Program Optimization Space Pruning for a Multithreaded GPU

Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi,
Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
{sryoo, cirodrig, ssstone2, bsadeghi, ueng, stratton, hwu}@crhc.uiuc.edu

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-parallel applications for these platforms, who lack the substantial experience and knowledge needed to maximize their performance. This creates a need for more structured optimization methods with means to estimate their performance effects. Furthermore these methods need to be understandable by most programmers. This paper shows the complexity involved in optimizing applications for one such system and one relatively simple methodology for reducing the workload involved in the optimization process.

This work is based on one such highly-parallel system, the GeForce 8800 GTX using CUDA. Its flexible allocation of resources to threads allows it to extract performance from a range of applications with varying resource requirements, but places new demands on developers who seek to maximize an application's performance. We show how optimizations interact with the architecture in complex ways, initially prompting an inspection of the entire configuration space to find the optimal configuration. Even for a seemingly simple application such as matrix multiplication, the optimal configuration can be unexpected. We then present metrics derived from static code that capture the first-order factors of performance. We demonstrate how these metrics can be used to prune many optimization configurations, down to those that lie on a Pareto-optimal curve. This reduces the optimization space by as much as 98% and still finds the optimal configuration for each of the studied applications.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

CGO'08, April 5–10, 2008, Boston, Massachusetts, USA.

General Terms

Performance, Languages

Keywords

GPGPU, parallel computing, optimization

1. INTRODUCTION

Programming 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 parallel systems such as the NVIDIA GeForce 8 Series are available for about a dollar per GFLOP, several orders of magnitude less expensive than supercomputers a decade ago. Already developers are using these desktop systems to perform work that would otherwise take a large compute cluster to accomplish. Unfortunately, the level of effort and expertise required to maximize application performance on these kinds of systems is still quite high. The resource restrictions of these systems also present unforeseen difficulties to optimization. Finally, it is often the case that successive generations of architectures require a complete reapplication of the optimization process to achieve the maximum performance for the new system.

Optimizing an application for maximum performance on the GeForce 8 Series is not a trivial task. At first glance, it appears to be a multi-variable optimization problem of applying a set of optimization techniques like tiling and loop unrolling to the code. However, the underlying hardware and threading model contain hard usage restrictions that affect performance and make the optimization space discontinuous. Consequently, the final performance of an optimization configuration is not always readily apparent. Furthermore, the difference in performance for differing optimization configurations is significant. For an MRI reconstruction application with a space size of 175 configurations, the difference in performance between a hand-optimized implementation and the optimal configuration was 17% and the difference in performance between the worst and optimal configurations was 235%. Since the intent is to run large, compute-intensive applications on these systems, a full exploration of the optimization space based on wall-clock performance is generally not feasible.

In response to these challenges, we have developed met-

rics to help prune the search space of optimization configurations. Our intent is to form these metrics into the underpinnings for an automated optimizing compiler for this platform. We developed our metrics after performing full explorations of the optimization spaces for some of our applications. The metrics take into account the observed first-order effects on performance, under the assumption that memory bandwidth is not the primary limiting factor on performance. Performance prediction is less critical for bandwidth-limited kernels since they are less sensitive to other optimization effects. The search space is pruned down to only those configurations that lie on a Pareto-optimal curve generated from a plot of the metrics. In contrast to a full exploration of the optimization space, this methodology eliminates the need to test as much as 98% of the optimization search space. The optimal configuration was found to be on the curve for each of the applications studied. Consequently, much faster performance optimization is possible.

We begin by discussing the execution hardware, threading model, and available tools in Section 2. This sets up the various factors that affect performance. Section 3 discusses the most effective optimizations for this architecture, and how one needs to consider them to attain the optimal configuration. Section 4 discusses the metrics we have developed given our experience while Section 5 shows how our metrics can be used to help find the optimal optimization configuration. We discuss related work in Section 6 before finishing with our conclusion.

2. ARCHITECTURE

This work uses the GeForce 8800 GTX GPU [1] as the basis for its study. The GeForce 8800 has a large set of processor cores that can 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 [1, 20]. 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 programming and compilation process. The CUDA programming model is ANSI C extended with several keywords 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.

2.1 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 SP 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.

