 of 16 bits or 4 of 8 bits. The subword parallelism is especially useful for graphics applications where primitive information is stored in 8 or 16 bits.

SRP's subword parallelism can be used in two situations. First, direct translation of a group of built-in vector data types of OpenCL becomes available. OpenCL supports subword vectors of 8-bit or 16-bit, where supported sizes are 2, 3, 4, 8 and 16. Considering 8-bit subword, char$\boldsymbol{n}$ and uchar$\boldsymbol{n}$, one 4x8bit SIMD operation can replace 4 scalar equivalents when $\boldsymbol{n}$ is equal to or smaller than 4. For larger $\boldsymbol{n}$, more than one SIMD operations can

jointly be utilized. Second, OpenCL programs using subword scalar data types can be vectorized at the level of serialization loops. The loop-level vectorization requires that data dependency is shorter than the vector length and the loop needs to be innermost. Both can be guaranteed by properties of the serialization loops. The loop is strip-mined by the vector length, two or four in this case, and then each scalar instruction within the loop body is replaced with the corresponding SIMD operation. Figure 8.11 demonstrates two examples of the transformation. Note that the usage of SIMD intrinsics, as shown in the Figure 8.11, is adapted from the intrinsic model of the IMPACT compiler [8].

8.4 Initial Evaluation

All experiment results are acquired using a cycle-accurate simulator for SRP. The simulator assumes all data reside in on-chip scratchpad memory and as such the compiler assigns a uniform latency to all load operations. We also assume the configuration memory preloads all kernels so that no additional costs are added other than a few cycles of the execution handover overhead when reconfiguration happens. The latency of load operation is set to four cycles. The architecture is configured as 2-way VLIW and CGRA with 4x4 FUs.

We used four benchmarks to evaluate the relation of portability and performance. They are vector addition(vecadd), matrix multiplication(mm), matrix transpose(transpose) and reduction. Also, we implemented five versions for each benchmark for comparison. They are unoptimized C code, innermost accelerated of the unoptimized C code, fully hand-optimized C code, OpenCL code with serialization and OpenCL code with serialization and post-optimization. For demonstration purpose, we did not use floating point operations as availability of floating point units varies across different configurations of SRP architecture. OpenCL code is assumed to have gone through the compiler pipeline as described in the framework shown in Figure 8.8. The unoptimized C code in the simplest form is used as a portable baseline throughout the experiment.

Figure 8.12 illustrates speedups of various implementations of the benchmarks. For vector addition, the performance of OpenCL code is approaching

