Porting the GROMACS Molecular Dynamics Code to the Cell Processor

Stephen Olivier[1], Jan Prins[1], Jeff Derby[2], Ken Vu[2]

[1]University of North Carolina at Chapel Hill
Dept. of Computer Science
Chapel Hill, NC 27599-3175 USA
{olivier, prins}@cs.unc.edu

[2]IBM Systems and Technology Group
3039 Cornwallis Rd.
Research Triangle Park, NC 27709 USA
{jhderby, kenvu}@us.ibm.com

Abstract

The Cell processor offers substantial computational power which can be effectively utilized only if application design and implementation are tuned to the Cell architecture. In this paper, we examine application characteristics which facilitate efficient use of the Cell processor, and those which present obstacles to it. Moreover, we consider possible solutions designed to mitigate inefficiencies. The target application in our study is the GROMACS molecular dynamics package. We have accelerated the most-often used compute-intensive kernel while maintaining the constraints imposed by the structure of the surrounding program. The significant contribution of this paper is the consideration of the kernel in the context of a complex end-to-end application, with irregular data and code patterns, rather than an isolated kernel code. For this challenging scenario, our results show a 2X speedup versus hand-tuned VMX/SSE code running on high-end PowerPC and x86 uniprocessor machines.

1 Introduction

Sony, IBM, and Toshiba (STI) collaborated to develop the Cell Architecture based on IBM's PowerPC technology. Parallelism and high-speed communication are key features of this technology which can accelerate many applications to high levels of performance. Although it was designed with gaming and multimedia processing in mind, other uses have been considered, including scientific computing. Williams et al. investigated the performance of Cell on several benchmark kernels for dense matrix multiplication, sparse matrix vector multiplication, stencil computations, and Fast Fourier Transforms (FFT) [13].

1-4244-0910-1/07/$20.00 ©2007 IEEE.

Molecular Dynamics (MD) belongs to the class of *n-body* scientific problems, which track the evolution of a system of particles based on the interactions between them. Classical MD is based on Newtonian mechanics of atoms in a system of molecules and is accurate enough for many uses of MD. In this study, we report on the porting to the Cell processor of a widely-used classical MD program, the GROningen MAchine for Chemical Simulation (GROMACS) [12, 9, 1].

Like many scientific codes, the evolution of GROMACS has been influenced by advances in both computing and the application domain, computational chemistry. The functionality enabled by its many lines of code is dependent on a mathematical model and an underlying representation that is used extensively throughout the software. Substantial change is undesirable, both because it will require extensive and complex modifications, and because it may interfere with the subsequent integration of scientific improvements into the code. In this study, we have maintained the constraints that arise from the structure of the program and its data. The integrity of the program is preserved, and in this respect, we offer a more complete investigation of the issues that arise when porting an entire real-world application to the Cell processor.

2 The Cell Processor

In its first incarnation, the Cell processor [7] features nine cores. One core, the Power Processor Element (PPE) consists of a 64-bit Power Processing Unit (PPU) with L1 and L2 caches. The other eight cores are Synergistic Processing Elements (SPE), each consisting of a Synergistic Processing Unit (SPU) designed for high performance compuation on 128-bit vectors and a Local Store (LS). An SPU has 128 128-bit registers and is capable of dual issue. The LS is a high-speed 256KB memory for both code and data used by an SPU; the SPU may execute loads and stores only upon the contents of the LS. DMA is used to transfer data

