Available online at www.sciencedirect.com
ScienceDirect
Journal of Computational Physics 227 (2008) 5342–5359
JOURNAL OF COMPUTATIONAL PHYSICS
www.elsevier.com/locate/jcp

General purpose molecular dynamics simulations fully implemented on graphics processing units

Joshua A. Anderson [a,*], Chris D. Lorenz [b], A. Travesset [a]

[a] *Ames Laboratory and Department of Physics and Astronomy, Iowa State University, Ames, IA 50011, USA*
[b] *Materials Research Group, Engineering Division, King's College, London Strand, London WC2R 2LS, UK*

Received 21 September 2007; received in revised form 24 January 2008; accepted 29 January 2008
Available online 8 February 2008

Abstract

Graphics processing units (GPUs), originally developed for rendering real-time effects in computer games, now provide unprecedented computational power for scientific applications. In this paper, we develop a general purpose molecular dynamics code that runs entirely on a single GPU. It is shown that our GPU implementation provides a performance equivalent to that of fast 30 processor core distributed memory cluster. Our results show that GPUs already provide an inexpensive alternative to such clusters and discuss implications for the future.

PACS: 02.70.Ns; 61.20.Ja; 61.25.He

Keywords: Graphics processing unit; GPU; NVIDIA; CUDA; Molecular dynamics; Polymer systems

1. Introduction

Fuelled by the dramatic increases in computing power over the years, the impact of computational methods in the traditional sciences has been gigantic. Yet, the quest for new technologies that enable faster and cheaper calculations is more fervent than ever, as there are a vast number of problems that are on the brink of being solved if only a relatively modest increase in computer power were available.

Molecular dynamics (MD) has emerged as one of the most powerful computational tools [2], as it is capable of simulating a huge variety of systems both in and out of thermodynamic equilibrium. During the last decade, general purpose MD codes such as LAMMPS [3], DLPOLY [4], GROMACS [5], NAMD [6] and ESPResSO [7] have been developed to run very efficiently on distributed memory computer clusters. More recently, graphics processing units (GPUs), originally developed for rendering detailed real-time visual effects in computer games, have become programmable to the point where they are a viable general purpose programming platform. Dubbed GPGPU for general purpose programming on the GPU, it is currently getting a lot of attention

* Corresponding author.
E-mail address: joaander@ameslab.gov (J.A. Anderson).

0021-9991/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcp.2008.01.047

in the scientific community due to the huge computational horsepower of recent GPUs, evident in Fig. 1 [8–13]. The use of GPGPU techniques as an alternative to distributed memory clusters in MD simulations has become a real possibility.

Until recently, the only way to make use of the GPU's abilities was to carefully cast the algorithm and data structures to be represented as individual pixels being written to an image via *fragment shaders*. In addition to the cumbersome nature of programming this way, there are various other limitations imposed. Perhaps the most severe is that each thread of execution can only write a single output value to a single memory location in a gathered fashion. Scattered writes to multiple memory locations are important in implementing a number of algorithms, including parts of molecular dynamics. Likely because of this limitation, all the early implementations of MD using GPUs have been based on a mixed approach. The GPU performs those computationally intensive parts of MD that can be implemented in a gather implementation, and the CPU handles the rest [8,9]. A typical MD simulation implemented on the CPU spends only 50–80% of the total simulation time performing these gather operations. So the speedup by using a mixed CPU/GPU approach is limited to a factor of 2 or 3 at most, even though, as shown in Fig. 1, the GPU has the ability to perform significantly more floating point operations (FLOPs) per unit time than a CPU, thus leaving room for a dramatic speedup.

In this paper we provide, to our knowledge, the first implementation of a general purpose MD code where all steps of the algorithm are running on the GPU. This implementation is made possible by the use of the NVIDIA® CUDA™ C language programming environment. CUDA provides low level hardware access, avoiding the limitations imposed in fragment shaders. It works on the latest G80 hardware from NVIDIA, and will be supported on future devices [1]. Algorithms developed for this work will be directly applicable to newer, faster GPUs as they become available. After the initial submission of this paper, it came to our attention that van Meel et al. [13] submitted a similar work nearly simultaneously, where they also used CUDA to put all the steps of MD onto the GPU. The differences in implementation and performance are discussed in Section 2.7.

Early on in the development, it was decided not to take an existing MD code and modify it to run on the GPU. Any existing package would simply pose too many restrictions on the underlying data structures and the order in which operations are performed. Instead, a completely fresh MD code was built from the ground up with every aspect tuned to make the use of the GPU as optimal as possible. Despite these optimizations, every effort was made to develop a *general* architecture in the code so that it can easily be expanded to implement any of the features available in current general purpose MD codes.

This work includes algorithms for a very general class of MD simulations. N particles, in either an NVE or NVT ensemble, are placed in a finite box with periodic boundary conditions, where distances are computed according to the minimum image convention [14]. Because our own interests are the simulation of non-ionic polymers [15], non-bonded short-range and harmonic bond forces are the only interactions currently

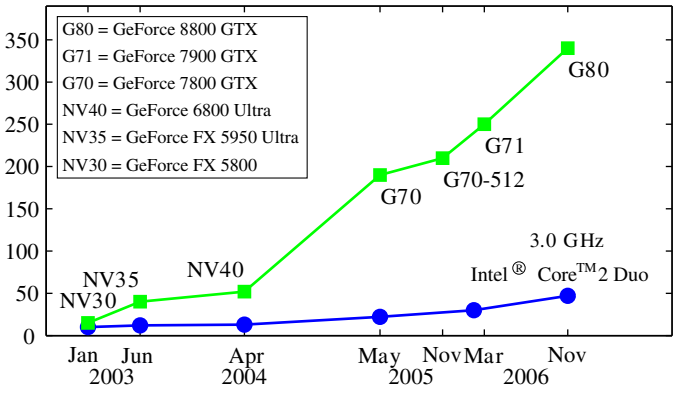

Fig. 1. Performance of CPUs (blue circles) and GPUs (green squares) over the last few years. Figure courtesy of NVIDIA, and adapted from Ref. [1]. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

implemented in our code. Three major computations are needed in every time step of the simulation: Updating the neighbor list, calculating forces, and integrating forward to the next time step.

Additional interactions such as angular and dihedral terms do not present new conceptual challenges, and can be implemented via adaptations of the algorithms presented here. Electrostatic forces are also required in a wide range of MD simulations. A GPU implementation of the PPPM solver [3] within the current code framework is certainly possible, but left for future work.

2. Implementation details

2.1. CUDA overview

Programming for the GPU with CUDA is very different from general purpose programming on the CPU due to the extremely multithreaded nature of the device. For an algorithm to execute efficiently on the GPU, it must be cast into a *data-parallel* form with many thousands of independent *threads of execution* running the same lines of code, but on different data. As a simple example, consider the pseudocode $a_i \leftarrow b_i \cdot c_i + d_i$ which needs to be calculated for all i from 0 to N. In a traditional CPU implementation, this instruction would be inside of a loop that calculates a_0, then a_1 ...one after the other. In contrast, one can imagine that all elements of a_i are calculated *simultaneously* on the GPU as the simplest model of thread execution. Because of this simultaneous execution, the output of one thread cannot depend upon the output of another, which can pose a serious challenge when trying to cast some algorithms into a data-parallel implementation.

