Analysis of a Computational Biology Simulation Technique on Emerging Processing Architectures

Jeremy S. Meredith, Sadaf R. Alam and Jeffrey S. Vetter

Computer Science and Mathematics Division
Oak Ridge National Laboratory, Oak Ridge, TN 37831 USA
{jsmeredith,alamsr,vetter}@ornl.gov

Abstract

Multi-paradigm, multi-threaded and multi-core computing devices available today provide several orders of magnitude performance improvement over mainstream microprocessors. These devices include the STI Cell Broadband Engine, Graphical Processing Units (GPU) and the Cray massively-multithreaded processors—available in desktop computing systems as well as proposed for supercomputing platforms. The main challenge in utilizing these powerful devices is their unique programming paradigms. GPUs and the Cell systems require code developers to manage code and data explicitly, while the Cray multithreaded architecture requires them to generate a very large number of threads or independent tasks concurrently. In this paper, we explain strategies for optimizing a molecular dynamics (MD) calculation that is used in bio-molecular simulations on three devices: Cell, GPU and MTA-2. We show that the Cray MTA-2 system requires minimal code modification and does not outperform the microprocessor runs; but it demonstrates an improved workload scaling behavior over the microprocessor implementation. On the other hand, substantial porting and optimization efforts on the Cell and the GPU systems result in a 5x to 6x improvement, respectively, over a 2.2 GHz Opteron system.

1 Introduction

Multi-paradigm, multi-threaded and multi-core computing devices available today provide several orders of magnitude performance improvement over mainstream microprocessors. These devices include the STI Cell Broadband Engine (BE) [15], Graphical Processing Units (GPUs) [8] and the Cray XMT [1] (Eldorado[12]) systems —available in desktop computing systems as well as proposed for Petaflops-scale supercomputing platforms. The Cell processor is a heterogeneous multi-core system, which is capable of yielding 256 GFLOPS for single-precision floating-point calculations. Presently multi-core system designs, although planned for virtually all future processing systems, are only available as homogeneous dual-core processors from Intel and AMD for general-purpose processing. These devices provide only incremental performance benefits as compared to emerging, unconventional processing devices like the Cell BE, GPU and the Cray Extreme Multi-threaded (XMT) systems. Note that the XMT system is a follow on of the MTA-2 system. The MTA-2 system, not as powerful in terms of the peak FLOPS as the Cell BE and GPUs, addresses the 'memory wall' problem by providing a uniform memory hierarchy and a latency tolerant multithreaded architecture.

The paper outline is as follows: section 2 outlines motivation for our research efforts. In section 3, we provide details of the three target systems:, Cell, GPU and MTA-2, and the MD calculation. The related research is presented in section 4. Section 5 outlines implementation and optimization details for of the MD calculation on the three target systems. Conclusions and future plans are outlined in section 6.

2 Motivation

The biological processes within a cell occur at multiple lengths and times scales. The processing requirements for bio-molecular simulations, particularly at large length and time scales, far exceed the available computing capabilities of the most powerful computing platforms today. Another challenge is scaling limits of popular bio-molecular simulation frameworks, which have not kept pace with the scaling of high-end, massively-parallel processing (MPP) systems [9]. Blue Gene/L, the most powerful supercomputer system today, has 64K processing cores, while the current scaling limits of most MD algorithms available in popular bio-molecular simulation frameworks is a few hundred processors. At the same time however, we recognize that a number of simulation users will not have access to these high-end supercomputing platforms. We therefore target extremely powerful, emerging processing technologies that can

1-4244-0910-1/07/$20.00 ©2007 IEEE

benefit a large community of desktop and small cluster users. The focus of this study is to characterize an MD calculation that is used in bio-molecular simulations on processing devices that offer: (1) a substantial theoretical speedup over the mainstream processor technologies, and (2) programming paradigms and interfaces that are considerably different from mainstream micro-processor architectures.

3 Background

3.1 The Cell Broadband Engine Processor

The Cell Broadband Engine processor is a heterogeneous multicore processor, with one 64-bit Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs) as shown in Figure 1. The PPE is a dual-threaded Power Architecture core containing extensions for SIMD instructions (VMX) [13]. The SPEs are less traditional, in that they are lightweight processors with a simple, heavily SIMD-focused instruction set, with a small (256KB) fixed-latency local store (LS), a dual-issue pipeline, no branch prediction, and a uniform 128-bit 128 entry register file [14].

