Experiences Porting a Molecular Dynamics Code to GPUs on a Cray XK7

Donald K. Berry
Indiana University
Bloomington, IN, USA
dkberry@iu.edu

Joseph Schuchart
Oak Ridge National Laboratory
Knoxville, TN, USA
schuchartj@ornl.gov

Robert Henschel
Indiana University
Bloomington, IN, USA
henschel@iu.edu

***Abstract*—GPU computing has rapidly gained popularity as a way to achieve higher performance of many scientific applications. In this paper we report on the experience of porting a hybrid MPI+OpenMP molecular dynamics code to a GPU enabled CrayXK7 to make a hybrid MPI+GPU code. The target machine, Indiana University's Big Red II, consists of a mix of nodes equipped with two 16-core Abu Dhabi X86-64 processors, and nodes equipped with one AMD Interlagos X86-64 processor and one Nvidia Kepler K20 GPU board. The code, IUMD, is a Fortran program developed at Indiana University for modeling matter in compact stellar objects (white dwarf stars, neutron stars and supernovas). We compare experiences using CUDA and OpenACC.**

***Keywords*-Molecular Dynamics; CUDA; OpenACC;**

I. Introduction

The advent of accelerators a few years ago came with a dramatic shift of paradigms in the area of high performance computing (HPC). So far, the majority of applications solely had to rely on MPI and OpenMP to leverage the potential of the many multicore CPUs available to them. The introduction of GPUs (and other accelerators), however, requires application developers to add a third level of parallelization, which differs substantially in the way it is programmed at the lower levels, providing a vast amount of parallel threads and extending the existing memory hierarchy by adding on-device memory and (asynchronous) memory transfers into the picture.

The fact that the majority of performance is now provided by accelerators on machines that provide them, requires application developers to adapt their codes to use as much of the provided performance potential as possible. The well established CUDA programming environment provides a powerful way of programming accelerators. However, it may also require deep thoughts about the memory and thread hierarchy available on the device. On the contrary, the OpenACC standard tries to provide developers with an easy to use way of porting applications using compiler annotations. In this paper, we compare the efforts of adapting a well known molecular dynamics code to both CUDA and OpenACC, and the performance achieved by each programming paradigm.

The remainder of the paper is structured as follows: Section II gives a brief review of the physics the target code performs. Section III gives an overview of previous work. Section IV dicusses the MPI algorithm. Section V provides information on the hardware used. The necessary adaptations to the code together with a performance comparison are presented in Sections VI and VII. Possible future work is discussed in Section VIII before conclusions are drawn in Section IX.

II. Molecular Dynamics and dense nuclear matter

A. Dense nuclear matter in astrophysics

IUMD is a classical molecular dynamics (MD) code for simulating dense matter in white dwarf stars, neutron stars and core collapse supernovae. For a gentle and pleasant introduction to the physics of *compact stellar objects* (CSO), see [1]. For an introduction to MD, see [2]. A white dwarf is the remnant of a sun-like star after it has gone through all its burning stages. After the last stage, there is no longer any pressure from nuclear fusion to resist the self gravity of the star, and it contracts to an object about the size of Earth. Atoms are completely ionized, and the liberated electrons form a degenerate Fermi gas whose pressure prevents further contraction. The electrons form a nearly uniform background of negative charge that exponentially screens the positive charge of the ions. Although very densely packed, white dwarf matter can be treated as a classical MD system where ions interact via the *screened Coulomb* potential,

$$V_{Cij} = \frac{Z_i Z_j e^2}{r_{ij}} e^{-r_{ij}/\lambda}. \tag{1}$$

In this expression Z_i and Z_j are the charge numbers of ions i and j, e^2 is the square of the proton charge (one factor for each ion), r_{ij} is the length of the position vector

$$\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j \tag{2}$$

of ion i relative to j, and λ is the screening length due to the electrons. Electrons are not modeled explicitly; their only effect is to provide the exponential screening factor in eq. (1). White dwarfs consist of carbon and oxygen with trace amounts of other elements, primarily neon. The force corresponding to (1) is,

$$\mathbf{f}_{Cij} = \frac{Z_i Z_j e^2}{r_{ij}^3} \left(1 + \frac{r_{ij}}{\lambda}\right) e^{-r_{ij}/\lambda} \mathbf{r}_{ij} \tag{3}$$

Neutron stars are the final, collapsed remnant of a Type II (core collapse) supernova. They are the end result of stars that are about 8 solar masses. During the supernova explosion, most of the mass is ejected, leaving about 1.4 solar masses compressed into an object about 10 km in radius. The mean density of a neutron star is thus very high, around 10^{13} that of ordinary matter. However, the outer 100 m or so consists of a mixture of ions similar to a white dwarf, but with many more, and different species. MD simulations of this matter are carried out in the same way as for white dwarfs, again using eq. (1) as the interaction.

Classical MD simulations of these ion systems, often called *Coulomb systems*, yield important insight into the physics of white dwarfs, and neutron star crust. The phase diagram (freezing temperature vs. composition) of carbon-oxygen, and carbon-oxygen-neon mixtures in white dwarfs can be derived from MD and compared against theory [3]. For the complex mixtures of ions in neutron stars phase diagram theory is inconclusive, and MD simulations acquire central importance. MD also yields insight into material properties such as thermal and electrical conductivity, and the stress–strain properties of neutron star crust that are important for interpreting observations.

Classical MD can be used to study matter deeper in neutron stars. At a depth around 100 m, ions begin to lose their identies. Nucleons (neutrons and protons) "drip" out of nuclei. As nuclei get near each other, their nucleons begin to interact directly via the short-range nuclear force. Theorists have predicted nucleons begin to form long spaghetti-like strands. At greater depth, the strands are believed to coalesce into lasagna-like sheets. Physicists have euphemistically coined the term *nuclear pasta* to describe these unusual phases of matter [4], [5]. Deeper yet, the lasagna sheets finally fuse together into almost uniform neutron matter.

Nuclear pasta is an interesting theory, but it is very hard to tell from observations of neutron stars and experiments that can be done on Earth, whether nuclear pasta actually occurs in nature. Classical Molecular dynamics again provides a way to bridge the gaps between theory, observation and experiment. Despite the enormous densities, and the fact that one is dealing with nuclear matter, treating nucleons as classical point particles subject to a classical force, and moving according to Newton's Second Law leads to useful physical results. Electrons again provide a screening background for the Coulomb potential between protons, so eq. (1) again applies, where now all $Z_i = 1$. In addition, nucleons interact via the strong nuclear force, whose essential physics for our purposes can be captured by the semi-classical two-body potential,

$$V_{Nij} = ae^{-r_{ij}^2/\Lambda} + \left[b + c\tau_z(i)\tau_z(j)\right] e^{-r_{ij}^2/2\Lambda} \qquad (4)$$

In this expression, $\tau_z = 1$ for a proton and $\tau_z = -1$ for a neutron. The constants a, b, c, and Λ are chosen so as to reproduce certain properties of pure neutron matter and symmetric nuclear matter (matter with equal numbers of protons and neutrons), and the binding energy of a few selected nuclei.

