To appear in the Journal of Parallel and Distributed Computing, Elsevier

A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Kevin Skadron [1]

University of Virginia, Department of Computer Science, Charlottesville, VA, USA

Abstract

Graphics processors (GPUs) provide a vast number of simple, data-parallel, deeply multithreaded cores and high memory bandwidths. GPU architectures are becoming increasingly programmable, offering the potential for dramatic speedups for a variety of general-purpose applications compared to contemporary general-purpose processors (CPUs). This paper uses NVIDIA's C-like CUDA language and an engineering sample of their recently introduced GTX 260 GPU to explore the effectiveness of GPUs for a variety of application types, and describes some specific coding idioms that improve their performance on the GPU. GPU performance is compared to both single-core and multicore CPU performance, with multicore CPU implementations written using OpenMP. The paper also discusses advantages and inefficiencies of the CUDA programming model and some desirable features that might allow for greater ease of use and also more readily support a larger body of applications.

Key words: Graphics processors, GPU, GPGPU, CUDA, OpenMP, Parallel Programming, Heterogeneous Computing Organizations, Multicore, Manycore

1 Introduction

Semiconductor scaling limits and associated power and thermal challenges, combined with the difficulty of exploiting greater levels of instruction level parallelism, have combined to limit performance growth for single-core microprocessors. This

Email addresses: sc5nf@cs.virginia.edu (Shuai Che), boyer@cs.virginia.edu (Michael Boyer), jm6dg@virginia.edu (Jiayuan Meng), dtarjan@cs.virginia.edu (David Tarjan), jws9c@cs.virginia.edu (Jeremy W. Sheaffer), skadron@cs.virginia.edu (Kevin Skadron).

[1] Skadron was on sabbatical with NVIDIA Research for the 2007-08 academic year.

has led most microprocessor vendors to turn instead to multicore chip organizations, even though the benefits of multiple cores can only be realized if the programmer or compiler explicitly parallelize the software.

In this context, graphics processors (GPUs) become attractive because they offer extensive resources even for non-visual, general-purpose computations: massive parallelism, high memory bandwidth, and general purpose instruction sets, including support for both single- and double-precision IEEE floating point arithmetic (albeit with some limitations on rounding). In fact, GPUs are really "manycore" processors, with *hundreds* of processing elements.

The advent of general purpose computing on GPUs makes it important to understand when GPUs are preferable to conventional, multicore CPUs. As new parallel computing platforms such as GPUs and multicore CPUs have come to dominate the market, it also becomes important to revisit parallel programming models and to find the best balance between programming convenience and hardware implementation costs. This paper begins to explore the extent to which traditionally CPU domain problems can be mapped to GPU architectures using current parallel programming models. A recent report from Berkeley [1] argued that successful parallel architectures should perform well over a set of 13 representative classes of problems, termed *dwarves*, which each capture a body of related problems.

Inspired by this work, and noting an apparent architectural convergence of CPUs and GPUs, our goal in this paper is to examine the effectiveness of CUDA as a tool to express parallel computation with differing sets of performance characteristics—problems from different dwarves—on GPUs. We quantitatively compare the performance of a series of applications running on an early engineering sample of a NVIDIA GeForce GTX 260 GPU and on a state-of-the-art multicore CPU system with dual 3.2 GHz, dual-core, hyperthreaded Intel Xeon processors.

We developed our GPU applications using CUDA and the CPU applications with OpenMP. We also examine the complexity of mapping CUDA computational kernels to better leverage the GPU's high core count, software-managed threads, and unique memory hierarchy, and explore some optimization techniques which are sometimes non-intuitive. For example, we introduce a pyramid data structure in the *HotSpot* benchmark to avoid the need for synchronization between thread blocks, and we show that lookup tables are useful for avoiding a large control flow overhead when an application involves many irregular data permutations. All of our applications show satisfactory speedups, but the main contribution of our work is a discussion of the advantages and disadvantages of the CUDA programming model in the context of a more general multicore world. For most of our benchmarks, the single-threaded, CUDA, and OpenMP code and sample inputs used for this study are publicly available on our website at `http://lava.cs.virginia.edu/wiki/rodinia`.

Performance results from the the broad application domains we examine, along with previously published data, demonstrate the tremendous potential of the CUDA model for parallel computing; however, we also note several modifications that would allow CUDA and the underlying hardware architecture to improve the ability of programmers to implement complex applications on the GPU.

2 Related Work

In recent years, a large body of work has explored how to use GPUs for general purpose computing, sometimes known as "GPGPU." Before the advent of general purpose languages for GPGPU, GPU implementations could only be achieved using existing 3D-rendering APIs: OpenGL [12] or DirectX [16]. The syntax, the need to pose problems in the context of polygon rasterization, and the limits imposed by pixel independence all made this approach cumbersome. Independently from GPU vendor efforts, several new languages or APIs were created to provide a general-purpose interface and abstract away the necessary 3D API calls. Brook [4], Sh [15] and its commercial successor RapidMind, and Microsoft's Accelerator [27] are notable examples.

Recognizing the value of GPUs for general-purpose computing, GPU vendors added driver and hardware support to use the highly parallel hardware of the GPU without the need for computation to proceed through the entire graphics pipeline (transforming vertices, rasterization, etc.) and without the need to use 3D APIs at all. NVIDIA's solution is the CUDA language, an extension to C described further in Section 3. AMD's solution was the combination of a low-level interface, the Compute Abstraction Layer (CAL) and extensions to Brook.

A wide variety of applications have achieved dramatic speedups with GPGPU implementations. A framework for solving linear algebra problems on graphics processors is presented by Krüger *et al.* [13]. Harris *et al.* present a cloud dynamics simulation using partial differential equations [8], and various other N-body (e.g. [21]) and molecular dynamics simulations (e.g. [23]) have also shown impressive speedups. Some important database operations have also been implemented on the GPU by using pixel engines [7], and a variety of other applications, such as sequence alignment [24] and AES encryption [30] have been successfully implemented on GPUs.

Some recent work has focused on developing sets of parallel primitives to simplify the development of GPGPU applications. One key example is a set of scan primitives developed by Sengupta *et al.* [25]. They implemented the classic scan operation using CUDA, providing a powerful set of library functions to deal with applications with more irregular data structures, and for the first time showed speedups for traditionally non-parallel algorithms including quicksort.

3 CUDA and Tesla Overview

3.1 Key CUDA Abstractions

CUDA is an extension to C based on a few easily-learned abstractions for parallel programming, coprocessor offload, and a few corresponding additions to C syntax. CUDA represents the coprocessor as a device that can run a large number of threads. The threads are managed by representing parallel tasks as kernels (the sequence of work to be done in each thread) mapped over a domain (the set of threads to be invoked). Kernels are *scalar* and represent the work to be done at a single point in the domain. The kernel is then invoked as a thread at every point in the domain. The parallel threads share memory and synchronize using barriers.

Data is prepared for processing on the GPU by copying it to the graphics board's memory. Data transfer is performed using DMA and can take place concurrently with kernel processing. Once written, data on the GPU is persistent unless it is deallocated or overwritten, remaining available for subsequent kernels.

As a trivial example, the following code transfers two vectors to the GPU and then sums them.

```
// CUDA kernel
__global__ void vec_sum(float * in1, float * in2, float * out)
{
    // blockIdx,threadIdx and blockDim are variables provided by CUDA
    int index = blockIdx.x*blockDim.x + threadIdx.x;

    out[index] = in1[index] + in2[index];

    return;
}

void host_function(float * h_in1, float * h_in2, float * h_out, int size)
{
    // allocate gpu memory for the three vectors
    float *d_in1, *d_in2, *d_out;
    cudaMalloc( (void**) &d_in1, size);
    cudaMalloc( (void**) &d_in2, size);
    cudaMalloc( (void**) &d_out, size);

    // copy over the two input vectors to the gpu
    cudaMemcpy( d_in1, h_in1, size, cudaMemcpyHostToDevice);
    cudaMemcpy( d_in2, h_in2, size, cudaMemcpyHostToDevice);

    // execute the kernel on the gpu
    vec_sum<<<size / 256, 256>>>(d_in1, d_in2, d_out);

    // copy the result back to system memory
    cudaMemcpy( h_out, d_out, size, cudaMemcpyDeviceToHost);

    cudaFree(d_in1);
    cudaFree(d_in2);
    cudaFree(d_out);
}
```

As this example illustrates, kernels consist of conventional, scalar C code representing the work to be done at a single point in the domain. CUDA's extensions to the C programming language are fairly minor. A function declaration can include a modifier specifying whether the function will execute on the CPU or the GPU, and each variable declaration in a GPU function can include a type qualifier specifying where in the GPU's memory hierarchy the variable will be stored. Kernels also have special thread-identification variables automatically defined to allow threads to identify their location in the domain and work on separate parts of a data set.