In CUDA, these independent threads are organized into *blocks* which can contain anywhere from 32 to 512 threads each, but all blocks must be the same size. Any number of blocks can be run on the device, though optimum performance requires more than a hundred blocks executing in parallel. Each block executes identical lines of code and is given an index starting from 0 to identify which piece of the data it is to operate on. Within each block, threads are numbered from 0 to identify the location of the thread within the block. Using this indexing scheme, the pseudocode mentioned above could be implemented by using $i \leftarrow \text{bid} \cdot M + \text{tid}$, where bid is the block index, M is the number of threads per block and tid is the index of the thread within the block.

While the simultaneous picture of thread execution on the GPU is illustrative in a conceptual way, implementing efficient algorithms on the GPU requires understanding a little more in detail how the hardware actually executes all N threads. After all, any real hardware can only have a finite number of processing elements that operate in parallel. In particular, a single 8800 GTX GPU contains 16 *multiprocessors*. A single multiprocessor can execute a number of blocks concurrently (up to resource limits) in *warps* of 32 threads. For instance, let's say that there are three blocks on a multiprocessor with 128 threads each. In this case, there are 12 warps available for execution. At any given moment, the hardware examines which warps are ready to execute and chooses one. The current instruction of this warp is then executed on the multiprocessor and it moves on to another warp. In this manner, the execution of threads on the device is not so much simultaneous as it is *interleaved*.

There are two important consequences of this execution model. First is that the *smallest* unit of execution on the device is the warp. All 32 threads in the warp *must* execute the same instruction in a data-parallel fashion. Branches (if statements and loops) in the code can lead to different threads executing different instructions. As long as the entire warp follows the same branch, everything is fine. If different threads in the same warp follow different branches, however, the device must serialize this *divergent warp* in an inefficient manner resulting in less than optimal performance. One way that this can be avoided is through the use of *predicated instructions*. If the compiler deems a branch in the code is short enough, it will generate instructions such that all threads in the warp follow *both* branches, but the instructions have a predicate that prevents incorrect output from being generated.

The second consequence is that the device is able to hide memory access latencies. The GPU sits on an expansion board with its own memory (referred to as *global* or *device memory*) separate from the normal RAM accessible by the CPU. Device memory has a very high bandwidth available, over 70 GB/s, but also has a fairly high delay from the time a memory address is requested to the time it is available. During this latency period, hundreds of arithmetic operations can be performed on a multiprocessor. Thus, the advantage

of the interleaved warp execution is that warps which have read their data can be performing computations while other warps running on the same multiprocessor are waiting for their data to come in from global memory, effectively hiding the latency entirely.

On the subject of global memory, there is one additional detail critical to obtaining optimal performance. All memory access within a warp should be arranged such that thread w within the warp accesses the array element base + w, where base is a multiple of the warp size. In other words, threads within a warp must access elements along a *row* of a matrix in global memory which has been created with a padding to ensure that the base address is a multiple of the warp size down the columns. When all of these requirements are met, the global memory read is said to be *coalesced*. If the requirements are not met, the application still runs correctly, but memory access performance is reduced by a factor of 10–20. For example, the simple pseudocode above has coalesced memory accesses as long as M is a multiple of 32.

Using a coalesced global memory read is the fastest, most efficient way to read device memory, but not all algorithms can be adapted to read memory in such a regular pattern. For random access memory patterns, the hardware provides a *texture unit*, so called because it is used to paint two dimensional images (textures) across three dimensional surfaces when the GPU is used as a graphics output device. Every multiprocessor has its own small texture cache to handle random memory access patterns faster than non-coalesced global memory reads. Textures are just another way of reading global memory and elements can be accessed with one or two dimensional indices. In this work, only one dimensional textures are used and reads are denoted by tex1Dfetch().

The last aspect of CUDA to be aware of for the purposes of this work is that *not all* threads are really executed independently from one another. Only *blocks* are completely independent. For the threads within a single block, there are a few mechanisms by which they can communicate. First is a barrier mechanism named syncthreads(). Any thread in the block is delayed at this synchronization point until all other threads in the block reach it. Second, each block is given a small area of *shared memory* that exists on the multiprocessor. Any thread in the block can access this shared memory area without the latency associated with global memory reads. There are still a few performance pitfalls even with shared memory though, as it is accessed via 16 banks. For the full details on this and many other aspects of GPU programming not discussed here, interested readers are referred to the CUDA Programming Guide [1].

Most current MD codes developed for execution on a distributed memory cluster [3–5] use the Message Passing Interface (MPI) library to perform their computations across many nodes. MPI is similar to NVIDIA's CUDA in some sense: both allow a developer to write one bit of code that is executed many times in parallel on a grid. The similarities end here, however. MPI provides a number of routines for very efficient communication and synchronization between individual processes in the grid. CUDA threads cannot even communicate with each other, and synchronization can only be performed by allowing the kernel call to finish, except for the limited block shared memory and synchronization. Still, many parallel programming concepts commonly used in MPI programming can be applied to a CUDA kernel. Additionally, CUDA and MPI do not need to be considered as two different options for the implementation of a code. A cluster of nodes, each with a GPU, could be constructed and MD code developed that uses CUDA on each GPU with a message passing library to communicate between the nodes. Stone et al. [12] are working on this using NAMD. In this work, we focus on the core algorithms of MD and optimize them to be as fast as possible on a single GPU.

2.2. Short range non-bonded forces

Short range pair forces define the force between any pair of particles as a function of the displacement vector separating them, cutoff to 0 for distances greater than r_{cut}. The forces on all such pairs must be summed to produce the final net force on each of the N particles in the simulation. The standard algorithm [3,14,16,17] is to employ a data structure that lists interacting pairs, the neighbor list, which has already been calculated.

The neighbor list contains particles k that are less than some distance $r_{\max}$ from particle i, where $r_{\max} \geqslant r_{\text{cut}}$. Neighbor list entries are stored in a matrix $\text{NBL}_{ji} \leftarrow k$ with the list of neighbors for each particle i going down a column of the matrix. The number of neighbors in each list varies from particle to particle, so an auxiliary structure NN_i stores the number of neighbors of particle i. Using a matrix to represent the neighbor list deviates from standard practice [3,14] where a linked list is typically used instead. The matrix is used here because

the neighbor list will need to be read in a coalesced way in parallel on the GPU, which is impossible with a linked list.

The calculation of the forces on all N particles is implemented on the CPU following Algorithm 1. Focusing on this CPU implementation for the moment, Newton's third law is used to increment to the forces on particle i and k at the same time, reducing the number of *floating point calculations* that need to be performed by half. However, doing so requires that the all of the scattered memory locations k are accessed via a read–modify–write operation inside inner loop for each particle i. The price of the increased number of memory accesses and their random nature is to drop performance increase from the saved computations down to approximately 1.5 times faster from a maximum of 2 on the CPU.

Algorithm 1. Calculate pairwise forces: CPU implementation

Require: $\vec{F}_k$ is initialized to $\vec{0}$ for all k
Require: NBL_{ji} only stores neighbors where $i < k$
for all particles i
$\vec{A} \Leftarrow \vec{R}_i$
for $j = 0$ to $\mathrm{NN}_i - 1$ **do**
$k \Leftarrow \mathrm{NBL}_{ji}$
$\vec{B} \Leftarrow \vec{R}_k$
$\mathrm{d}\vec{r} \leftarrow$ minimum image of $\vec{B} - \vec{A}$
if $|\,\mathrm{d}\vec{r}\,| \leqslant r_{\mathrm{cut}}$
$\vec{C} \leftarrow \mathrm{force}(\mathrm{d}\vec{r})$
$\vec{C} + \vec{F}_i \Rightarrow \vec{F}_i$
$-\vec{C} + \vec{F}_k \Rightarrow \vec{F}_k$
end if
end for
end for

When designing algorithms on the GPU, the tradeoffs between memory access complexity, number of memory accesses, and number of computations performed are even more significant, since the price of scattered memory access vs. coalesced access is so high. Given this sensitivity, care is taken in writing the pseudocode to call attention to main memory reads (RAM on the CPU, device memory on the GPU), denoted by $\Leftarrow$, and writes, denoted by $\Rightarrow$. Reads and writes from local registers are similarly denoted respectively by $\leftarrow$ and $\rightarrow$.