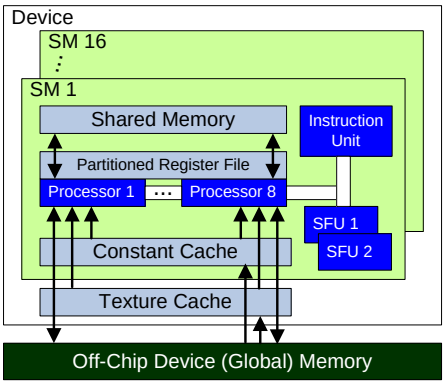


Figure 1: Organization of the GeForce 8800

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 12 bytes of source data, applications can easily saturate that bandwidth. Therefore, as described in Table 1 and depicted in Figure 1, 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.

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 thread 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 (single-instruction, multiple-data) 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.

2.2 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. Many optimizations that improve the performance of an individual

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

Table 1: 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	≃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	≃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 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	8,192 registers
Shared Memory per SM	16,384 bytes
Threads per Thread Block	512 threads

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 of 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 three to two, which decreases the number of threads on an SM by 33%. The GeForce 8800 can only assign two threads 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.

2.3 Software Tool 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`. Much of the information needed to compute the optimization metrics described in Section 4 is based upon the outputs of these flags. 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.

nvcc compiles kernel code to an assembly-like representation termed *PTX*. This is normally placed in an object file for consumption by the CUDA runtime, which processes this code, performs further optimization such as scheduling, and generates hardware-specific code for execution. The `-ptx` flag outputs the PTX in a developer-readable 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 an optimization is applied. In particular, information such as instruction count, instruction mix, and a rough idea of scheduling can be utilized reliably. Detailed instruction-level scheduling, however, is the domain of the runtime. For example, unrolling a loop with strided memory accesses creates successive operations that operate at different offsets from a base address. PTX shows that the group of memory operations only need the single base address calculation and use their constant offsets to avoid additional address calculations.

The CUDA runtime that generates executable machine code appears to reschedule code and allocate registers. This introduces an uncontrollable element during program optimization and makes the effects of optimizations on local resource usage less predictable. The `-cubin` flag outputs the resource usage of GPU kernel code, including the shared memory used per thread block and registers used per thread. This is critical to understanding the performance of the code because an SM runs the number of thread blocks that fit given their local resource usage. 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. We use the information provided by `-cubin` to calculate the number of thread blocks that can simultaneously reside on each SM.

3. OPTIMIZATION SPACE

The basic strategy for achieving good kernel code performance on the GeForce 8800 is to maintain high SP occupancy and reduce dynamic instruction count. 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 to an SM. Under high-occupancy conditions, a reduction of executed instructions improves performance by reducing the time to process each thread and thus increasing system throughput. This gives us five categories of machine-level behavior to optimize: independent thread count, thread-level work redistribution, instruction count reduction, intra-thread parallelism, and resource balancing.

Unfortunately, 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, threads, and shared memory. The most common way in which optimizations interact and interfere 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 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.

In this section we first discuss the major optimizations for performance. Using matrix multiplication as an example, we show how these optimizations can be applied to an application to find the optimal configuration.

3.1 Optimizations

The optimizations we consider can be grouped into five categories based on the primary mechanism by which they affect machine-level performance. The mechanisms are bolded for emphasis. We show examples of the optimizations using a matrix multiplication kernel, with baseline code shown in Figure 2(a). Information on CUDA syntax can be found in [21]. The code shown is executed by each thread. Variables `tx` and `ty` are initialized to 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 code shown.

The goal of the first category of optimization is to **provide enough warps to hide the stalling effects** of long latency and blocking operations. Loads from `A[indexA]` and `B[indexB]` are examples of long latency operations in matrix multiplication. Blocking operations include barrier synchronization, which stops a warp 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 can increase the number of thread blocks assigned to each SM and provide more independent warps from other blocks when one block reaches a barrier. This is discussed in more detail in Section 3.2.

The second category involves **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 data efficiency. The kernel of Figure 2(a) is tiled [19] so that each thread block computes a square 16x16 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 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, but we did not find this useful in applications with good cache behavior.

The third category is to **reduce the dynamic instruction count per thread**. Optimizations for this include traditional compiler optimizations such as common subexpression elimination, loop-invariant code removal, and loop unrolling. However, these optimizations frequently need to be balanced against increased resource usage. The unrolled matrix multiplication kernel in Figure 2(c) eliminates address calculation instructions by replacing variable array indices with constants after unrolling. Register usage is actually reduced in this example, though it can increase in other situations.

The fourth category of optimization, **intra-thread parallelism**, 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) long before the variable is used. This optimization category is primarily the jurisdiction of the instruction schedulers of the compiler and runtime. The CUDA runtime appears to reschedule 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 and potentially reduce the number of thread blocks on each SM.

