PROGRAM OPTIMIZATION STRATEGIES FOR DATA-PARALLEL MANY-CORE PROCESSORS

BY

SHANE RYOO

B.S., University of Illinois at Urbana-Champaign, 2000
M.S., University of Illinois at Urbana-Champaign, 2004

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, 2008

Urbana, Illinois

Doctoral Committee:

Professor Wen-mei W. Hwu, Chair
Professor David A. Padua
Associate Professor Steven S. Lumetta
Assistant Professor Matthew I. Frank

ABSTRACT

Program optimization for highly parallel systems has historically been considered an art, with experts doing much of the performance tuning by hand. With the introduction of inexpensive, single-chip, massively parallel platforms, more developers will be creating highly data-parallel applications for these platforms while lacking the substantial experience and knowledge needed to maximize application performance. In addition, hand-optimization even by motivated and informed developers takes a significant amount of time and generally still underutilizes the performance of the hardware by double-digit percentages. This creates a need for structured and automatable optimization techniques that are capable of finding a near-optimal program configuration for this new class of architecture.

My work discusses various strategies for optimizing programs on a highly data-parallel architecture with fine-grained sharing of resources. I first investigate useful strategies in optimizing a suite of applications. I then introduce program optimization carving, an approach that discovers high-performance application configurations for data-parallel, many-core architectures. Instead of applying a particular phase ordering of optimizations, it starts with an optimization space of major transformations and then reduces the space by examining the static code and pruning configurations that do not maximize desirable qualities in isolation or combination. Careful selection of pruning criteria for applications running on the NVIDIA GeForce 8800 GTX reduces the optimization space by as much as 98% while finding configurations within 1% of the best performance. Random sampling, in contrast, can require nearly five times as many configurations to find performance within 10% of the best. I also examine the technique's effectiveness when varying pruning criteria.

To my parents, for their love and support.

ACKNOWLEDGMENTS

My graduate studies would have not been completed without the support and contribution of numerous individuals and parties. I am very grateful to everyone who helped me grow as a person and as a researcher. I would like to recognize many of them here, and apologize to any whom I have accidentally omitted.

I first thank my father and mother, my sister Cherie, and Louise for their continuous support. Their faith in my decision to pursue engineering and graduate study has been an anchor in my life.

Many thanks go to the former and current members of the IMPACT research group and other researchers within the Center for Reliable and High-Performance Computing who have contributed to this work. They include Christopher Rodrigues, Sam Stone, John Stratton, Sara Sadeghi Baghsorkhi, Sain-Zee Ueng, John Kelm, Robert Kidd, Stephanie Tsao, Christopher Kung, Ian Steiner, and James Player. I thank the numerous past and current members of IMPACT who have contributed to our compiler infrastructure. Finally, I thank the former students who helped me in my research endeavors, particularly John Sias, Hillery Hunter, and Ronald Barnes.

I thank Matt Frank and Steve Lumetta for serving on my committee and lending their experience and intellect to my work. I thank David Padua for serving on my committee and for his substantial contribution to parallel computing compilation, much of which serves as the foundation for my work. I also thank Michael Wolfe for his advice and suggestions.

I would like to recognize and thank the U.S. Department of Defense, the American Society for Engineering Education, the University of Illinois, the UIUC College of Engineering, the Graduate College at UIUC, the UIUC Department of Electrical and Computer Engineering, and the Roy J. Carver Charitable Trust for their financial support of my graduate studies. Intel, HP, and NVIDIA have supported our group with equipment donations and loans during my tenure. Our research is supported by the Gigascale Systems Research Center, which is funded under the Focus Center Research Program, a Semiconductor Research Corporation program. Initial experiments with NVIDIA GPUs were made possible by NSF CNS grant 05-51665.

Finally, I thank my adviser, Professor Wen-mei Hwu, for teaching me the fundamentals of computer architecture and optimizing compilers during my undergraduate education, for giving me the opportunity to perform graduate study and research in the IMPACT research group, and for his continued guidance and support. Without his dedication to teaching at all levels, his willingness to take on new students without demonstrated background in compilers, his generosity and compassion, and his determination in doing research that others think impossible, this work would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
CPU	Central Processing Unit
CSE	Common Subexpression Elimination
DRAM	Dynamic Random Access Memory
EPIC	Explicitly Parallel Instruction Computing
FLOPS	Floating-Point Operations Per Second
FP	Floating-Point
GPU	Graphics Processing Unit
ILP	Instruction-Level Parallelism
LICM	Loop-Invariant Code Motion
MLP	Memory-Level Parallelism
SIMD	Single-Instruction Multiple-Data
SFU	Special Function Unit
SM	Streaming Multiprocessor
SP	Streaming Processor
SPMD	Single-Program Multiple-Data
TLP	Thread-Level Parallelism
VLIW	Very Long Instruction Word

CHAPTER 1

INTRODUCTION

In the past decade, computer processor designs have shifted to multiple cores per die due to power and performance limitations. While this computing power is often utilized by multiple applications today, vendors are searching for solutions that enable a single application to achieve speedup by running on multiple cores. The number of cores is expected to increase at a near-exponential rate for the next several years, making finding a solution a pressing issue for the application and software tools community. The issue is compounded by most developers' lack of familiarity with highly data-parallel systems, and the differences between these single-chip systems and older parallel architectures. This dissertation shows that systematic principles and automatable tools can be used to optimize programs for this emerging class of architecture.

1.1 Obstacles

Programming and optimizing applications for highly parallel systems has historically been the domain of relatively few experts, with performance tuning done primarily by hand. Because of the relative scarcity of highly parallel applications and the expense of highly parallel systems, there was limited opportunity for exhaustive performance experimentation. Today, however, single-chip, massively data-parallel systems such as the NVIDIA GeForce 8 Series GPUs are available for approximately one U.S. dollar per single-precision GFLOP, several orders of magnitude less expensive than supercomputers a decade ago. Other architectures, such as the IBM Cell Broadband Engine [1], Ageia PhysX processor [2], and custom processor designs [3], also provide significant speedup to applications that map well to those architectures. For the past few years, developers have been using these relatively low-cost systems to perform work that would otherwise take a large compute cluster of traditional microprocessor systems to accomplish.

Unfortunately, the level of effort and expertise required to maximize application performance on these kinds of systems has not significantly decreased, and if anything has increased. There are several reasons for this issue:

1. The appropriate granularity of parallelism for the target architecture may not be the "natural" one in which the developer conceived the problem. This difference is particularly true for developers who are used to developing applications for clustered systems. It may be necessary to change an application's computation granularity in order to improve its performance on the system. The burden of finding the appropriate granularity is generally left to the application developer, who must sample across a wide range of points in the configuration space to find a reasonably good configuration. Because granularity selection is generally done early in the software design process, an application developer who does not initially try or accommodate for multiple granularities runs the risk of being trapped at a local performance maximum when iteratively optimizing the application via a greedy approach.

2. The local memory available to each processing element in the system is very restricted compared to older compute clusters. The cost of spilling data from low-latency memories to larger, higher latency memories is significant. This cost is particularly high when those memories are on a separate die from the

processing units. Even with intimate knowledge of both application and architecture, an application developer must experiment with configurations to see which map well to the architecture. Because of the increasing complexity of both applications and architectures, the accuracy of an application developer's knowledge will generally be flawed during initial development and take substantial time to develop. In addition, programs often change over time to include new application features, changing the local memory usage amount or patterns.

3. Different product lines or even successive generations of the same product line may require reapplication of the optimization process to take advantage of the new system. New features and capabilities will change which configurations provide maximal performance. These features also may not be initially understood by application developers.

4. Applications may need to be mapped to multiple, differing processing platforms to determine the one that provides the most cost-effective performance for the product needs and goals. Although the embedded space has had the same issue for some time, it will be present in the desktop and scientific computing spaces until a particular paradigm establishes dominance.

The common theme is that of rapid, nonintuitive or unexpected changes in the systems that an application developer targets. Since the architecture and applications are continuously changing, application developers cannot be expected to consistently find a near-optimal configuration of a substantial application. Furthermore, traditional compilation cannot be expected to find a near-optimal configuration in the short term, as compiler writers will need a significant amount of time and effort to learn how to use the features of the target architecture and create processes and algorithms that optimize many applications well.

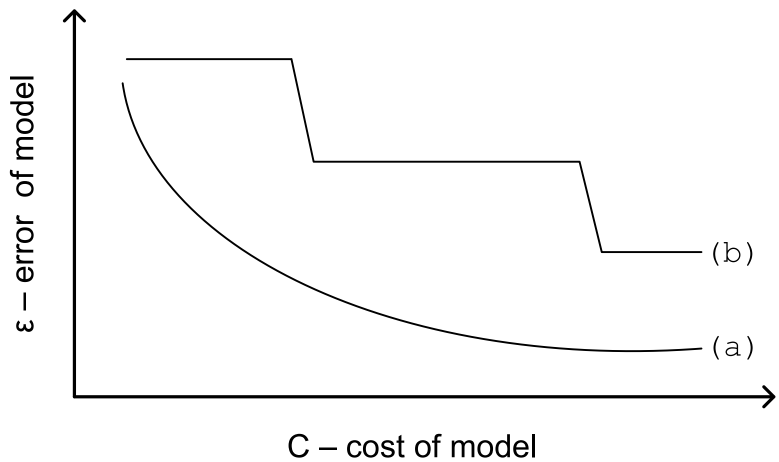


Figure 1.1: Examples of modeling spaces.

1.2 Architectural Models

In engineering and the sciences, a practitioner has a choice of models from which to choose. Figure 1.1 depicts abstract representations of the space of models for a domain. The common tradeoff made when selecting a model is the error of the model relative to the cost of using that model. Ideally the progression will be relatively smooth and a clear choice is evident for user needs, as shown with line `(a)`. For compilers that target single-core superscalar processors, a relatively cheap architectural model consisting primarily of instruction stream efficiency has been effective for many years because architectural features and capabilities reduce the impact of other potential performance factors. Compilers for Intel Itanium processors, on the other hand, must be more complex to provide high performance, since the architecture does not easily reduce to a few high-order concerns [4].

Many model spaces are generally not smooth, looking more like line `(b)` in Figure 1.1. The reason for this staircase-like structure is that multiple factors, each incurring significant cost, must be integrated before a significant reduction in modeling error is achieved. First-order concerns are relatively inexpensive to model but are not very usable for predicting behavior and performance. Second-order concerns are more accurate but incur substantial usage costs. A low-error model is actual hardware or a cycle-accurate simulator of the system.

I believe that line `(b)` in Figure 1.1 more closely resembles the model space for the emerging class of single-chip many-core processors. In some cases the error will *rise* with higher costs until a critical modeling mass is achieved, which means that a particular model may be less correct than one that is less precise and has lower cost. For example, measuring performance with the number of instructions executed may be misleading if the architecture may take a different number of cycles to execute each one: a version of a program with more low-latency instructions may have higher performance than another version with fewer high-latency instructions. Thus, application and compiler developers run the risk of using an architectural model that has higher cost and less accuracy than a less precise model. This issue is one of the more significant problems in attempting to optimize applications for architectures like the GeForce 8 Series GPUs.

1.3 Contributions and Organization

The purpose of this dissertation is to establish the systematic principles and discuss automatable tools that can be used to optimize programs for the emerging class of single-chip, data-parallel, many-core architectures. A few principles can be used to guide initial mapping of applications to the system and achieve reasonable performance. However, I believe iterative optimization techniques applied to these systems will be unable to keep pace with rapidly changing technology and often leave significant performance unrealized due to the nature of the architecture. The small size of many kernels and the limited number of performance-affecting optimizations for each kernel enables an alternate approach developed to find near-optimal configurations of applications for this class of architectures, which I term *optimization carving*.[1]

Instead of assuming that the architectural and program models have little error, optimization carving uses metrics to select multiple configurations that maximize metrics modeling high-order concerns. These are likely to be local performance maxima in the optimization space, but it is probable that one of them is the global maximum. The configurations are then executed with typical input data to determine the one with the highest performance. It differs from an exhaustive space search in that it only varies those parameters that can significantly change performance and does not execute configurations that are unlikely to be local maxima. As long as the ranking and modeling of concerns is reasonably correct, the technique is likely to select a good program configuration. Optimization carving can also be used as a tool to validate model assumptions about the system, similar to prior work [5].

The contributions of this work are as follows:

1. Characterization of the NVIDIA GeForce 8 series architecture and the ability of applications with certain characteristics to map well to the architecture. This information is presented in Chapter 2.

2. Presentation of performance and optimization principles for the GeForce 8 series in the beginning of Chapter 3. I begin with basic principles that apply to all computer architectures and then discuss how they should be observed on the NVIDIA GeForce 8 Series. Because of the wealth of execution resources and the long latencies to memory, programs must be decomposed into many threads in order to utilize the hardware well. Developers must also utilize the local memories to achieve good performance, as data local-

[1]The word "carve" is used in the context of sculpting stone, where unwanted portions are removed, leaving the finished product behind.

ity is not enabled by default in the programming model. I use a matrix multiplication kernel to demonstrate adherence to the principles in finding a high-performance kernel configuration.

3. Categorization of optimizations based on their effects on applications executing on the GeForce 8 Series in the latter part of Chapter 3. Some of the categories are optimizations that reduce memory bandwidth usage or improve the efficiency of executing code. I discuss explicit examples of optimizations within each category. I also discuss how optimizations may interact with each other in ways that may be unexpected to application or compiler developers. The major side effect of many optimizations is the use of additional registers, which may reduce thread-level parallelism (TLP).

4. Presentation of the performance of a suite of applications mapped to the GeForce 8 Series, in Chapter 4. Several applications are studied in further depth with their performance effects explained. I also discuss the optimizations which have the most significant effects on performance, specifically memory optimizations, control of register usage to maintain TLP, and balancing TLP with per-thread performance. This is presented in Chapter 4.

5. Introduction of optimization carving and discussion of its use for optimizing applications executing on the GeForce 8 Series processors in Chapter 5. I assert that it is generally not possible for application developers or compilers to find the best configuration through an iterative process, particularly because they cannot control part of the code generation process on the GeForce 8 Series. As an alternative approach, I propose optimization carving, which begins with a large optimization space and then prunes it to find configurations which are likely to be the best one.

6. Presentation of results of optimization carving for one target architecture, the GeForce 8800 GTX, in Chapter 6. I experiment with different metric calculations that can be used when applying the technique and analyze why they do better or worse than the original calculation. Results are also compared to random sampling of the space.

Related work is discussed in Chapter 7. I finish with concluding remarks in Chapter 8.

CHAPTER 2

ARCHITECTURE

This work uses the GeForce 8800 GTX[1] graphics processing unit (GPU) as the hardware target for my study. Previous generations of GPUs consisted of a highly specialized pipeline to which any nongraphics application must be mapped through graphics application programming interfaces (APIs). This process often involves reworking application algorithms to fit the API's model. The GeForce 8800, however, consists of a large set of processor cores that can directly address a global memory. Although still specialized for graphics processing, this general addressability allows for a more general and flexible programming model than previous GPU generations and allows developers to easily implement a wider variety of data-parallel kernels. It shares the same traits with other contemporary single-chip, many-core processors: a large number of processing units that can simultaneously execute independent threads in parallel, a limited amount of local memory per execution unit, and limited off-chip bandwidth compared to the available execution resources.

In this chapter, I discuss the programming model provided by NVIDIA, the Compute Unified Device Architecture (CUDA) and the tools provided to support it. I then discuss the architecture of the GeForce 8800 and how its structure creates an affinity for certain kinds of applications. A more complete description of these can be found in [6, 7]. I also discuss some of the complexity that may arise

[1]There are presently several versions of the GeForce 8800 GPU. References to GeForce 8800 are implied to be the GTX model in this work.

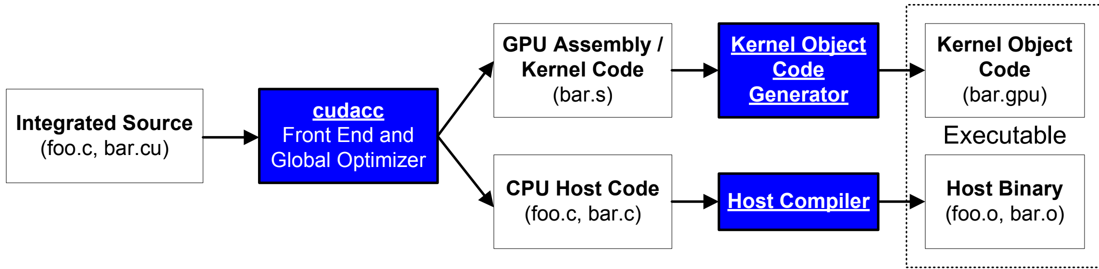


Figure 2.1: CUDA compilation flow.

when mapping applications to the architecture and optimizing them. Finally, I describe some of the software tools provided by NVIDIA, which are useful during the optimization process.

2.1 Threading Model

The CUDA programming model is ANSI C extended by several keywords and constructs. The GPU is treated as a coprocessor that executes data-parallel kernel code. The developer supplies a single source program encompassing both host (CPU) and kernel (GPU) code. These are separated and compiled as shown in Figure 2.1. Each CUDA program consists of multiple phases that are executed on either the CPU or the GPU. The phases that exhibit little or no data parallelism are implemented in host code, which is expressed in ANSI C and compiled with the host C compiler as shown in Figure 2.1. The phases that exhibit rich data parallelism are implemented as kernel functions in the device code. These kernel functions are compiled by the NVIDIA CUDA C compiler (cudacc) and the GPU object code generator.

A kernel function defines the code in single-program multiple-data (SPMD) [8] style, to be executed by each thread invoked for a data-parallel phase. These kernels typically comprise thousands to millions of lightweight threads per invocation. Creating enough threads to fully utilize the hardware often requires a

```
  float * gpu_A, float * gpu_B, float * gpu_C;

  // Allocate GPU input and output matrices
1 cudaMalloc((void **) &gpu_A, mem_size_A);
2 cudaMalloc((void **) &gpu_B, mem_size_B);
3 cudaMalloc((void **) &gpu_C, mem_size_C);

  // Copy input matrices from host to device
4 cudaMemcpy(gpu_A, host_A, mem_size_A,
             cudaMemcpyHostToDevice);
5 cudaMemcpy(gpu_B, host_B, mem_size_B,
             cudaMemcpyHostToDevice);

  // Setup execution parameters
6 dim3 blocks(16, 16);
7 dim3 grid(256, 256);

  // Execute the kernel
8 matrixMul<<< grid, blocks >>>(gpu_C, gpu_A, gpu_B);

// Copy result back to host
9 cudaMemcpy(host_C, gpu_C, mem_size_C,
             cudaMemcpyDeviceToHost);
```

(a) Host code

```
__global__ void
matrixMul (float* C, float* A, float* B)
{
  // Calculate index of the first element of A
  int indexA = 16 * blockIdx.y + threadIdx.y
  …

  // Initialize result variable to 0
  float Ctemp = 0;

  for (i = 0; i < widthA; i++)
    {
      Ctemp += A[indexA] * B[indexB];
      indexA++;
      indexB += widthB;
    }
  C[indexC] = Ctemp;
}
```

(b) Kernel code

Figure 2.2: Matrix multiplication example.

fine-grained decomposition of work. For example, each element of a result array might be computed by a separate thread if the computation for each element is relatively independent. It should be noted that fine-grained decomposition may create inter-thread instruction redundancy.

Host code initiates kernel execution by using CUDA-specific function call syntax. There are several restrictions on kernel functions: they cannot use recursion or static variable declarations, and must have a nonvariable number of arguments. The host code transfers data to and from the GPU's global memory using API calls.

I use a dense matrix multiplication example, operating on two 4096x4096 matrices, to illustrate the CUDA threading model. This example is shown in Figure 2.2. In this example, each thread calculates one element of the product matrix. This calculation involves a dot product of a row of the first input array and a column of the second input array, shown as the kernel function `matrixMul()` in Figure 2.2(b). Each thread begins with a pointer to a row of the first input matrix and a pointer to a column of the second input matrix. It calculates a dot product and writes the value into the corresponding element of the output array.

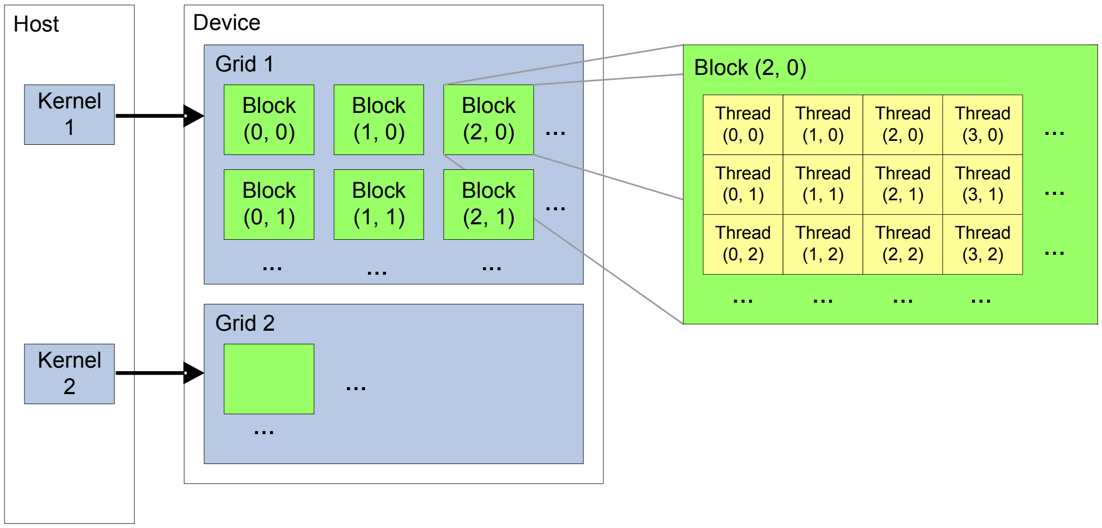


Figure 2.3: CUDA thread organization.
2D thread blocks are shown here; thread blocks can be up to three dimensions.

Threads executing on the GeForce 8800 are organized into a three-level hierarchy, depicted in Figure 2.3. At the highest level, all threads in a data-parallel execution phase form a *grid*. The matrix multiplication code in Figure 2.2 forms a grid consisting of a total of 2^{24} threads.

Each grid consists of many *thread blocks*. All thread blocks in a grid have the same number of threads. A grid can be at most $2^{16} - 1$ blocks in either of two dimensions, and each block has unique coordinates. On line 7 of Figure 2.2, the grid performing the matrix multiplication consists of 256 thread blocks in each dimension, for 2^{16} thread blocks in total.

Each thread block is a three-dimensional array of *threads*, whose dimensions are also explicitly defined by the application developer. These are scheduled as a unit on the hardware. Threads have unique coordinates within their thread block and up to 512 threads can exist in a thread block. On line 6 of Figure 2.2, each thread block consists of 16x16 threads; the third dimension is not used.

Threads in a thread block can share data through a low-latency, on-chip shared memory and can perform barrier synchronization by invoking the `__syncthreads` primitive. This ability to coordinate the execution of threads distinguishes CUDA's programming model from the shader programming of previous GPGPU models; I show how coordination can boost performance in the next chapter. It is important to note that the very local nature of the barrier synchronization enables it to be performed by a fast hardware mechanism and probably does not incur more than a few processor cycles once all threads in a thread block reach the barrier.

Barrier synchronization within thread blocks is the only supported synchronization on this architecture. No options for synchronization at any other level are explicitly supported; for example, barrier synchronization across thread blocks can only be safely accomplished by terminating a kernel. The only possible method of communication between threads in separate thread blocks is to use a "mailbox" memory location in the global address space, assuming that the developer can ensure that both sending and receiving threads are simultaneously scheduled on the device. The lack of synchronization mechanisms beyond barrier synchronizations limits the ways in which parallelism can be expressed in CUDA, but it also makes it easier for novice parallel programmers to reason about the correctness of their code.

When the host code invokes a kernel, it sets the grid and thread block dimensions by passing them as parameters. On lines 6-7 of Figure 2.2(a), two structures of type `dim3` are declared: the first is for thread blocks, which are defined as 16×16 groups of threads. The second is for the grid, which consists of 256×256 thread blocks. The following line of code invokes the kernel. Kernel code is shown in Figure 2.2(b). First, each thread calculates the starting positions in the input matrices based on its unique block and thread coordinates. It then iterates through a loop to calculate the result and store it to memory.

The multiple memory spaces of a GPU's memory system are exposed by CUDA to application developers. The host's memory is separate from the device's memory. The host uses API calls to allocate memory on the GPU and transfer data between host and device memories. Different memory spaces on the device are also separate from one another. Major speedup is possible when certain application characteristics can utilize the properties of local memories, but the developer bears the responsibility of selecting the appropriate data placement and layout for a given application. This selection requires knowledge of the characteristics of each memory, as is explained in the next section.

Finally, the hardware groups threads into *warps* of up to 32 threads. The threads in a warp operate in lockstep, with a single instruction being issued to eight threads per cycle for four cycles. Warps are not specified in the CUDA model, but are significant for optimization and performance purposes. Some of the key points are:

- Thread blocks that do not consist of an integral number of warps will leave execution resources idle.
- All threads in a warp issue in SIMD (single-instruction multiple-data) fashion, and a performance penalty will be incurred if threads in the same warp follow different control paths. Predication support can mitigate some of this penalty, but the NVIDIA compiler does not aggressively predicate code.
- Memory accesses of threads in the same warp can interact in synergistic or antagonistic ways. Warps should be organized so that threads in the same warp do not have bank or port conflicts.
- Because threads in a warp operate in lockstep, there are guarantees on memory access that enable communication through shared memory that would be unsafe otherwise because of race conditions. One optimization which capitalizes on this property is the use of a single warp to perform

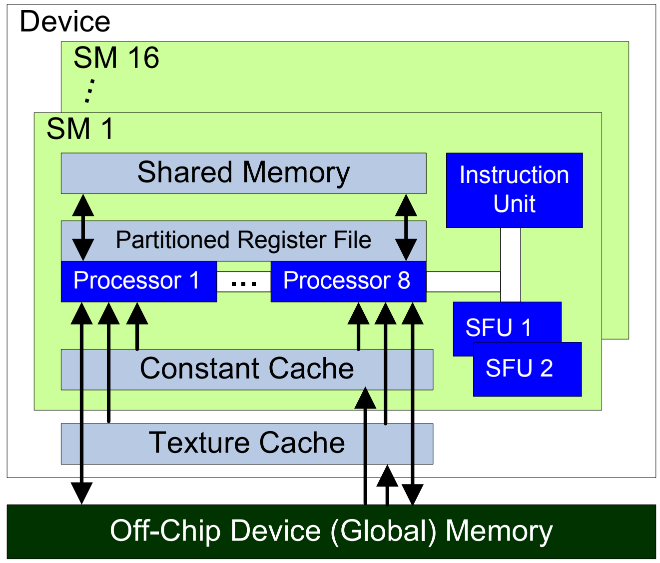


Figure 2.4: Organization of the GeForce 8800.

a reduction that accesses values in a tree pattern without synchronization. Synchronization is unnecessary in this case because the use of a single warp creates ordering guarantees among the threads in the warp. Optimizations using this property are not covered in depth in this work.

These issues are discussed further in the next section.

2.2 Microarchitecture

This section describes the microarchitecture of the GeForce 8800. I first discuss the execution resources of the system. Because the different memories play a major role in optimization, these are described separately.

2.2.1 Execution resources

Figure 2.4 depicts the microarchitecture of the GeForce 8800. The GPU consists of 16 *streaming multiprocessors* (SMs), each containing eight *streaming processors* (SPs), or processor cores, running at 1.35 GHz. There is a single instruction issue unit per SM. Each SP has one 32-bit, single-precision floating-point (FP), multiply-add arithmetic unit that can also perform 32-bit integer arithmetic operations. Additionally, each SM has two special function units (SFUs) that execute more complex FP operations such as reciprocal square root, sine, and cosine; two of these operations can be issued per processor cycle, per SM. The arithmetic units and the SFUs are fully pipelined, yielding 388.8 GFLOPS ($16\,\mathrm{SM} * 18\,\mathrm{FLOP/SM} * 1.35\,\mathrm{GHz}$) of peak theoretical performance for the GPU.

Each SM has 8192 registers that are dynamically partitioned among the threads running on it. The registers are contained within a banked register file, the workings of which have not been explained in detail by NVIDIA. The latency of the register file is generally assumed to be two SP cycles.

As explained in the previous section, threads on an SM are grouped into bundles of 32 threads called warps. Warps are formed from continuous sections of threads in a thread block: the first 32 threads in a block form the first warp, etc. A scoreboard indicates when all of a warp's operands are ready for execution. The SM then executes the same instruction for the 32 threads in the warp. An SM issues only one instruction at a time for all threads in a warp; when threads in a warp take different control paths, it is assumed that multiple passes with suppression of threads on divergent paths are required to complete execution [9]. Thus, execution is slowed as much as if each thread had executed all control paths. It is generally desirable to group threads to avoid this situation, if possible. Also, if the number of threads per thread block is not evenly divisible by the warp size, any remaining issue slots are wasted. Due to this design, the architecture naturally favors applications with little divergent control flow; kernels with large regions of data-dependent control flow are generally unsuitable for this architecture. Knowledge of warps also helps in avoiding memory bank conflicts, which are discussed in the next subsection.