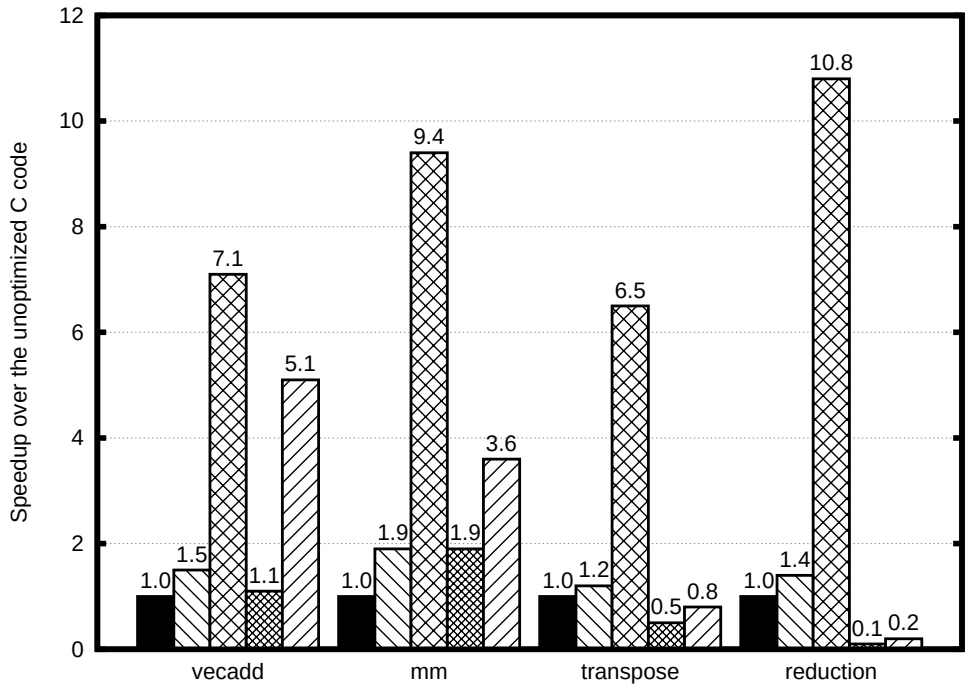


Figure 8.12: Experimental results for benchmarks. Five bars for a benchmark represent speedups of (1) the baseline, (2) the baseline with innermost acceleration, (3) fully hand-optimized C, (4) OpenCL kernel with serialization and (5) OpenCL kernel with serialization and post-optimization, respectively. The baseline is the unoptimized C code.

to the hand-optimized version, showing up to 5.1x speedup. The OpenCL code is optimized from unrolling the serialization loop eight times and mapped to the CGRA. In this simple kernel code, overhead compared to the hand-optimized one is attributed to the iterative execution of work-groups where the trip count of innermost loop execution is bounded within dimension of a work-group, leading to a frequent mode switching and execution in the VLIW.

The matrix multiplication demonstrates 3.6x speedup of the baseline, though it is slower than the hand-optimized version by 2.6x. The OpenCL code is optimized from loop flattening of the two nested serialization loops. Then the code is further split into three groups by loop fission, in that the first group sets global ids, the second group performs a dot product and the third group stores the results. The first and third group run in the CGRA. The second group is untouched and only the innermost loop runs in the CGRA. This is not as efficient as the hand-optimized version, where it flattens and unrolls

loops to form one innermost loop. We expect the gap would be narrowed as the post-optimizer extends its scope of optimizations beyond serialization loops so it can generate code similar to the counterpart.

The OpenCL code of matrix transpose shows 20% slowdown. The OpenCL code is optimized for GPUs where local memory tiling allows significantly better memory performance, which is not the case for SRP. From the serialization stage, local memory is lowered to a preallocated memory block in scratchpad memory and it becomes the sole burden on performance. It also introduces a barrier which limits the duration of CGRA execution. On the contrary, the baseline as well as the hand-optimized versions are implemented as a single innermost loop, loading and storing an item of an array using a different index. It exemplifies a challenge using an optimized OpenCL code for portability.

Reduction is a particularly interesting case as the performance gap between the hand-optimized C and OpenCL codes is the most significant. The OpenCL implementation of reduction uses a tree-shaped reduction over local memory, which is an algorithm well suited to a GPU. The code is written assuming that parallel threads execute in lockstep, and that barriers cost no more than any other single instruction. Serializing such code results in a loop for every level of the reduction tree. As the performance results show, the barriers are far from free, and the single-loop implementation used by the baseline implementation is much more effectively software pipelined. This also shows why reductions should be implemented as a library function by vendors [34].

8.5 Summary and Conclusion

For coarse-grained CPU architectures, accommodating memory systems and high inter-thread communication costs naturally lead to a model where the entire task group should be serialized into a single thread. However, the region formation algorithm ultimately results in small regions of code with perfect do-all parallelism that can be targeted to other architectures with different mechanisms. The efforts are somewhat less mature, but show high promise in initial implementations and experiments.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

9.1 Future Work: Additional Libraries for System-Specific Operations

There are a few ways in which the current hierarchical SPMD programming languages do not encourage portable programming practices. Reaching the goal of true performance portability will therefore require some language extensions and standard libraries with platform-specific implementations. Most often these were "collective" operations, where the threads in a group needed to coordinate with each other in common patterns that were inhibited by the barrier-dominated programming model. This section describes the set of new built-in library functions I would like to add to a hierarchical SPMD language in the future. The results of this dissertation have shown that in practice, these tend to be second-order performance effects, in large part because they are often sub-optimal accelerator programming patterns as well. Nevertheless, when they are necessary, these patterns cannot be implemented in a truly portable manner today.

9.1.1 Reductions

One good example of collective operation is a group-wide reduction, which is an algorithm whose efficient implementation depends too greatly on the real degree of parallelism in the target system.