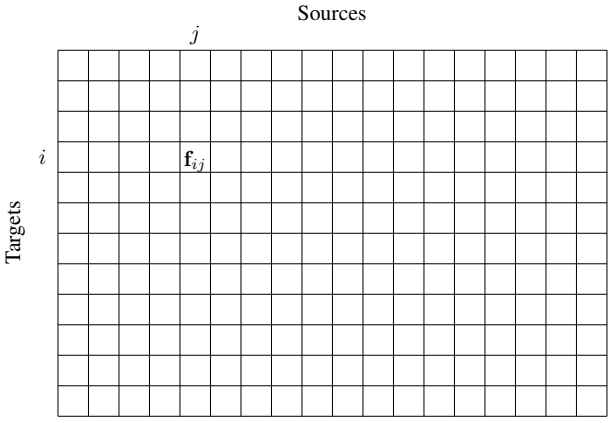

Figure 1. Force matrix. Source particles are listed along the top. Target particles are listed along the left side. Cell (i, j) represents the force that source particle j exerts on target particle i. The sum of all $\mathbf{f}_{ij}$ along row i (excluding the (i, i) term) is the total force on i.

Nuclear pasta has actually been observed in our MD simulations, and behaves much like theory predicts [6], [7]. What is more, by having full trajectory data of all the nucleons, we can make more precise and detailed predictions of its properties. This provides important input for theorists, observers and experimenters.

Nuclear pasta is also believed to exist momentarily in the shock front of a core collapse supernova as it propagates out from the collapsing core. Computational physicists who model supernovae have faced a problem for many years that their simulations tend to fizzle rather than explode. The properties of nuclear pasta gleaned from MD simulations may help solve this problem.

B. The Particle-Particle (PP) algorithm

IUMD uses the simple Particle-Particle (PP) algorithm to calculate forces. Let $\mathbf{N}$ be the number of particles, numbered from 0 to $\mathbf{N} - 1$. The force on i is,

$$\mathbf{F}_i = \sum_{\substack{j=0 \\ j \neq i}}^{\mathbf{N}-1} \mathbf{f}_{ij} \qquad (5)$$

where $\mathbf{f}_{ij}$ is the force *source* particle j exerts on *target* particle i. In the simplest PP algorithm, the $\mathbf{F}_i$ are calculated in a pair of nested DO loops. The outer loop goes over targets i, and the inner loop over sources j. Particle positions and velocities are updated by a velocity Verlet type of algorithm.

We can represent the force calculation in the PP algorithm by an $\mathbf{N} \times \mathbf{N}$ matrix as shown in Fig. 1, where element (i, j) represents $\mathbf{f}_{ij}$. In the full matrix, targets and sources are of course the same; however, we will need to distinguish

between targets and sources when we discuss submatrices of the force matrix below. Thus, Fig. 1 illustrates the general case, where sources and targets may be different particles, and of unequal number. In fact, the target and source arrays in each MPI process of IUMD are different arrays. For example, target and source coordinates are stored in arrays `xt(3,0:nt-1)` and `xs(3,0:ns-1)`, respectively, where `nt` and `ns` are the number of targets and sources the process is responsible for.

MD codes such as NAMD, AMBER, GROMACS and CHARMM, used for atomic and molecular systems, use more efficient algorithms. However these depend on the interactions being either very short range, such as the van der Waals force between atoms, or the infinite range unscreened Coulomb force. The screened Coulomb systems studied with IUMD interact via an intermediate range force. For most simulations, the simulation volume size is comparable to the range of the force. Thus algorithms for short range forces are not that useful. Algorithms for the unscreened Coulomb force are also not useful, since they depend on the specific $1/r$ dependence of the potential. There is a fast multipole method available for the screened Coulomb interaction, but it remains for future work to determine if it would really improve performance.

There are two important optimizations we do not use in IUMD 6.3.0. One is the Newton's Third Law trick. Since $\mathbf{f}_{ji} = -\mathbf{f}_{ij}$, each pair force needs to be calculated only once. This cuts the amount of work in half; however it complicates parallel implementations, even with OpenMP. For the OpenACC and CUDA kernels, synchronizations and memory transactions would have been required that were problematic, and possibly detrimental to performance. We therefore reverted to the simpler method of calculating the full force matrix. Moreover, our MPI algorithm is not amenable to Newton's Third Law. Second, a cutoff radius r_c could be employed, so that only particles within r_c of each other interact. For the typical 27,648-ion problems for which IUMD is used, r_c equal to about half the box edge length gives good physical results. The cutoff sphere then occupies about half the volume, resulting in another factor of two reduction. However, doing a distance test for each particle pair causes branching in the GPU code that might severely impair performance. Most of the advantage of the cutoff would then be lost.

Our benchmarking philosophy for this paper was to compare implementations of the same algorithm on different architectures. Thus we used a PP algrorithm without the above optimizations for the baseline OpenMP runs as well as the OpenACC and CUDA runs. If we had used an optimized OpenMP code for baseline runs, our speedup numbers in section VII might have been lower.

III. Previous Work on IUMD

In this paper we discuss IUMD 6.3.0. This version can only perform ion simulations. It does not implement the strong nuclear interaction of eq. (4). Previous versions of IUMD have been in use for over nine years and grew out of a serial Fortran 77 program. The first port to a high performance platform was an MDGRAPE-2 port in 2004. The MDGRAPE-2 was a special purpose PCI board designed specifically to do molecular dynamics for systems where particles interacted by central forces [8]. In effect, it was an accelerator for one class of MD problems, and it was very fast for its time, equal to about 84 cores of the current Kraken machine installed at the Oak Ridge National Lab (ORNL).

The first hybrid MPI+OpenMP version of IUMD was developed in 2006. It employed a simple MPI algorithm where targets were distributed across MPI processes, and processes applied the complete set of sources to their targets. Targets were further distributed among OpenMP threads on each process. After each process did the velocity Verlet update on its targets, they participated in an MPI_Allgather to update source positions. An MPI+MDGRAPE-2 version of this code was available for the four MDGRAPE-2 nodes Indiana University had in its IBM SP2 machine. Contrary to what one might guess, this simple MPI+OpenMP algorithm scaled quite well. Versions 6.0.x, employing this algorithm, were used for several years on Indiana University's MDGRAPE-2 machines, BigRed and Kraken. Nearly linear speedup was obtained for problems involving 27,648 ions, for up to 128 MPI processes, with either 4 OpenMP threads (on BigRed) or 6 OpenMP threads (on Kraken) per process.