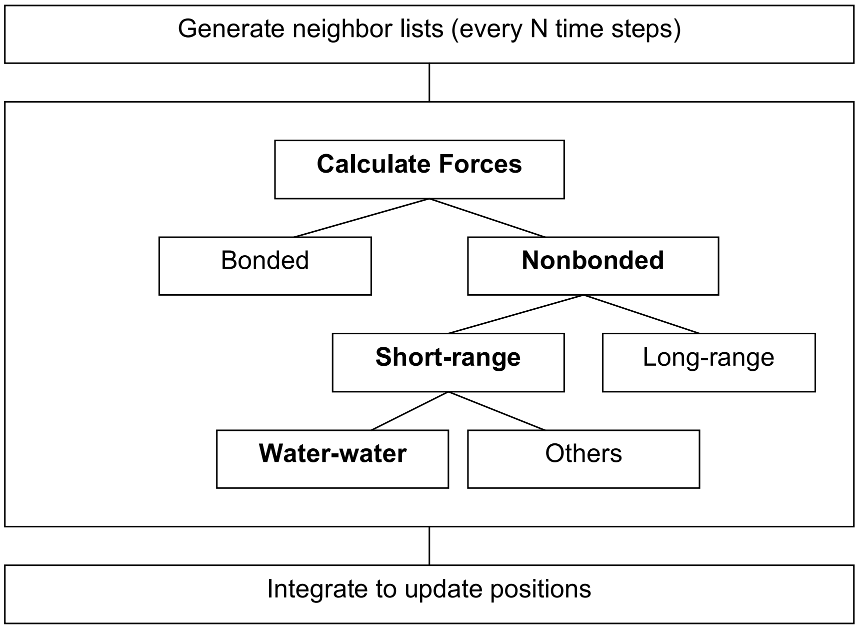

Generate neighbor lists (every N time steps)
Calculate Forces
Bonded
Nonbonded
Short-range
Long-range
Water-water
Others
Integrate to update positions

Figure 1. Tasks performed in a single time step of a GROMACS simulation. The kernel we optimized calculates forces due to short-range nonbonded interactions between water molecules.

between memory and the LS, between two SPEs, or between the PPE and an SPE. The Element Interconnect Bus (EIB) provides fast on-chip communication among the PPE, SPEs, and memory controller via four high speed rings.

There are some features of the chip which are of particular importance to programmers. The SPUs are quite adept at single-precision (SP) floating point, but significantly slower at double-precision (DP). The SPUs lack hardware branch prediction, so branches should be avoided or hinted well. Since the SPU operates on vectors, they should be used wherever possible. Since an SPE's DMA unit operates independent of the SPU, data transfers can occur simultaneously with computation. Signal notification and mailbox registers, and their accompanying access commands allow for passing of short messages among the PPE and SPEs; these are especially useful for synchronization.

The Cell processor offers significant potential speedup compared to a conventional scalar single-core processor running at the same clock speed. SPE vector operations offer a 4X speedup for single precision floating point operations. Using all eight SPEs yields an 8X speedup on top of that, for a total of 32X speedup versus a typical scalar single-core processor. Additional time may be saved by overlapping computation and data tranfers, using the EIB for SPE-to-SPE data transfer, and making good use of the low latency LS. However, contention for memory and other resources, thread coordination overheads, and load imbalance are just a few reasons peak performance may not be reached.

3 GROMACS

GROMACS is highly optimized for uniprocessor execution, with hand-tuned code, assembly loops, and manual loop unrolling. Commonly used and compute-intensive portions of the program have been optimized for SSE on Intel PCs and Altivec/VMX on PowerPCs. While it supports a DP floating point mode, it is often used in SP mode. The program reads in the initial configuration of the system from disk, then calculates the interactions between atoms and updates their coordinates over a series of time steps, periodically recording the results to disk.

Within each simulation step, several computational tasks must be completed. Bond forces must be calculated between bonded atoms. Electrostatic and van der Waals forces between nonbonded atoms must be calculated based on their positions and charges. Atom velocities and positions, as well as the enegry, pressure, and temperature of the system, must be updated based on the forces. Figure 1 represents a simplified view of the tasks performed in each simulation step, including a break-down of the different force calculations.

Nonbonded force calculations often dominate the running time, since each atom is only bonded to at most a few others. Calculating pair-wise forces between all nonbonded pairs of atoms in the system would take O(n^2) time. Instead, GROMACS computes forces upon each atom from other atoms within a certain cut-off radius. Long-range forces are calculated using particle-mesh methods.

3.1 Code Targeted for Optimization

GROMACS employs several dozen kernels for the short-range nonbonded interactions, each offering a different combination of methods for electrostatic and van der Waals forces. Many are optimized for interactions between water molecules. A common technique of MD is to simulate a protein in a box of water, resulting in many nonbonded interactions between water molecules. In the benchmark simulation configuration for the Villin headpiece [6], over 90% of the system is water. Eighty-three percent of the time in that simulation is spent in nonbonded kernel 112, which is optimized for calculating interactions between water molecules. Other simulations also spend the majority of their run time in that kernel. We chose it as the target of our enhancements for execution on the Cell processor.

We use the Villin headpiece benchmark as a the primary sample input for development and testing, including the tests for performance results presented in Section 5. This simulation spans 5000 time steps. The system consists of the Villin protein and 3,000 water molecules (9000 atoms), for a total of about 10,000 atoms.