Figure 9.1 shows the two most extreme implementations of a reduction, one entirely serial and the other maximally parallel. Neither solution is particularly portable, because the parallel implementation incurs much more synchronization overhead than necessary on a CPU, while the sequential algorithm seriously underutilizes the wide SIMD hardware of the GPU. Fur-

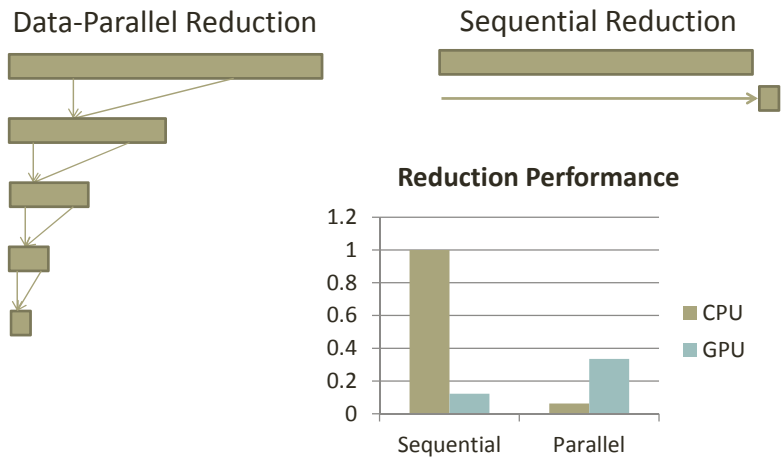


Figure 9.1: Graphical depictions of two reduction algorithms, and their runtimes on an example CPU and GPU architecture. The CPU results are from a quad-core x86 processor, while the GPU results are from an NVIDIA 9800 GX2.

thermore, neither implementation encourages the use of built-in reduction instructions across SIMD lanes in both CPU and GPU instruction sets. Other collective algorithms that typically require system-specific implementations are parallel scans [9], and sorts.

9.1.2 Memory copies and explicit locality control

For architectures without an implicit cache, programmers are required to use explicit scratchpad memory to control locality. On architectures with a strong implicit cache and no hardware scratchpad, copying data from "global" memory to "scratchpad" memory is often pure overhead, since both reside in the same cache hierarchy and memory space. Furthermore, architectures with scratchpad memory often also have the support of DMA engines for efficiently filling that scratchpad memory. However, the current hierarchical SPMD languages encourage the programmer to directly use the threads themselves to move data between global memory and scratchpad. This can significantly complicate code when the size of the data being cached does not directly match the size of the thread group working on that tile of data, as shown in Figure 9.2. Conversely, privatization of data results in block-shared arrays that eventually need to be copied or contributed to global output.

```
__global__ void block2D_reg_tiling(float c0,float c1,
    int nx, int ny, int nz, float Anext[nz][ny][nx],
    float A0[nz][ny][nx]) {
  dim3 tidx = threadIdx;
  dim3 bDim = blockDim;
  int i = blockIdx.x*bDim.x+tidx.x;
  int j = blockIdx.y*bDim.y+tidx.y;
  float bottom = A0[0][j][i], current = A0[1][j][i];
  if( i>0 && j>0 &&(i<nx-1) &&(j<ny-1) ) {
    for(int k=1;k<nz-1;k++) {
      float top =A0[k+1][j][i];
      Anext[k][j][i] = c1 * (top + bottom + A0[k][j+1][i] +
                  A0[k][j-1][i] + A0[k][j][i+1] +
                  A0[k][j][i-1]) - c0 * current;
      bottom=current;
      current=top;
    }
  }
}
```

