Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA

Shane Ryoo† Christopher I. Rodrigues† Sara S. Baghsorkhi† Sam S. Stone†
David B. Kirk∗ Wen-mei W. Hwu†

†Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign
∗NVIDIA Corporation

{sryoo, cirodrig, bsadeghi, ssstone2, hwu} @crhc.uiuc.edu, dk@nvidia.com

Abstract

GPUs have recently attracted the attention of many application developers as commodity data-parallel coprocessors. The newest generations of GPU architecture provide easier programmability and increased generality while maintaining the tremendous memory bandwidth and computational power of traditional GPUs. This opportunity should redirect efforts in GPGPU research from ad hoc porting of applications to establishing principles and strategies that allow efficient mapping of computation to graphics hardware. In this work we discuss the GeForce 8800 GTX processor's organization, features, and generalized optimization strategies. Key to performance on this platform is using massive multithreading to utilize the large number of cores and hide global memory latency. To achieve this, developers face the challenge of striking the right balance between each thread's resource usage and the number of simultaneously active threads. The resources to manage include the number of registers and the amount of on-chip memory used per thread, number of threads per multiprocessor, and global memory bandwidth. We also obtain increased performance by reordering accesses to off-chip memory to combine requests to the same or contiguous memory locations and apply classical optimizations to reduce the number of executed operations. We apply these strategies across a variety of applications and domains and achieve between a 10.5X to 457X speedup in kernel codes and between 1.16X to 431X total application speedup.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming

General Terms Design, Performance, Languages

Keywords parallel computing, GPU computing

1. 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 [20]. With the addition of new hardware interfaces, programming them does not require specialized programming languages or execution through graphics application programming interfaces (APIs), as with previous GPU generations. This makes an inexpensive, highly parallel system available to a broader community of application developers.

The NVIDIA CUDA programming model [3] was created for developing applications for this platform. In this model, the system consists of a *host* that is a traditional CPU and one or more compute *devices* that are massively data-parallel coprocessors. Each CUDA device processor supports the Single-Program Multiple-Data (SPMD) model [8], widely available in parallel processing systems, where all concurrent threads are based on the same code, although they may not follow exactly the same path of execution. All threads share the same global address space.

CUDA programming is done with standard ANSI C extended with keywords that designate data-parallel functions, called *kernels*, and their associated data structures to the compute devices. These kernels describe the work of a single thread and typically are invoked on thousands of threads. These threads can, within developer-defined bundles termed *thread blocks*, share their data and synchronize their actions through built-in primitives. The CUDA runtime also provides library functions for device memory management and data transfers between the host and the compute devices. One can view CUDA as a programming environment that enables software developers to isolate program components that are rich in data parallelism for execution on a coprocessor specialized for exploiting massive data parallelism. An overview of the CUDA programming model can be found in [5].

The first version of CUDA programming tools and runtime for the NVIDIA GeForce 8 Series GPUs has been available through beta testing since February 2007. To CUDA, the GeForce 8800 GTX[1] consists of 16 *streaming multiprocessors* (SMs), each with eight processing units, 8096 registers, and 16KB of on-chip memory. It has a peak attainable multiply-add performance of 345.6 single-precision GFLOPS[2], features 86.4 GB/s memory bandwidth, contains 768MB of main memory, and incurs little cost in creating thousands of threads. The architecture allows efficient data sharing and synchronization among threads in the same thread block [18].

A unique aspect of this architecture relative to other parallel platforms 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 the local resources are divided among threads as specified by the programmer. This flexibility

PPoPP '08, February 20–23, 2008, Salt Lake City, Utah, USA

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

[2] Particular mixes of instructions can achieve higher throughput, as will be explained in Section 3.

allows more tuning of application performance but changes the assumptions developers can make when performing optimizations. We discuss these issues in further detail.

Another question we address is how well applications can execute on the GeForce 8800 and what are the design features that contribute to or limit performance. As a collaborative effort between industry and academia, a set of complete numerical applications was ported and evaluated on the CUDA platform. Several application research groups in the areas of medical imaging, molecular dynamics, computational chemistry, electromagnetic analysis, and scientific visualization contributed to this effort. The following are the major principles when choosing code to be executed on this platform:

1. *Leverage zero-overhead thread scheduling to hide memory latency.* On the GeForce 8800 there are 128 execution units available for use, requiring hundreds of threads to completely occupy them. In addition, threads can be starved of data due to the long latency to global memory. The general philosophy of CUDA for tolerating this latency is to generate and maintain thousands of threads in flight. This is in contrast with the use of large caches to hide memory latencies in CPU designs. Developers used to traditional multicore systems may need to define threads at a finer granularity in order to generate enough threads. In addition, a high compute-to-memory-access ratio is necessary to avoid saturation of memory channels.
2. *Optimize use of on-chip memory to reduce bandwidth usage and redundant execution.* Working memory within a group of cores consists primarily of a register file and a software-managed on-chip memory called *shared memory*. These are high fan-out, low latency, limited-capacity memories which are partitioned among thread blocks that are assigned to the same SM at runtime. The data in shared memory can be shared among threads in a thread block, enabling interthread data reuse. An incremental increase in the usage of registers or shared memory per thread can result in a substantial decrease in the number of threads that can be simultaneously executed.
3. *Group threads to avoid SIMD penalties and memory port/bank conflicts.* CUDA is based on the SPMD model, but its current implementation on the GeForce 8800 imposes Single-Instruction, Multiple-Data (SIMD) mode among subsets of threads. The latter differs from the short-vector SIMD present in most contemporary processors. This is a cost-effective hardware model for exploiting data parallelism and allows the GeForce 8800 to share one instruction issue unit among eight execution units. However, it can be ineffective for algorithms that require diverging control flow decisions in data-parallel sections. In some algorithms, threads can be reorganized to avoid divergent control flow. Appropriate thread grouping can also preserve performance by avoiding port and bank conflicts in memories.
4. *Threads within a thread block can communicate via synchronization, but there is no built-in global communication mechanism for all threads.* This avoids the need for virtualization of hardware resources, enables the execution of the same CUDA program across processor family members with a varying number of cores, and makes the hardware scalable. However, it also limits the kinds of parallelism that can be utilized within a single kernel call.

We first discuss related work in Section 2. Section 3 introduces the threading model and execution hardware. Section 4 demonstrates the optimization process with in-depth performance analysis, using matrix multiplication kernels. Section 5 presents several studied applications with performance and optimization information. We conclude with some final statements and suggestions for future work.

2. Related Work

Data parallel programming languages are considered an intermediate approach between automatic parallelization efforts [7, 28] and explicit parallel programming models such as OpenMP [19] to support parallel computing. Fortran 90 [6] was the first widely used language and influenced following data parallel languages by introducing array assignment statements. Similar to array assignments in Fortran 90 is the lock step execution of each single instruction in threads executing simultaneously on a streaming multiprocessor in CUDA programming model. Later, High Performance Fortran (HPF) [15] was introduced as an standard data parallel language to support programs with SPMD. However, complexity of data distribution and communication optimization techniques, as discussed in the final two chapters of [13], were a hard-to-solve challenge. As a result application developers became more involved in explicitly handling data distribution and communication; message passing libraries such as [23] became a popular programming model for scalable parallel systems. Similarly in CUDA, the developer explicitly manages data layout in DRAM memory spaces, data caching, thread communication within thread blocks and other resources.

The interest in GPGPU programming has been driven by relatively recent improvements in the programmability of graphics hardware. The release of Cg [16] signified the recognition that GPUs were programmable processors and that a higher-level language was needed to develop applications on them. Others felt that the abstractions provided by Cg and other shading languages were insufficient and built higher-level language constructs. Brook [9] enables the usage of the GPU as a streaming coprocessor. Accelerator [26] is another system that uses data-parallel arrays to perform general-purpose computation on the GPU. A Microsoft C# library provides data types and functions to operate on data-parallel arrays. Data-parallel array computation is transparently compiled to shader programs by the runtime. Other efforts to provide a more productive stream processing programming environment for developing multi-threaded applications include the RapidMind Streaming Execution Manager [17] and PeakStream Virtual Machine [4]. These mainly target HPC applications that are amenable to stream processing. The achieved performance may be behind customized GPU/CPU code due to the virtual machine and dynamic compilation overhead. We refer the reader to a review of the main body of work done to map general purpose computation to GPUs by Owens et al. in [21].

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.

Previous GPU generations and their APIs also restricted the allowed 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 does not allow access to an individual element in parallel arrays: operations are performed on all array elements. Brook also executes its kernel for every element in the stream, with some exceptions. The GeForce 8800 allows for general addressing of memory via a unified processor model, which enables CUDA to perform unrestricted scatter-gather operations.

Traditional GPUs also provided limited cache bandwidth. Fatahalian et al. discuss in [11] that low bandwidth cache designs on GPUs limit the types of applications from benefiting from the computational power available on these architectures. Work discussed in [12] uses an analytical cache performance prediction model for GPU-based algorithms. Their results indicate that memory opti-

mization techniques designed for CPU-based algorithms may not be directly applicable to GPUs. With the introduction of reasonably sized low-latency, on-chip memory in new generations of GPUs, this issue and its optimizations have become less critical.

A programming interface alternative to CUDA is available for the AMD Stream Processor, using the R580 GPU, in the form of the Close to Metal (CTM) compute runtime driver [1]. Like CUDA, CTM can maintain the usage of the GPU as a graphics engine; however, instead of abstracting away architecture-level instructions, CTM completely exposes the ISA to the programmer for fine-grained control. Furthermore, the R580 continues to resemble previous generation GPUs with their divided architecture for vertex and pixel processing, whereas the GeForce 8800 has a more general, unified model. This is presented in the next section.

Intel's C for Heterogeneous Integration (CHI) programming environment [27] is a different approach to tightly integrate accelerators such as GPUs and general purpose CPU cores together based on the proposed EXOCHI [27] model. EXOCHI supports a shared virtual memory heterogeneous multi-threaded programming model with minimal OS intrusion. In CUDA execution model, GPU is a device with a separate address space from CPU. As a result, all data communication and synchronization between CPU and GPU is explicitly performed through the GPU device driver.

3. Architecture Overview

The GeForce 8800 GPU is effectively a large set of processor cores with the ability to directly address into a global memory. This allows for a more general and flexible programming model than previous generations of GPUs, making it easier for developers to implement data-parallel kernels. In this section we discuss NVIDIA's Compute Unified Device Architecture (CUDA) and the major microarchitectural features of the GeForce 8800. A more complete description can be found in [3, 18]. It should be noted that this architecture, although more exposed than previous GPU architectures, still has details which have not been publicly revealed.

3.1 Threading Model

The CUDA programming model is ANSI C extended by several keywords and constructs. The GPU is treated as a coprocessor that executes data-parallel kernel code. The user supplies a single source program encompassing both host (CPU) and kernel (GPU) code. These are separated and compiled as shown in Figure 1. Each CUDA program consists of multiple phases that are executed on either the CPU or the GPU. The phases that exhibit little or no data parallelism are implemented in host (CPU) code, which is expressed in ANSI C and compiled with the host C compiler as shown in Figure 1. The phases that exhibit rich data parallelism are implemented as kernel functions in the device (GPU) code. A kernel function defines the code to be executed by each of the massive number of threads to be invoked for a data-parallel phase. These kernel functions are compiled by the NVIDIA CUDA C compiler and the kernel GPU object code generator. There are several restrictions on kernel functions: there must be no recursion, no static variable declarations, and a non-variable number of arguments. The host code transfers data to and from the GPU's global memory using API calls. Kernel code is initiated by performing a function call.

Threads executing on the GeForce 8800 are organized into a three-level hierarchy. At the highest level, all threads in a data-parallel execution phase form a *grid*; they all execute the same kernel function. Each grid consists of many thread blocks. A grid can be at most $2^{16} - 1$ blocks in either of two dimensions, and each block has unique coordinates. In turn, each thread block is a three-dimensional array of threads, explicitly defined by the application developer, that is assigned to an SM. The invocation parameters of

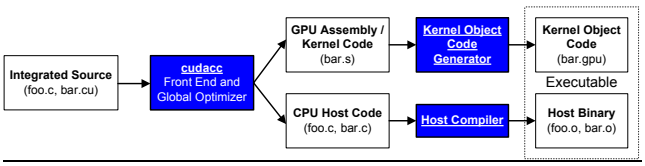

Figure 1. CUDA Compilation Flow

a kernel function call define the organization of the sizes and dimensions of the thread blocks in the grid thus generated. Threads also have unique coordinates and up to 512 threads can exist in a block. Threads in a block can share data through a low-latency, on-chip shared memory and can perform barrrier synchronization by invoking the `__syncthreads` primitive. Threads are otherwise independent; synchronization across thread blocks can only be safely accomplished by terminating a kernel. Finally, the hardware groups threads in a way that affects performance, which is discussed in Section 3.2.

An application developer for this platform can compile CUDA code to an assembly-like representation of the code called *PTX*. PTX is not natively executed, but is processed by a run-time environment, making it uncertain what instructions are actually executed on a cycle-by-cycle basis. Two examples we have observed are simple cases of loop-invariant code that can be easily moved and branches which are split into condition evaluations and predicated jump instructions. However, PTX is generally sufficient in the initial stages of estimating resource requirements of an application and optimizing it.

3.2 Base Microarchitecture

Figure 2 depicts the microarchitecture of the GeForce 8800. It 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 multi-cycle latency. The arithmetic units and the SFUs are fully pipelined, yielding 388.8 GLOPS (16 SMs * 18 FLOPS/SM * 1.35GHz) of peak theoretical performance for the GPU.

Each SM has 8192 registers which are dynamically partitioned among the threads running on it. Non-register memories with distinctive capabilities and uses are described in Table 1 and depicted in Figure 2. Variables in the source code can be declared to reside in global, shared, local, or constant memory. Texture memory is accessed through API calls which compile to special instructions. Bandwidth to off-chip memory is very high at 86.4 GB/s, but memory bandwidth can saturate if many threads request access within a short period of time. In addition, this bandwidth can be obtained only when accesses are contiguous 16-word lines; in other cases the achievable bandwidth is a fraction of the maximum. Optimizations to coalesce accesses into 16-word lines and reuse data are generally necessary to achieve good performance.

There are several non-storage limits to the number of threads that can be executed on the system. First, a maximum of 768 simultaneously active thread contexts is supported per SM. Second, an integral number of up to eight thread blocks can be run per SM at one time. The number of thread blocks that are simultaneously resident on an SM is limited by whichever limit of registers, shared memory, threads, or thread blocks is reached first. This has two con-

Table 1. Properties of GeForce 8800 Memories

Memory	Location	Size	Hit Latency	Read-Only	Program Scope	Description
Global	off-chip	768MB total	200-300 cycles	no	global	Large DRAM. All data reside here at the beginning of 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.
Local	off-chip	up to global	same as global	no	function	Space for register spilling, etc.
Shared	on-chip	16KB per SM	≃register latency	no	function	Local scratchpad that can be shared between threads in a thread block. Organized into 16 banks. Does not appear to have error detection. If instructions issued in the same cycle access different locations in the same bank, a bank conflict stall occurs. It is possible to organize both threads and data such that bank conflicts seldom or never occur.
Constant	on-chip cache	64KB total	≃register latency	yes	global	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	global	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.

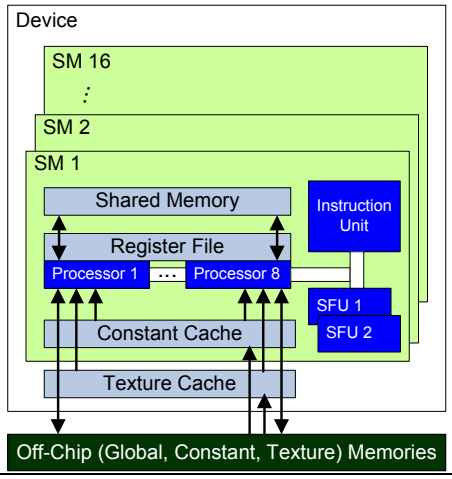

Figure 2. Basic Organization of the GeForce 8800

sequences. First, optimization may have negative effects in some cases because small changes have multiplicative resource usage effects (due to the large number of threads) that cause fewer thread blocks and thus threads to be simultaneously executed. Second, it is relatively easy to be "trapped" in a local maximum when hand-optimizing code. Developers may need to try widely varying configurations to find one with satisfactory performance.

During execution, threads within a block are grouped into *warps* of 32 parallel threads, which are the granular multi-threading scheduling unit. Warps are formed from continuous sections of threads in a thread block: the first 32 threads in a block form the first warp, etc. Although warps are not explicitly declared in CUDA code, knowledge of them can enable useful code and data optimizations on the GeForce 8800. A scoreboard indicates when all of a warp's operands are ready for execution. It then executes the same instruction for the 32 threads in the warp. An SM issues only one instruction at a time for all threads in a warp; when threads in a warp take different control paths, it is assumed that multiple passes with suppression of threads on divergent paths are required to complete execution. It is generally desirable to group threads to avoid this situation. If a thread block is not evenly divisible by the warp size, any remaining issue slots are wasted.

An SM can perform zero-overhead scheduling to interleave warps and 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 resident in 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 barrier synchronizations among threads in the same thread block.

In summary, there are hard limits to the memories, threads, and total bandwidth available to an application running on the GeForce 8800. Managing these limits is critical when optimizing applications, but strategies for avoiding one limit can cause other limits to be hit. They can also reduce the number of thread blocks that can run simultaneously. In addition, managing the behavior of threads so that those in the same warp follow the same control paths and load contiguous values from global memory can also improve performance.

4. Performance and Optimization

This section uses a microbenchmark to demonstrate how the proper balancing of shared resource usage is critical to achieving efficient execution resource utilization and thus high performance on the GeForce 8800. There are three basic principles to consider when optimizing an application for the platform. First, *the floating point throughput of an application depends on the percentage of its instructions that are floating point operations*. The GPU is capable of issuing 172.8 billion operations per second on the SPs. These include fused multiply-add operations, which we count as two operations for throughput calculations. If 1/4 of an application's instruction mix are fused multiply-adds, then its performance can be at most 2 * 1/4 FP * 172.8 billion ops per second = 86.4 GFLOPS. This performance is reached when the SPs are fully occupied, which is achievable in an application that has many threads, does not have many synchronizations, and does not stress global memory bandwidth. In this situation, reducing the number of instructions that do not contribute to data computation generally results in kernel speedup. However, maximizing computational efficiency can be challenging, due to discontinuities in the optimization space [22].

Second, *when attempting to achieve an application's maximum performance, the primary concern often is managing global memory latency*. This is done by creating enough threads to keep SPs occupied while many threads are waiting on global memory accesses. As previously stated, threads may need to of a finer granularity than those for traditional multicore execution to generate enough threads. The required number of threads depends on the percentage of global accesses and other long-latency operations in an application: applications consisting of a small percentage of these operations require fewer threads to achieve full SP occupancy. The

limit on registers and shared memory available per SM can constrain the number of active threads, sometimes exposing memory latency. We show one example where the use of additional registers in an attempted optimization allows one fewer thread block to be scheduled per SM, reducing performance.

Finally, *global memory bandwidth can limit the throughput of the system*. Increasing the number of threads does not help performance in this situation. Alleviating the pressure on global memory bandwidth generally involves using additional registers and shared memory to reuse data, which in turn can limit the number of simultaneously executing threads. Balancing the usage of these resources is often non-intuitive and some applications will run into resource limits other than instruction issue on this architecture.

The example we use to illustrate these principles is a matrix multiplication kernel. In matrix multiplication, the value of an element in the result matrix is calculated by computing the dot product of the corresponding row of the first matrix and column of the second matrix. For this example we assume densely populated input matrices. We analyze several code versions and their sustained performance when multiplying two square matrices with a height and width of 4096 elements. The stated resource usage is for CUDA version 0.8; later versions of CUDA may have different usages.

4.1 Initial Code Version

We begin with a simple version of matrix multiplication. The matrix multiplication kernel creates a thread for each result element for the multiplication, for a total of 4K*4K threads. Many threads are created in an attempt to hide the latency of global memory by overlapping execution. These threads loop through a sequence that loads two values from global memory, multiplies them, and accumulates the value. Figure 3(a) shows the core loops of the dot-product computation kernel; starting values for `indexA`, `indexB`, and `indexC` are determined by block and thread coordinates, which the hardware supports. This code uses ten registers per thread, allowing the maximum of 768 threads to be scheduled per SM. For convenience, we group them as three thread blocks of 256 threads each.

Performance for this code is 10.58 GFLOPS, which is lower than highly optimized libraries executing on a CPU using SIMD extensions. By examining the PTX for this code, we find that there is approximately[3] one fused multiply-add out of eight operations in the inner loop, for a estimated potential throughput of 43.2 GFLOPS. Because we have the maximum number of threads scheduled per SM, the bottleneck appears to be global memory bandwidth. 1/4 of the operations executed during the loop are loads from off-chip memory, which would require a bandwidth of 173 GB/s (128 SPs * 1/4 instructions * 4 B/instruction * 1.35GHz) to fully utilize the SPs.[4] Thus, the strategy for optimizing this kernel is to improve data reuse and reduce global memory access.

4.2 Use of Local Storage

In Figure 3(a), although the computations of two result elements in the same row or column share half their input data (the same `indexA` or `indexB` values), the previous code accesses global memory for each datum in every thread. A common optimization for this type of access pattern is to enhance data sharing via tiling [14]. In the GeForce 8800, developers can utilize shared memory to amortize the global latency cost when values are reused. Using low-overhead block synchronization values, can be shared *between* threads: one thread loads a datum and then synchronizes so that other threads in the same block can use it. Finally, we can also take advantage of contiguity in main memory accesses when loading in values as a block, reducing the cycles needed to access the data.

Figure 3(b) shows the code for a tiled matrix multiplication, with a tile size of 16x16, or 256 result elements and threads. During execution, the threads work within two input tiles that stride across 16 contiguous rows or columns in the input matrices. Each of the 256 threads is tied to a specific coordinate in a tile. It loads the element at that coordinate from the two input tiles into shared memory, so cooperatively the threads load the complete tiles. These loads are organized to take advantage of global access coalescing. The threads then synchronize to establish consistency, which enables each thread to load all of its inputs contained in the tiles from shared memory. Finally, the threads calculate the partial dot product for the inputs in shared memory within a loop.

The choice of tile shape and size is a key decision. For a given size, square tiles generally improve the computation to memory access ratio by improving data locality among threads. Larger tile sizes increase data sharing and thus global memory efficiency. The tiling support code adds several overhead instructions per tile, which also makes larger sizes more efficient. On this architecture, developers also need to consider whether a tile size provides enough threads to have good occupancy. Figure 4 shows the results of experimenting with different tile sizes. 4x4 tiles use only 16 threads, so half of the issue slots in a warp would go unused. This inefficiency, coupled with the 8 thread block limit, causes performance to be worse than the non-tiled code. 8x8 tiles create thread blocks that occupy two warps, but would still need 12 thread blocks to fully occupy an SM, 50% more than the supported limit. 12x12 tiles use 144 threads, which is also not an integral number of warps, and also requires padding of the arrays to prevent overrun. 16x16 is the largest convenient size for this platform, and we can schedule three thread blocks of 8 warps each, for the maximum of 768 threads. Global memory coalescing also happens naturally with this configuration. Other applications may have higher performance with smaller tile sizes when they allow a larger number of threads to be scheduled.

The use of 16x16 tiles reduces global loads by a factor of 16 over the non-tiled configuration, so instead of the bandwidth requirement being twice what is available, it is now approximately an eighth, removing it as the bottleneck to performance. The additional instructions reduce the potential throughput slightly below that of the original code. The 16x16 tiled version of matrix multiplication achieves 46.49 GFLOPS, or approximately 4.5X the execution throughput of the initial version. This is slightly higher than the estimated potential throughput of the original code, so it appears that the application achieves full usage of the SPs.

Register usage must also be managed to avoid performance losses. Some versions of this code use 11 registers per thread instead of 10. To run three thread blocks, this requires 3 blocks/SM * 256 threads/block * 11 registers = 8488 registers, which is larger than an SM's register file. Thus, each SM executes only two blocks simultaneously, which reduces performance.

4.3 Executed Instruction Reduction

As noted in the previous example, tiling reduces the global memory accesses at the expense of additional instructions. Now that our code achieves its potential throughput, we can examine whether the same work can be done with fewer instructions to improve efficiency and performance. The obvious targets for reduction are those operations which are not part of the core data computation, such as branches and address calculations. Common subexpres-

[3] As previously mentioned, PTX code does not necessarily translate to executed instructions, so instruction counts are estimates.

[4] This is also an estimate. Threads can simultaneously load the same value from memory and the memory system may be able to combine these into a single request.

```
Ctemp = 0;

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

C[indexC] = Ctemp;
```

(a) Initial Version

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

  // load input tile elements
  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

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

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

(b) Tiled Version

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

  // load input tile elements
  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  // compute results for tile
  Ctemp +=
     As[ty][0] * Bs[0][tx];
  ...
  Ctemp +=
     As[ty][15] * Bs[15][tx];

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

(c) Unrolled Version

Figure 3. Partial Kernel Codes for Matrix Multiplication. CUDA keywords are bold.

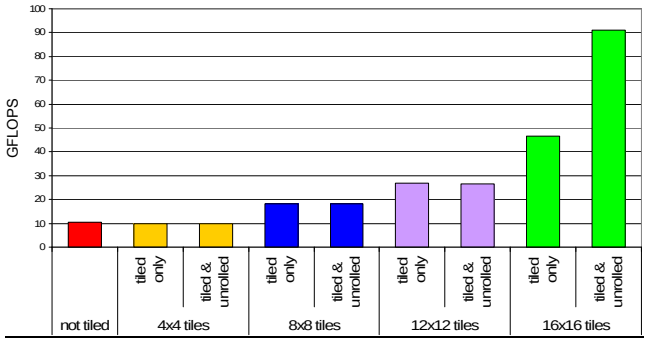

Figure 4. Performance of Matrix Multiplication Kernels

sion elimination and loop unrolling are two classical compiler optimizations that can achieve this goal. What is less clear is whether these operations increase or reduce the number of registers used per thread and thus affect the number of thread blocks that can be scheduled per SM. The compiler's scheduler further complicates matters, as it may attempt to improve the execution speed of each thread at the cost of extra registers.

For tiled matrix multiplication, the innermost loop that computes the partial dot product has a small body and constant iteration count. This can be unrolled by several different factors, each removing some test and branch instructions. However, the best performance can be achieved by completely unrolling the loop. This has the effect of removing all loop branches, induction variable increments, and inner loop address calculation instructions, since the offsets are now constants. It also reduces the register usage by one, to 9 registers, by eliminating an induction variable. The PTX code for the unrolled 16x16 tiled version shows that approximately 16 out 59 instructions, slightly higher than 1/4, are fused multiply-adds. From that, we can calculate potential throughput of this code at 93.72 GFLOPS, with memory bandwidth requirements still below the amount available. The achieved performance of the code is 91.14 GFLOPS, similar to highly-optimized CUDA 0.8 libraries provided by NVIDIA.

In general the unrolling of small inner loops will produce positive gain when memory bandwidth is not already an issue and scheduling does not trigger extra register usage that reduces the number of active thread blocks. Unrolling outer loops is less likely to provide benefit because they contribute fewer branches to overall execution and have more effect on instruction cache efficiency. In this case, shared memory usage is not affected and a register is saved by removing the unrolled loop's induction variable, although it is not used for anything else. The performance of other tile sizes is only marginally improved by unrolling.

4.4 Balancing Applications and Optimization Interaction

At this point, the matrix multiplication code appears to be well-optimized, with actual performance near that of the estimated potential. A large portion of the non-data computation instructions have been removed. Registers and threads are fully utilized. There is still a significant amount of shared memory available, but there is no obvious way to use it to increase performance.

In an effort to further improve performance, a developer can attempt to improve SP occupancy by reducing exposed intrathread global memory latency. We implemented a prefetching optimization that initiates loads to the next tiles prior to performing computation for the current tile. The optimization also increases the number of registers required by each thread by two, to 11. As previously mentioned, this reduces the number of blocks that can be scheduled per SM by 1, reducing simultaneous thread count by a third. This version was capable of 87.10 GFLOPS performance, inferior to performing only tiling and unrolling.

In this case, intra-thread latency reduction is insufficient to make up for the reduction of simultaneously executed threads. However, the difference between the performances of the two configurations is only 5%. Although we have reduced the number of simultaneously active threads by a third, these threads take nearly a third less time to execute because the prefetching optimization eliminates much of the time threads wait on global memory. This illustrates the principle that although many threads are generally desirable, full utilization of execution resources is achieved when there are enough threads to avoid being stalled on global memory access. These kinds of optimization interactions, plus the uncertainty of the architecture features and code executed, make it challenging to find the peak performance of an application on this architecture.

5. Application Study

We performed an application study with the intent of testing the applicability and effectiveness of the principles in Section 4 on real applications. We have selected a suite of applications acquired from various sources that have different purposes and code behavior but are also reasonably well-suited for execution on the GeForce 8800.

These applications, even ones with kernels of a few hundred lines, often have a large variety of instructions, operate on larger data sets, and have more control flow than microbenchmarks. Many of these contribute to bottlenecks other than instruction issue bandwidth on this platform, enabling a better evaluation of the system. To our knowledge, this is the first study of this breadth on a GPU.

Table 2 lists some of the applications that have been ported to CUDA, along with source and kernel lines of code (excluding comments and whitespace). Benchmark versions of the applications are available [2]. The larger codes often required more modification to port to CUDA; the most extreme case was H.264, which involved a large-scale code transformation to extract the motion estimation kernel from non-parallel application code. The percentage of single-thread CPU execution time spent in kernels is given to show the total application speedup that can be achieved as limited by Amdahl's Law. For example, FDTD's kernel takes only 16.4% of execution time, limiting potential application speedup to 1.2X. In general, kernel execution occupied the vast majority of CPU-only execution for these applications.

Table 3 shows characteristics of the optimized application implementations. The data for matrix multiplication is listed for comparison.[5] The maximum number of simultaneously active threads shows the amount of thread parallelism available on the hardware at a given time, taking resource constraints into account, with a maximum of 12288 across the 16 SMs. There is a wide range of values, with little correlation to actual speedup. The total threads in a given application often numbers in the millions. The number of registers and the amount of shared memory per thread show the degree of local resource utilization.

Other information in the table includes the ratio of global memory cycles to computation cycles after shared memory and caches are utilized to their fullest extent, expressing the global memory bandwidth requirements of the most time-consuming kernel of each application. We discuss how this correlates to performance in Section 5.1. GPU execution time expresses how much of the total execution time the application kernels occupy after being ported to the GPU. CPU-GPU transfer time is shown for comparison with the computation time. One interesting case is H.264, which spends more time in data transfer than GPU execution. Finally, we list the architectural bottleneck(s) that appear to be limiting these implementations from achieving higher performance.

The two rightmost columns of Table 3 list the performance of ported applications. The baseline, single-thread CPU performance is measured on an Opteron 248 system running at 2.2GHz with 1GB main memory. The choice was made with the intent of having a high-performance, single-core processor; similar CPU performance is found with newer, high clock rate multicore architectures. For applications with outstanding GPU speedup, we applied optimizations such as SIMD instructions and fast math libraries to the CPU-only versions to ensure that comparisons were reasonable. We measure both the speedup of CUDA kernel execution over single-thread CPU kernel execution and total application speedup, with all floating point numbers set to single-precision. Measurements were made with typical long-running inputs; e.g., for SPEC CPU benchmarks the reference inputs were used.

5.1 Performance Trends of Optimized Applications

In general, we obtain significant kernel and application speedup across our suite, as shown in Table 3. Compute-intensive kernels with relatively few global memory accesses achieve very high performance. Even kernels which are not as compute-intensive still achieve respectable performance increases because of the GeForce 8800's ability to run a large number of threads simultaneously. Low-latency floating-point execution is a major factor in speedup, as is the use of caches and shared memory to reduce latencies and global bandwidth usage. Loop unrolling is effective in improving performance for some applications. Careful organization of threads and data reduces or eliminates conflicts in shared memory and caches, most notably in the MRI applications.

The applications in Table 3 with the highest performance gains, namely TPACF, RPES, MRI-Q, MRI-FHD, and CP, have low global access ratios and spend most of their execution time performing computation or accessing low-latency memories. They also generate enough threads to hide potential stalls on long-latency operations and maintain high pipelined floating point throughput.

The MRI applications achieve particularly high performance and require additional explanation. One major reason for their performance is that a substantial number of executed operations are trigonometry functions; the SFUs execute these much faster than even CPU fast math libraries. This accounts for approximately 30% of the speedup. We also spent significant effort improving the CPU versions (approximately 4.3X over the original code) to ensure that these comparisons were reasonable. The opposite effect, where the native instruction set must emulate functionality, exists in RC-5: the GeForce 8800 lacks a modulus-shift operation. Performance of the code if a native modulus-shift were available is estimated to be several times higher.

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

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

5.2 Optimizations

In this section we discuss some of the optimizations that have significant effects on performance and the corresponding applications. In general, shared memory is useful for reducing redundant loads and thus pressure on memory bandwidth. Its use is straightforward when there are either no shared values between threads (each thread effectively has its own private space) or when neighboring threads share data in a simple pattern, similar to matrix multiplication. Care must be taken so that threads in the same warp access different banks of the shared memory. In addition, more complex applications often use more sophisticated data structures.

One use of shared memory is buffering to improve the access pattern of global memory. As stated previously, memory band-

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

Table 2. Application Suite

Application	Description	Source Lines	Kernel Lines	CPU Execution Parallelized
H.264	A modified version of the 464.h264ref benchmark from SPEC CPU2006. This is an H.264 (MPEG-4 AVC) video encoder. A serial dependence between motion estimation of macroblocks in a video frame is removed to enable parallel execution of the motion estimation code. Although this modification changes the output of the program, it is allowed within the H.264 standard.	34811	194	35%
LBM	A modified version of the 470.lbm benchmark from SPEC CPU2006. This uses the Lattice-Boltzman Method for simulating 3D fluid dynamics. The program has been changed to use single-precision floating point and print fewer status reports.	1481	285	$> 99\%$
RC5-72	This application accelerates distributed.net's RSA RC5-72 bit challenge, which performs brute-force encryption key generation and matching.	1979	218	$> 99\%$
FEM	Finite Element Modeling. Simulation of dynamic behavior of 3D graded materials.	1874	146	99%
RPES	Rys Polynomial Equation Solver. Calculates 2-electron repulsion integrals, which are a sub-problem of molecular dynamics.	1104	281	99%
PNS	Petri Net Simulation. Simulation of a mathematical representation of a distributed system.	322	160	$> 99\%$
SAXPY	Single-precision floating-point implementation of `saxpy` from High-Performance LINPACK, used as part of a Gaussian elimination routine.	952	31	$> 99\%$
TPACF	Implementation of Two Point Angular Correlation Function, used to find the probability of finding an astronomical body at a given angular distance from another astronomical body.	536	98	96%
FDTD	Finite-Difference Time-Domain. 2D electromagnetic wave propagation simulation in an arbitrary, user-defined medium.	1365	93	16.4%
MRI-Q	Computation of a matrix Q, representing the scanner configuration, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.	490	33	$> 99\%$
MRI-FHD	Computation of an image-specific matrix $F^H d$, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.	343	39	$> 99\%$
CP	Computation of electric potential in a volume containing point charges. Based on direct Coulomb summation, as described in [24].	409	47	$> 99\%$

Table 3. Application Implementation Performance For Typical, Long-Running Execution Profiles

Application	Max Simultaneously Active Threads	Registers per Thread	Shared Mem per Thread (B)	Global Memory to Computation Cycles Ratio	GPU Exec %	CPU-GPU Transfer %	Architectural Bottleneck(s)	Kernel Speedup on GPU	Application Speedup
Mat Mul	12288	9	8.1	0.276	16.2%	4%	Instruction issue	7.0X	2.0X
H.264	3936	30	55.1	0.006	2.6%	4.5%	Register file capacity and cache latencies	20.2X	1.47X
LBM	3200	32	84.2	0.415	98.3%	0.4%	Shared memory capacity	12.5X	12.3X
RC5-72	3072	42	0.3	$\simeq$0	64.3%	0.5%	Instruction issue	17.1X	11.0X
FEM	4096	18	61	1.135	91.4%	$\ll 1\%$	Global memory bandwidth	11.0X	10.1X
RPES	4096	23	24.8	0.01	37.5%	1%	Instruction issue	210X	79.4X
PNS	2048	32	9.9	0.241	98%	$\ll 1\%$	Global memory capacity	24.0X	23.7X
SAXPY	12288	7	0.3	0.375	88%	4.5%	Global memory bandwidth	19.4X	11.8X
TPACF	4096	24	52.2	0.0002	34.3%	$\ll 1\%$	Shared memory capacity	60.2X	21.6X
FDTD	12288	11	8.1	0.516	1.8%	0.9%	Global memory bandwidth	10.5X	1.16X
MRI-Q	8192	11	20.1	0.008	$> 99\%$	$\ll 1\%$	Instruction issue	457X	431X
MRI-FHD	8192	12	20.1	0.006	99%	1%	Instruction issue	316X	263X
CP	6144	20	0.4	0.0005	$> 99\%$	$\ll 1\%$	Instruction issue	102X	102X

width is easily saturated when requests are not performed at 16-word granularities. LBM, FEM, FDTD, and other lattice computations use arrays of small structures in global memory. Threads simultaneously read or write a given field of multiple elements, but these fields are not contiguous in memory. Each non-contiguous access is a separate DRAM access request, overwhelming the device's memory bandwidth. In LBM we alleviated the problem using contiguous accesses to prefetch the arrays into shared memory. Figure 5 illustrates the access patterns before and after the optimization. Before computation, threads cooperatively load blocks of memory into shared memory, as shown in Figure 5(b). They then synchronize, after which each thread operates on its own data. The buffering optimization may also be possible with FDTD if a substantial amount of data reorganization is performed, but FEM uses an irregular mesh data structure that has few contiguous accesses even with data reorganization.

On-chip caches are useful in several applications; we focus on two here. For the MRI applications, we placed data in constant memory, which reduced average access time [25]. We also performed a loop interchange to make all threads in a warp simultaenously access the same value in the table to remove conflicts. Constant memory is generally intended for small lookup tables, but any data that is read-only and has the same location simultaneously read by all threads is appropriate for it. Our implementation of H.264 uses texture memory for part of the input data, since the data use has 2D locality and the hardware provides boundary-value calculation support that would otherwise need to be calculated in software. However, a lack of registers restricts the number of threads

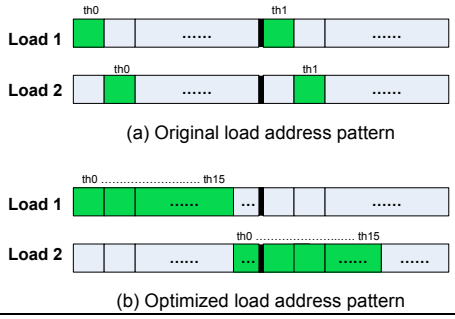

Figure 5. LBM Global Load Access Patterns

that could be scheduled, exposing the latency of texture memory. Even so, kernel performance improves by 2.8X over global-only access by the use of texture memory.

Loop unrolling and other "classic" compiler optimizations can have unexpected results, but in general local optimizations on the most frequently executed parts of the code has beneficial effects. Benefit is due to the reduction in the number of operations or strength reduction of individual operations such as integer multiply, thus increasing overall computational efficiency. In H.264, complete unrolling of the innermost loop obtains significant performance increase, as did register tiling [10] for the next two higher-level loops.

The common case of compiler optimizations having negative effects is when they increase the number of registers per thread as a side effect, forcing the GeForce 8800 to schedule fewer thread blocks per SM and thus degrading performance. The cases where this is most often seen are common subexpression elimination and redundant load elimination, the latter often storing thread and block coordinates in registers. Even relatively simple instruction scheduling can change the live ranges of variables and increase the register usage. Register pressure-sensitive code scheduling algorithms and optimization strategies have been investigated in the context of instruction-level parallelism compilers. Additional research is needed to apply these strategies to massively threaded environments like CUDA. We will address the control of register usage in future work.

6. Conclusion and Future Work

We present a performance evaluation of the GeForce 8800 GTX architecture using CUDA. Although its primary market is graphics processing, this GPU is also capable of impressive performance on a set of disparate non-graphics applications. This work presents general principles for optimizing applications for this type of architecture, namely having efficient code, utilizing many threads to hide latency, and using local memories to alleviate pressure on global memory bandwidth. We also present an application suite that has been ported to this architecture, showing that application kernels that have low global memory access after optimization have substantial speedup over CPU execution if they are not limited by local resource availability.

We are currently performing research on automated optimizations for this architecture. Although many of the optimizations are classical ones, the effects they have on this architecture can be different from the effects on traditional superscalar processors. It is also possible to get stuck in local maximums of performance when attempting to follow a particular optimization strategy. These maximums may be significantly lower than the peak achievable performance. Better tools and compilers that allow programmers to specify the types of reorganizations desired and automatically experiment with their performance effects would greatly reduce the optimization effort. In addition, two updated versions of CUDA have been released between the original and final submission of this paper, changing resource usages and optimal configurations of many applications. We are exploring methods to preserve or enhance performance of applications when shifts in the underlying architecture or runtime occur.

Acknowledgments

The authors thank John Nickolls, Mark Harris, and Michael Cox at NVIDIA for their advice on drafts of the paper. We also thank Sanjay Patel, Kurt Akeley, Pradeep Dubey, John Kelm, Hillery Hunter, and the anonymous reviewers for their feedback. We thank the Spring 2007 class of ECE498AL at the University of Illinois at Urbana-Champaign for their work on initial versions of many of the applications presented in this paper. We also thank the other members of the IMPACT research group for their support.

Sam Stone is supported under a National Science Foundation Graduate Research Fellowship. This work was supported by the Gigascale Systems Research Center, funded under the Focus Center Research Program, a Semiconductor Research Corporation program. Experiments were made possible by generous hardware loans from NVIDIA and NSF CNS grant 05-51665. 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.

References

[1] AMD Stream Processor. http://ati.amd.com/products/ streamprocessor/index.html.

[2] CUDA benchmark suite. http://www.crhc.uiuc.edu/impact/cudabench.html.

[3] NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html.

[4] The PeakStream platform: High productivity software development for multi-core processors. Technical report, 2006.

[5] ECE 498AL1: Programming massively parallel processors, Fall 2007. http://courses.ece.uiuc.edu/ece498/al1/.

[6] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 90 handbook: complete ANSI/ISO reference*. Intertext Publications, Inc.,/McGraw-Hill, Inc., 1992.

[7] R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Langugages and Systems*, 9(4):491–542, 1987.

[8] M. J. Atallah, editor. *Algorithms and Theory of Computation Handbook*. CRC Press LLC, 1998.

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

[10] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. *ACM SIGPLAN Notices*, 9(4):328–342, 2004.

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

[12] 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*, number 89, 2006.

[13] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[14] 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.

[15] D. B. Loveman. High Performance Fortran. *IEEE Parallel & Distributed Technology: Systems & Technology*, 1(1):25–42, 1993.

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

[17] M. D. McCool, K. Wadleigh, B. Henderson, and H.-Y. Lin. Performance evaluation of GPUs using the RapidMind development platform. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

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

[19] OpenMP Architecture Review Board. OpenMP application program interface, May 2005.

[20] J. Owens. Streaming architectures and technology trends. *GPU Gems 2*, pages 457–470, March 2005.

[21] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

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

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

[24] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28(16):2618–2640, December 2007.

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

[26] 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, 2006.

[27] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–166, 2007.

[28] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, 1990.