A new MPI+OpenMP algorithm was developed in 2010 for IUMD 6.1.0 that used Newton's Third Law and a cutoff. Target particles were again distributed among MPI processes, but now sources on each process were distributed among OpenMP threads. IUMD 6.1.0 was the subject of extensive tuning and benchmarking efforts in 2011 focused on serial performance [9]. IUMD 6.1.0 and 6.2.0, a follow-on version which implemented additional phyics, have been used in production on BigRed and Kraken since summer 2010.

IV. MPI version of the PP algorithm

IUMD provides a framework that allows several force kernels to be plugged in. IUMD has a kernel that can be compiled as a serial Fortran code or an OpenMP code. There is also an OpenACC kernel, and a Portland Group CUDA Fortran kernel. Each of these kernels is called by a control subroutine that implements an MPI algorithm to combine results. The serial code calls stub MPI routines, so the same source can be used for serial and MPI programs. In this section we describe the design of the MPI algorithm, and the OpenMP kernel. We describe the OpenACC and CUDA kernels in following sections.

The MPI algorithm used by IUMD partitions the force matrix into P block rows and Q block columns, and assigns one block to each process of a corresponding Cartesian grid of PQ MPI processes. See Fig. 2. Targets are block distributed down each process column, with $M = \mathbf{N}/P$ targets assigned to each process. Sources are block distributed along each process row, with $N = \mathbf{N}/Q$ sources assigned to each process. The horizontal colored bars in Fig. 2 show how sources are distributed. For example, each block in column 0 is assigned the same block of red sources. Each block in column 1 is assigned the green sources, and so on. Likewise, each process in a row is assigned the same block of targets (not shown in the figure). In the pure MPI program each process calls the serial kernel described above, which now calculates only the forces its sources exert on its targets. The OpenMP kernel is a simple variation of the serial kernel, where an `$omp parallel do` is applied to the i (target) loop.

After returning from the kernel, the control subroutine performs an MPI_Allreduce over all blocks of its row to get the total force $\mathbf{F}_i$ for each of the row's targets. The allreduce returns the forces to all processes of the row so that each process can independently update positions and velocities of its targets using the velocity Verlet integration routine. Although each process in a row updates the same targets, this requires only about $33M$ flops per process. It is most likely faster to repeat this in each process than to distribute the work among proceses, and communicate the $3M$ target coordinates back to all of them. Note also that the work involved in the velocity Verlet update will be insignificant compared to the order MN work of calculating forces when M and N are large.

After all target postions have been updated, they need to be distributed as source positions for the next MD time step. Another MPI_Allreduce, now one for each process column, accomplishes this. Each process supplies target coordinates for its targets that are among its column's sources, and zeros for those that are not. Summing over these with an allreduce then results in just that column's sources being returned to each process. Fig. 2 illustrates this. The vertical red bar in process $(0, 0)$ represents the targets that must be distributed to all processes in column 0. Process $(0, 0)$ contributes these targets to the MPI_Allreduce, while all other contributions are set to zero. A column-wise MPI_Allreduce then delivers the red targets to each process.

V. Hardware

Development and performance measurement of the GPU enabled code was done on Cray XE6/XK7 compute nodes that are part of a small test and development system for Indiana University's upcoming Big Red II supercomputer. The test bed, as well as Big Red II, are hybrid XE6/XK7 systems, equipped with NVIDIA Tesla K20 GPUs.

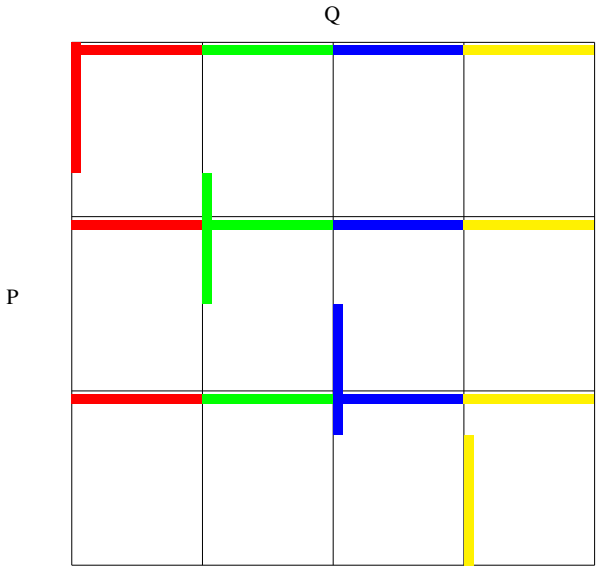

Figure 2. MPI process grid. The force matrix is partitioned into a $P \times Q$ grid of MPI processes. Each process is responsible for applying $\mathbf{N}/Q$ sources to $\mathbf{N}/P$ targets. In a pure MPI program, the processes use the simple PP algorithm of Fig. 1. The horizontal colored bars represent source particles. Each row has the complete set, distributed blockwise along the row. Likewise, all columns have the complete set of targets distributed blockwise. The vertical colored bars represent targets in each column that must be distributed to all processes in that column to update the sources after the velocity Verlet update.

A. Cray XE6 / XK7

The Big Red II test and development system is a hybrid XE6/XK7 system with twelve XE6 nodes and four XK7 nodes. The XE6 nodes contain two AMD Abu Dhabi 6380 16 core processors running at 2.5 Ghz. Each XE6 node has 64 GByte of main memory. The XK7 nodes are equipped with a single AMD Interlagos 6276 16 core processor running at 2.3 GHz and a single NVIDIA Kepler K20 GPU. The XK7 nodes have 32 GByte of main memory and 5 GByte of GPU memory. The GPU is connected to the processor via a PCIe-Bridge and a HyperTransport3 connection, capable of a transfer speed of 8 GByte/sec between the processor and the GPU. The compute nodes are interconnected with the Cray proprietary Gemini interconnect.

B. NVIDIA K20

We briefly highlight the architecture of the Kepler K20 GPU. For a full description of the architecture, as well as the CUDA programming environment, see [10]. The Kepler K20 GPU consists of 13 streaming multiprocessor (SMX) units. Each SMX has 192 single precision cores, 64 double precision cores, 32 special function units, 65536 registers and 64 KB of shared memory. The K20 has 5 GB of *device* memory, which serves as global, constant and local memory for all SMXs. It has a latency of 400 to 600 cycles. Shared memory is partitioned into a user-managed shared memory

and an L1 cache. There is always some fast shared memory allocated to L1 for caching reads and writes to device memory.

Kernel threads are grouped into *thread blocks*, which can have 1, 2 or 3 dimensions. A thread in a thread block of dimensions (m, l) has *thread index* (i, j), and corresponding *thread ID* $i + jm$. Registers are partitioned among threads for their private use. The K20 has compute capability 3.5, which allows 1024 threads per block. Blocks are organized into a *grid* of thread blocks of dimension 1, 2, or 3. Each block is mapped to an SMX, where it stays during its lifetime. At most 16 blocks may be resident on an SMX. Shared memory is partitioned among the resident blocks. Blocks cannot access each others shared memory. Threads are scheduled in *warps* of 32 threads each, with consecutive thread IDs. Best performance of a kernel is obtained when all 32 threads of each warp execute the same instruction path. If branches occur, the warp is said to *diverge*. In this case all threads of one path are scheduled, followed by all threads of the next path, and so on. The resulting serialization therefore negatively impacts performance.