For clarity of presentation, only a single pair force is calculated for all particles. It is, of course, easy to read in particle type identifiers and use different coefficients in the force calculation depending on what the type pair is.

Casting this calculation into a data-parallel algorithm for the GPU is easily done. Each thread simply calculates the total force on a single particle. Algorithm 2 lists the pseudocode for this implementation. Line 1 defines the indexing scheme for particles, which is the same as used in the simple example in Section 2.1. The choice of this indexing scheme is important as it sets later lines up for coalesced memory reads, like line 4, which reads the number of neighbors of particle i. Each thread now needs to loop over all of the neighbors of particle i starting on line 5. Not all particles have the same number of neighbors, so this loop will produce divergent warps. Fortunately, testing shows that the performance loss from this inefficient use of the device is only about 10%.

Algorithm 2. Calculate pairwise forces: GPU implementation

Require: bid is the index of this block
Require: tid is the index of this thread within the block
Require: M is the number of threads in each block
Require: $\lceil N/M \rceil$ blocks are run on the device
Require: NBL_{ji} stores *all* neighbors of each particle

1. $i \leftarrow \text{bid} \cdot M + \text{tid}$
2. $\vec{F}_{\text{sum}} \leftarrow \vec{0}$
3. $\vec{A} \Leftarrow \text{tex1Dfetch}(\vec{R}_i)$
4. $N_{\text{neigh}} \Leftarrow \text{NN}_i$
5. **for** $j = 0$ to $\text{N}_{\text{neigh}} - 1$ **do**
6. $\quad k \Leftarrow \text{NBL}_{ji}$
7. $\quad \vec{B} \Leftarrow \text{tex1Dfetch}(\vec{R}_k)$
8. $\quad \text{d}\vec{r} \leftarrow$ minimum image of $\vec{B} - \vec{A}$
9. $\quad \vec{C} \leftarrow \text{force}(\text{d}\vec{r})$
10. $\quad$ **if** $|\,\text{d}\vec{r}\,| > r_{\text{cut}}$ **then**
11. $\quad\quad \vec{C} \leftarrow \vec{0}$
12. $\quad$ **end if**
13. $\quad \vec{F}_{\text{sum}} \leftarrow \vec{C} + \vec{F}_{\text{sum}}$
14. **end for**
15. $\vec{F}_{\text{sum}} \Rightarrow \vec{F}_i$

Moving on to the inner loop, line 6 reads in the index of the neighboring particle from global memory. Since i increases with stride 1 as the thread index increases and i indexes along a row of NBL_{ji} this read is coalesced as long as the proper padding is chosen for the matrix. Line 7 reads the position of the neighboring particle k using the texture unit to take advantage of the texture cache for this random read. The memory access performance of this line is highly dependent on the nature of the layout of the particles in memory. If the neighbors k of particle i are randomly distributed from 1 to N there will be a high cache miss rate and the performance of this memory read can suffer by up to a factor of 4, see Fig. 3. Such a randomly ordered neighbor list is to be expected in a long running simulation. A solution to this problem is to reorder the particles in memory to improve data locality and the cache hit rate, see Section 2.3.

Continuing, line 9 calculates the force between this pair of particles i, k and adds it to the total in line 13. If this pair of particles is further apart than the cutoff, lines 10 and 11 ensure that the force sum is not modified. With such a simple statement (assigning 0 to a variable) inside the branch, the compiler generates predicated instructions for this operation, verified by reading the assembly output, thus avoiding the performance penalty of divergent warps here. Benchmarks show that this kernel is memory access bound, so any time wasted performing the comparison is hidden within the memory latency.

Options in the implementation of force($\text{d}\vec{r}$) are to calculate it directly using floating point operations or to use lookup tables via a texture. With Lennard–Jones potentials directly calculated, performance measurements show that this algorithm's performance is bound by the number of memory accesses it makes. Thus, using a lookup table that introduces more memory reads would slow performance. Potentials with more computationally intensive functional forms, such as those used in Ref. [12], may perform better when using lookup tables.

Finally, the total force summed for particle i is written to global memory in a coalesced manner at line 15. Notice that the GPU algorithm does *not* take advantage of the Newton's third law optimization used on the CPU. First of all, doing so would require atomic read–modify–write operations on the GPU which are not present for floating point numbers in current hardware. If such features did exist, the increment of $\vec{F}_k$ inside the inner loop still requires a scattered memory access pattern which would slow performance significantly. The additional read and write of $\vec{F}_k$ introduced also leads to an *increase* in the total number of memory accesses performed in an algorithm that is already memory bound. So even if they could be performed at full speed, the performance of the algorithm using Newton's third law would still be slower in the end.

2.3. Particle sort

Algorithm 2 reads memory in a random pattern determined by the distribution of the neighbors k of particle i, which are uncorrelated in a long running simulation. If the memory space taken up by the positions of

the particles in the system exceeds the cache size, nearly every memory access will then result in a cache miss, requiring the GPU's texture unit to constantly read scattered data from the device memory.

Several sorting techniques have been proposed to solve this issue [16,18] for MD on the CPU. These solutions rearrange the order in which particle positions are stored in memory so that neighboring particles are also nearby each other in memory. GPART [18] is based on a graph partitioning technique. It arranges particles so that the memory read patterns performed by Algorithm 1 are optimized for the L1 CPU cache. For instance, if the cache line size were such that four positions fit in a single cache line, GPART might produce the following neighbor list for a particle: 2 3 4 8 9 10 11 100 101 102 103...

Ideal for the serial CPU algorithm, it turns out that GPART generates a poor memory access pattern for the GPU implementation which reads neighbors of M particles simultaneously, a fact that GPART does not take into account. This actually leads to *worse* performance than randomly ordered data for $N < 20{,}000$. While it may be possible to develop a modified version of GPART, there is a simpler approach. Early testing of Algorithm 2 was performed on systems of particles in nicely arranged simple cubic arrays. Performance tests showed memory transfers approaching device limits, and a successful sort algorithm was developed that mimics this pattern for arbitrary systems. An even better option is to reorder the particles based on the path of a space filling curve [19,20], which have received a lot of attention lately for efficient database searches on multi-dimensional data sets. The Hilbert curve is chosen for this work because it has the best locality preserving properties [19]. Ref. [18] mentions the possibility of using a Hilbert curve for the data sort algorithm, but ignores it as a viable option in favor of another algorithm, RCB.

Ref. [20] discusses, in detail, a recursive algorithm for generating the Hilbert curve on the fly and assigning an index to each point in a point cloud. This index is then used as the primary key in a database record. The algorithm avoids ever computing the entire Hilbert curve so that large data sets which do not fit into system RAM can be indexed. In this work, all particles do fit in memory, so a simpler approach can be adopted, which we call the space-filling curve pack (SFCPACK) algorithm. SFCPACK consists of the following steps:

(i) Divide the simulation box into cells $\approx 1\sigma$ wide.
(ii) Place each particle into its corresponding cell.
(iii) Generate a traversal over the cells using the recursive algorithm in Ref. [20].
(iv) Loop over the cells in this traversal order, making an ordered list of particles visited.
(v) Reorder the particles based on the list.