The last category is best termed **resource-balancing**. The purpose of these optimizations is to shift the use of resources, some of which may be counterintuitive, to produce better overall performance. One example is proactive, explicit register spilling by the programmer. By reducing register usage, often a critical resource, more thread blocks may be assigned to each SM. The resulting application may have much better performance, despite the added latency from memory access and additional instructions, because the additional thread blocks improve overall resource utilization.

One optimization that was useful for all studied 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. For the experiments in this work, we apply this optimization unconditionally. We also do not constrain the optimizations or scheduling performed by NVIDIA's compiler and runtime.

```
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

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

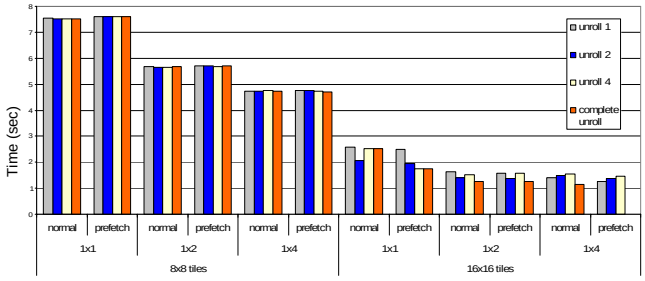
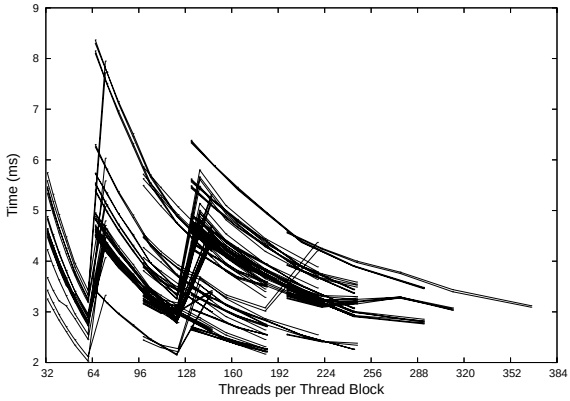


Figure 3: Matrix Multiplication Performance



Each line varies threads/block with other parameters constant.

Figure 4: SAD Optimization Space

3.2 Applying Optimizations

In this section we use matrix multiplication to show how to apply optimizations when searching for optimal performance. Our intent is to convey the complexity of the interaction between optimizations and resource limits. Figure 3 shows the run time of matrix multiplication across an abbreviated optimization space.

One of the first questions facing the developer is the granularity at which to spawn threads, since each SM can host up to 768 threads. Eight thread blocks at a tiling factor of 8x8 (64 threads per thread block) can be assigned to an SM, whereas only three thread blocks at a tiling factor of 16x16 (256 threads per thread block) can be assigned. Since matrix multiplication contains intra-thread block synchronization, it may be tempting to keep the number of thread blocks high. However, 16x16 consistently outperforms 8x8 because configurations with the latter tile size run into a memory bandwidth bottleneck.

When we consider the second category of optimizations, work redistribution, we see that allocating more work per thread by adjusting the tiling dimensions is generally good for matrix multiplication. This is true for both 8x8 and 16x16 tiles. It is interesting to note that for 1x4 tiling of 16x16 tiles, each SM only runs one thread block of 256 threads at a time due to heavy register usage, yet this configuration has the highest performance. Despite having only eight warps that must synchronize at a regular interval, the elimination of redundant instructions and enhanced instruction-level parallelism offset that downside.

Next, the performance effects of loop unrolling become more muddled. As shown in Figure 2(c), loop unrolling removes redundant instructions, reducing the instruction count while increasing register usage. When the loop is completely unrolled, the register usage sometimes drops back down to the same level as no unrolling. Since the mechanism by which the CUDA runtime performs scheduling and register allocation is not visible to the application developer, we do not have a clear explanation for this non-uniform behavior.

Finally, prefetching generally increases register usage but does not always reduce the number of thread blocks running on an SM. When it does, the reduction in exposed global latency often makes up for the loss of a thread block. Thus, there are only a handful of cases where there is a significant difference in performance. The exception is the configuration at the far right, where prefetching increased register usage beyond what is available, producing an invalid executable.

In summary, there is a significant number of optimization configurations to be considered for an application as simple as matrix multiplication. An expert with in-depth understanding of both the algorithm, including its behavior and usage patterns, and the hardware, including its memory bandwidth and resource availability, may have been able to bypass some of the pitfalls we present here. However, a developer that is not intimately familiar with the application, hardware, and the CUDA runtime *generally cannot determine if the upside of an optimization will more than compensate for potential downsides without experimentation.* As