The data path to device memory is 128 bytes wide. Each data value is read/written by one thread of a warp. If addresses are within the same 128 byte memory segment, the reads/writes will be coalesced, meaning the I/O is done in one transaction. For devices of compute capability 1.0, data order must exactly match thread ID order. This requirement is relaxed for compute capabilities 2.x and 3.x. For 3.x, the only requirement is that all data lie in the same segment.

VI. Porting to GPU

The Kepler K20 GPU has 2496 single precision cores, 832 double precision cores and 416 special function cores, all running at 705.5 MHz. This is a powerful computational capacity. The calculation of pair forces $\mathbf{f}_{Cij}$ has a high arithmetic intensity, in addition to being straightforward to program. Once particle data are in registers and shared memory, the calculation will go very quickly. However, each pair force requires 11 memory transacations. The main issue therefore is not the processing capabilities of the cores, but getting data into and out of them. This is especially true of host-device transactions, which must go across the PCIe. Each time step, the host program must transfer all target and source data to GPU device (global) memory. After the kernel is launched, threads must then read data into either shared memory or their private registers. The maximum number of bytes per device memory transaction is 128, and only when they are coalesced. Each device memory transaction incurs a latency of 400 cycles. Even when coalesced, latency is an important issue. If reads are serialized, there are fewer bytes per transaction. The latency could be especially damaging to performance if incurred by each of the 11 memory transactions for an $\mathbf{f}_{ij}$. Thus the main issue in algorithm design is to minimize the impact of memory transactions.

In this section we describe two methods for programming GPUs, and our attempts to optimize memory transactions in each. First we describe programming with OpenACC, a set of directives for GPUs that is similar to OpenMP for multi-threaded programming on ordinary CPUs. We then describe a CUDA program based on Portland Group's CUDA Fortran.

A. OpenACC

The goal of OpenACC is to provide users with an intuitive, less intrusive way of developing code to run on accelerators of different types [11]. It provides an abstract interface that does not contain instructions specific to any hardware. Instead, developers annotate code using pre-processor statements, a concept that has proven to be useful in OpenMP. In fact, using a thread-parallel implementation that employs OpenMP such as IUMD might serve well as a starting point for developing OpenACC code.

However, simply replacing the OpenMP statements with OpenACC directives is not always sufficient. The OpenMP standard defines certains types of directives that are not to be found in the OpenACC standard, e.g., synchronization and scoping directives. Hence, the computationally extensive parts of the application might need to be rewritten to match the requirements of OpenACC. In many cases, parts that use synchronization operations can be replaced by redundant computation to make use of the massive parallel nature of the GPU. The goal is to have nested loops that are not data-dependent.

A first port of the main kernel to OpenACC was straightforward. The occurence of `!$omp parallel` was replaced by `!$acc kernels` with the loops being annotated by `!$acc loop`. While the PGI compiler is capable of detecting required reduction operations based on data dependencies, we had to explicitly specify reduction statements to obtain correct results when using the Cray compiler. For both compilers, the reduction of Fortran array variables is not supported.

1) Optimizing Data Transfers: The OpenACC standard allows for explicit data management of data allocated on the device using the `!$acc data` directive. By default, device memory is allocated and data copied every time a `kernels` or `parallel` region is entered. The memory is then deallocated as soon as the region is left, incurring additional overhead.

It is therefore desirable to explicitly control the lifetime of data objects on the device. Some of the data used in the main kernel is valid over the runtime of the application. It is created upon initialization and only read from there on. This was a first step to drastically reduce the amount of data copied on every invocation of the kernel by using a global `data` directive after the initializing to copy the constant

data onto the device. No further copying of this data is then necessary.

Furthermore, the OpenACC standard provides programmers with a way to asynchronously copy data to and from pre-allocated device memory using the `!$acc update async` directive. This can be used to mitigate the cost for device memory allocation and to overlap computation and memory transfers. In the case of IUMD, we were able to overlap the transfer of particle coordinate arrays with the velocity Verlet integration.

2) Using Scalar Variables: The original version of the kernel used arrays on the stack containing three elements each to hold temporary distance vectors and the pressure tensor, mainly to allow the Fortran compiler to apply vector optimizations. However, the OpenACC compiler was able to produce significantly faster code if the intermediate results and constants were stored in scalar variables. This requires the developer to manually unfold loops and introduce and keep track of additional variables in the code. This optimization is only viable where arrays are small enough.

3) Sequential Execution of Remaining Loops: As described in Section IV, the Verlet integration routine only requires a small amount of computation compared to the main kernel. It came as a surprise that the computation of this integration was more efficient when done sequentially as opposed to using OpenMP threads. It seems that the overhead for creating and synchronizing threads outweighs the benefit of doing the computation in parallel. Hence, all remaining loops are executed sequentially.

No further performance optimizations have been applied to the OpenACC code.

B. PGI CUDA Fortran kernel

In this section we describe a carefully designed CUDA kernel. In section VII we compare benchmark results of this kernel against the OpenACC kernel. In the CUDA kernel, we again focus our effort on minimizing host-device memory transactions, the number of global memory transactions on the device itself, and optimizing those that are unavoidable. We make maximum use of shared memory and the large number of registers available to cores.

Consider an $M \times N$ submatrix of the full force matrix. Fig. 1 suggests one way of mapping the matrix onto a GPU: partition it into a grid of submatrices of dimensions (m, n), and assign each $\mathbf{f}_{ij}$ to a thread of a corresponding grid of (m, n)-dimensional CUDA thread blocks. However, this may not be the best use of resources. The number of threads per thread block is limited to 1024 for devices of compute capability 3.x, like the K20, and even less for previous compute capabilities. For large $\mathbf{N}$, the number of thread blocks required might be too large. Worse yet, if data is not laid out carefully in device global memory, and memory accesses are not carefully orchestrated, the 11 memory transactions for each $\mathbf{f}_{ij}$ may not be coalesced. Reads might be partially or even completely serialized, each one incurring the 400 cycle latency of device memory.

Thus we take a slightly different approach. We still use a two-dimensional thread block, now of dimensions (m, l). However, as we will explain below, threads in the same row apply sets of $m/4$ sources to the same target. Thus each thread block is responsible for applying

$$n = lm/4$$

sources to m targets. We also partition the sources into C chunks, with

$$N_C = N/C$$

sources per chunk. By dividing the matrix into submatrices of size (M, N_C), and looping over the C source chunks, we can keep target data and accumulated forces in registers longer, providing further opportunity for optimization. The user selects C, m and l subject to the restrictions that m exactly divide M, it be a multiple of the warp size 32, and that $lm/4$ exactly divide N_C. Thus the $M \times N$ force matrix is partitioned into C submatrices of dimensions (M, N_C), each of which is further partitioned into a grid of submatrices of dimensions (m, n). Each of these is assigned to thread block of dimensions (m, l). The grid dimensions are,

$$\begin{aligned} A &= M/m \\ B &= N_c/n. \end{aligned}$$

The basic CUDA kernel is shown in Fig. 3.

