On the Energy Efficiency of Graphics Processing Units for Scientific Computing

S. Huang, S. Xiao, W. Feng
Department of Computer Science
Virginia Tech
{huangs, shucai, wfeng}@vt.edu

Abstract

The graphics processing unit (GPU) has emerged as a computational accelerator that dramatically reduces the time to discovery in high-end computing (HEC). However, while today's state-of-the-art GPU can easily reduce the execution time of a parallel code by many orders of magnitude, it arguably comes at the expense of significant power and energy consumption. For example, the NVIDIA GTX 280 video card is rated at 236 watts, which is as much as the rest of a compute node, thus requiring a 500-W power supply. As a consequence, the GPU has been viewed as a "non-green" computing solution.

This paper seeks to characterize, and perhaps debunk, the notion of a "power-hungry GPU" via an empirical study of the performance, power, and energy characteristics of GPUs for scientific computing. Specifically, we take an important biological code that runs in a traditional CPU environment and transform and map it to a hybrid CPU+GPU environment. The end result is that our hybrid CPU+GPU environment, hereafter referred to simply as GPU environment, delivers an energy-delay product that is multiple orders of magnitude better than a traditional CPU environment, whether unicore or multicore.

1. Introduction

GPGPU, short for general-purpose computation on graphics processing units, has already demonstrated its ability to accelerate the execution of scientific applications, e.g., [8], [10], [12], [14], [18], [19]. These accelerated applications benefit from the GPU's massively parallel, multi-threaded multiprocessor design, combined with an appropriate data-parallel mapping of an application to it.

The emergence of more general-purpose programming environments, such as Brook+ [1] and CUDA [2], has made it easier to implement large-scale applications on GPUs. The CUDA programming model from NVIDIA was created for developing applications on NVIDIA's GPU cards, such as the GeForce 8 series. Similarly, Brook+ supports stream processing on the AMD/ATi GPU cards. For the purposes of this paper, we use CUDA atop an NVIDIA GTX 280 video card for our scientific computation.

GPGPU research has focused predominantly on accelerating scientific applications. This paper not only accelerates scientific applications via GPU computing, but it also does so in the context of characterizing the power and energy consumption of the GPU. To date, there has been little reseach done in evaluating the energy consumption and energy efficiency of GPUs for general-purpose computing. Furthermore, even less research has been conducted on the optimization of GPU energy consumption and energy efficiency. This paper serves as a first step in this direction.

As a starting point, we begin with a well-known biological code that calculates the electrostatic properties of molecules in order to aid in understanding the mechanism behind their function. Specifically, we use a software code called GEM [6], [7], [16] to compute the electrostatic potential map of macromolecules in a water solution. In addition to the provided serial version of GEM, we implement two additional versions — a multithreaded CPU version and a hybrid CPU+GPU version, which we refer to hereafter as the GPU version as the computation runs nearly exclusively on the GPU — and evaluate their performance, energy consump-

tion, and energy efficiency. The GPU version of GEM performs the best in all three aspects, i.e., performance, energy consumption, energy efficiency. Moreover, we explore the design space of running GEM on the GPU and analyze its variation with respect to performance, energy consumption, and energy efficiency.

The contributions of this paper are as follows:

- An evaluation of the performance, energy consumption, and energy efficiency of general-purpose computations on an NVIDIA GPU.
- An exploration of the GPU design space, i.e., threads, blocks, and kernel launches, using CUDA and its subsequent impact on performance, energy consumption, and energy efficiency.
- A comparison of the performance, energy consumption, and energy efficiency of our GPU implementation versus a multithreaded CPU version and the original serial CPU version.

2. Background

Here we present some background material in order to put our work in perspective, specifically the GEM software [7] and the NVIDIA CUDA programming model [2].

2.1. GEM Software

GEM calculates the electrostatic potential generated by charges inside a molecule. The utility of the electrostatic potential to gain understanding of the function of proteins and nucleic acids has long been established [5], [9]. Electrostatic effects in turn may be critical in understanding the function of viruses and other structures integral to the advancement of medical science and biology.

Traditionally, methods based on numerical solutions of the Poisson-Boltzman (PB) equation have been used to calculate the potential. Though this method is the most accurate among practical approaches, it is often associated with high algorithmic complexity and high computational costs, especially for large structures [4]. To address this problem, Gordon et al. [6] proposed an analytical approximation of the Poisson equations. In [7], which is the continuation of [6], the authors proposed a more efficient algorithm that served as the basis for GEM. In short, GEM is a tool for computing, extracting, visualizing, and outputting the electrostatic potential around macromolecules. In addition, GEM supports reading and writing potential field files in the format of the MEAD package, mapping electrostatic potential to the molecular surface, presenting image output in Targa file format (TGA) format, and providing a graphical user interface [7].

2.2. CUDA on Graphics Processors

GPUs are dedicated graphics devices that have been used to accelerate the computer graphics rendering for many years. Recently, it has evolved into a device that can be dual-purposed into one that can also support scientific computing across a range of data-parallel applications with the promise of tremendous speed-up.

The GPU version of GEM is implemented using the Compute Unified Device Architecture (CUDA) on the NVIDIA GTX 280 GPU, whose execution units are CUDA-capable and organized into multiprocessors with each one of them contains 8 scalar cores. In CUDA, the multiprocessor architecture is called SIMT (Single Instruction, Multiple Thread), which is effectively SIMD (Single Instruction, Multiple Data).

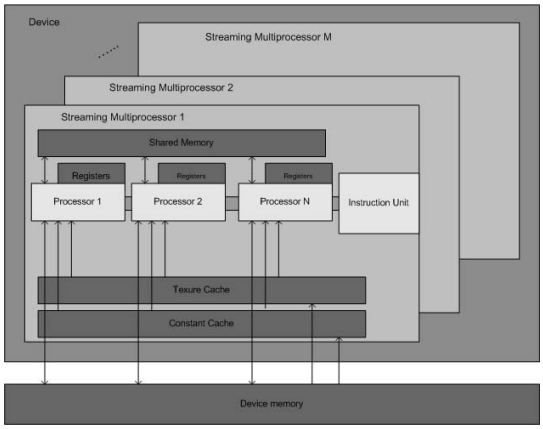

Figure 1. Overview of the NVIDIA Compute Unified Device Architecture (From the NVIDIA CUDA Programming Guide [2])

The CUDA programming model is an extension to the C programming language, which makes it easy for programmers to offload computationally intensive computations to the GPU to take the advantage of its computational power. A kernel in CUDA is a global function that can be executed in the GPU device and called by the host (CPU). Logically, threads are grouped into blocks and thread blocks are grouped into grids. Because a GPU device contains multiple multiprocessors, multiple blocks and threads can be triggered and executed in parallel on the device. The ID of each thread can be calculated uniquely from

the indexes of the block and the grid it belongs to. With respect to data sharing, each thread has data for its own usage, and threads *within* a block can share data amongst each other, thus allowing for data synchronization *within* a block. But GPGPU-supported data synchronization *across* different blocks in a grid currently does not exist.

When a kernel is launched, threads in the same block will execute on the same multiprocessor. Thus, if there are too many threads in a block to fit inside the resources of a single multiprocessor, the kernel launch fails. Similarly, multiple blocks are grouped into a grid, but different blocks in a grid may or may not execute in parallel, depending on the number of multiprocessors on the GPU, the amount of shared memory they use, and the number of registers per multiprocessor.

In addition, groups of 32 threads, i.e., *warps*, execute the same instruction as each other. Optimal performance is achieved when all threads within the warp are performing the same instruction. The reason for this is that when divergence occurs within a warp, every thread needs to execute every instruction within both branches, taking up to double the time per branch.

An execution kernel can access memory in a number of ways. One is via the on-chip memory; each multiprocessor contains a set of 32-bit registers along with a shared memory region which is quickly accessible by any core on the multiprocessor, but hidden from other multiprocessors. There is also off-chip memory that contains both local memory and global memory. Compared to on-chip memory, the read latency of the off-chip memory is much higher, but its size is much larger. Besides the above two types, a multiprocessor contains two read-only caches, one for textures and the other for constants, to improve memory access to texture and constant memory, respectively.

3. Related Work

Initial research on GPU power and thermal behavior can be traced back to Sheaffer et al. [17], who propose Qsilver — a simulation framework for graphics architectures to explore a series of thermal management techniques, e.g., dynamic voltage scaling, clock gating, multiple clock domains, etc.

Ramani et al. [13] present a modular architectural power estimation framework that helps GPU designers with early power efficiency exploration. The paper also demonstrates the utility of the framework by employing the optimizations of bus encoding and repeater sizing/spacing.

Takizawa et al. [20] propose a programming framework called SPART, short for stream programming with runtime auto-tuning, that dynamically selects the best available processor for execution of a given task on a hybrid computing system of CPU and GPU so as to improve the energy efficiency.

Most recently, Rofouei et al. [15] present an experimental investigation on the power and energy cost of GPU operations and a cost/performance comparison versus a CPU-only system.

NVIDIA developed its power-management solution for all NVIDIA graphics processing units called PowerMizer [11]. PowerMizer can effectively extend battery life in laptops, and more generally, make more effective use of electrical power. The counterpart of PowerMizer for the ATi GPU card is PowerPlay [3].

4. Implementation

In this section, we present three implementations of the GEM code: serial CPU, multithreaded CPU, and GPU-based, respectively. The original GEM package provides a serial CPU implementation, which we make use of here. We implemented the multithread CPU and GPU-based versions of the GEM code.

4.1. Serial GEM

As mentioned in Section 2, GEM is a tool for computing, extracting, visualizing, and outputting the electrostatic potential around macromolecules. For the electrostatic potential of each macromolecule, the software computes the potential at each of a set of points on or around its surface by taking the summation of the potential contributed by all atoms in the macromolecule. In its most basic form, the GEM potential calculation for each point and each atom is independent and has nothing to do with other atoms or points. Therefore, the calculation of all parts of a macromolecule can be performed in parallel.

4.2. Multithreaded GEM

Because the implementation of the serial version of GEM is embarrassingly parallel, it makes the implementation of the multithreaded version and the GPU version much easier. We use `gprof` to profile the execution of GEM and find the function `calc_potential_single_step()` (performing the potential calculation) accounts for 99% of the

execution time. Consequently, we devote our efforts in parallelizing this function.

We use Linux pthreads to implement the multithreaded GEM. Since the work of the potential calculation function is data parallel, work can be divided efficiently, and the workload can be balanced nearly optimally. In essence, after the main process enters the function of calculating potentials, it forks a number of threads. Each thread is assigned a piece of work. After all the threads finish their work, the main process combines all the results. With respect to thread-to-core mapping, we choose to schedule one thread per processor core.

4.3. GPU GEM

When mapping the original GEM C code onto the NVIDIA GPU using the CUDA programming model, we made the following decisions based on the properties of CUDA.

Thread and Block Scheduling: Our most recent version of an NVIDIA GPU is the GTX 280, which has 30 multiprocessors, each of which has 8 cores for a total of 240 cores. And as noted before, CUDA groups threads into warps, where each warp contains 32 threads. Threads in the same block execute on the same multiprocessor. Therefore, to make full use of the GTX 280, multiple blocks should be used. In other words, the block number per kernel launch should be no less than 30 to get good performance. For number of threads per block, according to [2], we can achieve the best results when the number of threads per block is a multiple of 64. Another factor that should be considered is resource limitations of the multiprocessor. Our experimental study demonstrated that 64 threads per block achieves the best result.

Memory Usage: When the GPU is used for the potential calculation, input data should be transferred to the device memory before they can be used by the multiprocessors. As described previously, different types of memories are available on the GPU. One is the global memory, and another is the texture memory. The global memory has a large size, and it can better support the case where large amounts of data need to be copied between the host and the GPU device. The disadvantage of the global memory is the read latency. It is much higher than that of the texture memory. However, for the texture memory, the array size that is supported is limited. For example, a two-dimensional CUDA array has a maximum width of 2^{13} and a maximum height of 2^{15}. So, when texture memory is used, the number of points in the input data is constrained by this limitation. Also, when there are multiple active warps on a multiprocessor, the CUDA scheduler can overlap the reading latency of a warp by making other warps do the computation. Because of these two reasons and our experimental results, global memory is used for mapping GEM onto the GPU.

Parameter Transfer: Parameters with the `double` data type cannot be transferred to the GPU device directly when the kernel function is called. Instead, they are wrapped in a structure, and the function `cudaMemcpy()` is used to copy those parameters from the host to the device.

5. Experiment Setup

In this section, we describe the hardware and software platform used in our experiments.

5.1. Hardware Platform

The hardware platform in our experiment includes a computing server and a profiling computer and "Watts Up? PRO ES" power meter. The computing server is equipped with an Intel®Core™2 Duo CPU with 2 MB of shared L2 cache, which has two processing cores running at 2.2 GHz. The computing server has 2 GB of DDR2 SDRAM memory. The computing server also contains a NVIDIA GTX 280 graphics card with GT200 GPU, which has 30 multiprocessors and 240 GPU processing cores running at 1296 MHz. The graphics card has 1024 MB of GDDR3 RAM with a memory bandwidth of 141.7 GB per second. Furthermore, this GPU has 16,384 registers per multiprocessor and can support 32 active warps and 1,024 active threads. In addition, it supports for double-precision floating point.

In our experiments, performance and energy of the serial version and multithreaded version are measured with GPU being taken out. Power and energy measurements are performed by the power meter. Figure 2 shows how the computing server, profiling computer, and power meter are interconnected. The power comes from the wall outlet and feeds into the computing server. Power and energy data are recorded by the power meter, and the profiling computer is used to fetch the data from power meter periodically. Energy values are recorded into a log file immediately before and after the execution of GEM code. The difference is the energy consumed by the computing server when running GEM code. The energy value we measure does not include the energy consumed in the idle time before program execution or after the computation completes.

5.2. Software Platform

We run 64-bit Ubuntu GNU/Linux 7.04 distribution (kernel 2.6.20-16) on the computing server. The NVIDIA CUDA 2.0 toolkit and SDK are installed with NVIDIA driver version 177.67 in the computing server. We use the GEM software [6], [7] as our application to study the behavior of performance, energy consumption, and energy efficiency on the three different implementations, as described in Section 4. We choose a sizable nuclesome dataset (about 258,000 points) for our workload. The performance is measured in execution time, and the energy is measured in energy consumed by the computing server in joules. The energy efficiency is measured using the energy-delay (ED) product in joule-seconds.

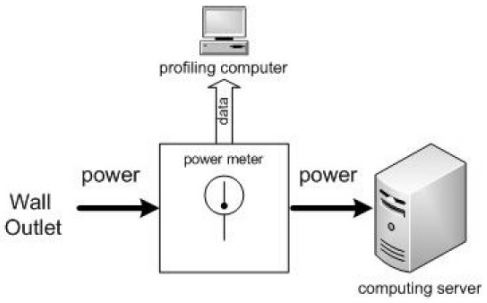

Figure 2. Setup of Experimental Testbed

6. Results

In this section, we show the performance, energy consumption, and energy efficiency of all three implementations of GEM by profiling the execution time and power consumption of the computing server when running GEM. We will show our analysis in Section 7. Figure 3 shows the execution of serial version, multithreaded version, and GPU version of GEM. With respect to the results for the GPU version of GEM, shown in Figures 3, 4, and 5, we use a single kernel launch with as many blocks as possible to run GEM. As will be shown in Section 7, a single kernel launch with as many blocks as possible has the best performance over all other cases for a particular number of threads per block. With respect to the number of threads per block, 64 delivers the best performance.

Table 1. Energy delay product (megajoule-seconds) of GEM. Smaller is better.

	Serial	Multi	GPU
ED	409.1	127.1	0.5

We execute the different versions of GEM at different starting times so that they can be more easily differentiated in the figure. Although the *power* consumption of the GPU is much higher, the task finishes *much* faster. Overall, the energy consumption of the GPU version of GEM is significantly less than that of the serial version and multithreaded version.

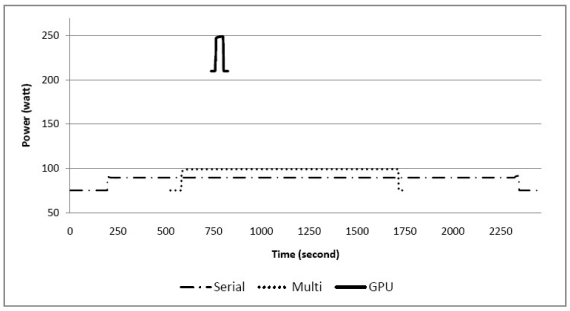

Figure 3. The Execution Time and Power Consumption of the CPU and GPU

6.1. Performance Evaluation

We present here the performance, energy consumption, and energy efficiency of GEM on the three implementations presented in Section 4.

Figure 4 shows the execution time of our three versions of the GEM implementation. The performance of GPU version of GEM is 46 times faster than the serial version and 25 times faster than multithreaded version. (Aside: With a single-precision GPU version of GEM, the results are another order of magnitude better than the results presented here.) Figure 5 shows the energy consumption of the three versions of GEM implementations. The energy consumption of the GPU version is 17 times better than the serial CPU version and 9 times better than the multithreaded CPU version. Table 1 shows the energy delay product (ED) of all three implementations. The energy efficiency of the GPU version of GEM, as measured by the ED product, is a whopping 763 times better than serial version and 237 times better than the multithreaded version.

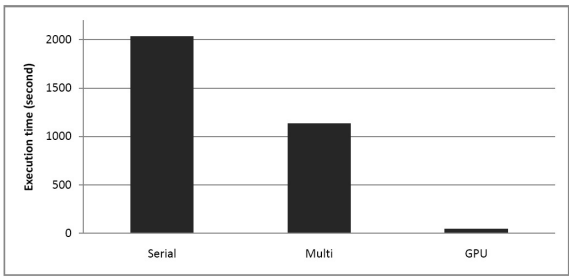

Figure 4. Execution Time of GEM

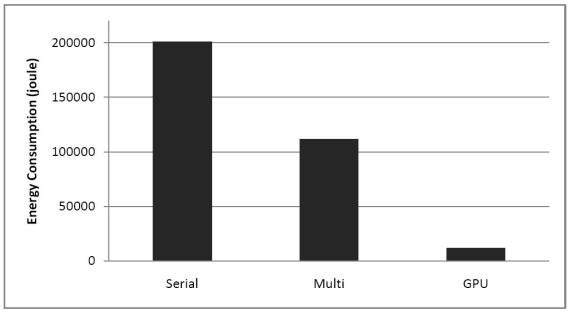

Figure 5. Energy of GEM

6.2. Performance Optimization

In the previous section, we evaluated the GPU version of GEM implemented using a single kernel launch with all blocks. Developers can also choose to use multiple kernel launches to implement GEM on the NVIDIA GPU. This is because NVIDIA CUDA allows developers to choose different numbers of blocks per grid and different numbers of threads per block. We believe that the choice of these two parameters will affect the performance, energy consumption as well as energy efficiency of the computing server. We want to explore the design space of CUDA programming model to discover the optimal solution for mapping data-parallel scientific applications to the NVIDIA GPGPU.

Figure 6 shows the execution time of the GPU version of GEM with various number of blocks and various number of threads. Similarly, Figure 7 shows the energy consumption, and Figure 8 shows the energy-delay (ED) product. In all of the three figures, the horizontal line right below each zigzag line represents the one kernel launch implementation of GPU version of GEM for that thread numbers. For example, in figure 6, the horizontal line right below the 16 threads zigzag line represents the execution time of one kernel launch of GPU version of GEM implemented using 16 threads per block.

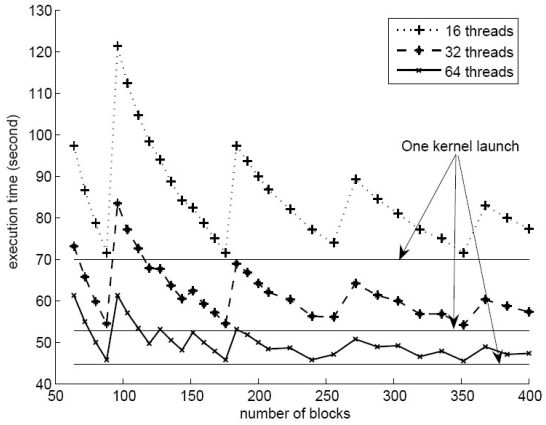

Figure 6. Execution Time of GEM on the GPU

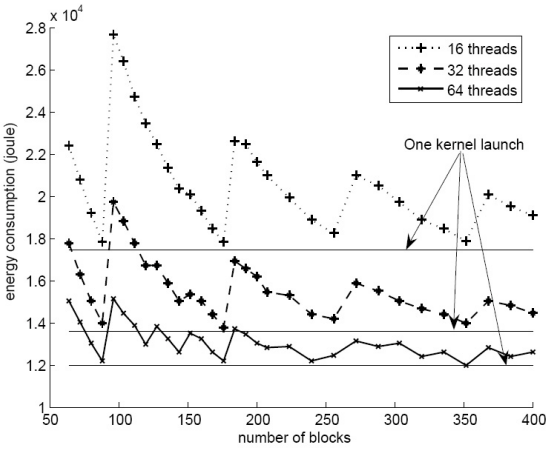

Figure 7. Energy Consumption of GEM on the GPU

7. Analysis

In this section, we confirm three overarching (self-evident) principles from our experiments. These principles can help with making decisions in choosing efficient application-design strategies.

7.1. GPGPU for Data-Parallel Applications

Among the serial CPU, multithreaded CPU, and GPU versions of GEM, the GPU implementation is clearly the best choice in terms of performance, energy consumption, and energy efficiency. Figure 4 and 5 and

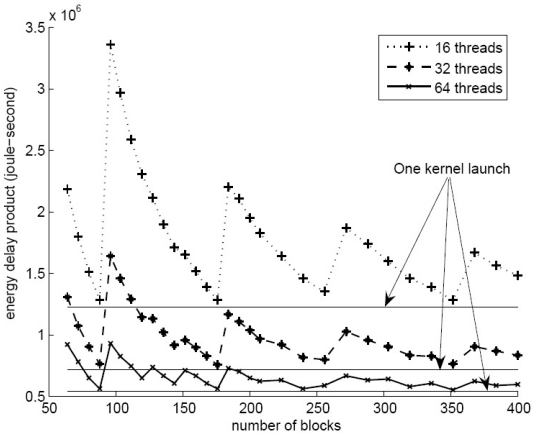

Figure 8. Energy-Delay Product of GEM on the GPU

Table 1 show that the performance, energy consumption and energy delay product of GPU version of GEM is much better than serial version and multithreaded version.

7.2. (Obvious) Synchronous Behavior

From Figures 6, 7, and 8, we observe that the performance, energy consumption, and energy efficiency have similar curves over all execution point (number of blocks and number of threads). The performance behavior is sub-linear to the energy behavior, and the performance and energy consumption are highly synchronized. This observation is due to the fact that the power consumption of GPU when running GEM code does not vary much. Under this circumstance, the energy consumption will be proportional to the execution time. As a result, the energy-delay product is proportional to the square of the execution time. In short, the behavior of performance, energy consumption, and energy efficiency are highly synchronized.

7.3. Optimal # of Kernel Launches

In Figure 6, we use a horizontal line to represent the execution time of one kernel launch for 16 threads, 32 threads, and 64 threads, respectively. Each thread line approaches the one kernel-launch line as the number of blocks increases but stays above this horizontal line. That is, as the number of block increases, the execution time will shrink to this asymptotic limit. This limit is the optimal execution point for executing GEM on the GPU.

Table 2. Optimal Execution of GEM on GPU

	16 threads	32 threads	64 threads
Time (seconds)	70.0	52.8	44.6
Energy (joules)	17,460	13,625	11,996
ED (joule-sec)	1,221,426	719,068	535,392

As shown in Table 2, the execution times of the "one kernel launch" case of GEM for 16 threads per block, 32 threads per block, and 64 threads per block are 70.0 seconds, 52.8 seconds and 44.6 seconds, respectively. These numbers are less than any other test case with multiple kernel launches.

Table 2 shows the energy consumption of the one kernel launch version of GEM for 16 threads per block, 32 threads per block, and 64 threads per block are 17,460 joules, 13,625 joules, and 11,996 joules, respectively.

With respect to the energy-delay product, Figure 8 shows that one kernel launch has the optimal energy-delay product value for each number of threads. As shown in Table 2, the optimal energy-delay products are 1,221,426 joule-seconds for 16 threads, 719,068 joule-seconds for 32 threads, and 535,392 joule-seconds for 64 threads.