2.4. Neighbor list generation

The objective of the neighbor list generation algorithm is to examine the current positions of all N particles and build a list for each containing those particles separated by a minimum image distance [14] less than the cutoff. This can be trivially done by an all vs. all comparison, but doing so requires $O(N^2)$ execution time making simulations of large systems prohibitively time consuming. Algorithmic improvements on the CPU have reduced the time to scale as $O(N)$ in the average case [3,16,17]. Implementing this more complicated neighbor list algorithm on the GPU poses the biggest challenge in developing a general purpose MD code running fully on the device. It requires the building of variable length lists using scattered memory writes, which cannot be realized with the fragment shader approach, but can be easily performed using CUDA.

To improve the overall performance of the entire system, MD codes such as LAMMPS [3] employ a standard trick. The cutoff distance for the neighbor list is chosen as $r_{\max}$, greater than the value of r_{cut} used for the pair forces. Then, the neighbor list only needs to be updated when any particle has moved a distance more than $\frac{1}{2}(r_{\max} - r_{\text{cut}})$, which is usually every 10 or more time steps.

Actually updating the list in $O(N)$ time involves looping through the N particles and placing them into N_{cell} cells of width r_{cell}, called *binning* the particles. When building the neighbor list, only the particles in the set of neighboring cells need to be considered [16,17]. When r_{cell} is chosen to be equal to $r_{\max}$, 27 bins need to be read for each particle, but a significant fraction of these particles will not make it into the neighbor list. Floating point computations can be saved at the cost of increased memory accesses (125 neighboring cells need to be accessed, many of which are empty) by choosing $r_{\text{cell}} = \frac{1}{2} r_{\max}$ [16,17]. Performance tests were made on the

GPU, showing that the additional memory accesses cost more than the extra floating point operations, so r_{cell} will be fixed at $r_{\max}$ for this work.

After the particles have been placed in their cells, updating the neighbor list matrix is accomplished on the CPU with the pseudocode in Algorithm 3. It is included so that comparisons can be drawn with the GPU implementation.

Algorithm 3. Neighbor list build: CPU implementation

for all particles i
  $\vec{A} \Leftarrow \vec{R}_i$
  $A_{\text{cell}} \leftarrow$ cell containing $\vec{A}$
  $N_{\text{neigh}} \leftarrow 0$
  for all 27 cells C in the neighborhood of A_{cell} **do**
    for all particles j in cell C **do**
      $\vec{B} \Leftarrow \vec{R}_j$
      $\mathrm{d}\vec{r} \leftarrow$ minimum image of $\vec{B} - \vec{A}$
      if $|\,\mathrm{d}\vec{r}\,| \leqslant r_{\max}$ **and** $i \neq j$ **then**
        $j \Rightarrow \mathrm{NBL}_{N_{\text{neigh}},i}$
        $N_{\text{neigh}} \leftarrow N_{\text{neigh}} + 1$
      end if
    end for
  end for
end for
$N_{\text{neigh}} \Rightarrow \mathrm{NN}_i$

The first portion of the algorithm, binning the particles, is challenging to implement efficiently on the GPU. Our implementation copies the particle positions from the device, bins them on the CPU and then copies the resulting cell lists back to the device. The binning process is slow and the overhead from the memory copies to and from the device is appreciable. But, the whole operation only needs to be performed once approximately every 10 time steps, reducing the total overhead in a full simulation to 6%.

There are two methods that can be used to move this step to the GPU. First, the newer generation G92 graphics cards add the ability to perform read–modify–write operations atomically, which could be used to implement the binning on the GPU. Lacking one of these GPUs, we could not try this method. Another method involving double buffered partial updates of the cell list is presented in Ref. [13]. There was not sufficient development time available to try this method here before publishing. Either of the two could significantly reduce the 6% overhead mentioned previously, a reasonable but not significantly large speedup.

The implementation of the second step on the GPU is listed in Algorithm 4. Each block executed on the device handles the calculation of all of the neighbor lists for particles i of a single cell. The advantage of this layout is that all particles in a given cell must be compared against the same set of particles in neighboring cells. Thus all of the memory operations reading particle positions need only be done on a per block basis and then accessed via shared memory. Line 1 of the algorithm starts this off by reading in the particle index i from the $\mathrm{CELL}_{a,C}$ array in main memory. Like the neighbor list matrix NBL_{ji}, $\mathrm{CELL}_{a,C}$ is a matrix listing the particles j belonging to bin C. The difference is that the particle indices are listed along the *rows* of the matrix so that the reads on lines 1 and 6 will be coalesced (as long as the matrix has the correct padding).

Algorithm 4. Neighbor list build: GPU implementation

Require: bid is the index of this block
Require: tid is the index of this thread within the block
Require: M is the number of threads in each block
Require: $M \geqslant$ the largest number of particles in any given cell
Require: N_{cell} blocks are run on the device

```
Require: B_j and K_j are stored in shared memory
 1. i ⇐ CELL_{tid,bid}
 2. N_neigh ← 0
 3. A ⇐ tex1Dfetch(R_i)
 4. for all 27 cells C in the neighborhood of bid do
 5.    syncthreads()
 6.    K_tid ⇐ CELL_{tid,C}
 7.    B_tid ⇐ tex1Dfetch(R_{K_tid})
 8.    syncthreads()
 9.       if not empty(i) then
10.          for j = 0 to M − 1 do
11.             if empty(K_j) then
12.                break
13.             end if
14.             C ← B_j − A
15.             dr ← minimum image of B − A
16.             if | dr |< r_max and K_j ≠ i
17.                K_j ⇒ NBL_{N_neigh,i}
18.                N_neigh ← N_neigh + 1
19.             end if
20.          end for
21.       end if
22. end for
23. N_neigh ⇒ NN_i
```

Line 3 reads in the position of particle i for which this thread is going to build the neighbor list. As the loop on line 4 goes over all of the neighboring cells, the particle indices K_j and positions $\vec{B}_j$ are loaded into shared memory on lines 6 and 7. Particle positions are read using the texture unit to take advantage of the cache. The synchthreads() barriers surrounding these memory reads are needed to ensure that there are no race conditions when accessing shared memory. Line 9 skips computations if the particle i is actually an empty entry in the cell. This statement will produce divergent warps, but it prevents a large number of FLOPs from being wasted on empty cell elements. It cannot be placed any earlier in the code, since all threads in the block need to participate in the data loading done on lines 6 and 7.

Continuing on, line 10 starts a loop over all particles in cell C. This loop is done in every thread because every particle i must be checked against every other particle in the neighboring cells for inclusion into the neighbor list. Line 11 immediately checks if the current particle index K_j is an empty value, quitting the loop on line 10 if it is. Performing this early-out check inside the loop is important for performance, since a typical cell may have 30 particles in it, but the maximum has 60. Finally, lines 14 through 18 check the current pair of particles for inclusion into the neighbor list. All threads in any given warp in the block are reading the same indices of K_j and $\vec{B}_j$ simultaneously, so the shared memory broadcast will be invoked and there are no bank conflicts to slow shared memory read performance [1]. Scattered writing to the neighbor list on line 17 is not coalesced and thus will not perform optimally on the device. Fortunately, the bottleneck in this algorithm is the floating point operations so the inefficient memory write does not change its overall performance significantly.

2.5. Integration

The explicit Nosé–Hoover integration method from Ref. [21] is implemented on the GPU to move particle positions and velocities forward one time step in the NVT ensemble. Implementation of this method is straightforward and not provided here. It is important, however, that this step is done *on the GPU*, even if

the computation takes very little time on the CPU. When done on the CPU, forces calculated on the GPU must be copied to the CPU, the integration performed, and then the new particle data copied to the GPU. All of these memory copies amount to a significant fraction of the simulation time, as they are done on every time step. Additionally, the speedup factor of 20 executing on the GPU prevents the execution time of the integration on the CPU from becoming a bottleneck.

The NVE ensemble is similarly implemented using a velocity Verlet method [3].

Multiple time step methods [22] can be used to reduce computational cost in some types of MD simulations containing both slowly and quickly varying forces. Such methods can be implemented on the GPU in a straightforward manner, identical to any CPU implementation. None are implemented or benchmarked for this work. However, any relative reduction in computational cost on the CPU by using one of these methods would translate in a similar reduction on the GPU.

2.6. Bond forces

Harmonic bond forces are implemented on the GPU for the same reasons as the integration. Algorithm 2 is modified to read a different NBL_{ji} that includes a list of all particles k *bonded* to i instead of the neighbor list. Speedups of a factor of 75 compared to the CPU implementation are achieved.

Angle and dihedral terms are not implemented in this work, but could be achieved in a similar fashion requiring only a slightly more complicated list structure. Similar speedups are expected.

2.7. Brief comparison to other recent works

Refs. [12,13] also implement Molecular Dynamics using CUDA on the same hardware as is used here. Both work around the complexity of generating a neighbor list by storing particle positions in a cell data structure and using that directly in the pair force computation on the GPU.

Stone et al. [12] target large biomolecule simulations with their implementation. It only computes pair forces on the GPU, leaving all other tasks for the CPU. The pair potential they use is much more computationally intensive than the simple Lennard–Jones potential we use here. So instead of performing the computations directly on the GPU, they use a texture as a lookup table to interpolate pre-calculated terms. No performance comparisons can be made with this work because the potentials and systems benchmarked vary significantly. Another notable difference in their work is that they have linked their CUDA implementation of MD directly into NAMD and can thus distribute jobs across a cluster of nodes with GPUs.

The implementation by van Meel et al. [13] is closer to our own, modeling simple Lennard–Jones particles and fully implementing every step of MD on the GPU. A direct performance comparison can be made to quantify the tradeoffs between the neighbor list and cell based approaches. At parameters relevant to our work, $\rho = 0.4\sigma^{-3}$ and $N = 100{,}000$ they report 0.02 s of computation time per step, or 50 time steps per second. Configuring our software to implement the Lennard–Jones liquid with identical parameters, we measure 160 time steps per second, 3.2 times faster.

3. Performance measurements

3.1. Hardware

All single CPU/GPU benchmarks are run on a Dell Precision Workstation 470 with a 3.0 GHz 80546K Xeon processor and 1 GB of RAM. The original graphics card in this system was upgraded to a NVIDIA GeForce 8800GTX manufactured by EVGA® with the standard core clock speed of 575 MHz. The system dual boots Microsoft® Windows® XP and Gentoo Linux running the latest 2.6.21 AMD64 kernel. CUDA 1.1 is installed in both operating systems. Under Windows, code is compiled using Visual Studio Express 2005 with the compiler optimization options "/arch:SSE2/Ox/Ob2/Oi/Ot/Oy/fp:fast". Linux executables are produced with gcc version 4.1.2 using compiler optimization options "-march=nocona -O3 -funroll-loops".

CPU benchmarks are approximately 20% faster in Linux compared to Windows. GPU benchmarks are approximately 5% faster in Windows compared to Linux. The fairest comparison possible is given in this work by presenting CPU benchmark results obtained in Linux and GPU benchmark results obtained in Windows.

Lightning, a cluster on the ISU campus, is used for the LAMMPS cluster benchmarks. It has 90 compute nodes, each of which contain two dual-core AMD Opteron™ 280 processors and 8 GB of RAM. Nodes are interconnected with a high speed, low latency InfiniBand® network. PathScale™ 2.4 is used to compile LAMMPS on Lightning. In order to get the best performance possible out of LAMMPS for the fairest comparison, the new optimized Lennard–Jones routines from style_opt are used.

3.2. Lennard–Jones force calculation

To test the performance of Algorithms 1 and 2 in computing Lennard–Jones forces, particle systems are created with N particles randomly placed in a box with side lengths chosen so that the number density of particles is $n = 0.382\sigma^{-3}$. The force cutoff is set to $r_{\text{cut}} = 3.0\sigma$, which is typical in coarse-grained simulations [15] and $r_{\max}$ is set to a reasonable value of 4.0σ. Relative GPU/CPU performance differs little for various values of $r_{\max}$, so the precise value is irrelevant. The random placement of the particles is to emulate a typical time step in a simulation of a Lennard–Jones liquid. With the chosen cutoff values, there are 102 neighbors for each particle on average. While the particles are randomly placed, the same random seed is used each time so that the GPU and CPU benchmarks at the same N are calculating forces on exactly the same system of particles. Timing is performed by generating the neighbor list once and then running the force calculation repeatedly until 5 s have elapsed or the calculation has been done five times, whichever comes first.

Benchmark runs are performed on the CPU with the particle data unsorted, sorted by GPART [18], and sorted by SFCPACK. Results are plotted in Fig. 2. Performance is improved greatly with the sorting algorithms applied, especially for large N due to the more effective use of the memory cache. It is also worth noting that the much simpler SFCPACK algorithm beats the performance of GPART [18].

On the GPU, the performance of Algorithm 2 is sensitive to the choice of the block size M. Even though the same number of calculations and memory accesses are performed regardless, memory access patterns, register read after write dependencies, and the warp occupancy of the kernel running on the device can all change the running time [1]. It is impossible to predict which block size will lead to the best performance, so benchmarks on the GPU are performed at *all possible* block sizes from 32 to 448 in steps of 32, and the fastest time is chosen for the benchmark result. It would make no sense to select a value for M that is not a multiple of 32 since that is the smallest unit of execution on the device. The fastest block size for each system size is recorded in Fig. 5. When performing actual MD simulations of a specific system, these results can be used to determine which block size to use for the best performance.

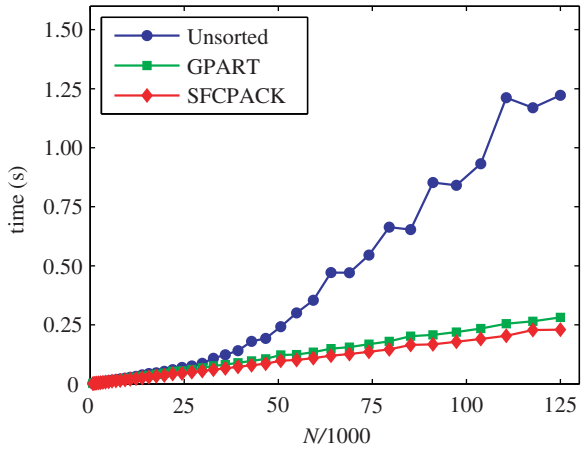

Fig. 2. Time to calculate Lennard–Jones forces for all N particles on the CPU plotted vs. system size. Results are shown with various sorting algorithms applied to the particles.