At the beginning of each time step, the host subroutine copies target and source data to device global memory. Target data is ordered so that all x-coordinates are stored first, followed by all y-coordinates, and then all z-coordinates. Source data is copied in transpose order. The charge Z_j, and coordinates (x_j, y_j, z_j) are stored consecutively for each source particle.

In Step A of the CUDA kernel, threads read their target's coordinates x, y and z from global memory. Since the thread ID of thread (i, j) is $i + jm$, thread order exactly matches data order. Since we required m to be a multiple of the warp size, these reads will therefore be perfectly coalesced even for devices of compute capability 1.0. Moreover, since each thread column reads the same set of targets, the remaining columns may read their data from cache. Recall that for compute capability 3.x, part of shared memory is reserved as an L1 cache for reads from device memory. Each thread keeps its target's coordinates in registers throughout the calculation. Fig. 4 shows an example where $(m, l) = (32, 2)$.

In step B, threads read in a chunk of source data and write it to shared memory, where it can be accessed at low latency and high bandwidth throughout the calculation. Since source data is stored in the order $(Z_0, x_0, y_0, z_0), (Z_1, x_1, y_1, z_1), \ldots$ in global memory, each set of 4 threads reads one particle's data, and each

```
!Step A: Read target coordinates into registers.
xt=xtg(a*m+i,1)
yt=xtg(a*m+i,2)
zt=xtg(a*m+i,3)
fx=0.0; fy=0.0; fz=0.0

do c=0,C-1

  !Step B: Copy chunk c of source data from global
  !to shared memory.
  xss(mod(i,4),j*(m/4)+i/4) =  &
      xsg(mod(i,4),(c*B+b)*n+j*(m/4)+i/4)
  call syncthreads

  !Step C: Accumulate force.
  F = 0
  do k=j*(m/4),(j+1)*(m/4)-1
    F = F + f(k)
  end do
  call syncthreads

enddo

!Step D: Combine contributions to F from threads.
!of my block.
if(j.eq.0) then
  Fs(i) = F
endif
call syncthreads
do k=1,l-1
  if(j.eq.k) then
    Fs(i) = Fs(i)+F
  endif
  call syncthreads
end do

!Step E: Copy Fs to Fbg.
  Fbg(a*m+i,b) = Fs(i)

!Step F: Sum over b.
  F = 0
  Do b=0,B-1
    F = F+Fbg(a*m+i,b)
  end do

!Step G: Repeat step D.

!Step H: Write to Fg
  Fg(a*m+i) = Fs(i)
```

Figure 3. Basic steps in CUDA kernel. Thread (i, j) of block (a, b) reads coordinates for target $am + i$, then loops over C chunks of sources, accumulating their forces on the target. Variables ending in `g` are in global memory, those ending in `s` are in shared memory. Remaining ones are in registers. f(k) represents the calculation of the force that source `k` exerts on the thread's target. Note that we distinguish between c and C, and between b and B in this bit of Fortran code, to maintain consistency with discussion in the text. Fortran does not normally distinguish upper and lower case.

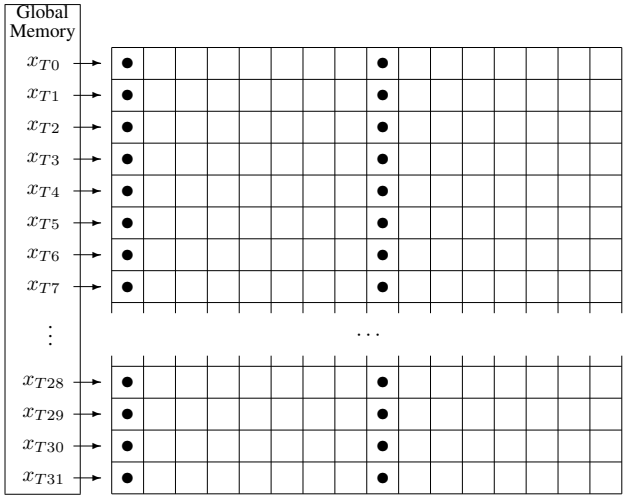

Figure 4. This figure shows how a thread block of dimensions $(m, l) = (32, 2)$ reads target particle x-coordinates from device global memory. Threads are represented by the black circles. Each thread column comprises a complete warp. The 32×16 matrix of cells is the submatrix of the force matrix the thread block is responsible for. Each thread column reads the same target data into registers. Since data is read from consecutive memory locations by consecutive threads, reads are coalesced for all compute capabilities. After reading the x-coordinates, the process is repeated for y- and z-coordinates. The thread block shown here represents any block in row 0 of the grid of thread blocks. Blocks in succeeding block rows read successive sets of 32 targets.

warp reads 8 particles' data, again in a perfectly coalesced fashion. Fig. 5 illustrates how this works. We again take $(m, l) = (32, 2)$. The 32 threads of column 0 read $(Z_0, x_0, y_0, z_0), \ldots, (Z_7, x_7, y_7, z_7)$ from global memory and write it to shared memory. Meanwhile, the 32 threads of column 1 read $(Z_8, x_8, y_8, z_8), \ldots, (Z_{15}, x_{15}, y_{15}, z_{15})$. Because thread order matches data order, and because of the fortuitous circumstance that the number of data elements for each source exactly divides the warp size, these reads are perfectly coalesced, even for compute capability 1.0. This is therefore the optimal way to read source data.

Once the sources are in shared memory, we come to Step C, which is the actual force calculation. Each thread applies $m/4$ sources to its target. With target coordinates in registers, and source charges and coordinates in shared memory, memory transactions are the most efficient possible for given (m, l).

Steps B and C are repeated for each chunk of sources. Target coordinates and the accumulated force on the target are kept in registers throughout the thread's lifetime. The freedom to choose C, m and l allows one to select parameters that optimize the overall runtime of the kernel. In section VII we will discuss benchmark runs we have done to determine an optimal parameter set.