The domain is actually defined with a 5-dimensional structure, in the form of a 2D *grid* of 3D *thread blocks*. Thread blocks are limited to 512 total threads. The significance of the thread block construct is that each thread block is assigned in its entirety to a single *streaming multiprocessor* (SM) and runs as a unit to completion without preemption. All threads within the thread block are simultaneously live and the threads are temporally multiplexed onto the processing elements in a fine-grained, time-sliced manner, but their resources cannot be reclaimed until the entire block of threads completes. The number of thread blocks in a grid can greatly exceed the hardware resources, in which case fresh thread blocks are assigned to SMs as previous thread blocks retire.

In addition to global shared memory, each thread block has available a private, *per-block shared memory* (PBSM) that is only visible to threads within that thread block. The amount of this PBSM that will be used must be defined by the kernel but is limited to 16 kB because it is implemented using fast SRAM, similar to a first-level cache. The PBSM allows threads within a thread block to cooperate in a fine-grained fashion by sharing data among themselves with low latency. Data can also be shared between thread blocks through global memory, which is generally not cached, but the latency is of course much longer.

Synchronization *within* a thread block is entirely managed in hardware. Synchronization *among* thread blocks is achieved by allowing a kernel to complete and starting a new kernel; in effect, a global barrier. It is important to note that the order in which thread blocks are assigned to SMs is arbitrary. Because order of execution among thread blocks within a grid is non-deterministic, and because thread blocks run to completion, it is important to note that thread blocks should never have a producer-consumer relationship due to the risk of deadlock. Producer-consumer relationships must be confined within thread blocks or separated across global barriers (i.e., back-to-back kernels).

By separating the size of the domain from the underlying hardware, CUDA allows the programmer to focus on *available parallelism*. The restrictions on communication among thread blocks define a virtual machine so that the same CUDA program will run on a wide variety of parallel platforms. Indeed, nothing in the CUDA specification prevents CUDA applications from running effectively on other platforms. Recent research has shown that CUDA programs can be compiled to execute efficiently on multicore CPUs [26]. For CPUs, of course, specialized hardware, such as support for transcendentals and texturing, must be implemented in software.

For a more detailed description of CUDA, please refer to Nickolls *et al.* [17].

3.2 GPU Architectural Highlights

NVIDIA's Tesla unified computing architecture is designed to support both graphics and general purpose computing. The programmable processing elements share a common, very general-purpose instruction set that is used by both graphics and general-purpose computation. Each processing element (PE) supports 128 concurrent thread contexts, allowing a very simple pipeline. Latencies are simply tolerated by switching threads. Current Tesla-architecture products can support up to 30720 concurrent threads. Although it describes the previous generation—the GeForce 8800 GTX and related products—Lindholm *et al.* [14] provide a nice description of contemporary NVIDIA GPU architectures.

Each SM consists of 8 processing elements, called *Stream Processors* or SPs. To maximize the number of processing elements that can be accommodated within the GPU die, these 8 SPs operate in SIMD fashion under the control of a single instruction sequencer. The threads in a thread block (up to 512) are time-sliced onto these 8 SPs in groups of 32 called *warps*. Each warp of 32 threads operates in lockstep and these 32 threads are quad-pumped on the 8 SPs. Multithreading is then achieved through a hardware thread scheduler in each SM. Every cycle this scheduler selects the next warp to execute. Divergent threads are handled using hardware masking until they reconverge. Different warps in a thread block need not operate in lockstep, but if threads within a warp follow divergent paths, only threads on the same path can be executed simultaneously. In the worst case, if all 32 threads in a warp follow different paths without reconverging—effectively resulting in a sequential execution of the threads across the warp—a $32\times$ penalty will be incurred. Unlike vector forms of SIMD, Tesla's architecture preserves a scalar programming model, like the Illiac [5] or Maspar [2] architectures; for correctness the programmer need not be aware of the SIMD nature of the hardware, although optimizing to minimize SIMD divergence will certainly benefit performance.

When a kernel is launched, the driver notifies the GPU's work distributor of the kernel's starting PC and its grid configuration. As soon as an SM has sufficient thread and PBSM resources to accommodate a new thread block, a hardware scheduler randomly assigns a new thread block and the SM's hardware controller initializes the state for all threads (up to 512) in that thread block.

The Tesla architecture is designed to support workloads with relatively little temporal data locality and only very localized data reuse. As a consequence, it does not provide large hardware caches which are shared among multiple cores, as is the case on modern CPUs. In fact, there is no cache in the conventional sense: variables that do not fit in a thread's register file are spilled to global memory. Instead, in addition to the PBSM, each SM has two small, private data caches, both of which only hold read-only data: the texture cache and the constant cache. (The name texture comes from 3D graphics, where images which are mapped onto polygons are called textures.) Data structures must be explicitly allocated into the PBSM, constant, and texture memory spaces.

The texture cache allows arbitrary access patterns at full performance. It is useful for achieving maximum performance on coalesced access patterns with arbitrary offsets.

The constant cache is optimized for broadcasting values to all PEs in an SM and performance degrades linearly if PEs request multiple addresses in a given cycle. This limitation makes it primarily useful for small data structures which are accessed in a uniform manner by many threads in a warp.

4 Methodology

4.1 Application Domains

We use Berkeley's *dwarf* taxonomy [1] to choose our applications, which we then implement using CUDA and OpenMP. Each dwarf represents a set of algorithms with similar computation and data movement. We limit our focus to a subset of the dwarves: *Structured Grid, Unstructured Grid, Combinational Logic, Dynamic Programming*, and *Dense Linear Algebra*. Previous work has covered several other dwarves, such as *Fast Fourier Transform* (FFT) [18], *N-Body* [21], and *Monte Carlo* [20]. Our applications exhibit different parallelism and data-sharing characteristics, and thus take advantage of the GPU's parallel computing resources to different degrees. The level of programmer effort required to achieve satisfactory performance also varies widely across different applications. As Figure 1 illustrates, the application access patterns range from bit-level parallelism to row-level parallelism. Each application is discussed in detail in Section 6.

4.2 Experimental Setup

We chose representative commercial products from both the CPU and GPU markets: a dual-socket machine with 4GB of main memory and two hyperthreaded Intel Xeon dual-core processors, each running at 3.2 GHz with 2MB of L2 cache, and an NVIDIA GeForce GTX 260 with NVIDIA driver version 177.11 and CUDA ver-

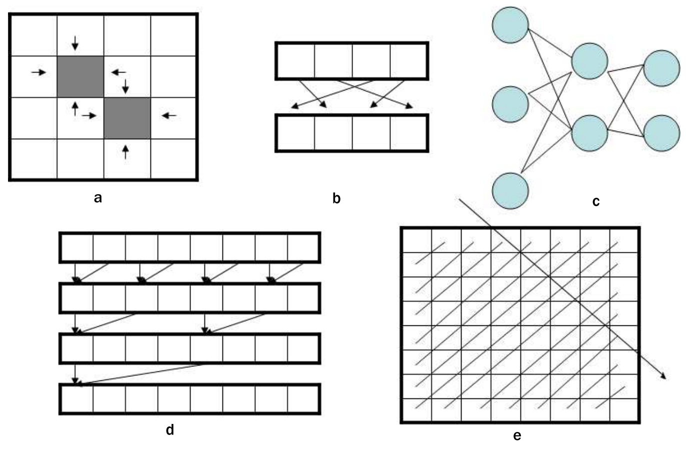


Fig. 1. Communication patterns of different applications: (a) in *SRAD* and *HotSpot*, the value of each point depends on its neighboring points; (b) *DES* involves many bit-level permutations; (c) *Back Propagation* works on a layered neural network; (d) in *SRAD* and *Back Propagation*, parallel reductions can be performed using multiple threads; (e) the parallel *Needleman-Wunsch* algorithm processes the score matrix in diagonal strips.

sion 1.1. The GTX 260 is comprised of 24 streaming multiprocessors (SMs). Each multiprocessor has 8 streaming processors (SPs) for a total of 192 SPs. Each group of 8 SPs shares one 16kB of fast per-block shared memory (similar to scratch-pad memory). Each group of three SMs (i.e., 24 SPs) shares a texture unit. An SP contains a scalar ALU and can perform floating point operations. Instructions are executed in a SIMD fashion across all SPs in a given multiprocessor. The GPU we used for performance measurements is actually an engineering sample of the GTX 260, in which the SPs are clocked at 1.08 GHz instead of the 1.24 GHz clock rate of the commercial version. The commercial board has only 896 MB of device memory while our engineering sample has 1 GB of device memory. The current, top-of-the-line GeForce GTX 280 has 240 SPs and 1 GB of device memory.