3.2 Constraints Imposed by the Code

As explained in Section 1, we leave in place the constraints imposed by the code and data structures of the GROMACS program as a whole. A key constraint is the neighbor list generation. The composition of the neighbor lists determines which groups of atoms will interact in the simulation, in which kernels, and in which order. We refrain from redistributing or reordering the neighbor lists for several reasons: to avoid disturbing the kernels which we are not accelerating, to keep program complexity to a minimum, and to avoid the added cost of reorganizing the data in memory. The following segment of pseudocode demonstrates how the contents of the neighbor lists drive the inner loops at the heart of our computational kernel:

```
$outerlist = i_0, i_1, i_2, ...$
$innerlist[0] = j_{00}, j_{01}, j_{02}$
$innerlist[1] = j_{10}, j_{11}, j_{12}, j_{13}, j_{14}, j_{15}, j_{16}$
$innerlist[2] = j_{20}, j_{21}, j_{22}, j_{23}, j_{24}$
...
For each molecule $i$ in $outerlist$
  For each molecule $j$ in $innerlist[i]$
    Calculate forces
```

Note that the trip count of the inner loop is variable, and dependent on the number of j molecules with which each i molecule interacts. This irregularity presents difficulties for code vectorization.

The neighbor lists also reflect the fact that GROMACS employs the Newton's Third Law of Motion, calculating the force between two atoms only once. While this cuts the running time in half, it also limits the possibility of spatial decomposition and introduces the possibility of concurrent writes. The latter is the subject of Section 4.2, considering a multi-SPE version in which forces are evaluated in parallel.

Also complicating the vectorization of the code is the inherent three-dimensional nature of the MD problem, which must be mapped to four-element vector registers in the Cell SPU. This issue is addressed in Section 4.1.2.

4 Porting GROMACS to Cell

We chose an iterative approach to port the code to the Cell processor. We first compiled GROMACS for the PPE, which required no changes to the code, and confirmed that it runs successfully. After that, we extracted the portion of the code targeted for enhancement, kernel 112. GROMACS provides optimized VMX, SSE, and scalar versions of this kernel. We created a test environment for the kernel based on its interface to the rest of the GROMACS program and the data structures it uses. We ported the kernel to the SPE, made some improvements, and then parallelized across the SPEs. We made improvements on the code through testing on the Cell system simulator and a 2.4 Ghz Cell blade.

4.1 PPE to Single SPE

When moving code from the PPE to the SPE, some key distinctions between them must be kept in mind. One is that the instruction sets, and low-level intrisics in code which depend upon them, are not the same. The VMX vector permute intrinsic, for example, does not map directly to an SPU intrinsic. Another is that all data and code needed for processing on the SPE must reside in the SPE LS, and DMA calls must be used to move data into and out of the LS. These issues must be addressed for code to run at all on the SPE.

We had two kernel implementations from which to choose as a starting point for our SPE port. The VMX code, hand tuned for exactly 32 registers, uses conditionals within the loops and VMX intrinsics for vector permutes and shifted loads. It calculates interactions for four pairs of water molecules at a time. The SPE does not support the VMX permute load instruction, lacks hardware branch prediction to efficiently execute conditionals in code, and has 128 registers rather than 32. For these reasons, the VMX

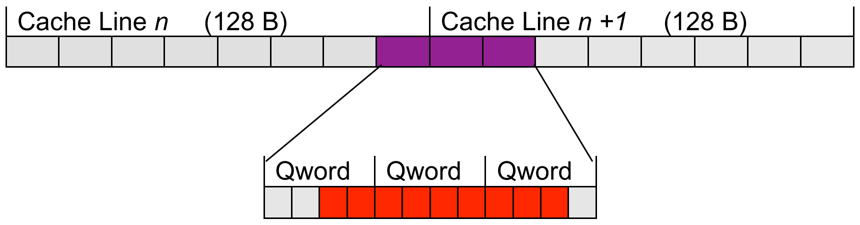


Figure 2. Unused data from DMA transfers: the worst case scenario. Of the 16 quadwords transfered in the two cache lines, 13 quadwords are fetched but not used. Among the remaining three quadwords (48B) fetched, only 36B are actually used while the other 12B are not. This problem does not occur in the later versions where data for all atoms is transfered in at once.

code is not well suited for the SPE. The scalar code is much simpler and shorter, only calculating interactions for two water molecules at a time and using no VMX-specific intrinsics. We decided to start from the scalar code, vectorizing "from scratch" and adding the needed DMA calls between LS and memory.

