A Performance Study of General Purpose Applications on Graphics Processors

Shuai Che sc5nf@cs.virginia.edu
Jiayuan Meng jm6dg@cs.virginia.edu
Jeremy W. Sheaffer jws9c@cs.virginia.edu
Kevin Skadron skadron@cs.virginia.edu
The University of Virginia, Department of Computer Science

Abstract

*Graphic processors (GPUs), with many light-weight data-parallel cores, can provide substantial parallel computational power to accelerate general purpose applications. To best utilize the GPU's parallel computing resources, it is crucial to understand how GPU architectures and programming models can be applied to different categories of traditionally CPU applications. In this paper we examine several common, computationally demanding applications—*Traffic Simulation*, *Thermal Simulation*, and* K-Means*—whose performance may benefit from graphics hardware's parallel computing capabilities. We show that all of our applications can be accelerated using the GPU, demonstrating as high as* $40\times$ *speedup when compared with a CPU implementation. We also examine the performance characteristics of our applications, presenting advantages and inefficiencies of the programming model and desirable features to more easily and completely support a larger body of applications.*

1 Introduction

Traditional single-core microprocessors are having difficulty achieving higher clock frequencies. The limitations imposed by deep pipelining, transistor scaling, and power and thermal constraints have forced CPU vendors to find other ways to meet high performance computing needs.

One solution is that of multi-core architectures, which integrate multiple cores onto a single chip. Examples are off-the-shelf Intel *Duo-core* and *Quad-core* products, the Sony/Toshiba/IBM alliance's *Cell Broadband Engine* [9], MIT *RAW* [24], and Sun's *Niagra* [13].

Another powerful solution is the GPU. GPUs represent highly specialized architectures designed for graphics rendering, their development driven by the computer gaming industry. Recently, there has been a trend to accelerate computationally intensive applications, including scientific applications, on graphic processors. This trend introduced the new term *GPGPU* or *General-Purpose computation on the GPU*.

The GPU has several key advantages over CPU architectures for highly parallel, compute intensive workloads, including higher memory bandwidth, significantly higher floating-point throughput, and thousands of hardware thread contexts with hundreds of parallel compute pipelines executing programs in a SIMD fashion. The GPU can be an attractive alternative to CPU clusters in high performance computing environments.

The term GPGPU causes some confusion nowadays, with its implication of structuring a general-purpose application to make it amenable to graphics rendering APIs (OpenGL or DirectX) with no additional hardware support. NVIDIA has introduced a new data-parallel, C-language programming API called *CUDA* (for *Compute Unified Device Architecture*) that bypasses the rendering interface and avoids the difficulties of classic GPGPU. Parallel computations are instead expressed as general-purpose, C-language kernels operating in parallel over all the points in a domain.

To best utilize GPUs' powerful computing resources, it is necessary to examine to what extent traditionally CPU domain problems can be mapped to GPU architectures, and what kinds of features the GPU parallel programming model should support. 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 and include *Structured Grid*, *N-Body problem* and *Dynamic Programming* dwarves.

Inspired by this work, and noting an apparent architectural convergence of CPUs and GPUs, our goal is to find a good programming model that can provide a rich and useful feature set for today's parallel computing needs. This paper introduces our work in progress, still developmental. We port some widely used applications to CUDA and analyze their performance. Our applications—*traffic simulation*, *thermal simulation* (with HotSpot [8]), and k*-means*—have a great deal of data-parallelism and benefit from the GPU's parallel computing resources. We compare results on an NVIDIA Geforce 8800 GTX against and an Intel Pentium 4 both under Windows XP. The CPU code is compiled with Visual Studio 2005 with the SSE2 (Streaming SIMD Extensions 2) option and O2 optimization turned on.

All of our applications show satisfactory speedups. The maximum observed speedups are about $40\times$ for traffic simulation, $6.5\times$ for HotSpot and $8\times$ for k-means. Also, to better utilize the GPU under CUDA, we make novel, architecturally-aware changes to the algorithms and data structures, such as the introduction of a pyramid data structure in the HotSpot benchmark to avoid the need for excess synchronization between thread blocks.

In addition to showing speedups using CUDA, another contribution of our work is that we present some advantages and inefficiencies of the CUDA model. For example, we find that a global synchronization mechanism is necessary in order to avoid the need to separate a logical kernel into disjoint parts. Double buffering or a bulk load/store mechanism would be useful as an additional mechanism to help reduce memory access latency.

2 Related Work

A lot of recent work has focused on GPGPU. A framework for solving linear algebra problems on graphics processors is presented by Krüger et al [14]. Harris et al. present a cloud dynamics simulation using partial differential equations [7]. Some important database operations are implemented on the GPU by efficiently using pixel engines [6]. Bioinformatics algorithms such as sequence alignment [15] have also been successfully implemented on GPUs.