At the end of the loop over chunks, each thread has its contribution to its target's force in registers. The remainder of the algorithm is summing these partial forces to get

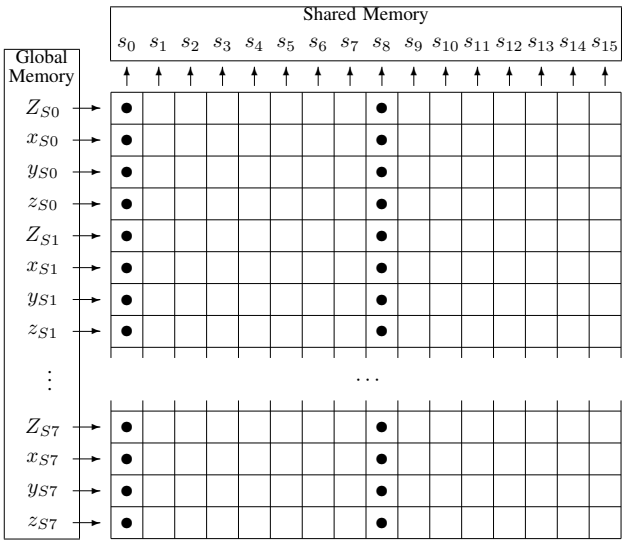

Figure 5. This figure shows how a thread block of dimensions $(m, l) = (32, 2)$ reads source particle data from device global memory into the block's shared memory. Threads in column 0 read $(Z_{Sj}, x_{Sj}, y_{Sj}, z_{Sj})$ for $j = 0, 1, 2, \ldots, 7$. Since the data is read from consecutive memory locations by consecutive threads, the reads are coalesced for all compute capabilities. Source data written to shared memory are denoted by $s_j = (Z_{Sj}, x_{Sj}, y_{Sj}, z_{Sj})$. While thread column 0 is reading $s_0, \ldots, s_7$, thread column 1 reads $s_8, \ldots, s_{15}$. The thread block shown here represents any block in column 0 of the grid of thread blocks. Blocks in succeeding block columns read successive sets of 16 sources.

the total force $\mathbf{F}_i$ for each target. First, in Step D the $\mathbf{F}_i$ from each thread column of a block are combined to give the contribution from the entire block. Since threads cannot access each others' registers, this reduction is done through array $\mathbf{F}_s$ in shared memory. Threads could add their contribution to $\mathbf{F}_s$ by using the CUDA `atomicAdd` function; however, this is likely to be inefficient. Another way is for thread $(i, 0)$ to first copy its registers to $\mathbf{F}_{s,i}$, after which all threads of the block synchronize, and then threads $(i, 1), \ldots, (i, l-1)$ take turns adding their contributions, with a synchronization between each one. There are more efficient reduction algorithms, but for small l, say $l <= 8$, this simple method should work. Since all m threads of a column update separate locations, updates will go by half-warps at a time. Moreover, access to shared memory is fast, and synchronizations go warp-by-warp. Note that if $l = 1$, this step is not necessary.

Next we need to accumulate the partial forces along each block row. Since blocks cannot access each others' shared memory, we must finally give in and do this through global memory. However, there are some optimizations. Since global memory is large, 5 GB on the Kepler K20, there is plenty of room for a large array. So in Step E each block (a, b) writes its forces to an array section reserved exclusively for it. This completely removes the need for synchronizations among thread blocks. Moreover, writes will be maximally coalesced, since the order of threads in thread columns matches the order of addresses they write to. We also take advantage of the multiple thread columns. Since the force has three components, if $l > 1$ we can use different columns to write the different components. Note that global memory accesses in this step as well as in steps F and H follow the same pattern as in Fig. 4.

The last block in a row to write its forces to global memory is tasked with reading and accumulating the results. This is done in Steps F, G and H. It takes several expensive reads from global memory, one for each force component for each of the B array sections, but we again optimize by having thread columns share the work. Reads will again be maximally coalesced.

VII. Performance Comparison

We compared the OpenACC and CUDA kernels using different numbers of particles as input. The number of particles was carefully chosen to reflect the number of particles per node with an increasing number of MPI processes. In this scenario, the baseline is 27,648 particles on one node. Increasing the number of nodes to 4 in the form of a 2×2 process grid without increasing the overall number of particles results in 13,824 particles per node. Similarly, if we distribute the 27,648 targets and sources over square process grids of dimensions $4 \times 4, 8 \times 8, 16 \times 16, 18 \times 18$, and 24×24, there are 6912, 3456, 1728, 1536, and 1152 sources and targets per node (one MPI process per node). The intent is to estimate the performance behavior when employing strong scaling, neglecting any overhead induced by MPI communication. Performance of the MPI code is limited by single node performance.

A. CUDA kernels tested

Fig. 3 shows the CUDA kernel consists of eight steps. The later steps involve expensive device memory transactions. We wrote several variations of the basic CUDA kernel to test the effect of doing some of these steps on the host.

If all steps are done in the kernel, all the host has to do is copy the forces from device global memory back to host memory. We wanted to see the effect of doing some of the later steps on the host, rather than the GPU. Steps E, F and H comprise a great deal of memory accesses to global memory. What would be the effect if after Step E, the host reads the partial results from `Fbg` and does the final reduction? On the other hand, the result of Step E is a large amount of data that has not yet been reduced. The overhead of copying all of it back to the host for the final reduction might outweigh the cost of reducing it on the device.

To test this, we wrote kernels 03b, 05b, 06b. Increasing numbers indicate more of the calculation is done on the GPU. The letters "a", "b" indicate how reductions are

	03a	03b	05a	05b	05c	06b
Step A	K	K	K	K	K	K
Step B	K	K	K	K	K	K
Step C	K	K	K	K	K	K
Step D	A	C	A	C	-	C
Step E	K	K	K	K	K	K
Step F	H	H	K	K	K	K
Step G	-	-	A	C	-	C
Step H	-	-	K	K	K	K

Table I
TABLE SHOWING WHICH STEPS OF THE CUDA KERNEL ARE DONE ON THE HOST AND DEVICE. "K" INDICATES THE STEP IS DONE ON THE DEVICE. "H" INDICATES IT IS DONE ON THE HOST. FOR REDUCTIONS, "A" INDICATES `atomicAdd` IS USED, "C" INDICATES A COORDINATED SUM AMONG THREAD COLUMNS, "-" INDICATES THE STEP IS NOT NEEDED

done on the GPU. The letter "a" means the kernel uses `atomicAdd` to accumulate sums. This is easy to program, but probably not efficient. At any rate, we wanted to find out how (in)efficient it was. The letter "b" indicates a coordinated or synchronized sum, without `atomicAdd`s. These should be faster, but require synchronization between thread columns. Letter "c" indicates a variation of the kernel that allows only one thread column per block (i.e., $l = 1$). Table I shows for each kernel, which steps are done by the host and which by the kernel. In the table "K" indicates the step is done by the kernel, while "H" indicates it is done by the host. For reduction steps, "A" indicates atomic adds, "C" indicates coordinated sum.

Doing a complete (C, m, l) parameter sweep over all cuda kernels was too ambitious a task for this paper. Instead, we did a few experiments with the atomic add kernels. Then for each of the other kernels we chose some (C, m, l) randomly, and others based on expectation they would perform well. Our goal was to do this for each kernel, but time constraints prevented even that. Nonetheless, the parameter sets we chose did reveal at least the broad outline of the kernels' performance, and provided data for comparison with the OpenACC and OpenMP kernels. A more complete parameter sweep can only result in finding better performance.

