

Program Optimization Study on a 128-Core GPU

Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, and Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
{sryoo, cirodrig, sssstone2, bsadeghi, ueng, hwu}@crhc.uiuc.edu

Abstract—The newest generations of graphics processing unit (GPU) architecture, such as the NVIDIA GeForce 8-series, feature new interfaces that improve programmability and generality over previous GPU generations. Using NVIDIA’s Compute Unified Device Architecture (CUDA), the GPU is presented to developers as a flexible parallel architecture. This flexibility introduces the opportunity to perform a wide variety of parallelization optimizations on applications, but it can be difficult to choose and control optimizations to give reliable performance benefit. This work presents a study that examines a broad space of optimization combinations performed on several applications ported to the GeForce 8800 GTX. By doing an exhaustive search of the optimization space, we find configurations that are up to 74% faster than those previously thought optimal. We explain the effects that optimizations can have on this architecture and how they differ from those on more traditional processors. For some optimizations, small changes in resource usage per thread can have very significant performance ramifications due to the thread assignment granularity of the platform and the lack of control over scheduling and allocation behavior of the runtime. We conclude with suggestions for better controlling resource usage and performance on this platform.

I. INTRODUCTION

As a result of continued demand for programmability, modern graphics processing units (GPUs) such as the NVIDIA GeForce 8-series are designed as programmable processors employing a large number of processor cores [1]. Programming interfaces such as NVIDIA’s CUDA allow developers to see the GPU as a massively data-parallel coprocessor and write code for it using ANSI C with minor restrictions. The NVIDIA GeForce 8800 GTX¹ consists of 16 streaming multiprocessors (SM), each with eight processing units, registers, and on-chip memory, has a peak attainable multiply-add performance of 388.8 single-precision GFLOPS, features 86.4 GB/s memory bandwidth, contains 768MB of main memory, incurs little cost in creating thousands of threads, and allows efficient data sharing and synchronization across subsets of threads [2].

An interesting aspect of this architecture is the flexibility in the assignment of local resources, such as registers or local memory, to threads. Each SM can run a variable number of threads, and certain local resources are divided among threads. The resulting performance in most cases is impressive, yielding speedups up to 351X that of a contemporary superscalar processor.

The flexibility allows better tuning of applications for performance, but raises two issues. First, the possible combination of optimizations and configurations is very large and makes the

optimization space tedious to explore by hand. However, certain features and optimizations provide multiple times speedup over a baseline implementation, so it is important to perform some exploration. For example, use of the constant cache for MRI kernels results in approximately 40X speedup over the baseline. Second, the overall effects that parallelization optimizations have on this architecture can be very different from more traditional architectures. Specifically, because the total amount of local resources and global memory bandwidth are constrained, the processor may need to reduce the number of coarse-grain thread assignment units in order to execute the code. This makes performance sensitive to even small changes in the code, making the effects of optimizations unpredictable. These two issues can make finding near-optimal configurations of the application difficult without searching the entire optimization space. For example, an exhaustive search of the entire optimization space of a small set of applications found multiple configurations that were up to 74% faster than versions previously thought to be optimal.

The purpose of this work is to determine the characteristics of optimizations for a variety of applications by exploring their entire optimization space. We then discuss some principles and techniques for finding near-optimal configurations of the applications.

We begin by introducing the execution hardware and threading model in Section II. Section III is a discussion of the space search process and the classifications and characteristics of the program optimizations. Section IV presents and discusses the results of the search for several applications. We discuss the variance in seemingly close configurations. Related work is discussed in Section V, and the paper concludes with some final statements and suggestions for future work.

II. ARCHITECTURE OVERVIEW

The GeForce 8800 has a large set of processor cores which are able to directly address a global memory. This allows for a more general and flexible programming model than previous generations of GPUs, and allows developers to easily implement data-parallel kernels. In this section we discuss NVIDIA’s Compute Unified Device Architecture (CUDA), with emphasis on the features that significantly affect performance. A more complete description can be found in [2], [3]. It should be noted that this architecture, although more disclosed than previous GPU architectures, still has details that have not been publicly revealed.

Before discussing the hardware, it is useful to describe the general programming and compilation process. The CUDA programming model is ANSI C supported by several keywords

¹The GeForce 8800 GTS, with fewer resources than the GTX, is also available. References of GeForce 8800 are implied to be the GTX model.