We developed the GPU versions of the applications using NVIDIA's CUDA API, and we developed our multithreaded CPU code using the OpenMP API. The performance of each GPU implementation is compared against both single-threaded and four-threaded versions of the CPU implementation. Given the same input dataset, the speedup is calculated by taking the wall-clock time required by the application on the CPU divided by the time required on the GPU. Times are measured after initial setup (e.g., after file I/O) but do include the time required to transfer data between the disjoint CPU and GPU memory spaces. The parallelizable sections of code that we implemented for our GPU kernels all consume within the range of 95.8%–99.7% percent of the total of execution time of their respective single-thread implementations. Our graphs only shows the speedups of these parallel code sections.

When we are only interested in the performance of a specific section of a kernel, we calculate the number of cycles executed using the `clock()` function provided by CUDA. Note that since each SP is time-sliced among different warps, the number of cycles returned by this function does not necessarily represent the actual number of cycles spent executing the thread of interest, but may also include cycles spent executing other warps [19].

5 Hardware Specific Features

Effective CUDA programming requires knowledge of the GPU's underlying hardware architecture. Specific features of the GPU architecture, such as memory transfer overhead, shared memory bank conflicts, and the impact of control flow need to be considered when programming. Programmers can reduce the overhead and improve the performance of their applications by tailoring their algorithms specifically for execution on the GPU.

5.1 Memory Overhead

In modern PCs, the graphics card is connected via a PCI-Express bus to a North Bridge chip, which also connects the CPU and main memory. The data transfer rate of this bus is crucial to the performance of GPGPU applications. Unlike OpenMP—a shared memory programming model—CUDA requires programmers to explicitly manage the data communication between main memory and GPU memory. This memory overhead can have a significant impact on the overall application performance. The time necessary to transfer data increases linearly with the amount of data.

Programmers should thus try to avoid frequent data transfer between the GPU and CPU memories. Often it is better to replicate computation on the GPU rather than increase the amount of communication required with the CPU, and when possible programmers should overlap computation and data communication.

5.2 Bank Conflicts

In the GTX 260, each multiprocessor has a 16 kB, on-chip, software-controlled shared memory which enables efficient data-sharing among threads within a thread block. Physically, each shared memory unit is organized into 16 banks, with successive 32-bit words mapped onto successive banks. Simultaneous accesses to different banks occur in parallel while simultaneous accesses to different addresses within the same bank must be serialized [19]. To measure the overhead of this serialization, we created a kernel which traverses either columns or rows in parallel within a matrix that is comprised of 16×16 data blocks. When traversing

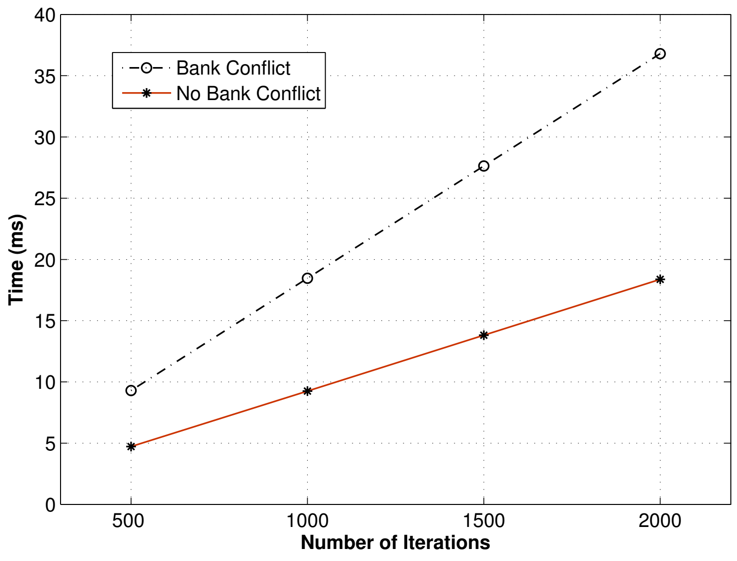


Fig. 2. Overhead of Bank Conflicts.

columns, the kernel exhibits maximal bank conflicts; when traversing rows, the kernel exhibits no bank conflicts. As Figure 2 illustrates, the existence of bank conflicts approximately doubles the kernel's execution time. Although recent work has partially automated the process of detecting bank conflicts [3], it is ultimately the programmer's responsibility to structure their memory accesses in such a way as to avoid bank conflicts, which can require significant performance tuning. On the other hand, dealing with bank conflicts is generally lower priority than maximizing parallelism and data locality.

5.3 Control Flow Overhead

In CUDA, control flow instructions, such as those generated from `if` and `switch` statements, can significantly impact the instruction throughput if threads within the same warp follow different branches. When executing divergent branches, either the execution of each path must be serialized or all threads within the warp must execute each instruction, with predication used to mask out the effects of instructions that should not be executed [19]. Either way, the performance of the warp suffers. We measured the overhead of divergent control flow by executing a simple kernel containing 32 threads, since there are 32 threads in a warp. If the number of divergent threads is zero, all 32 threads in a warp execute the same sequence of instructions; when it is one, one thread is executing a different path than the other 31 threads. As illustrated in Figure 3, the overhead of divergent control flow increases linearly as the number of divergent threads increases. Programmers should try to avoid excessive use of control flow instructions, or ensure that the value of the controlling condition is the same across the entire warp.

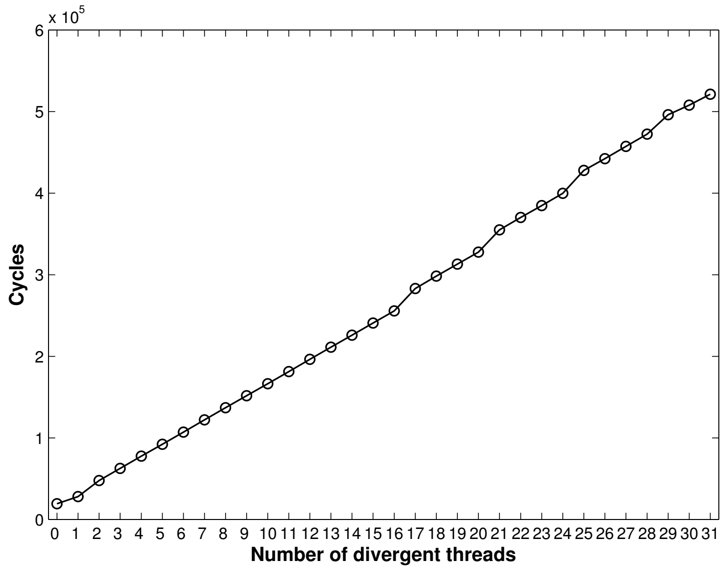


Fig. 3. Performance overhead of divergent threads.

6 Application Performance

In this section, we discuss the performance of our CUDA implementations of a variety of applications, and we describe the algorithmic changes we made to better map the applications onto the GPU hardware. We also compare the programming effort and resulting performance of the CUDA programs executed on the GPU and the single-threaded and multithreaded OpenMP programs executed on the multi-core CPU.

6.1 Structured Grid

Structured grid applications are at the core of many scientific computations. Some notable examples include Lattice Boltzmann hydrodynamics [29] and Cactus [6]. In these applications, the computation is regionally divided into sub-blocks with high spatial locality, and updating an individual data element depends on a number of neighboring elements. A major challenge of these applications comes from dealing with the elements that lie at the boundary between two sub-blocks. In this work, we examine two flavors of structured grid applications: *Speckle Reducing Anisotropic Diffusion* (SRAD) and *HotSpot*.

6.1.1 SRAD

SRAD is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations. It is used to remove locally correlated noise, known as speckles, without destroying important image features [31]. Our CUDA implementation is based on MATLAB code provided by Prof. Scott Acton's group in the U.Va. Department of Electrical Engineering. The inputs to the program are ultrasound images. Each point in the computational grid represents a pixel in the image. Our CUDA implementation of *SRAD* is composed of three kernels. In each grid update step, the first kernel performs a reduction in order to calculate a reference value based on the mean and variance of a user specified image region which defines the speckle. Using this value, the second kernel updates each data element using the values of its cardinal neighbors. This communication pattern is shown in Figure 1a. The third kernel updates each data element of the result grid of the second kernel using the element's north and west neighbors. The application iterates over these three kernels, with more iterations producing an increasingly smooth image.

For this application, CUDA's domain-based programming model is a good match. The whole computational domain can be thought of as a 2D matrix which is indexed using conceptual 2D block and thread indices. Also, since the application involves computation over sets of neighboring points, it can take advantage of the on-chip shared memory. While accessing global memory takes 400 to 600 cycles, accessing shared memory is as fast as accessing a register, assuming that no bank conflicts occur. Thus, for improved performance, the application prefetches data into shared memory before starting the computation. However, the working size of each thread block is limited by the size (16 kB) of the shared memory of each multiprocessor; therefore, multiple thread blocks are needed, and the data set must be divided among them. Because the execution of different thread blocks cannot be efficiently synchronized, in order to deal with the issue of boundary data needed by different thread blocks, for each 16×16 data block we actually declare a 18×18 data block and read the bordering elements into the block.

Figure 4 illustrates the performance of the CUDA version of *SRAD* in comparison to the two CPU versions as a function of the size of the input. For a 2048×2048 input, the CUDA version achieves a $17\times$ speedup over the single-threaded CPU version and a $5\times$ speedup over the four-threaded CPU version. For this application, the development of the CUDA version was more difficult than the development of the OpenMP version. This was chiefly due to the need to explicitly move data and deal with the GPU's heterogeneous memory model in the CUDA version, whereas the OpenMP version was simply an extension of the single-threaded CPU version using compiler pragmas to parallelize the `for` loops.

6.1.2 HotSpot

We use *HotSpot* to demonstrate a technique for reducing synchronization among different thread blocks. *HotSpot* [9] is a widely used tool to estimate processor temperature based on an architectural floorplan and simulated power measurements. The thermal simulation iteratively solves a series of differential equations for block

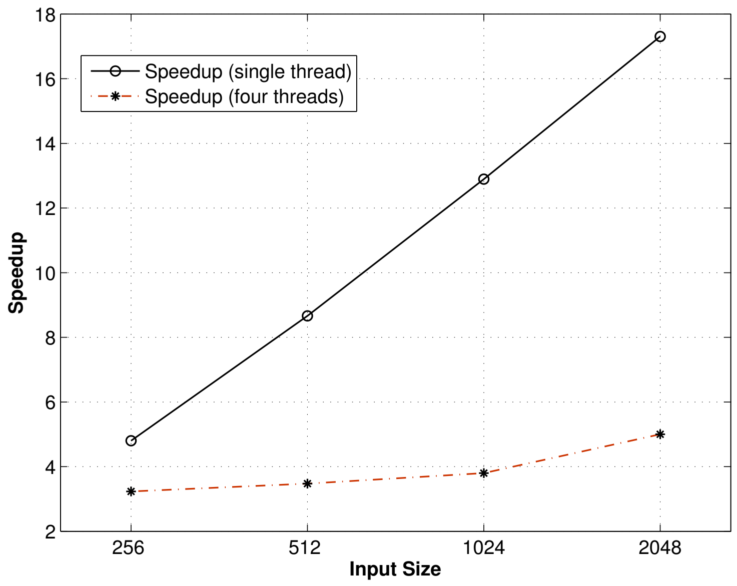


Fig. 4. Speedup of the CUDA version of *SRAD* over the CPU versions. The x-axis represents the size of the x- and y-dimensions of the computation grid.

temperatures. The inputs to the program are power and floorplan files. Each output cell in the computational grid represents the average temperature value of the corresponding area of the chip. Our CUDA implementation re-implements the transient thermal differential equation solver from *HotSpot*. Similarly to *SRAD*, the result of the computation of a cell in the grid is obtained by referencing its neighboring cells, and at the end of each iteration, the data that lies on the boundaries between blocks must be exchanged. Our original solution incurred a substantial global memory overhead which significantly reduced performance.

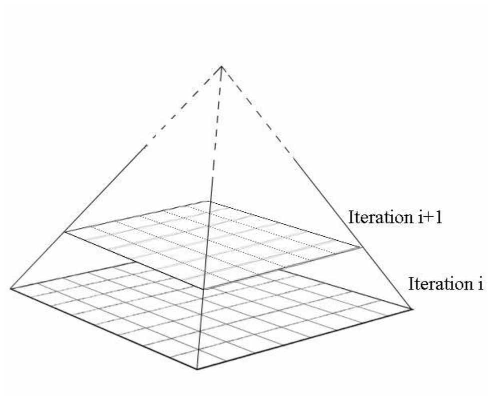


Fig. 5. Pyramid structure used in the CUDA version of *HotSpot*. In this example, starting with an 8×8 block, it takes one iteration to compute the result of the inner 6×6 block.

To ameliorate the effects of this synchronization problem, we want to avoid exchanging data between blocks at the end of each iteration. A novel solution, based on the pyramid structure shown in Figure 5, can improve the performance. In this approach, we assign to each thread block a region that is larger than the final result. If the pyramid base is an $N \times N$ data block, then after one iteration, the inner $(N-2) \times (N-2)$ data block contains valid results. For instance, if we want to compute the result of a grid which is comprised of many 4×4 blocks, we can instead designate 16×16 blocks and load each block into shared memory local to each SM for computation. The data processed in adjacent blocks overlaps so that, after all of the iterations have completed, each of the inner cells in the grid contains a valid result. So in this case, after 6 iterations, we can get the result of 4×4 block by each SM, and then write the result back to the global memory. Using this approach can reduce global memory read/write traffic by a factor of five, compared to the original solution. Again, effective use of the on-chip shared memory is important for an efficient implementation.

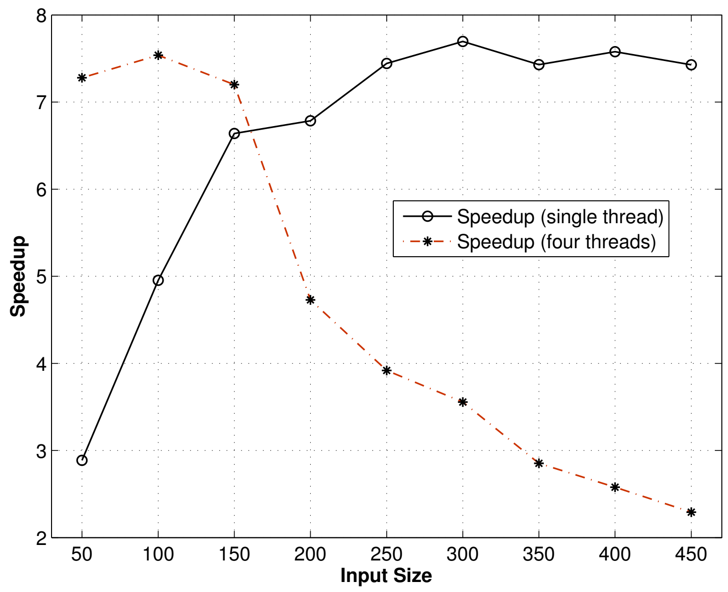


Fig. 6. Speedup of the CUDA version of *HotSpot* over the CPU versions. The x-axis represents the size of the x- and y-dimensions of the computation grid.

Figure 6 shows the speedup of the CUDA version of *HotSpot* compared to the two CPU versions. Using the pyramid data layout, the CUDA version achieves a maximum speedup of approximately $7\times$ over the single-threaded CPU code. However, the speedup of the CUDA version over the four-threaded OpenMP version decreases as the size of the input increases. Although the pyramid architecture can effectively reduce communication between thread blocks, it also increases the amount of computation and memory required. For example, if the result grid is comprised of a number of 4×4 blocks, and each block is expanded to 16×16, then the simulation requires six iterations to converge. For a 256×256 grid, both algorithms need to allocate $64^2 = 4096$ blocks. However, the original algorithm only needs to allocate 4×4 blocks, whereas the pyramid algorithm must allocate 16×16 blocks, increasing the amount of memory required by a factor of 16 and increasing the number of computations by a factor of almost six. Even with this increased memory and computational overhead, the decreased communication overhead of the pyramid structure significantly improves the performance of *HotSpot* compared to both our original CUDA implementation and both CPU implementations.

6.2 Unstructured Grid

In structured grid applications, the connectivity of each data element is implicitly defined by its location in the grid. In unstructured grid applications, however, connectivity of neighboring points must be made explicit; thus, updating a point typically involves first determining all of its neighboring points [1]. Our chosen unstructured grid application is *Back Propagation*, which is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network. The communication structure of this application is shown in Figure 1c. The application is comprised of two phases: the *Forward Phase*, in which the activations are propagated from the input to the output layer, and the *Backward Phase*, in which the error between the observed and requested values in the output layer is propagated backwards to adjust the weights and bias values [28]. In each layer, the processing of all the nodes can be done in parallel.

We implement the data structure required by back propagation in a way that can take advantage of CUDA's domain-based programming model. For each two adjacent layers, we create a weight matrix, with each data element representing the weight of the corresponding column of the input layer and corresponding row in the output layer. In this way, the whole computational domain can be partitioned in a straightforward manner.

To calculate the value of each node in the output layer, we must compute the sum of all of the values of the input nodes multiplied by the corresponding weights connecting to the output node. These multiplications can be done in a massively parallel fashion. There are several methods for performing such reductions using CUDA. In each iteration of the reduction, the number of working threads in a block reduces to half, requiring $\log_2(N)$ iterations for N threads. The communication pattern of a reduction operation is shown in Figure 1d. The reduction process can share data efficiently through the local shared memory; however, for larger inputs, the program must also process the partial sums which are written to global memory by different thread blocks.

Because manual, parallel reductions are complex and error-prone, an abstraction for parallel reductions is helpful. Currently, CUDA programmers can choose to use functions from the CUDPP library, while OpenMP provides a native reduction directive.

Figure 7 shows that for a network containing 65,536 input nodes, our CUDA im-

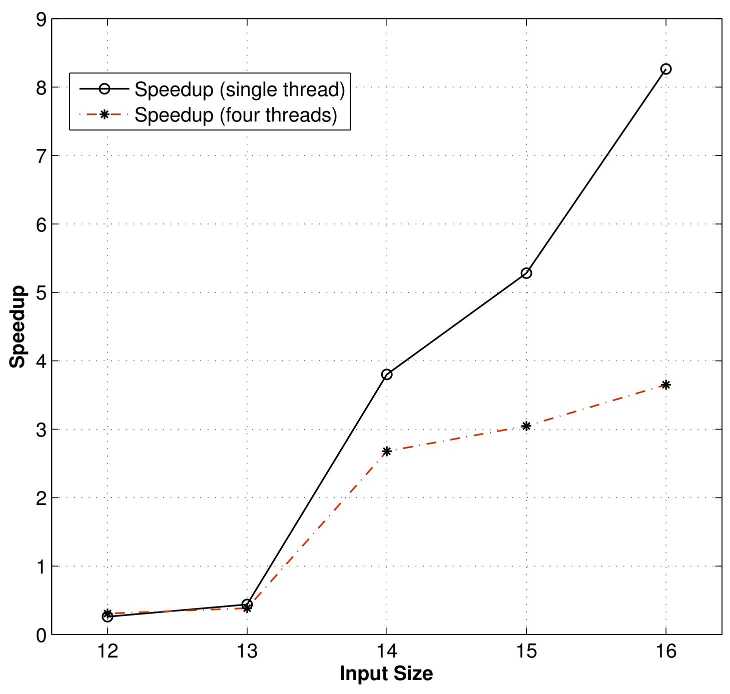


Fig. 7. Speedup of the CUDA version of *Back Propagation* over the CPU versions. The x-axis represents the $\log_2$ of the number of input nodes in the neural network.

plementation achieves more than a $8\times$ speedup over the single-threaded CPU version and approximately a $3.5\times$ speedup over the four-threaded CPU version. However, efficiently implementing multithreaded parallel reductions is non-intuitive for programmers who are accustomed to sequential programming. For such programmers—the vast majority of them—the OpenMP reduction is much simpler since it is a relatively straightforward extension of the sequential CPU reduction using the OpenMP reduction pragma. The CUDA version performs a manual reduction using a combining tree. However, the scan primitives [25] developed for CUDA provide a convenient and efficient way for CUDA programmers to perform parallel reductions through library functions.

6.3 Combinational Logic

The combinational logic dwarf encompasses applications implemented with bit-level logic functions. Applications in this dwarf exhibit massive bit-level parallelism. *DES* is a classic encryption method using bit-wise operations. The *DES* algorithm encrypts and decrypts data in 64-bit blocks using a 64-bit key. It takes groups of 64-bit blocks of plaintext as input and produces groups of 64-bit blocks of ciphertext as output by performing a set of bit-level permutations, substitutions, and iterations. Figure 1b provides an example of a bit-level permutation that might be employed by *DES*. In our CUDA *DES* implementation, we use a straightforward mapping of data to threads in 64-thread blocks. Since different 64-bit data blocks have no interdependencies, they can be assigned to different thread blocks and processed completely in parallel. Inside each thread block, the intermediate data can all reside in shared memory for efficient access.

While developing this application, we found that the entire *DES* algorithm is too large to fit in a single CUDA kernel. When compiling such a large kernel, the compiler runs out of registers to allocate. To reduce the register allocation pressure and allow the program to compile, we divided the large kernel into several smaller ones. However, because the data in shared memory is not persistent across different kernels, dividing the kernel results in the extra overhead of flushing data to global memory in one kernel and reading the the data into shared memory again in the subsequent kernel.

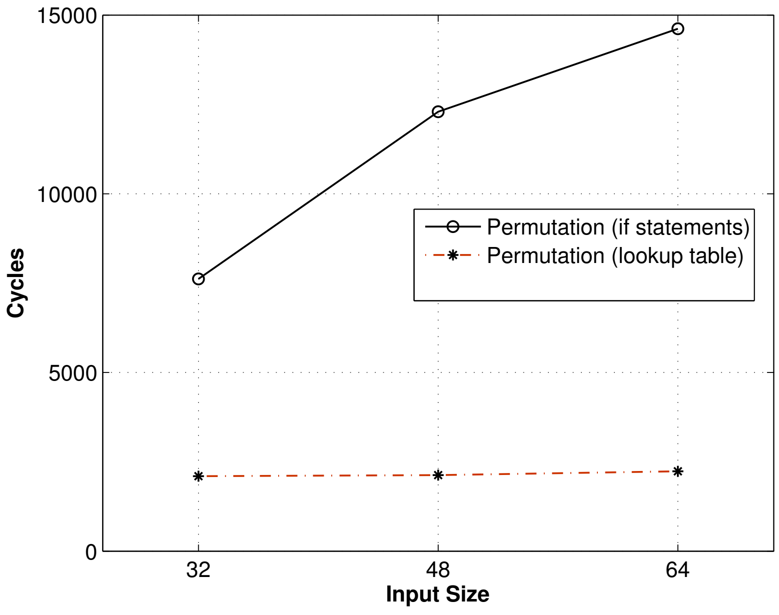


Fig. 8. Overhead of control flow in the CUDA implementations of *DES*. The x-axis represents the number of bits to be permuted.

Some standard sequential programming techniques are inappropriate in the context of massively multithreaded GPUs. For example, although *DES* exhibits a great deal of data-parallelism in many phases of execution, most of the parallelism is in the form of irregular data permutations and shifts. In a sequential program, a programmer might use `if` statements to specify such permutations. However, as we saw earlier in Section 5.3, using such an approach in a CUDA program will significantly degrade performance. Our original CUDA implementation of *DES* exhibited very poor performance because of the dominance of control flow. To improve the performance, we modified the program to use lookup tables to reduce the number of control flow instructions. Figure 8 demonstrates the control flow overhead by showing the number of cycles required to execute two versions of the CUDA program: the original version implemented using `if` statements and the new version implemented using lookup tables. The results demonstrate that using `if` statements can degrade performance by a factor of 2.6 to 5.5 compared to using lookup tables.

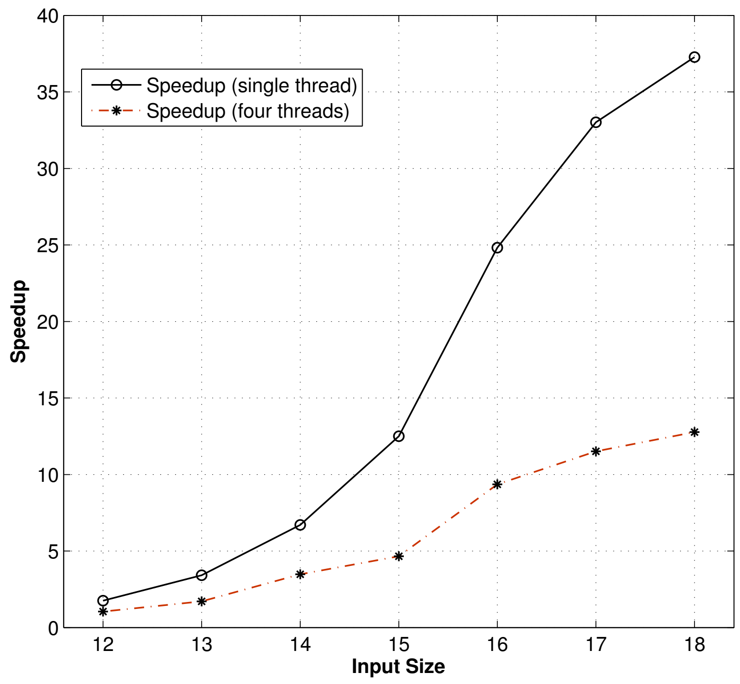


Fig. 9. Speedup of the CUDA version of *DES* over the CPU versions. The x-axis represents the $\log_2$ of the number of bits to be encoded.

Figure 9 shows the speedup of the CUDA version of *DES* over the CPU versions. The improved CUDA version significantly outperforms both of the CPU versions due to the massive parallelism that is exploited among different blocks. For an input size of 2^{18}, the CUDA version achieves a speedup of more than $37\times$ over the single-threaded CPU version and more than $12\times$ over the four-threaded CPU version. In addition, we are currently exploring further optimizations to our CUDA implementation of *DES*, and expect to obtain even more significant performance improvements. For example, some permutation lookup tables are read many times and thus might benefit from being allocated in the GPU's constant memory—which is cached on-chip—for faster access.

6.4 Dynamic Programming

A dynamic programming application solves an optimization problem by storing and reusing the results of its subproblem solutions. *Needleman-Wunsch* is a nonlinear global optimization method for DNA sequence alignments. The potential pairs of sequences are organized in a 2D matrix. In the first step, the algorithm fills the matrix from top left to bottom right, step-by-step. The optimum alignment is the pathway through the array with maximum score, where the *score* is the value of the maximum weighted path ending at that cell. Thus, the value of each data element depends on the values of its northwest-, north- and west-adjacent elements. In the second step, the maximum path is traced backward to deduce the optimal alignment.

Our CUDA implementation only parallelizes the first step, the construction of the matrix. Naively, only a strip of diagonal elements can be processed in parallel, as shown in Figure 1e). By assigning one thread to each data element, our first implementation exhibited poor performance because of the high overhead of accessing global memory.

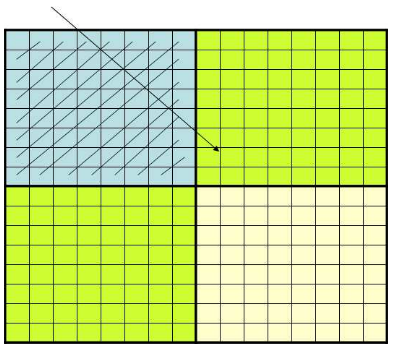


Fig. 10. A graphical representation of the parallelism within the *Needleman-Wunsch* algorithm. Small blocks represent individual data elements and large blocks represent all of the data elements assigned to a single thread block. Blocks at either scale on the same diagonal line can be processed in parallel.

We reconsidered this algorithm and found that each data element is used four times to calculate the values of four different elements, which provides an opportunity to make use of shared memory. In our second implementation, the CUDA program has two levels of parallelism with similar patterns: one among threads within a single block and one among several blocks that can be processed in parallel. Figure 10 demonstrates these two levels of parallelism. Each cell represents a single data element, while the larger blocks represent thread blocks. Data elements on the same diagonal within a thread block can be processed concurrently. Likewise, thread blocks on the same diagonal within the overall matrix (i.e., thread blocks with the same color in the figure) can be processed concurrently.

Figure 11 shows the speedup of this less naïve CUDA version over the CPU implementations. For a 4096×4096 input, the CUDA version can achieve a $2.9\times$ speedup over the single-threaded CPU version. This limited speedup is due to the fact that this application is not inherently computationally intensive. Interestingly, unlike other applications, the four-threaded CPU version outperforms the CUDA version. We still have not performed any extensive performance tuning, such as working to reduce bank conflicts, and our CUDA version is wasteful of resources.

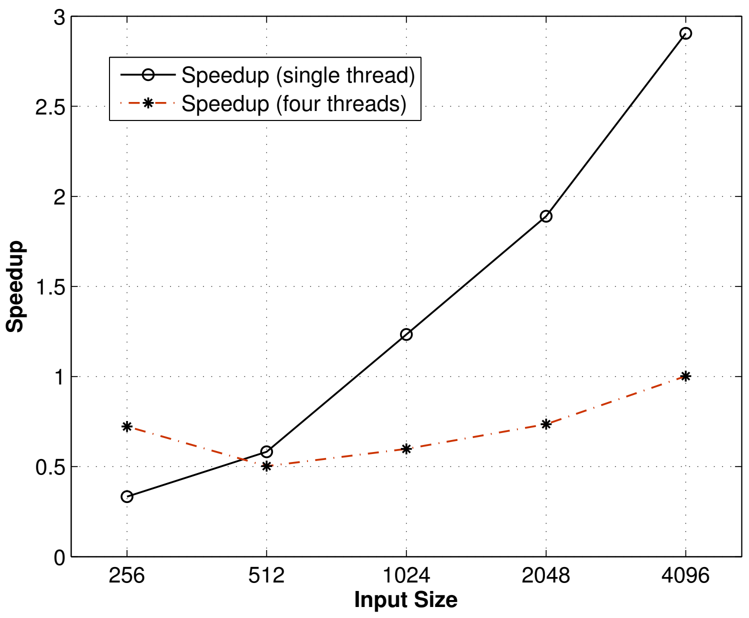


Fig. 11. Speedup of the CUDA version of *Needleman-Wunsch* over the CPU versions. The x-axis represents the size of the x- and y-dimensions of the computation grid.

For example, we universally assign 16 threads to each block, but the number of actual working threads increases from 1 to 16 and then decreases from 16 to 1. Thread blocks of 16 are chosen to maximize SM occupancy: a few large thread blocks fill up the per-block shared memory quickly, because a single thread block of N threads loads a data block of N×N into the PBSM–in fact, the largest possible thread block is only 64 threads. Smaller thread blocks therefore allow greater utilization of each SM, and indeed thread blocks of 16 gave best performance. Considering both the performance and programming cost, OpenMP may prove a better choice for *Needleman-Wunsch*, but the exercise provides us with important insights.

6.5 Data Mining

Data mining algorithms are classified into *clustering, classification*, and *association rule mining*, among others [22]. Many data mining algorithms show a high degree of task parallelism or data parallelism. K*-means* is a clustering algorithm used extensively in data mining and elsewhere, important primarily for its simplicity. Our reference OpenMP implementation of k*-means* is provided by the *Minebench* suite [22]. Our goal with this benchmark is to test the applicability of CUDA to data mining.

In k*-means*, a data object is comprised of several values, called *features*. By dividing a cluster of data objects into k sub-clusters, k*-means* represents all the data objects by the mean values or *centroids* of their respective sub-clusters. In a basic implementation, the initial cluster center for each sub-cluster is randomly chosen or derived from some heuristic. In each iteration, the algorithm associates each data object with its nearest center, based on some chosen distance metric. The new centroids are calculated by taking the mean of all the data objects within each sub-cluster respectively. The algorithm iterates until no data objects move from one sub-cluster to another [22].

In our CUDA implementation, the clusters of data objects are partitioned into thread blocks, with each thread associated with one data object. The task of searching for the nearest centroid to each data object is completely independent, as is the task of computing the new centroid for each cluster. Both of these steps can take advantage of the massive parallelism offered by the GPU.

In addition to implementing the computation of the nearest centroid for each data object in a straightforward manner, we applied several optimizations to the CUDA version to make better use of the GPU:

- The data access pattern in the inner loop of the distance recomputation loop was optimal for the single-threaded version, but meant that each thread in a warp accessed a different cache line in each iteration, leading to poor utilization of the external bandwidth. By reorganizing the main data structure from an array of structures into a structure of arrays, all threads in a warp access adjacent data elements and make efficient use of the available bandwidth.
- We recognized that the array holding the centroids is small enough to fit in constant memory, which is cached. Since the centroid array is also accessed in the inner loop, this optimization eliminates half the accesses to external memory.
- We bound the main array (which is read-only) to a texture to take advantage of the texture cache. The texture cache is useful in reducing external memory bandwidth during the centroid calculation, because the memory accesses are not perfectly aligned on 64 byte boundaries, which would be required to get the best performance when using normal memory accesses.
- We moved a large fraction of the computation of the new centroids to the GPU. Each recomputation is basically a number of parallel reductions. We implemented a parallel per thread block reduction, which reduces the number of additions performed by the CPU from the number of data points to the number of thread blocks.

Figure 12 shows the speedup of the CUDA version of *k-means* over the CPU versions, using a dataset from an intrusion detection problem in the 1999 KDD Cup [11]. For a dataset with 819,200 elements, the CUDA version achieves a 72$\times$ speedup compared to the single-threaded CPU version and a 35$\times$ speedup compared to the four-threaded CPU version. Because the CPU is ultimately responsible for calculating new cluster centroids, the memory transfer overhead is relatively large. In each iteration, we first copy data to the GPU in order to compute the membership of each data object and perform part of the reduction, and then we copy the

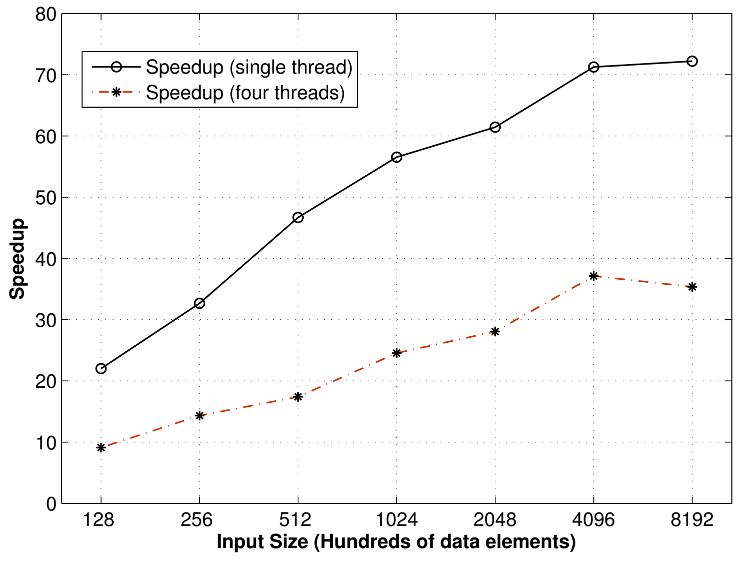


Fig. 12. Speedup of the CUDA version of *k-means* over the CPU versions. The x-axis represents the number of input elements divided by 100.

partial reduction data back to the CPU to complete the reduction. Clearly this application would benefit from further optimization. Also, data mining applications often work on very large datasets which may not fit into a GPU's limited device memory. When computation of a result requires the entire dataset, programmers must be conscious of the overhead of transferring data between the CPU and GPU.

6.6 Summary

In developing these applications, we have gained some insights into GPU programming and the CUDA programming model in particular:

- To get good performance, programmers must understand some basic properties of the underlying *architecture* in addition to the programming model. The most important aspect is the heavy focus on throughput (via deep multithreading) at the expense of individual thread performance, requiring users to expose large quantities of fine-grained parallelism. Other important aspects are the warp concept, the role of SIMD execution within thread blocks (requiring a focus on data rather than task parallelism and—within warps—coherent control flow) and the lack of thread-private local storage (a small number of registers per thread), which requires efficient use of other fast local memories (per-block shared memory, constant, and texture).
- Programmers must find efficient mappings of their applications' data structures to CUDA's domain-based model. For matrix-like data structures, this is straightforward. The *Back Propagation* algorithm also presents a simple mapping of

an unstructured grid, but for more complex and irregular applications—such as applications in the *Graph Traversal* dwarf—-with complex structures built with indirection, it can be non-intuitive. This can be true even when the application exhibits a high degree of data parallelism!

- Programmers must be aware of the data-locality and memory access patterns implied by the implementation of their algorithms in order to take advantage of a GPU memory hierarchy. The most important factor is localizing data access patterns and inter-thread communication within thread blocks and the SM's local data storage units. For frequently accessed, read-only values shared across a warp, cached constant memory is a good choice; per-block shared memory is small but has far lower memory access latency than global memory; and excessive global memory accesses are undesirable due to both latency and bandwidth considerations. Almost all of our implementations use shared memory.
- For large, read-only data structures, binding those data structures to textures to exploit the benefits of texture caches can be beneficial. To benefit from the small texture caches, programmers need to make sure data reuse is local (between threads in a thread block) and that most threads in a warp touch the same cache line for a given load, so as not to thrash the texture cache.
- Programmers should strive to avoid the overhead of global synchronization as much as possible. For example, our *HotSpot* implementation uses novel data structures to reduce global communication to manageable levels.
- Control flow instructions can have a significant impact on performance. Our *DES* implementation illustrates one useful approach, namely using lookup tables rather than control flow instructions.

7 Discussion of the CUDA Programming Model

We have shown the GPU's potential to support interesting applications with diverse performance characteristics. In the course of developing these applications, we made many observations about the CUDA programming model.

Threads in CUDA are scalar, and the kernel is therefore a simple scalar program, without the need to manage vectorization, packing, etc. as is common in some other programming models. In fact, in CUDA data accesses do not need to be contiguous at all, that is to say each thread can access any memory location and still obtain the benefits of SIMD execution as the instruction sequence stays in lockstep within a warp. Although non-contiguous memory references may reduce effective memory bandwidth, this is only a concern for applications that are memory bound. Even in that case, packing is not a prerequisite for working code, but rather an optimization step, which dramatically reduces the software development burden.

The CUDA model is not a purely data-parallel model. For example, programmers can specify task parallelism within a warp, but they must keep in mind that this might cause a severe performance penalty due to thread divergence. Alternatively, task parallelism can be specified between warps within a thread block, but programs are limited to synchronizing all warps via `syncthreads()`—thread blocks can perform different work, but cannot have producer-consumer relationships except across kernel calls.

Barrier synchronization is widely perceived as inefficient, but can actually be more efficient than a large quantity of fine-grained synchronizations. Barriers are mainly detrimental in those cases where the program is forced to synchronize all threads to satisfy the needs of only a few. It is not clear how often this is a concern. Barriers also provide a much simpler abstraction to the programmer. These tradeoffs are poorly understood when the synchronization occurs on chip with hardware synchronization primitives.

Currently, programmers must specify the number of working threads explicitly for a kernel, and threads cannot fork new threads. Often some thread resources are wasted, as in our Needleman-Wunsch implementation. Add to these limitations a lack of support for recursion, and the interface is missing a set of powerful, key abstractions that could hinder their uptake as programmers struggle to restructure their old code as CUDA programs.

Lack of persistent state in the per-block shared memory results in less efficient communication among producer and consumer kernels than might be otherwise possible. The producer kernel has to store the shared memory data into device memory; the data is then read back over the bus by the consumer kernel. This also undercuts the efficiency of global synchronization which involves kernel termination and creation; however, a persistent shared memory contradicts the current programming model, in which thread blocks run to completion and by definition leave no state afterwards. Alternatively, a programmer can choose to use a novel algorithm that involves less communication and global synchronization, such as the pyramid algorithm that we use in *HotSpot*, but this often increases program complexity.

CUDA's performance is hurt by its inability to collect data from a set of producer threads and stream them to a set of consumer threads. Intermediate data has to be stored in device memory before it is consumed by another thread in a new kernel.

8 Conclusions and Future Work

This work compared the performance of CPU and GPU implementations of six naturally data-parallel applications. Our experiments used NVIDIA's C-like CUDA language and compared performance on an NVIDIA GeForce GTX 260 engineering sample with that on an Intel dual dual-core, hyperthreaded Xeon based system with OpenMP. Even though we did not perform extensive performance tuning, the

GPU implementations of these applications obtained impressive speedups and add to the growing body of GPGPU work showing the potential of CUDA for general purpose computing on GPUs. We generally found CUDA convenient to work with; in fact, across all our applications, we were able to offload work to the GPU responsible for 95.8–99.7% of the applications' original, single-threaded execution time excluding disk I/O. We also found CUDA far easier than traditional rendering-based GPGPU approaches using OpenGL or DirectX. CUDA's focus on *available parallelism*, the availability of local (per-block) shared memory, and the kernel-domain abstraction made these applications vastly easier to implement than traditional SPMD/thread-based approaches. In the case of *k-means*, CUDA was probably a bit more difficult than OpenMP, chiefly due to the need to explicitly move data and deal with the GPU's heterogeneous memory model. In *HotSpot*, with the pyramidal implementation, CUDA's "grid-of-blocks" paradigm was a natural fit and probably simplified implementation compared to OpenMP.

The work we presented in this paper only shows a developmental stage of our work. We are continuing to use CUDA to examine the programmability of various applications with different data structures and memory access patterns and hope to be able to draw general lessons about how to best use manycore architectures. With greater architectural convergence of CPUs and GPUs, our goal is to find a parallel programming model that can best aid developers to program in today's high-performance parallel computing environments, including GPUs and multicore CPUs. However, new metrics are needed to better understand the convenience and efficiency with which a particular application maps to different architectures.

Our sample applications mapped nicely onto CUDA and would map easily enough to most parallel programming environments. In all cases, however, managing data placement, communication, and synchronization becomes a nuisance at best—and intractable at worst—with more complex applications. *Higher-level programming APIs are needed!* Ideally, these should promote use of higher-level data structures and programming primitives that implicitly convey information about dependencies and data layout to the compiler or middleware, which can then manage much of the concurrency, data placement, communication, and synchronization. Operations on these data structures then would implicitly convey parallelism while preserving a more natural, quasi-sequential programming style [10]. Ideally, programmers should still be able to "drill down"—to manage the hardware themselves using a lower-level API such as CUDA—albeit at their own risk.

Finally, our results show the benefits of exploiting fine-grained parallelism by maximizing total throughput with a large number of simple, deeply multi-threaded processors, even at the expense of poor single-thread latency.

9 Acknowledgements

This work is supported by NSF grant nos. `IIS-0612049` and `CNS-0615277`, and by a grant from NVIDIA Research. We would like to thank NVIDIA for the opportunity to have access to a pre-production engineering sample of the GTX 260 graphics card. We would also like to thank Michael Garland of NVIDIA Research for pointing out the pyramidal algorithm for the *HotSpot* problem, Karthik Sankaranarayanan for his help with the *HotSpot* grid solver and John Stuppy and S. R. Siddarth for their help with the k*-means* implementation.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick, The landscape of parallel computing research: A view from Berkeley, Tech. Rep. UCB/EECS-2006-183, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley (Dec 2006).

[2] T. Blank, The MasPar MP-1 architecture, Compcon Spring '90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference. (1990) 20–24.

[3] M. Boyer, K. Skadron, W. Weimer, Automated dynamic analysis of CUDA programs, in: Third Workshop on Software Tools for MultiCore Systems, 2008.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, Brook for GPUs: stream computing on graphics hardware, ACM Transactions on Graphics 23 (3) (2004) 777–786.

[5] M. Damrudi, K. J. Aval, Parallel sorting on ILLIAC array processor, in: Proc. of the 7th Conference on 7th WSEAS International Conference on Systems Theory and Scientific Computation, 2007.

[6] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, J. Shalf, The Cactus framework and toolkit: Design and applications, in: Proc. of Vector and Parallel Processing, 2002.

[7] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, D. Manocha, Fast computation of database operations using graphics processors, in: Proc. of the ACM SIGMOD International Conference on Management of Data, 2004.

[8] M. J. Harris, W. V. Baxter, T. Scheuermann, A. Lastra, Simulation of cloud dynamics on graphics hardware, in: Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, 2003.

[9] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, M. R. Stan, Hotspot: A compact thermal modeling methodology for early-stage VLSI design., IEEE Transactions on VLSI Systems 14 (5) (2006) 501–513.

[10] W. W. Hwu, S. Ryoo, S.-Z. Ueng, J. H. Kelm, I. Gelado, S. S. Stone, R. E. Kidd, S. S. Baghsorkhi, A. Mahesri, S. C. Tsao, N. Navarro, S. S. Lumetta, M. I. Frank, S. J. Patel, Implicitly parallel programming models for thousand-core microprocessors, in: Proc. of the 44th ACM/IEEE Design Automation Conference, 2007.

[11] KDD Cup 1999 Data, `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`.

[12] J. Kessenich, D. Baldwin, R. Rost, The OpenGL shading language, `http://www.opengl.org/documentation/glsl`.

[13] J. Krüger, R. Westermann, Linear algebra operators for GPU implementation of numerical algorithms, ACM Transactions on Graphics 22 (3) (2003) 908–916.

[14] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, NVIDIA Tesla: A unified graphics and computing architecture, IEEE Micro 28 (2) (2008) 39–55.

[15] M. D. McCool, Metaprogramming GPUs with Sh, AK Peters, 2004.

[16] Microsoft, DirectX 10, `http://www.gamesforwindows.com/en-US/AboutGFW/Pages/DirectX10.aspx`.

[17] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with CUDA, ACM Queue 6 (2) (2008) 40–53.

[18] NVIDIA, CUDA CUFFT library, `http://developer.download.nvidia.com/compute/cuda/1_1/CUFFT_Library_1.1.pdf`.

[19] NVIDIA, CUDA programming guide 1.1, `http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf`.

[20] NVIDIA, Monte-carlo option pricing, `http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/MonteCarlo/doc/MonteCarlo.pdf`.

[21] L. Nyland, M. Harris, J. Prins, Fast N-Body simulation with CUDA, GPU Gems 3, Addison Wesley, 2007, pp. 677-795.

[22] J. Pisharath, Y. Liu, W. Liao, A. Choudhary, G. Memik, J. Parhi, NU-MineBench 2.0, Tech. Rep. CUCIS-2005-08-01, Department of Electrical and Computer Engineering, Northwestern University (Aug 2005).

[23] C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten, W.-M. W. Hwu, GPU acceleration of cutoff pair potentials for molecular modeling applications, in: Proc. of the 2008 Conference on Computing Frontiers, 2008.

[24] M. Schatz, C. Trapnell, A. Delcher, A. Varshney, High-throughput sequence alignment using Graphics Processing Units, BMC Bioinformatics 8 (1) (2007) 474.

[25] S. Sengupta, M. Harris, Y. Zhang, J. D. Owens, Scan primitives for GPU computing, in: Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, 2007.

[26] S. S. Stratton, J.A. Stone, W. W. Hwu, M-CUDA: An efficient implementation of CUDA kernels on multi-cores, Tech. Rep. IMPACT-08-01, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign (March 2008).

[27] D. Tarditi, S. Puri, J. Oglesby, Accelerator: using data parallelism to program GPUs for general-purpose uses, in: Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, 2006.

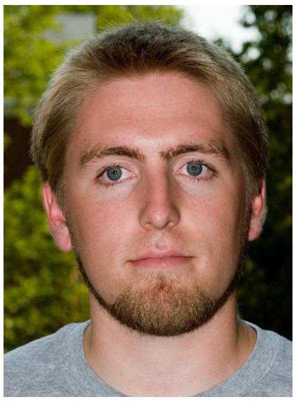
[28] F. M. Thiesing, U. Middelberg, O. Vornberger, Parallel back-propagation for sales prediction on transputer systems, in: Proc. of the World Transputer Congress, 1995.

[29] G. Vahala, J. Yepez, L. Vahala, M. Soe, J. Carter, 3D entropic Lattice Boltzmann simulations of 3D Navier-Stokes turbulence, in: Proc. of the 47th Annual Meeting of the APS Division of Plasma Physics, 2005.

[30] T. Yamanouchi, AES encryption and decryption on the GPU, GPU Gems 3, Addison Wesley, 2007, pp. 785-803.

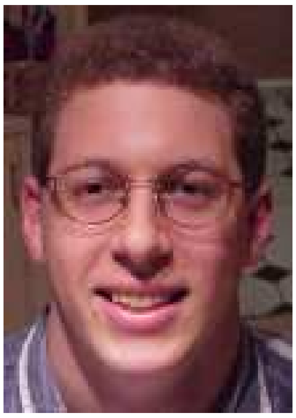
[31] Y. Yu, S. Acton, Speckle reducing anisotropic diffusion, IEEE Transactions on Image Processing 11 (11) (2002) 1260–1270.

Shuai Che is a graduate student in the Department of Computer Science at the University of Virginia. He earned his B.S. degree in Instrument Science and Engineering at Shanghai Jiaotong University in 2004. His research interests include GPGPU programming and computer architectures.



Michael Boyer is a graduate student in the Department of Computer Science at the University of Virginia. He earned his B.S. degree in Computer Engineering in 2006 from Union College. His research interests include adaptive computer architectures and GPGPU programming.

Jiayuan Meng is a graduate student in the Department of Computer Science at the University of Virginia. He earned his M.S. degree in computer science from the University of Virginia in 2007. He also earned his B.S. degree in computer science at Zhejiang University in 2005.



David Tarjan is a graduate student in the Department of Computer Science at the University of Virginia. He earned his Diploma in interdisciplinary sciences from ETH Zurich in 2004 and his M.S. in computer science from the University of Virginia in 2007.

Jeremy Sheaffer is a Research Associate in the Department of Computer Science at the University of Virginia. He earned his Ph.D. in computer science from the University of Virginia in 2007. He has also earned M.S. and B.S. degrees in computer science from Temple University and Millersville University, respectively. His research interests lie in GPU and multicore architectures, their convergence, and their hardware/software interfaces.

Kevin Skadron is an Associate Professor in the Department of Computer Science at the University of Virginia. Skadron's research interests focus on physical design challenges and programming models for multicore/manycore architectures, including graphics and other specialized architectures. Skadron has a Ph.D. in computer science from Princeton University and B.S. and B.A. degrees in Electrical and Computer Engineering and Economics from Rice University. He is co-founder and associate editor-in-chief of IEEE Computer Architecture Letters, and is a senior member of the ACM and IEEE.