4.1.1 Dealing with Data Layout

The main data elements are the positions and forces of the atoms in the water molecules, which are stored separately in two large arrays. A rectangular 3D coordinate system is used, and the components x, y, and z for each position or force are stored contiguously in the array. Recall that our kernel handles interactions between water molecules. Data for the three atoms which comprise each water molecule are also contigous. A complete set of forces or positions for a water molecule appears as $O_xO_yO_zH_xH_yH_zH_xH_yH_z$ in the appropriate array. In SP floating point, that constitutes 12B per atom and 36B per molecule in each of the two arrays (the force and position arrays). On most machines, this is of little consequence, but on Cell it is quite significant.

Addresses for data to be transferred by DMA into the SPE LS must be aligned to quadword (16B) boundaries, and data is fetched from memory one cache line (128B) at a time. Thus, to get the 36B of data in the atom or force array, we will have to bring in 12B of extra data just to avoid a bus error on the DMA request, and 92B or 220B of extra data are actually transferred, as illustrated in Figure 2. There would be no problem if the data were used sequentially, but that is not the nature of the kernel. Data accesses may occur in any order.

Within each simulation step, the force calculations are done by a set of two nested loops. The outer loop iterates though index arrays that point to the position and force data for a molecule i and a group $j_1..j_n$ of molecules with which i interacts. Another index array points to position and force data for the j atoms. The index arrays are constructed by the rest of the GROMACS program, based on other conditions in the system, and updated periodically. Therefore, it is typical that data for two consecutive i or j atoms may be far from each other in memory. Thus, for now, we initiate separate DMAs for each new i or j atom force set and position set.

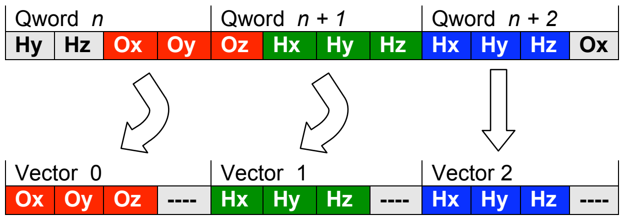
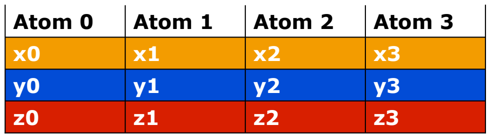


Figure 3. Shuffling data from the Local Store into the SPE's vector registers.

Once the data is transferred, it is extracted from the surrounding data and moved into vectors. These also are 16B long, another mismatch with our data and a source of extra work. Unless we change the memory layout of the complete GROMACS program to pad the arrays, we must perform this extra work. The SPU's shuffle operation is used to correctly position the 36B needed into the vector registers of the SPE. An example shuffle is shown in Figure 3.

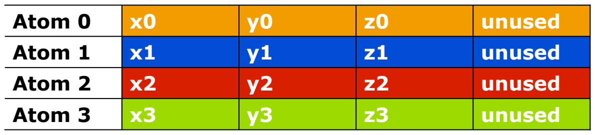
4.1.2 Vectorization

There are two options to vectorize the data and the code that processes it. The first is to vectorize using the Structure of Arrays (SoA) form. A basic SoA arrangement is illustrated in Figure 4. This arrangement is a poor match with the SPU vector intrinsics, which operate between two vectors (rows in the diagram) rather than across elements of a single vector. One way to exploit the vector operations in an SoA solution would be to interact four molecules at a time. However, code structure irregularities make this difficult to do efficiently. In particular, the inner loop in the original scalar code has a variable trip count. If we set



Atom 0	Atom 1	Atom 2	Atom 3
x0	x1	x2	x3
y0	y1	y2	y3
z0	z1	z2	z3

Figure 4. Vectors in Structure of Arrays (SoA) form.



Atom 0	x0	y0	z0	unused
Atom 1	x1	y1	z1	unused
Atom 2	x2	y2	z2	unused
Atom 3	x3	y3	z3	unused

Figure 5. Vectors in Array of Structures (AoS) form.

up four i molecules in the outer loop, each i may interact with a different number of j molecules. To get the full benefit of the four SIMD elements available, we would have to change some i molecules while leaving others in place. We would in general be operating on four (i, j) pairs, of which some may have the same i atoms and while others do not. This scheme would require flattening the loops, then summing the forces on a single i using a segmented sum, a technique developed for programming large vector machines [2]. Also, such a scheme would require additional shuffles to carry out the computation.

The other option is to vectorize using the Array of Structures (AoS) form, shown in Figure 5. We always interact two molecules at a time, using vector operations to calculate x, y, and z components simultaneously. The code would also be simpler, as it would not need to be concerned with how many trips the inner loop makes. Each inner loop iteration will always see a roughly 3X speedup over the scalar code. However, the last vector slot in each four-element vector would go unused. Thus, at most 75% of the ideal performance boost from vectorization will be realized. However, considering the time spent shuffling extra data in the first method, it also may not achieve the full 4X speedup. Due to that cost and the complexity required for the flattening and segmented sum, we chose the second method, vectorizing across spatial dimensions x, y, and z.

Through an iterative development cycle, we improved our single-SPE code before adapting it for use on multiple SPE's. In our second version of the single-SPE code, we moved the outer loop index arrays and other meta-data into the LS. In the third version, we double-buffered the position data and accumulated the forces for all atoms in the LS, updating the forces in memory after the outer loop is finished. This version served as a starting point for a multi-SPE version.

4.2 Single SPE to Multiple SPEs

Parallelizing across SPEs not only speeds up the execution through concurrent computation, but also increases total available LS space for the problem's data. However, it also introduces the possibility of contention for resources such as the DMA controller and the potential for concurrent writes to the same memory location. The former impacts only performance; the latter may result in incorrect program output. In our kernel, a molecule may be referenced as a j molecule in two neighbor list groups simultaneously, and thus have its force total updated simultaneously by two SPEs! Additionally, there is a need for synchronization between SPEs.

In our multi-SPU kernel, we partition the work of the outer loop (the list of i-groups) across the SPEs. We address the issue of concurrent writes by having each SPE accumulate local force totals for all atoms. These per-SPU totals are then gathered up, summed, and written back to memory. The overall program structure resembles the Bulk Synchronous Processing (BSP) [11] paradigm, where each iteration in the simulation loop consists of successive rounds of computation, communication, and barrier synchronization. Once all of the SPEs have started, a signal from the PPE tells them to begin calculating forces. As each SPE finishes the force calculations, it signals back to the PPE. When all have finished, the PPE signals them to begin gathering up the forces in a sum-reduction pattern, to be described shortly. Finally, each SPE signals back once more to the PPE. When all have reported back, the PPE signals either to begin another simulation step or quit.

The gathering of forces is accomplished in parallel by assigning each SPE a block of atoms for which it is responsible, using a similar partitioning process to that used for the loop partitioning. The last SPE is responsible for some additional data items, the per-shift force totals, in addition to its atoms. We use the high speed EIB rings to pass the force totals from each SPE to the SPEs responsible for each of the blocks, in a series of $N - 1$ rounds, where N is the number of SPEs. In each round R, SPE S retrieves force totals from SPE $(S + R)\%N$ for the atoms which it has been assigned. Once all the rounds are complete, S updates memory with the sum of force totals for its assigned atoms. The potential totals from each SPE are sent in the signals to the PPE, making use of the 32-bit mailbox message lengths, and the PPE sums and updates those in memory.

In our second version of the multi-SPE kernel, we removed some apparent branches from the inner loop, resulting in significantly better performance. This illustrates the point that inefficiencies in single-SPE code are no less important than multi-SPE parallelization issues.

In the third multi-SPE version, we moved all the atom positions into the local stores. Spreading the index arrays

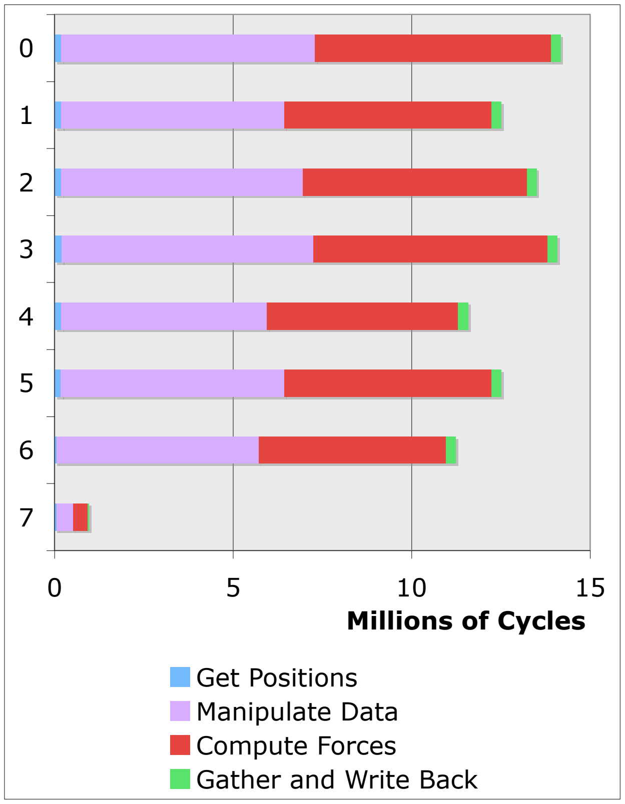


Figure 6. Time spent by SPEs in the different phases of each kernel execution: 1) Getting the atom positions, 2) Manipulating data, 3) Computing the forces, 4) Gathering up the forces and writing them back to memory.

Kernel Version	Time (sec)	Speedup vs. PPE w/VMX
PPE with VMX	137	1.00
Single-SPE v. 1	918	0.15
Single-SPE v. 2	804	0.17
Single-SPE v. 3	158	0.87
Multi-SPE v. 1	28	4.89
Multi-SPE v. 2	20	6.85
Multi-SPE v. 3	16	8.56

Table 1. Run time comparison for the three single-SPE and the three multi-SPE versions of the kernel, with speedup measured against the original VMX version on the PPE. Multi-SPE versions use eight SPEs of a 2.4 Ghz Cell Blade.

Platform	Speed (Ghz)	Time (sec)	Speedup vs. Xeon
Intel Xeon	3.4	186	1.0
PowerPC 970	2.0	135	1.38
Intel Itanium2	1.6	64	2.91
Intel Xeon SSE	3.4	39	4.77
PPC 970 VMX	2.0	31	6.00
Cell (8 SPEs)	2.4	16	11.6
Cell (8 SPEs)	3.2	12	15.5

Table 2. Run time comparison for the kernel on Cell versus other platforms. Kernel code SSE and VMX was hand-optimized, including manual loop unrolling. The SSE version is assembly code. Run time for the 3.2 Cell BE is estimated.

across the SPEs freed the space needed to do this. With both the forces and the positions now in quickly accessible LS space, many cycles spent reloading previously-used data are saved. Figure 6 shows that at most 10% of the time in each outer loop iteration is spent transferring data into and out of the SPE LS. The figure reveals that there is some load imbalance among the SPEs, which is due to the coarseness of our work partitioning. We partitioned solely based on the outer loop, dividing it such that each partition's data set would begin on a 128B cache-line boundary.

5 Results and Analysis

We tested our code using the system simulator and a 2.4 Ghz Cell blade prototype throughout the porting and enhancement process, and compared the final version to other leading platforms. Table 1 shows the performance of the successive versions of the kernel compared to the original VMX version on the PPE. The single SPE versions were slower than the original PPE version, largely due to the time required to DMA the atom data and shuffle it into the correct vector positions. The third single SPE version saw a great improvement by double-buffering the position data and keeping the force data in the LS. Even the first multi-SPE version outperforms the PPE version, and further enhancements result in a final multi-SPE run time of 16 seconds. This is more than 8X faster than the third single SPE version due to the cost savings from bringing all position data into the SPE LS at once at the beginning of the kernel.

Of course, we must also consider the improvement over other PowerPC and x86 machines. Table 2 compares the performance of our last multi-SPE version to versions of the original code on other machines. The last multi-SPE version on Cell is nearly twice as fast as the next-best performer, the PPC 970 running hand-tuned VMX code. If we (optimistically) estimate that a 3.2 Ghz Cell will offer a proportial speedup over the 2.4 Ghz Cell used in these tests, the run time would decrease to only 12 seconds, over three times as fast as the Xeon running hand-tuned SSE assembly code.

We are encouraged by the results, although the gains are more modest over other machines with vector units, most notably the PowerPC 970. There is no competitive advantage from vector processing in the SPU in those comparisons, so all speedups must be accomplished by parallelizing across SPEs and using the LS and communication features of Cell effectively. Also, the need for PPE processing for other tasks in GROMACS necessitates the reloading and write-back of atom data to and from LS each step, whereas machines with large L2 caches may see all or most data accesses hit in L2 cache. The cost of dealing with misaligned vector loading is a major contributor to less-than-optimal Cell performance: Of the 326 cycles per inner loop iteration in the last multi-SPE version, only 157 cycles are spent on floating point operations. As shown in Figure 6, the remainder of the cycles are spent manipulating data, including shuffling vector elements and reading the inner lists for the neighbor groups.

6 Related Work

Ab-initio (first principles) simulations study MD at quantum-level precision. Car-Parrinello Molecular Dynamics (CPMD), a high performance ab initio MD program, has been experimentally ported to Cell [3].

Folding@Home is a large-scale distributed computing project devoted to protein folding, a major application of molecular dynamics [8]. A modified version of GROMACS is at the core of the x86 and PowerPC software client implementations. In partnership with Sony, Folding@Home will be offering a client for the PlayStation 3, although it is unclear whether that client is based on GROMACS code [10].

There is also an ongoing effort toward compiler optimizations and tools to simplify programming on Cell. Researchers at IBM are refining compiler techniques for automatic workload partitioning and local store buffer allocation and management on the Cell BE [5, 4].

7 Conclusions and Future Work

GROMACS is representative of many scientific applications in that it has been highly optimized for sequential execution and exhibits irregular data access patterns and code structure. This makes it difficult to implement efficiently on Cell. We have achieved speedup on the water interaction kernel 112, the most-often used in GROMACS using vectorization, parallelization across SPEs, the quick access LS, and the high speed EIB rings. There is a significant potential for improvement by more efficiently handling the data manipulation (particularly shuffle operations) in the LS. We are investigating better use of shuffle patterns and code ordering for software pipelining, especially since the SPU is a dual-issue processor.

In terms of Cell developer enablement, several tools would be helpful for carrying out a similar porting task in future. Ensure that all code and libraries are branchless if possible. Provide library functions for double buffering and for efficient collective operations across SPEs such as gather, scatter, and sum reduction. Difficulty handling misaligned data and irregular data accesses could be helped by utilities for efficient index array usage and indirection. Writing back to memory and reloading LS is costly but inevitable if the parts of the application run in the PPE or coherency is needed between SPEs. This should be considered when examining an application's suitablility for Cell.

GROMACS as a whole consists of dozens of kernels and several thousand lines of code. Optimizing the entire program for the SPEs could potentially take several man-years. Since we have optimized only one kernel, the application as a whole would not see as large a speedup. Overhauling the whole application could also help to create code and data layout better suited to the Cell processor, but we feel that the effort required would be prohibitive. Optimizing a handful of the frequently used kernels would be reasonable. Optimization for Cell can be time-consuming, but incremental improvements do lead to better and better performance and are well positioned to take advantage of future increases in the number of SPEs.

References

[1] H. J. C. Berendsen, D. van der Spoel, and R. van Drunen. GROMACS: a message-passing parallel molecular dynamics implementation. *Computational Physics Communications*, 91(1-3):43–56, Sept. 1995.

[2] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, 1990.

[3] A. Curioni. Ab-initio molecular dynamics on cell processor: Early experiences. In *First Workshop on Real Time and Interactive Digital Media Supercomputing (RIDMS-1)/12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, Feb. 2006.

[4] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. *IBM Syst. J.*, 45(1):59–84, 2006.

[5] A. E. Eichenberger, K. M. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind.

Optimizing compiler for the cell processor. In *IEEE PACT*, pages 161–172, 2005.

[6] GROMACS. GROMACS benchmarks for MD, 2005. [Online] http://www.gromacs.org/gromacs/benchmark/benchmarks.html.

[7] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Systems Journal*, 49(4), 2005.

[8] S. M. Larson, C. D. Snow, M. Shirts, and V. S. Pande. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology. *Modern Methods in Computational Biology, Horizon Press*, 2003.

[9] E. Lindahl, B. Hess, and D. van der Spoel. GROMACS 3.0: A package for molecular simulation and trajectory analysis. *Journal of Molecular Modelling*, 7:306–317, 2001.

[10] V. S. Pande. Folding@Home on the PS3 FAQ, 2006. [Online] http://folding.stanford.edu/FAQ-PS3.html.

[11] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[12] D. Van Der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark, and H. J. Berendsen. Gromacs: fast, flexible, and free. *Journal of Computational Chemistry*, 26(16):1701–1718, Dec. 2005.

[13] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *ACM International Conference on Computing Frontiers*, May 2006.