OpenGL [12] and *DirectX* [20] are the two major API interfaces for programming graphics applications, but they are not convenient for developing general purpose applications on GPUs. With increased programmability in recent GPU generations, languages such as *Brook* [2], *Sh* [18], and *Cg* [17] were developed to provide simpler programming environments for GPU developers.

CUDA is a new language and development environment from NVIDIA, allowing execution of applications with thousands of data-parallel threads on NVIDIA G80 class GPUs. AMD recently introduced their own GPU direct compute API, *Close To the Metal* (CTM). CTM also provides a way around graphics APIs, with a driver and programming interface designed specifically for compute.

3 GPU Architectures

3.1 The NVIDIA Geforce 8800 GTX GPU

GPUs have developed over the course of the last decade as highly specialized processors for the acceleration of raster graphics. GPUs have been developed hand-in-hand with graphics APIs, thus each stage in the traditional GPU pipeline corresponds very closely with a corresponding stage in the conceptual OpenGL pipeline. Recent GPUs have added programmability to the vertex and fragment shading stages of the pipeline. Instead of separate processing units for different shaders, the *Unified Shader Model* (USM) was introduced with DirectX 10, allowing for better utilization of GPU processing resources. ATI's *Xenos* chip for the Xbox 360, AMD R600, and NVIDIA's Geforce 8800 [16] all implement USM. The computing resources needed by shaders varies greatly among different applications. The unified design can overcome this issue by balancing loads between geometry, vertex, and fragment shader functionality, yielding maximum utilization of computing resources.

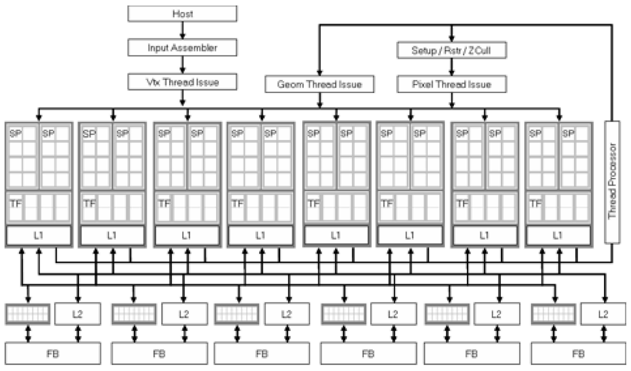

Figure 1. The Geforce 8800 GTX Architecture, with 16 multiprocessors, each with 8 streaming processors.

A diagram of the NVIDIA Geforce 8800 GTX architecture is shown in Figure 1. Its design is a radical departure from traditional mainstream GPU design. The 8800 GTX is comprised of 16 multiprocessors. Each multiprocessor has 8 streaming processors (SPs) for a total of 128 SPs. Each group of 8 SPs shares one L1 data cache. An SP contains a scalar ALU and can perform floating point operations. Instructions are executed in a SIMD fashion. The 8800 GTX has 768 MB of graphics memory, with a peak observed performance of 330 GFLOPS and 86 GB/s peak memory bandwidth [3]. This specialized architecture can sufficiently meet the needs of many massively data-parallel computations.

3.2 CUDA

Direct3D and OpenGL include many functions provided to render complex graphics scenes and abstract away the communication between applications and graphics drivers.

Unfortunately, for non-graphics applications, these APIs introduce many hurdles to the general purpose application programmer, including the inherent problems born of the need to caress general purpose problems into looking like rendering problems and the functional problem imposed by lack of scatter functionality.

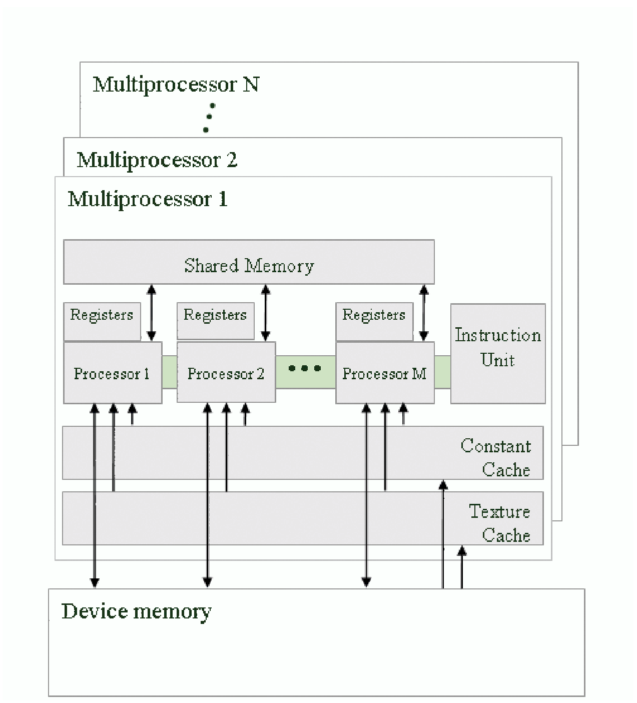

Figure 2. CUDA's shared memory architecture. Threads in one block can share data using on-chip shared memory [21].

CUDA provides a cleaner interface than traditional GPGPU to program the GPU for general purpose applications, exposing important architectural features of G80 and eliminating the need to map computations to a graphics API. In CUDA, the GPU is a device that can execute multiple concurrent threads. Threads are executed in SIMD *thread blocks*, which facilitate efficient data sharing. Blocks and threads are indexed by block and thread ids, and the GPU is conceptually viewed as a set of multiprocessors. One or more thread blocks is dispatched to each processor, and executed using time sharing [21]. Blocks are further organized into grids, with blocks within a grid being run to completion and not currently guaranteed to execute in any particular order; hence, blocks should not communicate except by allowing an entire grid to complete, returning to the host, and subsequently starting a new kernel. The 3D capability of blocks, coupled with the 2D capability of grids, can potentially allow expression of 5D domains. As is illustrated in Figure 2, each multiprocessor has a set of register files, a shared memory data cache, and a read-only (constant) cache. Threads within one block can share data through shared memory, allowing very high access speeds. Constant memory is optimized for fast read-only access to a region of global, read/write device memory [21].

Below, we give a simple example of CUDA program which assigns the values in an $n \times n$ matrix B to a matrix A and an example of the same operation in C. If n is 128 and each block is 16 by 16, CUDA will divide the whole matrix into 64 blocks, and each assignment becomes the responsibility of a single thread. The CUDA program model abstracts away the loop needed in the CPU implementation, a powerful abstraction very useful in solving more complicated problems.

- Matrix copy in CUDA

```
index = n * BLOCK_SIZE * blockIdx.y +
        BLOCK_SIZE * blockIdx.x +
        n * threadIdx.y + threadIdx.x;

A[index] = B[index];
```

- Matrix copy on the CPU in C

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    A[i][j] = B[i][j];
```

3.3 Parallel Execution Model

The two dominant parallel architectural models are the *message passing* and *shared memory* models [26]. CUDA's shared memory model allows threads within the same block to share data using high-speed on-chip shared memory. Threads from different blocks can share data via global memory. CUDA adopts a SIMD data-parallel model in which one instruction is executed multiple times in parallel on different data elements.

In CUDA, the CPU is viewed as a control processor that is responsible for parameter setup, data initialization, and execution of the serial portions of the program. The GPU is viewed as a co-processor whose job is to accelerate data-parallel computations.

Though CUDA provides a powerful API with a swift learning curve, there are several challenges for CUDA developers. First, before a kernel can be launched on the graphics processor, data must be copied from host memory to GPU device memory. Copy overhead is proportional to the amount of data to be copied. Also CUDA has an explicitly controlled memory hierarchy that is critical to performance and requires fine-grained tuning. In addition, barrier functionality can only be imposed on threads within the same block. Synchronizing with threads in different blocks can be achieved by terminating a function call, otherwise read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) hazards become a concern [21].

4 Experiment Setup and Methodology

We chose representative commercial products from both of the GPU and CPU markets: an NIVIDIA Geforce 8800 GTX (128 stream processors clocked at 675 MHz with 768 MB of graphics memory and a 16 kB parallel data cache per multiprocessor block) with NVIDIA driver version 6.14.11.6201 and CUDA 1.0, and a traditional single-core Intel Pentium 4 CPU (2.4 GHz with 512 MB main memory, 512 kB L2 and 8 kB L1). We developed our GPU code using NVIDIA's CUDA API. The CPU code is compiled under Visual Studio with O2 and SSE2. The GPU implementation results are compared against the Intel CPU results under Windows. Given the same input data set, the speedup is calculated by taking the wall-clock time required by applications on the CPU divided by the time required by the GPU. Times are measured after initial setup (e.g. after file I/O) but do include PCI-E bus transfer times.

5 Experimental Results

5.1 Memory Overhead

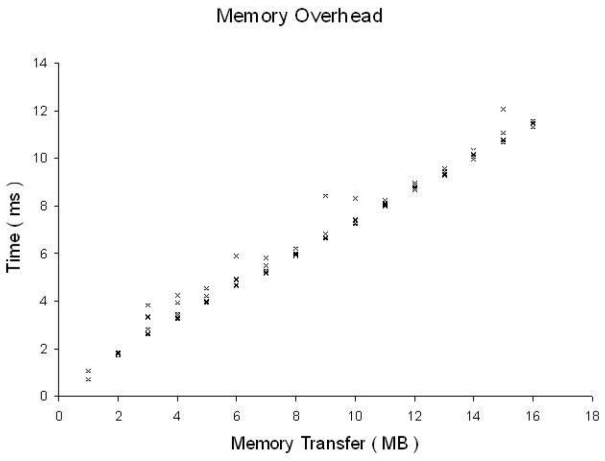

Figure 3. Memory transfer overhead between our CPU and GPU. As the size of data increases, the time overhead for a data transfer increases linearly.

Classically, in a PC the graphics card is connected via the AGP or PCI Express bus to a North Bridge chip which also connects to the CPU and main memory, so the data transfer rate of bus and memory bandwidth are very crucial to the performance of applications. We measured the memory overhead by varying the amount of data transferred between our CPU and GPU. We found that the time necessary to transfer date increases linearly with the amount of data, as illustrated in Figure 3, thus the constant overhead of each individual transfer is immeasurable so long as transfers are large or infrequent.

5.2 Traffic Simulation

A great deal of civil engineering work exists on simulating automotive traffic. Aimed at improving traffic control to reduce congestion and accidents, these simulations tend to require a lot of compute power. There has been no previous work that uses graphics processors to accelerate traffic simulation. Our work is based on a part of the *MITSIM* model [27], which simulates traffic networks.

5.2.1 Algorithm Overview

Our work re-implements part of MITSIM's lane-changing model—only a portion of the MITSIM infrastructure, which provides a much broader and more comprehensive traffic simulation model than our benchmark—with cars running in 4 lanes [23]. The lanes are implemented as a 4-wide 2-D array with each cell representing a position. The car structure carries the information about the car's speed, position, the lane the car resides in and so forth. All of this information is dynamically updated during the lifetime of the simulation. Cars can change lanes only if the *behind-diagonal*, *next-to*, and *forward-diagonal* cells are clear of cars. A car can accelerate when there are 2 blank positions in front of it.

This model is straightforward to implement on CUDA's shared memory model and benefits from using the GPU's large number of concurrent threads. In each iteration, the determination of the next action of each car (e.g. to change lanes, accelerate, brake, etc.) is dependent only on the locations of the cars withing a small, local neighborhood.