Like the results obtained on the CPU, performance of the Lennard–Jones force calculation on the GPU is sensitive to the order of the particles in memory, so benchmarks are performed with particles sorted by the various algorithms as before. Additionally, and unlike the CPU, performance is also sensitive to the order of particles *in the neighbor list*. When the neighbor list is generated by the simple $O(N^2)$ method, $NBL_{ji} < NBL_{j+1,i} \forall j$. Even though the neighbors k may be randomly distributed from 0 to $N-1$, this forced ordering in the neighbor list improves the locality of memory reads over a randomly ordered neighbor list. On the contrary, the more efficient $O(N)$ neighbor list does not generate ordered neighbor lists. Instead, it generates 27 independent sorted sequences in NBL_{ji} as j goes from 0 to $NN_i - 1$. While it is possible to use merge sort techniques to generate a fully ordered neighbor list here, the performance hit is significant. In order to evaluate the tradeoffs between an ordered neighbor list vs. an unordered one, the force computation is benchmarked using each on the GPU. Timing results for all these cases are provided in Fig. 3. The same trend seen on the CPU (SFCPACK is faster than GPART which is faster than unsorted) is repeated here. As expected, force computations done using ordered neighbor lists are faster than with unordered ones. Although, with the SFCPACK sort the difference is only a few percent. With these results in hand, the clear choice for implementing the fastest MD on the GPU is to use the fast, unordered neighbor list generator and the SFCPACK sort.

The fastest evaluations of the pair force calculations for the CPU and GPU, obtained with the SFCPACK sorted particles, are compared in Fig. 4. At 60 times faster for large enough systems, the GPU is an incredibly powerful tool for performing the pair force computation. While crunching numbers in the linear performance region ($N > 15{,}000$), the device is reading and writing memory at a rate of 60–70 GB/s which is close to theoretical peak device capability [1]. At the same time, only about 100 GFLOP/s are performed which is far from its theoretical peak of 340 GFLOP/s indicating that many of the multiprocessors sit idle with all of the active warps waiting on the global memory access latency. Thus, short-range pair forces that require more floating point operations than Lennard–Jones could also be implemented in Algorithm 2 without a significant loss in performance compared to the values presented here.

3.3. Neighbor list generation

The neighbor list generation code in Algorithms 3 and 4 is benchmarked on the same random particle systems used to benchmark the pair forces. The parameter $r_{\max}$ is not changed from 4.0σ. Removing the checks that only update the neighbor list when a particle has moved a sufficient distance, the same neighbor list is generated over and over again until 5 s has elapsed or the neighbor list has been generated five times. As the neighbor list generation algorithms read particle positions randomly from the data arrays based on the cell list, they also benefit from the particle data sort. Timings for these benchmarks are plotted in Figs. 6

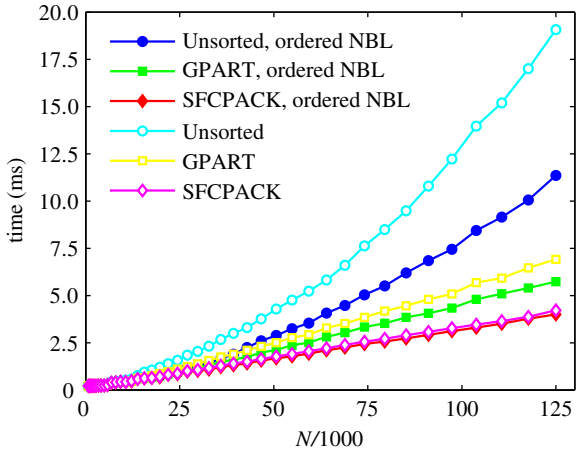

Fig. 3. Time to calculate Lennard–Jones forces for all N particles on the GPU plotted vs. system size. Results are shown with various sorting algorithms applied to the particles. Open symbols indicate times were obtained with unordered neighbor lists, and closed symbols indicate the neighbor list was ordered.

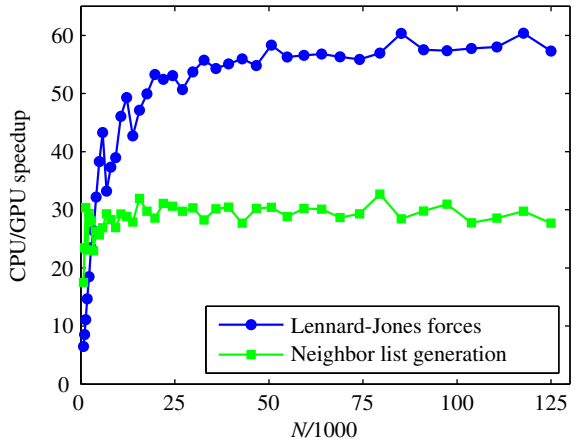

Fig. 4. CPU time/GPU time for the Lennard–Jones force calculation and neighbor list generation plotted vs. system size. Particles are sorted using the SFCPACK algorithm in these benchmarks.

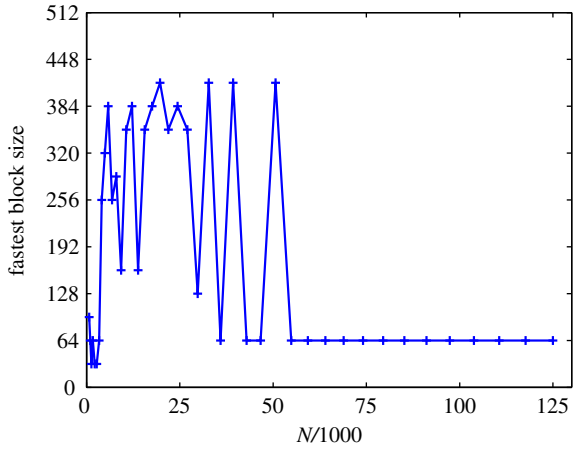

Fig. 5. Best performing block size M for the GPU implementation of the Lennard–Jones force calculation plotted vs. system size.

and 7, which show the total time taken to bin the particles and update the neighbor list matrix. Execution time clearly increases linearly for both the CPU and GPU implementations with the total speedup on the GPU hovering around 30, as shown in Fig. 4. While not quite as impressive a speedup as with the Lennard–Jones forces, the neighbor list generation is still significantly faster on the GPU.

3.4. Benchmark of MD simulations

For a practical comparison with an existing software solution, benchmarks of the full MD simulation are run using both the GPU implementation on a single GPU and LAMMPS [3] running on the Lightning computer cluster. It should be expected that some differences might be obtained by testing other general purpose MD packages, but in this paper all comparisons are done with LAMMPS as this is the code we use in our applications.

The neighbor list generation and pair force computations algorithms interact with $r_{\max}$ in different ways. If the value is chosen too small, the neighbor list will need to be updated every other time step, slowing performance significantly. A larger value leads to fewer neighbor list updates, but then the length of the neighbor list gets larger and more forces need to be computed slowing down overall performance again. In all benchmarks presented here, short test runs with $r_{\max}$ varying in steps of 0.1 are performed to determine the optimal value.

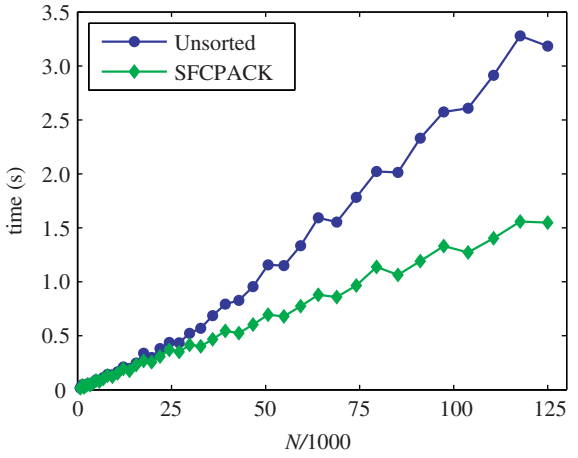

Fig. 6. Time to generate the neighbor list for all N particles using Algorithm 3 on the CPU plotted vs. system size. Results are shown with and without the SFCPACK sort applied to the particle data.

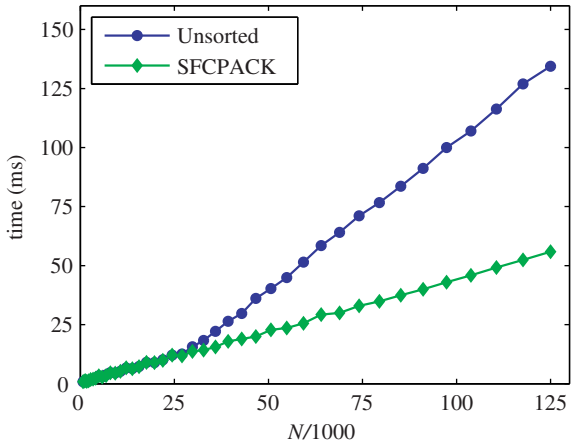

Fig. 7. Time to generate the neighbor list for all N particles using Algorithm 4 on the GPU plotted vs. system size. Results are shown with and without the SFCPACK sort applied to the particle data.

3.4.1. Lennard–Jones liquid

A full simulation of a Lennard–Jones liquid is implemented with the GPU algorithms presented in this work. Initial conditions are prepared by starting from a random arrangement of N particles at density $n = 0.382\sigma^{-3}$, then running the simulation in the NVT ensemble at $\frac{k_B T}{\varepsilon} = 1.2$ for 50,000 time steps to equilibrate the system. An additional 50,000 time steps are then timed for the benchmark. The timed runs for both the GPU and LAMMPS both start from the same prepared initial condition. For all simulations, $r_{\text{cut}} = 3.0\sigma$.

On the GPU, the optimal value of r_{max} is found to be 3.8σ for this system. Additionally, the block size for the force computation is chosen from Fig. 5 and SFCPACK is used to resort the particles every 1000 time steps. LAMMPS performed best with $r_{\text{max}} = 3.4\sigma$ and was timed running on various numbers of processor cores up to 40.

Results of these tests are shown in Fig. 8 and summarized in Table 1. As expected from the previous comparisons, Fig. 4, the GPU performs at the same speed as around 36 processor cores on Lightning.

3.4.2. Coarse-grained polymer systems

To show the versatility of our GPU implementation, another test case is implemented that uses a portion of the polymer model from Ref. [15]. In short, polymers are built with A and B particles connected by harmonic

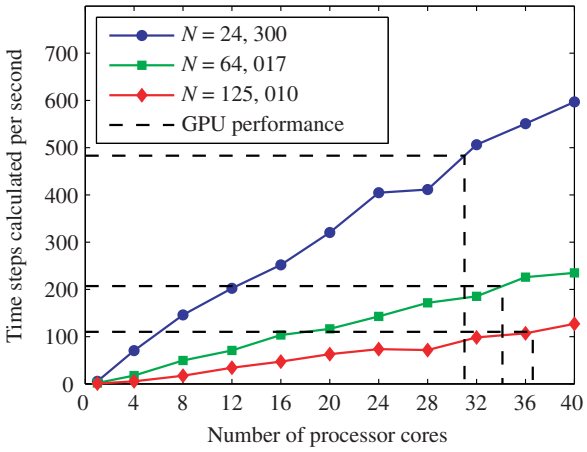

Fig. 8. Benchmark of LAMMPS simulating the Lennard–Jones liquid model for various system sizes on Lightning. The dotted lines mark the performance of the GPU running the same simulation.

Table 1
Average performance results summary

System	Lennard–Jones liquid	Polymer model
Density	$0.382\sigma^{-3}$	$0.382\sigma^{-3}$
Average neighbors per particle	43.2	43.2
CPU time/particle/time step (LAMMPS)	2.64 μs	2.34 μs
GPU time/particle/time step	0.0778 μs	0.0806 μs

bonds to form a chain: $A_{10}B_7A_{10}$. Non-bonded interactions are implemented with forces derived from Lennard–Jones potentials:

$$U_{AA}(r) = 4\varepsilon\left(\frac{\sigma}{r}\right)^{12}$$

$$U_{BA,BB}(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]$$

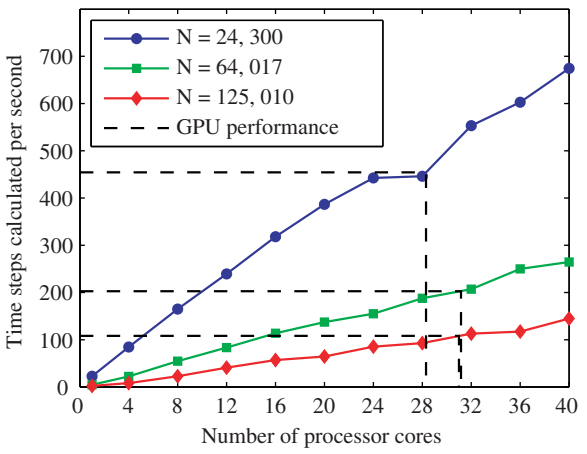

Fig. 9. Benchmark of LAMMPS simulating the polymer model for various system sizes on Lightning. The dotted lines mark the performance of the GPU running the same simulation.

Preparation and timing of these systems is performed as before with the Lennard–Jones liquid, with all parameters identical.

The benchmark timings for various system sizes are plotted in Fig. 9 and summarized in Table 1. On the GPU, absolute performance numbers are slightly lower when compared to the Lennard–Jones liquid system from the resulting overheads of the bond force sum. LAMMPS on the other hand is actually slightly *faster* at executing the polymer system. Examination shows that the Lennard–Jones liquid requires significantly more time spent on inter-node communication. These two effects combine to bring the overall comparative performance of the GPU down a bit to 32 processor cores on Lightning.

4. Numerical precision tests

Current GPUs only offer support for single precision floating point arithmetic, and not all operations meet the IEEE 754 standards [1]. It may be argued that the precision of results obtained via MD simulation on the GPU are thus suspect. To demonstrate that this is not the case, simulations of the polymer system are performed on the CPU and GPU using single precision math, on the CPU using double precision math, and using LAMMPS on Lightning. Identical initial conditions are used in all runs. The equilibrated initial condition is produced via the same method used in Section 3.4 with $N = 24,300$, and then all test cases continue the simulation from that point. LAMMPS and the code developed for this work use different implementations of the NVT integrator and thus cannot be compared. Continuation runs are instead performed in the NVE ensemble, where both codes use a Velocity Verlet integrator [3].