(a) Without explicit locality control

```
__global__ void block2D_reg_tiling(float c0,float c1,
    int nx, int ny, int nz, float Anext[nz][ny][nx],
    float A0[nz][ny][nx]) {
  dim3 tidx = threadIdx;
  dim3 bD = blockDim;
  int i = blockIdx.x*bD.x+tidx.x;
  int j = blockIdx.y*bD.y+tidx.y;
  float bottom=A0[0][j][i];
  __shared__ float cur_plane[blockIdx.y+2][blockIdx.x+2];
  if( i>=0 && j>=0 && (i<nx) && (j<ny) )
    cur_plane[tidx.y+1][tidx.x+1] = A0[k][j][i];
  for(int k=1;k<nz-1;k++) {
    if(tidx.x == 0) {
      if(i != 0) cur_plane[0][tidx.x+1] = A0[k][j][i-1];
      if(i != nx-1)
        cur_plane[bD.y+1][tidx.x+1] = A0[k][j][i+bD.y];
    }
    if(tidx.y == 0) {
      if(j != 0) cur_plane[tidx.y+1][0] = A0[k][j-1][i];
      if(j != ny-1)
        cur_plane[tidx.y+1][bD.x+1] = A0[k][j+bD.x][i];
    }
    __syncthreads();
    if( i>=0 && j>=0 && (i<nx) && (j<ny) ) {
      float top =A0[k+1][j][i];
      Anext[k][j][i] = c1*( top + bottom +
             cur_plane[tidx.y+2][tidx.x+1] +
             cur_plane[tidx.y][tidx.x+1] +
             cur_plane[tidx.y+1][tidx.x+2] +
             cur_plane[tidx.y+1][tidx.x] )
          - c0 * cur_plane[tidx.y+1][tidx.x+1];
      bottom=cur_plane[tidx.y+1][tidx.x+1];
      cur_plane[tidx.y+1][tidx.x+1]=top;
    }
    __syncthreads();
  }
}
```

(b) With scratchpad usage

Figure 9.2: Simple stencil kernels demonstrating the complexity of scratchpad usage in a simple stencil benchmark, where the data tile size does not directly match the computational tile size.

I propose to develop an interface for explicit locality control that abstracts away these implementation details, allowing system designers to implement them as is best for each architecture. The primary features of the interface will be the ability to specify tiles of data in up to three dimensions, with cache, release, copy, and memset operations. On current GPUs, the implementations of these functions would use the threads directly to copy data as necessary. On CPUs, certain caching optimizations would be replaced with NOOPs, prefetches, locking of specific cache lines, or direct calls to `memcpy`. On systems with DMA engines, many of these operations would be directed to those engines.

9.2 Extending Serialization Techniques to Handle True Functions

Several possibilities exist for extending either the region-based or vector-based implementations to include true function support. One possibility is to treat all function calls as synchronization points, so that all threads are guaranteed to enter and exit the function. This would allow each function to be compiled separately, but it imposes additional constraints on the programming model.

An alternative approach is to use interprocedural analysis or a multipass compilation framework to first identify which functions may contain synchronization themselves. Functions that may contain synchronization are translated normally. The previously described translation of a function assumes that all threads will enter and leave synchronously. If the function contains a barrier, then this condition is necessarily true in any context from which it is called, due to the restrictions on barrier placement. It can always be compiled using the described techniques.

Functions that do not contain synchronization must be specially considered. In the case where a function cannot possibly be the entry point of a kernel of spawned threads and contains no synchronization, each invocation of the function will exist within a thread loop of another translated function. Implicit variables must still be supplied to the function as parameters, including the thread index itself, but the internal structure of the function need not be transformed. In the case where the function is the entry point of a kernel

invocation, it should be translated as shown in this chapter to implement the computation of all logical threads. If a function without synchronization is both a potential entry point of a kernel and called from other device functions, the function must be duplicated. One instance of the function is referenced only by other device functions and the other referenced only by kernel invocations. Each instance should be translated appropriately.

9.3 Alternatives and Related Work

This system design proposal is not the only feasible way of achieving performance portability, but is a minimum set of requirements for what the collection of systems must do to achieve it. Continued tools development can help move additional optimizations reliably under the systems control, e.g., by applying existing work on automatic locality management [38].

Other languages, particularly those at a higher level of abstraction, may provide tools with additional opportunities to generate good code for various platforms [47]. The work of those system developers should be assisted by this proposal, which provides a single low-level program representation that is portable across many parallel architectures. The developer of the high-level language is then shielded from the burden of architecture-specific optimizations for each new architecture, and can instead focus on the more universal but challenging issues of tasks of decomposition and locality management. Similarly, with strong high-level language and tool support targeting a unified interface, system developers can initially focus on implementing the low-level interface well.