8. Conclusion and Future Work

This paper presents our evaluation and analysis of the efficiency of GPU computing for data-parallel scientific applications. Starting with a biomolecular code that calculates electrostatic properties in a data-parallel manner (i.e., GEM), we evaluate our different implementations of GEM across three metrics: performance, energy consumption, and energy efficiency.

In the future, we will continue this work by investigating the effects of memory layout (global, constant, texture) on GPU performance and efficiency. In addition, we will delve further into potential techniques for proactively reducing power and conserving energy on the GPU.

Acknowledgements

We would like to thank Prof. Alexey Onufriev and his team, particularly John Gordon and Alex Fenley, for introducing us to the world of biomolecular electrostatic potential and Tom Scogland for his assistance in optimizing the GEM code for the NVIDIA GPU.

This work was supported in part by an NVIDIA Professor Partnership Award.

References

[1] AMD Brook+ Presentation. In *SC07 BOF Session*, November 2007.

[2] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide, Version 2.0, June 2008.

[3] AMD/ATI. Powerplay. http://ati.amd.com/products/powerplay/index.html.

[4] N. A Baker. Improving Implicit Solvent Simulations: A Poisson-Centric View. In *Theory and Simulation/Macromolecular Assemblages*, volume 15, issue 2, pages 137-143, April 2005.

[5] T. Douglas and D. R. Ripoll. Calculated Electrostatic Gradients in Recombinant Human H-chain Ferritin. In *Protein Science*, volume 7, issue 5, pages 1083-1091, July 1998.

[6] J. Gordon, A. Fenley, and A. Onufriev. An Analytical Approach to Computing Biomolecular Electrostatic Potential. I. Derivation and Analysis. In *The Journal of Chemical Physics*, August 2008.

[7] J. Gordon, A. Fenley, and A. Onufriev. An Analytical Approach to Computing Biomolecular Electrostatic Potential. II. Validation and Applications. In *The Journal of Chemical Physics*, August 2008.

[8] N. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors. In *Proc. of ACM/IEEE SuperComputing*, 2008.

[9] B. Honig and A. Nicholls. Classical Electrostatics in Biology and Chemistry. In *Science*, volume 268, issue 5214, pages 1144-1149, May 1995.

[10] P. LeGresley. Case Study: Computational Fluid Dynamics (CFD). In *International Supercomputing Conference*, 2008.

[11] NVIDIA. Powermizer. http://www.nvidia.com/object/feature_powermizer.html.

[12] J. Phillips, J. Stone, and K. Schulten. Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters. In *Proc. of ACM/IEEE Super-Computing*, 2008.

[13] K. Ramani, A. Ibrahim, and D. Shimizu. PowerRed: A Flexible Power Modeling Framework for Power Efficiency Exploration in GPUs. In *Worskshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2007.

[14] C. Rodrigues, D. Hardy, K. Schulten J. Stone, and W. Hwu. GPU Acceleration of Cutoff Pair Potential for Molecular Modeling Applications. In *Proceedings of the International Conference on Computing Frontiers*, May 2008.

[15] M. Rofouei, T. Stathopoulos, S. Ryffel, W. Kaiser, and M Sarrafzadeh. Energy-Aware High Performance Computing with Graphic Processing Units. In *Workshop on Power Aware Computing and System*, December 2008.

[16] H. Scheraga. Recent Developments in the Theory of Protein Folding: Searching for the Global Energy Minimum. In *Biophysical Chemistry*, volume 59, issue 33, pages 329-339, April 1996.

[17] J. Sheaffer, K. Skadron, and D. Luebke. Studying Thermal Management for Graphics-Processor Architectures. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.

[18] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. Owens. Efficient Computation of Sum-products on GPUs Through Software-Managed Cache. In *Proc. of ACM/IEEE SuperComputing*, 2008.

[19] S. Stone, J. Haldar, S. Tsao, W. Hwu, Z. Liang, and B. Sutton. Accelerating Advanced MRI Reconstructions on GPUs. In *Proceedings of the International Conference on Computing Frontiers*, 2008.

[20] H. Takizawa, K. Sato1, and H. Kobayashi. SPRAT: Runtime Processor Selection for Energy-aware Computing. In *The Third international Workshop on Automatic Performance Tuning*, 2008.