B. Performance Comparison

Table II shows for each problem size the CUDA kernel and (C, m, l) parameters that gave the best performance. The performance metric used is the wall clock seconds to complete one call to `subroutine newton`. This subroutine encapsulates almost all the work of each time step, including host and kernel portions of the force calculation, and the velocity Verlet update. Typically, over 99 % of wallclock time per MD step is spent in `newton`.

Figure 6 shows the time per call to `newton` of the 32-thread OpenMP runs together with the CUDA and OpenACC runs on a logarithmic scale. Most strikingly, it shows the drastic difference between the CPU run and the GPU runs, which is up to one order of magnitude and more for the

N	Kernel	C	m	l	msec
1152	cuda05b	04	32	04	0.801
1536	cuda06b	04	64	02	1.091
1728	cuda05c	02	96	01	1.231
3456	cuda05c	02	128	01	3.152
6912	cuda05c	04	128	01	10.148
13824	cuda05b	04	128	03	41.389
27648	cuda05c	08	128	01	140.982

Table II
BEST PERFORMING CUDA KERNEL AND PARAMETERS FOR EACH PROBLEM, AND THE WALLCLOCK MILISECONDS TO DO ONE CALL TO THE NEWTON SUBROUTINE.

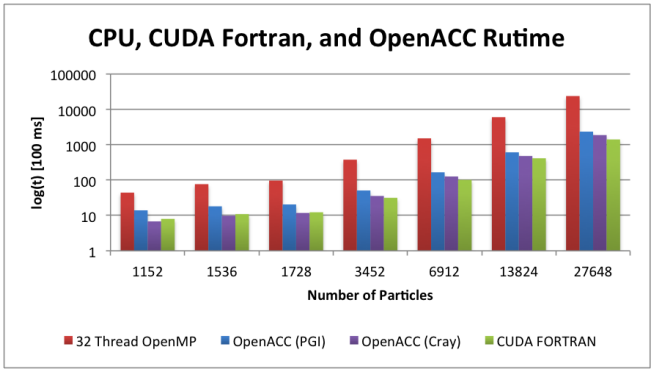

Figure 6. Runtimes of the CPU, CUDA Fortran, and OpenACC kernels for different number of particles (logarithmic scale). One node was used.

optimized CUDA version. It also shows that this difference in runtime gets higher with an increasing number of particles. This can be explained by better performance of the OpenMP version for smaller problem sizes, where the CPU cache is better utilized, and a larger overhead for the GPU versions, where data has to be copied in and out. This is also reflected in Figure 7, which shows that the utilization of the accelerator improves with an increasing problem size. The explanation is that the computation increases quadratically, which amortizes the required data transfer overhead. We caution that these graphs should be read backward as well as forward. In an MPI code, the number of pariticles per node decreases as the number of MPI processes increases for fixed problem size. The graphs show the best per node performance one can hope for in the MPI code when using a square process array on a 27648-ion problem.

We note as an aside that the 27648-ion problem we used in these benchmarks is similar to problems we used to run on the MDGRAPE-2. We find our CUDA enabled code runs the newton subroutine 5.5 times faster than the old MDGRAPE-2. This is a remarkable improvement, considering the MDGRAPE-2 was designed specifically for this kind of molecular dyanmics problem.

Figure 8 shows the speedup of the CUDA Fortran and OpenACC versions using a 32 thread CPU run as the baseline. It becomes clear that the speedup of the accelerated versions depends on the problem size. Again the influence

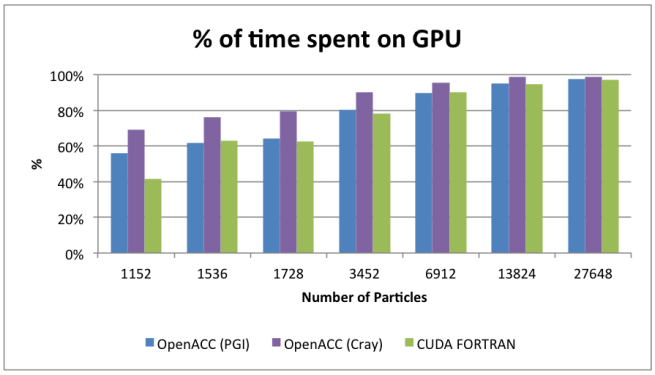

Figure 7. Percentage of time spent on the GPU per timestep.

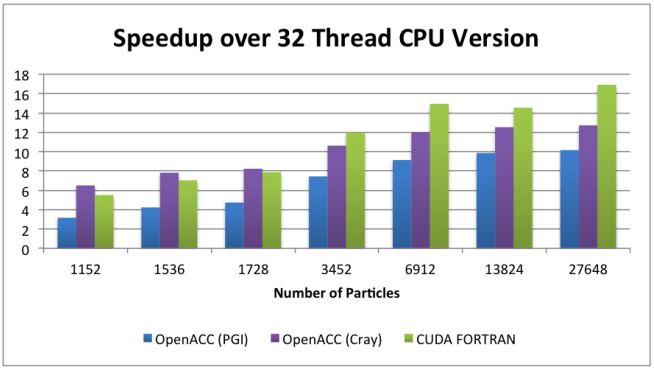

Figure 8. Speedup of the CUDA Fortran and OpenACC version when compared to a 32 thread CPU version

of the data transfer overhead becomes smaller and smaller for larger problem sizes due to the quadratic increase in computation.

It is also clearly visible that the CUDA Fortran version outperforms the OpenACC versions by up to 25 %. Since the developer has very fine-grained control over the placement of data on the device, the code can leverage much more of the potential of the accelerator. However, even the 13x speedup of the OpenACC version over a 32 thread CPU run for the largest problem size is remarkable considering the little changes that had to be made to the code compared to the effort of porting the application to CUDA Fortran.

Another point to make is the difference in performance between the PGI and Cray compiler. It seems that the Cray compiler is capable of producing code that is at least 30 % faster for all problem sizes. The advantage of the Cray OpenACC version over the CUDA version for smaller amounts of particles should be attributed to non-optimal kernel parameters (see Section VII-A).

Figure 9 shows a typical variation in performance of the CUDA kernel as one varies (m, l). Plots are shown of time to execute the cuda05b kernel vs. l for $m = 64,\ 128,\ 256$. The number of chunks is $C = 3$ in this figure. It is hard to find a pattern in this plot, other than the kernel tends to perform

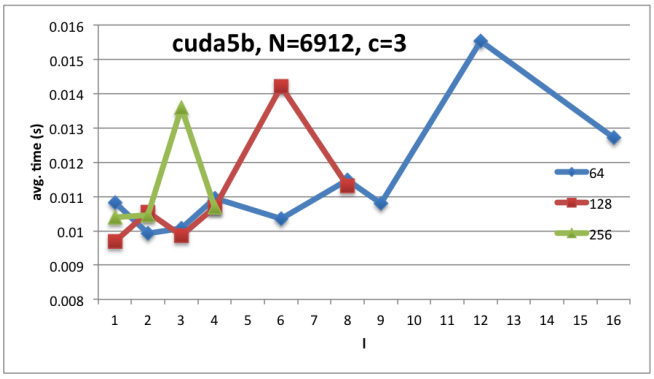

Figure 9. Average wallclock seconds per call to cuda05b for 6912 ions. Shown are plots of time vs. l for $m = 64,\ 128,\ 256$. Times are for the kernel only, not the whole newton subroutine.