In this dissertation, I identify the subset of optimizations and implementation decisions that must be handled by the system on a per-architecture basis. Without this support, performance portability is not possible, because it would expose conflicting software requirements from the various platforms.

REFERENCES

[1] Advanced Micro Devices. AMD accelerated parallel processing OpenCL programming guide, May 2012.

[2] E. Ashcroft and Z. Manna. Transforming 'goto' programs into 'while' programs. In *Proceedings of the International Federation of Information Processing Congress*, pages 250–255, Aug. 1971.

[3] E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, X. Martorell, and R. Silvera. Is the schedule clause really necessary in OpenMP? In *Proceedings of the International Workshop on OpenMP Applications and Tools*, pages 147–159, June 2003.

[4] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-M. W. Hwu. An adaptive performance modeling tool for GPU architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 105–114, Jan. 2010.

[5] M. Billeter, O. Olsson, and U. Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *Proceedings of the 2009 Conference on High Performance Graphics*, HPG '09, pages 159–166, New York, NY, USA, Aug. 2009. ACM.

[6] I. Buck. GPU computing with NVIDIA CUDA. In *SIGGRAPH '07*, page 6, New York, NY, USA, 2007. ACM.

[7] J. M. Bull. Feedback guided dynamic loop scheduling: Algorithms and experiments. In *European Conference on Parallel Processing*, pages 377–382, Sept. 1998.

[8] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W.-M. W. Hwu. IMPACT: an architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, pages 266–275, New York, NY, USA, 1991. ACM.

[9] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pages 666–675, 1990.

[10] clang: a C language family frontend for LLVM. http://clang.llvm.org/, accessed 4-Jan-2013.

[11] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. Platform-based behavior-level and system-level synthesis. In *SOC Conference, 2006 IEEE International*, pages 199–202, Sept. 2006.

[12] D. Gajski. NISC: The ultimate reconfigurable component. Technical Report TR 03-28, Center for Embedded Computer Systems, Irvine, CA, Oct. 2003.

[13] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. 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.

[14] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah. Optimus: efficient realization of streaming applications on FPGAs. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pages 41–50, New York, NY, USA, 2008. ACM.

[15] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 76–103, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A practical and robust method for scheduling parallel loops. In *Proceedings of the 5th Annual ACM International Conference of Supercomputing*, pages 610–632, June 1991.

[17] W.-M. W. Hwu, editor. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, San Francisco, CA, USA, Feb. 2011.

[18] W.-M. W. Hwu, editor. *GPU Computing Gems Jade Edition*. Morgan Kaufmann, San Francisco, CA, USA, Oct. 2011.

[19] Intel Corporation. Intel OpenCL optimization guide, Apr. 2012.

[20] D. R. Johnson, M. R. Johnson, J. H. Kelm, W. Yuohy, S. S. Lumetta, and S. J. Patel. Rigel: A 1,024-core single-chip accelerator architecture. *IEEE Micro*, vol. 31, no. 4, pages 30–41, July-Aug. 2011.

[21] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[22] K. Keutzer, B. L. Massingill, T. G. Mattson, and B. A. Sanders. A design pattern language for engineering (parallel) software: merging the PLPP and OPL projects. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 9:1–9:8, New York, NY, USA, 2010. ACM.

[23] Khronos OpenCL Working Group. The OpenCL specification, version 1.2, Nov. 2011.

[24] H.-S. Kim, M. Ahn, J. A. Stratton, and W.-M. W. Hwu. Design evaluation of OpenCL compiler framework for coarse-grained reconfigurable arrays. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 1–8, Dec. 2012.

[25] S. M. Kofsky, D. R. Johnson, J. A. Stratton, W.-M. W. Hwu, S. J. Patel, and S. S. Lumetta. Implementing a GPU programming model on a non-GPU accelerator architecture. In *Proceedings of the 2010 Workshop on Applications and Multi- and Many-Core Processors*, pages 40–51, Berlin, Heidelberg, 2010. Springer-Verlag.

[26] P. Kogge, editor. ExaScale computing study: Technology challenges in achieving exascale systems. Technical Report TR-2008-13, DARPA, Sept. 2008.

[27] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 318–328, New York, NY, USA, 1988. ACM.

[28] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[29] S. Lee, T. Johnson, and R. Eigenmann. Cetus - An extensible compiler infrastructure for source-to-source transformation. In *16th Annual Workshop on Languages and Compilers for Parallel Computing*, pages 539–553, 2003.

[30] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *Proceedings of the 6th Annual ACM International Conference on Supercomputing*, pages 104–113, July 1992.

[31] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, Boston, MA, USA, first edition, Sept. 2004.

[32] Microsoft Corporation. C++ AMP: Language and programming manual, version 1.0, Aug. 2012.

[33] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, vol. 6, no. 2, pages 40–53, 2008.

[34] NVIDIA Corporation. CUDPP: CUDA Data-Parallel Primitives Library, 2009.

[35] NVIDIA Corporation. NVIDIA OpenCL Best Practices Guide, 2009.

[36] NVIDIA Corporation. NVIDIA CUDA programming guide 5.0, 2012.

[37] OpenACC Corporation. The OpenACC application programming interface, version 1.0, Nov. 2011.

[38] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, vol. 29, no. 12, pages 1184–1201, Dec. 1986.

[39] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *7th IEEE Symposium on Application Specific Processors*, pages 35–42, Apr. 2009.

[40] H. Park, K. Fan, S. A. Mahlke, T. Oh, H.-S. Kim, and H.-S. Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 166–176, New York, NY, USA, 2008. ACM.

[41] M. Pharr and W. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Proceedings of the IEEE Conference on Innovative and Parallel Computing*, pages 1–13, May 2012.

[42] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *In Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, 1994.

[43] S. Rul, H. Vandierendonck, J. D'Haene, and K. De Bosschere. An experimental study on performance portability of opencl kernels. In *2010 Symposium on Application Accelerators in High Performance Computing*, pages 4–6, 2010.

[44] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-M. W. Hwu. Program optimization space pruning for a multithreaded GPU. In *Proceedings of the 2008 International Symposium on Code Generation and Optimization*, pages 195–204, Apr. 2008.

[45] Samsung Electronics. Samsung Exynos 4210 RISC Microprocessor, User's Manual, Aug 2011.

[46] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS parallel benchmarks in OpenCL. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC, pages 137–148, Nov. 2011.

[47] A. Sidelnik, S. Maledi, B. L. Chamberlain, M. J. Garzarán, and D. Padua. Performance portability with the Chapel language. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 582–594, May 2012.

[48] J. A. Stratton, N. Anssari, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, G. D. Liu, and W.-M. W. Hwu. Optimization and architecture effects on GPU computing workload performance. In *Proceedings of the IEEE Conference on Innovative Parallel Computing*, pages 1–10, May 2012.

[49] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, Mar. 2012.

[50] J. A. Stratton, C. Rodrigues, I.-J. Sung, L.-W. Chang, N. Anssari, G. Liu, W.-M. W. Hwu, and N. Obeid. Algorithm and data optimization techniques for scaling to massively threaded systems. *Computer*, vol. 45, no. 8, pages 26–32, 2012.

[51] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu. MCUDA: An effective implementation of CUDA kernels for multi-core CPUs. In *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing*, pages 16–30, July 2008.

[52] I.-J. Sung, G. D. Liu, and W.-M. W. Hwu. DL: A data layout transformation system for heterogeneous computing. In *Proceedings of the IEEE Conference on Innovative Parallel Computing*, pages 1–11, May 2012.

[53] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core

applications. In *Conference on Parallel Architectures and Compilation Techniques*, pages 513–522, Sept. 2010.

[54] V. Volkov and J. W. Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, CA, May 2008.

[55] J. Williams, C. Massie, A. D. George, J. Richardson, K. Gosrani, and H. Lam. Characterization of fixed and reconfigurable multi-core devices for application acceleration. *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 4, pages 19:1–19:29, Nov. 2010.

[56] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. AutoPilot: A platform-based esl synthesis system. In P. Coussy and A. Morawiec, editors, *High-Level Synthesis*, pages 99–112. Springer Netherlands, 2008.