stated earlier, the optimal configuration for matrix multiplication runs only one thread block of 256 threads per SM, contrary to the intuition of more concurrent threads equaling better performance.

In addition, the problem becomes much more difficult for larger and more complex applications. We present in Figure 4 a full exploration of the optimization space for a sum of absolute differences (SADs) kernel, which computes a metric used in MPEG video encoders. The number of possible configurations is much larger than matrix multiplication and the response of performance to optimizations even more complex.

4. PERFORMANCE METRICS

Given our observations of the effects of optimizations, we have developed metrics that estimate the performance of kernel code to the first order. Instead of being required to fully compile and run configurations of an application to determine its performance, a developer can use these metrics, which leverage data such as PTX instructions and resource usage extracted by the `-cubin` and `-ptx` flags, to quickly estimate the configurations' relative performances. This opens up the possibility of finding a near-optimal configuration without performing an exhaustive search of the optimization space.

In order for these metrics to correlate to performance, global memory bandwidth must not be the bottleneck on performance. This is easily calculated by examining the percentage of memory accesses in the instruction stream and determining the average number of bytes being transferred per cycle. Dealing with the memory bandwidth issue using software-managed local memories has been discussed in prior work [5, 22] and is outside the scope of this work.

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

Equation 1 estimates the *efficiency* of the kernel to be run on the GPU. *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. We manually annotate the average iteration counts of the major loops in the kernel to obtain this data. *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 is made explicit in the invocation of the kernel function. This efficiency metric indicates the overall efficiency of the configuration in terms of how many total instructions must execute before the kernel finishes. Assuming high SP occupancy and no bottlenecks, such as memory bandwidth, high efficiency is a very good indicator of better performance.

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

Equation 2 estimates the *utilization* of the compute resources on the GPU. *Regions* is the number of dynamic instruction intervals delimited by blocking instructions or the start or end of the kernel. We consider long latency instructions, such as global and texture memory operations and synchronization instructions, to be blocking instructions. Sequences of independent, long-latency loads are considered a unit. We consider SFU instructions to have long latency when longer latency operations are not present. 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. 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`.

The fraction $\frac{Instr}{Regions}$ indicates the average number of non-blocking instructions a single warp is expected to execute before running into its own blocking instruction. The quantity within the brackets indicates the number of independent warps in the SM, other than the one currently executing, that can be executed while the blocking 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. We divide 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. The second term in the bracket is the number of warps in other thread blocks on the SM that can execute. Altogether this is a metric of the utilization of the compute resources on the GPU by taking into account how often a warp is expected to wait and the amount of work available (from other warps) when it does.

We chose to group synchronization instructions together with long latency memory operations in order to simplify the calculation of the *Regions* term, even though they display different behaviors. For example, execution at a synchronization point proceeds only when all of the threads in a thread block have reached that point, while global load operations execute immediately and do not block execution until a use of the destination operand is encountered. We believe that the division by two in the first term in the bracket captures the first order effects. We are developing a more detailed cost model to achieve more precise results.

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

We use the matrix multiplication kernel of Figure 2(c) as an example. The kernel is first compiled with `-cubin` to obtain the resource usage, which shows that each thread uses 13 registers, and each block uses 2088 bytes of shared memory for its 256 threads. To determine the number of blocks per SM, we check the per-SM resource limits in Table 2. In this case, register usage is the limiting factor: $B_{SM} = \lfloor 8192/(13 * 256) \rfloor = 2$. Also, the number of warps per thread block is $W_{TB} = \lceil 256/32 \rceil = 8$.

This kernel is then compiled with `-ptx` to determine its execution profile. The outer 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 counted statically. A single thread runs 15150 instructions, including 512 barriers and 256 pairs of loads, so $Instr = 15150$ and $Regions = 769$. The last bit of information needed is the number of threads in the kernel. Based on knowledge of the program, we know there is one thread for each element of the 4k-by-4k output matrix: $Threads = 2^{24}$. From these numbers, we compute

Efficiency $= 3.93 * 10^{-12}$ and *Utilization* $= 227$. The relative values of these metrics among different configurations is more meaningful than their absolute values.

5. EXPERIMENTS

This section presents the values and use of the metrics for the applications in Table 3. The speedup over highly optimized, single-thread CPU performance shows why porting these applications to the GeForce 8800 is desirable. Table 4 lists the optimization parameters we chose to vary and the number of configurations in the search space. We show how the metrics can be used to find optimal or near-optimal configurations. We also discuss certain shortcomings of the metrics.

We used CUDA version 1.0 for our experiments. Experiments were performed on an Intel Core2 Extreme Quad running at 2.66 GHz with 4GB of main memory. Our presented data represent runs with smaller inputs than those considered typical, which allowed us to explore the entire optimization space in a reasonable amount of time and determine the proximity of our selected configurations to the optimal ones. Separate experiments have shown that execution time will scale accordingly with an increase in input data size for these applications on this architecture.

5.1 Individual Metrics

The efficiency and utilization metrics both carry part of the information needed to predict the performance of a kernel configuration, though neither is sufficient in isolation for useful performance comparisons. We use CP as an example to show what aspect of performance is captured by each metric. Figure 5 shows how CP's execution time and performance metrics vary with the results-per-thread tiling factor. We plot the normalized reciprocals of the performance metrics, so lower is better in both plots. The efficiency data points overlap and appear as a single curve. Efficiency closely follows the acutal execution time at tiling factors of 1, 2, 4, and 8. Although utilization varies over this range, it remains good enough that changes in utilization do not greatly slow down the machine's execution throughput. At a tiling factor of 16, utilization falls enough to bring down the machine's throughput, countering further increases in efficiency. Overall, efficiency improves monotonically while utilization worsens monotonically with increasing tiling factor, and the optimum configuration balances both metrics.

While this observational approach can be used to explain performance, an automated method of choosing good configurations needs to combine both metrics, taking into account their relative importance. We have found that the metrics are not detailed enough to combine into a single robust cost function that selects good configurations over a wide range of benchmarks. Instead, we use the metrics to narrow the space of possible configurations, as explained in the next section.

5.2 Searching by Pareto-Optimality

Figure 6 shows plots of the metric values for each optimization configuration for all of the applications. The maximum metric value along each axis has been normalized to one for comparison purposes. In general the best performance should come from configurations with both high efficiency and utilization. Thus, we desire configurations located towards the upper right corner of the graph.

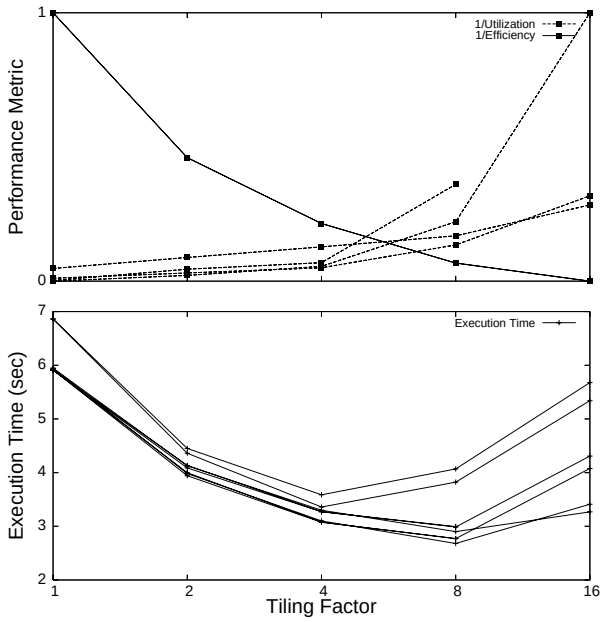


Figure 5: CP Metrics Versus Performance

In order to reduce the amount of time spent on performance evaluation, we choose the small set of configurations that have no superior in both the efficiency and utilization metric. This is the Pareto-optimal subset, which is connected by a line for each application. Visually, each point in this set has no other point both above and to the right of it. For all benchmarks, the Pareto-optimal subset contains the best configuration found by exhaustive search. In other words, instead of exhaustively evaluating the performance for every configuration, the search can be pruned down to just the configurations in the Pareto-optimal subset according to the metrics. This significantly reduces the search space and thus the search time, as shown in Table 4. The optimum configuration for each application in Figure 6 is circled for higher visibility. The relative values of efficiency and utilization are different for each benchmark, reflecting the difficulty of establishing a simple cost function to find the optimal configuration.

Figure 6(b) shows the metric plot for the MRI-FHD application. In this graph, configurations tend to be clustered in groups of seven because changing the tiling factor affects neither the efficiency nor the utilization of this benchmark, appearing as a single point at this resolution. Differences in actual performance within each cluster are small, the maximum within a cluster being 7.1%. In the cluster that contains the optimal configuration, the variation between the slowest configuration and the optimal configuration is 5.4%, and the variation between the median configuration and optimal configuration is only 0.2%. Hence, when several configurations have identical or nearly identical metrics, it may be sufficient to randomly select a single configuration from that cluster, rather than evaluating all the configurations.

5.3 Shortcomings of the Metrics

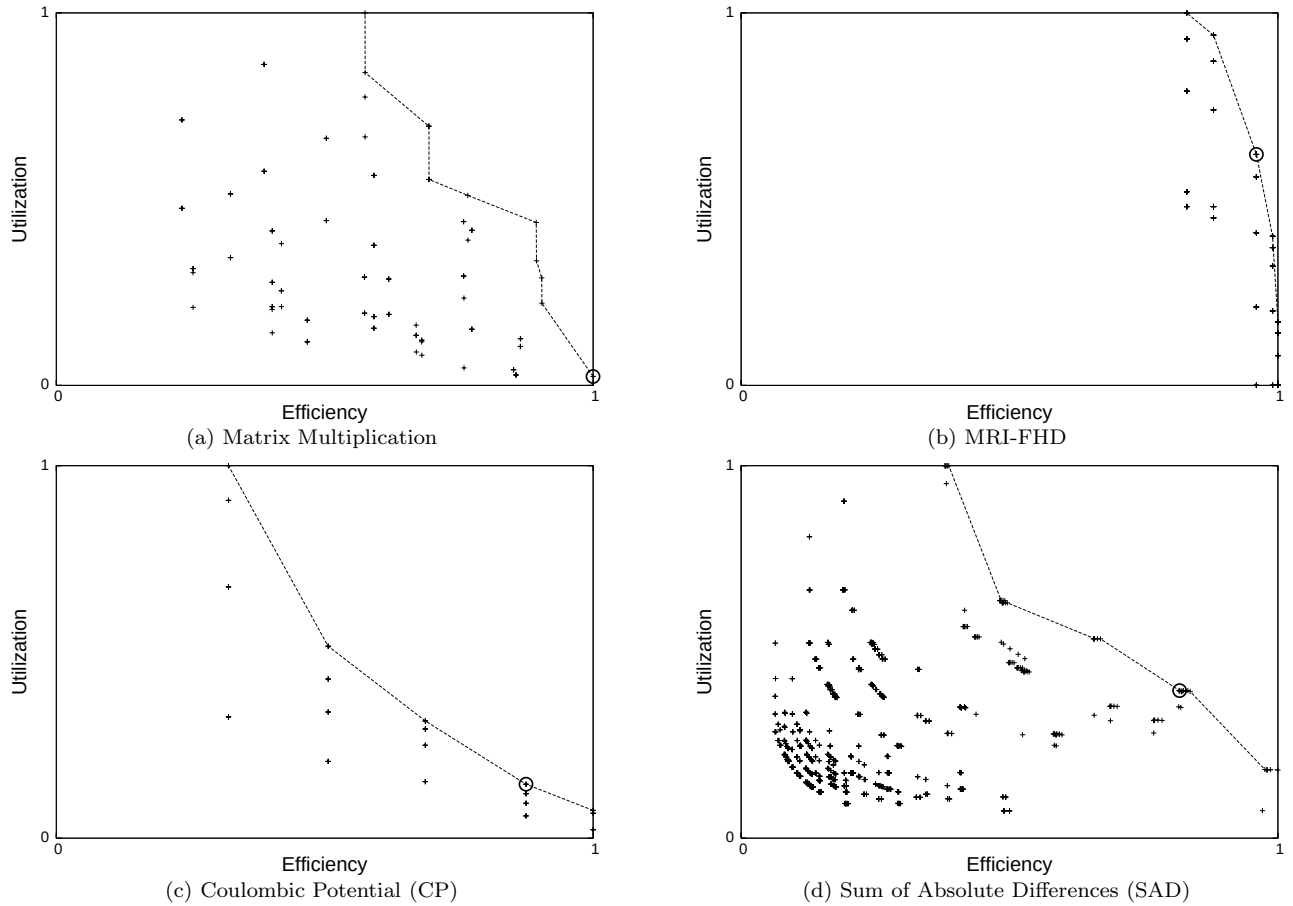
Although the Pareto-optimal subset of the metrics always contained the optimal optimization configuration for our test programs, the metrics do have certain shortcomings. Not all of the configurations in that subset are necessarily close to

Table 3: Application Suite

Application	Description	Speedup over Single-Thread 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
CP	Calculation of the electric potential at every point in a 3D grid. This kernel is derived from the "Unroll8y" kernel in [23].	647X
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.	5.51X
MRI-FHD	Computation of an image-specific matrix $F^H d$, used in a 3D magnetic resonance image reconstruction algorithm that operates on scan data acquired in a non-Cartesian space [24].	228X

Table 4: Parameter Search Properties

Kernel	Parameters Varied	Configurations	Evaluation Time	Selected Configurations	Space Reduction	Selected Evaluation Time
Matrix Multiplication	tile/block size, rectangular tile dimension, unroll factor, prefetching, register spilling	93	363.3 s	11	88%	48.6 s
CP	block size, per-thread tiling, coalescing of output	38	159.5 s	10	74%	42.95 s
SAD	per-thread tiling, unroll factor (3 loops), work per block	908	7.677 s	16	98%	0.127 s
MRI-FHD	block size, unroll factor, work per kernel invocation	175	771.9 s	30	77%	208.0 s



(a) Matrix Multiplication

(b) MRI-FHD

(c) Coulombic Potential (CP)

(d) Sum of Absolute Differences (SAD)

The optimal performance configuration is circled in each graph. In (b), each point actually represents as many as seven configurations that have indistinguishable efficiency and utilization.

Figure 6: Searching by Pareto-Optimal Performance Metric

optimal performance. In the cases we could identify, the reason was performance impact from factors assumed to be perfect and not considered in the metrics. We discuss a few of these issues here.

One interesting aspect of the Pareto-optimal curve for matrix multiplication, in Figure 6(a), is that all of the configurations on it except the optimum are 8x8 tile size configurations. As seen in Figure 3, none of the 8x8 configurations perform better than any of the 16x16 configurations due to memory bandwidth issues. This follows our statement in Section 4, which stated that memory bandwidth issues must be neutralized before efficiency and utilization become the dominant performance determinants. In general, the Pareto-optimal curve is more likely to miss a near-optimal configuration when a factor other than instruction count and latency overlap is a significant performance bottleneck. One should screen away such points prior to defining the curve.

Cache conflicts represent another potential performance bottleneck that falls outside the scope of the proposed performance metrics. However, discrepancies between the observed performance of a set of configurations and the performance trends predicted by the metrics can still help the programmer diagnose such bottlenecks. For example, a preliminary version of the MRI-FHD kernel had steadily decreasing performance as the tiling factor increased, although efficiency and utilization metrics remained constant. This informed the developer that other factors affecting performance, namely the layout of the data in the caches, was causing frequent misses. Changing the data layout yielded a kernel that is insensitive to changes in the tiling factor and 17% faster than the previous best configuration.

6. RELATED WORK

Code transformation and optimization for parallel programs have a long history, with much of the foundational work performed under the auspices of projects such as Polaris [6] and SUIF [11]. Many optimization techniques are detailed in [16, 29]. Our work builds on past work by determining when transformations are likely to provide higher performance on this new class of parallel architecture.

Our transformation guidance technique is based on a full exploration of the optimization space, an approach that has been explored by others in various fashions. Wolf et al. [28] 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. [12] 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 range from those based on analytical models [9, 17] to those that use statistical models [13] and those that utilize adaptive learning and intelligent search techniques [3, 4, 26, 27] to find an optimal configuration. Finally, work by the SPIRAL project [2] generally uses an iterative approach to find desirable code, whereas we do not. Our work is most similar to that of Wolf et al. [28], but our performance metrics are customized for a massively data parallel architecture with a high bandwidth and latency-hiding memory system. To our knowledge, the only similar study of this emerging family of data-parallel architectures being used for general purpose computing domains is work by Jimenez-Gonzalez et al. [15]. They present an evaluation of communication bandwidth between different storage and computing components of the Cell processor and general guidelines in terms of optimizations, communication, data access patterns, and programming models for full utilization.

Our work is related to previous work in phase ordering [18]. The effects of optimizations on the GPU are unlike those on manycore CPU, due to the high thread count and fine-grained sharing of resources. Transformations tightly interact on the GeForce 8 Series GPUs and must be evaluated based on their joint effects.

Previous attempts at general purpose programming on GPU systems have been limited in size and complexity. In particular inflexibility of memory accesses [7, 25] and memory performance [8, 10] were major hurdles. A previous study on performance tuning for GPU [14] was also constrained by the programming environment and the necessity of mapping algorithms to existing GPU features. 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. Choice of memory usage and optimization for this new generation of GPUs is critical to achieving good performance.

7. CONCLUSION AND FUTURE WORK

In this work we have proposed an approach for attacking the complexity of optimizing code for the NVIDIA GeForce 8 Series. Because predicting the performance effects of program optimizations is difficult, developers or compilers may need to experiment to find the configuration with the best performance. To aid in this, we developed metrics to judge the performance of an optimization configuration. By plotting the configurations and examining only those configurations on a Pareto-optimal curve, we were able to reduce the search space by up to 98% without missing the configuration with the highest performance. The cases where the Pareto-optimal curve may not contain a near-optimal configuration are attributable to factors that are not usually first-order performance determinants and thus not considered in the metrics.