5.2.2 Traffic Simulation Results and Analysis

We used 256 threads per block. The number of blocks are determined by the number of cars the user specifies on the command line. We assign each car a unique id and map that to a thread. The number of cars (number of threads) is varied in the experiment. As is evident in Figure 5, the G80 can achieve speedups of about $40\times$ with respect to the Pentium 4. When the number of cars is 800 or less, our CPU implementations outperforms the GPU simulator; however, as the number of cars increases, the GPU implementation gradually surpasses the CPU, which demonstrates that GPU is more appropriate for data-parallel operation with many threads given our configuration. Applications like the traffic simulation present an especially good fit for GPUs and the CUDA programming model as, except for synchronizing the blocks for each iteration, there are no data dependencies between threads within one block or between the blocks.

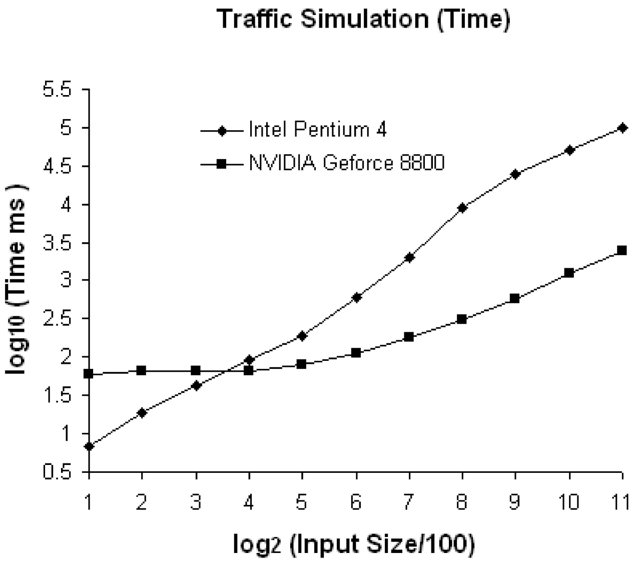

Figure 4. Execution time of the traffic simulation. The input size is the number of cars. The GPU implementation outperforms the CPU implementation on simulations with about 1600 or more cars.

5.3 Thermal Simulation

HotSpot [8] is a widely used tool to estimate processor temperature based on block layout and simulated performance measurement in architectural simulation. It is a representative of the *structured grid* dwarf [1], in which the computation is regionally divided into sub-blocks with high spatial locality. Structured grid applications are at the core of many scientific computations. Other notable examples include Lattice Boltzmann hydrodynamics [25] and Cactus [4]. A major challenge of these applications comes from dealing with the boundary data between sub-blocks.

5.3.1 HotSpot Algorithm Overview

In HotSpot, a silicon die is partitioned into functional blocks based on the floorplan of the microprocessor, with a thermal RC network connecting the various blocks [8]. The thermal simulation iteratively solves a set of differential

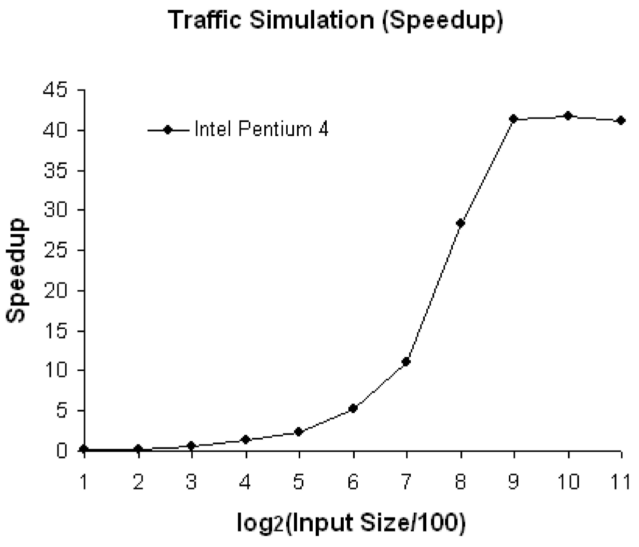

Figure 5. Speedup of the traffic simulation. The maximum speedup is more than $40\times$.

equations for block temperatures. Our GPU implementation re-implements the transient thermal differential equation solver from HotSpot. Given the power density and temperature at time t and the simulation differential time Δt, the transient solver can compute the processor temperature at time $t + \Delta t$.

In HotSpot, each cell in the grid represents the average power density and temperature of the corresponding area on the chip. The grid is divided into sub-blocks along the x- and y-axes. Therefore at the end of each iteration, the data that lies on the boundaries between blocks should be exchanged. In CUDA, data communication among blocks is not supported. The threads in one block can only read data written by another block from global memory. This solution involves heavy global memory read/write overhead, which significantly reduces the performance of the program.

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 a pyramid structure (see Figure 6), can achieve satisfactory speedups. Using this method, we assign each processing thread block a much bigger region than the final result. If the pyramid base is an $N \times N$ data block, after one iteration the inner $(N - 2) \times (N - 2)$ data block has precise results. For instance, if we want to compute the result of a grid which is comprised of many 4×4 blocks, at the beginning we can designate 16×16 as the size of each block in CUDA and load each block to shared memory for computation. The data processed in adjacent blocks should overlap in order that after 6 iterations, each of the cells in the grid has a correct result.

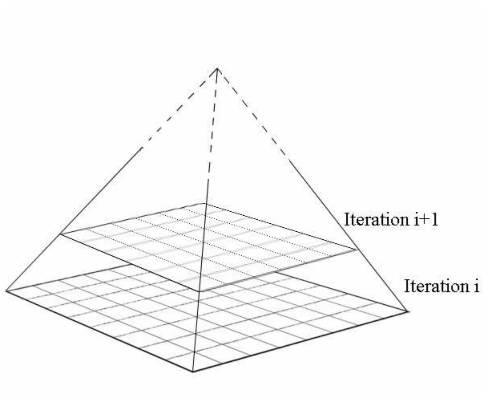

Figure 6. Starting from an 8×8 block, it takes 1 iteration to compute the result of the inner 6×6 block

5.3.2 HotSpot Simulation Results

Our experiments show that our pyramid architecture can achieve a maximum speedup of more than $6.5\times$ when compared with the Pentium 4 CPU. This result is much better than our original naïve implementation, which was based more closely on the CPU implementation and did not use the pyramid structure. Obviously, efficient use of shared memory is important for an efficient implementation.

However, as demonstrated in Figure 8, the thermal simulation does not show the high speedups seen in the traffic simulation. Additionally the HotSpot speedup curve saturates early. We believe that this phenomenon is due to the fact that, though the pyramid architecture can efficiently reduce communications between thread blocks, it simultaneously creates more tasks for the GPU to perform. For example, if the result grid is comprised of a number of 4×4 blocks, for each block, starting from a 16×16 block, the simulations requires 6 iterations to converge. For a grid of size 256×256, the GPU must allocate 64×64 blocks to run in parallel on 8 multiprocessors. A large number of blocks makes the speedup saturate early. However, using the pyramid architecture, the performance improves significantly when compared with our original implementation.

In developing this benchmark, we found that to develop an application with good performance, CUDA's parallel programming model requires that programmers have familiarity with the G80 architecture, as data must be explicitly managed in the memory hierarchy. Additionally, programmers also need to be careful when dealing with shared memory, because inappropriate scheduling can cause bank conflicts which may force threads to access memory in serialized order [21].

Another problem with CUDA is that there is no mechanism to enforce all the threads of different blocks to synchronize at a certain point. Inside a GPU function call, using a `for` loop is dangerous when there are data dependencies between the two adjacent loop iterations because the CUDA parallel model is a *for-all* model.

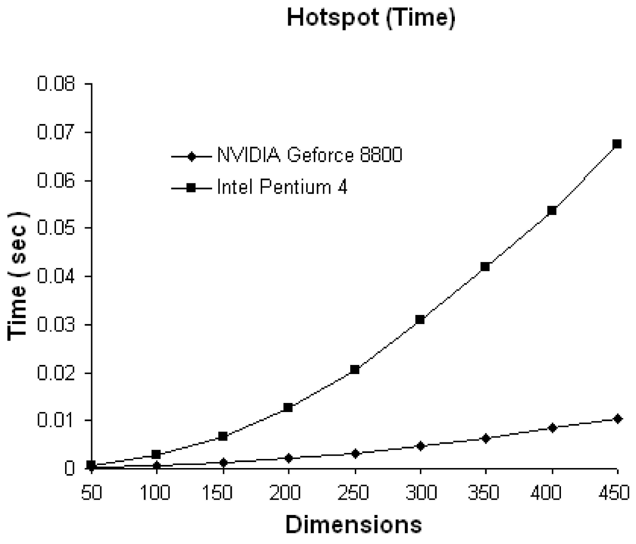

Figure 7. Execution Time of HotSpot. The x-axis represents the dimensions of the computation grid. a dimension of 50 means a 50×50 grid.

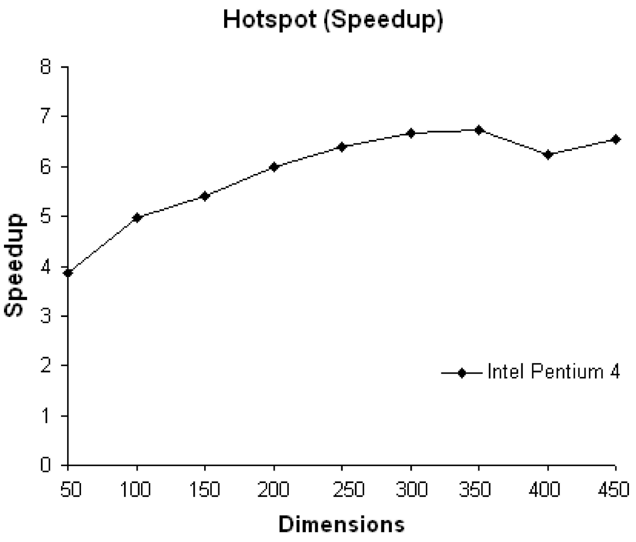

Figure 8. Speedup of HotSpot. The maximum speedup is more than $6.5\times$ for HotSpot

5.4 K-Means

K-means is a clustering algorithm used extensively in data-mining and elsewhere, important primarily for its simplicity. 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. Researchers at Northwestern University developed *Minebench* using OpenMP [22]. Our goal with this benchmark is to test the applicability of the GPU to data-mining using k-means.

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.

5.4.1 K-Means Algorithm Overview

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 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].

The data objects can be organized in a 2-D array with each row representing one data object and each column representing the value of the corresponding feature. Some implementations use K-d trees [10] to accelerate the execution time. Advantages of k-means over other clustering algorithms include its fast convergence and ease of implementation.

In our CUDA implementation, the clusters of data objects are partitioned into thread blocks, each thread associated with one data object. The task of searching for the nearest centroid to each data object is independent. Our program uses the Euclidean distance as its distance metric. After all the threads find their nearest centroid, a reduction step will produce the new centroid for each sub-cluster. An error function signals convergence and terminates the iteration.

We make the GPU is responsible for calculating the distances of each object to the k clusters in parallel, while the CPU takes over the reduction step. CUDA does provide a handful of atomic operations that could be used to implement this reduction; however, these operations are integer only and are restricted to the Geforce 8600, so we choose to ship this latter step back to the CPU. The speedup measurement is performed on the distance calculation only, not on the serial reduction! The algorithm can be further parallelized by calculating the partial sum for the threads in each block and then adding the partial sums together for each sub-cluster to calculate a new centroid.

5.4.2 K-Means Results

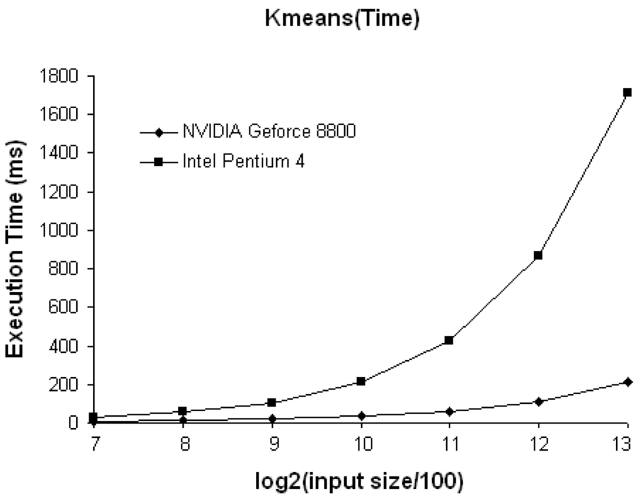

Figure 9. Execution time of k-means (reduction part not included). The input size is the number of the data objects. The time showed is for one iteration

In Figure 9, we compare the running times of k-means separately on a Geforce 8800 and a Pentium 4 on a dataset from an intrusion detection problem in the 1999 KDD Cup [11]. Figure 10 shows that for a dataset with 1,638,400 elements, the CUDA program on the G80 can achieve about $8\times$ speedup as compared with the Pentium 4. Because currently the CPU is responsible for calculating new cluster centroids, the memory overhead is relatively large. In each iteration we have to copy the data to the GPU to compute the membership of each data object, then copy back to the CPU for the reduction.

6 Discussion: CUDA Programming Model

In the course of developing these applications for CUDA, we have developed a short "wish-list" of features that could help improve performance. Firstly, we believe that CUDA would benefit from facilities for asynchronous bulk data movement from host memory to device memory. Currently, CUDA allows only synchronous *direct memory access* (DMA) transfers between the GPU and CPU. As a result, a kernel cannot begin execution until the DMA download completes, which introduces overhead.

Secondly, a pre-fetching mechanism can serve as a complementary memory access latency hiding tool in addition to hardware thread context switching. Especially when all

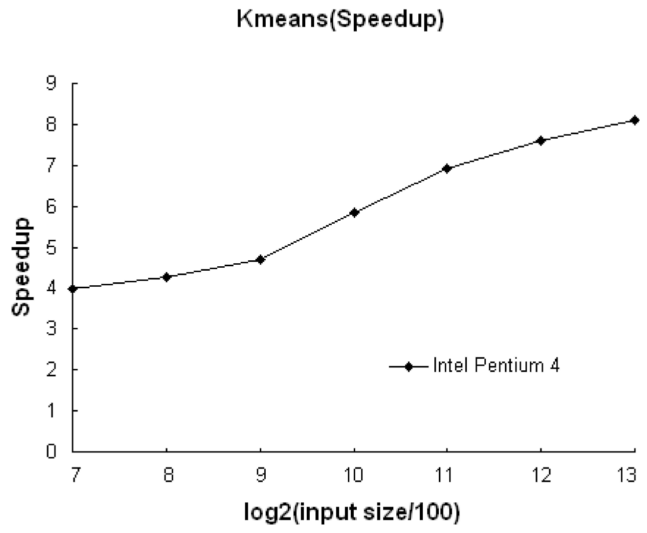

Figure 10. Speedup of k-means (reduction part not included). The maximum speedup is about $8\times$.

threads are accessing device memory intensively, switching threads is unable to sufficiently overlap computation with memory access because all threads are memory bound. Correlated threads within the same kernel are executed simultaneously, so it is probable that many threads will reach a memory intensive phase simultaneously.

It may also be beneficial to provide a faster local store. This could be useful in situations where data has to be loaded from device memory and reused many times. Options include caching, scratchpads, and bulk DMAs. Caching supports existing programming models and toolchains, but caching in shared memory raises the traditional problems of coherence and memory-ordering guarantees. The other two options also have drawbacks. Scratchpads require explicit compiler management and DMA may require double buffering, using extra memory.

Lack of persistent state in the shared memory data cache results in less efficient communication among producer and consumer kernels than might be 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 with the current programming model, in which threads 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 leads to—often undesirable—tradeoffs in program complexity that would be better left unconsidered.

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. By allowing fine grained synchronization among producer and consumer threads, programs would be able to consume data at a higher rate, earlier freeing its storage for reuse. Since thread blocks must run to completion, and only one shared memory can be allocated to a thread block, it is unwieldy to support high-quality producer/consumer steaming given the current implementation of CUDA's shared memory model.

Finally, even though we did not perform extensive performance tuning (tiling, managing bank conflicts, etc.), we found that CUDA programming required extensive knowledge of the underlying hardware architecture. This is actually true of almost any parallel programming environment today. This makes parallel programming an excellent vehicle for teaching computer architecture, and in fact we used CUDA to good effect in a senior-level computer-architecture course. We worry, however, that the associated learning curve will deter adoption of parallel programming, especially for more complex algorithms and data structures that do not map as easily to a manycore environment.

7 Conclusions and Future Work

This work compared the performance of CPU and GPU implementations of three, naturally data-parallel applications: *traffic simulation*, *thermal simulation*, and *k-means*. Our experiments used NVIDIA's C-based *CUDA* interface and compared performance on an NVIDIA Geforce 8800 GTX with that on an Intel Pentium 4 CPU. 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 GPUs for general-purpose computing. Furthermore, the CUDA interface made programming these applications vastly easier than traditional rendering-based GPGPU approaches (in particular, we have prior experience with structured grids [5]). We also believe that the availability of shared memory and the domain abstraction provided by CUDA made these applications vastly easier to implement than traditional SPMD/thread-based approaches. In the case of k-means and the traffic simulation, 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 the case of HotSpot with the pyramidal implementation, CUDA's "grid-of-blocks" paradigm probably simplified implementation compared to OpenMP.

The work we presented in this paper only shows a developmental stage of our work. We plan to extend our

GPGPU work by comparing with more recent commodity configurations such as Intel dual-core processors and examining the programmability of more complex applications with various kinds of data structures and memory access patterns. In addition, in order to better understand the pros and cons of GPU architectures for general-purpose parallel programming, new metrics are needed for characterizing the applications. 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 multi-core CPUs.

Our sample applications mapped nicely into CUDA and would map easily enough to most manycore 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 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 [19]. 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.

The common view that effective parallel programming requires low-level, explicit management of concurrency and hardware resources only applies to the most expert programmers! We contend that high-level APIs and automatic parallelism will boost productivity *and*, for many programmers, will yield *better* speedups. Higher-level APIs that abstract away hardware details also simplify portability among different parallel platforms.

8 Acknowledgements

This work is supported by NSF grant nos. CNS-0306404 and CNS-0615277, and by a grant from NVIDIA Research. We would like to thank Michael Garland of NVIDIA Research for pointing out the pyramidal algorithm for the HotSpot problem, Kevin Stammetti for his help with the traffic simulation algorithm and implementation, 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, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM Press.

[3] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Gpuqp: query co-processing using graphics processors. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1061–1063, New York, NY, USA, 2007. ACM Press.

[4] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. The cactus framework and toolkit: Design and applications. In *VECPAR*, pages 197–227, 2002.

[5] N. Goodnight, G. Lewin, D. Luebke, and K. Skadron. A multigrid solver for boundary value problems using graphics h ardware. Technical report, University of Virginia, January 2003. University of Virginia Technical Report CS-2003-03.

[6] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 206, New York, NY, USA, 2005. ACM Press.

[7] M. J. Harris, W. V. Baxter, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[8] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. R. Stan. Hotspot: A compact thermal modeling methodology for early-stage vlsi design. *IEEE Trans. VLSI Syst.*, 14(5):501–513, 2006.

[9] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.,49(4/5):589C604*, 2005.

[10] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu. An efficient k-means clustering algorithm: analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, 2002.

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

[12] J. Kessenich, D. Baldwin, and R. Rost. The opengl shading language. Technical report, The OpenGL Architectural Review Board, 2006. http://www.opengl.org/documentation/-glsl.

[13] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 2005.

[14] J. Krüger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 234, New York, NY, USA, 2005. ACM Press.

[15] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig. Streaming algorithms for biological sequence alignment on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 18(9):1270–1281, 2007.

[16] D. Luebke and G. Humphreys. How gpus work. *IEEE Computer*, 40(2):96–100, 2007.

[17] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

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

[19] W. mei 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, and S. J. Patel. Implicitly parallel programming models for thousand-core microprocessors. In *DAC*, pages 754–759. IEEE, 2007.

[20] Microsoft Corp. DirectX. http://www.gamesforwindows.-com/en-US/AboutGFW/Pages/DirectX10.aspx.

[21] NVIDIA. Cuda programming guide 1.0, 2007. http://developer.nvidia.com/object/cuda.html.

[22] J. Pisharath, Y. Liu, W. Liao, A. Choudhary, G. Memik, and J. Parhi. Nu-minebench 2.0. Technical report, Northwestern University Department of Electrical and Computer Engineering, 2005. http://cucis.ece.northwestern.edu/techreports/pdf/CUCIS-2004-08-001.pdf.

[23] K. Stammetti. Testing the feasibility of running a computationally intensive real-time traffic simulation on a multicore programming graphics processor, May 2007. U.Va. SEAS Senior Thesis.

[24] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 2, Washington, DC, USA, 2004. IEEE Computer Society.

[25] G. Vahala, J. Yepez, L. Vahala, M. Soe, and J. Carter. 3d entropic lattice boltzmann simulations of 3d navier-stokes turbulence. In *Proceedings of the 47th Annual Meeting of the APS Division of Plasma Phsyics*, 2005.

[26] B. Wilkinson and M. Allen. *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[27] Q. Yang and H. Koutsopoulos. Microscopic traffic simulator for evaluation of dynamic traffic managment systems. *Transportation Research Part C(Emerging Technologies),4(3)*, 1996.