Accelerating CoMD using GPUs

ABSTRACT

CoMD is a molecular dynamics proxy application used in the Department of Energy (DOE) "co-design" effort, which provides a suite of realistic applications for a platform to design future supercomputer systems from both the software and hardware perspectives. The upper echelon of the modern high-performance computing landscape is dominated by a wide assortment of heterogeneous hardware, due to its tremendous computing performance and memory bandwidth compared to traditional CPUs. However, many applications, including CoMD, do not take advantage of these accelerators at present. It is essential for "co-design" applications to support heterogeneous hardware so that the future design space can be thoroughly and accurately searched. Developing highly efficient and scalable hybrid applications that take full advantage of multi-GPU clusters using both MPI and CUDA is a difficult task, but it must be done to fully utilize machines. In this project, we add GPU support to CoMD, improving performance, making it more representative of production molecular dynamics codes, and increasing the relevance of this application for future design efforts. We highlight significant difficulties in dealing with MPI+CUDA hybrid applications and describe possible solutions to alleviate these difficulties. Finally, we attempt many common optimizations such as leveraging shared and constant memory and not so common ones that take advantage of the underlying GPU hardware, and characterize their efficacy for this type of HPC application. Our MPI+CUDA implementation achieves in excess of 12x speedup over the reference MPI implementation. We also demonstrate scaling across multiple GPUs on Blue Waters.

1. INTRODUCTION

The tremendous computational capabilities of GPUs have made them into a fixture of the current high performance computing landscape. Oak Ridge National Lab's Titan, the world's second most powerful supercomputer[1], derives most of its computational capabilities (24.5 petaFLOPS of its 27.1 petaFLOPS[2]) from GPUs of the NVIDIA K20X GPU architecture. Similarly, NCSA's Blue Waters and the Swiss National Supercomputing Center's Piz Daint use GPUs to obtain the majority of their impressive computational performance.

[1]`http://top500.org/lists/2013/11`

[2]`http://on-demand.gputechconf.com/supercomputing/2012/presentation/SB005-Bland-Titan-Oak-Ridge-National-Labs.pdf`

This impressive increase in computational capability has empowered HPC applications, previously limited mainly by existing paradigms of computation, to tackle ever larger problems and tackle problems at a finer granularity. One such application is simulation of molecular dynamics (MD). There has been much work in attempting to accelerate MD using GPUs [6], [7], [5].

The most computational intensive part of MD involves the evaluation of the forces acting on each atom due to all other atoms in the system and the numerical integration of the Newtonian equations of motion. Naïvely, this pairwise calculation of forces requires $\mathcal{O}(n^2)$ operations. This quickly becomes unmanageable when dealing with hundreds of thousands of atoms. However, due to the property that the magnitude of forces decrease with distance, cutoff techniques can reduce its complexity to a more manageable $\mathcal{O}(n)$. CoMD takes advantage of the cutoff technique.

In Section 2, we give a brief introduction on the CUDA and MPI paradigms and the significance of MD followed by an overview of CoMD and the Embedded Atom Model that CoMD uses to calculate forces. Section 3 discusses the results of profiling CoMD to ascertain which regions were computationally intensive and suitable for acceleration using GPUs. When offloading these regions, we kept the structure of CoMD as close to the original implementation as possible. In addition, we highlight significant difficulties when dealing with MPI+CUDA hybrid applications in Section 3.1 and possible techniques that can be attempted to overcome them. In Section 4, we describe our experimental setup and present results showing significant speed up with our basic implementation. We then perform different types of optimizations in order to show their effectiveness when dealing with HPC MPI+CUDA applications.

2. BACKGROUND

2.1 CUDA

Compute Unified Device Architecture (CUDA) is NVIDIA's parallel computing paradigm. To be more precise, it is a set of extensions to the C programming language that eases access to the GPU for general purpose parallel computing on accelerators. It is the most widely adopted programming platform for GPU development [4].