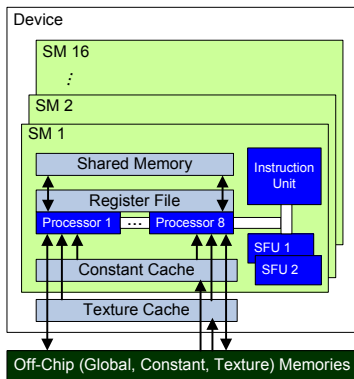


Fig. 1. Basic Organization of the GeForce 8800

and constructs. The GPU is treated as a coprocessor that executes data-parallel kernel functions. The user supplies a single source program encompassing both host (CPU) and kernel (GPU) code. These are separated and compiled by NVIDIA’s compiler. The host code transfers data to and from the GPU’s global memory via API calls. It initiates the kernel code by performing a function call.

A. Microarchitecture

Figure 1 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 1.35GHz. Each core executes a single thread’s instruction in SIMD (single-instruction, multiple-data) fashion, with the instruction unit broadcasting the current instruction to the cores. Each core has one 32-bit, single-precision floating-point, multiply-add arithmetic unit that can also perform 32-bit integer arithmetic. Additionally, each SM has two special functional units (SFUs), which execute more complex FP operations such as reciprocal square root, sine, and cosine with low latency. The arithmetic units and the SFUs are fully pipelined, yielding 388.8 GFLOPS ($16 \text{ SM} * 18 \text{ FLOP/SM} * 1.35\text{GHz}$) of peak theoretical performance for the GPU.

The GeForce 8800 has 86.4 GB/s of bandwidth to its off-chip, global memory. Nevertheless, with computational resources supporting nearly 400 GFLOPS of performance and each FP instruction operating on up to 8 bytes of data, applications can easily saturate that bandwidth. Therefore, as depicted in Figure 1 and described in Table I, the GeForce 8800 has several on-chip memories that can exploit data locality and data sharing to reduce an application’s demands for off-chip memory bandwidth. For example, each SM has a 16KB *shared memory* that is useful for data that is either written and reused or shared among threads. For read-only data that is accessed simultaneously by many threads, the constant and texture memories provide dramatic reduction in memory latency via caching. The constant cache is most beneficial when all threads in a *warp* (SIMD vector) access the same address in constant memory simultaneously, while the texture cache exploits 2D data locality.

Threads executing on the GeForce 8800 are organized

into a three-level hierarchy. At the highest level, each kernel creates a single *grid*, which consists of many *thread blocks*. The maximum number of threads per block is 512. Each thread block is assigned to a single SM for the duration of its execution. Threads in the same block can share data through the on-chip shared memory and can perform barrier synchronization by invoking the `__syncthreads` primitive. Threads are otherwise independent, and synchronization across thread blocks can only be safely accomplished by terminating the kernel. Finally, threads within a block are organized into warps of 32 threads. Each warp executes in SIMD fashion, issuing in four cycles on the eight SPs of an SM.

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 some other thread block assigned to the SM. The SM stalls only if there are no warps with ready operands available. Scheduling freedom is high in many applications because threads in different warps are independent with the exception of explicit synchronizations among threads in the same thread block.

B. 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 II. 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 the 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 within individual threads and the thread-level parallelism (TLP) among all threads.

For example, consider an application that uses 256 threads per block, 10 registers per thread, and 4KB 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 blocks per SM from 3 to 2. This decreases the number of threads on an SM by 33%. Although the register file can actually handle 744 threads with 11 registers apiece, the GeForce 8800 can only assign 2 thread blocks (512 threads) to an SM because a third block would increase the number of threads to 768 and register usage to 8,448, above the 8,192 registers per SM limit. By contrast, an optimization that increases each thread block’s shared memory usage by 1KB (an increase of 25%) does not decrease the number of blocks per SM. Clearly, the optimization space is inherently non-linear.

TABLE I
PROPERTIES OF GEFORCE 8800 MEMORIES

Memory	Location	Size	Latency	Read-Only	Description
Global	off-chip	768MB 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	16KB per SM	\approx 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 such that bank conflicts seldom or never occur.
Constant	on-chip cache	64KB total	\approx register latency	yes	8KB cache per SM, with data originally residing in global memory. The 64KB 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	16KB 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.

TABLE II
HARDWARE AND PROGRAMMING MODEL 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	8,192 registers
Shared Memory per SM	16,384 bytes
Threads per Thread Block	512 threads

III. OPTIMIZATION SPACE SEARCH

The basic strategy for achieving good kernel code performance on the GeForce 8800 is to reduce dynamic instruction count while maintaining high SP occupancy. High occupancy, where warps are always available for execution, is accomplished in three ways. First, one can have sequences of independent instructions within a warp so that the same warp can make forward progress. Second, a developer can place many threads in a thread block so that at least one warp can execute while others are stalled on long-latency operations, such as memory loads. Third, the hardware can assign up to eight independent thread blocks on an SM. Under high-occupancy conditions a reduction of executed instructions improves performance since each instruction occupies the same execution resources. This gives us four categories of machine-level behavior to optimize: thread-level work redistribution, instruction count reduction, intra-thread parallelism, and resource balancing.

As mentioned above, independent warps can come either from a few large thread blocks or from many small thread blocks. The two sources provide equal numbers of threads when shared memory is not the limiting factor on thread availability. Nonetheless, they do not produce equivalent performance for several reasons. Larger thread blocks enhance the benefit of data sharing in those kernels that take advantage of it. On the other hand, warps within a thread block must wait for one another during synchronization events. All else being equal, the number of schedulable threads can still vary with different block sizes because the hardware assigns integral numbers of thread blocks to an SM. A kernel with 256-thread thread blocks can run up to 768 threads per SM, while one

with 512-thread thread blocks can run only 512 threads per SM; two would exceed the 768-thread limit. The size of thread blocks for peak performance of an application depends on the application’s characteristics, and general performance may be insensitive to thread block size.

Unfortunately, optimizations rarely improve one 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, threads, and shared memory.

We will show examples of optimizations using a matrix multiplication kernel. The baseline kernel is shown in Figure 2(a). The code shown is run by each thread. Variables tx and ty are each thread’s coordinates in the thread block; variables $indexA$, $indexB$, and $indexC$ are initialized according to thread coordinates to positions in the two flattened input matrices and single output matrix prior to the shown code.

The first category of optimizations encompasses redistribution of work across threads and thread blocks. For matrix multiplication, each element of the output matrix can be computed independently and the work is grouped into threads and thread blocks for the sake of efficiency. The kernel of Figure 2(a) is tiled [4] so that each thread block computes a square 16-by-16 tile of the output matrix. Threads in a block cooperatively load parts of the input matrices into shared memory, amortizing the cost of global load latency and reducing the pressure on global memory bandwidth. Using larger tiles enhances the benefit of data sharing, but reduces scheduling flexibility since a greater fraction of the threads on an SM must wait at barrier synchronizations. Redistribution

```

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;

```

(a) Base Version

```

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;

```

(b) 1x2 Rectangular Tiling

```

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

```

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;

```

(d) Prefetching

Fig. 2. Matrix Multiplication Optimization Examples. Code differences from base version are shown in bold.

can also be applied at the thread level: in Figure 2(b), the kernel has been further tiled at the thread level so that each thread now computes two matrix elements instead of one. In other words, for every tile in the first input matrix, two tiles in the second input matrix are consumed at a time for a 1x2 rectangular tiling dimension. This presents opportunities for eliminating redundant instructions that were previously distributed across threads, such as the loads of values from *As* in the loop body. A third, occasionally useful technique is to distribute work across multiple invocations of a kernel. This can have secondary performance effects such as changing cache behavior, which we will show for one application in Section IV-B.

The second category of optimizations reduces the dynamically executed instruction count within a thread. Optimizations for this include traditional compiler optimizations such as common subexpression elimination, loop-invariant code removal, and loop unrolling. However, these optimizations need to be balanced against increased resource usage; e.g., common subexpression elimination tends to increase register usage due to extended live ranges of variables. The unrolled matrix multiplication kernel in Figure 2(c) eliminates address calculation instructions by replacing variable array indices with constants. Register usage is reduced by unrolling in this example, though it can also increase in other situations.

The third category of optimizations ensures the availability of independent instructions within a thread. A developer can unroll loops to facilitate code scheduling in the compiler or explicitly insert prefetching code. Prefetching for matrix multiplication (Figure 2(d)) is achieved by initiating long-latency global loads into an additional local variable (register) well before the variable is used. This optimization category is primarily the jurisdiction of the instruction schedulers of the compiler and runtime. The NVIDIA compiler appears to schedule operations to hide intra-thread stalls. However, it sometimes does this to the detriment of inter-thread parallelism. As with optimizations to reduce instruction count, scheduling to reduce intra-thread stalls may increase register usage to the point where the number of independent warps on

an SM is diminished.

The last category is best termed “resource-balancing.” The purpose of these optimizations is to trade certain resource usages, some of which may be counterintuitive, to produce a better performing application. An example of this is using shared memory to buffer data for reuse, regardless of whether it is shared with other threads. Another example is proactive register spilling by the programmer. By reducing register usage, often a critical resource, more thread blocks can be assigned to each SM. The resulting application may have much better performance despite the added latency from memory access and extra instructions. Though this category is included for completeness, the tradeoffs involved are not explored in this study.

The most common way in which these optimizations interact and interfere is by their effects on register use. For kernels where SM occupancy is limited by register pressure, including the three kernels we discuss in detail in the following section, register usage ties optimizations within a thread to optimizations that change the assignment of thread blocks to SMs. Unfortunately, register pressure effects are currently difficult to model and predict analytically.

One optimization applied beforehand as needed across all of the applications is the use of shared memory and caches to improve data locality for reused values: without this, performance was generally limited by global memory bandwidth and insensitive to other optimizations. We apply this optimization unconditionally in this work. We also did not constrain the optimizations performed by NVIDIA’s compiler and runtime.

IV. EXPERIMENTS

This section presents the results of the optimization space walks performed on the applications in Table III. They were selected from the larger set of CUDA applications we possess for the large number of optimizations and optimization parameters that had an effect on their code. Loop unrolling is one optimization common to all of the applications; the unrolling factor refers to the number of original loop bodies present in the code, so an unroll factor of 1 represents the original loop.

TABLE III
APPLICATION SUITE

Application	Description	Max Speedup over CPU
Matrix Multiplication	Multiplication of two dense 4k x 4k matrices. The CPU version uses version 9.0 of the Intel C++ Compiler and version 8.0 of the Intel Math Kernel Library.	6.98X
SAD	Computation of sums of absolute differences. SADs are computed between 4x4 pixel blocks in two QCIF-size images over a 32 pixel square search area.	19.6X
MRI-Q	Computation of a matrix Q , representing the scanner configuration, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.	351X
MRI-FHD	Computation of an image-specific matrix $F^H d$, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.	220X

We used CUDA version 0.8 for our experiments. GPU experiments were performed on a base AMD Opteron 248 system running at 2.2GHz with 1GB main memory. The performance of the CPU versions used as comparison in Table III was measured with single-thread code executing on an Intel Core2 Extreme Quad running at 2.66 GHz with 4GB main memory. We use this system because it gave the highest single-thread performance of the systems available to us. Applications were compiled with the `-O3` option, which controls optimizations on host code only. Binaries were compiled with SSE3 and fast math libraries when beneficial. Our presented data represent runs with smaller inputs than those considered typical, so that we can explore the entire optimization space in a reasonable amount of time. Separate experiments have shown that execution time will scale accordingly with an increase in input data size for these applications.

A. Matrix Multiplication

Figure 3 shows the results of our optimization space exploration when multiplying two 4k x 4k matrices. We varied tiling sizes, tiling dimensions, prefetching, and unroll factors, like those shown in Figures 2(b)-(d). The general trend is larger tiles sizes and more work per thread gives higher performance on average, due to better data sharing. Initially, the configuration of 1x1 tiling of 16x16 tiles with complete unrolling and prefetching was thought to be optimal with a performance of 87.5 GFLOPS. Upon traversing the optimization space, our peak performing configuration of 1x1 tiling of 16x16 tiles with complete unrolling and no prefetching delivered a performance of 91.3 GFLOPS, an improvement of 4.2%. We also tried 4x4 and 12x12 tile sizes, but these do not map as well to the platform and provided results significantly lower than that of 8x8 and 16x16 tile sizes, respectively. Square tiles larger than 16x16 were not possible because of the 512 threads per thread block limit.

Although computing more results per thread by increasing the tiling dimensions has more stable performance and a slight advantage in average performance, it does not result in peak performance. This is primarily due to the negative effects of unrolling by more than a factor of two for the higher tiling dimensions. Tests show that the CUDA runtime promotes loads in the loop body, increasing register lifetimes and register pressure, and reducing the number of thread blocks assigned per SM. One example is the 1x2 tiling dimension, completely unrolled and without prefetching configuration.

This configuration uses 23 registers per thread, allowing only a single thread block of 256 threads per SM. Given that the equivalent 1x1 tiling dimension configuration requires only nine registers, we believe that this configuration could have been executed using only 10 registers, which would allow three thread blocks per SM and provide approximately 120 GFLOPS performance, excluding stalls. The probable explanation is that the runtime’s scheduling and register allocation is intended to improve intra-thread scheduling without regard to its impact on the number of thread blocks running on the system. The performance of configurations with prefetching that are inferior to the equivalent non-prefetching configurations are also afflicted with register pressure issues. We believe that higher performance can be achieved by making an effort to increase the number of thread blocks per SM even at the expense of intra-thread schedulability.

In summary, larger thread blocks are beneficial due to the data sharing between threads. Complete unrolling often gives good performance by reducing the branches and address calculations. However, the runtime’s scheduling may increase register pressure such that the number of thread blocks assigned to each SM is reduced. Rectangular tiling is impacted by this effect, having lower performance even though the code is more efficient. Prefetching can have positive benefits as long as the additional register usage does not reduce the number of thread blocks assigned per SM.

B. Magnetic Resonance Imaging

Another application studied was a magnetic resonance imaging (MRI) reconstruction algorithm that builds on the work of [5], [6]. MRI scanners typically sample the spectrum emitted by the scanned object at spatial frequencies aligned on a Cartesian grid. Compared to Cartesian scans, non-Cartesian scan trajectories are often faster and more immune to noise and artifacts. However, reconstruction of non-Cartesian data requires delicate tradeoffs between image quality and computational complexity. The MRI reconstruction kernels studied here are designed to reconstruct high-quality images from non-Cartesian trajectories. The computation required to perform these reconstructions is substantial.

The MRI kernels studied compute two quantities, Q and $F^H d$, which a linear solver then uses to complete the reconstruction. The algorithms that compute Q and $F^H d$ are quite similar and thus have similar optimization space characteristics. We discuss $F^H d$ in detail here because it is

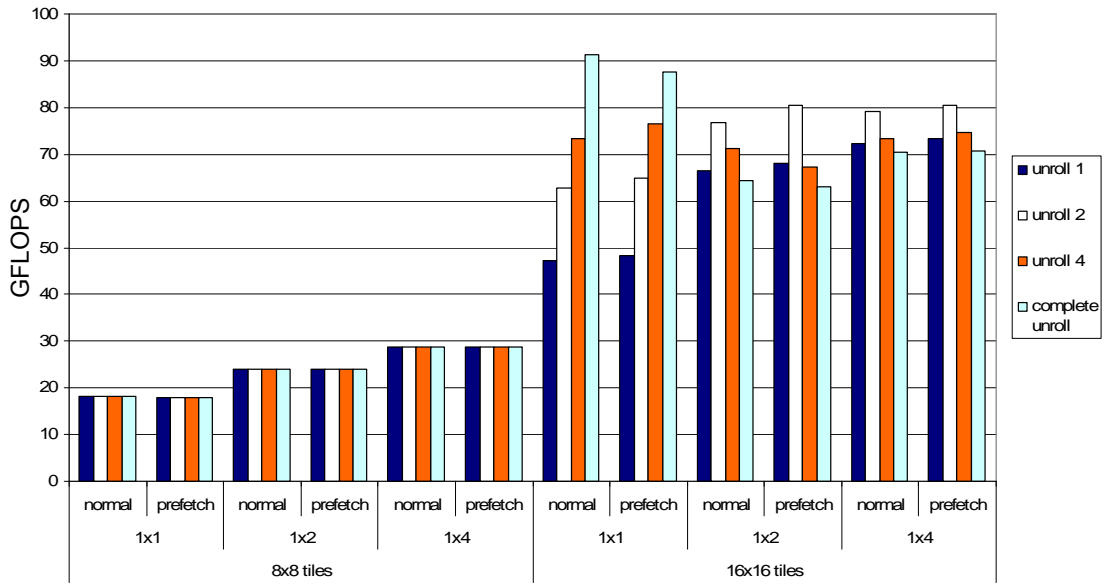


Fig. 3. Matrix Multiplication Results

computed for each reconstructed image and has more variance in performance between configurations. The performance of both kernels is sensitive to three parameters: the loop unrolling factor, the number of threads per block (tpb), and the number of scan points processed by each grid (ppg). For points per grid, the amount of data computed by each thread is varied but the number of threads stays the same, so the amount of data each kernel invocation processes scales accordingly.

Figure 4 shows the execution time, in seconds, of the MRI-FHD kernel as (a) the loop unrolling factor is varied from 1 to 16, (b) the number of threads per block is varied from 32 to 512, and (c) the number of scan points per grid is varied from 32 to 2048. In each case, parameters are varied by powers of 2. In (a) and (b), the configurations with higher points per grid take longer, while lower points per grid are represented by the lines clustered near the 5 second execution time. The best performance, at 4.56s, was obtained with an unrolling factor of 8, 64 tpb, and 64 ppg. Our original kernel had no unrolling, 512 tpb, and 512 ppg, with a time of 7.93s. We believed this configuration was optimal because it had fewer invocations of the kernel, thus having less invocation overhead, and maintained many threads on an SM. The improvement was a 43% reduction in execution time, or a 74% speedup.

The first obvious pattern in Figure 4(a) is shorter execution time for an unrolling factor of 8 compared to most other unrolling factors, while an unrolling factor of 4 is often worse than either 2 or 8. On traditional architectures, this behavior would be non-intuitive. However, it is easily explained by examining the usage of local resources and its result on thread scheduling. As an example, we examine the configuration with 256 tpb and 64 ppg. This configuration uses 12 registers when the loop is not unrolled, admitting 2 thread blocks per SM and executing in 6.17s. If the unrolling factor is 2, register usage stays the same and execution time drops to 5.52s. However, if the unrolling factor is 4, execution time increases to 5.89s.

Although the overhead per thread is reduced, register usage increases to 24 registers per thread at this unrolling factor. With 8192 registers each SM now admits a single thread block, reducing overall throughput. When the unrolling factor is increased to 8, the register usage per thread is 30, which still allows one thread block per SM. Execution time drops to 4.64s due to better instruction efficiency.

Varying the other optimization factors of MRI-FHD exposes issues that appear to be related to cache effects and data layout. Although Figure 4(b) appears to show that generally more threads per thread block yields better performance, the configurations which exhibit this behavior all have high points-per-grid values. As shown in Figure 4(c), which varies points per grid, higher points-per-grid values generally result in poorer performance. We originally believed that more work per grid would result in reduced kernel invocation overhead because fewer grids would need to be executed. However, it appears that more work per grid causes conflicts to occur in the constant cache due to the larger working set. Each thread block operates on its own set of scan points in the constant cache, so there is a smaller chance of conflicts when fewer thread blocks run on an SM. This is the probable reason for high points-per-grid configurations having better performance with more threads per thread block.

To summarize, performance of the MRI-FHD kernel is relatively insensitive to the size of thread blocks, except for high values of scan points per grid. An unrolling factor of 8 provided the highest performing configurations when other parameters are kept constant, but is not an obvious choice since the transition from a factor of 2 to 4 provided worse performance. This reduction in performance is due to an increase in register usage and a reduction in active threads. Fewer scan points per grid provides better performance due to cache effects, but is also a non-obvious choice because it increases the total kernel invocation overhead.

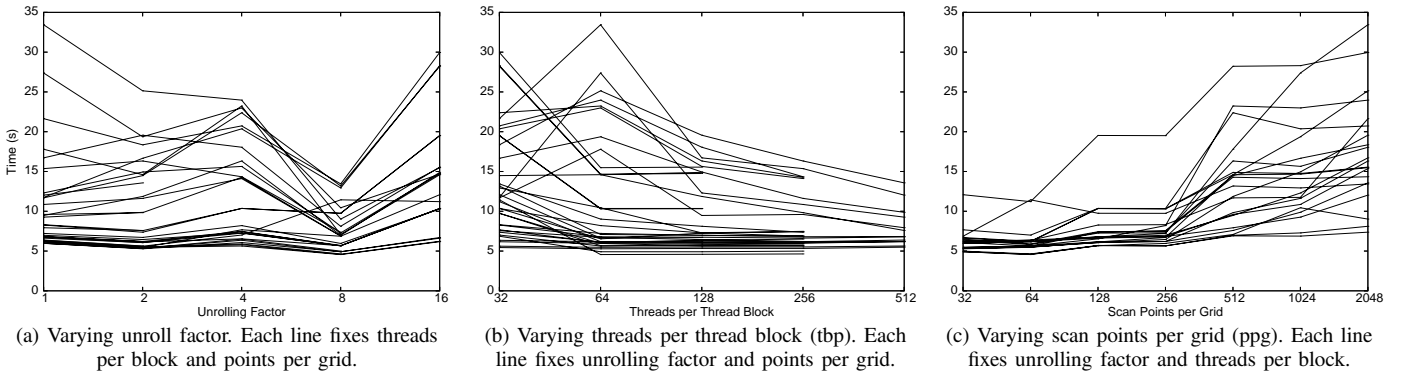


Fig. 4. Logarithmic Plots of MRI-FHD Space Search Results. Each graph varies one optimization parameter while keeping others constant.

C. Sums of Absolute Differences

Sums of absolute differences (SADs) are a metric used in some video encoders to estimate motion that has occurred between adjacent video frames. The SAD kernel we investigated has six parameters for optimization exploration. Three of the parameters are tiling factors. One tiling factor adjusts how coarsely one thread block’s work is distributed among threads by specifying the number of SAD values each thread computes. Two tiling factors control the size of the 2D image tile processed by a thread block. The other three parameters are unrolling factors for each of three nested loops within a thread. Unroll-and-jam [7] was performed for the outer loops. We considered unrolling factors up to 4 that evenly divided the loop’s trip count. Some combinations of parameters did not produce runnable kernels because they exceeded resource limits. For example, the register usage from maximum unrolling of all loops, combined with the number of threads needed for the minimum SADs-per-thread tiling, exceeded the limit of 8192 registers per thread block. We present results for the 963 working configurations produced out of 2016 possible configurations.

Unlike the previous benchmarks, many parameters did not have a consistent relationship to performance. In part, this resulted from competition for resources among different optimizations. Increasing an unrolling factor, for example, would typically speed up a low-register-usage configuration but slow down a high-register-usage configuration. More significantly, there were abrupt jumps in the performance curves that we were unable to fully characterize. Consequently we discuss our results in the context of the SADs-per-thread tiling parameter, for which the jumps are most visible against the background performance trend. All other parameters had a range of two to four values, making it hard to distinguish a jump from the background performance trend.

The highest-performing twenty configurations all had the same unrolling factor: the outer loop was unrolled by 3 and the inner loop was unrolled by 4. This combination exposed redundant texture fetches that could be eliminated to reduce the total number of executed instructions. Register pressure introduced by unrolling the middle loop negated the benefit from reduction in executed instructions. The eliminated instructions had a substantial effect on performance: every

configuration with these unrolling factors ran in less than 3.4ms, below the mean of 3.9ms.

In spite of the large number of long-latency texture fetches executed by this kernel, thread parallelism does not correlate well with performance. Figure 5(a) plots the amount of thread parallelism against run time for all configurations. To emphasize that performance is not a smooth function of kernel parameters, configurations that differ only in the SADs-per-thread tiling parameter are connected with lines. While there is a general trend of faster execution with more active threads, the vertical spread in performance is quite large and the fastest configuration has 244 threads, near the middle of the concurrency range.

The performance effect of tiling in the SAD kernel is not straightforward. The three tiling factors together determine the number of threads per thread block, which is related to the kernel’s performance as shown in Figure 5(b). Each line in the plot is generated by varying the SADs-per-thread tiling factor while keeping all other parameters constant, tracing the effect of this parameter on thread count and performance. Lines mostly slope down and to the right, indicating that distributing the same amount of work among more threads reduces run time. However, run time rises sharply for many configurations when the number of threads increases beyond 64 or 128, as well as for a few configurations at 192 or 256 threads. Not all configurations exhibit this step in run time. We were unable to explain why the step only appears for some configurations; the presence of a step coincides neither with particular values of any single parameter, nor with changes in the number of thread blocks that can run concurrently.

Due to the irregular shape of the performance curves, a local parameter search may fail to find the optimal performance. The slopes of most of the curves in Figure 5(b) suggest that performance improves with an increased number of threads per thread block. Indeed, we noticed this trend before performing the space search and manually selected a kernel with 246 threads per thread block. Yet, the optimum configuration from the space search has only 61 threads per thread block. This is the 244-thread optimum (four thread blocks per SM) in Figure 5(a). Relative to the manually selected kernel, it uses the same unrolling, a smaller SADs-per-thread tiling factor, and the smallest possible 2D image tile per thread block. It represents a 7.3% reduction in execution time over the

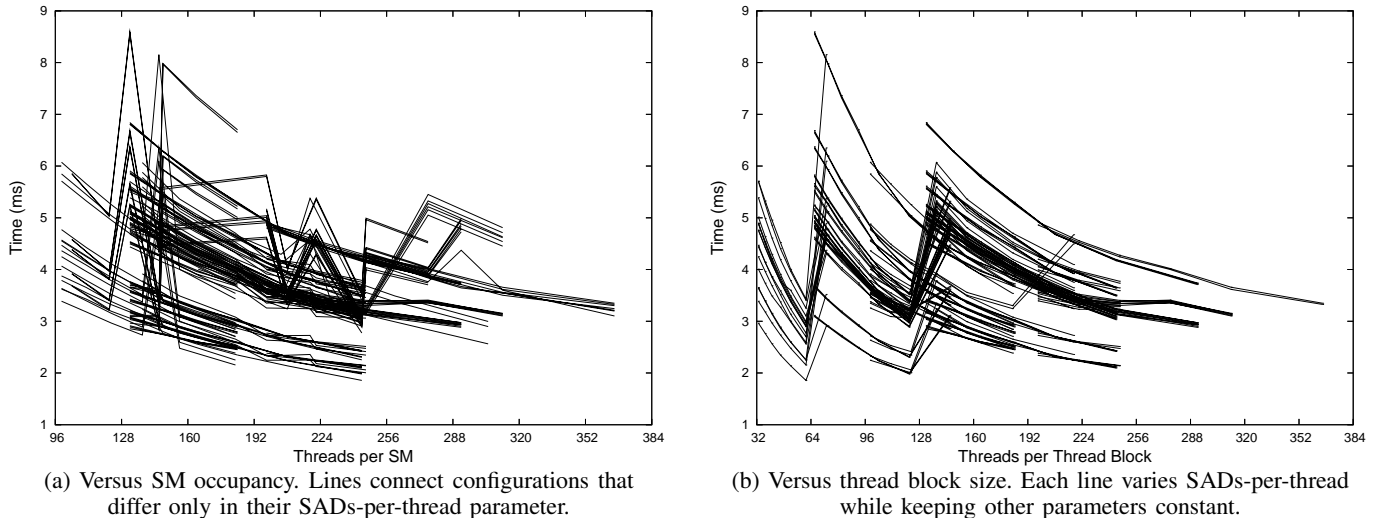


Fig. 5. Run Time of SAD Kernel

manually selected kernel.

In summary, the most important performance optimizations for this kernel reduce the number of executed long-latency instructions, using the large register file to gain efficiency. Significant additional performance comes from tiling the kernel with relatively few threads per block. Performance trends with respect to individual parameters are not smooth, making it impossible to empirically optimize each parameter separately; on the other hand, systematic performance trends are evident, so a smarter non-exhaustive search may be possible.

V. RELATED WORK

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 with a new hardware interface and instruction caching. The ability to write longer and more complex kernel codes gives rise to our work.

Previous GPU generations and their APIs also had restrictive memory access patterns, usually allowing only sequential writes to a linear array. This is due primarily to limits in graphics APIs and corresponding limits in the specialized pixel and vertex processors. Accelerator [8] does not allow access to an individual element in parallel arrays: operations are performed on all array elements. Brook [9] also executes its kernel for every element in the stream with restrictions. 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 many bandwidth and hide memory latency.

Traditional GPUs also provided limited cache bandwidth for GPGPU applications compared to graphics applications. Fatahalian et al. discuss in [10] that low-bandwidth cache designs on GPUs prevent general purpose applications from benefiting from the computational power available on these architectures.

Work discussed in [11] 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 such on-chip memories requires creative programmer effort.

In general new graphics architectures provide a variety of storage resources. To get high performance for data-parallel applications running on these architectures, it is crucial to efficiently allocate and manage these resources. This fact along with the demand for application domain extension for GPGPUs [12], [13], [14] complicates performance tuning and optimization for general purpose applications on GPUs.

Liao et al. [15] developed a framework on top of Brook [9] to perform aggressive data and computation transformations. Their modeling and optimizations goal is to speed up GPU streaming application on CPU multiprocessors. Breternitz et al. [16] also developed a compiler to generate efficient code on a CPU for SIMD graphic workloads by extending the base ISA to SSE2 [17]. These differ from our work, which investigates the effects of optimizations specifically on GPU architectures.

VI. CONCLUSION AND FUTURE WORK

We have described the non-traditional effects that classical optimizations can have on applications executing on the GeForce 8800 GTX architecture. Gradual changes in optimization parameters can have wildly varying effects on an application. This happens when the amount of local resources used by a thread increases to points where fewer thread blocks can be assigned to each SM, reducing overall performance. We identify several cases of this in our optimization space searches, particularly the effects of scheduling performed by the runtime. We believe that scheduling should be better controlled, possibly by the compiler rather than the runtime.

We are currently performing research on automated optimizations for this architecture. Our initial effort is focused on control of register usage so that the performance of applications after small code changes does not radically change. We are also looking at better models and heuristics that will allow us to achieve near-peak performance without examining the entire optimization space. Finally, we are investigating the creation of better tools and compilers that will allow programmers to specify the types of reorganizations desired and automatically experiment with their performance effects. This would greatly reduce developer optimization effort.

ACKNOWLEDGEMENT

We would like to thank David Kirk, John Nickolls, and Massimiliano Fatica at NVIDIA for their feedback and invaluable insights. The authors acknowledge the support of the Gigascale Systems Research Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. This work was performed with generous hardware loans from NVIDIA and donations from Intel.

REFERENCES

- [1] M. Pharr, Ed., *GPU Gems 2*. Addison-Wesley, 2005.
- [2] J. Nickolls and I. Buck, "NVIDIA CUDA software and GPU parallel computing architecture," Microprocessor Forum, May 2007.
- [3] "NVIDIA CUDA," <http://developer.nvidia.com/object/cuda.html>.
- [4] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proc. of the 4th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 63–74.
- [5] F. T. A. W. Wajer, "Non-Cartesian MRI scan time reduction through sparse sampling," Ph.D. dissertation, Technische Universiteit Delft, Delft, Netherlands, 2001.
- [6] J. A. Fessler and D. C. Noll, "Iterative reconstruction methods for non-Cartesian MRI," in *Proc. ISMRM Workshop on Non-Cartesian MRI*, 2007.
- [7] K. Kennedy and R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA: Morgan Kaufmann Publishers, 2002.
- [8] 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*, 2006, pp. 325–335.
- [9] I. Buck, *Brook Specification v0.2*, October 2003.
- [10] 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*, 2004, pp. 133–137.
- [11] 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, 2006.
- [12] P. Kipfer, M. Segal, and R. Westermann, "Overflow: a GPU-based particle engine," in *Proceedings of the 2004 ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2004, pp. 115–122.
- [13] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware," in *Proceedings of the 2003 ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2003, pp. 41–50.
- [14] F. Xu and K. Mueller, "Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware," *IEEE Transactions on Nuclear Science*, vol. 52, no. 3, pp. 654–663, June 2005.
- [15] 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.
- [16] J. Mauricio Breternitz, 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.
- [17] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel, May 2007.