The SPEs operate independently from the PPE and from each other, have an extremely high bandwidth DMA engine for transferring data between main memory and other SPE local stores, and are heavily optimized for single-precision vector arithmetic. Regrettably, these SPEs are not optimized for double-precision floating point calculations, making the Cell an uncertain target for scientific applications in the minds of many developers.

3.1.1 Programming the Cell Processor

Our Cell hardware (i.e. blade servers) runs a PowerPC Linux operating system, with a 2.6 series kernel modified to be aware of the SPEs, available at the Barcelona Supercomputing Center. Standard compilers, such as the 4.x series GNU development tool chain, are available, which currently are unable to perform significant code optimization for PPEs and SPEs. At the time of this writing, IBM also offers its XL C/C++ compilers – however, as they are an Alpha Edition, and not native to the Cell platform (only as cross-compilers from x86), we are not evaluating them here. To create code that can execute on an SPE, one must use the SPE-specific compiler from these toolchains. A SPE-specific port of the GNU and IBM compilers can both generate the requisite object code and link against SPE-specific libraries to produce code that can be loaded and run on the SPEs.

The Cell processor architecture enables great flexibility with respect to programming models [7]. From one perspective, the fact that each SPE has its own memory that is not automatically kept coherent with main memory makes the collection of SPEs within a single processor look like a distributed memory system-on-a-chip. This view of the architecture suggests task parallel programming models, where each SPE operates more-or-less independently of the other SPEs, possibly with orchestration by a master thread running on the PPE. Because of the high concurrent bandwidth of the Element Interconnect Bus that connects the SPEs, and because SPE DMA transfers are cache coherent, data parallel programming models like that of OpenMP are also an attractive approach for programming the Cell processor. For our application case study, we used the "Asynchronous Thread Runtime" programming model, creating SPE threads as needed, starting the new threads at the address of a performance critical application function that has been ported to run on the SPE.

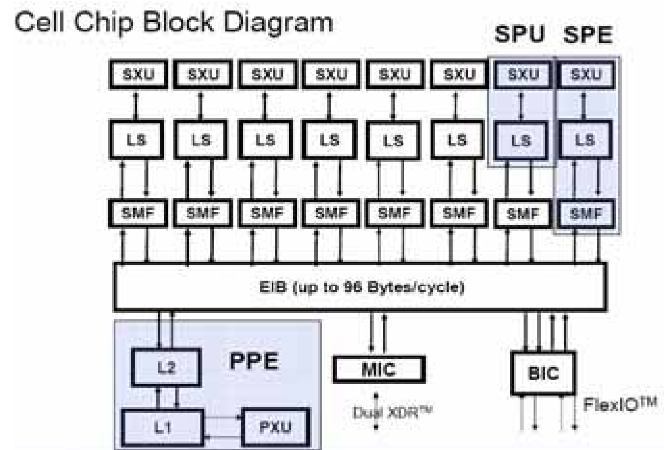

Figure 1: Design components of the Cell BE [http://www.research.ibm.com/cell/heterogeneousCMP.html]
.

3.2 The GPU Architecture

The origin of Graphics Processing Units, or GPUs, is in accelerating the real-time graphics rendering pipeline. As developers demanded more power and programmability from graphics cards, these cards became appealing for general purpose computation, especially as mass markets force even high-end GPUs into low price points [5]. The high number of FLOPS in GPUs comes from the parallelism in the architecture.

Figure 2 shows an earlier generation high-end part from NVIDIA, with 16 parallel pixel pipelines. It is these programmable pipelines that form the basis of general purpose computation on GPUs, and the parallelism is increasing; the next generation from NVIDIA contained 24 pipelines, and that number is growing. Typical high end cards today have 512MB of local memory or more,

and support from 8-bit integer to 32-bit floating point data types, with 1, 2, or 4 component SIMD operations.