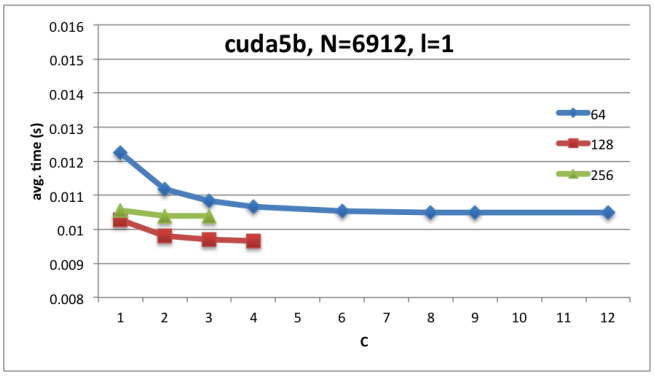

Figure 10. Average wallclock seconds per call to cuda05b, for 6912 ions. Shown are plots of time vs. C for $m = 64,\ 128,\ 256$

better for lower l. However, it does show that varying the thread block dimensions has a large effect. Figure 10 shows the effect of increasing C. In this plot, $l = 1$ and curves are shown for wallclock time for cuda05b vs. C for $m = 64,\ 128,\ 256$. It is clear that increasing the number of chunks improves performance. This is probably because the grid dimension B decreases as C increases, which decreases the amount of device memory transactions in the later stages of the algorithm. These figures provide only a brief look into the effect of different parameters. It remains for future work to investigate (C, m, l) parameter space more completely.

VIII. Future Work

Currently, the OpenACC version only runs correctly when compiled with the PGI and Cray compilers. Future work should make sure that all OpenACC enabled compilers (PGI, Cray, CAPS) are supported. Furthermore, it might be worth looking at other kernels in the application to port to the accelerator using OpenACC to further improve the runtime. This step might require the use of GPUDirect to enable access to on-device buffers for MPI operations in order to reduce the data transfer overhead.

We focused on single node performance in this paper. Although we have some upper bounds on performance of an MPI code from testing different problem sizes, we need to test the MPI version properly.

We need to more thoroughly explore (C, m, l) parameter space for each of our CUDA kernels, and see if there might be a better kernel. It is interesting that the Cray compiler produced OpenACC code that outperformed our kernels on small problems. We may not have found the best kernel and parameter combination. The Cray compiler may have produced better code for host-device memory transaction. It may also have produced a better kernel. We would also like to see if it can produce better OpenMP code than PGI.

We only implemented the screened Coulomb interaction in IUMD 6.3.0. Work is already in progress to include the short range nuclear force.

IX. Conclusion

In this paper we have layed out the process of adapting a single OpenMP based molecular dynamics application to leverage the potential of modern GPU accelerators. We layed out the changes necessary for porting the application to both OpenACC and CUDA, the two most important GPU programming paradigms at the time of this writing. We also presented performance data gained from running the adapted application.

As we have shown, the usage of OpenACC can provide a significant speedup over OpenMP (13x compared to 32 threads) while requiring only relatively small effort. For OpenACC, much of the existing code could be reused and just had to be adapted to fit the specifics of the programming paradigm. It is noteworthy that the performance of the code is highly dependent on the chosen compiler. However, we are confident that the compiler vendors will eventually close the gap and provide an equal level of performance for OpenACC. On the contrary, specifically engineering the code to run on a CUDA enabled GPU by using CUDA Fortran yields an even higher speedup since the developer has a very fine-grained control over the data placement and the execution on the device.

We consider OpenACC to be an easy to use entrance point into GPU programming that is very well suited to leverage the potential of modern accelerators. However, in order to achieve the maximum possible performance of the accelerator, developers still have to rely on CUDA programming, which might require reengineering parts of the application in order to meet the specifics of the memory and thread hierarchy.

Acknowledgment

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This material is based upon work supported by the National Science Foundation under Grant No. ACI-0338618l, OCI-0451237, OCI-0535258, and OCI-0504075. This research was supported in part by the Indiana METACyt Initiative. The Indiana METACyt Initiative of Indiana University is supported in part by Lilly Endowment, Inc. This work was supported in part by Shared University Research grants from IBM, Inc. to Indiana University

References

[1] I. Kaufmann, W. J., *Black Holes and Warped Spacetime*. San Fancisco: W. H. Freeman and Co., 1979.

[2] D. C. Rapaport, *The Art of Molecular Dynamics Simulation*. Cambridge Univ. Press, 2004.

[3] C. J. Horowitz, A. S. Schneider, and D. K. Berry, "Crystallization of carbon-oxygen mixtures in white dwarf stars," *Phys. Rev. Lett.*, vol. 104, p. 231101, Jun 2010. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevLett.104.231101

[4] M. Hashimoto, H. Seki, and M. Yamada, *Prog. Theor. Phys.*, vol. 71, p. 320, 1984.

[5] C. J. Horowitz, M. A. Pérez-García, and J. Piekarewicz, "Neutrino-"pasta" scattering: The opacity of nonuniform neutron-rich matter," *Phys. Rev. C*, vol. 69, p. 045804, Apr 2004. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevC.69.045804

[6] C. J. Horowitz, M. A. Pérez-García, J. Carriere, D. K. Berry, and J. Piekarewicz, "Nonuniform neutron-rich matter and coherent neutrino scattering," *Phys. Rev. C*, vol. 70, p. 065806, Dec 2004. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevC.70.065806

[7] A. S. Schneider, C. J. Horowitz, J. Hughto, and D. K. Berry, "Nuclear pasta formation," *(to be published)*.

[8] R. Susukita, T. Ebisuzaki, B. G. Elmegreen, H. Furusawa, K. Kato, A. Kawai, Y. Kobayashi, T. Koishi, G. D. McNiven, T. Narumi, and K. Yasuoka, "Hardware accelerator for molecular dynamics: Mdgrape-2," *Comput. Phys. Commun.*, October 2003.

[9] T. William, D. K. Berry, and R. Henschel, "Analysis and optimization of a molecular dynamics code using PAPI and the Vampir toolchain," *Proc. CUG-2012*, May 2012.

[10] *"CUDA C Programming Guide, v5.0"*, 2012. [Online]. Available: http://docs.nvidia.com /cuda/pdf/CUDA_C_Programming_Guide.pdf

[11] *"OpenACC Standard Version 1.0"*, 2012. [Online]. Available: http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf