Direct N-body Kernels for Multicore Platforms

Nitin Arora
School of Aerospace Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332-0150
n.arora@gatech.edu

Aashay Shringarpure
School of Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332-0765
aashay.shringarpure@gatech.edu

Richard W. Vuduc
Computational Science
and Engineering Division
Georgia Institute of Technology
Atlanta, Georgia 30332-0765
richie@cc.gatech.edu

***Abstract*—We present an inter-architectural comparison of single- and double-precision direct n-body implementations on modern multicore platforms, including those based on the Intel Nehalem and AMD Barcelona systems, the Sony-Toshiba-IBM PowerXCell/8i processor, and NVIDIA Tesla C870 and C1060 GPU systems. We compare our implementations across platforms on a variety of proxy measures, including performance, coding complexity, and energy efficiency.**

I. Introduction

The goal of this study is to understand the differences in the implementation techniques required for direct (particle-particle) n-body simulations to achieve good performance on a variety of modern multicore CPU and accelerator desktop platforms. These platforms include an 8-core (dual-socket quad-core) 2-way/core multithreaded Intel Nehalem processor-based system (16 effective threads); a 16-core (quad-socket quad-core) AMD Barcelona-based system; IBM dual-socket QS22 blade based on the Sony-Toshiba-IBM PowerXCell/8i processor; and NVIDIA Tesla C870 and C1060 graphics processing unit (GPU) systems.

We began the study with a focus on direct n-body computations primarily because of our interest in this application domain. The structure of the computation is a fundamental building-block in larger applications [1], [2] as well as approximate hierarchical tree-based algorithms for larger systems, *e.g.*, Barnes-Hut or fast multipole method (FMM) [3]–[5]. Moreover, lessons-learned in implementing these kernels for physics applications readily extend to new application domains in statistical data analysis, search, and mining [6].

We initially believed it would be simple to achieve near-peak performance on these platforms. The key computational bottleneck is the $O(n^2)$ evaluation of pairwise interaction forces among a system of n particles. This kernel is highly regular and floating-point (flop) intensive, and therefore well-suited to accelerators like GPUs or PowerXCell/8i. However, to our surprise, each implementation required more significant tuning effort than expected. We describe these implementations and some "lessons-learned" in this paper. As far as we know, ours is among the first comprehensive *cross-platform* multicore studies for computations in this domain, and in particular unique in its contrasting of GPU and Cell-based accelerated systems.

Given that we consider only the $O(n^2)$ direct evaluation, our study will be limited in several ways. First, we achieve and highlight the best performance at large values of n. Small values of n, which are of great interest in, for instance, hierarchical tree-based approximation n-body algorithms, may require a different emphasis on low-level tuning techniques. Secondly, the computation is compute intensive with largely regular access patterns, and so we do not stress the memory system. Still, we believe this study can make a useful contribution to other computations more generally. In the numerical domain, hardware reciprocal square root is typically fast for single precision but not double, making our performance far from peak even though the computations are relatively compute-bound. Furthermore, for our accelerator architectures, we still need to carefully manage the local store to make effective use of available memory system resources. This points to non-trivial implementation trade-off considerations across multicore architectures, in further support of recent work [7].

II. Related Work

Parallelization of direct n-body problems is well-studied [8], [9], and has a long history that includes custom hardware (*e.g.*, GRAPE systems). Most recently, there has been one thorough study for x86-based CPU tuning [10], as well as several studies on GPUs both prior to [11], [12] and following [13], [14] the introduction of the high-level CUDA model, which we use in the present study. The post-CUDA direct n-body GPU applications show particularly impressive performance, with comparison to existing custom hardware used for n-body simulations, like the GRAPE-6AF. These prior GPU studies focus primarily on single-precision kernels, as that was the only precision available in hardware at the time. To our knowledge there have not been any published n-body implementation performance studies for the STI Cell/B.E. processor family. This work tries to fill these comparison gaps by including both GPUs and STI Cell/B.E. platforms, as well as multicore CPU systems.

III. Direct N-body Implementations

We implemented various parallel versions of a simple n-body gravitational simulation, using direct (particle-particle) $O(n^2)$ force evaluation. In particular, we numerically integrate the equations of motion for each particle i,

$$\vec{F}_i = -Gm_i \sum_{\substack{1\leq j\leq n \\ j\neq i}} m_j \frac{\vec{r}_i - \vec{r}_j}{||\vec{r}_i - \vec{r}_j||^3} \tag{1}$$

where particle i has mass m_i, is located at the 3-D position $\vec{r}_i$, and experiences a force $\vec{F}_i$ from all other particles; G is the universal gravitational constant, which we normalize to 1. We use the Verlet algorithm for our numerical integration scheme.

A. Characteristics, costs, and parallelization

The bottleneck is the $O(n^2)$ force evaluation, which computes the acceleration for all bodies i (with $G = 1$):

$$\vec{a}_i \approx \sum_{1\leq j\leq n} m_j \frac{\vec{r}_i - \vec{r}_j}{(||\vec{r}_i - \vec{r}_j||^2 + \epsilon^2)^{\frac{3}{2}}} \tag{2}$$

Here, we adopt the common convention of a "softening parameter" term, ϵ^2, in the denominator, to improve the overall stability of the numerical integration scheme [2]. Computing the acceleration dominates the overall simulation cost, so that in this paper we focus on its parallelization and tuning for accelerators, and may effectively ignore the cost of the numerical integrator.

The scalar pseudocode for Equ. (2) can be written as follows (comments prefixed by "//"):

```
1: for all bodies $i$ do
2:   // Load $(x_i, y_i, z_i)$ here
3:   $\vec{a}_i \equiv (a_x, a_y, a_z) \leftarrow (0, 0, 0)$ // Init acceleration
4:   for all bodies $j \neq i$ do
5:     // Load $(x_j, y_j, z_j)$ here
6:     $(\Delta x, \Delta y, \Delta z) \leftarrow (x_i - x_j, y_i - y_j, z_i - z_j)$
7:     $\gamma \leftarrow (\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2 + \epsilon^2$
8:     $s \leftarrow m_j / (\gamma \cdot \sqrt{\gamma})$ // $\gamma^{\frac{3}{2}} = \gamma\sqrt{\gamma}$
9:     $(a_x, a_y, a_z) \leftarrow (a_x + s \cdot \Delta x, a_y + s \cdot \Delta y, a_z + s \cdot \Delta z)$
10:  end for
11:  // Store acceleration
12:  $\vec{a}_i \leftarrow (a_x, a_y, a_z)$
13: end for
```

Lines 5–9 compute a single pairwise interaction. As written, there are 18 floating-point operations: 3 subtractions (line 6), 3 multiplies and 3 adds (line 7), 1 multiply, 1 square root, and 1 divide (line 8), and 3 multiplies and 3 adds (line 9). For consistency in comparing flop-rates to other papers, we consider the "cost" of lines 5–8 to be 20 flops, *i.e.*, we count 20 flops per pairwise interaction, or $20n(n-1)$ flops total.

There are $4n^2+6n$ loads and stores, with the dominant term coming from the loads of the 3 coordinates on line 5–6 and mass on line 8. However, there is a significant amount of reuse, so that with appropriate cache blocking we can incur close to the minimum of $4n$ compulsory misses, so that the overall computation should be compute-bound with a computational intensity of $\approx 20n^2/4n = 5n$ flops per word.

For all platforms, we adopt the standard approach to parallelization that exploits both coarse-gained parallelization of the outermost i loop, followed by finer-grained data-parallelism across the i and j loops. We may visualize the general

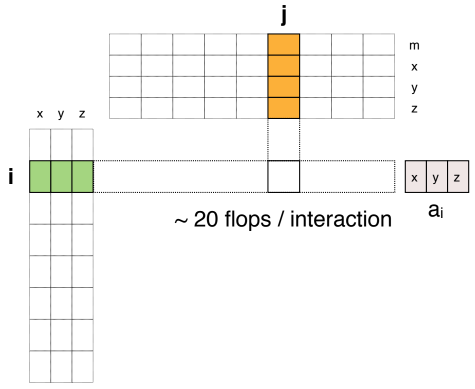


Fig. 1. Computation and data access pattern for direct force evaluation

computation and data access pattern as shown in Figure 1. The n 3-D points appear as the $n \times 3$ matrix on the left. The i loop iterates over these points (rows); for each i, we stream over the same points j, shown mirrored (transposed) along the top, and accumulate an acceleration vector $\vec{a}_i$. The simplest coarse-grained parallelization is an owner-computes approach that partitions the the i loop into n/p chunks among p threads. This approach requires no synchronization on writes to the final acceleration matrix.

B. x86 Implementations

For the general-purpose multicore platforms (2×4-core×2-threads/core=16 thread Intel Nehalem and 4×4=16-core AMD Barcelona systems), we consider several "standard" optimization techniques. These techniques are well-known but essential to implement if we wish to fairly evaluate the accelerator-based GPU and PowerXCell/8i platforms.

First, we exploit coarse-grained owner-computes parallelization at the outermost loop of the interaction calculation. Each thread computes the forces on a subset of $\frac{n}{p}$ particles. In this decomposition, all writes are independent.

We consider both OpenMP and Pthreads programming models, though we expect that OpenMP-parallelization will be sufficient since the kernel is largely compute-bound. In the case of Pthreads, we create a team of threads at the beginning of program execution. These threads busy-wait at a barrier for a signal from the master thread to begin the interaction calculation. Upon completion, the threads again meet at a barrier. (Reported performance includes the time of these barriers.)

Secondly, we apply cache-level blocking of both loops. Since the memory footprint scales like $O(n)$ while flops scale as $O(n^2)$, we do not expect this technique to provide much benefit until n becomes very large, since (a) typical L1 latencies for data that resides in L2 are relatively small, and (b) 2 MB or larger L2 and L3 caches today can easily accommodate $n \approx 100,000$ points.

Thirdly, we manually combine data alignment, unroll-and-jam, and SIMD vectorization transformations, to improve register usage and exploit fine-grained data parallelism. In particular, we store the x, y, z coordinates and mass values in separate arrays aligned on a SIMD-friendly boundary. To enable SIMD vectorization, we then perform unroll-and-jam the outer loop and unroll the inner loop. In the single-precision code, the reciprocal and combined reciprocal square-root instruction (`_mm_rcp_ps` and `_mm_rsqrt_ps`). We also use SSE3-based reductions (*e.g.*, `_mm_hadd_ps`) for some parts of the computation. Although the most aggressive compiler considered in this study, the Intel C v11.0, does perform vectorization, we observe significant benefits from manual vectorization ($\approx 2\times$).

C. PowerXCell/8i Implementation

The STI PowerXCell/8i processor architecture consists of a single general-purpose core, the PowerPC Element (PPE), coupled to eight Synergistic Processing Elements (SPEs), each of which has short-vector (SIMD) processors (SPUs) and small 256 KB software-managed local store. The primary purpose of the PPE is to run the operating system and supervise the working of the SPEs by dispatching jobs, and facilitating synchronization. The bulk of achievable performance for the direct n-body kernel comes from fully utilizing the SPEs, and a naïve implementation, properly vectorized by the compiler, gives reasonable baseline performance. However, we find that additional optimization, some very trivial, enable an increase of almost ten times over the naïve code. In this section, we discuss the most significant of these optimizations.

1) Parallelization strategy: We rely on the SPEs for acceleration, using the PPE only for dispatching the SPE threads, performing a single synchronization barrier, and coordinating the overall integrator. We follow the general owner-computes parallelization strategy outlined above. However, since the local-store on each SPE is purely software-managed, we must explicitly coordinate sending and receiving of all other bodies. Since the SPEs themselves update the contents in PPE's main storage (to return the computed accelerations, velocities, and positions), a single synchronization barrier is necessary to enforce consistency. The barrier synchronization is done using mailboxes and is non-blocking in the SPEs. The SPEs may thereby overlap any potential delay with useful computation.

2) Data organization and vectorization: To best exploit the SIMD capabilities of the SPUs, we use the same storage scheme discussed in the CPU case, in which each coordinate of the positions is stored in its own SIMD-aligned array. Where possible, we manually coalesce squaring and addition operations into fused multiply-add (spu_madd) instructions, which execute with half the latency of separate multiply and add (6 vs. 12 cycles). Finally, we write our code to enable all combinations of coordinate data alignments between source and target points (16 combinations in single-precision, 4 in double-precision). We can do this compactly using the SPU hardware rotate instructions to align the outer vector with all possible configurations of the inner vector. These rotate instructions do not execute in the same pipelines as the floating point arithmetic instructions and thus incur virtually no cost.

3) Double buffering the DMA: As already mentioned, data for all the bodies has to be transferred from the PPU to all other SPUs before they can process the information. The STI PowerXCell/8i allows up to 16KB of data to be transferred using the available direct-memory access (DMA) engine between the various local stores. An SPE initiated DMA was preferred here as the DMA would start only when (and as soon as) the SPE needed the data. DMA requests are asynchronous, which allows us to fetch data in small chunks and overlap data transfer with processing. This technique is well-known among STI PowerXCell/8i developers as "double buffering."

Data organization plays a vital role in how the DMA is organized. Our choice of separately storing the x, y, and z coordinates (and mass) in their own arrays (4 in all) makes them easy to prefetch in a unit-stride streaming fashion. A single DMA can be dispatched to bring a chunk of bodies from the PPU to the SPU (and vice versa). Every DMA is now split into four smaller DMAs, to fetch the chunks from four different arrays. We use the scatter-gather DMA support to fetch from each of the four elements in the lists in 16 KB chunks, bringing the total data transferred by a single list operation to 64KB, allowing us to bring in four times as many bodies as before.

As expected, the code is compute-bound and highly tolerant of the relatively small DMA latencies. In principle, it might seem that double buffering could be eliminated altogether. However, the small 256KB local store on each SPU cannot store all the data at once, making double buffering necessary.

4) SPU pipelines: The STI PowerXCell/8i SPUs have two pipelines, with each pipeline executing a different subset of the possible instructions. As a result, achieving peak performance is only possible if the instruction mix is such that the pipelines can be perfectly balanced. The n-body kernel has a particularly bad instruction mix, as it is made up mostly of floating point arithmetic instructions that can only execute in *pipeline 0*. The remaining and relatively fewer number of integer instructions are relegated to *pipeline 1*. In our case, these include the rotate instruction already mentioned along with floating point reciprocal and reciprocal square root estimate instructions.

D. NVIDIA GPU Implementation

1) GPU architecture overview: The NVIDIA Tesla GPU architecture is designed for applications with abundant fine-grain parallelism [15]. The latest G200 architecture consists of 30 multiprocessor "cores" that each support 8 simultaneous hardware threads (240 simultaneous threads in all). Synchronization is allowed within a multiprocessor via a low latency 16 KB local store—called "shared memory" in NVIDIA/CUDA parlance—as well as local register storage. In addition, each multiprocessor also has two automatically cached *read-only* memories, referred to as constant and texture memory. Finally, the GPU has a relatively high latency *device memory*, which is read/write addressable by all multiprocessors. The C1060 has device memory capacity of 4 GB.

The CUDA computing architecture consists of a C-like programming language with extensions that provide the programmer with a fine-grained multithreaded local-store machine abstraction. The CUDA model encourages a large number of threads (on the order of thousands) in parallel, grouped in units known as thread blocks. Each thread block can have at most 512 threads, which can be synchronized via shared memory (16kb).

The GPU architecture can sustain extremely high bandwidth and numbers of threads compared to conventional CPU-based architectures. However, exploiting these capabilities requires that the number of independent thread blocks be large. The existence of many thread blocks helps to hide the memory operations that may decrease the performance of the program.

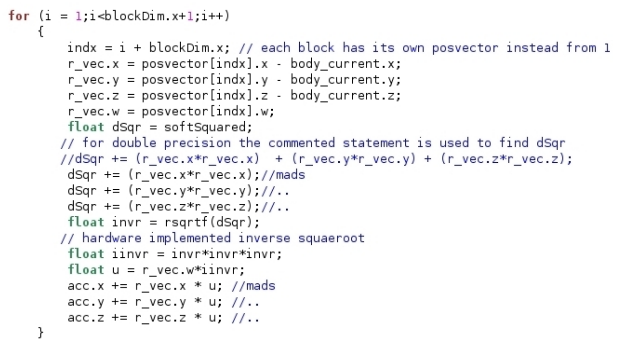
Another important utilization factor on the GPU is register usage. Very high register usage decreases the occupancy (utilization) of each multiprocessor, as it limits the number of threads that can be active simultaneously on the multiprocessor. This may affect the performance of the kernel, especially if the algorithm is not compute-bound. Optimizations like loop unrolling can also affect the performance by increasing or decreasing register usage.

Device-to-shared memory transfers can be a limiting factor. To achieve the highest possible bandwidth from device memory, memory accesses should be *coalesced*, *i.e.*, consecutively numbered threads should access consecutive word-aligned memory locations.

2) Parallelization strategy: We follow the general owner-computes parallelization strategy outlined at the top of Section III, with some refinements.

Our overall strategy follows the one used in the "Gpugems3" example.[1] In particular, we first partition the bodies in chunks of size q. Since we use an owner-computes strategy, this partitions the $O(n^2)$ total work into chunks of size O($q \times n$) each, each of which we assign to a thread block. Within the thread block, we will assign 1 thread to each of the q points, and make it responsible for computing the acceleration (force) due to the other $n-1$ points. We will iterate over the other interaction points q at a time, so that at each step the entire thread block is simultaneously computing q^2 interactions. We synchronize these threads (using `_syncthreads()`) every q^2 interactions.

There are several constraints on q. First, we need q to be a positive integer multiple of the so-called half-warp size (on current NVIDIA GPUs, 16). The warp-size is, effectively, a vector length unit. Secondly, we need the $2q$ points to fit in the thread block's shared memory (local store). In single-precision, that means 3 coordinates + 1 mass at 4 bytes each, or $2 \cdot q \cdot 4 \cdot 4$ bytes $\leq$ shared memory capacity (*e.g.*, a current typical value is 16 KB). Thirdly, we must satisfy the register capacity constraint. For our n-body gravity kernel, each thread has a register working set of approximately 31, so that $q \times 31 \leq$ number of registers (on C870, ≤ 8192, and on the C1070, ≤ 16384). Finally, we cannot have more than a certain number of warps per thread block. With current warp sizes of 32 and the max number of warps at 8, there can be at most $q \leq 32 \cdot 8 = 256$. Within these constraints, we did a limited search to find a good value for q on a given platform.



```
for (i = 1;i<blockDim.x+1;i++)
    {
        indx = i + blockDim.x; // each block has its own posvector instead from 1
        r_vec.x = posvector[indx].x - body_current.x;
        r_vec.y = posvector[indx].y - body_current.y;
        r_vec.z = posvector[indx].z - body_current.z;
        r_vec.w = posvector[indx].w;
        float dSqr = softSquared;
       // for double precision the commented statement is used to find dSqr
       //dSqr += (r_vec.x*r_vec.x)  + (r_vec.y*r_vec.y) + (r_vec.z*r_vec.z);
        dSqr += (r_vec.x*r_vec.x);//mads
        dSqr += (r_vec.y*r_vec.y);//..
        dSqr += (r_vec.z*r_vec.z);//..
        float invr = rsqrtf(dSqr);
       // hardware implemented inverse squaeroot
        float iinvr = invr*invr*invr;
        float u = r_vec.w*iinvr;
        acc.x += r_vec.x * u; //mads
        acc.y += r_vec.y * u; //..
        acc.z += r_vec.z * u; //..
    }
```

Fig. 2. Compute kernel for pairwise interaction

Finally, we design the kernel to expose as many explicit fused multiply-add (fma) instructions as possible, and to expose opportunities to use the hardware reciprocal square-root function, `rsqrtf()`. We show such a kernel in Figure 2.

The unoptimized implementation of the GPU above algorithm was able to reach approximately 65% of the final Gflop/s performance.

3) Optimizing the implementation: We tried a number of optimization techniques, which we enumerate below roughly in decreasing order of effectiveness.

Tuning the optimal block size (number of threads per block): We tried to choose the block size so that $n/q \geq 100$, to ensure enough independent thread blocks were ready for scheduling. On the Tesla C1060, the number of independent blocks that worked well was 120, suggesting that each of the 30 multiprocessors context switches among 4 thread blocks. The number of threads in each block varied from 64 to 512 depending upon the number of bodies being simulated.

Coalescing memory accesses via padding: We aligned float4 data types and C type struct consisting of 4 double variables to achieve padding for the single precision and the double precision implementation respectively. Doing so ensured coalesced shared memory loads from device memory and also coalesced shared memory access within each thread block. The CUDA profiler tool verified this fact.

Loop unrolling: We used the `#pragma unroll` to perform loop unrolling of the compute kernel. This pragma accepts an unrolling depth, which we tuned manually. This technique was particularly effective when the total number of bodies was small. The unrolling factor was a multiple of q and had to be tuned manually.

Encouraging FMAs explicitly: Rather than long expression sequences consisting of multiplies and adds, we broke up these expressions as shown in Figure 2. This technique worked well in single precision. However, for double precision, this technique had the opposite effect.

Register latency tuning: We found that there was actually some benefit to manually reordering statements in our code. We ordered the statements heuristically keeping in mind instruction latencies. For the double-precision computations, we only have one double-precision execution unit per mul-

[1]http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/nbody/doc/nbody_gems3_ch31.pdf

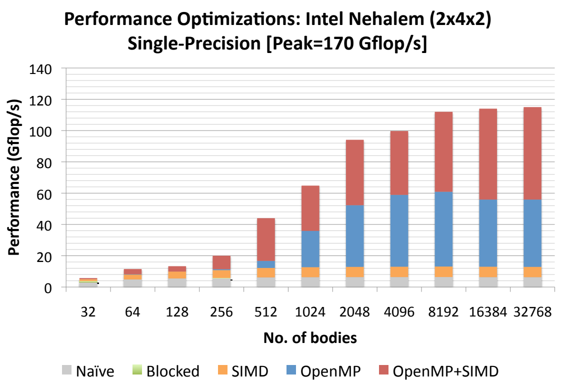


Fig. 3. Optimization breakdown: Single-precision Intel Nehalem

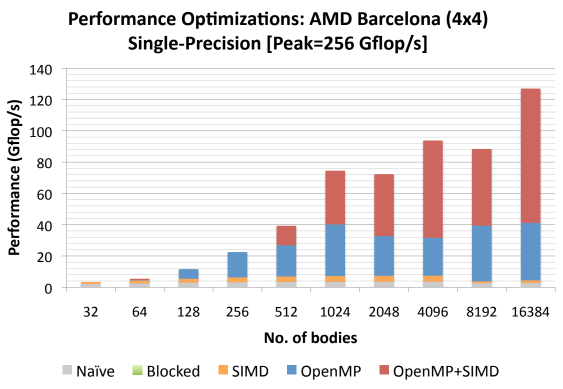


Fig. 4. Optimization breakdown: Single-precision AMD Barcelona

tiprocessor, which effectively serializes threads and reduces overall performance compared to single precision.

IV. Results and Discussion

A. Performance

We evaluate the implementations described in Section III on the hardware platforms given in Table I. We consider single-precision and, where possible, double-precision implementations. When reporting "Gflop/s," recall that we use 20 flops per iteration, counting 1 divide (reciprocal) and 1 square-root as single flops each. Since these operations normally have much higher latencies, we should not expect to get close to peak on any platform. Note that we time the force evaluation *in context* of the integrator, which executes on the host CPU (x86 on GPU, PowerPC on PowerXCell/8i) but which accounts for a negligible number of flops. Therefore, the Gflop/s rates *includes* the time to transfer data between host and accelerator.

1) CPU performance: We begin with the "baseline" CPU performance, to have a reference for comparing the subsequent GPU and PowerXCell/8i performance.

First, we consider the effect of parallelization, cache-level blocking, and manual vectorization on the Intel Nehalem and AMD Barcelona platforms, for single-precision and double-precision in Figures 3 - 6. The least useful optimization was blocking, which never helped. The presence of large caches on both systems helps to explain these results. As expected, the most effective optimization was parallelization.

With regard to manual SIMD vectorization, the results are qualitatively similar across the two platforms, but differ markedly between single- and double-precision. For single-precision, manual SIMD vectorization pays off significantly, boosting sequential performance by 2× (compare gray and yellow bars) and boosting parallel performance by 2× as well (compare blue and red bars).

However, in double-precision, SIMD vectorization made no difference, and overall double-precision performance is a much

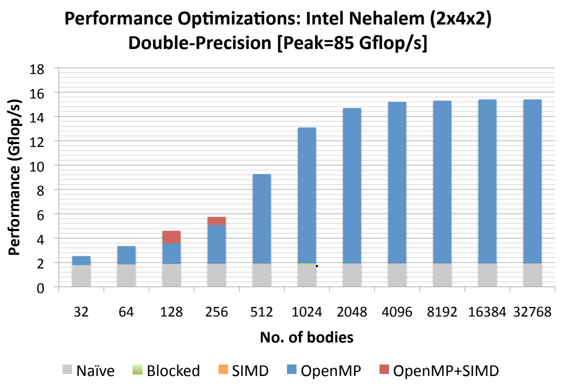


Fig. 5. Optimization breakdown: Double-precision Intel Nehalem

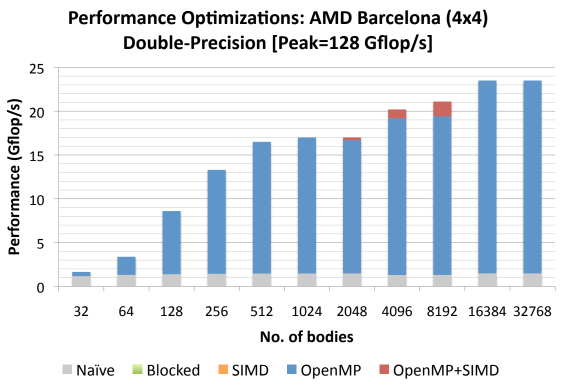


Fig. 6. Optimization breakdown: Double-precision AMD Barcelona

Criteria	NVIDIA Tesla C870	NVIDIA Tesla C1060	IBM Cell/B.E. QS22 blade	Intel Xeon X5550 4-core "Nehalem"	AMD Opteron 8350 4-core "Barcelona"
SP Gflops	512	993	409.6	170	256
DP Gflops	NA	78	204.8	85	128
Peak Power (W)	220	250	250	2*95*4=760	2*75*4=600
No. of sockets	1	1	2	2	4
Cores per socket	16	30	8+1	8	16
H/W threads per core	8	8	1	2	1
Total threads	128	240	16+2	16	16
Approx. price	$2250	$2229	$9555	$3000	$8000

TABLE I

SUMMARY OF EVALUATION PLATFORMS. "COST" IS A SYSTEM COST WITH MEMORY UP TO 4 GB. FOR GPUS, COST INCLUDES A NOMINAL AMOUNT FOR A DESKTOP SYSTEM IN WHICH THE CARD MAY RESIDE.

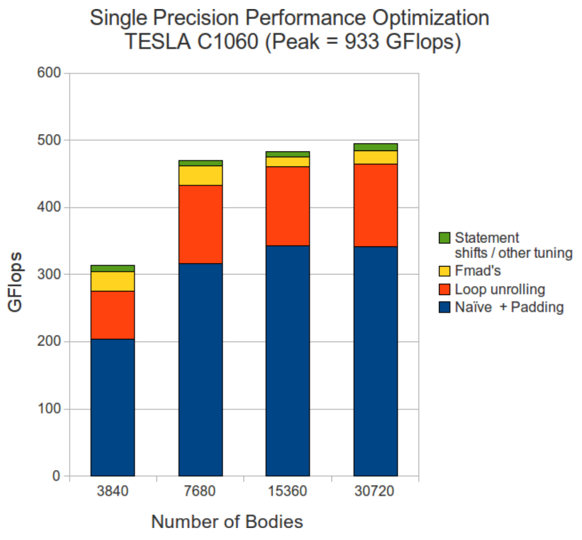


Fig. 7. Breakdown of the effects of performance tuning

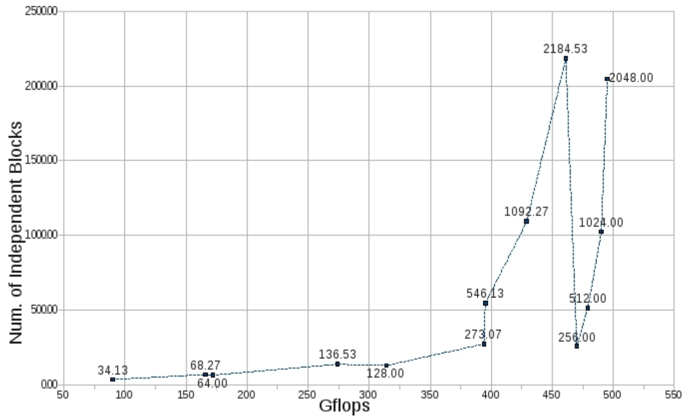


Fig. 8. Performance vs no. of independent thread blocks.

smaller fraction of peak than is the case in single-precision. When we simply removed the square root and reciprocal operations from the double-precision code, we saw a big boost in double-precision performance. Thus, the lack of good hardware support for these operations hampers performance.

2) GPU performance: The GPU implementation was the simplest of all the systems and required the least amount of tuning. Still, some degree of tuning was needed to get the best possible performance, with loop unrolling being a particularly effective optimization. Figure 7 summarizes these results for the Tesla C1060 running in single-precision.

The results for double precision (not shown) were similar. The main difference is that the $rsqrtf()$ instruction does make a significant difference for the double precision implementation as compared to the single precision kernel.

Figure 8 shows how performance varies as the number of (independent) thread blocks increases. Not surprisingly, the number of thread blocks have to be some positive multiple of the number of multiprocessors for the architecture to perform at its best. The Tesla C1060 has 30 multi-processors while the Tesla C870 has 16. Hence Tesla C870 performs best for number of bodies as increasing power of 2 while we need slightly different number of bodies for the Tesla C1060 (for example instead of 4096 we would use 3840 number of bodies).

3) PowerXCell/8i performance: The performance of the STI PowerXCell/8i implementations exhibit good scalability for a moderately large number of bodies, as shown in Figure 9. If the number of bodies is sufficiently large, we attain up to 61% of peak. The relatively low performance at small numbers of bodies is due in large part to the cost of scheduling the SPU threads. Once this overhead is overcome, we can expect the algorithm to perform linearly as the number of bodies go on increasing. Although one might guess that this overhead might be due to the DMA that is done between the PPU and the SPU, our microbenchmarking the double-buffered DMA indicated that we do in fact overlap DMA transfer with computation.

Apart from double buffering, some other techniques which tend to improve the overall throughput include loop unrolling, manual vectorization and FMA insertion, and choosing the correct data layout. Though we do not break-down the impact of these optimizations due to technical limitations of doing so,[2] we observed about 2x performance increase by switching the data layout using separate coordinate arrays, as compared to a layout that packs coordinates+mass together for each

[2]The program wouldn't run at all in absence of double buffering; loop unrolling is mandatory to perform SPU SIMD-ization and also depends on the data layout.

point. Furthermore, unroll-and-jam applied to the outer loop combined with using hardware rotate instructions to maintain the alignment gave additional improvement.

Curiously, performance decreases as the number of SPEs used increases, usually from 4 SPEs to 8 SPEs. We were able to attribute this phenomenon to the SPE thread dispatch overhead, which triples for this increment but only doubles for every other increment. This causes a sharp fall in the Gflop/s measured at these points (4 to 8). However, as the number of interactions increases, this fixed overhead is hidden by the computation.

The double precision implementation builds upon the single precision algorithm. The performance is very low—6 Gflop/s in the best case—compared to the peak of over 200 Gflop/s. However, this relatively poor performance is due to the lack of hardware support for square roots and reciprocals. Note that we are using IBM's vectorized software implementations from the SIMDMath library provided with the SDK. To estimate the latency of these operations, we replaced the reciprocal and square root by simple multiplies. Omitting only one of either reciprocal or square root boosted performance from 6 Gflop/s to approximately 10–11 Gflop/s, and removing both boosted performance to over 100 Gflop/s. This demonstration shows the severity of a lack of hardware support for these operations.

Both the implementations were compiled with the IBM XLC (v 10.1) cross-compiler for the STI Cell platform, with optimization level -05. The single precision program benefited from the compiler optimizations, with the xlc adding a few tens of Gflop/s over the gcc compiled code. However, we found that compiler optimizations, or the compiler itself for that matter, made almost no difference for the double precision implementation.

4) Overall Performance Comparison: To summarize, we compare all implementations across all platforms in Figure 10 and 11. In absolute performance, the NVIDIA Tesla C1060 achieves the best single-precision performance by a factor of 2×. Perhaps somewhat surprisingly, the two GPU systems "win" even at very small particle sizes.

The CPU platforms achieve large fractions of peak as well in single-precision (67% on Intel Nehalem and 50% on the AMD Barcelona). In double-precision, the lack of hardware support for double-precision hampers performance. However, it is possible that more extensive and careful tuning of the double-precision implementation could make these platforms more competitive.

The Tesla C1060 and Tesla C870 both achieve approximately 50% of their theoretical peak in single-precision, delivering a near constant performance of 500 Gflop/s and 250 Gflop/s respectively. For double precision implementation, Tesla C1060 was able to reach 67% of its theoretical peak which was impressive.

The Cell implementation was able to reach about 60% of its advertised peak of 410 Gflop/s of an entire blade for a single precision implementation. The double precision implementation on the other hand failed to achieve a substantial fraction of peak, due largely to the lack of good hardware support for square root and reciprocal operations.

B. Productivity and Ease of Implementation

We relate the implementation complexity using very approximate estimates of the number of days of effort to get a baseline but tuned parallel implementation running correctly, and estimates of the code line sizes, in Table II. These estimates may be useful as a guide of development cost and productivity. We estimate programming complexity in terms of both number of days spent to reach the fully optimized version and the lines of code required. All essential parts of the code were included for each architecture. However, these measures are only a very rough guide and not intended to be interpreted too literally. For instance, we our "time" measures do not account for the time to learn the architectures of the target platforms.

For the GPU implementation most of the time was spent to design an algorithm which maps optimally to the GPU memory architecture. The actual implementation was straightforward. The double precision implementation was quite similar to the single precision one, except that we had to take into account the absence float4 intrinsic type counterpart in double. However, we did not split the 64 bit double precision number to store as two 32 bit value, even though this technique is known to reduce bank conflicts while accessing data in shared memory (local store). We assume that the flops intensity is sufficient to hide such conflicts.

In case of of IBM's QS22, the situation was different. Given the simplicity of the PowerXCell/8i architecture, the algorithm development was relatively easy. However, the actual coding required much more fine tuning and non-trivial optimization to reach the documented performance then expected. Moreover, the lack of hardware implemented function in double precision (reciprocal and reciprocal square root) limited performance.

The CPU implementation was moderately complex to code compared to the other platforms and its double-precision performance suffers for the same reasons as PowerXCell/8i.

Another interesting criteria for comparison is the "$/GFlops" parameter for each architecture. Figure 12 shows the variation of this criteria for 8192 number of bodies for each architecture. The GPU, no doubt ranks the highest as the most cost effective technology for this kernel. We however would like the reader to note that the PowerXCell/8i architecture could potentially outperform the others if the code was not square root and divide heavy.

V. Conclusion

This paper joins a recent cross-platform multicore performance studies [7], [16], [17] with the direct force evaluation component from n-body simulations. Direct n-body has two distinguishing characteristics from the kernels considered in these prior efforts, which include sparse matrix-vector multiply, stencil computations, and statistical data analysis. First, the direct n-body force evaluation is more heavily compute-bound, and thus stresses different aspects of the architectures and programming models. Secondly, this computation includes

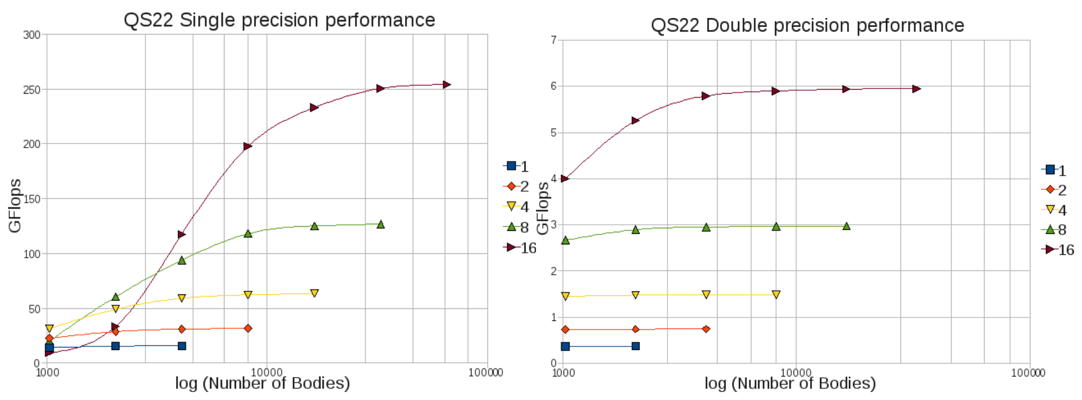


Fig. 9. STI PowerXCell/8i (QS22 blade with 2 CELL processors) performance.

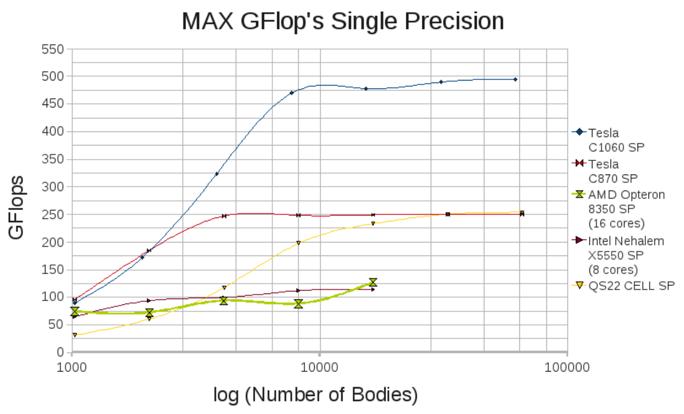


Fig. 10. Cross-architecture performance comparison (single-precision)

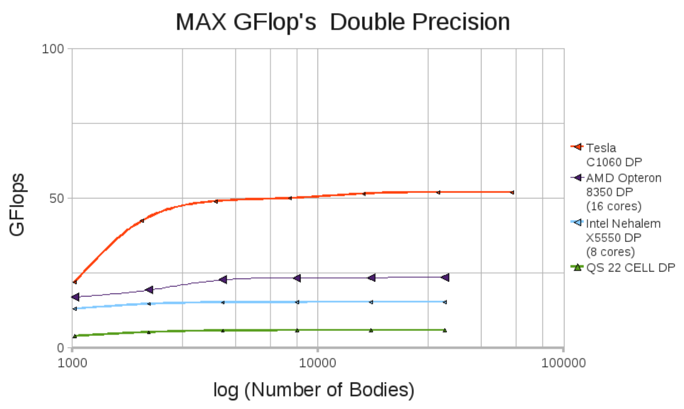


Fig. 11. Cross-architecture performance comparison (double-precision)

Criteria	**Tesla C870**	**Tesla C1060**	**QS22**	**Nehalem**	**Barcelona**
Code line size	380	390	850	500	500
No. of days spent	2	3	20	5	5

TABLE II
SUMMARY OF IMPLEMENTATION COMPLEXITY FOR VARIOUS PLATFORMS.

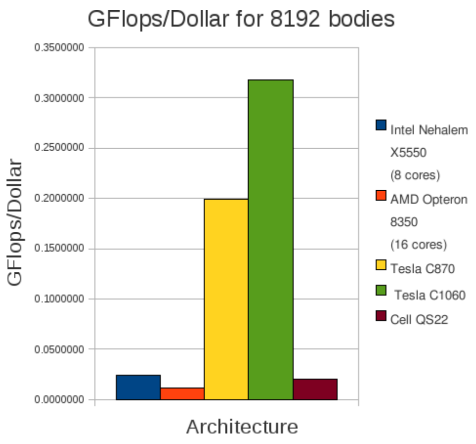


Fig. 12. Gflops/Dollar vs Number of bodies

square root and divide, whose latencies have large impacts in double-precision across a number of our evaluation platforms. Indeed, the slowness of these operations obviated the need to carry out extensive computation-oriented low-level tuning, and results in double-precision being more than $2\times$ slower than single-precision. We can conclude that for this class of computations, having these hardware features is essential.

We have also tried in this paper to assess, at least anecdotally, measures of cost, energy efficiency, and end-user programmer productivity on these platforms. By our proxy measures, GPUs and CUDA prove to be both the most cost- and power-efficient and the simplest to implement. Still, controlling for the basic coarse-grained parallelization approach, we still find a need for non-trivial tuning on all platforms. And looking to other applications, our findings will be somewhat limited to the case of semi-dense, semi-sparse methods.

The results of this paper are only an initial performance study. We are interested in higher-level hierarchical tree-based codes for the n-body problem, not only in physics but also in statistical data analysis and mining [6]. Since the direct n-body calculation appears as the leaf-leaf interaction of those problems, we expect our results to be useful in those contexts.

Acknowledgments

This work was supported in part by the National Science Foundation (NSF) under award number 0833136, NSF TeraGrid allocation CCR-090024, and a grant from the Defense Advanced Research Projects Agency (DARPA). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF or DARPA. We also acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this research.

We also acknowledge Dr. Hyesoon Kim for her advice during this project. We thank the various readers of the IBM Cell/B.E. and the NVIDIA CUDA community forums for additional advice and assistance throughout.

References

[1] S. P. Zwart, S. McMillan, D. Groen, A. Gualandris, M. Sipior, and W. Vermin, “A parallel gravitational n-body kernel,” *New Astronomy*, 2007.

[2] S. Aarseth, in *Gravitational N-Body Simulations*, 1st ed. . Cambridge University Press., 2003.

[3] J. Barnes and P. Hut., “A hierarchical o(n log n) force calculation algorithm,” *Nature*, 1986.

[4] L. Greengard, “The rapid evaluation of potential fields in particle systems,” *The MIT Press*, 1987.

[5] T. Hamada and T. Iitaka, “The chamomile scheme: An optimized algorithm for n-body simulations on programmable graphics processing units,” *New Astronomy*, 2008.

[6] A. G. Gray and A. W. Moore, “‘n-body’ problems in statistical learning,” in *Proc. Advances in Neural Information Processing Systems (NIPS)*, Vancouver, British Columbia, Canada, December 2000.

[7] S. Kang, D. Bader, and R. Vuduc, “Understanding the design tradeoffs among current multicore systems for numerical computations,” in *Proc. IEEE Int’l. Parallel and Distributed Processing Symp. (IPDPS)*, Rome, Italy, May 2009.

[8] M. Warren and J. Salmon, “A parallel hashed oct-tree n-body algorithm,” *Supercomputing 93*, vol. 15, no. 19, 1993.

[9] S.-H. Teng, “Provably good partitioning and load balancing algorithms for parallel adaptive n-body simulation,” *SIAM Journal on Scientific Computing*, vol. 19, no. 2, 1998.

[10] K. Nitadori, J. Makino, and P. Hut, “Performance tuning of n-body codes on modern microprocessors: I. Direct integration with a Hermite scheme on x86_64 architecture,” *New Astron.*, vol. 12, pp. 169–181, 2006.

[11] F. Chinchilla, T. Gamblin, M. Sommervoll, and J. F. Prins, “Parallel n-body simulation using GPUs,” *Technical Report TR04-032, University of North Carolina*, 2004.

[12] E. Elsen, V. Vishal, M. Houston, V. Pande, P. Hanrahan, and E. Darve, “N-body simulations on GPUs,” *Stanford University*, 2007.

[13] M. J. Stock and A. Gharakhani, “Toward efficient GPU-accelerated n-body simulations,” *AIAA*, 2008.

[14] R. G. Belleman, J. Bedorf, and S. F. P. Zwart, “High performance direct gravitational n-body simulations on graphics processing units II: An implementation in CUDA,” *New Astronomy*, 2008.

[15] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “NVIDIA Tesla: A unified graphics and computing architecture,” *Micro IEEE*, 2008.

[16] S. Williams, R. Vuduc, L. Oliker, J. Shalf, K. Yelick, and J. Demmel, “Optimizing sparse matrix-vector multiply on emerging multicore platforms,” *Journal of Parallel Computing*, 2009.

[17] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. A. Patterson, J. Shalf, and K. A. Yelick, “Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures,” in *Proc. ACM/IEEE Conf. on Supercomputing (SC)*, Austin, TX, USA, November 2008.