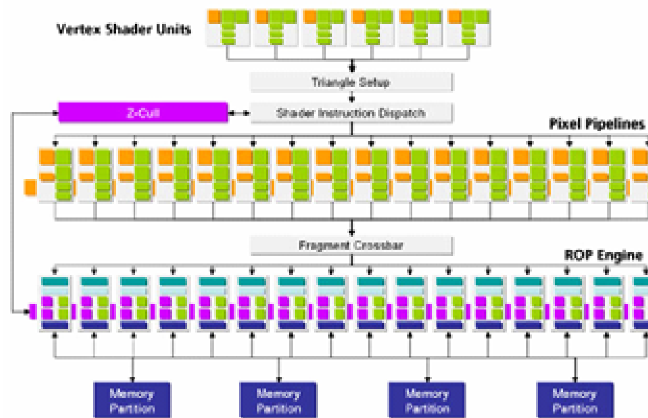

Figure 2: The NVIDIA GeForce 6800 GPU

3.2.1 Programming GPUs

There are several ways to program the parallel pipelines of a GPU. The most direct way is to use a GPU oriented assembler or a compiled C-like language with graphics related intrinsics, like Cg from NVIDIA [8]. Furthermore, as GPUs are coprocessors, they require interaction from the CPU to handle high level tasks such as moving data to and from the card and setting up these "shader programs" to execute on the pixel pipelines. At this high level, GPUs have typically required programming in a graphics oriented API, such as OpenGL or DirectX. Graphics card manufacturers and third parties have recognized the need for non-graphics oriented APIs at every level, and a variety of solutions have now been announced or released to abstract or bypass the specialized graphics knowledge traditionally needed to make use of the computational horsepower of these cards.

Inherently, GPUs are stream processors, as a shader program cannot read and write to the same memory location. Thus, arrays must be designated as either input or output, but not both. There are technical limitations on the number of input and output arrays addressable in any particular shader program. Furthermore, the execution paradigm is inherently a gather-based system: a shader program may read from any input locations, but it has only one location in each output array to which it may write, and this location is designated before the shader program begins execution. Together these restrictions form a set of design challenges for accelerating a variety of algorithms using GPUs.

3.3 The MTA-2 System

The Multi-Threaded Architecture (MTA) uses a high degree of multi-threading instead of data caches to address the gap between the rate at which modern processors can execute instructions and the rate at which data can be transferred between the processor and main memory. The MTA uses processors that support a high degree of multi-threading compared to current commercial off-the-shel processors (as shown in Figure 3) [2]. An MTA processor tolerates memory access latency by supporting many concurrent streams of execution (128 in the MTA-2 system processors). A processor can switch between streams on each clock cycle. To enable such rapid switching between streams, each processor maintains a complete thread execution context for each of its 128 streams. An MTA-2 system consists of a collection of processor modules and a collection of memory modules, connected by an interconnection network. Unlike conventional designs, MTA-2 processor modules contain no local memory; it does include an instruction stream shared between all of its hardware streams [4].

3.3.1 Programming the Cray MTA-2

The Cray MTA-2 platform is significantly different from contemporary, cache-based microprocessor architectures. These differences are reflected in the MTA-2 programming model and, consequently, its software development environment [3]. The key to obtaining high performance on the MTA-2 is to keep its processors saturated, so that each processor always has a thread whose next instruction can be executed. If the collection of threads presented to a given processor is not large enough to ensure this condition the processor will be under-utilized.

In the high-level language source code of an MTA-2 program, parallelism can be expressed both implicitly and explicitly. Implicit parallelism is expressed using the source language's loop constructs, such as a C `for` loop or Fortran `DO` loop. The MTA-2 compilers automatically parallelize the body of such loops so that a collection of threads executes the loop, with each thread executing some of the loop iterations. There are some restrictions on the types of loops the MTA-2 compilers can parallelize automatically due to data and control dependencies, and sometimes compiler directives must be used to indicate that a given loop can be parallelized.

The MTA-2 is no longer an active product in the Cray product line. However, an extreme multi-threaded processor is recently announced as the Cray XMT system. Although the XMT system uses multithreaded processors similar to the MTA-2, there are several important differences in the memory and network architecture; it will not have the MTA-2's nearly uniform memory access latency, so data placement and access locality will be an important consideration when programming these systems. The XMT multithreaded processors will operate at a higher clock rate and the XMT design allows systems with up to 8000 processors, whereas the largest possible MTA-2 system contains only 256 processors.

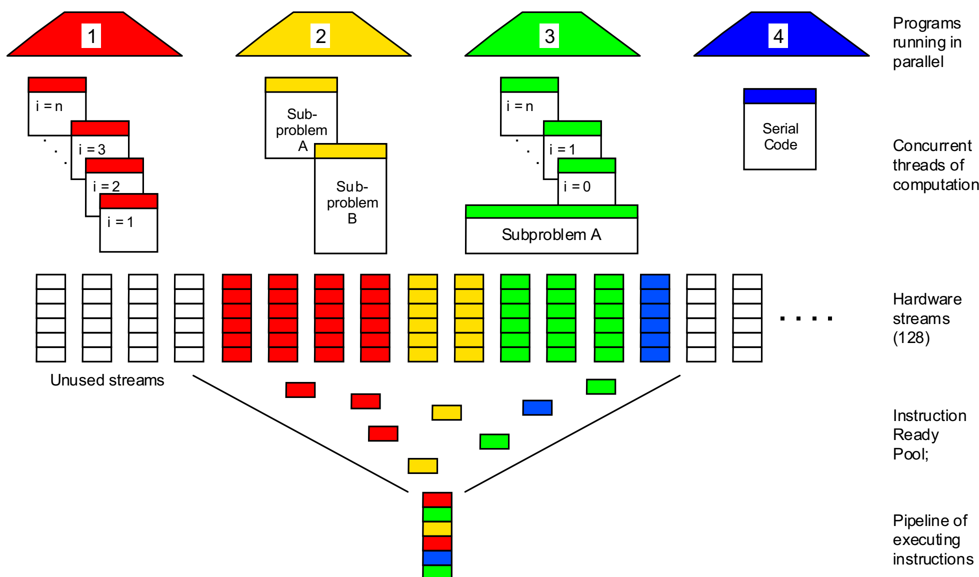

Figure 3: Block diagram of the MTA-2 system

3.4 Molecular Dynamics Calculations

Molecular Dynamics (MD) is a computer simulation technique where the time evolution of a set of interacting atoms is followed by integrating the equations of motion [16]. In the Newtonian interpretation of dynamics, the translational motion of a molecule is caused by force exerted by some external agent. The motion and the applied force are explicitly related through Newton's second law: $F_i = m_i a_i$. m_i is the atom's mass, $a_i = \frac{d^2 r_i}{dt^2}$ is its acceleration, and F_i is the force acting upon it due to the interactions with other atoms. MD techniques are extensively used in many areas of scientific simulations including biology, chemistry and materials.

MD simulations are computationally very expensive. Typically the computational cost is proportional to N^2, where N is the number of atoms in the system. In order to reduce the computational cost, a number of algorithm-oriented techniques such as a cutoff limit are used. It is assumed that atoms within a cutoff limit contribute to the force and energy calculations on an atom. As a result, the MD simulations do not exhibit a cache friendly memory access pattern. An atom and its neighbors continuously move during a simulation run, and an atom does not interact with a fixed pair or set of atoms. Since the positions of atoms are usually stored in arrays, multiple accesses to the position arrays in a random manner is required to calculate the cutoff distance, and subsequently to perform force calculations.

Several techniques have been proposed and implemented in bio-molecular frameworks to address the unfriendly and unpredictable cache behavior of the MD calculations. One of the most common techniques is the neighboring atom pairlist construction, which is updated every few simulation time steps. This scheme results in a small memory and computation overhead. We do not employ any optimization technique that has been proposed for cache-based systems. Instead, we calculate the distances on the fly and perform calculations accordingly.

3.5 MD Kernel

Our MD kernel contains two important parts of an MD calculation: force evaluation and integration. Calculation of forces between bonded atoms is straightforward and less computationally intensive as there are only a very small numbers of bonded interactions as compared to the non-bonded interactions. The effect of non-bonded interactions are modeled by a 6-12 Lennard-Jones (LJ) potential model:

$$V(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right].$$

LJ potential combines large distance attractive forces (r^{-6} term) and short distance repulsive force (r^{-12} term) between two atoms. The integration in our kernel is implemented using the velocity Verlet algorithm, which calculates the trajectories of atoms from the forces. The Verlet algorithm uses positions and acceleration at time t and positions from time $t + \delta t$ to calculate new positions at time $t + \delta t$. The pseudo code for our implementation is given in Figure 4. Steps are repeated for n simulation time steps. n depends on the time-scale of the simulated system and the value of δt.

```
1. advance velocities
2. calculate forces on each of the N atoms
      compute distance with all other N-1 atoms
           if(distance within cutoff limits)
                  compute forces
3. move atoms based on their position,
    velocities & forces
4. update positions
5. calculate new kinetic and total energies
```

Figure 4: MD kernel implemented on MTA-2

The most time-consuming part of the calculation is step 2, in which an atom's neighbors are determined using the cutoff distance and subsequently the calculations are performed (N^2 complexity). We attempt to optimize this calculation on the target platforms, and we compare the performance to the reference implementation on a 2.2 GHz Opteron. Note that we implement single-precision versions of the calculations on the Cell BE and the GPU accelerated system, while the MTA-2 implementation is in double-precision.