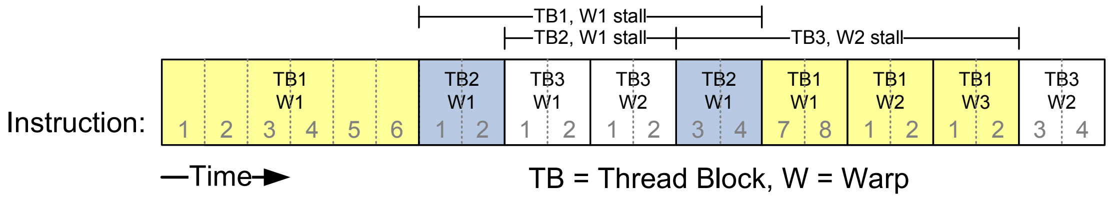


Figure 2.5: CUDA thread scheduling within an SM.
Warps are interleaved to tolerate intrawarp stalls.

SMs can perform zero-overhead scheduling to interleave warps on an instruction-by-instruction basis to hide the latency of global memory accesses and long-latency arithmetic operations. When one warp stalls, the SM can quickly switch to a ready warp in the same thread block or a ready warp in any other thread block assigned to the SM. This ability to utilize thread-level parallelism allows an SM to tolerate long memory latencies by executing work from other threads/warps while a warp is stalled. The dynamic warp scheduler reportedly adopts a round-robin schedule among ready warps to ensure fairness.

Figure 2.5 shows an example of thread scheduling on the architecture. Warp 1 from thread block 1 is able to execute the first six instructions without stalling. Because its operands for instruction 7 are not ready, the SM switches execution to warp 1 from thread block 2. Similarly, that warp's operands for the third instruction are not ready, so the SM switches to warp 1 from thread block 3. Execution on the SM stalls only if none of its resident warps have ready operands. Because warps are independent with the exception of synchronizations between those in the same thread block, there is relatively high execution scheduling freedom in many applications.

Each SM supports a maximum of 768 simultaneously active thread contexts. An integral number of up to eight thread blocks are scheduled to an SM at any time, to the limit imposed by resource constraints. When scheduling a thread block, the hardware automatically allocates the necessary amount of several hardware resources, primarily thread contexts, shared memory, and registers. When optimizing kernel code, developers need to be aware of how these limits affect the number of parallel threads that can run on the device. Optimizations may have negative effects in some cases because a small change in resource use can cause fewer thread blocks, and thus many fewer threads, to be simultaneously executed. This issue is discussed further in the next chapter.

In addition to these explicit execution resources, the texture cache also can be programmed to perform interpolation and other operations useful in graphics applications. Specific uses of this feature are discussed in Chapter 4.

2.2.2 Memories

The GeForce 8800 has 86.4 GB/s of bandwidth to its off-chip memory. Nevertheless, with computational resources supporting nearly 400 GFLOPS of peak performance and each FP instruction operating on up to 12 bytes of source data, applications can easily saturate that bandwidth. In the worst case, the GeForce 8800 can demand 2.25 TB/s (1.35 GHz * [128 multiply-add operations * 12 bytes + 32 SFU operations * 4 bytes]) of memory bandwidth. Therefore, as depicted in Figure 2.4 and described in Table 2.1, the GeForce 8800 has several on-chip memories that can be used to exploit an application's data locality and data sharing to reduce the demand for off-chip memory bandwidth.

The GPU's memories are highly specialized and have different latencies and throughput limitations. Memories furnish fast or efficient access only for particular patterns of memory references. Poor use of local memories, or an intrinsically poor fit between an application's memory requirements and the architecture's available memory, can result in poor application performance on the architecture. On the other hand, applications that can effectively use the local memories can

Table 2.1: Properties of GeForce 8800 Memories

Memory	Location	Size	Latency	Read-Only	Description
Global	off-chip	768 MB total	200-300 cycles	no	Large DRAM. All data reside here at the beginning of kernel execution. Directly addressable from a kernel using pointers. Backing store for constant and texture memories. Used more efficiently when multiple threads simultaneously access contiguous elements of memory, enabling the hardware to coalesce memory accesses to the same DRAM page.
Shared	on-chip	16 KB per SM	$\simeq$register latency	no	Local scratchpad that can be shared among threads in a thread block. Organized into 16 banks. It is often possible to organize both threads and data so that bank conflicts seldom or never occur.
Constant	on-chip cache	64 KB total	$\simeq$register latency	yes	8 KB cache per SM, with data originally residing in global memory. The 64 KB limit is set by the programming model. Often used for lookup tables. The cache is single-ported, so simultaneous requests within an SM must be to the same address or delays will occur.
Texture	on-chip cache	up to global	>100 cycles	yes	16 KB cache per two SMs, with data originally residing in global memory. Capitalizes on 2D locality. Can perform hardware interpolation and have configurable returned-value behavior at the edges of textures, both of which are useful in certain applications such as video encoders.
Local	off-chip	up to global	same as global	no	Space for register spilling, etc.

achieve superlinear speedups compared to traditional processors. Developers must use their understanding of the memory system to structure both data and kernel code for high performance.

Global memory is a large, long-latency memory that exists as off-chip DRAM and is the default storage location. Kernel output data must be written to global memory in order to be readable after kernel termination. When acting as a graphics processor, most accesses in the architecture are to textures or lookup tables. Because of this, global memory is optimized to provide data in aligned, contiguous, 16-word lines. This pattern is termed a *coalesced* access.

Coalesced accesses can be achieved by designing kernels so that each thread in a half-warp accesses the corresponding word, in ascending thread order, in an aligned, 16-word section of data. The memory system detects the threads jointly accessing the complete line and aggregates the accesses into a single access. This optimization is termed *global memory coalescing*. Other patterns can achieve only a small fraction of the effective memory bandwidth of coalesced accesses. Unless memory requests are relatively rare, application performance will generally be limited by memory bandwidth when memory accesses are not coalesced.

Attempting to coalesce global memory accesses may require major changes to data structures in both host and device code. For example, data organized as an array of structures will cause threads to load nonunit-strided addresses when they access fields of adjacent structures. Figure 2.6 shows an example of this, with threads accessing a field `x` of contiguous elements in an array of structures. Loads to non-contiguous memory addresses does not enable global memory coalescing, as shown in Figure 2.6(a): a separate memory request is required for each thread. Reorganizing the data as separate arrays, as is done in Figure 2.6(b), will place these fields next to each other and naturally set up global memory coalescing if the alignment requirements are satisfied.

The *shared memory* is a low-latency scratchpad memory in each SM that is useful for data that is either written and reused or shared among threads in a thread block. In CUDA kernel code, shared variables (generally arrays) are declared with specific sizes[2] and an instance of the variable exists for each thread block. This memory is accessible only during execution of its thread block and its contents are discarded after the thread block completes execution. Data must be loaded to and from shared memory similar to global memory: transfers from

[2]Support for shared variables with sizes determined at run time exists but is not used in this work.

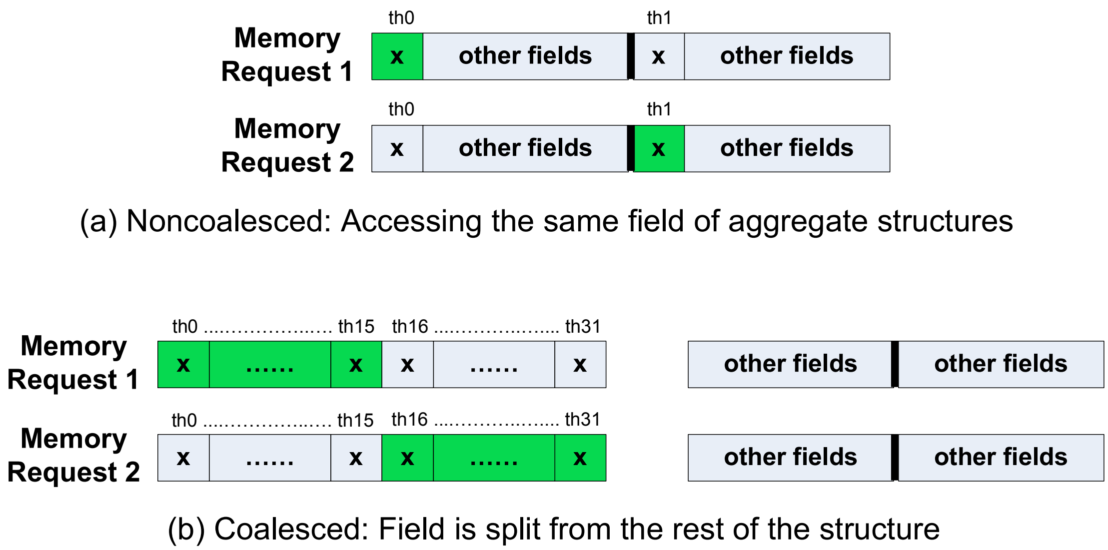


Figure 2.6: Contrasting noncoalesced and coalesced access patterns.

global memory to shared memory must go through the registers of individual threads. The shared memory consists of 16 banks; simultaneous requests to different addresses in the same bank will conflict and cause a stall. Accesses to the same address in the same bank are broadcast to requesting threads.

The use of scratchpad memories has been common in embedded systems. However, shared memory has one major capability beyond those, which is enabling reuse of data across the entire thread block with the assistance of synchronization. For example, if each thread in a thread block reads a portion of a block of data, they can cooperatively load the values of the data into registers and then store them into the shared memory for use by the entire thread block. An example of this within a matrix multiplication kernel is visited in the next chapter.

There are two read-only data caches. The first, the *constant cache*, holds a working set of the constant memory data. It has the ability to broadcast the same value to all threads in a warp, making it useful for lookup tables and other data where data and memory access patterns can be manipulated to fit this pattern. This ability provides an effective memory bandwidth many times greater than that provided to global DRAM. The applications with the most dramatic speedups in Chapter 4 are the ones that can take advantage of this capability. When threads do not load the same value from the cache, it takes several cycles to provide all requested values.

The *texture cache* is the other data cache and holds a working set of the texture memory data. It has a much longer latency than the constant cache but is useful for data with 2D spatial locality. It does not suffer as greatly as the constant cache from irregular memory read patterns. The texture cache can also interpolate values and provide boundary values for 2D arrays, which is a useful feature for graphics applications. Use of the texture cache requires special API calls with the input texture and coordinates passed as parameters, while addressing the constant cache is done implicitly by declaring a data structure with the `__constant__` keyword.

Memory bank and port conflicts are issues that must be addressed when developing applications for this architecture. In general, simultaneous accesses to the same location are desirable for the constant cache and individual banks of the shared memory. If threads in the same warp load different locations, the memory will take additional cycles to satisfy the request. For example, assume that the primary datum of an application is an array of structures, each 16 words long, which is placed in shared memory. Threads in a warp, accessing the same field of successive objects, will attempt to access different locations within the same bank and therefore conflict. This conflict can be avoided with data repartitioning; e.g., two separate arrays, one of 15-word size and the other 1-word, will not result in conflicts when the same field is accessed for different objects. Data padding is another technique to address this problem.

Unlike bank conflicts in shared memory, reorganizing the data in global memory is generally more difficult because it requires either data structure changes in

Table 2.2: Constraints of GeForce 8800 and CUDA

Resource or Configuration Parameter	Limit
Threads per SM	768 threads
Thread Blocks per SM	8 blocks
32-bit Registers per SM	8192 registers
Shared Memory per SM	16 384 bytes
Threads per Thread Block	512 threads

the host code portions of the application or a separate data reorganization phase, either in host or kernel code. One possible work-around is to load data in bulk into shared memory and then use it as necessary, even if little reuse or no use occurs for some of the loaded data. Data structures of new applications whose execution is primarily in parallel kernels should be designed and constructed to avoid these issues.

2.3 Architectural Interactions

Accurately predicting the effects of one or more compiler optimizations on the performance of a CUDA kernel is often quite difficult, largely because of interactions among the architectural constraints listed in Table 2.2. Many optimizations that improve the performance of an individual thread tend to increase a thread's resource usage. However, as each thread's resource usage increases, the total number of threads that can occupy an SM decreases. Occasionally this decrease in thread count occurs in a dramatic fashion because threads are assigned to an SM at the granularity of thread blocks. In short, there is often a tradeoff between the performance of individual threads and the thread-level parallelism on an SM.

For example, consider an application that uses 256 threads per block, 10 registers per thread, and 4 KB of shared memory per thread block. This application can schedule 3 thread blocks and 768 threads on each SM. However, an optimization that increases each thread's register usage from 10 to 11 (an increase of only 10%) will decrease the number of thread blocks per SM from three to two, which decreases the number of threads on an SM by 33%. In this case, the GeForce 8800 can only assign two thread blocks (512 threads) to an SM because a third block would increase the number of threads to 768, with total register usage of 8448 (768 threads * 11 registers/thread), above the 8192 registers per SM available. In contrast, an optimization that increases each thread block's shared memory usage by 1 KB (an increase of 25%) does not decrease the number of thread blocks per SM. The optimization space is inherently nonlinear, with register usage in particular often triggering changes because many optimizations affect it.

It is important to note that additional optimizations or a change in the granularity of work that each thread performs may eventually obtain more performance, even though an initial change significantly reduced performance. This possibility makes iterative optimization uncertain and difficult, since one cannot be certain that a seemingly high-performance configuration is not a local performance maximum. Chapter 3 goes into more detail on the effects of the various optimizations.

2.4 Software Environment and Support

For CUDA compilation, NVIDIA provides a compiler wrapper called *nvcc* that handles all parts of the compilation flow, including linking host and kernel binaries. The compiler also supports several options that programmers can use to debug kernels and to gain intuition on their performance. Two flags are especially useful: `-ptx` and `-cubin`. The amount of time it takes to run nvcc with these flags is much shorter than actual compilation because only the kernel code is processed when using the flags.

Nvcc compiles kernel code to an assembly-like representation termed *PTX*. PTX is normally encoded in an object file for consumption by the CUDA runtime, which processes this code, performs further optimization such as instruction scheduling, and generates hardware-specific code for execution. The purpose of PTX is to provide code portability and to abstract the exact workings of the architecture away from application developers.

The `-ptx` flag for nvcc outputs PTX code in a human-readable text format. Although PTX is not the exact code that is executed on the hardware, it often gives insights into why performance degrades or improves after a manual optimization is applied. In particular, information such as instruction count, instruction mix, and a rough idea of scheduling can be reliably utilized. For example, unrolling a loop with strided memory accesses creates successive operations that operate at different offsets from a base address. PTX shows the reduction in address calculations that results from this transformation.

The CUDA runtime that generates executable machine code reschedules instructions and allocates registers. The runtime introduces an uncontrollable factor during program optimization and makes the effects of optimizations on local resource usage less predictable. In addition, if the PTX code uses more registers than are physically available, the kernel will fail to execute.

The `-cubin` flag outputs the resource usage of GPU kernel code, including the shared memory used per thread block and registers used per thread. The resource usage is critical to understanding the performance of the code because each SM runs the number of thread blocks that can fit into the local resources. As previously discussed, a small change in code can result in resource usage that changes the number of thread blocks executing on an SM, which can significantly impact performance. The information provided by `-cubin` is used in this work to calculate the number of thread blocks that can simultaneously reside on each SM.

CHAPTER 3

OPTIMIZATIONS

This chapter discusses optimization principles and strategies for data-parallel, many-core processors, such as the GeForce 8800. I first explain the principles that should be kept in mind when optimizing applications for the architecture. I then apply these principles to a matrix multiplication kernel. The chapter closes with descriptions of specific categories of optimization with, concrete examples.

3.1 Performance Principles

There are three basic principles to consider when optimizing an application for any platform. Performance of an application on a highly parallel system is generally measured by its throughput of useful instructions. For applications with good performance on the GeForce 8800, these are generally floating-point operations. *The floating-point throughput of an application depends on the percentage of its instructions that are floating-point operations.* The GPU is capable of issuing 172.8 billion operations per second on the SPs. These include fused multiply-add operations, which are counted as two operations for throughput calculations. If a quarter of an application's instruction mix are fused multiply-adds, then its performance can be at most 2 * 1/4 FP per op * 172.8 billion ops per second = 86.4 GFLOPS. This performance can be achieved when the SPs are fully utilized. Assuming that utilization of execution resources is not reduced, decreasing the number of nonfloating-point operations is beneficial to performance. Transformations that reduce the proportion of these instructions are discussed in the next sections.

Periods in which the execution resources are not utilized, due to stalls, detract from performance. A stall generally means that there are no threads whose operands are available for execution. Because of the long (and ever increasing) latency to off-chip memory, the primary utilization concern often is ensuring that enough independent work exists to be executed while threads are stalled on global memory accesses.

As noted in the previous chapter, there is significant scheduling freedom on the GeForce 8800. First, an SM can continue to execute independent instructions after a long-latency instruction within the same warp. Second, different warps in the same thread block can execute while a warp is stalled on a long-latency operation. Third, up to eight independent thread blocks can be simultaneously assigned to an SM. A wide variety of techniques enhance one or more of these aspects of schedulability, although they can incur an instruction efficiency cost.

Lastly, *global memory bandwidth can limit the throughput of the system.* The 86.4 GB/s global memory bandwidth on the GeForce 8800 must feed the 388.8 GFLOPS of execution resources, so only a small fraction of instructions can be global memory accesses if the hardware is to be utilized effectively. Chapter 4 shows examples of optimized applications that are still bandwidth-limited on the GeForce 8800. This issue will continue to be a first-order concern for many-core processors, as execution resources can increase at a near-exponential rate while bandwidth into and out of a chip will grow at a much slower rate.

These principles apply to any form of processor. The methods of following these principles, however, can differ between architectures. For the GeForce 8 Series and similar GPUs, the process for optimizing applications can be constructed from a list of five principles.

1. **Leverage zero-overhead thread scheduling to hide memory latency.** The latency of global memory is at least 200 cycles, requiring 1600 or more instructions to cover the latency for the eight SPs in each SM. The architecture provides zero-overhead warp scheduling to enable thread-level parallelism to supply instructions to hide this latency, whereas traditional processors have been limited to using instruction-level parallelism (ILP) and sometimes a small amount of TLP. The impact on program developers is that their applications may need to be decomposed to a much finer level than they previously practiced to provide a sufficient number of threads.

2. **Threads within a thread block can communicate via synchronization, but there is no built-in global communication mechanism for all threads.** From the hardware design perspective, this design avoids the need for virtualization of hardware resources, enables the execution of the same CUDA program across processor family members with a varying number of SMs, and makes the execution hardware relatively scalable. However, this limits the kinds of parallelism that can be utilized within a single kernel call, and affects which algorithms map well to the architecture.

3. **Optimize use of on-chip memory to reduce global memory bandwidth usage and redundant execution.** For most unoptimized applications the primary performance bottleneck is global memory bandwidth. The bandwidth limitation was recognized by researchers over a decade ago [10, 11], but mainly in the context of superscalar processors. With initial versions of applications, optimizations generally will not improve application performance unless the application uses bandwidth more efficiently as a result. Working memory within a group of cores consists primarily of a large register file and the shared memory. Shared memory's strength is enabling

the sharing of data among threads in a thread block. Texture and constant caches can also reduce the load on global memory bandwidth. Developers still have to control the number of registers and shared memory used per thread block, since using too many of these resources can reduce the number of thread blocks simultaneously running per SM, exposing memory stalls.

4. **Group threads to avoid SIMD penalties and memory port/bank conflicts.** CUDA is based on the SPMD model, but its current implementation on the GeForce 8800 imposes SIMD mode among threads in the same warp. Good performance requires that threads in a warp follow mainly the same control flow path during execution. In some algorithms, threads can be reorganized to avoid divergent control flow. Appropriate thread grouping can also preserve performance by avoiding memory port and bank conflicts, as well as enable global memory coalescing.

5. **Trade resource usage to improve utilization of hardware resources and efficiency of the instruction stream.** Because of the wide variety of execution resources, it is possible to use resources to free up demands on other resources and improve overall system performance. For example, the contents of registers can be spilled to shared or global memory to reduce the number of registers per thread and thus potentially increase concurrency on the system. Conversely, using more registers per thread can reduce the number of loads and stores executed per thread and increase instruction stream efficiency, although this shift may reduce the total number of threads simultaneously running on the system. Examples for specific applications are discussed in Section 4.3.

The next section illustrates how these principles enable efficient optimization for a simple kernel running on a many-core processor.

3.2 Optimization Example: Matrix Multiplication

This section applies the optimization principles described in the previous section to a matrix multiplication kernel. Matrix multiplication is a highly parallel algorithm with heavy data sharing across threads. To calculate an element of a result matrix, one takes the dot product of the corresponding row of the first input matrix and the column of the second input matrix. Figure 3.1 shows multiple versions of a matrix multiplication kernel, while Figure 3.2 depicts their memory access patterns. Experiments were performed on a 4096x4096 dense matrix. Starting values for `indexA`, `indexB`, and `indexC` are determined by block and thread coordinates. I use thread blocks consisting of 256 threads arranged in a 16x16 square.

Figure 3.1(a) shows a simple matrix multiplication kernel: each thread loops through a sequence that loads two values from global memory, multiplies them, and accumulates the value. This code provides 10.58 GFLOPS of performance. Before making any assumptions about the source of performance limitations, the developer can calculate the theoretical performance of the code assuming full SP utilization. There is one fused multiply-add out of eight operations in the inner loop, for an estimated potential throughput of 43.2 GFLOPS. Since several hundred threads are being executed per SM and all threads access global memory for their input data, the problem appears to be lack of available global memory bandwidth.

Although the algorithm has inherent data reuse, this implementation does not capitalize on that. All memory operations access global memory. One quarter of the operations executed during the loop are loads from off-chip memory, which would require a bandwidth of 173 GB/s (128 processor cores * 1/4 instructions * 4 B/instruction * 1.35 GHz) to fully utilize the instruction issue bandwidth, which

```
Ctemp = 0;

for (i = 0;
     i < widthA;
     i++)
  {
    Ctemp += A[indexA]
      * B[indexB];
    indexA++;
    indexB += widthB;
  }

C[indexC] = Ctemp;
```

(a) Initial Version

```
Ctemp = 0;
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  for (i = 0; i < 16; i++)
    {
      Ctemp += As[ty][i]
        * Bs[i][tx];
    }

  __syncthreads();
}
C[indexC] = Ctemp;
```

(b) Tiled, 16x16

```
Ctemp = 0;
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  Ctemp +=
     As[ty][0] * Bs[0][tx];
  ...
  Ctemp +=
     As[ty][15] * Bs[15][tx];

  __syncthreads();
}
C[indexC] = Ctemp;
```

(c) Complete Unroll

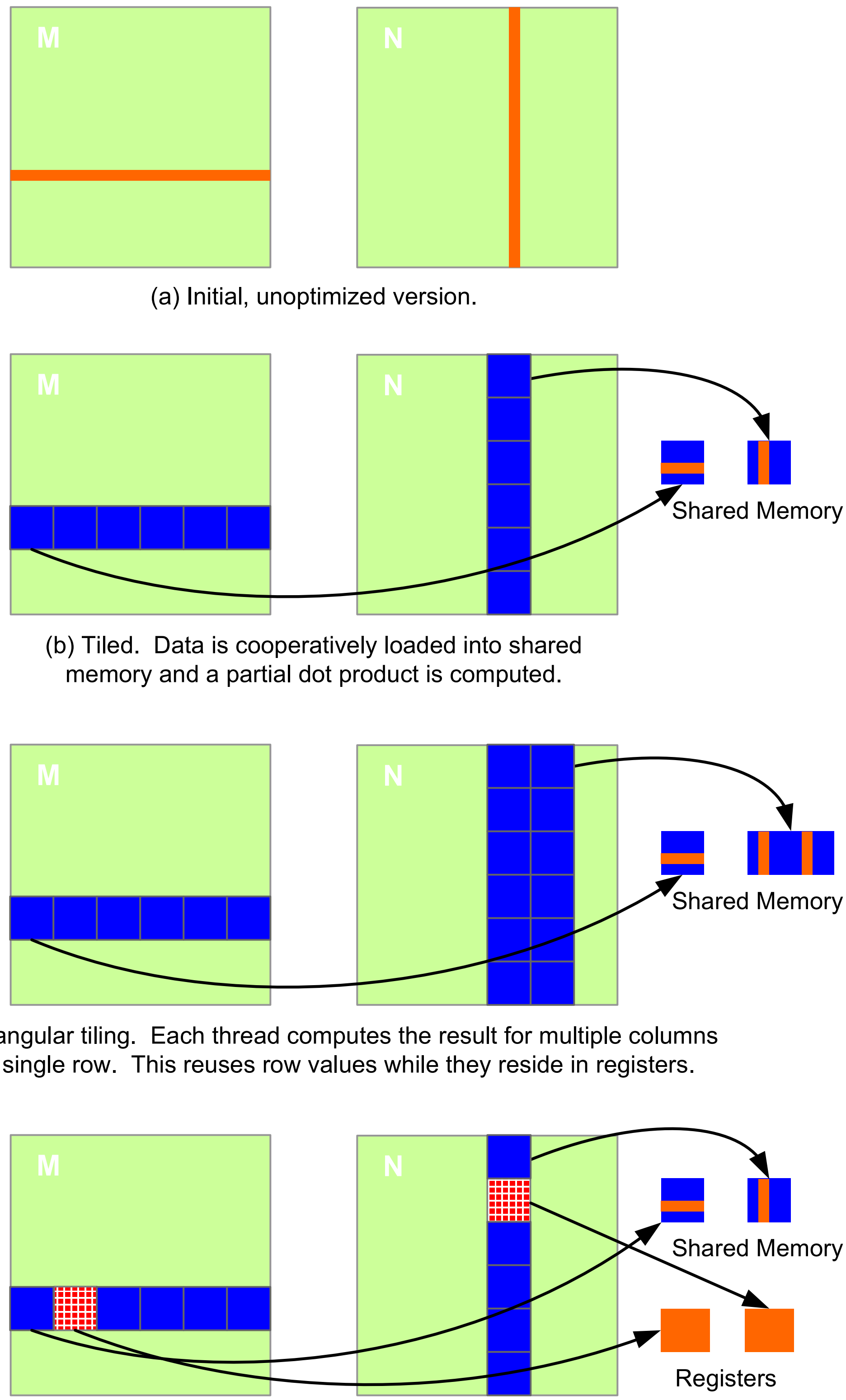
```
Ctemp = Dtemp = 0;
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][32];

  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  Bs[ty][tx+16] = B[indexB+16];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  for (i = 0; i < 16; i++)
    {
      Ctemp += As[ty][i]
        * Bs[i][tx];
      Dtemp += As[ty][i]
        * Bs[i][tx + 16];
    }

  __syncthreads();
}
C[indexC] = Ctemp;
C[indexC+16] = Dtemp;
```

(d) 1x2 Rectangular Tiling

```
a = A[indexA];
b = B[indexB];
Ctemp = 0;
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  As[ty][tx] = a;
  Bs[ty][tx] = b;
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  a = A[indexA];
  b = B[indexB];
  for (i = 0; i < 16; i++)
    {
      Ctemp += As[ty][i]
        * Bs[i][tx];
    }

  __syncthreads();
}
C[indexC] = Ctemp;
```

(e) Prefetching

Figure 3.1: Matrix multiplication optimization examples. Code differences from tiled version are shown in bold.



(a) Initial, unoptimized version.

(b) Tiled. Data is cooperatively loaded into shared memory and a partial dot product is computed.

(c) Rectangular tiling. Each thread computes the result for multiple columns with a single row. This reuses row values while they reside in registers.

(d) Prefetching. While the partial dot product is computed for one set of tiles, the next set is prefetched into registers.

Figure 3.2: Graphical depictions of the memory access patterns of different matrix multiplication versions.

is more than twice the available amount. In order to achieve better performance, a developer must reduce the number of accesses of global memory.

For a given application, there are three choices for local memories: texture cache, constant cache, and shared memory. The appropriate choice for matrix multiplication is the shared memory; constant cache is unsuitable because threads in a warp will simultaneously load different values for at least one input matrix, and the access patterns and amount of reuse favor shared memory over texture cache. By using memory tiling [12–15], the kernel's global memory accesses can be reduced by a factor equal to the length of a square tile. The choice of thread block is a major decision because larger tile sizes increase data sharing and thus global memory efficiency, but potentially reduce schedulability because fewer thread blocks can be executed per SM. On the GeForce 8800 there are a limited number of useful sizes because of the requirements for global memory coalescing, the need to have full warps for good SP utilization, and the maximum of 512 threads per thread block.

For matrix multiplication, I choose a thread block and memory tile size of 16x16 because the memory system coalesces accesses of 16 aligned, contiguous words, using off-chip bandwidth effectively. This configuration reduces the total number of global loads performed by each thread by a factor of 16. The code for this version is shown in Figure 3.1(b), with a graphical depiction in Figure 3.2(b). Two 16x16 shared memory arrays are declared, one for each input matrix tile. Each thread loads a single element from each tile: cooperatively they load both 16x16 tiles into registers. The elements each thread loads are chosen so that half-warps load full lines from memory, enabling coalesced memory accesses. The threads then store the tiles into the shared memory. Finally, a small loop calculates the partial dot product. Two synchronizations are required for correctness: the first ensures that values in the shared memory are written before being used by other threads; the second ensures that all threads have finished using the values in shared memory before a new tile is loaded.

The 16x16 tiled version of matrix multiplication achieves 46.49 GFLOPS, or approximately 4.5X the execution throughput of the initial version. This performance is slightly higher than the estimated potential throughput of the original code, so it appears that the application achieves full usage of the SPs. The use of 16x16 tiles reduces global loads by a factor of 16 over the nontiled configuration, which reduces the demand on global memory bandwidth to approximately an eighth of the available bandwidth. Since utilization appears to be high, a developer can now focus on improving the efficiency of the code.

Code efficiency can be improved by reducing the number of operations that are not floating-point calculations. Branches and address calculations are two examples that are easily removed. Figure 3.1(c) shows removal of these operations via loop unrolling. By unrolling the inner loop, address calculation instructions are eliminated by replacing variable array indices with constant offsets. By completely unrolling the loop, all branch instructions have been eliminated. This version of the matrix multiplication kernel can often achieve double or greater performance over the simple tiled version.

Another method of improving efficiency is to perform rectangular tiling, as shown in Figure 3.1(d) and depicted in Figure 3.2(c). This transformation changes the granularity of computation such that a single thread calculates multiple elements in the result matrix. The primary benefit to efficiency is register tiling [16]: the value `As[ty][i]` only needs to be loaded into a register once for the calculation of two different result elements, reducing the total number of loads. The total number of executed control flow operations is also reduced. Rectangular tiling can be combined with unrolling for further gains. The next section will discuss other ways in which rectangular tiling may affect performance.

A final optimization that should be mentioned is prefetching, which some developers perform instinctively. In the code shown in Figure 3.1(e), loads of the values for the next tiles to be processed are initiated prior to calculating the partial dot product, hiding the latency of the global memory access. This transformation improves individual thread performance, which is generally easy for a developer to reason. However, prefetching in this situation improves performance only slightly. Prefetching can significantly improve execution resource utilization on traditional processors because it reduces or eliminates the exposed stall time prior to the use of a loaded value. On a highly threaded system such as the GeForce 8800, exposed stall time is generally covered by the execution of other threads, so prefetching does not have as great a performance impact.

Prefetching can sometimes *decrease* overall performance by reducing thread-level parallelism. In the case of matrix multiplication, prefetching requires two additional registers per thread to receive the prefetched values, which can result in fewer thread blocks executing on each SM. Experiments on matrix multiplication in Section 5.1 show that prefetching has a minor effect on performance at best and in one case decreases performance due to reduced TLP. The optimized versions of applications in Chapter 4 generally do not use prefetching.

3.3 Categories of Optimization

One fact that is ignored by many novice developers is that "optimizations" are simply transformations that may decrease application performance unless they target a specific behavior which inhibits application performance for the target application and architecture. Thus, optimization should always be done by first considering what effects are desired, then analyzing which transformation will produce the effect with minimal negative side effects. In this section I divide the three principles in Section 3.1 into various categories of optimization that apply to the GeForce 8800, then mention specific examples of optimizations.

The basic strategy for achieving good kernel code performance on the GeForce 8800 is to first ensure that executing threads are not starved for data: global memory bandwidth should not be the limiting factor on performance. After that, optimizations should balance an efficient instruction stream with high utilization of execution resources.

Optimizations rarely improve an aspect of machine-level behavior in an isolated manner. Many optimizations affect several aspects, producing a give-and-take situation between different categories. Moreover, many optimizations increase resource usage and thus compete for a limited budget of registers, thread contexts, and shared memory. The most common way in which optimizations interact and interfere on the GeForce 8800 is by their effects on register usage. For example, an optimization that increases the number of independent instructions after a long-latency instruction generally uses additional registers. This optimization causes register usage of each thread and thread block to increase, which in turn can cause the number of thread blocks assigned to each SM to decrease. I mention specific cases below.

3.3.1 Memory bandwidth optimization

The first class of optimizations deals with reducing pressure on memory bandwidth, particularly global memory. Attempts to improve instruction efficiency or execution resource utilization matter little to performance if the executing threads are starved for data. This problem was recognized by Ding and Kennedy for traditional processors [11]; the issue is exacerbated for data-parallel many-core processors. Listed here are some techniques for reducing global memory accesses.

Ding and Kennedy mention several in [17], with more sophisticated techniques described in [18].

- **Capitalize on the reuse of data via local, low-latency memories.** I showed an example of the use of shared memory in a matrix multiplication kernel. The caches also assist in reuse of data. Utilizing the memories effectively may require restructuring of computation using transformations such as loop interchange [19] to create an amenable access pattern.

- **Improve data locality by coalescing global memory accesses** to fully utilize the width of the global memory bus. Noncoalesced accesses utilize only a fraction of the total memory bandwidth. Optimizations include:

  - **Using shared memory as a buffer to load data in bulk**, then accessing them in whatever pattern the programmer finds convenient from that memory. Even if bank conflicts occur during shared memory accesses, overall performance may improve even without much data reuse. Note that memory tiling naturally achieves this effect. It is conceivable that loading some data that will not be used may be worth the effects of coalescing, although no example has been encountered in this study.

  - **Reorganizing data and restructuring the application so that the data the kernel accesses are located in aligned, contiguous regions of memory.** One example is to split an array of structures into multiple arrays so that fields that formerly required nonunit-strided access are now adjacent in memory. Doing so manually is generally considered poor software engineering practice, but it may be possible for the compiler to perform the transformation or for a smart memory system to reorganize data at runtime. Work by Yamada et al. [20] proposed a combined hardware and software technique to address this issue. Truong et al. [21] recognized the positive cache effects of similar optimizations on different instances of dynamically allocated objects, while Chilimbi et al. automate the optimizations [22].

It is possible for applications to still be performance-limited by global memory bandwidth even after these optimizations have been performed. Such applications often have little data reuse and have data organizations that cannot be significantly modified or require major effort to be modified.

3.3.2 Dynamic instruction reduction

The second category is to reduce the dynamic instruction count per thread, or increase the efficiency of the instruction stream. Because these are common techniques for improving the performance of applications executing on traditional superscalar processors, their first-order effects are generally well understood. Some of the most effective examples of these optimizations are listed below, along with a short description of their intended effects and common side effects.

- **Common subexpression elimination (CSE)**. This optimization removes repeated calculations of an expression, such as an arithmetic operation or the load of a value, and instead uses the saved value of a previous computation from a register. It tends to use additional registers unless a sequence of redundant expressions can be eliminated.

- **Loop-invariant code motion (LICM).** Related to common subexpression elimination, this optimization involves the movement of an expression within a loop body to a program point prior to the loop. LICM is possible when the sources of the expression do not change during the loop's execution and when the expression is calculated on every path of the loop (for nonspeculative motion). Similar to CSE, it tends to uses additional registers.

- **Loop unrolling.** When a loop has a constant or otherwise constrained trip count, it is often advantageous to "unroll" the loop by an even divisor of the trip count, replacing it with a loop that iterates fewer times but performs several times more work per iteration. This optimization has several benefits:

  - Fewer loop iteration instructions, such as branches and induction variable increments, are executed.
  - Instructions can be combined. For example, a loop may increment a pointer and then load from the location. When the loop is unrolled, instead of multiple increment operations, subsequent loads incorporate an offset and a single increment operation is performed. The unrolled matrix multiplication kernel in Figure 3.1(c) is an example of this effect.

  Complete unrolling of a loop is the extreme case of loop unrolling, and is very profitable for loops with small bodies since the majority of dynamic operations may be loop counter and branch instructions. It also frees up the register that was used as the loop counter. Loop unrolling must be balanced against additional pressure on instruction cache capacity, so in general only the innermost loops of a kernel will be the ones unrolled. It may also trigger other optimizations that change resource usage.

3.3.3 Increasing thread-level parallelism

The goal of the third category of optimization is to improve execution resource utilization by providing enough threads/warps to hide the stalling effects of long latency and blocking operations. Examples of long latency operations are loads from global or texture memories. Blocking operations include barrier synchronization, which stops a warp's execution until all warps in the same block have reached the barrier.

A common optimization in this category is to decrease the thread block size and increase the number of thread blocks. This optimization can increase the number of thread blocks assigned to each SM and provide more independent warps from other thread blocks when one thread block reaches a barrier. However, this transformation often requires changing the granularity of computation. In cases where memory tiling is utilized, it may have the effect of reducing data sharing and increasing pressure on memory bandwidth.

3.3.4 Increasing intrathread parallelism

The fourth category of optimization, intrathread parallelism, ensures the availability of independent instructions within a thread. These optimizations have a multiplicative effect with thread-level parallelism, so small changes can have major effects on performance. These optimizations can be broken down into two subcategories.

- **Instruction-level parallelism.** ILP-increasing code transformations have been extensively studied and applied to many architectures, particularly EPIC/VLIW architectures. John Sias' Ph.D. dissertation [23] discusses many of these in detail. One interesting case is loop unrolling, which is also an efficiency-increasing optimization. An example is shown in Figure 3.1(c), where the loads from shared memory for the different tiles can execute in any order.

This subcategory is primarily the jurisdiction of the instruction scheduler of the CUDA runtime. It appears to reschedule operations to hide intrathread stalls, sometimes to the detriment of interthread parallelism. As with optimizations to reduce instruction count, scheduling to reduce intrathread stalls may increase register usage and potentially reduce the number of thread blocks on each SM.

- **Memory overlap.** A special case of ILP, memory operations can be scheduled to amortize and overlap latencies, reducing the likelihood that execution resources will stall on global memory accesses. The GeForce 8800 supports a large number of in-flight memory accesses, making the overlap of memory access latencies a key aspect of performance on that architecture. This category breaks down into memory-level parallelism (MLP) [24] and prefetching.

  – Memory-level parallelism refers to the reordering of memory operations to ensure that many loads are in flight before a stalling use is encountered. For example, in Figure 3.1(b), loads of elements of arrays `A` and `B` can be scheduled before the store to `As` in order to amortize the exposed latencies of the loads. This concept is similar to the overlap of cache misses on traditional processors [25]. Proper extraction of MLP also causes more work to be available from each thread block during the stall, since each warp can execute two load operations prior to a consuming store. This optimization is generally the domain of nvcc and is difficult for a developer to control manually, but is still an important aspect of extracting performance from applications.

– The developer can explicitly insert prefetching code to initiate long-latency global loads as early as possible and hide its latency with other instructions. An example is shown in Figure 3.1(e). As discussed in the matrix multiplication example, the abundant TLP supported by the architecture reduces the performance benefits of prefetching relative to its effects on traditional processors. In addition, prefetching generally requires additional registers in code regions where register usage is already high, sometimes causing fewer thread blocks to reside on each SM. Good management of ILP and MLP can minimize the incremental benefit of prefetching.

3.3.5 Work redistribution

The fifth category involves redistribution of work across threads and thread blocks. These optimizations can change both instruction efficiency and resource utilization, with some effects on memory bandwidth usage. Because of their nature, optimizations in this category can have unpredictable results due to changes in register usage.

Thread block and memory tile shape and size selection is one optimization that falls into this category. Although tiling is primarily done to improve memory bandwidth usage, the size of the tile also has efficiency and utilization implications. Large tiles generally have higher efficiency than smaller tiles, but reduced scheduling flexibility (utilization) since a larger percentage of threads must stall at barrier synchronizations.

One example is rectangular tiling in Figure 3.1(d): instead of each thread computing one result, they compute multiple results. This transformation allows register tiling (also referred to as unroll-and-jam), where values can be reused in a register within a single thread to eliminate loads. The total number of threads is reduced, which may potentially reduce thread-level parallelism. Another benefit is the reduction in the total number of control flow operations executed by a kernel: more useful work is done per thread block and the total number of thread blocks is reduced. There is often a small increase in ILP, but the increase in register usage often causes fewer threads and thread blocks to be executed per SM.

Another optimization that is occasionally useful is to distribute work across multiple invocations of a kernel. This optimization is sometimes necessary to use the constant cache, due to its size limitation. Splitting execution across multiple invocations may also help improve cache behavior. The additional kernel invocation overhead is generally negligible compared to the kernel execution time.

3.3.6 Resource balancing

The last category is best termed resource balancing. The purpose of these optimizations is to make changes in resource usage to alleviate pressure on oversubscribed resources and shift to underutilized ones to produce better overall performance. Unless the whole system is taken into account, the optimization may be counterintuitive. Because of the large amount of execution resources, most optimizations involve giving up instruction efficiency to optimize another aspect of execution.

One optimization in this category is the movement of data from capacity-constrained, low-latency memories to larger, high-latency memories to allow more threads to run simultaneously. For example, explicit register spilling to shared or global memory can be performed by the application developer to allow more thread blocks to be simultaneously executed per SM. The resulting configuration may achieve higher performance, despite the reduced instruction efficiency, because the additional thread blocks improve overall resource utilization. Similarly, a kernel with small input and output data sets but large working sets may find it advantageous to spill data to global memory to avoid congesting registers or shared memory.

CHAPTER 4

APPLICATION STUDY

This chapter discusses an application mapping study performed in Spring 2007. Several groups of students ported applications to the GeForce 8800 with the goal of achieving maximal speedup. These applications were of interest to existing developers and users, who saw value in obtaining significant performance increases. They are more interesting and useful than microbenchmarks because of their larger code sizes and data sets, and variety of instructions and control flow. Work of a similar nature has been performed by Che et al. [26]. I discuss the applicability and effectiveness of the principles and techniques in Chapter 3 on these applications.

To accommodate the SIMD nature of the SMs on the GPU, threads of an application must follow nearly uniform paths of execution to have good performance. They should also have regular memory accesses and high compute-to-memory ratios or significant data reuse. Table 4.1 lists applications that have these characteristics in varying amounts and have been ported to CUDA, along with source and kernel lines of code (excluding comments and whitespace).

Table 4.1: Application Suite

Application	Description	Source Lines	Kernel Lines	CPU Execution Parallelized
H.264	A modified version of the 464.h264ref benchmark from SPEC CPU2006. This application is an H.264 (MPEG-4 AVC) video encoder. A serial dependence between motion estimations of macroblocks in a video frame is removed to enable parallel execution of the motion estimation code. Although this modification changes the output of the program, it is allowed by the H.264 standard.	34 811	194	35%
LBM	A modified version of the 470.lbm benchmark from SPEC CPU2006. This application uses the Lattice-Boltzmann Method for simulating 3D fluid dynamics. The program has been changed to use single-precision floating-point and print fewer status reports.	1481	285	$> 99\%$
RC5-72	This application accelerates distributed.net's RSA RC5-72 bit challenge, which performs brute-force encryption key generation and matching.	1979	218	$> 99\%$
FEM	Finite element modeling. Simulation of dynamic behavior of 3D graded materials.	1874	146	99%
RPES	Rys polynomial equation solver. Calculates 2-electron repulsion integrals, which are a sub-problem of molecular dynamics.	1104	281	99%
PNS	Petri Net Simulation. Simulation of a mathematical representation of a distributed system.	322	160	$> 99\%$
SAXPY	Single-precision floating-point implementation of `saxpy` from high-performance LINPACK, used as part of a Gaussian elimination routine.	952	31	$> 99\%$
TPACF	Implementation of two point angular correlation function, used to find the probability of finding an astronomical body at a given angular distance from another astronomical body.	536	98	96%
FDTD	Finite-difference time-domain. 2D electromagnetic wave propagation simulation in an arbitrary, user-defined medium.	1365	93	16.4%
MRI-Q	Computation of a matrix Q, representing the scanner configuration, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.	490	33	$> 99\%$
MRI-FHD	Computation of an image-specific matrix $F^H d$, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.	343	39	$> 99\%$
CP	Computation of electric potential in a volume containing point charges. Based on direct Coulomb summation, as described in [27].	409	47	$> 99\%$

Benchmark versions of some of the applications are presently available at [28]. The larger codes often required more modification to port to CUDA; the most extreme case was H.264, which involved a large-scale loop distribution to extract the motion estimation kernel from nonparallel application code. The percentage of single-thread CPU execution time spent in kernels is given to show the total application speedup that can be achieved as limited by Amdahl's Law. For example, FDTD's kernel takes only 16.4% of execution time, limiting potential application speedup to 1.2X. In general, kernel execution occupied the vast majority of execution for these applications when run entirely on a single CPU core.

Application selection is biased towards particular classes of problems such as linear algebra and stencil and grid-based computations. Most of these have data structures laid out in simple arrays, enabling easy coalescing of memory accesses by developers. Kernels generally have little variance in control flow and corresponding predictability in memory accesses.

Table 4.2 shows characteristics of the optimized application implementations using CUDA version 0.8.[1] The data for matrix multiplication are listed for comparison.[2] The maximum number of simultaneously active threads shows the amount of thread parallelism available on the hardware at a given time, taking resource constraints into account, with a maximum of 12 288 across the 16 SMs. There is a wide range of values, with little correlation of number of threads to actual speedup. The total threads in a given kernel often number in the millions. The number of registers and the amount of shared memory per thread show the degree of local resource utilization.

[1]Due to incompatibilities in both base architecture and CUDA versions, several of the applications could not be directly mapped to the evaluation system used in later chapters. Rather than remove the applications, I chose to keep the performance information for the older system.

[2]The GPU speedup for matrix multiplication uses a highly optimized library with SSE2 support as comparison. Kernel speedup compared to a CPU binary without SIMD support and optimized only for cache usage is on the order of 100X.

Table 4.2: Application Performance for Typical Long-Running Execution Profiles

Application	Max Simultaneously Active Threads	Registers per Thread	Shared Mem per Thread (B)	Global Memory to Computation Cycles Ratio	GPU Exec %	CPU-GPU Transfer %	Architectural Bottleneck(s)	Kernel Speedup on GPU	Application Speedup
Mat Mul	12 288	9	8.1	0.276	16.2%	4%	Instruction issue	7.0X	2.0X
H.264	3936	30	55.1	0.006	2.6%	4.5%	Register file capacity and cache latencies	20.2X	1.47X
LBM	3200	32	84.2	0.415	98.3%	0.4%	Shared memory capacity	12.5X	12.3X
RC5-72	3072	42	0.3	$\simeq$0	64.3%	0.5%	Instruction issue	17.1X	11.0X
FEM	4096	18	61	1.135	91.4%	$\ll 1$%	Global memory bandwidth	11.0X	10.1X
RPES	4096	23	24.8	0.01	37.5%	1%	Instruction issue	210X	79.4X
PNS	2048	32	9.9	0.241	98%	$\ll 1$%	Global memory capacity	24.0X	23.7X
SAXPY	12 288	7	0.3	0.375	88%	4.5%	Global memory bandwidth	19.4X	11.8X
TPACF	4096	24	52.2	0.0002	34.3%	$\ll 1$%	Shared memory capacity	60.2X	21.6X
FDTD	12 288	11	8.1	0.516	1.8%	0.9%	Global memory bandwidth	10.5X	1.16X
MRI-Q	8192	11	20.1	0.008	> 99%	$\ll 1$%	Instruction issue	457X	431X
MRI-FHD	8192	12	20.1	0.006	99%	1%	Instruction issue	316X	263X
CP	6144	20	0.4	0.0005	> 99%	$\ll 1$%	Instruction issue	102X	102X

Other information in the table includes the ratio of global memory cycles to computation cycles after shared memory and caches are utilized to their fullest extent, expressing the global memory bandwidth requirements of the most time-consuming kernel of each application. Section 4.1 discusses how this ratio correlates to performance. GPU execution time expresses how much of the total execution time the application kernels occupy on the GPU. CPU-GPU transfer time is shown for comparison with the computation time. One interesting case is H.264: a highly optimized version spends more time in data transfer than GPU execution. The last item is the architectural bottleneck(s) that appears to be limiting these implementations from achieving higher performance.

The two rightmost columns of Table 4.2 list the performance of ported applications. The baseline, single-thread CPU performance is measured on an Opteron 248 system running at 2.2 GHz with 1 GB main memory. The choice of processor was made with the intent of having a high-performance, single-core processor; similar CPU performance is found with newer, high clock rate multicore architectures.[3] CPU versions were heavily optimized for applications with outstanding GPU speedup to ensure that comparisons were fair: SIMD instructions and fast math libraries were some of the more effective optimizations. Applications were compiled with gcc version 4.1.3, with -O3 or the maximum optimization level that worked correctly. Both the speedup of CUDA kernel execution over single-thread CPU kernel execution and total application speedup were measured, with all floating-point numbers set to single-precision. Measurements were made with typical long-running inputs; e.g., for SPEC CPU benchmarks the reference inputs were used. I do not discuss the precision requirements of applications in this

[3]My claim of similar performance does not factor in performance using the CPU's SIMD execution resources: recently released processors can have double or more SIMD resources than the processor used here. Experiments in later chapters and work on individual applications [29, 30] were performed with a more contemporary evaluation system. As explained previously, several of these applications could not be directly ported to the newer system.

work, although experiments have shown that at least two applications, LBM and FDTD, suffer from insufficient floating-point precision.

4.1 General Performance Trends of Optimized Applications

In general, significant kernel and application speedup is obtained across the suite, as shown in Table 4.2. Compute-intensive kernels with relatively few global memory accesses achieve high performance. Kernels that are not as compute-intensive still achieve respectable performance increases because of the GeForce 8800's ability to run a large number of threads simultaneously. Low-latency floating-point execution is a major factor in speedup, as is the use of caches and shared memory to reduce latencies and global bandwidth usage. Careful organization of threads and data was generally useful in optimizing the usage of the specialized memories, most notably in the MRI kernels.

The applications in Table 4.2 with the highest performance gains, namely TPACF, RPES, MRI-Q, MRI-FHD, and CP, have low global access ratios and spend most of their execution performing computation or accessing low-latency memories. They also generate enough threads to hide potential stalls on long-latency operations and maintain high pipelined floating-point throughput.

One major reason for the high performance of the MRI kernels is that a substantial number of executed operations are trigonometry functions; the SFUs execute these much faster than CPU fast math libraries. SFU performance accounts for approximately 30% of the speedup of those kernels. Significant effort was spent improving the CPU versions (approximately 4.3 times faster than the original code) to ensure that the CPU-GPU performance comparison was reasonable [31]. The CP kernel has a significant number of reciprocal square root operations and has a similar performance advantage on the GPU. The opposite effect, where the processor must emulate functionality that is not supported natively in the instruction set, exists in RC-5: the GeForce 8800 lacks a modulus-shift operation. Performance of the code if a native modulus-shift were available is estimated to be several times higher.

Another reason for the MRI kernels' high performance, as well as that of CP, is that threads can be organized so that every thread reads the same sequence of memory addresses within the primary data-parallel loop. Since every thread in a warp reads the same value, the data naturally map to constant memory, making global memory bandwidth limitations a nonissue. The advantage of using constant memory in the MRI kernels is discussed in more detail in Section 4.2.3.

The H.264 kernel is notable for its use of texture memory to accomplish calculations that would be performed as normal processor instructions in a CPU-based implementation. The texture memory provides both 2D locality and boundary-value calculations. Although the kernel spends 20% of execution time stalled on texture memory, this configuration is still 2.8 times faster over global-only access. Section 4.2.1 discusses the H.264 kernel in more depth.

LBM, FEM, and FDTD are notable for being time-sliced simulators, where a portion of the simulation area is processed per thread. For each time step, updates must propagate through the system, requiring global synchronization. Since there is no efficient means to share data and perform barrier synchronization across thread blocks, the kernel processes a single time step and then terminates to ensure that all data writes to global memory in the previous time step are visible to the next time step. This pattern places high demand on global memory bandwidth since the kernel must fetch the entire working set from global memory and store it back after performing a small amount of computation. PNS does not have this issue because a separate simulation is performed per thread. One possible solution to this issue is to relax the global synchronization requirement by changing application algorithms.

Memory-related bottlenecks appear in LBM, FEM, PNS, SAXPY, and FDTD, all of which have high memory-to-compute ratios. These high ratios cause bottlenecks in two ways. First, LBM and PNS are limited in the number of threads that can be run due to memory capacity constraints: shared memory for the former, global memory for the latter. Second, FEM, SAXPY, and FDTD saturate memory bandwidth. Even though the latter two have the highest number of simultaneously active threads of the suite, having many threads does not address the large memory to compute ratio, which causes memory bandwidth to be the primary performance bottleneck.

4.2 Case Studies

This section focuses on three applications and details their mapping to CUDA. I explain what makes the application easy or difficult to map to the architecture, the changes required to achieve higher performance over a straightforward implementation, and the obstacles to further increases in performance.

4.2.1 H.264: Sum of absolute differences

The SPEC CPU2006 benchmark 464.h264ref is an implementation of an H.264 video encoder (also referred to as MPEG-4 Advanced Video Coding). The original code, running on a single core of an X86 processor, spends approximately 60% of its execution time in the routine `SetupFastFullPelSearch` and its callee functions. This routine compares fixed-size regions from two frames of a video to determine how well the blocks match. The metric used for the quality of a match is called a *sum of absolute differences* (SAD): the absolute value of the difference between pels (video pixels) at the same location of the two regions is calculated, and the sum of those differences for all pixels is the result. Efficient and fast SAD computation has been researched extensively due to its importance in contemporary video encoders [32]: a high-quality encoder can produce smaller bitstreams or a higher quality encoded video.

The CPU-only version used for performance comparison was hand-optimized for a more fair comparison. In the inner loop that corresponds to the 4x4 SAD kernel, an indirect function call was replaced with a conditional branch and inlined function calls. The optimized version was also rewritten to use the `abs` library call instead of a lookup table for the calculation of absolute values. The rest of the application was left unmodified. The modified CPU-only implementation spends only 35% of its time in the `SetupFastFullPelSearch` routine.

The encoder's design made extracting data parallelism difficult. The encoder partitions video frames into 16x16 pel *macroblocks*, which are processed singly through all stages of encoding. Data parallelism is limited within the processing of a single macroblock, consisting of loops with few iterations and small bodies. Furthermore, real data dependences between encoding of neighboring macroblocks serialize many stages of the encoding process. The portion of the code performing motion estimation was distributed out of the loop and optimized for GPU execution, which required significant developer effort.

In several stages of encoding, the encoding parameters for a macroblock depend on results produced for neighboring macroblocks in the same frame. For example, in motion estimation a starting point for the search is obtained by averaging the best-fit locations of neighboring blocks that have already been processed. This dependence was broken to parallelize the code, which is acceptable by the H.264 standard but may result in lower-quality and larger encodings for fast-moving video. More frame-level parallelization may be possible with algorithm changes, but changing the algorithm is a design decision that affects the quality and size of the encoded output and would be made with the target platform and purpose in mind.

When encoding a CIF-size[4] (352x288) video with a 16-pel search range, 1.7 million 4x4 SADs and 1 million larger SADs are generated to encode one P-frame. The huge degree of parallelism across many uniform and simple SAD computations makes it appealing for GPU execution. The SPEC version of the encoder was rewritten to compute all SADs for a frame at once, rather than interleaving SAD computation with other steps of encoding. This change increases the available parallelism and amortizes the time spent in data transfers. Three GPU kernels were created to replace the function `SetupFastFullPelSearch`. The first kernel, taking 61% of GPU time, computes SADs over 4x4 areas. The second and third kernels, taking 31% and 8% of GPU time, respectively, compute SADs for larger areas using the results of the first kernel. The discussion below focuses on the first kernel since it comprises most of the computation on the GPU.

An initial GPU implementation computes one 4x4 SAD per thread, yielding a 15X kernel speedup relative to the CPU-only code. However, speedup can be doubled by writing the kernel to exploit data reuse by selectively grouping computation into threads and thread blocks and by applying software-managed caching, loop transformations, and instruction-level optimizations. These are described in detail below.

To illustrate the available data reuse, the code for a simplified sequential version of the SAD computation loop is shown in Figure 4.1. Each execution of lines 5-10 reads a 4x4 square area of two video frames and computes a single SAD

[4]CIF stands for *Common Intermediate Format*. With regards to image size, H.264 tends to split into two categories. The first is the mobile market, where small screens are the norm. H.264 is also used for high-definition video, with length and width in the thousands of pels. Both categories benefit from high-quality motion estimation: mobile devices require less data to be transmitted, and high-definition can achieve better video quality with similar data usage.

```
1 for (blockY = 0; blockY < height / 4; blockY++)
2    for (blockX = 0; blockX < width / 4; blockX++)
3       for (offY = -16; offY <= 16; offY++)
4          for (offX = -16; offX <= 16; offX++)
5             sad = 0;
6             for (pixY = 0; pixY < 4; pixY++)
7                for (pixX = 0; pixX < 4; pixX++)
8                   sad += abs(reference[blockY*4+offY+pixY][blockX*4+offX+pixX]
9                              - frame[blockY*4+pixY][blockX*4+pixX]);
10            sad_array[blockY][blockX][offY][offX] = sad;
```

Figure 4.1: Simplified SAD kernel.

value. Iterations of the loops at lines 3-4 select different 4x4 areas of the reference frame for comparison. These 4x4 areas partially overlap with one another. The two outermost loops select different 4x4 areas of the frame being encoded. Reads from `frame` are fully reused across the loops at lines 3-4, since the array index is not dependent on `offX` or `offY`. The two outer loops carry no reuse in `frame`. There is partial reuse in `reference` across all of the outer four loops, since the 36x36 area from `reference` used for each block is shifted by 4 pels between iterations of either loop.

Each thread block is responsible for several iterations of the outer two loops. Reuse is exploited in `frame` by caching the data in shared memory, and in `reference` by using the texture cache. Within a thread block, each thread executes several iterations of the middle two loops, taking advantage of register tiling [16] to eliminate redundant memory accesses. The innermost loop is completely unrolled and the `offX` loop is unrolled by a factor of 3; register pressure prevents further profitable unrolling.

Memory access delays account for more than half of the kernel's execution time. Memory transfers related to software-managed caching of input and output data, using the shared memory to enabled memory coalescing, consume 5.5% of execution. In contrast, operating directly out of global memory increases kernel execution time by a factor of 2.8. Another 21% is consumed reading values from a constant memory array to obtain index values for the spiral search pattern. This array fits entirely in the cache, but threads in this kernel do not read the same address in the same cycle, causing serialized reads from the single constant cache port. While texture memory provides 2D image caching and special behavior for out-of-bounds image accesses, both of which assist performance, 24% of kernel execution is spent waiting on results of texture fetches. Cache misses account for less than 3% of that 24%. The large per-thread register usage limits the number of threads per SM to 256 or fewer, which is not sufficient TLP to hide the latency of a texture fetch. 123 threads per block[5] was chosen to utilize close to 256 threads per SM, primarily to minimize the effects of texture fetch latency.

The remaining half of the 4x4 SAD kernel's execution time is spent in computation out of the register file and shared memory. The code was optimized through manual application of classical optimizations to reduce the instruction count in threads, primarily strength reduction of integer multiplication and divide-modulus operations. One GPU-specific optimization was to broadcast values through shared memory rather than recompute them in each thread.

The other two GPU kernels load SADs calculated by the first kernel, add them together, and write the new SADs to global memory. They are nearly identical, differing only in address calculations. The inner loop is very memory-intensive with 11 global memory instructions and 9 other instructions. Thus, the most important optimizations were to take advantage of global memory coalescing and to keep many threads active simultaneously to enhance thread- and memory-level parallelism.

While latency tolerance through thread parallelism affords speedup in the primary kernel with a shader-style computation model, additional speedup was achievable through CUDA's programming model by consolidating a larger amount

[5]This seemingly odd number is primarily due to the way the 33x33 search area is divided.

of work into threads and then applying loop-level optimizations. These optimizations used additional registers over a naïve version, reducing the number of in-flight threads and exposing more of the hardware's latency. However, they reduced the dynamic instruction count sufficiently to improve overall performance. Overall, the largest remaining bottleneck is not in GPU computation (2.6% of application execution time) but in moving memory between the CPU and GPU (4.5%), which exceeds the time spent in GPU computation and limits the available speedup.

4.2.2 Fluid dynamics using the Lattice-Boltzmann method

The SPEC CPU2006 benchmark 470.lbm is a fluid dynamics simulator that implements the Lattice-Boltzmann method. The program simulates the collisions between fluid particles across a series of time steps. The most heavily executed code region is an iteration-parallel loop within the function `performStreamCollide`, which accounts for over 99% of single-core CPU-only execution time when running the SPEC reference input.

Figure 4.2 outlines the basic algorithm. The primary data structure for the program is a 3D grid of cells, which represents the space through which fluid flows. Each cell within the grid characterizes a set of fluid particles and their velocities toward neighboring cells. Some cells are designated as obstacles to interfere with the fluid particles and are treated accordingly during simulation. Each cell is a 20-element floating-point array: one element is used as a flag to specify properties related to the cell, one is used for flow information for the cell itself, and the other 18 are used to specify velocities towards neighboring cells that share an edge with the cell (6 faces and 12 edges).

```
initializeGrid(src)
initializeGrid(dst)
for(each time step)
   if (mode == CHANNEL)
      handleInOutFlow(src)
   performStreamCollide(dst,src)
   swapGrids(dst,src)
```

Figure 4.2: LBM algorithm.

```
for (all cells in the grid)
   /* check to see if cell is not an obstacle */
   if (!(src[off+19] & OBSTACLE))
      rho = src[off+0] + src[off+1] + ... + src[off+18];

      /* calculate directional vectors */
      ux = src[off+3] - src[off+4] + ... - src[off+18];
      . . .
      /* compute and store velocity field elements to neighboring cells */
      dst[above_off+1] = . . .
      dst[below_off+2] = . . .
      . . .
   else
      /* copy velocity field elements of current cell to neighboring cells */
```

Figure 4.3: Partial code for `performStreamCollide`.

Figure 4.3 shows partial code of the `performStreamCollide` function. Every thread operates on a single cell of the grid and all 20 array elements are consumed. A major issue is that the usage pattern of data is dispersed, as depicted in Figure 4.4(a), which takes a memory cycle per datum to satisfy when these are loaded from global memory. Because all data of each cell are used, threads can cooperate to buffer the data in shared memory and gain performance from access coalescing, as shown in Figure 4.4(b). By also unrolling loops, this implementation has 12.3X performance speedup over the CPU-only version on the SPEC reference input. An unoptimized version, in comparison, achieves 6X speedup.

Further optimization of `performStreamCollide` was limited due to the algorithm's access patterns and resource limitations of the GPU. First, the 16 KB available in an SM's shared memory can buffer data for only 200 threads (16 KB /

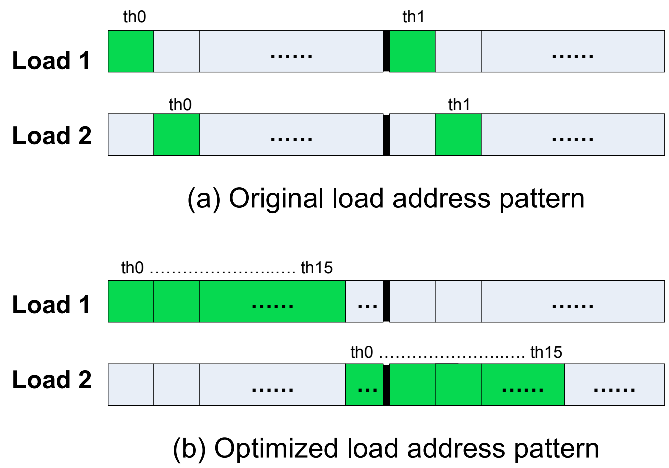


Figure 4.4: Address pattern for two consecutive loads in LBM.
(a) Requires a memory cycle per address due to non-contiguous access.
(b) Can obtain several values per cycle via coalescing but requires shared memory buffering.

(20 elements/thread * 4 B/element)). Second, a global synchronization is required at the end of every time step, which CUDA can only provide with a kernel termination. As a result, the contents of shared memory are not preserved between kernel invocations and a given element is used at most three times before being discarded. Lastly, stores of result data are not coalesced, although the buffers used to coalesce memory reads could be utilized to achieve the same effect for memory writes.

4.2.3 Magnetic resonance imaging in non-Cartesian trajectory space

The reader is likely to be familiar with the use of magnetic resonance imaging (MRI) in the medical domain. MRI uses algorithms based on inverse Fourier transforms to reconstruct an image from a set of RF signals collected from a sample in a slowly time-varying magnetic field gradient. What is not commonly known is that conventional MRI does not have high resolution compared to radiation-based imaging because it makes tradeoffs to reduce the computation requirements of the algorithm. Cartesian/grid-based scans are commonly used so that an image can be reconstructed using a fast Fourier transform (FFT). However, this choice often produces imaging artifacts, particularly in 3D images, limiting its usefulness for identifying small features in biological tissue. While radial- and spiral-based scans can produce higher-quality images, they require significantly more computational power, as the computations to reconstruct the image must be performed in an arbitrary, non-Cartesian trajectory ("k-") space. The computational power supplied by the GPU makes it possible to perform reconstruction using these scans in a practical amount of time, opening up the possibility of fast, high-resolution MRI. Work by Stone et al. [31] discusses this application and its optimizations in greater depth.

$$d(t) = \int \rho(x) e^{-i2\pi kx} dx + \eta \tag{4.1}$$

$$F^H d = (F^H F)\rho = Q\rho \tag{4.2}$$

Equation (4.1) shows the generic MRI reconstruction problem: $d(t)$ is the measured input, ρ is the desired image, k represents the trajectory, and η is external noise. This equation can be transformed into Equation (4.2): F is the Fourier matrix and F^H is its conjugate transpose. The algorithm operates by precomputing a matrix Q and a vector $F^H d$ and then finding an iterative linear solution via conjugate gradient. The time-intensive part of this algorithm is the computation of Q and $F^H d$. Q is specific to the scanner configuration and only needs to be computed once for a given scanner setup, while $F^H d$ is specific to both the scanner and the image being reconstructed, and is computed on a per-image basis. Both computations involve matrix or vector multiplication by a Fourier matrix whose elements are generated on the fly. Figure 4.5 shows the code to compute Q, while Figure 4.6 shows the $F^H d$ computation.

```
for (idxK = 0; idxK < numK; idxK++)
   phiMag[idxK] = phiR[idxK] * phiR[idxK] + phiI[idxK] * phiI[idxK];


for (idxK = 0; idxK < numK; idxK++)
   for (idxX = 0; idxX < numX; idxX++)
      expArg = 2 * PI * (kx[idxK] * x[idxX] + ky[idxK] * y[idxX]
               + kz[idxK] * z[idxX]);
      Qr[idxX] += phiMag * cos (expArg);
      Qi[idxX] += phiMag * sin (expArg);
```

Figure 4.5: Sequential algorithm to compute Q.

```
for (idxK = 0; idxK < numK; idxK++)
   realRhoPhi[idxK] = phiR[idxK] * dR[idxK] + phiI[idxK] * dI[idxK];
   imagRhoPhi[idxK] = phiR[idxK] * dI[idxK] - phiI[idxK] * dR[idxK];

for (idxK = 0; idxK < numK; idxK++)
   for (idxX = 0; idxX < numX; idxX++)
     expArg = 2 * PI * (kx[idxK] * x[idxX] + ky[idxK] * y[idxX]
              + kz[idxK] * z[idxX]);
     cosArg = cos(expArg);
     sinArg = sin(expArg);
     rFHD[idxX] += realRhoPhi[idxK] * cosArg - imagRhoPhi[idxK] * sinArg;
     iFHD[idxX] += imagRhoPhi[idxK] * cosArg + realRhoPhi[idxK] * sinArg;
```

Figure 4.6: Sequential algorithm to compute $F^H d$.

The experimental results were collected on an image size of 64x64x64 pixels and a trajectory size of 147 258. The highest CPU-only performance was measured at 0.199 GFLOPS for Q and 0.263 GFLOPS for $F^H d$ using single-precision floating-point, fast math code generation for trigonometry functions, SSE vector instructions, and manual tuning of the code. This performance corresponds to an execution time of approximately 5 hours for Q and 40 minutes for $F^H d$.

Direct, unoptimized ports of the kernels to the GPU are respectively 81X (18.8 GFLOPS) and 73X (22.9 GFLOPS) faster than the CPU-only versions, but higher performance can be achieved. The primary performance optimization was to store trajectory values for each step in constant memory. To enable this optimization, the computation was divided into steps small enough that each step's data fit entirely in the constant cache. The loops of both kernels were interchanged so that threads simultaneously iterate over the `idxK` indices in their inner loop, thus loading the same value from constant memory in a given cycle. The large register file also allows five values (`x`, `y`, `z`, `Qr`, and `Qi`) to be held in registers for each thread during the inner loop. The optimized kernel has the character of a texture shader: it loads a small number of values, proceeds through a sequence of floating-point operations and constant-memory lookups, and finally stores a small number of values out to memory.

There are a significant number of trigonometric calculations, approximately one for every seven standard floating-point operations. Fast math code generation was used on both CPU-only and GPU versions. On the GPU, fast math uses the SFUs rather than perform a software Taylor expansion. Fast math does not unduly affect the image quality; the signal-to-noise ratios are reduced by only 3 dB and still exceed generally accepted quality standards by 25 dB. These changes result in a throughput of 96.1 GFLOPS for Q and 74.0 GFLOPS for $F^H d$, or 457X and 316X respectively over CPU-only performance. Further speedup appears to be limited primarily by the availability of instruction issue bandwidth: a higher issue and execution rate in each SM would improve application performance.

Several aspects of the GPU design contribute to this application's impressive speedup. First, the SPs and SFUs are heavily optimized for floating-point performance, so floating-point instructions, particularly the trigonometry functions, take fewer clock cycles than on the CPU. Second, the GPU can maintain high pipelined floating-point throughput without dependence stalls by overlapping the execution of separate threads. Third, the abundance of registers reduced the number of memory instructions and the associated stall times on the GPU. In contrast, the CPU-only code has a large proportion of floating-point loads and stores, due to the CPU's limited register capacity and the lack of direct register transfer instructions between the SSE and FP units. Finally, the broadcast ability of the constant cache enables a much higher effective memory bandwidth than is possible on traditional processors.

Continued work by Stone et al. [30] further improved the performance of both CPU-only and GPU versions of the application. They use a newer, four-core processor with wider SSE capabilities as the base platform, use the Intel compiler to achieve higher per-thread performance, and thread the code to utilize all cores on the superscalar processor. The authors were able to improve the performance of the CPU-only implementation so that the GPU's performance advantage was reduced to 23 times over a four-core CPU-only implementation, as opposed to over 100 times in the results shown here. They found that only when constant memory and the SFUs were used did the GPU achieve significant performance benefit over a highly optimized, superscalar, multicore solution. This indicates that for the MRI kernels and similar applications, the primary bottlenecks of superscalar processors are memory bandwidth and trigonometry calculation efficiency.

4.3 Optimization Practice

This section discusses the relative effects of optimizations and specific experiences with the applications studied. It is intended as a guide to developers optimizing their applications and to give specific examples of methods to follow the principles in Section 3.1.

4.3.1 Memory optimization

Optimization of memory accesses is generally necessary to achieve significant speedup for the applications in the suite. For some applications the choice of using texture or constant memory is intuitive, such as texture memory for H.264 and other video applications. It may be necessary to transform the code in order to expose the proper memory access patterns to efficiently utilize these memories. Otherwise, shared memory is the default choice for reducing redundant loads and thus pressure on memory bandwidth. Its use is straightforward when there are either no shared values between threads (each thread has its own private space) or when neighboring threads share data in a simple blocked/tiled pattern, similar to matrix multiplication. Care must be taken so that threads in the same warp access different banks of the shared memory; e.g., array elements may need to be padded to ensure nonconflicting accesses. More complex applications often use more sophisticated data structures, requiring splitting of data structures or other effort by the developer.

One use of shared memory is as a buffer to improve the access pattern of global memory. As stated previously, memory bandwidth is easily saturated when accesses are not coalesced into 16-word, aligned regions. LBM, FEM, FDTD, and other lattice computations use arrays of small structures in global memory. Threads simultaneously read or write a given field of multiple elements and these fields are not contiguous in memory. Each noncontiguous access is a separate DRAM access request, overwhelming the device's memory bandwidth. In LBM the problem can be alleviated using contiguous accesses to transfer the arrays in bulk into shared memory; this transformation is discussed in Section 4.2.2. The buffering optimization may also be possible with FDTD if a substantial amount of data reorganization is performed, but FEM uses an irregular mesh data structure that has few contiguous accesses even with data reorganization.

Even with the use of shared memory, coalescing accesses to global memory generally improves performance. In some cases, noncoalesced accesses cause memory bandwidth to be a performance bottleneck, as effective bandwidth is significantly lower without coalescing. It is also important to remember that the SIMD nature of warp instruction issue on the SMs means that a warp cannot execute until all of its input operands are available. Because noncoalesced accesses must be processed sequentially, they cause a warp's memory instructions to have a longer latency than if accesses were coalesced. As a result, noncoalesced accesses require more parallelism to cover their latencies. The performance difference may not be multiple times, but in general global coalescing is worth the cost of any extra instructions that may be needed in order to coalesce accesses.

Tiling factors of thread blocks also play a role in performance. Large thread blocks have the potential to increase data sharing but increase the coarseness of thread block-to-SM scheduling and can reduce TLP. The choice of tile shape should be first motivated by global memory coalescing and other memory system optimizations, such as the use of shared memory for applications with large per-thread data sets, or the constant cache. After tiling, optimizations such as register tiling can be performed to further improve performance.

As previously discussed, prefetching is not often used as an optimization because it generally has little positive effect and often causes reduced performance due to decreased TLP. I refer to Chapter 3 for further discussion on prefetching.

4.3.2 Register usage

In Section 3.1, I stated that performance depends on the percentage of "core" operations (instruction efficiency) and the utilization of execution resources. On the GeForce 8800, utilization is achieved through TLP on the SM, with a multiplicative effect from threads' ILP and MLP. Efficiency is addressed through "classical" compiler optimizations. In a highly optimized application, improving one is often done at the cost of the other. On the GeForce 8800, efforts to improve efficiency may inadvertently damage utilization by the reduction of TLP, due to an increase in registers per thread. This TLP reduction can reduce overall kernel performance.

There are many optimizations which can increase register usage, many of which are described and cited in compiler texts [33–35]. They are generally performed with the intent of reducing the number of executed instructions or replacing slower operations with faster ones (e.g., integer multiply of a power of two replaced with a shift operation). The common optimizations that increase register usage are CSE and redundant load elimination, which are performed by nvcc. Developer-directed loop unrolling also may increase register usage by exposing instruction scheduling opportunities to the runtime.

In general, developers should be aware of potential changes in register and shared memory usage that can result from optimizations, as detailed in Chapter 3. Limiting the scope and scale of optimizations can also help control usage. Optimization on the innermost, most frequently executed kernels generally has beneficial effects without a major increase in register usage.

Three techniques can also help mitigate these effects:

- The performance loss from register usage is most apparent in application configurations with few, large thread blocks. Configuring an implementation to use a larger number of smaller thread blocks results in fewer threads being lost when one fewer thread block can be simultaneously scheduled per SM. Using smaller thread blocks is to be avoided unless other performance benefits are expected: kernels employing tiling optimizations usually benefit from large thread block sizes because they eliminate more memory operations. In tiled kernels, one may have to try a range of block sizes to find the best configuration.

- A developer can explicitly spill and fill values, normally mapped to registers, to shared or global memory. Register spilling is done by default for the matrix multiplication kernel used in this work, as it adds only two additional instructions to each thread's execution while saving at least one register. Overzealous register control can hurt performance due to the additional executed operations.

- Register pressure-sensitive code scheduling algorithms and optimization strategies, such as that proposed in [36], have been investigated in the context of instruction-level parallelism compilers. Additional research is needed to apply these strategies to architectures such as the GeForce 8800 because the "correct" number of registers to use is uncertain, as discussed in Section 4.3.3.

4.3.3 Balancing thread-level parallelism with individual thread performance

From the point of view of a developer performing transformations by hand, optimization becomes an effort of improving individual thread performance while maintaining enough threads to hide memory latencies and utilize execution resources effectively. Developers must consider that the highest-performing configurations may have relatively few threads and thread blocks per SM. For example, Table 4.2 shows that some application implementations use only a third of the available thread contexts. Even when there is an option to use more threads, these configurations have superior performance because the ILP within the threads makes up for reduced TLP compared to other configurations, while code efficiency is higher due to the optimizations which use the larger number of registers per thread.

Developers should be also aware that a configuration found by manual, iterative optimization may be a local performance maximum. For example, in H.264, an initial implementation had many threads simultaneously executing per SM, each with a small number of registers. An exhaustive optimization space search found that a configuration with fewer threads, each with more registers, achieved higher performance by performing fewer memory operations. This configuration was often stalled on accesses to texture memory, but execution efficiency during nonstalled periods was higher than that of other configurations.

Attempting to manually find the best balance of parallelism and a highly efficient instruction stream is difficult for the GeForce 8800 for several reasons. First, there are generally multiple potential high-performance configurations and they will be dissimilar in several optimizations, meaning that it is possible to be trapped at a local maximum during iterative optimization using a greedy approach. Second, resource usage and instruction scheduling is not under complete control of the developer, so targeting particular parallelism levels is difficult at best. Instead, in the next chapter I propose a technique that begins with the entire optimization space and prunes it to find configurations which are likely to have high performance.

CHAPTER 5

OPTIMIZATION CARVING

When developing an application with a high performance requirement, a developer begins with a mental model of the target platform and creates the application with this model in mind. One usually starts with the first-order concern for performance, which historically has been the application algorithm and its instruction count. After finding a seemingly reasonable base configuration, the application will be tested and then iteratively optimized until an acceptable configuration is found.

This approach is often adopted by compilers that generate high-performance code for traditional uniprocessors and multiprocessors: algorithms cannot be changed, so the efficiency of the instruction stream is the first-order performance concern. A plethora of instruction removal, redundancy elimination, and strength reduction optimizations have been developed to address this concern.

However, the iterative optimization approach has major detriments when applied to many-core compilation. The broad issues were discussed in the first chapter, and some specifics are discussed here:

1. It is well understood by the compiler community that particular orderings of optimizations can trap optimization processing into a local performance maximum. Optimization phase ordering [37, 38] has not received as much attention as it could have. The reason is that the performance difference between two configurations of an application executing on a single-core superscalar or EPIC/VLIW processor might be on the order of several percent.

On many-core platforms such as the GeForce 8800, a performance difference of several times is possible. Optimizations for the system have complex effects, and a poor decision by an application developer or compiler can trap the application in a local performance maximum. This possibility makes it imperative to perform a broad search of the space when increased application performance is valuable.

2. Contemporary many-core architectures have a wealth of execution resources, but these are separated both by physical space and by architecture structure, such as the lack of direct global-to-scratchpad memory transfers on the GeForce 8800. The cost of communication is significant, which is well-understood by parallel program optimization experts. However, the degree of tradeoffs on many-core systems is different: for example, there is a much more vague tradeoff between whether a processing unit should recompute a value locally or obtain that value from another processing unit, due to shorter communication latencies between processing units on the same chip. Determining a good combination of tradeoffs will be difficult for most developers.

3. We are currently in a period of rapid innovation in both applications and architecture; the GPU innovation cycle is particularly short. New features are introduced very regularly, with some residing on hardware and disabled until their correctness can be validated. However, it takes time and effort for application developers and compiler designers to utilize new features; they will not be well understood for a significant amount of time. Also, to preserve competitive advantage, limited information may be available about these features. The primary examples encountered during the application study for the GeForce 8800 were global memory coalescing, cache attributes,

and SFU performance. Iterative compilation is likely to make the wrong assumptions about how to use these features and create suboptimal application optimization configurations. In addition, because application features are added over time, a configuration that worked well for a previous version of the application may not stay that way. Changes may invalidate assumptions about the effects of optimizations and generally require reapplication of the optimization process.

Instead of selecting a single starting configuration and iteratively optimizing it, I propose an approach that begins with a large space of configurations and then prunes away those that are likely poor performers. This pruning is done based on knowledge of the primary and secondary performance concerns for the application and architecture, quantified as metrics. Because I found a single cost function to be ineffective across multiple applications, the technique uses thresholds and Pareto-optimal curves to determine which configurations should be pruned. The purpose of the technique is to find a near-optimal configuration without detailed, accurate knowledge of the system by either a developer or the compiler.

I begin this chapter with one of the most simple examples of application optimization, that of matrix multiplication, and explain why it can be difficult to find the best configuration. I then describe the optimization carving technique and how metrics are used to prune the optimization space. I identify and create metrics for the GeForce 8800 and then apply them to the optimization of kernels for the architecture.

5.1 Example Revisited: Matrix Multiplication Optimization Space

The purpose of this section is to show the primary performance facets of the architecture and the discontinuity of the optimization space, using the matrix multiplication kernel used in previous chapters. I start with a tiled configuration of matrix multiplication, and vary several parameters to explore the entire optimization space. The parameters used for matrix multiplication are:

- **Tile size:** I choose thread blocks of size 8x8 and 16x16. They process an integral multiple of data tiles. Tiling reduces memory bandwidth usage.

- **Unrolling factor:** The innermost loop of the kernel has been unrolled by an even divisor of the loop trip count, varying between one (no unrolling) and complete unrolling of the loop. Unrolling reduces the number of dynamic instructions executed.

- **Rectangular tiling factor:** The default configuration, 1x1, processes one tile in the first input matrix and one tile in the second input matrix at a time. Increasing the rectangular tiling factor to 1x2 processes two tiles in the second input matrix for every one in the first, and so on. Rectangular tiling enables the register tiling optimization and reduces memory bandwidth usage.

- **Prefetching:** In an effort to improve ILP and reduce exposed memory latency, a prefetching optimization is applied to overlap global memory load latencies with computation.

Figure 3.1 on page 32 shows code examples of these optimizations, while Figure 3.2 depicts their memory access patterns.

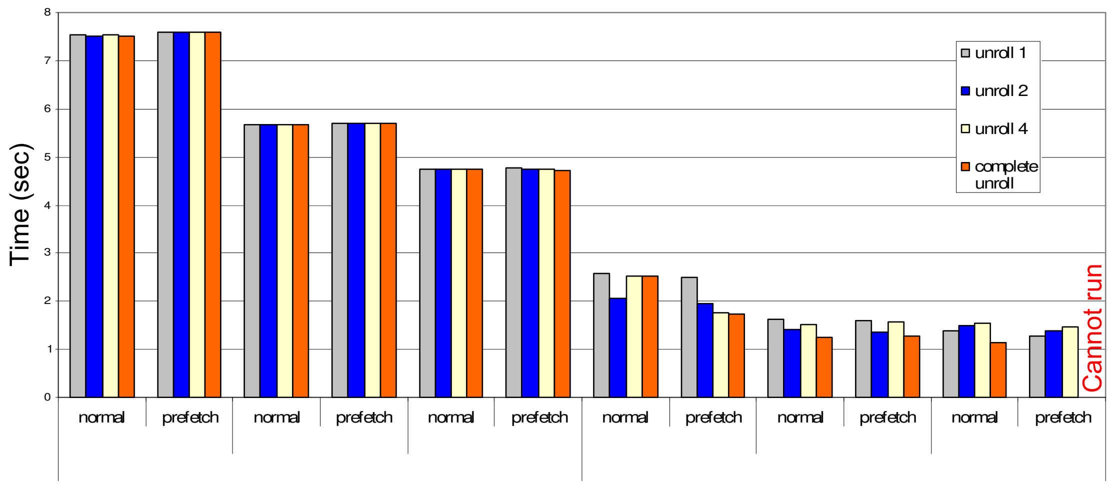


Figure 5.1: Matrix multiplication optimization space runtime.

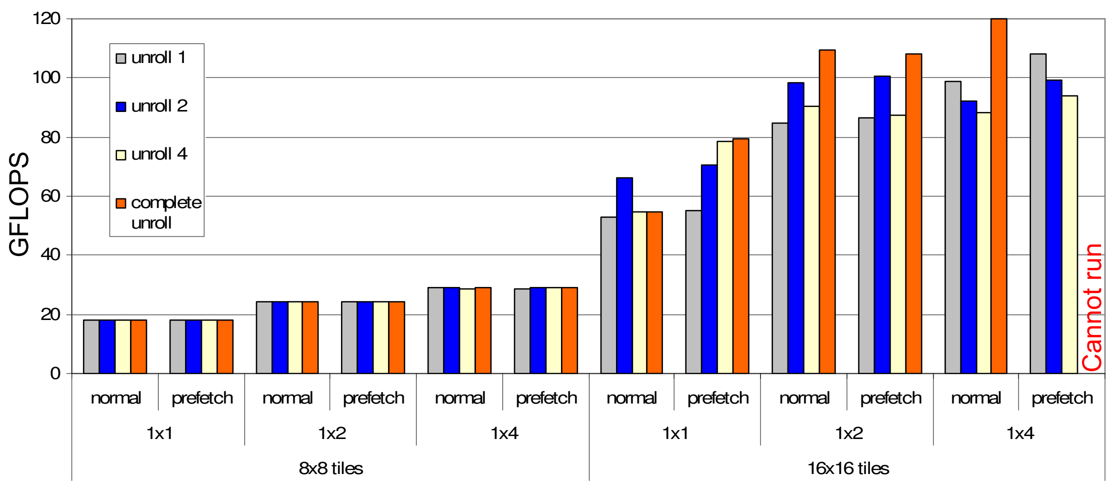


Figure 5.2: Matrix multiplication optimization space performance.

Figure 5.1 shows the run time of these configurations, while Figure 5.2 shows the performance in GFLOPS. One configuration, with 16x16 tiles, 1x4 rectangular tiling, and prefetching, does not execute because it is oversubscribed on registers. There are several performance trends shown by the results.

- When bandwidth is the major performance limitation of the kernel, little else matters; e.g., increasing instruction stream efficiency or ILP and MLP (through prefetching) has no appreciable effect. This effect is shown for the 8x8 tiles: the tiling format does not allow for coalescing of global memory accesses and its performance is severely restricted by memory bandwidth limitations. Although rectangular tiling alleviates the pressure on bandwidth, other optimizations still have little effect until memory bandwidth is removed as a performance limitation via global memory coalescing using 16x16 tiles.

- Unrolling is a hit-or-miss proposition for increasing performance on this architecture. The reason for performance losses, other than instruction cache effects, is that the runtime creates an instruction schedule that increases register usage and allows fewer thread blocks to be simultaneously executed per SM. This effect causes a loss of SP utilization that must be gained, if possible, through further optimization: complete unrolling is generally superior to an unroll factor of 4.

- Increasing the rectangular tiling factor generally improves performance. It improves the efficiency of the instruction stream, global memory bandwidth utilization, and ILP. However, to take advantage of register tiling, additional registers must be used, which in one case reduces the number of thread blocks executing per SM and kernel performance.

- As mentioned in Chapter 3, prefetching has a limited benefit for this system because there is often much useful work that can be done while a long-latency load request is being satisfied. It provides a minor improvement in performance when there are sufficient registers to not cause fewer thread blocks to be executed per SM. For bandwidth-limited configurations, such as those with 8x8 tiling factors, there is no appreciable benefit.

Table 5.1 shows the resource usage of the 16x16 matrix multiplication configurations, excluding register spilling. It shows how the application of optimizations

Table 5.1: Resource Usage and Performance for 16x16 Matrix Multiplication Configurations Excluding Register Spilling

Rectangular Tiling Factor	Unrolling Factor	Prefetching	Registers per Thread	Thread Blocks per SM	Performance (GFLOPS)
1	1	no	11	2	53.13
	1	yes	13	2	55.17
	2	no	11	2	66.40
	2	yes	13	2	70.52
	4	no	12	2	54.71
	4	yes	14	2	78.54
	complete	no	13	2	54.58
	complete	yes	13	2	79.22
2	1	no	13	2	84.53
	1	yes	15	2	86.60
	2	no	14	2	98.17
	2	yes	16	2	100.76
	4	no	16	2	90.30
	4	yes	19	1	87.26
	complete	no	16	2	109.51
	complete	yes	16	2	108.22
4	1	no	18	1	98.66
	1	yes	20	1	107.96
	2	no	19	1	92.18
	2	yes	24	1	99.09
	4	no	24	1	88.21
	4	yes	29	1	94.01
	complete	no	27	1	119.82
	complete	yes	35	0	does not execute

may sometimes decrease performance: one example is shown with the configurations that are 1x2 rectangularly tiled with unroll factor of 4. In this case, the prefetching optimization reduces performance because only one thread block can be executing on the SM at a given time, as opposed to two thread blocks for the nonprefetching configuration.

Register spilling can sometimes enable an additional thread block to be resident per SM. Major performance differences are not demonstrated with the use of register spilling on this matrix multiplication kernel because two thread blocks, each with 256 threads, contain enough work to hide each others' global memory access latencies. Explicit register spilling with CUDA v1.0 only causes some configurations with two thread blocks to move to three blocks per SM, which results in a small performance increase at best. The 0.8 version of CUDA used in previous experiments [39] caused some configurations to move from one thread block to two per SM, resulting in a more significant performance increase.

The optimal configuration of matrix multiplication is a 16x16 tiled, 1x4 rectangular tiled, completely unrolled configuration. This result is initially nonintuitive to most developers: the configuration allows only a single thread block executing per SM, causing block synchronizations to have a more significant effect on scheduling. However, the sheer number of operations, due to rectangular tiling, creates a significant amount of ILP. This tradeoff between TLP and ILP will be unique for every application because of the variation in work granularity. A point I wish to make is that very few, if any, developers will arrive at this configuration by guessing a starting point and iteratively optimizing. For example, there is a continuous decrease in performance for unroll factors of 2 and 4 for the 16x16 tile, 1x4 rectangular tiled configuration.

There are a significant number of optimization configurations to be considered for an application as simple as matrix multiplication. Figure 5.3 shows the optimization space for an SAD kernel adapted from H.264, which is several times larger in code size. In this kernel, the best configuration uses fewer threads (just under 64) than a hand-optimized version (just under 128). The large "peaks" in the graph generally represent transitions where per-thread resource usage permits fewer thread blocks to be resident per SM, relative to the same configuration with slightly fewer threads per thread block. Because a developer is likely to emphasize thread-level parallelism in an initial version of the code, iterative approaches will begin at configurations towards the right side of the graph. It is unclear during an iterative optimization process whether superior performance can be gained by go-

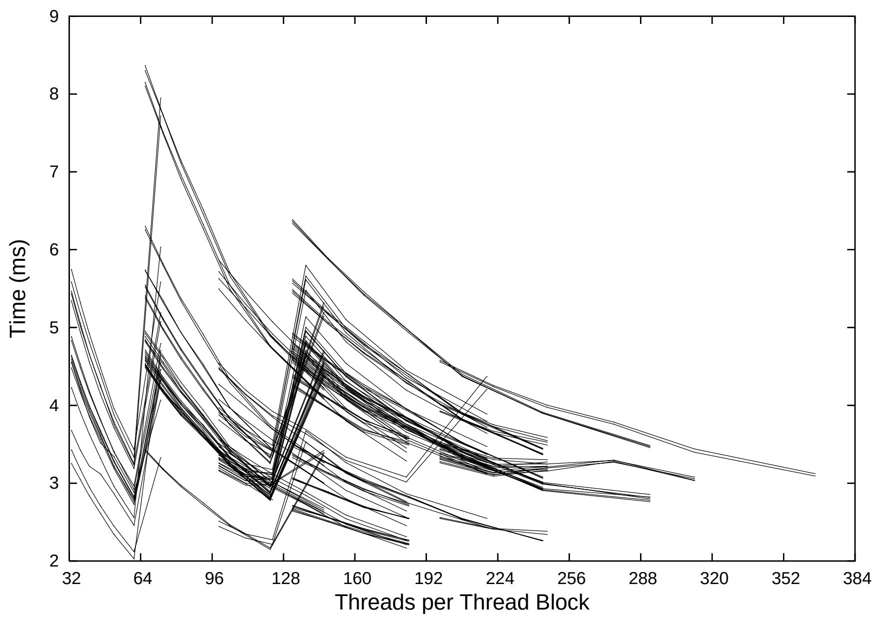


Figure 5.3: SAD optimization space.
Lines connect configurations that differ only in their SADs-per-thread parameter.

ing to fewer threads per thread block, especially since many of the configurations at the performance valley with approximately 60 threads per block have worse performance than configurations at the other valleys.

Previous work in optimization space exploration [39, 40] for this system discusses the optimization spaces and searches for SAD and other kernels in greater detail. The primary findings of these works are: For applications with multiple dimensions of optimization, no hand-optimized configuration was the best one found by an exhaustive search; the best was generally unexpected to the developers who optimized the kernel; and the best and hand-optimized configurations differed in several optimization parameters, making it unlikely that automated, iterative optimization using a greedy approach would have found the best configuration, starting from the hand-optimized one.

An expert with in-depth understanding of both the algorithm, including its behavior and memory usage patterns, and the hardware, including its memory bandwidth and resource characteristics, may have been able to bypass some of the pitfalls presented here. However, the goal of this work is to develop a technique that allows developers with imperfect understanding of the system to still find a near-optimal configuration.

5.2 Description of the Technique

Conceptually, optimization carving begins with the entire optimization space for an application. By examining metrics extracted from application configurations for the system, it removes configurations in the space that are unlikely to get high performance. This approach is in contrast to traditional compilation techniques that can be likened to oil drilling: an initial guess is made and strong effort is put forth to find a desirable result from that point. Each carving prunes the optimization space, eventually leaving a few potentially optimal configurations that can then be evaluated via hardware execution to determine the superior one.

There are several reasons why optimization carving is practical or reasonable, particularly for the GeForce 8800:

- The kernel codes studied here have a small number of independent configuration axes. The optimization search space is relatively small.
- The effects of code transformations, particularly in combination, are unpredictable. This situation is especially true for the GeForce 8800, because of the application developer's lack of control over the runtime's instruction scheduling and register allocation. Thus, iterative optimization is unlikely to find a configuration with close to the best performance.
- On single-chip, many-core architectures, configurations that are trapped in local performance maxima may be significantly removed from the optimal

in both performance and transformations, as shown in previous work [39, 40]. A partial search of the optimization space may achieve a substantial performance advantage over iterative optimization.

Optimization carving is done in order of performance impact. First-order issues are addressed first, then second-order, and so on. Carving must be done correctly to find a near-optimal configuration, but it is easier to understand and correctly model the high-order performance concerns of an architecture than the exact effects of optimizations. The technique as presented here begins with the complete optimization space and full knowledge of resource usage for this reason. How to perform more speculative optimization carving is left for future work.

I anticipate that optimization carving would be more effective if it is customized to the needs of each particular application. In addition, for architectures other than the GeForce 8800, more complex carving may be necessary to usefully prune the search space. For this work, I address the needs of the kernels studied in the previous chapter on a relatively simple but highly parallel architecture. This choice results in a simple, two-stage carving process that is easily explained.

Although it might seem that optimization carving has a sufficiently high cost that users may as well perform a full optimization space search, I assert that significant time savings is achieved with the technique. Kernel code dominates executed instructions and often execution time, but not static instructions or compile time, so static analysis can be very inexpensive. Table 5.2 shows compilation times for several GPU kernels and their test harnesses: compilation for discovering the static code and resource usage of the kernel takes significantly less time than complete compilation. Although the harness code does not need to be recompiled for every configuration, my point is that statically generating kernel code is less expensive than full compilation, even before factoring in compilation time for complete applications or execution time for each configuration.

Table 5.2: Compile Time for Several Kernels with Basic Optimizations

Kernel	-cubin Compile Time	Complete Compile Time
Matrix Multiplication	0.278 s	0.972 s
CP	0.300 s	0.887 s
SAD	0.288 s	0.873 s
MRI-FHD	0.266 s	0.893 s

I define two kinds of carving, *threshold carving* and *tradeoff carving*. Each has different selection criteria, discussed here.

5.2.1 Threshold carving

Threshold carving is performed when some performance aspect must be satisfied or mitigated in order to achieve good performance. The prime example on the GeForce 8800 is off-chip memory bandwidth: as shown in Figure 5.2, performance is not positively impacted by efficiency-increasing optimizations when the hardware is constantly stalled on off-chip memory accesses.

Selection for threshold carving involves pruning all configurations that do not surpass either a relative or absolute threshold, quantified in a metric. In the matrix multiplication example, the developer or compiler performing the carving could require that all global memory accesses be coalescible, or that the estimated bandwidth requirement of the application be below a certain limit. This requirement represents an absolute threshold. A relative threshold might be established for an application where memory bandwidth is always a bottleneck, in which case only the configurations with the best memory usage are evaluated.

5.2.2 Tradeoff carving

Tradeoff carving has a different nature from threshold carving: in some cases, it is not clear that one should maximize or minimize a particular performance aspect. An example mentioned in previous chapters is instruction stream efficiency: some redundant computation may improve performance by allowing more threads to execute in parallel or by reducing communication between threads. The purpose of tradeoff carving is to retain all configurations that balance two or more aspects of an application because the optimal balance is unclear.

Selection for tradeoff carving involves metrics for two or more aspects of an application. The configurations that are retained are those that lie on a Pareto-optimal curve: no point on the curve is inferior in every dimension to any other point in the space. Any configuration that is inferior in every dimension to any other configuration is pruned.

During tradeoff carving, good configurations may lie just off the Pareto-optimal curve because the metrics may not capture all performance aspects of the architecture. The probability of this situation is *higher* with more detailed metrics, particularly those that introduce more variables into the calculations, because any effects that are not exactly modeled are likely to inflate the metric values for poor-performing configurations. This possibility of "false precision" means that metrics used for tradeoff carving should either be very simple or extremely exact. Another method for avoiding this situation is to retain configurations that are within a certain distance from the Pareto-optimal curve, although the reduced pruning increases the number of configurations to be evaluated. I explore the use of more precise metrics in Section 6.4.2.

5.3 Optimization Carving for the GeForce 8800

In this section I discuss the use of optimization carving for applications executing on the GeForce 8800. To keep the demonstration of the technique simple, I use two carving stages. More carving stages are easily conceived; for example, if the application being optimized has a known issue with on-chip network conflicts or instruction cache thrashing, another carving stage can be created to address that.

5.3.1 Off-chip bandwidth

The first-order performance concern for the GeForce 8800 is global memory bandwidth. As shown in Section 5.1, only optimizations that affect global memory bandwidth have any effect when bandwidth is a performance bottleneck. Thus, the optimization space's first carving is a threshold carving targeting efficient global memory bandwidth usage.

Global memory bandwidth usage is easily estimated by examining the percentage of memory accesses in the instruction stream and determining the average number of bytes being transferred per cycle. The global memory coalescing effect can also be included by observing whether memory accesses in kernel code are to contiguous locations within a warp and are aligned (offset by a multiple of 16 from the beginning of the structure).

The performance of many of the applications studied here is limited by global memory bandwidth for versions that do not take advantage of reuse or coalescing. This carving focuses attention on those optimizations that improve bandwidth and in many cases eliminates it as a bottleneck.

5.3.2 Instruction stream efficiency versus execution resource utilization

When the performance of the GeForce 8800 is not limited by memory bandwidth capacity, it is determined by two factors: instruction stream efficiency and execution resource utilization. In many cases one can be traded off for the other with potential improvement in performance. An example of this tradeoff is the use of shared memory to capitalize on data reuse: instead of loading a value multiple times from global memory, a thread can load the value once, store it in shared memory (an extra instruction), and subsequently load the value from there. Even though additional instructions must be executed to initialize the shared memory, the thread no longer is stalled on global memory access and therefore can make faster progress. On the other hand, threads that use too much shared memory may cause fewer threads to be simultaneously executed per SM, potentially reducing performance.

The concepts of efficiency and utilization are very general and can be applied to any computer architecture. However, it is possible to calculate reasonably accurate metrics for applications executing on the GeForce 8800. The initial versions developed for this work are explained below.

$$Efficiency = \frac{1}{Instr * Threads} \tag{5.1}$$

Equation (5.1) estimates the instruction efficiency of the kernel to be run on the GPU by counting the total number of instructions that will be executed. *Instr* is an estimate of the number of dynamic instructions that will be executed per thread on the GPU, derived from the PTX code generated by nvcc. For this work, the average iteration counts of the major loops in the kernel are manually annotated to obtain this data. Some instructions count as multiple instructions;

32-bit integer multiplication is the primary example, taking multiple processor cycles to execute on the GeForce 8800. *Threads* is the number of threads that will run on the GPU for a given problem size, known to the developer when writing the code. This value is made explicit in the invocation of the kernel function and does not have to be an absolute value as long as the relative values of different configurations are correct.

In the absence of a memory bandwidth bottleneck and assuming nearly full SP utilization, I expect that efficiency will correlate directly to the performance of different configurations. Because it counts the total number of instructions executed, the metric measures the instructions that are redundant across threads. This metric penalizes configurations that have more redundancy, such as ones with finer-grained threads.

$$Utilization = \frac{Instr}{Regions}\left[\frac{W_{TB}-1}{2} + (B_{SM}-1)(W_{TB})\right] \qquad (5.2)$$

Equation (5.2) estimates the utilization of the compute resources on the GPU. The goal of this metric is to encapsulate the schedulability of warps in the system. Utilization is estimated primarily by looking at TLP and determining how often a warp is expected to wait and the amount of available work from independent warps. The fraction $\frac{Instr}{Regions}$ represents the average number of nonstalling instructions a single warp is expected to execute before encountering an instruction that causes it to stall. Again, *Instr* is the per-thread number of dynamic instructions that will be executed on the GPU. *Regions* is the number of dynamic instruction intervals delimited by warp-stalling instructions or the start or end of the kernel. Examples of warp-stalling instructions are those that consume the results of long latency operations (generally global and texture memory loads) and synchronization operations. SFU instructions are considered to have long latency when longer latency operations are not present. Issues such as memory bank and port conflicts, SFU oversubscription, and so on have been ignored because their effect is small compared to the stall time incurred by global memory accesses.

The quantity within the brackets represents the number of independent warps in the SM, other than the one currently executing, that can be executed while the warp-stalling instruction is being resolved.

- The first term in the bracket is the number of other warps in the same thread block as the currently executing warp. W_{TB} is the number of warps in a thread block, which is determined by dividing the number of threads in a thread block by 32. There is a division by two because if the blocking instruction is a synchronization instruction, on average half of the warps in the same block still need to execute until they also reach the synchronization point. This assumption represents the worst case, since load stalls do not have a similar requirement.

- The second term in the bracket is the number of warps from other thread blocks on the SM that can be executed. B_{SM} is the number of thread blocks assigned to each SM. The runtime assigns the maximum number of thread blocks possible to each SM, up to eight, without violating local resource usage. Consequently, this number can be calculated from the local resource usage obtained via `-cubin`.

Synchronization instructions are grouped with the consumers of long latency memory operations in order to simplify the calculation of the *Regions* term, even though they display different behavior. Execution at a barrier synchronization proceeds only when all of the threads in a thread block have reached that point, whereas stalls on uses of global load operations do not stall the execution of other warps. I anticipate that the division by two in the first term in the bracket captures the first order effects. The next chapter experiments with metric calculations of both lesser and greater detail.

There is a distinct upper limit on how efficiently the execution resources can be used. If utilization were used in a single cost metric (e.g., efficiency * utilization), it would be expected that the value would be capped or asymptotically approach the peak theoretical limit of 10 operations per cycle, per SM. However, because my intent is to use this metric as part of a Pareto-optimal selection, it is only necessary that the superiority or inferiority of a configuration relative to other configurations is retained. I do not model capped or asymptotic utilization because it is unnecessary for this purpose. Because of this decision, the relative utilization values of configurations may not be meaningful.

There are aspects of application behavior which are not modeled in the metrics. Some examples of these are cache behavior, SFU performance, the penalty incurred when threads in the same warp take different control paths, and memory bank and port conflicts. These are generally not first-order concerns for the kernels studied in this work, although I show one exception in the experiments.

As discussed previously, running nvcc with `-cubin` and `-ptx` flags is faster than full compilation of a kernel or application. Computing the efficiency and utilization metrics is relatively fast after this information and a few numerical inputs from the developer or a profile are obtained. This approach allows for fast exploration of a large search space.

5.3.3 Individual metrics and a single cost function

In the initial phase of this work, an attempt was made to create a single cost function that would approximate the performance of the code. This approach is akin to using static schedules to estimate EPIC/VLIW code performance. The

```
z = gridspacing * (float) k;

for (j=0; j<grid.y; j++)
    y = gridspacing * (float) j;

    for (i=0; i<grid.x; i++)
        x = gridspacing * (float) i;
        energy = 0.0f;

        for (n=0; n<atomarrdim; n+=4)
            float dx = x - atoms[n];
            float dy = y - atoms[n+1];
            float dz = z - atoms[n+2];
            energy += atoms[n+3] / sqrtf(dx*dx + dy*dy + dz*dz);

        energygrid[grid.x*grid.y*k + grid.x*j + i] = energy;
```

Figure 5.4: Coulombic potential grid code.

efficiency and utilization metrics both carry part of the information needed to predict the performance of a kernel configuration, but experience has shown that neither is sufficient in isolation for useful performance comparisons. The CP kernel is used as an example to show what aspect of performance is captured by each metric.

The CP kernel, shown in Figure 5.4, computes electric potentials at every point in a 3D grid. One of its uses is in setting up initial conditions for aqueous molecular dynamics simulations. In the original configuration, each thread computes the potential at a single grid point with a loop that processes one charge in each iteration. The optimization with the largest effect is tiling the computation at the thread level, computing multiple grid points per thread. This optimization greatly improves efficiency by eliminating redundant floating-point computation.

Figure 5.5 shows how CP's execution time and performance metrics vary with the results-per-thread tiling factor. The normalized reciprocals of the performance metrics are plotted, so lower is superior in both plots. The efficiency data points overlap and appear as a single curve. Efficiency closely follows the actual execution time at tiling factors of 1, 2, 4, and 8. Although utilization varies over this

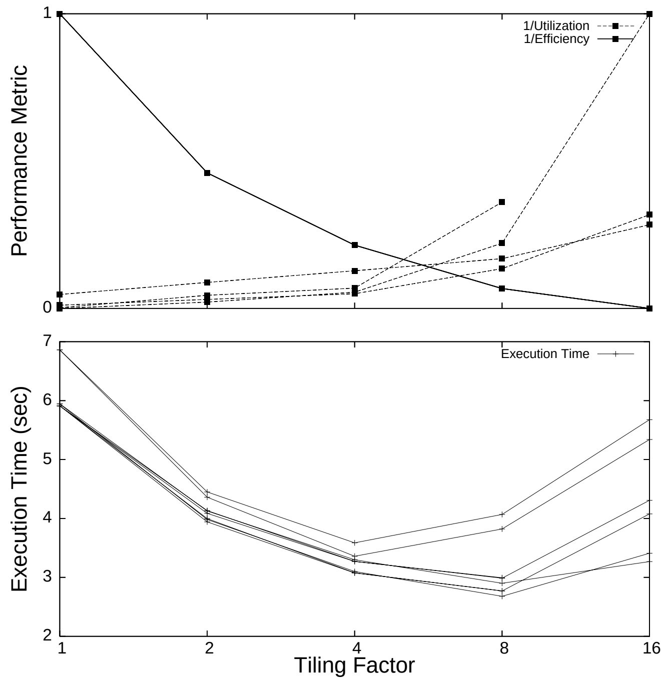


Figure 5.5: Performance metrics versus execution time for CP.

range, it remains sufficiently high that decreases in utilization do not significantly slow down the machine's execution throughput. At a tiling factor of 16, the decrease in utilization reduces the machine's throughput, countering a further increase in efficiency. For this kernel, efficiency improves monotonically while utilization worsens monotonically with increasing tiling factor, and the optimum configuration balances both metrics.

It should be noted that CP does not stress global memory bandwidth. In applications where memory bandwidth was often but not always a bottleneck, it was more difficult to estimate the performance of kernels using a single cost function.

5.3.4 Applying metrics to matrix multiplication

The matrix multiplication kernel shown in Figure 5.6 is used to demonstrate the calculation of the metrics. The kernel is first compiled with `-cubin` to obtain

```
Ctemp = 0;

// trip count = 256
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  // Region #1
  // It is assumed that both loads execute before
  // the stores, which mark the end of the region.
  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];

  // Region #2
  indexA += 16;
  indexB += 16 * widthB;

  // This synchronization marks the end of region #2.
  __syncthreads();

  // Region #3
  Ctemp +=
     As[ty][0] * Bs[0][tx];
  ...
  Ctemp +=
     As[ty][15] * Bs[15][tx];

  __syncthreads();
  // This is the end of region #3.
}
C[indexC] = Ctemp;
```

Figure 5.6: Matrix multiplication example for calculating metrics.
This code is the same as Figure 3.1(c) with regions delineated for clarity.

the resource usage, which shows that each thread uses 13 registers, and each thread block uses 2088 bytes of shared memory for its 256 threads. Determining the number of resident thread blocks per SM is done by referring to the per-SM resource limits in Table 2.2. In this case, register usage is the limiting factor: $B_{SM} = \lfloor 8192/(13 * 256) \rfloor = 2$. The number of warps per thread block is $W_{TB} = \lceil 256/32 \rceil = 8$.

The kernel is then compiled with `-ptx` to determine its execution profile. The loop is annotated with a trip count of 256, found by dividing the matrix size (4096) by the tile length (16). With this annotation, the number of dynamically executed instructions can be determined statically. A single thread executes 15 150 instructions, including 512 barriers and 256 load-consumer pairs: I assume that the load from `B` is performed prior to the store to shared memory array `As`, so that the global load latencies are overlapped and the store to `Bs` does not incur a stall. Thus, $Instr = 15\ 150$ and $Regions = 769$. The final piece of information needed is the number of threads in the kernel. There is one thread for each element of the 4k x 4k output matrix: $Threads = 2^{24}$. From these numbers, $Efficiency = 3.93 * 10^{-12}$ and $Utilization = 227$. The relationship of these metrics among different configurations is more meaningful than their absolute values. Even relative values are not necessarily comparable, since the utilization metric does not taper off to a constant value as would the true utilization of a system.

CHAPTER 6

EXPERIMENTS

This chapter discusses the use of optimization carving to find high-performance configurations of several applications. I also compare the results of the technique to random sampling of the entire optimization space, and examine the use of different computations for the utilization metric. I close the chapter with a discussion of the shortcomings of the technique as used here.

6.1 Methodology

The results in this chapter were obtained with CUDA version 1.0. Experiments were performed on an Intel Core2 Extreme Quad running at 2.66 GHz with 4 GB of main memory. The presented data represent runs with smaller inputs than those considered typical, which allows an exploration of the entire optimization space in a reasonable amount of time for comparison to optimization carving results. Informal experiments have shown that execution time will scale accordingly with an increase in input data size for these applications on this architecture, due to the regular and otherwise data-independent execution of the kernels. The data are gathered from a single run of each configuration; repeated experiments have shown that the gathered run times are reliable.

Table 6.1 lists the applications studied, the optimization parameters varied, the number of configurations in the optimization space, and the total time needed to evaluate the performance of every configuration in the optimization space.

Table 6.1: Parameter Search Properties

Kernel	Parameters Varied	Total Configurations	Total Evaluation Time
Matrix Multiplication (MM)	memory tile/thread block size, rectangular tile dimension, unroll factor, prefetching, register spilling	93	363.3 s
CP	thread block size, per-thread tiling, coalescing of output	38	159.5 s
SAD	per-thread tiling, unroll factor (3 loops), work per thread block	908	7.677 s
MRI-FHD	thread block size, unroll factor, work per kernel invocation	896	2875 s

These applications were selected from the larger suite presented in Chapter 4 because there were a large variety of optimizations that could be combined, with interesting effects and sometimes discontinuous results. The SAD kernel is the first kernel of the three in the H.264 application, discussed in Section 4.2.1. It is modified to not follow a spiral search pattern, thus avoiding the constant cache port conflicts encountered in the original version. Optimizations varied for carving were performed at the source code level; e.g., loops of different unroll factors were manually written and selected at compile time.

6.2 Initial Results

Figure 6.1 shows Pareto plots of the metric values for each optimization configuration for all of the applications. Threshold carving was performed only for matrix multiplication. The maximum metric values have been normalized to one for comparison purposes. Most Pareto plots are configured so that smaller values are superior and the optimal curves are close to the origin; I have kept larger values as superior because it is more intuitive, although it creates Pareto-optimal curves which may appear odd to readers. In general the best performance should come from configurations with both high efficiency and utilization, meaning points to-

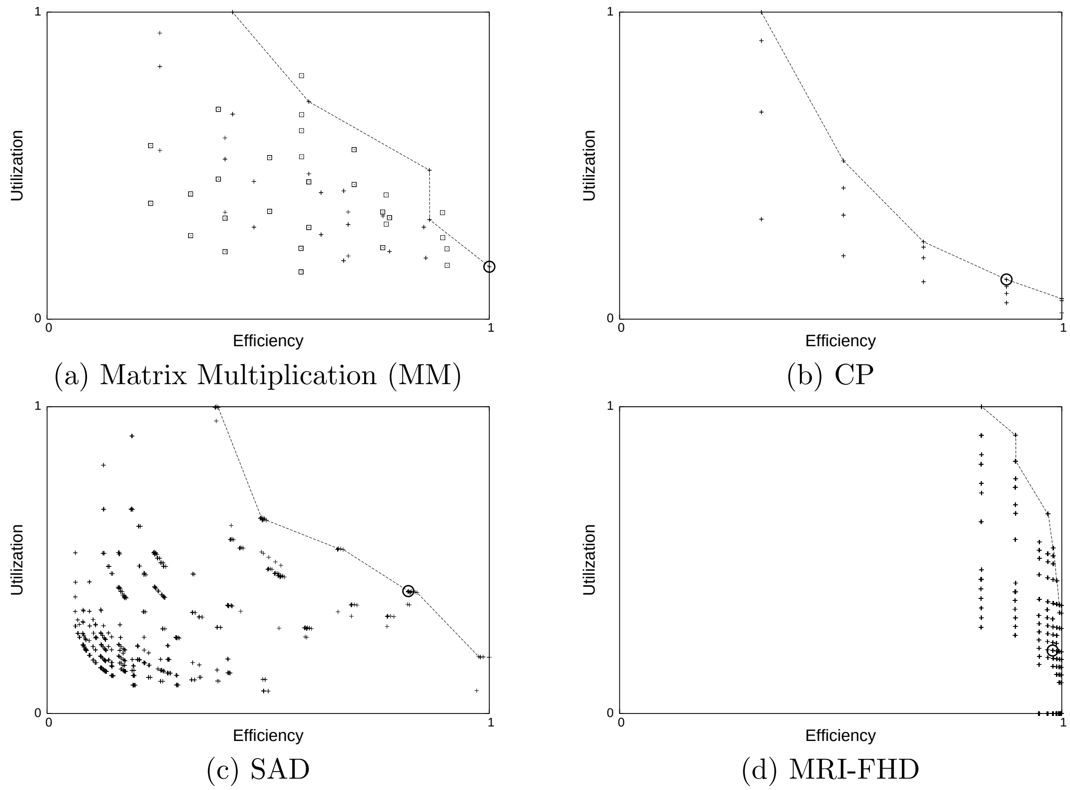


Figure 6.1: Optimization carving for four benchmark kernels.
The best performing configuration is circled in each graph. Configurations pruned by threshold carving are marked with squares rather than '+'. In (d), each point actually represents as many as seven configurations with indistinguishable efficiency and utilization.

wards the upper right corner of the graph. These points create a Pareto-optimal curve, which is drawn in the figure. The best performing configuration for each application is circled. Configurations that are inferior to another configuration in only one metric value are eligible for inclusion on the Pareto-optimal curve; for example, there are several configurations in CP and MRI-SAD that have the best efficiency and varying utilization values.

The matrix multiplication kernel has thread blocks of size 8x8 and 16x16 in its configuration space. Although a developer with detailed knowledge of global memory coalescing would know to exclude 8x8 blocks, the requirements for global memory coalescing were not well understood when these experiments were first

Table 6.2: Optimization Carving Space Reduction

Kernel	Selected Configurations	Space Reduction	Selected Evaluation Time	Time Reduction	Selected Best Relative to Overall Best
Matrix Multiplication (MM)	8	91%	10.2 s	97%	100%
CP	10	74%	42.95 s	73%	100%
SAD	19	98%	62.21 ms	99%	100%
MRI-FHD	58	93%	270.0 s	91%	99.2%

performed. I use this opportunity to show the effects of threshold carving: configurations pruned by threshold carving are marked with a square rather than '+'. This pruning changes the Pareto-optimal curve in subsequent tradeoff carving, as can be seen by the squares on the "outside" of the curve in Figure 6.1(a). These configurations run significantly more slowly than the plot would indicate because they are limited by memory bandwidth. A Pareto-optimal curve that includes these configurations will still find the best configuration, but at the cost of evaluating several configurations with poor performance.

Table 6.2 shows the number of configurations selected by carving and the resulting reductions in space and evaluation time, which were significant. It also shows the relative performance of the best configuration on the Pareto-optimal curve compared to the best performance found via exhaustive search. For three kernels, the Pareto-optimal subset contains the best overall configuration. The best configuration of the MRI-FHD kernel does not lie on the Pareto-optimal curve, but the second-best configuration does, with a performance difference of less than 1%. The variation in runs is relatively close to this difference, and there are several points on the Pareto-optimal curve that are within 2% of the best performance. As I discuss in Section 6.3, the MRI-FHD kernel has a large number of high-performance configurations.

It is difficult to make a judgment about whether a given value of utilization is "good" or "bad." As previously stated, the utilization metric measures the relative ability of a configuration to utilize the execution units, but the differences in values are of less importance. For example, consider that the best configuration for matrix multiplication in Figure 6.1(a) has one of the lowest utilization values of the entire space. The reason low utilization is not a reliable predictor of poor performance for matrix multiplication is because all configurations are quite good at ensuring that execution resources are almost always occupied. Thus, instruction stream efficiency is the primary determinant of performance for this kernel. This fact is not necessarily true for the other kernels.

Figure 6.1(d) shows the Pareto plot for the MRI-FHD application. In this graph, configurations tend to be clustered in groups of seven, appearing as a single point. This effect is due to the fact that MRI-FHD is not a blocked algorithm in the way that matrix multiplication is, so changes in thread block size affect neither the efficiency nor the utilization of this kernel. Differences in actual performance within each cluster are small, generally a few percent. Hence, when several configurations have identical or nearly identical metrics and similar optimization parameters, it may be sufficient to randomly select a single configuration from that cluster, rather than evaluating all the configurations.

6.3 Comparison to Random Sampling

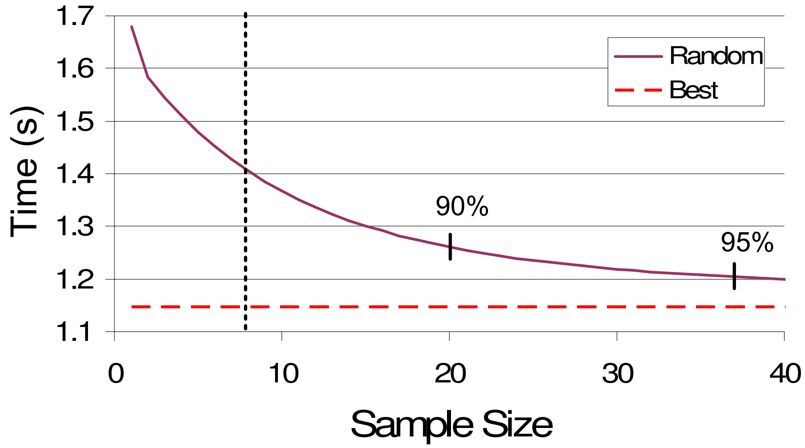
The last stage of optimization carving requires execution of the remaining configurations to determine the best one. A question that arises is whether a random sample of the optimization space might be capable of achieving similar results. In this section I compare random sampling of the optimization space to the results of optimization carving.

Table 6.3: Random Sampling Results

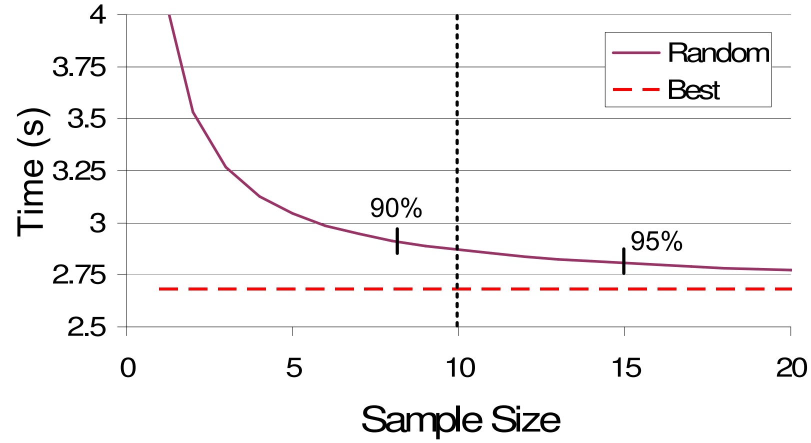
Kernel	Best Configuration Time	Optimization Carving Configurations	Expected Maximum Performance of Same Size Random Sample	Random Sample Performance Relative to Best	Random Sample Size for 90% of Best	Random Sample Size for 95% of Best
MM (16x16 only)	1.147 s	8	1.404 s	81.7%	20	37
CP	2.679 s	10	2.869 s	93.4%	8	15
SAD	2.029 ms	19	2.479 ms	81.8%	78	93
MRI-FHD	3.727 s	58	3.763 s	99.0%	2	4

The value of interest in random sampling is the highest performance of the configurations in the sample. I term this value the *expected maximum performance* of a sample. Conceptually, a sample of unit size has an expected maximum performance equal to the expected value (arithmetic mean) of the space. On the other end of the scale, a sample consisting of the entire space would provide the maximum performance in the space. This calculation is not described in any introductory statistics or probability text I have referenced, so I describe it in Appendix A.

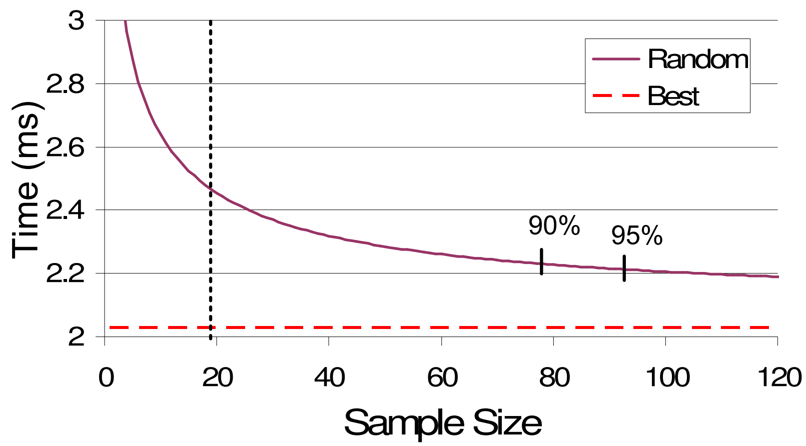
Figure 6.2 shows the expected maximum performance of random samples of varying size, while Table 6.3 shows the size of the samples required to have an expected maximum performance of 90% and 95% of the best configuration. For matrix multiplication I use only 16x16 thread blocks, since thread blocks that do not take advantage of global memory coalescing have mediocre performance (shown in Figure 5.1 on page 74) and can be eliminated prior to sampling. A vertical dotted line corresponds to the number of selected configurations from optimization carving for comparison.



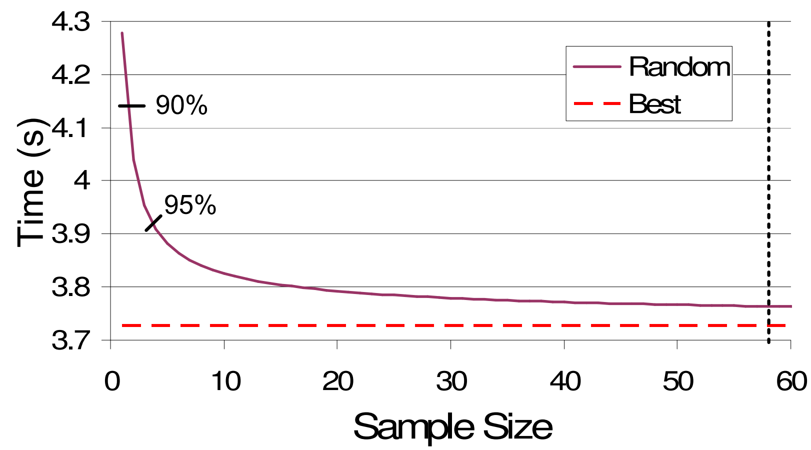
(a) MM (16x16 only)



(b) CP



(c) SAD



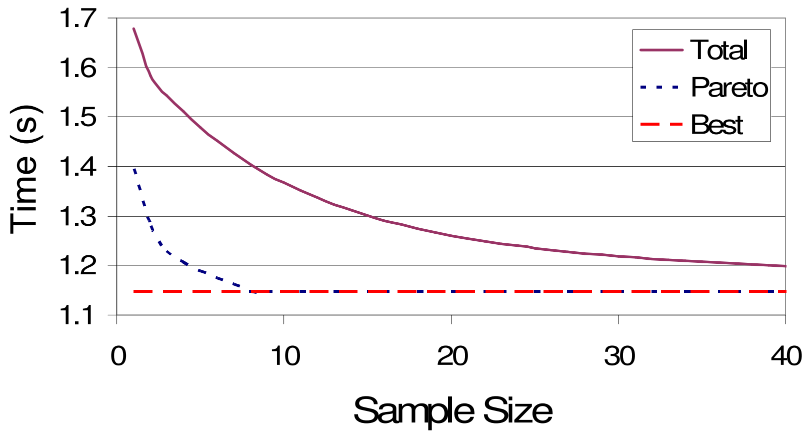
(d) MRI-FHD

Figure 6.2: Expected maximum performance of a random sample of the optimization space. A vertical dotted line marks the number of configurations that optimization carving selects.

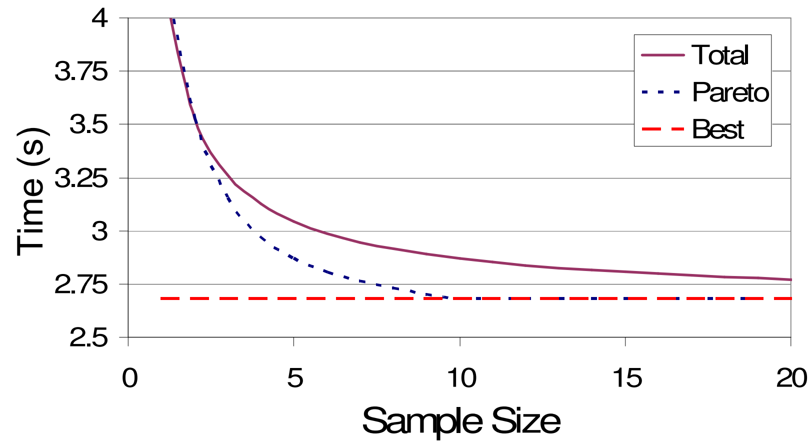
Random sampling is an effective method for finding good configurations for the CP and MRI-FHD kernels, with an expected maximum performance close to the best configuration's performance with a small random sample. It is effective because there are many configurations with near-best performance in the optimization spaces of those kernels.

Matrix multiplication and SAD require much larger random samples for a near-best configuration to be expected. Unlike CP and MRI-FHD, there are only a few configurations with performance near the best, and even these can still be several percent away from the best. The right combination of local memory tiling, register tiling, and loop unrolling gives significant performance advantages to the highest-performing configurations.

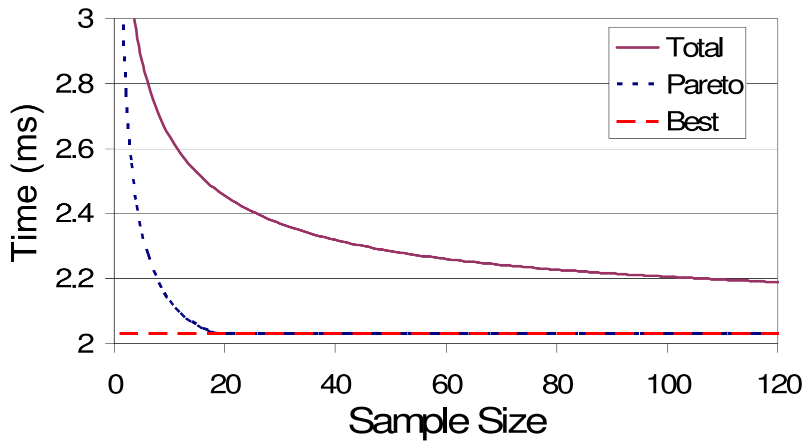
One possibility that should be examined is whether randomly sampling the pruned space produced by optimization carving can find a near-best configuration effectively. The dotted lines in Figure 6.3 show the expected maximum performance of a random sample of the Pareto-optimal configurations selected by optimization carving. The important observation is that, with the exception of matrix multiplication, small samples of the Pareto-optimal configurations may not be significantly better than sampling of the entire space, and are worse for MRI-FHD. Not every Pareto-optimal configuration should be expected to have good performance; one should only expect that the best performing configuration is Pareto-optimal. Larger samples of the pruned space often have a much better expected maximum performance because it is more likely that the best or a near-best configuration is included in the sample, since the pruned space is smaller. MRI-FHD is a notable exception, where optimization carving is not significantly superior to random sampling. It appears that variations in the performance of the MRI-FHD kernel are due to differences in how the configurations interact with the constant cache and SFUs, neither of which is modeled in the metrics.



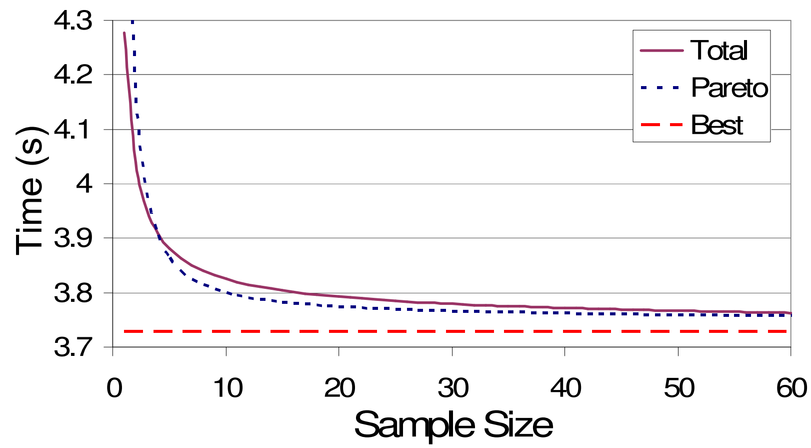
(a) MM (16x16 only)



(b) CP



(c) SAD



(d) MRI-FHD

Figure 6.3: Expected maximum performance of random sampling of the Pareto-optimal configurations from optimization carving and the entire optimization space.

6.4 Varying Metrics

The key factors in the usability of the technique are the effectiveness of relatively simple metrics and the results' robustness to approximations and inaccuracies in the metrics. In this section I develop both simpler and more complex versions of the metric for execution resource utilization. I use these to perform tradeoff carvings and compare the results to those of the initial metrics.

6.4.1 Simpler metric: Discounting synchronization effects

Here, I create a simpler version of the utilization metric and test its effectiveness versus Equation (5.2) in reducing the search space while still finding a near-best optimization configuration. In selecting which aspects to include in the simpler metric, I observe that stalls due to global loads and the longest latency operations are the greatest detriment to machine utilization for our application suite. The primary way to avoid these losses is to execute more threads in parallel per SM. This intuition is codified into Equation (6.1). I remove the stalling effects of synchronizations and assume that barrier synchronization is no more expensive than any other instruction. Instead of regions being delineated by consumers of global loads and synchronizations, I now only count regions delineated by consumers of global loads or the longest latency operations, represented by *LongRegions*. Because the effects of synchronizations are no longer being considered, all other warps are potentially available for execution during a stall, so I calculate the total number of warps executing on the SM and subtract one. Compared to the previous utilization calculation, this metric will increase, relative to other configurations, the utilization values of configurations that have fewer thread blocks with many warps and do not perform prefetching.

$$Utilization_{simple} = \frac{Instr}{LongRegions} \left[B_{SM} W_{TB} - 1 \right] \quad (6.1)$$

The reason I remove the effects of synchronizations is that the previous metric potentially "double-counts" the effects of load stalls. First, barrier synchronization on the GeForce 8800 appears to be extremely fast, unlike on traditional multiprocessing systems. Thus, synchronizations stall execution of a thread block only when the warps that have not reached the barrier cannot execute useful instructions due to another stall. These stalls will generally be due to loads from texture and global memory that have not returned their values. However, most synchronizations exist to ensure that values have been stored to shared memory for use by all threads, and these preceding stores often consume the results of global loads. Thus, a warp generally encounters a synchronization stall when another warp is stalled on the use of a load.

In addition, a combination of a good instruction schedule, a good dynamic thread scheduling policy, and many warps per SM may ensure that synchronizations rarely cause significant stalls. Consider the matrix multiplication instruction schedule in Figure 6.4, corresponding to the code in Figure 5.6 on page 90. When a warp enters the loop, loads of elements in arrays `A` and `B` are executed for the first warp. No forward progress can be made in that warp because the next instruction is a store of value being loaded, so execution proceeds to other warps in the thread block. Once those have stalled, execution proceeds to another thread block if multiple are simultaneously running on an SM. Once the results of the first set of loads arrive, the first warp can execute loads and other instructions before encountering the synchronization barrier. Due to MLP, several other warps' load results have also arrived by that time and execution can continued unstalled until the last warp in the thread block reaches the barrier. Because all other warps

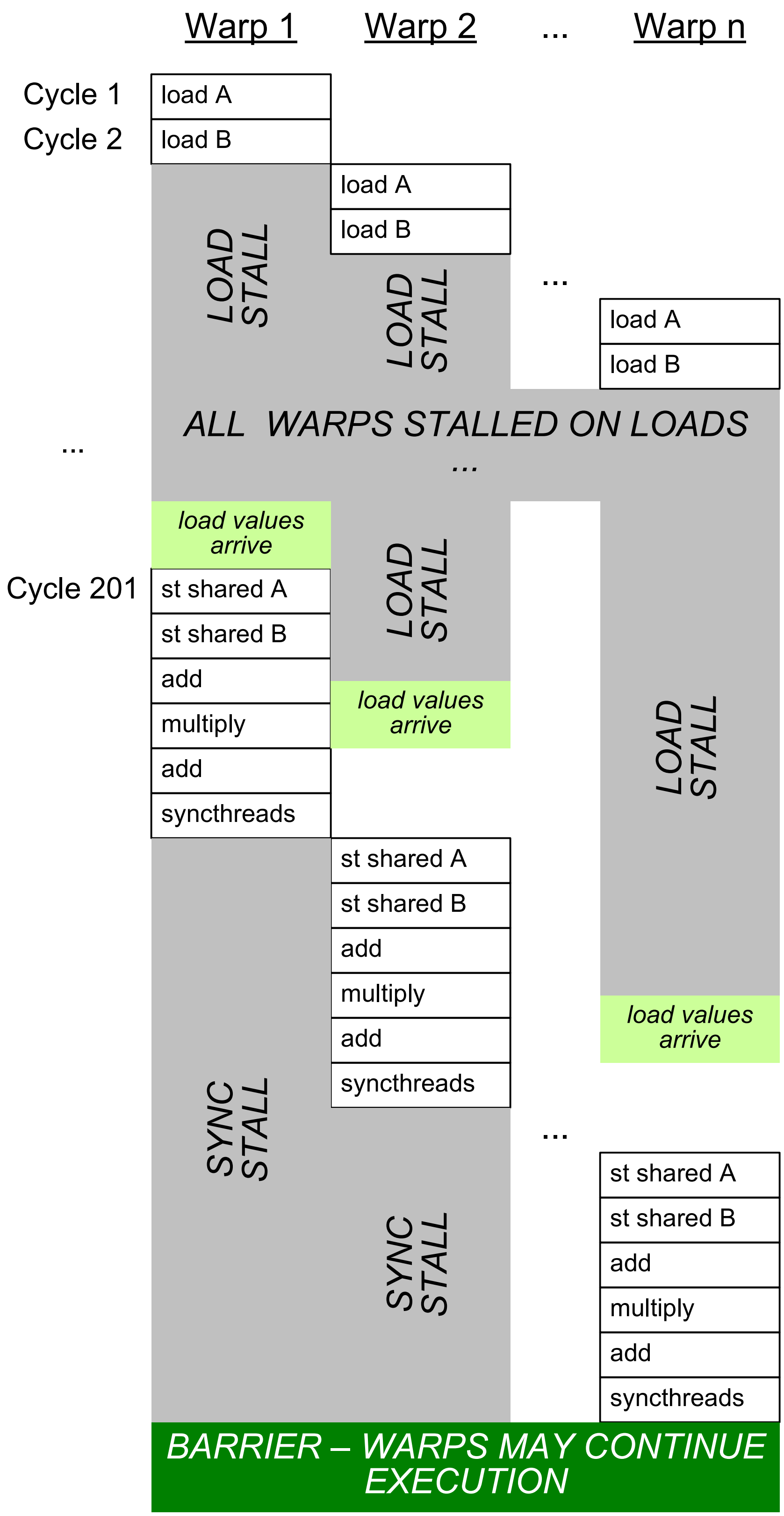


Figure 6.4: A hypothetical instruction schedule for a thread block of the matrix multiplication kernel from Figure 5.6.

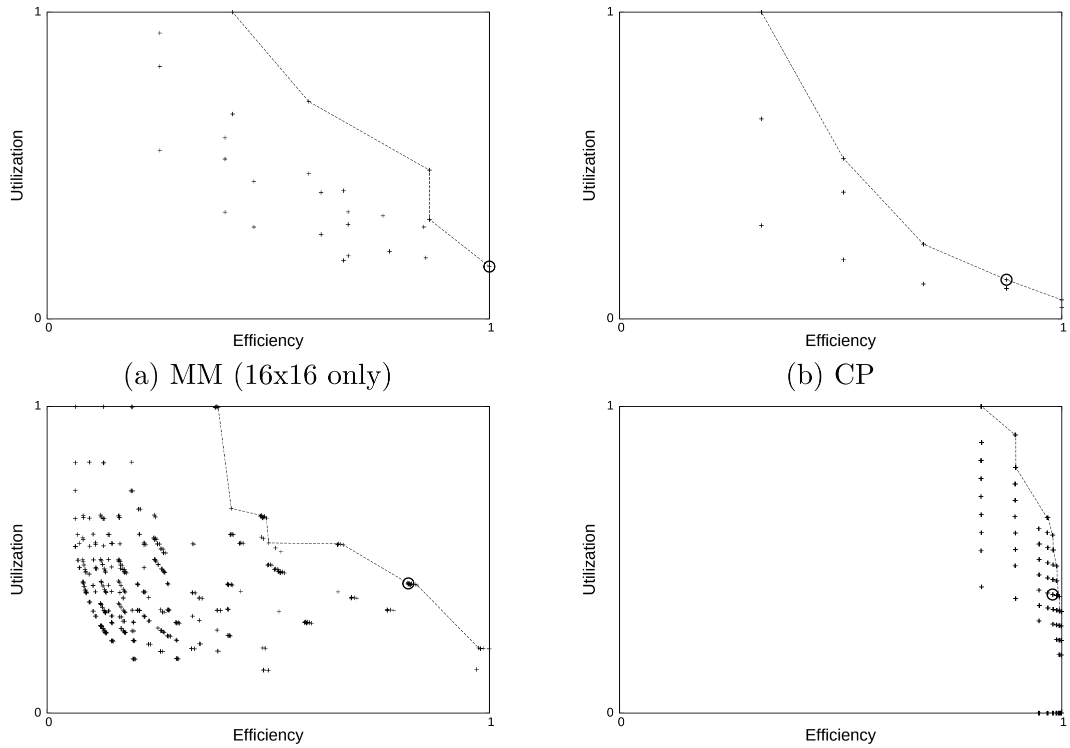
have reached the barrier, execution can proceed unabated past the barrier. With this schedule, the barrier never causes execution to stall on the SM.

Finally, there are no synchronizations in the inner loops of the kernels other than matrix multiplication. The previous metric penalizes configurations with larger thread blocks due to the division by two: warps in the same thread block should theoretically be equally schedulable as warps in different blocks for kernels without synchronization.

Figure 6.5 shows the results of optimization carving using the simplified utilization metric for the four applications. Matrix multiplication carving is performed only with 16x16 thread block configurations for clarity. Some shifting of the locations of configurations has occurred, but in general the plots appear similar to results using the original metric calculation, with the largest difference being further clustering of formerly scattered configurations in the MRI-FHD plot. For matrix multiplication, the Pareto-optimal line has not changed from Figure 6.1(a), containing the same configurations. Similar to the original metric calculation, the best configuration is on the Pareto-optimal curve for all but the MRI-FHD plot.

Table 6.4 compares the number of points on the Pareto-optimal curve using the original and simplified utilization metrics. In applications other than matrix multiplication there is a significant increase in the number of points on the curve, mainly because there is less to differentiate configurations in the simplified metric. Even though the simplified metric still serves its purpose, for three kernels a larger number of configurations must be evaluated to find the configuration with the highest performance.

The MRI-FHD kernel is notable in that even though the simplified metric produces a Pareto-optimal curve with three times the number of configurations of the original utilization metric, no configuration on the curve is closer than 5% in performance from the best configuration. Given that 152 configurations are



(a) MM (16x16 only) (b) CP

(c) SAD (d) MRI-FHD

Figure 6.5: Optimization carving using the simplified utilization metric.

evaluated, the simplified metric fares poorly compared to random sampling of the optimization space. As stated previously, the primary difference between the simplified and original utilization metrics is that the simplified metric improves the standing of configurations that have fewer thread blocks and more threads per block. A simple experiment shows that there may be a significant performance penalty when a single thread block with heavy SFU usage is executed per SM, as opposed to multiple, smaller thread blocks.

Table 6.4: Space Reduction Using Simplified Utilization Metric

Kernel	Total Configurations	# of Original Selections	# of Simplified Selections	Performance Relative to Best
MM (16x16 only)	45	8	8	100%
CP	38	10	18	100%
SAD	908	19	23	100%
MRI-FHD	896	58	152	94.9%

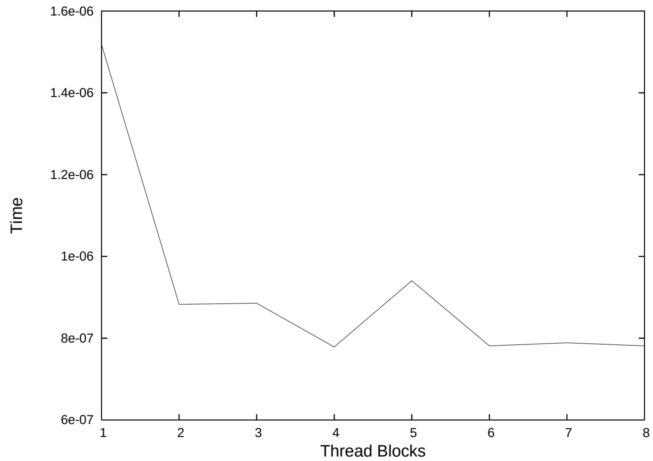


Figure 6.6: Reciprocal square-root execution time for varying numbers of thread blocks per SM.

The test kernel for this experiment consists of a kernel that loops over a single reciprocal square-root operation 2000 times. The number of thread blocks on an SM is varied while the total number of threads is kept roughly constant (slight variations are made to create an integral multiple of half-warps, to avoid impacting instruction scheduling and the memory system). Figure 6.6 shows the results. Although the variations from 2 to 8 thread blocks may be due to variations in the total number of threads, there is a clear performance loss when a single thread block is running on an SM. The MRI-FHD kernel illustrates a situation where behavior of an architectural feature was not anticipated or initially understood by application developers or optimizers, making the metric and the optimization carving technique less effective.

6.4.2 Modeling cycle count

It is evident that the metrics displayed so far are relatively crude. I have discussed the combined effects of ILP, TLP, and MLP but have not attempted to relate their effects within a single metric until this point. Here, I attempt to capture the behavior of the application kernels more precisely by factoring in the stall time and work performed by each thread as a fraction of a single thread block's minimum execution time.

In the previous utilization calculations, a region represents a nonstalling instruction sequence and by association the stall period between it and the next region. This abstract stall time is assumed to be the same for all configurations. In reality, ILP and MLP effects can reduce the exposed stall time for a single warp by executing other instructions after a load operation. They have a multiplicative effect with TLP. Kernels with low TLP and high ILP compared to other configurations are unfairly penalized with the previous utilization calculations, and may have much higher utilization and overall performance than the metric predicts. I shown an example here.

Consider the code in Figure 6.7, taken from a 1x4 rectangularly tiled matrix multiplication kernel with 16x16 thread blocks. The first load will take approximately 200 SP cycles to return its values. Four more load operations can execute immediately after the first. These five loads execute in 20 SP cycles, after which the warp must stall for load results. In a 16x16 thread block, there are a total of 8 warps, and each warp can execute for 20 cycles before encountering a stall. Thus, approximately 160 cycles of the first load's latency are covered by the execution of other load instructions in a single thread block. High ILP is one of the factors that enables the 1x4 rectangularly tiled kernel to achieve the best performance of the matrix multiplication configurations.

Figure 6.8 depicts how the scheduling of warps hides a single warp's compulsory stall time. Compulsory stall time is the latency from loads or other long-latency operations, reduced by the execution of operations from the same warp after the long-latency operation is issued. Work from other threads hides the compulsory stall time and helps the SM avoid stalls. Conceptually, utilization should represent the percentage of execution time where the SM is not stalled. Here, I attempt to

```
ld.global.f32   $f5, [$rd9+0];
ld.global.f32   $f6, [$rd16+0];
ld.global.f32   $f7, [$rd16+64];
ld.global.f32   $f8, [$rd16+128];
ld.global.f32   $f9, [$rd16+192];
st.shared.f32   [$rd20+0], $f5;
st.shared.f32   [$rd23+0], $f6;
...
```

Figure 6.7: Example PTX code from a matrix multiplication kernel with 1x4 rectangular tiling.

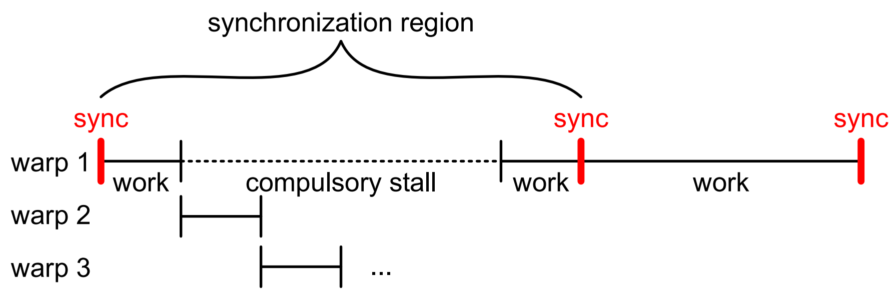


Figure 6.8: Scheduling warps to hide compulsory stall time.

calculate utilization by determining the amount of compulsory stall time that is not covered by the work from the thread blocks executing on the SM.

It should be noted that within a *synchronization region*, or the region between two barrier synchronizations, warps in the same thread block must be executing code within the same region. However, work in other thread blocks can come from any part of the kernel. This fact should be correctly factored into the calculation of the metric. For example, the synchronization region with global loads in matrix multiplication can have relatively few instructions compared to the synchronization region that performs the partial dot product calculation. A relatively small percentage of the thread block's total work is available to cover the latency of the global loads.

Another important detail is that only a portion of the work of other thread blocks is available to cover a stall, since other thread blocks will encounter their own stalls. I assume that the amount of this work is the total number of instructions executed by a thread divided by the total number of long latency stalls, or $\frac{Instr}{LongRegions}$. For the simple kernels used here, using this value is probably a reasonable assumption. However, there is a possibility that thread blocks running on the same SM will have a tendency to be either in-phase or out-of-phase with each other, with potential increases or decreases in the number of instructions available for execution during a stall. How to predict this situation and include it in the utilization metric is left for future work.

I first introduce the concept of a thread block's minimum execution time for a synchronization region, $mintime_{sr}$, shown in Equation (6.2). The letters sr stand for synchronization region. The available work from a single warp in a synchronization region is $warpwork_{sr}$, calculated directly from the instructions between the synchronization operations. The compulsory stall time for a single warp in each synchronization region is $stall_{sr}$ and is measured by taking the latency of a long-latency instruction (200 cycles for global memory loads) and subtracting the work from operations between the long-latency instruction and its consuming instruction in the same warp. If the warps in a thread block can cover all of the latencies, then the minimum execution time is equivalent to the total number of work cycles in the thread block. Otherwise, some part of the compulsory stall time is exposed and must be covered by work from other thread blocks. One important note is that this calculation and the following ones are optimistic about the ability of other warps' work to cover a stall: the warps may encounter their own stalls after performing a fraction of their work in the synchronization region, as depicted in Figure 6.8.

$$mintime_{sr} = warpwork_{sr} + \max[stall_{sr}, (W_{TB} - 1)warpwork_{sr}] \tag{6.2}$$

I next calculate the amount of work that an SM can execute during a synchronization region, $work_{sr}$, shown in Equation (6.3). The work from a thread block is added to available work from other thread blocks. Only a portion of the other thread blocks' work is available to cover a stall, as I previously described, but there may be multiple stalls within a synchronization, represented by $LongRegions_{sr}$. The combined work may completely cover the latencies in the synchronization region, in which case the value of $work_{sr}$ is capped at $mintime_{sr}$. I cap the value because the SM reaches maximum utilization when all latencies are hidden.

$$\begin{aligned} work_{sr} = \min \Bigg[& mintime_{sr}, W_{TB} \cdot warpwork_{sr} \\ & + \frac{(B_{SM} - 1)(W_{TB}) Instr \cdot LongRegions_{sr}}{LongRegions} \Bigg] \end{aligned} \tag{6.3}$$

Equation (6.4) shows the new utilization metric, which approximates the fraction of execution time that the SM is executing instructions. The denominator represents the total time that a single thread block requires to execute the code, calculated by adding up the minimum execution times for all synchronization regions. The numerator is the total amount of work that an SM can schedule during the period represented by the denominator. Capping the value of $work_{sr}$ at $mintime_{sr}$ for each synchronization region prevents extra work in one synchronization region from covering the latencies in another synchronization region, which was possible in the previous utilization calculations but cannot happen in reality.

$$Utilization_{cycle} = \frac{\sum^{all\ sr} work_{sr}}{\sum^{all\ sr} mintime_{sr}} \tag{6.4}$$

I make several assumptions about SFU behavior. SFU latency is assumed to be 20 cycles. I assume that the SM can issue instructions to SPs after four cycles of issuing SFU instructions, but must finish issuing SFU instructions before issuing another set of SFU instructions; otherwise, the SM must stall. Finally, I assume no effects from execution capacity limitations are exhibited. A related assumption I make is that warps in the same thread block are as schedulable as warps in other thread blocks, although Figure 6.6 shows otherwise.

Because I do not have access to a cycle-accurate scheduler, I manually scheduled instructions and calculated the work and exposed stall time for each configuration. SAD and MRI-FHD are not included in this experiment because of the large number of configurations in those kernels. I assume that the compiler and runtime schedule instructions to maximize MLP and minimize compulsory stall time. It should be noted that this assumption can create significant error if it is incorrect: if two load latencies are sequential instead of overlapped, a warp will have twice as much compulsory stall time.

Since CP generally benefits from global memory coalescing, for the baseline comparison I remove noncoalesced configurations from the optimization space. Tradeoff carving selects nine configurations for evaluation, as shown in Figure 6.9; only one of the configurations on the original Pareto-optimal curve in Section 6.2 did not have coalescing.

Table 6.5 shows the results of using the new utilization calculation on the MM and CP kernels. Unlike the previous experiments, I no longer include configurations that are equivalent in one metric value and inferior in another compared to other configurations. The reason is that all configurations in both kernels have maximum utilization: the available work should be able to cover all latencies in each synchronization region.

Only the best MM configuration is selected by carving. The advantage of capping the utilization for each synchronization region is that carving can determine that all configurations have very high utilization. In reality, the best MM

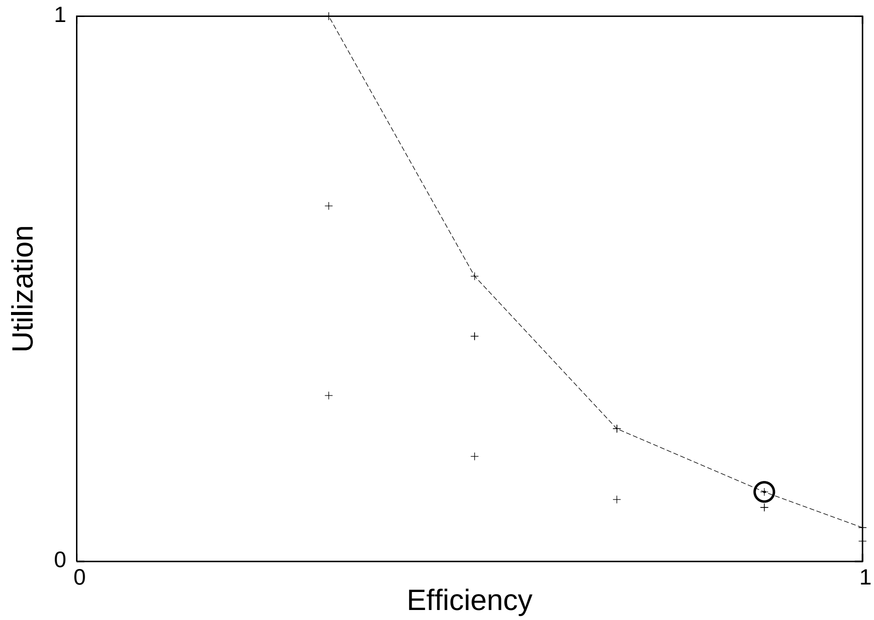


Figure 6.9: Tradeoff carving using the original utilization metric calculation for CP with coalesced configurations.

Table 6.5: Space Reduction Using Cycle-Based Utilization Metric

Kernel	Total Configurations	# Original Selections	# Cycle-Based Selections	Performance Relative to Best
MM (16x16 only)	45	8	1	100%
CP (coalesced only)	19	9	3	79%

configuration should have lower utilization than other configurations: the warps in the single thread block cannot completely cover the stall time of the first load, since they cannot execute all of their work before encountering their own stalls. This inaccuracy could be corrected by splitting synchronization regions into several work-and-stall regions, similar to the *LongRegions* of the previous utilization calculation. Refinement of the calculation is left for future work.

The Pareto-optimal configurations for CP have a loop unroll factor of 16, as opposed to the best-performing configuration with an unroll factor of 8. More unrolling gives those configurations higher efficiency at the cost of fewer total warps per SM (4 or 6) than the best configuration, which has 10 warps per SM.

The second- and third-best configurations, which have the same efficiency as the best configuration, have 8 warps per SM. It appears that the additional warps in the best configurations have a positive effect on utilization that is not modeled by the metric. One likely reason is that the the code being executed on the system may be essentially the same as the PTX code, which places the consumers of the results of reciprocal square root operations directly after their producers. The manually scheduled code used as the input for the metric calculation moves those instructions downward in the schedule, for much higher ILP and far lower compulsory stall time. If the executed code resembles the PTX code, SFU latencies will be exposed and the additional warps will be necessary to cover the latencies.

The CP case illustrates a potential danger of using more precise metric calculations: unless they capture execution details very accurately, they may inadvertently penalize and possibly prune top-performing configurations. Accidental pruning can be mitigated by not pruning points that are close to the Pareto-optimal curve, which is counter to the purpose of using more precise metrics. Given this danger, the use of precise metrics will probably be most useful when the optimization space needs to be severely pruned, even at the risk of missing the best configuration in the overall optimization space.

6.5 Impact of Performance Factors Not Included in Metrics

Although the configurations selected by optimization carving contained a near-best optimization configuration for the test kernels, one should examine the effectiveness of the technique when the metrics do not incorporate certain aspects of performance behavior. I observe one aspect here, that of cache behavior.

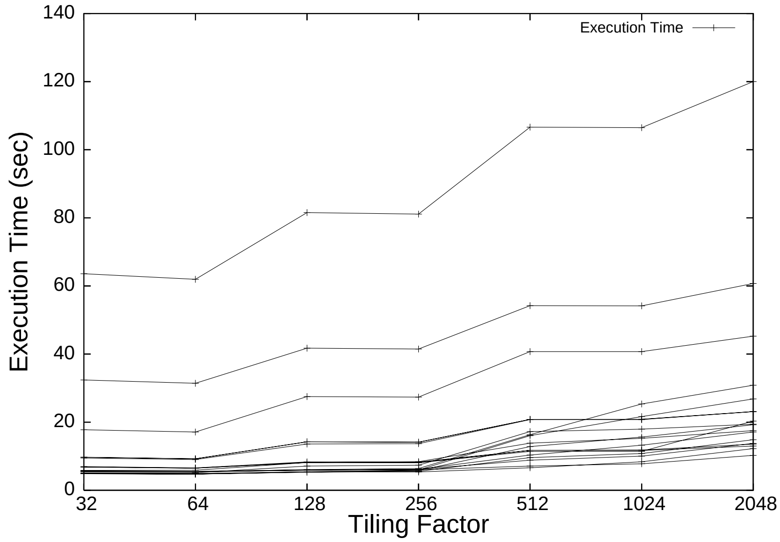


Figure 6.10: Execution time of MRI-FHD with cache conflicts.

Consider Figure 6.10, which depicts the performance of a preliminary version of the MRI-FHD kernel as the tiling factor (number of data points processed by each thread block) changes. The performance metrics indicate that efficiency and utilization remain constant as the tiling factor changes, predicting no significant change in performance. However, experiments revealed that performance decreases as the tiling factor increases, as shown in Figure 6.10.

The sharp contrast between the predicted performance trend and the actual performance led the developer of the MRI kernels to consider that the layout and traversal of data in constant memory might be causing frequent constant cache conflicts. Changing the data layout yielded a kernel that is insensitive to changes in the tiling factor and performance up to 17% faster than the previous best configuration.

6.6 Future Work

In this section I discuss future work related to program optimization carving. One thrust concerns methods to reduce the amount of compilation or configurations searched for optimization carving. Another thrust is alternate approaches for finding high-performance application configurations. Finally, I discuss the possibility of architectural support for mitigating the discontinuity of the optimization space and tools to help developers in the optimization process.

The effectiveness of the metrics would improve with the inclusion of effects of architectural features that are first-order performance determinants for some applications. The MRI-FHD kernel is the prime example of the need for more refined metrics. More precise modeling would also enable the combination of the efficiency and utilization metrics into a single cost function. Even if the cost function assumes no memory bandwidth limitations, it would remove the need to do a tradeoff carving, replacing it with a threshold carving based on a combined cost function. As stated previously, so far I have not been able to find a sufficiently accurate cost function that consistently finds near-best configurations across multiple, relatively simple kernels.

It is feasible for a runtime system like that of the GeForce 8800 to automatically perform many of the code optimizations used by optimization carving. If performance feedback information were produced during execution, it could be consumed by the runtime to optimize kernels, enabling speedup of the kernel prior to its completion. Run time modification of kernels is relatively easy to manage for programs written for CUDA, due to the independence of thread blocks: newly initiated thread blocks can execute optimized code, while thread blocks already running can complete execution of unoptimized code.

The fundamental issue that optimization carving addresses is the difficulty for either an automatic system or a human developer to determine what mix of optimization dimensions, such as tiling, will achieve high performance on the GeForce 8800. However, it is often much easier to determine which of two similar configurations will have greater performance. For example, most of the matrix multiplication kernels studied here have decreased performance when unrolling by a factor of four because the number of thread blocks simultaneously executing on an SM is reduced compared to configurations unrolled by a factor of two. Since the number of thread blocks per SM cannot change further when unrolling completely (assuming no oversubscription of registers), it is likely that complete unrolling will have results superior to those of unrolling by a factor of four. It would be desirable to avoid compiling or evaluating configurations with poor unrolling factors. This approach requires either precise knowledge about the effects of optimization or prediction of their effects, similar to work by Zhao et al. on codes for embedded processors [41].

A more directed approach for optimizing code for the GeForce 8800 is to target specific granularities of parallelism and maintain them throughout the optimization process. Instead of trying various optimizations without regard to their effects, the compiler can compile multiple configurations that have varying values of specific characteristics, such as the number of thread blocks simultaneously executing per SM or the degree of register tiling. It can then control optimizations that allow configurations to stay within the specified limits. Although the compiler will still need to compile multiple configurations, targeting specific granularities would reduce the number of configurations that need to be compiled, effectively targeting the configurations on the Pareto-optimal curve from tradeoff carving. This approach also requires careful study of optimization phase ordering and prediction of optimization effects in order to prevent the accidental bypass of desirable configurations.

One major issue with the GeForce 8800 architecture is that the number of thread blocks assigned to each SM is directly related to a thread block's local resource usage. Although this issue is a fundamental limitation for many-core processors, the problem is exacerbated by unpredictable performance changes when small changes are made to the kernel. It may be possible to create support in the runtime to automatically spill registers when it would allow significantly better utilization of execution resources. This technique could be enhanced with architectural support to spill registers to unused local memory, avoiding additional burden on global memory bandwidth.

A more integrated and structured environment for optimizing kernels would make the application developer's efforts more focused and efficient. Environments such as those constructed by Adve et al. [42] have been constructed for past systems, languages, and compilers. The issue is complicated by the opacity of some architectural feature of the GeForce 8800, but the basics of global memory bandwidth, instruction efficiency, and resource utilization can be combined with visualization tools to improve developer efficiency. This approach can be integrated with profiling information and feedback into the compiler and runtime to automate much of the optimization process.

CHAPTER 7

RELATED WORK

Code transformation and optimization for parallel programs have a long history, with much of the foundational work performed by the Parafrase [43, 44], PTRAN [45], and PFC [46–48] projects, followed later by Polaris [49] and SUIF [50]. This work builds on past work by examining a particular class of parallel architecture, namely single-chip, many-core architectures that enable fine-grained sharing of local execution resources and memories, and how to optimize applications for the architecture.

7.1 Parallel Programming Languages

Data-parallel programming languages are considered an intermediate approach between automatic parallelization and vectorization efforts [51, 52] and explicit parallel programming models such as OpenMP [53] to support parallel computing. APL was one of the first data-parallel languages and was developed from notation work by Iverson [54]. Fortran 90 [55] was one of the most widely used data-parallel languages and was notable for its use of array assignment statements. Later, High Performance Fortran (HPF) [56] was introduced as a standard data-parallel language to support programs with SPMD. However, the complexity of data distribution and communication optimization techniques, as discussed in the final two chapters of [34], was a difficult challenge. As a result, application developers became involved in explicitly handling data distribution and communication; message passing libraries such as MPI [57] became a popular programming model for scalable parallel systems. CUDA has similar management capability, where the developer explicitly manages data layout in DRAM and local memory spaces, data caching, thread communication within thread blocks, and other resources to achieve high performance.

The interest in general-purpose GPU programming has been driven by relatively recent improvements in the programmability of graphics hardware. The release of Cg [58] signified the recognition that GPUs were programmable processors and that a higher-level language was needed to develop applications for them. Others felt that the abstractions provided by Cg and other shading languages were insufficient and built higher-level language constructs. Brook [59] enables the usage of the GPU as a streaming coprocessor. Accelerator [60] is another system that uses data-parallel arrays to perform general-purpose computation on the GPU. A Microsoft C# library provides data types and functions to operate on data-parallel arrays. Data-parallel array computation is transparently compiled to shader programs by the Accelerator runtime.

Other efforts to provide a more productive stream processing programming environment for developing multithreaded applications include the RapidMind Streaming Execution Manager [61] and PeakStream Virtual Machine [62]. These mainly target high-performance computing applications that are amenable to stream processing. Their achieved performance may be inferior to customized GPU/CPU code due to virtual machine and dynamic compilation overhead as well as the inability to use platform-specific features. CUDA supports kernels with much larger code sizes and avoids the use of graphics APIs, although it currently does not map to other architectures as the RapidMind and PeakStream environments do. The hardware aspects of mapping general-purpose computation to GPUs is discussed later in the chapter.

A programming interface alternative to CUDA is available for the AMD Stream Processor, using the R580 GPU, in the form of the Close to Metal (CTM) compute runtime driver [63]. Instead of abstracting away architecture-level instructions, CTM completely exposes the ISA to the programmer for fine-grained control.

Intel's C for Heterogeneous Integration (CHI) programming environment [64] is a different approach to tightly integrate accelerators such as GPUs and general purpose CPU cores together, based on the proposed EXOCHI model. EXOCHI supports a shared virtual memory heterogeneous multithreaded programming model with minimal OS intrusion. In the CUDA execution model, the GPU is a device with a separate memory space from the CPU's. As a result, all data communication and synchronization between CPU and GPU is explicitly performed through the GPU device driver.

7.2 Optimizations and Performance Tuning

Chapter 3 discusses the effects of optimizations on the GeForce 8800. Many classical optimization techniques for data-parallel architectures are described in [34, 35, 65]. The effects of such optimizations, especially in combination, can be unexpected due to their effects on local resource usage, as discussed in [66, 67]. In particular, tiling at both the shared memory and register levels has major effects on performance and is the cause of significant discontinuities in the optimization spaces of some applications.

The memory subsystem of the GeForce 8800 favors access patterns which can be enabled by data layout transformations. The encompassing work on compiler-driven optimization for data locality was performed by Kennedy et al. [68, 69]. They split data locality optimization into three stages: loop optimizations, tiling, and register tiling. The fine-grained sharing of resources permits a variable number of threads and registers per thread to execute on this architecture, and this particular ordering of concerns has the potential of being trapped at a local performance maximum.

Later work in this area focused mainly on cache behavior of different instances of dynamically allocated objects on superscalar processors, whereas performance on the GPU involves structuring of accesses, usually to single arrays of structures, so that contiguous memory regions are simultaneously accessed. Yamada et al. [20] propose a combined hardware and software approach to improve memory performance. Truong et al. [21] use a library, driven by profile information, to reorganize fields of data or separate structures into different fields for better performance. Chilimbi et al. used automated techniques [22] for the same purpose.

The need for developers to understand the behavior of optimizing compilers for data-parallel architectures was discussed in work by Adve et al. [5]. They discuss the need to identify the modeling assumptions made about the system and a mechanism to validate those assumptions; optimization carving is a technique that does both of these for the GeForce 8 Series GPUs. NVIDIA provides tools, such as the `-ptx` and `-cubin` flags of nvcc, to give developers visibility into compiler behavior for assisting performance optimization. They have also recently enabled run-time profiling of applications on newer products. However, the GeForce 8800 is not sufficiently observable for application developers to easily find optimal configurations, and no support for compiler or runtime consumption of performance information is currently exposed to third-party developers.

7.3 Phase Ordering

Phase ordering concerns the order in which optimizations are applied and the decisions of whether to apply them or not. There is much previous work in phase ordering, particularly for classical optimizations directed towards single-thread applications. Vegdahl published one of the first papers on phase ordering [37], looking at the interaction of code generation and compaction. Whitfield and Soffa developed a framework to control phase ordering to improve performance [38]. Cooper et al. [70] use genetic algorithms to find good phase orderings. Kulkarni et al. mention other previous work [71].

This work is not directly related to phase ordering because the optimizations varied in the study are effectively orthogonal and applied unconditionally, making phase ordering a nonissue. Chapter 3 instead focuses on understanding the effects of optimizations on this class of architecture, while Chapter 5 shows how different optimizations interact. Prior work [39] has examined how optimal configurations differ from hand-optimized ones for several applications. This work contributes to future construction of effective phase orderings for this class of architecture.

If a compiler targets particular resource usages (thread blocks per SM, dimensions of thread blocks, etc.) instead of using optimization carving, phase ordering will become an important issue in finding the best optimization configuration that matches the desired resource usage. This issue is discussed briefly in Section 6.6. Studies such as that performed by Cooper et al. [72] or more recently Kulkarni et al. [71] will be necessary to find good phase orderings for particular program characteristics. The effects of optimizations may need to be predicted, similar to what Zhao et al.'s framework performs for embedded systems compilation [41].

7.4 Optimization Space Exploration

Program optimization carving as presented here is derived from a full exploration of the optimization space, an approach that has been explored by others in various fashions. Wolf et al. [73] introduced a compiler that explores the entire optimization space to find the optimal optimization configuration, but they do not use metrics to prune the space. Han et al. [74] also use static models to search for the optimal tiling and padding size for a conventional multiprocessor. Work has also been done to study the interaction among different optimizations and between optimizations and the hardware without a full search. These are based on analytical models [75, 76], statistical models [77], genetic algorithms [72], and adaptive learning and intelligent search techniques [78–81] to find an optimal configuration.

The optimization carving technique is most similar to the work of Wolf et al., but the performance metrics presented here are customized for a massively data-parallel architecture with a large memory bandwidth and latency-hiding memory system. To my knowledge, the only similar study of this emerging class of data-parallel architectures directed at broader computing domains is work by Jimenez-Gonzalez et al. [82]. They present an evaluation of communication bandwidth between different storage and computing components of the Cell Broadband Engine, and general guidelines in terms of optimizations, communication, data access patterns, and programming models for full utilization.

Iterative approaches to space exploration, such as the approach taken by the SPIRAL project [83], start at one or several basic configuration points and then apply optimizations in an attempt to find a good optimization configuration. In this work I do not take an iterative approach, since such an approach is easily trapped in a local maximum. Instead I examine the effects of optimizations on the GeForce 8800. Transformations tightly interact on the GeForce 8 Series GPUs and must be evaluated based on their joint effects to avoid being trapped at local maxima. Methods to direct iterative approaches to optimization will be important when particular resource usage targets are defined or when even a partial space exploration is prohibitively expensive.

7.5 GPU Application Mapping and Optimization

Owens et al. review previous work in mapping general purpose computation to GPUs in [84]. In general, previous GPU programming systems limit the size and complexity of GPU code due to their underlying graphics API-based implementations. CUDA supports kernels with much larger code sizes via a new hardware interface and instruction caching. The ability to write larger and more complex kernel codes gives rise to this work.

Previous GPU generations and their APIs also had restricted memory access patterns, usually allowing only sequential, contiguous writes to a linear array. This restriction is due primarily to limits in graphics APIs and corresponding limits in the GPU's specialized pixel and vertex processors. Accelerator [60] does not allow separate access to an individual element in parallel arrays: operations are performed on all array elements. Brook [59] also executes its kernel for every element in the stream, with some exceptions. The GeForce 8800 allows for general addressing of memory by each thread, which supports a much wider variety of algorithms. However, the increased generality also makes it important to apply locality enhancement optimizations to applications in order to conserve memory bandwidth and hide memory latency.

Liao et al. [85] developed a framework on top of Brook [59] to perform aggressive data and computation transformations. Their goal was to speed up GPU streaming applications on CPU multiprocessors. Breternitz et al. [86] also developed a compiler to generate efficient code on a CPU for SIMD graphic workloads by extending the base ISA to SSE2 [87]. These efforts differ from this work, which investigates the effects of optimizations specifically on a contemporary GPU architecture.

Previous attempts at general purpose programming on GPU systems have been limited in size and complexity. In particular, inflexibility of memory accesses [59, 60] and memory performance [88, 89] were major hurdles. A previous study on performance tuning for GPU [90] was also constrained by the programming environment and the necessity of mapping algorithms to existing GPU structures. The CUDA programming model, along with the hardware support of the GeForce 8800, allows larger, more complex kernel code to access the low-latency, high-bandwidth on-chip memory in a more general manner.

Traditional GPUs also provided limited cache bandwidth for nongraphics applications on the GPU. Fatahalian et al. [88] discuss how low-bandwidth cache designs on GPUs prevent general purpose applications from benefiting from the computational power available on these architectures. Work by Govindaraju et al. [89] uses an analytical cache performance prediction model for GPU-based algorithms. Their results indicate that memory optimization techniques designed for CPU-based algorithms may not be directly applicable to GPUs. With the introduction of reasonably sized, low-latency, high-bandwidth, on-chip memories in new generations of GPUs, this issue and its optimizations have become less critical. The efficient use of these on-chip memories still requires creative programmer effort.

One of the important optimizations for these processors is management of data layout. Automated efforts on data layout began with distributed memory machines [91]. Recent work has focused on removing the need for application developers to manually designate the use of specific memories in CUDA. Baskaran et al. [92] have developed a technique to automatically map global memory accesses to the shared scratchpad memory on the GeForce 8 Series. They are currently working on techniques to map data to the caches on the GPU.

CHAPTER 8

CLOSING REMARKS

Because of power constraints and performance bottlenecks, computer processors now consist of an increasing number of processing cores per silicon die. This situation requires an evaluation of the properties of applications that achieve good performance on such processors and the optimization techniques required to take advantage of the large number of execution resources. This situation is little different from previous eras of parallel computing research, but the newest data-parallel platforms are very inexpensive, opening up the area to nearly anyone that has interest in performing research on these platforms. It is also not clear how these systems should evolve, opening up possibilities for researchers to define fundamental computer architecture for many years to come. As a computer architect and compiler designer, I feel that now is an exciting time to be working in these areas.

8.1 Applicability to Future Many-Core Processors

I began this work with an examination of the GeForce 8800 GTX GPU and the CUDA programming model. Although highly specialized as a graphics architecture, the GeForce 8800 permits exploration of some of the fundamental concepts and issues that every single-chip, many-core processor will have in the future. The most notable of these are data management and use of local memories, the balance between multithreading and efficient use of resources, and the use of specialized function units and caches to improve performance beyond that of a minimal design.

I mentioned several times in this work that off-chip bandwidth is a first-order concern for most applications. This issue is unlikely to change in the future; the number of transistors per die and potential execution resources will increase at a faster rate than off-chip bandwidth. Management of data locality and reuse should be the initial focus of application and compiler developers for these systems. There are two issues here that must be addressed in the future:

- Due to software engineering concerns, programmers define data structures with multiple fields to logically group data. Many of these fields may not be used in a given kernel or application phase, and waste bandwidth when they are loaded from DRAM and sent through the memory system, either on the GeForce 8800 or a traditional processor with caches. In addition, the GeForce 8800 requires global memory coalescing when possible to achieve good performance, which, due to the SIMD nature of the SM, means that developers must either split up the structure (generally poor software engineering practice) or load entire structures in bulk into shared memory. This work motivates the need for additional research into reconciling software engineering needs with the desire for performance.

- Expressing data locality in code can be done easily with local variables, but many application developers are used to a flat memory space and often operate directly on variables in memory, relying on hardware caches to manage reuse and locality. Copying to local memory space in previous architecture generations generally resulted in poor performance because the loaded data would usually be in the processor's cache anyway. Future many-core processors are likely to have a scratchpad-style memory or lockable cache to ensure both locality and availability for some data. Whether the CUDA method of declaring some thread-specific local variables should be used or a compiler should attempt to automatically determine and map the appropriate variables to those memories is open to question.

The partitionable register file and shared memory on each SM of the GeForce 8800 enable an interesting tradeoff between the number of threads on each SM and the amount of resources available to each thread. Since the ideal balance will be different for every application, this arrangement enables tuning of application performance beyond what an architecture with a constant amount of resources per thread allows. It is not entirely clear what the chip area and design costs are for enabling this flexibility, but the performance benefit has shown that the idea has merit.

The specialized caches and the SFUs on the GPU provided significant performance benefit to the MRI kernels, but it is not entirely clear whether such features would be useful in a more general system. Value broadcast from the constant cache is enabled by the SIMD nature of the SM, ensuring that threads in a warp execute the same load instruction at the same time. How to preserve this advantage when threads can simultaneously take divergent control paths is unclear. On the other hand, given the abundance of transistors expected in future processors, we may see functionality similar to that of the SFUs' in future many-core processors that are intended for a wider variety of applications.

One of the largest optimization issues is the discontinuity in the optimization space due to the strict limits on local memories. Limited local memory is a fundamental trait of many-core processors, and it will be important to find hardware and software mechanisms to mitigate the discontinuities. It should be remembered that application development is an iterative process, with new features added in successive versions. These will often require additional memory, potentially exceeding available local memory and causing performance to drop precipitously compared to the previous version. Structures such as intermediate caches or other storage that lie between local memories and off-chip DRAM are conceptually simple, but it will be important to provide automated or mostly automated mechanisms to make their use simple for developers.

8.2 Thoughts on Optimization

The GeForce 8800 is highly threaded and enables fine-grained sharing of resources among threads. Thus, the effects of well-known optimizations on this architecture can be different from their effects on more traditional multicore systems. Without understanding the effects of optimizations, a useful, systematic optimization process is not possible. I discussed several categories of optimizations and their effects on this architecture. I also discussed how these optimizations can interact in unexpected ways, particularly how register usage can reduce the number of threads simultaneously executing on the system.

The change needed in developers' assumptions may be one of the largest obstacles in adoption of massively data-parallel, many-core systems. Until developers build up intuition of how applications map to this platform and how optimizations affect application performance, they will not be able to extract good performance from the system. The most prominent example is the relatively small effect of prefetching and the possibility of significant TLP loss due to increased register usage. One of the goals of this work is to develop that intuition and communicate it to others.

Manual, iterative optimization on this system may become trapped in a local performance maximum and underutilize the execution resources of the GPU. Rather than attempt an iterative optimization process, I instead proposed program optimization carving. This technique avoids becoming trapped in local performance maxima by examining the entire optimization space and pruning away configurations that are unlikely to be local maxima (and thus not the global maximum). This technique is capable of eliminating up to 98% of the space while still retaining the optimal configuration. I have shown that the technique can be far superior to random selection for some applications. I have also shown that less precise metrics provide similar functionality but are not effective at reducing the space. More precise metrics may be better at reducing the space but run the risk of being inaccurate and pruning the top-performing configurations.

I anticipate that some compiler developers will instinctively rebel against the idea of generating multiple code configurations and insist that the best code version can be created through careful tuning. Even for the author, it took some time to become accustomed to the idea, but on the GeForce 8800 there is little choice because of the lack of developer control of resource usage. More generally, the complexity of compilers and architecture, as well as the enormous range of applications, means that compilation systems will not have the perfect understanding of all the factors necessary to get the "best" optimization configuration via iterative optimization using greedy approaches. This issue might be of little concern except that on many-core systems the difference may be substantial (double-digit percentage or more), and some developers desire the additional performance.

Future work in this area includes techniques to further prune the search space or find near-best configurations more reliably. A combination of space search and iterative optimization may also be effective in finding near-optimal configurations of applications (although multiple carvings could be considered a iterative approach). Finally, developer tools can make the optimization task more automated and visible. Visualization tools in particular can help guide developers' thinking into the fundamental issues of bandwidth, efficiency, and utilization.

As the computing community continues to move forward with many-core processors, we should reexamine our assumptions about the systems and tools we develop as well as what application developers are willing to do or are capable of doing. This work has shown that assumptions about optimization effects may be less valid because of the nature of the target architecture. By questioning our assumptions, we are forced to revisit the fundamentals of performance and programmability. Such a mindset will be vital to making continued, rapid progress in the development of single-chip, many-core processors.

APPENDIX A

CALCULATION OF THE EXPECTED MAXIMUM VALUE OF A SAMPLE

This appendix describes the calculation of the expected maximum value of a randomly selected sample from a list of numbers. The calculation is used for comparison to the optimization carving technique demonstrated in Section 6.3. That section actually uses expected minimum values, which translate to expected maximum performance. I demonstrate the expected maximum value calculation here because I believe it to be more natural to most readers; the calculation of expected minimum value is a straightforward derivation. I first work through the calculation assuming replacement of selected values as a simple example, then demonstrate the calculation without replacement.

I start with a list of numbers S, sorted in ascending order. $|S|$ is the size of S. For simplicity I assume there are no repeated values in the list. S_i is defined as the ith-smallest element of S, where $1 \leq i \leq |S|$. A sample X is a collection of elements taken from S and is of size n. The calculation shown here is for the expected value of the maximum element of X.

The calculation of the expected maximum value of X is conceptually simple if broken down into several parts. The calculation is based around finding the probability of the maximum value of X being S_i for each i. The first stage is the calculation of the probability that all elements in X are less than or equal to S_i. From this, one can calculate the probability of the maximum value of X being exactly S_i. The calculation concludes with the computation of the expected value, which is the probability of S_i being the maximum of X, times S_i, for every i in S. In Section A.1 I allow X to contain repeated elements of S. For optimization carving, configurations are sampled no more than once. This is modeled in Section A.2 by not allowing X to contain repeated elements of S.

A.1 With Replacement

I first make the observation that the probability that an element chosen randomly from S has a value smaller than or equal to S_i is $(i/|S|)$. Assuming replacement of elements after being chosen, the probability of the values of all elements in X being smaller than or equal to S_i is $(i/|S|)^n$. This is shown in Equation (A.1).

$$\begin{aligned} P(\forall x \in X : x \le S_i) &= P((X_1 \le i) \ \wedge \ \ldots \ \wedge \ (X_n \le i)) \\ &= \prod_{k=1}^{n} P(X_k \le i) \\ &= \left(\frac{i}{|S|}\right)^n \end{aligned} \tag{A.1}$$

Next, I calculate the probability of the maximum of X being S_i, or more precisely, the maximum value of the elements within X being exactly S_i. The probability that the maximum of X is greater than S_i is one minus the probability that all values in the sample are less than or equal to S_i. This is the same as Equation (A.1), so the the calculation that the maximum is greater than S_i can be performed as shown in Equation (A.2).

$$\begin{aligned} P(\max[X] > S_i) &= 1 - \prod_{k=1}^{n} P(X_k \le i) \\ &= 1 - \left(\frac{i}{|S|}\right)^n \end{aligned} \tag{A.2}$$

The probability of the maximum of X being S_i is the same as the probability that the maximum is greater than S_{i-1} but no more than S_i. This calculation is shown in Equation (A.3).

$$\begin{aligned} P(\max[X] = S_i) &= P(\max[X] > S_{i-1}) - P(\max[X] > S_i) \\ &= \left[1 - \left(\frac{i-1}{|S|}\right)^n\right] - \left[1 - \left(\frac{i}{|S|}\right)^n\right] \\ &= \left(\frac{i}{|S|}\right)^n - \left(\frac{i-1}{|S|}\right)^n \end{aligned} \tag{A.3}$$

The expected value of the maximum value of the sample X can be calculated by summing the products of S_i and the probability that the maximum of the sample is S_i for all i in S. This calculation is shown in Equation (A.4).

$$\begin{aligned} E[\max[X]] &= \sum_{i=1}^{|S|} [P(\max[X] = i) \cdot S_i] \\ &= \sum_{i=1}^{|S|} \left[\left(\left(\frac{i}{|S|}\right)^n - \left(\frac{i-1}{|S|}\right)^n\right) S_i\right] \end{aligned} \tag{A.4}$$

A.2 Without Replacement

For optimization carving, choosing a sample is done without replacement of elements already chosen from the list. A sample consisting of the entire sample space should provide the element with the maximum value in the space, which is not the case with replacement. Without replacement, the size and the valid number of remaining choices decreases with an increase in the size of the sample. The probability of the values of all elements in X being smaller than or equal to S_i is shown in Equation (A.5). For this calculation, n must be less than i; otherwise, the maximum value of X will be equal to or greater than S_i and the probability of the maximum being equal to S_i is one or zero, respectively.

$$\begin{aligned}
P(\forall x \in X : x \leq S_i) &= P((X_1 \leq S_i) \ \wedge \ ... \ \wedge \ (X_n \leq S_i)) \\
&= \left(\frac{i}{|S|}\right)\left(\frac{i-1}{|S|-1}\right)...\left(\frac{i-n+1}{|S|-n+1}\right) \\
&= \left(\frac{i!}{(i-n)!}\frac{(|S|-n)!}{|S|!}\right)
\end{aligned} \tag{A.5}$$

The probability of the maximum of X being S_i is calculated similarly to that of Equations (A.2) and (A.3). The calculations are shown in Equations (A.6) and (A.7).

$$\begin{aligned}
P(\max[X] > S_i) &= 1 - P(\forall x \in X : x \leq S_i) \\
&= 1 - \left(\frac{i!}{(i-n)!}\frac{(|S|-n)!}{|S|!}\right)
\end{aligned} \tag{A.6}$$

$$\begin{aligned}
P(\max[X] = S_i) &= P(\max[X] > S_{i-1}) - P(\max[X] > S_i) \\
&= \left[1 - \left(\frac{(i-1)!}{(i-1-n)!}\frac{(|S|-n)!}{|S|!}\right)\right] - \left[1 - \left(\frac{i!}{(i-n)!}\frac{(|S|-n)!}{|S|!}\right)\right] \\
&= \left[\frac{i!}{(i-n)!} - \frac{(i-1)!}{(i-1-n)!}\right]\frac{(|S|-n)!}{|S|!} \\
&= \left[\frac{i! - (i-1)!(i-n)}{(i-n)!}\right]\frac{(|S|-n)!}{|S|!} \\
&= \left[\frac{i(i-1)! - (i-n)(i-1)!}{(i-n)!}\right]\frac{(|S|-n)!}{|S|!} \\
&= \frac{n(i-1)!}{(i-n)!}\frac{(|S|-n)!}{|S|!}
\end{aligned} \tag{A.7}$$

The computation for the expected value of the maximum of the sample is shown in Equation (A.8). The summation begins with $i = n$, since the maximum of X cannot be any smaller than S_n. A derivation of this equation for minimum expected value is used to calculate the expected maximum performance of samples of various sizes in Section 6.3.

$$
\begin{aligned}
E\left[\max\left[X\right]\right] &= \sum_{i=n}^{|S|} \left[P(\max\left[X\right] = i) \cdot S_i\right] \\
&= \sum_{i=n}^{|S|} \left[\left(\frac{n(i-1)!}{(i-n)!} \frac{(|S|-n)!}{|S|!}\right) S_i\right] \\
&= \frac{n(|S|-n)!}{|S|!} \sum_{i=n}^{|S|} \left[\frac{(i-1)!}{(i-n)!} S_i\right]
\end{aligned}
\tag{A.8}
$$

REFERENCES

[1] D. Pham et al., "The design and implementation of a first-generation CELL processor," in *IEEE International Solid-State Circuits Conference*, February 2005.

[2] AGEIA, "PhysX by AGEIA," March 2008. [Online]. Available: http://www.ageia.com.

[3] D. E. Shaw et al., "Anton, a special-purpose machine for molecular dynamics simulation," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, December 2007, pp. 1–12.

[4] J. W. Sias, S.-Z. Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W. W. Hwu, "Field-testing IMPACT EPIC research results in Itanium 2," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004, pp. 26–39.

[5] V. S. Adve, C. Koelbel, and J. M. Mellor-Crummey, "Compiler support for analysis and tuning data parallel programs," in *Proceedings of the 1994 Workshop on Parallel Processing Tools and Environments*, May 1994.

[6] NVIDIA, "NVIDIA CUDA," February 2008. [Online]. Available: http://www.nvidia.com/cuda.

[7] J. Nickolls and I. Buck, "NVIDIA CUDA software and GPU parallel computing architecture," presented at Microprocessor Forum, San Jose, CA, May 2007.

[8] M. J. Atallah, Ed., *Algorithms and Theory of Computation Handbook*. Boca Raton, FL: CRC Press, 1998.

[9] S. Woop, J. Schmittler, and P. Slusallek, "RPU: A programmable ray processing unit for realtime ray tracing," *ACM Transactions on Graphics*, vol. 24, no. 3, pp. 434–444, July 2005.

[10] D. Burger, J. R. Goodman, and A. Kagi, "Memory bandwidth limitations of future microprocessors," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996, pp. 78–89.

[11] C. Ding and K. Kennedy, "Bandwidth-based performance tuning and prediction," in *Proceedings of the IASTED International Conference on Parallel Computing and Distributed Systems*, November 1999.

[12] M. Wolfe, "Iteration space tiling for memory hierarchies," in *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, December 1987, pp. 357–361.

[13] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proceedings of the 1988 Conference on Principles of Programming Languages*, January 1988, pp. 319–329.

[14] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, June 1991, pp. 30–44.

[15] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 63–74.

[16] D. Callahan, S. Carr, and K. Kennedy, "Improving register allocation for subscripted variables," *Proceedings of the ACM SIGPLAN 1990 Conference on Program Language Design and Implementation*, pp. 53–65, June 1990.

[17] C. Ding and K. Kennedy, "The memory bandwidth bottleneck and its amelioration by a compiler," in *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, May 2000, pp. 181–190.

[18] C. Ding and K. Kennedy, "Improving effective bandwidth through compiler enhancement of global cache reuse," *Journal of Parallel and Distributed Computing*, vol. 64, no. 1, pp. 108–134, January 2004.

[19] J. Allen and K. Kennedy, "Automatic loop interchange," in *Proceedings of the 1984 ACM SIGPLAN Symposium on Compiler Construction*, June 1984, pp. 233–246.

[20] Y. Yamada, J. Gyllenhaal, G. Haab, and W. W. Hwu, "Data relocation and prefetching for large data sets," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, 1994, pp. 118–127.

[21] D. N. Truong, F. Bodin, and A. Seznec, "Improving cache behavior of dynamically allocated data structures," in *Proceedings of the Seventh International Conference on Parallel Architectures and Compilation Techniques*, October 1998, pp. 322–329.

[22] T. Chilimbi, B. Davidson, and J. Larus, "Cache-conscious structure definition," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, May 1999, pp. 13–24.

[23] J. W. Sias, "A systematic approach to delivering instruction-level parallelism in EPIC systems," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2005.

[24] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," in *Proceedings of the 31th Annual International Symposium on Computer Architecture*, June 2004, pp. 76–88.

[25] V. S. Pai and S. Adve, "Code transformations to improve memory parallelism," in *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, November 1999, pp. 147–155.

[26] S. Che, J. Meng, J. Sheaffer, and K. Skadron, "A performance study of general purpose applications on graphics processors," in *The First Workshop on General Purpose Processing on Graphics Processing Units*, October 2007.

[27] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," *Journal of Computational Chemistry*, vol. 28, no. 16, pp. 2618–2640, December 2007.

[28] IMPACT Research Group, "Parboil benchmark suite," March 2008. [Online]. Available: http://www.crhc.uiuc.edu/IMPACT/parboil.php.

[29] C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten, and W. W. Hwu, "GPU acceleration of cutoff pair potentials for molecular modeling applications," in *Proceedings of the ACM International Conference on Computing Frontiers*, to be published.

[30] S. S. Stone, J. P. Haldar, S. C. Tsao, W. W. Hwu, Z. Liang, and B. P. Sutton, "Accelerating advanced MRI reconstructions on GPUs," in *Proceedings of the ACM International Conference on Computing Frontiers*, to be published.

[31] S. Stone, H. Yi, J. Haldar, W. W. Hwu, B. Sutton, and Z. Liang, "How GPUs can improve the quality of magnetic resonance imaging," in *The First Workshop on General Purpose Processing on Graphics Processing Units*, October 2007.

[32] H.-M. Hang, Y.-M. Chou, and S.-C. Cheng, "Motion estimation for video coding standards," *Journal of VLSI Signal Processing*, vol. 17, pp. 113–136, 1997.

[33] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Reading, MA: Addison-Wesley, 2006.

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

[35] S. Muchnick, *Advanced Compiler Design and Implementation.* San Francisco, CA: Morgan Kaufmann Publishers, 1997.

[36] R. Gupta and R. Bodik, "Register pressure sensitive redundancy elimination," in *Proceedings of the 8th International Conference on Compiler Construction, LNCS 1575*, March 1999, pp. 107–121.

[37] S. R. Vegdahl, "Phase coupling and constant generation in an optimizing microcode compiler," in *Proceedings of the 15th Annual Workshop on Microprogramming*, October 1982, pp. 125–133.

[38] D. Whitfield and M. L. Soffa, "An approach to ordering optimizing transformations," in *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990, pp. 137–146.

[39] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, and W. W. Hwu, "Program optimization study on a 128-core GPU," in *The First Workshop on General Purpose Processing on Graphics Processing Units*, October 2007.

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

[41] M. Zhao, B. Childers, and M. L. Soffa, "Predicting the impact of optimizations for embedded systems," in *Proceedings of the 2003 Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2003, pp. 1–11.

[42] V. S. Adve, J. Mellor-Crummey, M. Anderson, J.-C. Wang, D. A. Reed, and K. Kennedy, "An integrated compilation and performance analysis environment for data parallel programs," in *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, November 1995, pp. 50–67.

[43] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The structure of an advanced vectorizer for pipelined processors," in *Proceedings of the 4th International Computer Software and Applications Conference*, October 1980, pp. 709–715.

[44] D. J. Kuck et al., "The effects of program restructuring, algorithm change, and architecture choice on program performance," in *Proceedings of the 13th International Conference on Parallel Processing*, August 1984, pp. 129–138.

[45] F. E. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An overview of the PTRAN analysis system for multiprocessing," in *Proceedings of the 1st International Conference on Supercomputing*, June 1987, pp. 194–211.

[46] J. R. Allen and K. Kennedy, "PFC: A program to convert Fortran to parallel form," in *Supercomputers: Design and Applications*, K. Hwang, Ed. Los Alamitos, CA: IEEE Computer Society Press, August 1984, pp. 186–203.

[47] R. Allen, D. Callahan, and K. Kennedy, "Automatic decomposition of scientific programs for parallel execution," in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 1987, pp. 63–76.

[48] J. R. Allen and K. Kennedy, "Automatic translation of Fortran programs to vector form," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 4, pp. 491–542, October 1987.

[49] W. Blume et al., "Polaris: The next generation in parallelizing compilers," University of Illinois at Urbana-Champaign, Tech. Rep. 1375, 1994. [Online]. Available: polaris.cs.uiuc.edu/publications/1375.pdf.

[50] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *IEEE Computer*, vol. 29, no. 12, pp. 84–89, 1996.

[51] R. Allen and K. Kennedy, "Automatic translation of Fortran programs to vector form," *ACM Transactions on Programming Langugages and Systems*, vol. 9, no. 4, pp. 491–542, 1987.

[52] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers*. Cambridge, MA: MIT Press, 1990.

[53] OpenMP Architecture Review Board Members, *OpenMP Application Program Interface*, OpenMP Architecture Review Board, May 2005.

[54] K. E. Iverson, *A Programming Language*. New York, NY: John Wiley and Sons, 1962.

[55] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener, *Fortran 90 Handbook: Complete ANSI/ISO Reference*. New York, NY: Intertext Publications, Inc.,/McGraw-Hill, Inc., 1992.

[56] D. B. Loveman, "High performance Fortran," *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 1, pp. 25–42, February 1993.

[57] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: The Complete Reference*. Cambridge, MA: MIT Press, 1995.

[58] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: a system for programming graphics hardware in a C-like language," in *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, 2003, pp. 896–907.

[59] I. Buck, "Brook specification v0.2," Stanford University, Tech. Rep. CSTR 2003-04, October 2003. [Online]. Available: http://merrimac.stanford.edu/brook/brookspec-v0.2.pdf.

[60] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using data parallelism to program GPUs for general-purpose uses," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006, pp. 325–335.

[61] M. D. McCool, K. Wadleigh, B. Henderson, and H.-Y. Lin, "Performance evaluation of GPUs using the RapidMind development platform," poster at the 2006 ACM/IEEE Conference on Supercomputing, Tampa, FL, 2006.

[62] PeakStream Staff, "The PeakStream platform: High productivity software development for multi-core processors," PeakStream Inc., Tech. Rep., 2006.

[63] AMD, "AMD stream computing," March 2008. [Online]. Available: http://ati.amd.com/products/streamprocessor/index.html.

[64] P. H. Wang et al., "EXOCHI: Architecture and programming environment for a heterogeneous multi-core multithreaded system," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2007, pp. 156–166.

[65] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*. Reading, MA: Addison-Wesley Publishing Company, 1991.

[66] V. Sarkar and R. Thekkath, "A general framework for iteration-reordering loop transformations," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, June 2007, pp. 175–187.

[67] C. Click and K. D. Cooper, "Combining analyses, combining optimizations," *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 2, pp. 181–196, March 1995.

[68] K. Kennedy and K. S. McKinley, "Optimizing for parallelism and data locality," in *Proceedings of the 6th International Conference on Supercomputing*, July 1992, pp. 323–334.

[69] S. Carr, K. S. McKinley, and C.-W. Tseng, "Compiler optimizations for improving data locality," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994, pp. 252–262.

[70] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Proceedings of the 1999 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, May 1999, pp. 1–9.

[71] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson, "Evaluation heuristic optimization phase order search algorithms," in *Proceedings of the 2007 International Symposium on Code Generation and Optimization*, March 2007, pp. 157–169.

[72] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "Exploring the structure of the space of compilation sequences using randomized search algorithms," *The Journal of Supercomputing*, vol. 36, no. 2, pp. 135–151, 2006.

[73] M. E. Wolf, D. E. Maydan, and D.-K. Chen, "Combining loop transformations considering caches and scheduling," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1996, pp. 274–286.

[74] H. Han, G. Rivera, and C.-W. Tseng, "Software support for improving locality in scientific codes," in *Proceedings of the 8th International Workshop on Compilers for Parallel Computers*, January 2000.

[75] S. Ghosh, M. Martonosi, and S. Malik, "Precise miss analysis for program transformations with caches of arbitrary associativity," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998, pp. 228–239.

[76] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation," in *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, October 2000, pp. 237–248.

[77] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff, "Automatic selection of compiler options using non-parametric inferential statistics," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, September 2005, pp. 123–132.

[78] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams, "Using machine learning to focus iterative optimization," in *Proceedings of the 4th Annual International Symposium on Code Generation and Optimization*, March 2006, pp. 295–305.

[79] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "Finding effective compilation sequences," *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 231–239, March 2004.

[80] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," in *Proceedings of the 2003 International Symposium on Code Generation and Optimization*, March 2003, pp. 204–215.

[81] K. Vaswani, M. J. Thazhuthaveetil, Y. N. Srikant, and P. J. Joseph, "Microarchitecture sensitive empirical models for compiler optimizations," in *Proceedings of the 2007 International Symposium on Code Generation and Optimization*, March 2007, pp. 131–143.

[82] D. Jimenez-Gonzalez, X. Martorell, and A. Ramirez, "Performance analysis of Cell Broadband Engine for high memory bandwidth applications," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, April 2007, pp. 210–219.

[83] M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, "SPIRAL: A generator for platform-adapted libraries of signal processing algorithms," *Journal of High Performance Computing and Applications (Special Issue on Automatic Performance Tuning)*, vol. 18, no. 1, pp. 21–45, 2004.

[84] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[85] S.-W. Liao, Z. Du, G. Wu, and G.-Y. Lueh, "Data and computation transformations for Brook streaming applications on multiprocessors," in *Proceedings of the 4th International Symposium on Code Generation and Optimization*, March 2006, pp. 196–207.

[86] M. Breternitz Jr., H. Hum, and S. Kumar, "Compilation, architectural support, and evaluation of SIMD graphics pipeline programs on a general-purpose CPU," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003, pp. 135–145.

[87] Intel Technical Staff, *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel, May 2007.

[88] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," in *Proceedings of the 2004 ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, August 2004, pp. 133–137.

[89] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A memory model for scientific algorithms on graphics processors," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, no. 89, August 2006, pp. 89–99.

[90] C. Jiang and M. Snir, "Automatic tuning matrix multiplication performance on graphics hardware," in *Proceedings of the 14th International Conference on Parallel Architecture and Compilation Techniques*, September 2005, pp. 185–196.

[91] K. Kennedy and U. Kremer, "Automatic data layout for distributed memory machines," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 4, pp. 869–916, July 1998.

[92] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2008, pp. 1–10.

AUTHOR'S BIOGRAPHY

Shane Ryoo was born on 18 July 1978 in Chicago, Illinois. He received a Bachelor of Science degree in Electrical Engineering with Highest Honors from the University of Illinois at Urbana-Champaign in May of 2000. He continued his studies at the University of Illinois, with a Master of Science in Electrical Engineering in May of 2004 and his Doctor of Philosophy in Electrical and Computer Engineering in May of 2008. Shane's primary research interests lie in program optimization and the architectural features and software analyses that enable high-performance execution.

Shane was a recipient of the National Defense Science and Engineering Graduate Fellowship, as well as a University of Illinois Fellowship, ECE Distinguished Fellowship, and Carver Fellowship. Shane served as a research assistant under Wen-mei W. Hwu for eight years. During his studies, Shane performed internships at Advanced Micro Devices in Austin, Texas, and Intel Corporation in Portland, Oregon, and Folsom, California.