Particle positions are dumped every 100 time steps in each simulation run performed. The $\langle(\Delta\vec{r}/\sigma)^2\rangle$ deviation is computed between the particle positions of two different simulations at identical time steps. Using a LAMMPS double precision run on a single processor as a baseline for comparison, Fig. 10 shows the results. Initially, LAMMPS and the code developed in this work do indeed calculate the same trajectory. Using double precision computations on the CPU, 2000 time steps are performed before the trajectories start to differ significantly. Single precision calculations on the CPU are slightly less precise with the trajectories starting to diverge at 1000 time steps. The GPU single precision implementation is no worse than the CPU, diverging at exactly the same point.

To demonstrate what one *should expect* of a precise MD trajectory, the same simulation run is performed using LAMMPS again, but this time on four processor cores in parallel. In this case, the trajectory starts to deviate at 2000 time steps, which is no different from the double precision calculation on the CPU. These two simulations starting from identical initial conditions can vary so drastically even when executed with the same code because forces on the particles are summed in a different order on 1 CPU vs. 4 CPUs. With floating point

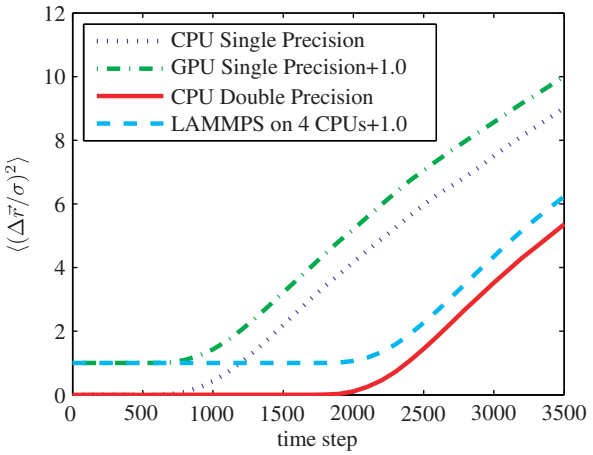

Fig. 10. Deviation of simulation trajectories from a baseline generated by LAMMPS running on a single processor. Single and double precision modes on the CPU and the single precision GPU calculations are compared to the baseline. Data from a LAMMPS run on four processors is also presented. Two of the four curves have their 0's artificially shifted to visually separate them from their counterparts.

arithmetic, summing identical values in a different order can produce different results. The slightest difference in a computed force introduces a difference in the particles' positions between the two simulations at the next time step. These differences then compound time step after time step until the trajectories vary significantly. A typical MD simulation will integrate millions of time steps, so a deviation at 1000 vs. 2000 steps is irrelevant.

Energy conservation during an NVE simulation run is a more rigorous metric to measure the precision of MD simulations. We check that energy is conserved on the GPU, but do not provide a detailed quantitative study here. Future GPUs will include support for double precision [1], so the capability will eventually be there for those simulations that need the extra precision.

5. Conclusions

We have presented a general purpose MD simulation fully implemented on a single GPU. We have compared our GPU implementation against LAMMPS running on a fast parallel cluster, see Fig. 8, and we have shown that the GPU performs at the same level as up to 36 processor cores. Smaller, less power hungry, easier to maintain, and inexpensive compared to such a cluster, GPUs offer a compelling alternative. And this is only the beginning. As Fig. 1 shows, the trend towards faster GPUs is inexorable, allowing a simple component upgrade to enable even more demanding MD simulations. The fact that all future NVIDIA GPUs will support the CUDA environment [1] warrants the generality of the implementation presented here.

Current technology allows even faster MD simulations in a single workstation. Up to four GPUs can be hosted in a single workstation potentially bringing the power of a 144 core cluster to the desktop at a fraction of the cost. While GPUs do not currently provide a viable alternative to supercomputers able to simulate systems with more than a million particles, we expect that GPUs will very soon become a commonplace tool for smaller simulations. The only major element missing for many types of simulations is the implementation of electrostatic forces, which we leave for future work.

Acknowledgement

This work is funded by NSF through Grant DMR-0426597 and by DOE through the Ames lab under Contract No. DE-AC02-07CH11358.

References

[1] Cuda programming guide 1.1. <http://developer.nvidia.com/object/cuda.html>.
[2] D. Frenkel, B. Smit, Understanding Molecular Simulations, Academic Press, San Diego, CA, 2002.
[3] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, J. Comp. Phys. 117 (1995) 1–19.
[4] W. Smith, T.R. Forester, DL_POLY_2.0: a general-purpose parallel molecular dynamics simulation.
[5] H.J.C. Berendsen, D. van der Spoel, R. van Drunen, GROMACS: a message-passing parallel molecular dynamics implementation, Comp. Phys. Commun. 91 (1995) 43–56.
[6] J.C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R.D. Skeel, L. Kalé, K. Schulten, Scalable molecular dynamics with NAMD, J. Comp. Chem. 26 (2005) 1781–1802.
[7] H.-J. Limbach, A. Arnold, B.A. Mann, C. Holm, ESPResSo – an extensible simulation package for research on soft matter systems, Comput. Phys. Commun. 174 (2006) 704–727.
[8] S. Kupka, Molecular dynamics on graphics accelerators, in: CESCG, 2006.
[9] J. Yang, Y. Wang, Y. Chen, GPU accelerated molecular dynamics simulation of thermal conductivities, J. Comp. Phys. 221 (2007) 799–804.
[10] R. Belleman, J. Bedorf, S.P. Zwart, High performance direct gravitational N-body simulations on graphics processing units – II: an implementation in CUDA, New Astronomy 13 (2) (2008) 103–112.
[11] H.-Y. Schive, C.-H. Chien, S.-K. Wong, Y.-C. Tsai, T. Chiueh, Graphic-card cluster for astrophysics (GraCCA) – performance tests, New Astronomy (in press) doi:10.1016/j.newast.2007.12.005.
[12] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, K. Schulten, Accelerating molecular modeling applications with graphics processors, J. Comp. Chem. 28 (2007) 2618–2640.
[13] J.A. van Meel, A. Arnold, D. Frenkel, S.F.P. Zwart, R.G. Belleman, Harvesting graphics power for MD simulations, Mol. Sim. (in press).
[14] M.P. Allen, D.J. Tildesley, Computer Simulation of Liquids, Clarendon Press, Oxford, 1987.
[15] J.A. Anderson, A. Travesset, Coarse-grained simulations of gels of nonionic multiblock copolymers with hydrophobic groups, Macromolecules 39 (15) (2006) 5143–5151.

[16] Z. Yao, J. Wang, G. Liu, M. Cheng, Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method, Comp. Phys. Commun. 161 (2004) 27–35.
[17] T. Maximova, C. Keasar, A novel algorithm for non-bonded-list updating in molecular simulations, J. Comp. Biol. 13 (2006) 1041–1048.
[18] H. Han, C.-W. Tseng, A comparison of locality transformations for irregular codes, in: Fifth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR'2000), 2000, pp. 253–275.
[19] B. Moon, H. Jagadish, C. Faloutsos, J.H. Saltz, Analysis of the clustering properties of the hilbert space-filling curve, IEEE Trans. Knowl. Data Eng. 13 (1).
[20] J. Wang, J. Shan, Space-filling curve based point clouds index, in: Geocomputation, 2005.
[21] S.D. Bond, B.J. Leimkuhler, B.B. Laird, The Nosé–Poincaré method for constant temperature molecular dynamics, J. Comp. Phys. 151 (1999) 114–134.
[22] M. Tuckerman, B.J. Berne, G.J. Martyna, Reversible multiple time scale molecular dynamics, J. Chem. Phys. 97 (3) (1992) 1990–2001. <http://link.aip.org/link/?JCP/97/1990/1>.