Future work for this approach is directed towards better control of optimizations and improved pruning of the search space. First, we would like to achieve better control of scheduling and thus register usage, so that the performance of applications after small code changes does not radically change. This would make the effects of optimizations more predictable and potentially further reduce the search space. Second, we wish to account for factors such as memory access coalescing that are currently not factored into the performance metrics, so that they may be more effective predictors of performance. Finally, we will compare the effectiveness of our method to random sampling of the optimization space.

Acknowledgments

We would like to thank David Kirk and NVIDIA for generous hardware loans and support. We also thank Michael O'Boyle and the anonymous reviewers for their feedback and suggestions. Sam Stone is supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed in

this publication are those of the authors and do not necessarily reflect the views of the NSF. The authors acknowledge the support of the Gigascale Systems Research Center, funded under the Focus Center Research Program, a Semiconductor Research Corporation program. Experiments were made possible by NSF CNS grant 05-51665. This work was performed with equipment and software donations from Intel.

8. REFERENCES

[1] NVIDIA CUDA. http://www.nvidia.com/cuda.

[2] SPIRAL project. http://spiral.net.

[3] F. Agakov et al. Using machine learning to focus iterative optimization. In *Proceedings of the 4th Annual International Symposium on Code Generation and Optimization*, pages 295–305, March 2006.

[4] L. Almagor et al. Finding effective compilation sequences. *Proceedings of the 2004 ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 231–239.

[5] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1):6–26, November 2002.

[6] W. Blume et al. Polaris: The next generation in parallelizing compilers. Technical Report 1375, University of Illinois at Urbana-Champaign, 1994.

[7] I. Buck. *Brook Specification v0.2*, October 2003.

[8] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proceedings of the 2004 ACM Conference on Graphics Hardware*, pages 133–137.

[9] 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*, pages 228–239, 1998.

[10] N. K. Govindaraju et al. A memory model for scientific algorithms on graphics processors. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 89–99.

[11] M. W. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.

[12] H. Han, G. Rivera, and C.-W. Tseng. Software support for improving locality in scientific codes. In *8th Workshop on Compilers for Parallel Computers*, January 2000.

[13] 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*, pages 123–132, September 2005.

[14] 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*, pages 185–196, September 2005.

[15] 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*, pages 210–219, April 2007.

[16] K. Kennedy and R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, 2002.

[17] 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*, pages 237–248.

[18] P. A. Kulkarni et al. Evaluation heuristic optimization phase order search algorithms. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization*, pages 157–169, March 2007.

[19] 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*, pages 63–74, April 1991.

[20] J. Nickolls and I. Buck. NVIDIA CUDA software and GPU parallel computing architecture. Microprocessor Forum, May 2007.

[21] NVIDIA Corporation. *CUDA Programming Guide*, February 2007.

[22] S. Ryoo et al. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2008.

[23] J. E. Stone et al. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28(16):2618–2640, December 2007.

[24] S. Stone et al. How GPUs can improve the quality of magnetic resonance imaging. The First Workshop on General Purpose Processing on Graphics Processing Units, October 2007.

[25] 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*, pages 325–335, October 2006.

[26] S. Triantafyllis et al. Compiler optimization-space exploration. In *Proceedings of the 2003 International Symposium on Code Generation and Optimization*, pages 204–215.

[27] K. Vaswani et al. Microarchitecture sensitive empirical models for compiler optimizations. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization*, pages 131–143.

[28] 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*, pages 274–286, December 1996.

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