4 Related work

The three systems presented in this paper have been targeted and evaluated by researchers for a range of scientific kernels. The focus of much of this research is numerical functions such as the Basic Linear Algebra Subroutine (BLAS) functions, equation solvers and integer-based bio-informatics search and sequence alignment methods. Here we highlight recent efforts in the areas of computational biology and bio-informatics calculations. For example, W. Liu *et. al.* [17] and Y. Liu *et. al.* [18] presented mapping and performance results of Smith-Waterman calculations onto the GPU devices, and showed a substantial speedup over the contemporary processors. I. Buck presents acceleration strategies for GROMACS [6], an MD framework, on GPU using a streaming language, Brook [11].

Similarly, due to the raw processing power of the Cell system, 256 GFLOPS for single-precision floating-point calculations and the availability of SIMD units, it has been evaluated for scientific applications. Williams *et. al.* [19] present a performance model for the Cell system and validation results from a Cell simulator for four scientific kernels. Although the authors identify different programming environments for the Cell system, they only consider one programming model, the data-parallel programming model, which is widely used for parallel scientific calculations. Likewise, the MTA-2 system and its predecessor have been investigated extensively for scientific computing. Bokhari and Sauer [10] investigated dynamic programming sequence alignment algorithms for DNA sequences on the Cray MTA-2 system. Their algorithms are reported to scale to up to eight MTA-2 processors and the implementation relies extensively on the use of full/empty bits in MTA-2 memory to facilitate parallel execution in the dynamic programming algorithms.

Our research follows these earlier investigations on evaluating the feasibility of scientific calculations on unconventional processing architectures. The unique contribution of our study is that it provides an insight into programming and optimization strategies, and performance potential of three powerful emerging but significantly different systems for a floating-point intensive bio-molecular simulation kernel. The systems evaluated in this paper are available and planned for desktop system users as well as high-end, petaFLOPS-scale parallel systems.

5 Experiments and Results

5.1 The Cell System

Our programming model for the Cell processor involves finding time consuming functions that map well to the SPE cores, and instead of calculating these functions on the PPE, we launch "threads" on the SPEs to read the necessary information into their local stores, perform the calculations, and write the results back into main memory for the next calculation steps on the PPE. Because of its high percentage of the total runtime, the acceleration computation piece alone was offloaded to SPEs.

The molecular dynamics application kernel deals with three dimensional positions, velocities, accelerations, forces, and other vectors, so the most natural way to make use of the 4-component SIMD operations on the SPE is to use the first three components of the inherent SIMD data types for the `x`, `y`, and `z` components of each of these arrays. Figure 5 shows the runtime of the acceleration computation function for 2048 atoms, when running on a single SPE, across various SIMD optimizations.

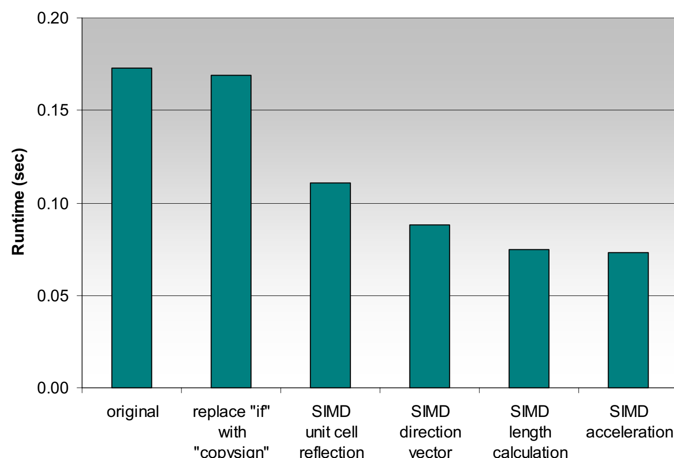

Figure 5: SIMD optimization for the MD kernel.

One expensive part of this acceleration computation is searching the 27 neighboring unit cells for the instances of each atom pair which are closest. The first step in optimizing this piece was to replace an "if" test in that section with extra math; as the SPEs lack branch prediction, this provided a small speedup. However, the real advantage here was that instead of looping over all three dimensions, all three axes could be searched simultaneously using the SIMD intrinsics on the SPE. This further optimization provided a very large speedup, running over 1.5x faster than the original. The next two optimizations replaced loops over the three components for finding the direction and calculating its length with SIMD versions, resulting in 21% and 15% improvements, respectively.

Once an interacting atom pair is found, the force between them must be converted into a 3D acceleration vector. The SIMDization of this operation is the final optimization step in this figure. Unfortunately, since so few of the tested atoms interact, very little runtime is actually spent in this loop, and so the total improvement in runtime was only 3%.

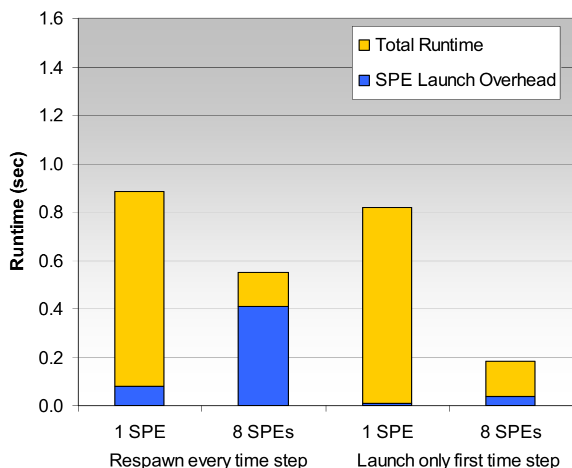

Figure 6: SPE launch overhead on MD using newer Linux kernel.

Figure 6 shows the total runtime of the whole program, and the percentage which is devoted to launching SPE threads. In the first case, we see that when a single SPE is tasked to compute all accelerations, it takes enough time to execute that the thread launch overhead is a small fraction of the runtime. In the second case, we see that parallelization across all 8 SPEs on the Cell processor scales well; each SPE checks approximately one eighth of the total number (N^2) of atom pairs. However, the thread launch overhead grows by a factor of eight, which makes even an efficient parallelization run only about 1.5x faster using all SPEs. In this case, however, there is a simple solution. The communication between the PPE and SPEs is not limited to large asynchronous DMA transfers; there are other channels ("mailboxes") that can be used for blocking sends or receives of information on the order of bytes. As we are offloading only a single function, we can launch the SPE threads only on the first time step, and signal them using mailboxes when there is more data to process. Thus the thread launch overhead is amortized across all time steps. This helps the scaling greatly – this eight-SPE version is now 4.5x faster than this single-SPE version.

Thanks to its effective use of SIMD intrinsics on the SPE, even a single SPE just edges out the Opteron in total performance. Runtime results are listed in Table 1 for a 2048-atom experiment that runs for 10 simulation time steps. With an efficient parallelization, using all 8 SPEs results in a better than 5x performance improvement relative to the Opteron, and 26x faster than the PPE alone. Amortizing the thread launch overhead across even more time steps would further increase this performance gap.

Number of Atoms	2048
Opteron	0.925 sec
Cell, 1 SPE	0.816 sec
Cell, 8 SPEs	0.181 sec
Cell, PPE only	4.701 sec

Table 1: Performance comparison of MD calculations.

5.2 The GPU Architecture

Like the Cell implementation, step 2 was offloaded to the GPU, which is the part of the algorithm that calculates new accelerations from only the locations of the atoms and several constants. For our streaming processor, then, the obvious choice is to have one input array comprising the positions, and one output array comprising the new accelerations. The constants were compiled into the shader program source using the provided JIT compiler at program initialization.

We set up the GPU to execute our shader program exactly once for each location in the output array, i.e. each shader program calculates the acceleration for one atom. This shader program scans the entire input array, i.e. all the atom positions, for atoms close enough to interact, and accumulates their contributed forces into a single acceleration value. After the GPU is finished, the resulting accelerations are read back into main memory where the CPU proceeds with the current time step. At the next time step, the updated positions are re-sent to the GPU and new accelerations computed again.

There is one complexity here; the potential energy (PE) of the system is calculated every time step, and every interacting atom pair contributes to this sum. It is most naturally calculated as part of the acceleration computation, where the interactions from each atom are accumulated into a single potential energy value. However, in the GPU programming paradigm, there is no

communication between the executing instances of the shader programs, so a sum across all atoms directly on the GPU is impossible in a single pass. One option is to introduce one or more additional passes to accumulate each atom's contribution to the total PE in a gather-type fashion, called a reduction operation. However, this method introduces significant overheads. Instead, since we must perform a readback from the GPU to retrieve the accelerations anyway, it makes more sense to simply read back each atom's contribution to PE as well and sum them in linear time on the CPU, which is well suited to this scalar task. There is a subtlety which makes this even less expensive: the accelerations are 3-component vectors, but on a GPU we must use 4-component arrays. Thus, we can simply store each atom's PE contribution in the fourth component, and when we read back the accelerations these values are retrieved for free.

Figure 7 shows performance results using an NVIDIA GeForce 7900GTX GPU versus a 2.2GHz Opteron. There is a startup cost associated with the GPU implementation; however, it is a fraction of a second, and since it occurs only once it will be quickly amortized across the time steps for any non-trivial runtime, so it is not included in these results.

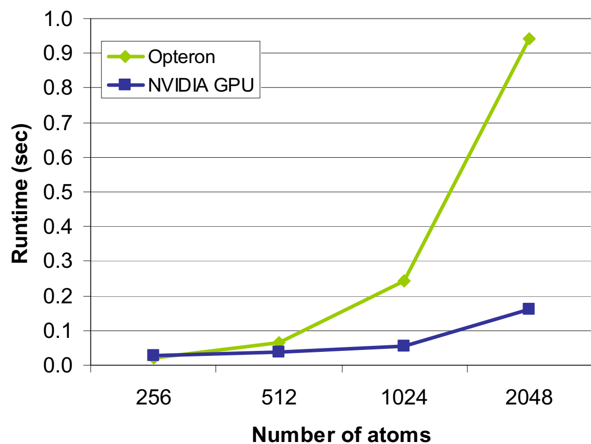

Figure 7: Performance results on GPU

However, there are other constant and $O(N)$ costs associated with each time step on the GPU, and these costs are included. These include sending the position array and reading the acceleration array across the PCIe bus every time step. It is these costs which make the GPU implementation take longer to run than the CPU version at very small numbers of atoms, despite the massive parallelism of the GPU we use to speed up the actual computation of the accelerations. For a run of 2048 atoms, the GPU implementation is almost 6x faster than the CPU.

5.3 The MTA-2 system

The MTA-2 architecture provides an optimal mapping to the MD calculations because of its uniform memory latency architecture. In other words, there is no penalty for accessing atoms outside the cutoff limit or the cache boundaries, in an irregular fashion, as in the microprocessor-based systems.

Nevertheless, the most time consuming part, i.e. step 2 of the kernel, was not automatically parallelized by the MTA compiler because it found a dependency on the reduction operation. The rest of the kernel is parallelized by the MTA compiler without any code modification. In order to parallelize calculations in step 2, we moved the reduction operation inside the loop body. Moreover, we hinted the compiler using an MTA directive that the loop has no dependencies and hence it is parallelizable.

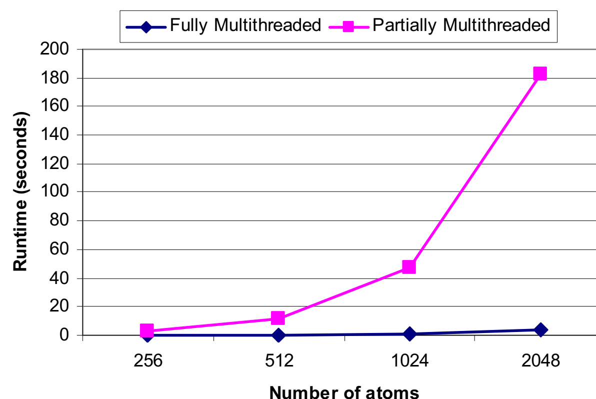

Figure 8: Performance comparison of fully vs. partially multithreaded versions of the MD kernel

Figure 8 shows the performance difference before and after the code modification. The figure also shows the importance of exploiting the multi-threaded feature of the MTA system, since the performance of the fully multi-threaded version can be significantly higher than of that of a partially multithreaded version for a similar application. In case of the MD kernel, the performance difference increases with the increase in the number of atoms in the system.

We then compared performance of the optimized version with a contemporary Opteron processor. Note that the clock speed of the 200 GHz MTA-2 system is about 11x slower than the 2.2 GHz Opteron processor. We observe that the runtime on the Opteron processor increases at a relatively faster rate by increasing the number of atoms in the system as shown in Figure 9. In other words, the effect of cache misses are shown in the Opteron processor runs as the array sizes become larger than the cache capacities of the Opteron processor. The increases in the MTA runtime, on the other hand, are

proportional to the increase in the floating-point computation requirements.

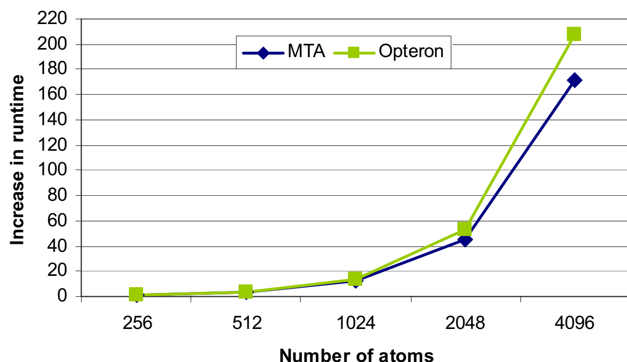

Figure 9: Increase in runtime with respect to simulation run with 256 atoms

6 Conclusions and Future Plans

We investigated and explored the performance attributes of emerging, high-performance processing devices for bio-molecular MD calculations. We identified that the traditional micro-processor optimization and mapping strategies are not applicable to systems like Cell and GPU that require explicit data management and control. Although these requirements resulted in additional porting and optimization effort, we showed a 5x to 6x gain in performance of the total runtime from these architectures. The MTA-2 architecture was relatively straightforward to program, but did not show similar performance gains; it, on the other hand, demonstrates an improved workload scaling behavior. We anticipate significant performance gains from the upcoming XMT technology, however. In conclusion, the three devices are capable of providing supercomputing-scale power to biological simulations users that have access to desktop and small cluster systems. Currently, the outstanding issues are the availability and support for double-precision floating-point calculations and a standard programming interface to these diverse set of high-performance computing platforms. We plan to investigate the performance potential of these devices for full-scale bio-molecular simulation frameworks using high-level language interfaces.

Acknowledgements

The submitted manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

References

[1] Cray Inc. “Cray XMT Platform”, available at http://www.cray.com/products/xmt/index.html

[2] Cray Inc., “Cray MTA-2 Computer System User's Guide,” Cray Inc. S-2317-10, 2005.

[3] Cray Inc., “Cray MTA-2 Programmer's Guide,” Cray Inc. S-2320-10, 2005.

[4] Cray Inc., “Cray MTA-2 System - HPC Technology Initiatives,” http://www.cray.com/products/programs/mta_2/, 2006.

[5] GPGPU (General Purpose computation using GPU hardware, http://www.gpgpu.org/

[6] GROMACS, http://www.gromacs.org/

[7] International Business Machines Corporation, “Cell Broadband Engine Programming Tutorial Version 1.0,” 2005.

[8] http://www.nvidia.com

[9] S. R. Alam, P. K. Agarwal, *et. al.*, “Performance Characterization of Bio-molecular Simulations using Molecular Dynamics,” *ACM Symposium of Principle and Practices of Parallel Programming*, 2006.

[10] S. Bokhari and J. Sauer, “Sequence alignment on the Cray MTA-2,” *Concurrency and Computation: Practice and Experience (Special issue on High Performance Computational Biology)*, 16(9):823–39, 2004.

[11] I. Buck, “Brook—Data Parallel Computation on Graphics Hardware,” *Workshop on Parallel Visualization and Graphics*, 2003.

[12] J. Feo, D. Harper *et al.*, “ELDORADO,” *Conference on Computing Frontiers*. Italy: ACM Press, 2005.

[13] B. Flachs, S. Asano *et al.*, “The microarchitecture of the synergistic processor for a cell processor,” *IEEE Journal of Solid-State Circuits,* 41(1):63-70, 2006.

[14] O. Hwa-Joon, S.M. Mueller *et al.*, “A fully pipelined single-precision floating-point unit in the synergistic processor element of a CELL processor,” *IEEE Journal of Solid-State Circuits,* 41(4):759-71, 2006.

[15] J. A. Kahle, M.N. Day *et al.*, “Introduction to the Cell Microprocessor,” *IBM Journal of Research and Development*, 49(4/5):589-604, 2005.

[16] A. R. Leach, Molecular modeling: principles and applications, 2nd ed: Prentice Hall, 2001.

[17] W. Liu, et. al. “Bio-Sequence Database Scanning on a GPU”, *IEEE International Workshop on High Performance Computational Biology*, 2006.

[18] Y. Liu, et. al. “GPU Accelerated Smith-Waterman.” *International Conference on Computational Science*, 2006.

[19] S. Williams, J. Shalf et al., “The Potential of the Cell Processor for Scientific Computing,” *Proc. of Computing Frontiers*, 2006.