Both APIs use the CUDA programming model, which, at the highest level, has a special function called a kernel that is executed on the GPU. This kernel function is written using an inherently SIMD style, with multiple threads of execution exposed to the programmer. Kernels are invoked with two structural parameters, a block and a grid. A grid is a 1D, 2D, or 3D arrangement of logical blocks. The 3D arrangement is only supported on devices with compute capabilities of 2.x or higher [2]. The blocks themselves are composed of a 1D, 2D, or 3D arrangement of physical threads that execute the instructions present in the kernel function. Threads inside a block can synchronize execution among themselves, and have a small amount of collaborative memory called shared memory that is extremely fast. Other memory types present on the GPU with larger capacity but slower access times include texture memory and global memory. In recent versions of CUDA, CPU memory (also called host memory) can be accessed by the GPU without the need for explicit transfers of data to the GPU before launching the kernel. However, we did not use a version of CUDA which supports this in our project.

2.2 MPI

Modern HPC systems are distributed memory computers, composed of a collection of many individual shared memory compute nodes. Communication between two shared memory domains is accomplished by sending and receiving messages over a communication network. Messaging Passing Interface (MPI) is far and away the dominant paradigm of programming HPC systems. MPI presents a message passing API to developers, which includes primitive for point-to-point and collective communication and abstracts the underlying hardware to enhance the portability of parallel programs.

MPI applications embody the single program multiple data (SPMD) paradigm, where a single program source is executed by many processes, here known as ranks. Personalized control flow is achieved by parameterizing communication functions on rank IDs. MPI ranks belong to communicators that allow for the logical grouping of ranks for application needs as well as collective communications.

MPI which is designed to be both scalable and portable, has language bindings for C, C++, and Fortran, and has been scaled to machines with over 100,000 cores. MPI has been used to develop applications in a variety of fields, from linear algebra solvers to molecular dynamics simulations and many others.

2.3 Molecular Dynamics

Molecular dynamics is the study of the motion, energy, and other thermodynamic properties of molecular systems. Due to the complexity of the forces involved and the size of the systems, it is intractable to model these by hand for any but the most trivial system. Molecular dynamics is widely used in biology, physics, materials science, and medical research. Example applications include simulating protein folding, the mechanics of graphene, and pharmacokinetic interactions.

2.3.1 Lennard-Jones

The Lennard-Jones potential is a model which represents the pairwise interaction of forces between two atoms. It is a relatively simple mathematical formula parametrized by the distance between two atoms. The simple nature of this model leads to inaccuracies in MD simulations by not accounting for embedding energy, as well as other approximations. Significantly, the Lennard-Jones potential does not account for the embedding energy of molecules. However, its simplicity is its greatest strength in the parallel environment, due to a reduction in communication and computation. In general, it is the canonical "first" model that is used when developing MD applications.

2.3.2 Embedded Atom Model

The Embedded Atom Model (EAM) improves in accuracy over the Lennard-Jones model by incorporating the embedding energy, but this comes with some noticeable drawbacks. The Embedded Atom Model is not as delightfully parallel as Lennard-Jones due to the extra communication and computation for the embedding energy. However, EAM is still a widely used model of atomic interactions in simple metals. It is more difficult to parallelize than Lennard-Jones, but doing so is worthwhile for the increased accuracy.

In the EAM, the total potential energy is written as a sum of a pairwise potentials, φ, and the embedding energy, F:

$$U = \sum_{ij} \varphi(r_{ij}) + \sum_{i} F(\bar{\rho}_i)$$

The pairwise potential, φ_{ij}, is a two-body inter-atomic potential, similar to the Lennard-Jones potential. $F(\bar{\rho})$ is interpreted as the energy required to embed an atom in an electron field with density $\bar{\rho}$. The local electron density at site i is calculated by summing the "effective electron density" due to all neighbors of atom i:

$$\bar{\rho}_i = \sum_{j} \rho_j(r_{ij})$$

The force, F_i on atom i, is given by:

$$\begin{aligned} F_i &= -\nabla_i \sum_{ij} U(r_{jk}) \\ &= -\sum_{j} (\varphi'(r_{ij}) + [F'(\bar{\rho}_i) + F'(\bar{\rho}_j)]\rho'(r_{ij}))\hat{r}_{ij} \end{aligned}$$

In this equation, primes indicate the derivative of a function with respect to its argument, and $\hat{\rho}_{ij}$ is a unit vector in the direction of atom j from atom i.

The form of this force expression has two significant consequences. First, unlike with a simple pairwise potential, it is not possible to compute the potential energy and the forces on the atoms in a single loop over the pairs. The terms involving $F'(\bar{\rho})$ cannot be calculated until $\bar{\rho}$ is known, but calculating $\bar{\rho}$ requires a loop over the atom pairs. Hence, the EAM force routine contains three loops.

1. Loop over all pairs, compute the two-body interaction and the electron density at each atom.
2. Loop over all atoms, compute the embedding energy and its derivative for each atom.
3. Loop over all pairs, compute the embedding energy contribution to the force and add to the two-body force.

The second loop over pairs doubles the data motion requirement relative to a simple pair potential.

For distributed memory machines, some of the molecules will be located on remote nodes. To obtain the needed density for remote molecules, a halo exchange is used after the second loop, which computes the embedding energy, in order to communicate $F'(\bar{\rho})$ for remote atoms. This provides the necessary data to complete the third loop, but at the cost of introducing a communication operation in the middle of the force routine.

In pseudocode, EAM can be summarized as follows:

```
Data: current atoms and their locations
Result: forces that should be applied to each atom
for phase in {pairwise, energy, apply} do
    for all local boxes do
        for neighbor box of current box do
            for all atoms in current box do
                for all local boxes do
                    calculate energy, potential, momentum;
                    apply cutoff for total energy
                    contribution;
                    apply Newton's second law;
                end
            end
        end
    end
end
```

Algorithm 1: Embedded Atom Model Pseudocode

Of the two types of potential calculation, the Embedded Atom Model is more accurate and more time consuming; therefore, we decide to concentrate on this specific function for the rest of the paper, with Lennard-Jones left as future work.

2.4 CoMD

CoMD is a classical molecular dynamics proxy application, developed by The Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx) at Los Alamos National Laboratory, for the purpose of testing the performance of new architectures and programing models. The publicly available code allows users to improve, modify, or extend it as they see fit. CoMD simulates the time evolution of atoms in metals and provides trajectory information for the molecular system. Fundamentally, the work is done by looping over pairs of atoms giving rise to $\mathcal{O}(n^2)$ evaluations of the interaction in each time step. CoMD implements a simple 3D geometric domain decomposition to divide the total problem space into subdomains which are then mapped to MPI ranks. Based on the command line parameters given at runtime, it will divide the atoms among each available rank. The default value for the maximum number of atoms per box is 64, which allows for ample padding of the box structure to allow for density fluctuations. CoMD then proceeds to calculate and apply potentials per box by looping over the neighbor boxes. As described in Section 2.3.2, this calculation involves communicating molecules from neighboring remote boxes.

Timer	# Calls	Avg/Call (s)	Total (s)	%Loop
total	1	623.0573	623.053	101.05
loop	1	616.5959	616.5959	100.00
time_step	10	61.6595	616.5949	100.00
position	100	0.0088	0.8820	0.14
velocity	200	0.0047	0.9407	0.15
redistribute	101	0.0604	6.0996	0.99
atom_Halo	101	0.0086	0.8671	0.14
force	101	6.0858	614.6637	99.69
eamHalo	101	0.0012	0.1212	0.02
commHalo	606	0.0001	0.0514	0.01
commReduce	39	0.0000	0.0002	0.00

Table 1: Profile of CoMD using MPI

3. DESIGN

After a full suite of profiling runs of CoMD, of which results are presented in Table 1, we conclude that the most computationally intense section of code is the calculation of the potential and kinetic forces between atoms. This section of the program dominates the total running time and is responsible for over 90% of the time of the program.

We strove to keep the original structure of CoMD as close to the original implementation as possible. This endeavor became a challenge as most of the data is stored in complex data structures, which are not ideal when transferring pertinent data to the GPU kernel for computation. Each box on a node became a block in the CUDA model and each atom in a box is assigned to a single CUDA thread. Therefore the grid and block dimension for the kernel became:

```
dim3 Dimgrid(numlocalBoxes,1,1);
dim3 Dimblock(MAXATOMS,1,1);  //MAXATOMS is per box
eamForce<<<Dimgrid, Dimblock>>>(...);
```

The three serial loop steps described on the previous two section were implemented as two separate kernels; combining the pairwise and embedding energy computation into a single kernel and the application of forces to all atoms into a second kernel. The need for two separate kernels arises from the fact that, as stated before, there is an MPI communication operation between all neighboring nodes is needed before the second kernel can be launched. This adds extra overhead as most of the data needed for the second kernel is the same as the first kernel but we must transfer the data twice, to and from device, because of the communication step. The basic version of the EAM kernel takes advantage of constant memory for the obvious set of constants, but uses CUDA atomic operations to keep data integrity amongst the different threads. This is needed due to the possibility of data races that arise when multiple CUDA threads try to write to the same location when computing pairwise forces.

3.1 Challenges Using MPI+CUDA

1. To be able to take advantage of constant memory optimizations, the CUDA kernel must know the size of the allocation at compile time. However, we do not know the size of the allocation until runtime since it depends on the manner in which the decomposition

of the MPI ranks divides the work, and that is determined by the number of processes in the job.This problem limits our constant and shared memory optimizations. Although CUDA supports dynamic shared memory allocation by passing the number of bytes required as the third argument of the kernel launch, the runtime only supports a single dynamically declared allocation per block. This limitation hinders our optimizations, but gives us a few possible solutions to this problem. We could use pointers to offsets within that single allocation; however, we must be aware when using pointers that shared memory uses 32-bit words, and all allocations must be 32-bit word aligned, irrespective of the type of the shared memory allocated. Another solution is to use a single large allocation and then dynamically swap data in and out as we need them inside the kernel.

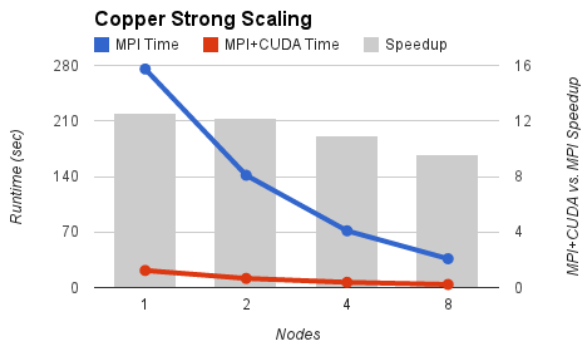
2. When using multiple processors per node and only a single GPU device we encounter contention among all processors trying to launch the CUDA kernel. The first processor will launch the kernel correctly, but all subsequent processor will fail to launch their CUDA kernel successfully. If multiple GPU devices are on the same node, we can develop a scheme where each processes in a node is assigned a different GPU device. Thus far, we have only run on systems with a single device per node; therefore, we are restricted to one process per node to avoid contention. Another solution would be to use GPU virtualization such as [3]. Nvidia's Hyper-Q also presents a potential solution to this problem for GPUs of the Kepler architecture and newer.

3. Communication between multiple GPUs requires copies to and from the host processor in order to communicate with MPI. This incurs extraneous time penalties when data is being shuttled across the PCIe bus. During this time, the programs have to block, since the remote data is necessary for the computation to continue. There is no reason why the data has to go back to the CPU; it is not modified at all in this process. Nvidia GPUDirect solves this problem for certain Infiniband machines by implementing RDMA between GPU devices. This solution relies on network, memory, and GPU hardware being aware of each other and having pathways to directly access each other, bypassing the CPU and its memory.

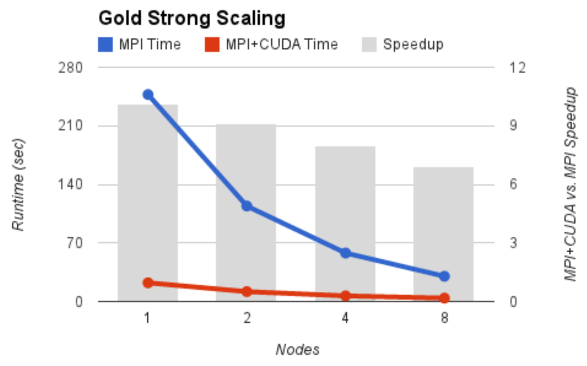
4. EXPERIMENTAL RESULTS

Performance results were collected on Blue Waters. Blue Waters is the fastest supercomputer on a college campus and has a peak performance of more than 13 quadrillion calculations per second. Blue Waters represents a balanced HPC system in which computational power is complemented by a fast network and storage system. Nodes on Blue Waters are divided into two different types, XE and XK. The XE6 dual-socket nodes are populated with two AMD Interlagos 6276 CPU processors (one per socket) with a nominal clock speed of 2.3 GHz and 64 GB of physical memory. XK7 accelerator nodes will are equipped with one Interlagos 6276 CPU processor and one NVIDIA GK110 "Kepler" accelerator K20X.

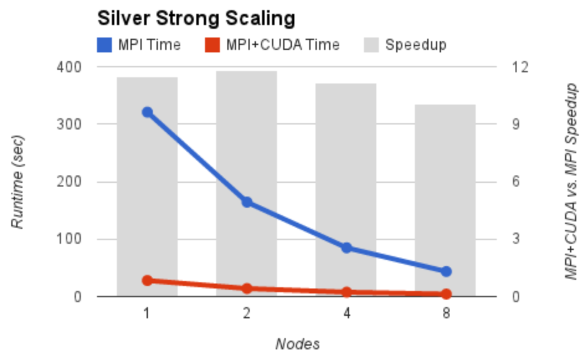
For our experiments we use only XK7 nodes. We select three source files from the NIST Material Measurement Laboratory's Interatomic Potentials Repository Project[3], `Cu01.eam.alloy`, `Au-Grochola-JCPOS.eam.alloy`, and `Ag.set`, to use in strong scaling experiments. We run simulations of 256,000 atoms for each input file and the results are shown in Figure 1. The x axis shows the number of nodes (with 1 MPI process per node), the left y axis shows the total runtime (wall clock time) measured in seconds, and the right y axis shows the speedup acheieved using MPI+CUDA over the original MPI code. The speedup we achieve from our MPI+CUDA implementation is related to the overall problem size and how many MPI ranks take part in the computation.



(a) Results for Copper



(b) Results for Gold



(c) Results for Silver

Figure 1: Scaling Results for Copper, Gold, and Silver simulation using MPI and MPI+CUDA

[3]`http://www.ctcms.nist.gov/potentials/`

Figure 1 shows runtime and speedup results for strong scaling from 1 to 8 nodes on Blue Waters. For copper, Figure 1a our MPI+CUDA implementation achieves between 12.5X to 9.6X speedup over the original MPI code. As described in Section 2.4, when increasing the number of ranks, CoMD evenly distributes the atoms. This effectively reduces the amount of work each rank can offload to the GPU; therefore, it makes sense that the effectiveness of our EAM kernel decreases when increasing the number of ranks and thus reducing the amount of work. This is evident in Figure 2 by the efficiency of our MPI+CUDA implementation decreasing faster than the pure MPI version as we increase the number of nodes. Additionally, the 2, 4, and 8 node runs all involve network communication which was not accelerated by our methods. Since we decreased the total computation time and the network communication time was constant, the impact of communication time is larger by percent of the total time. Due to these reasons, gold and silver also show a wide range of speedups, 10.1x to 6.9x and 11.8x to 10.1x respectively. For copper and silver, when one and two MPI ranks are used, our speedup largely remains the same, only mildly losing efficiency. Scaling up to four MPI ranks, we begin to see a much more noticeable decrease in efficiency that does not occur in the pure MPI version. This drop in efficiency results in the sizable drop in speedup. This is not the same for gold, as shown in Figure 2, the original MPI version benefits from some superlinear speedup when increasing from one to two ranks, likely due to caching effects. This speedup explains the reason we get less benefit from our MPI+CUDA version than can be seen in Figure 1b.

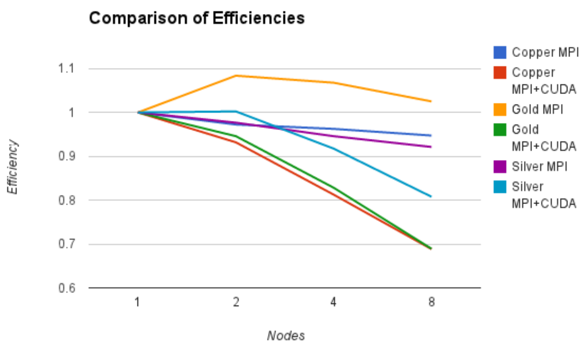


Figure 2: Efficiencies over all experiments

4.1 Further Optimizations

Next, we try to exploit some of the underlying details of how CUDA kernels execute for further optimizations. First, we utilize the fact that CUDA hardware handles memory accesses per half-warp. This fact allows us to remove the atomic operations needed for code correctness as long as we run the program with a fixed number of atoms per box to be less than a half-warp. Atomic operations are needed due to the problem of races among the threads when writing, but if we know only a half-warp number of threads are writing at the same time and each one of them is always writing to a different memory location, then we can remove the atomics without problems in the overall correctness of the program. However, we are not able to run with a half-warp or fewer number of atoms per box since our input files are too dense.

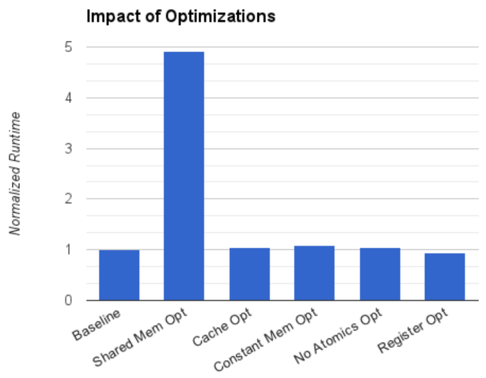


Figure 3: Impact of Various Optimizations

Another optimization is removing atomics memory operations from kernels, instead using thread local variables with reductions. To our surprise, we saw a small slowdown to the overall execution time when removing atomics. This was a counterintuitive result, as this contradicted the usual impact of removing atomics in CUDA. After further testing and research into the details of the Kepler architecture, we found that the throughput of atomic global memory operations have been extensively improved, and now an atomic operation takes about the same time as a regular load to global memory [1]. This only answers why we do not get any speedup, but does not effectively answered why we get a small slowdown. To answer this it requires some knowledge of how atomic operations work compared to regular global memory operations. When modifying a variable in global memory, we take its current value perform an operation and then overwrite that same variable. Thus, we require two memory operations to accomplish this. In contrast, an atomic operation only requires the address of the variable that needs to be overwritten, and it performs the same computation with only one memory operation. We attribute the small slowdown to the program to the improvements to global memory atomics and the extra memory operations paid when removing atomics. In our kernel, we do not have have synchronization inside our tight computation loop, so we allow the threads to diverge and run without blocking.

Section 3.1 describes some of the difficulties of using shared memory optimizations when using MPI+CUDA programs. Nevertheless, even if we did not know at compile time the amount of shared memory we needed to allocate, we can leverage shared memory by hard-coding the size of one of the data structures that is widely accessed in the kernel. Only one of the three possible large data structures could benefit from shared memory due to the large memory footprint of each data structure. This limitation of shared and constant memory is another issue that makes using accelerators on scientific HPC applications difficult. Current state of the art GPUs have 48KB of shared memory and 64KB of constant memory. HPC applications usually need large data structures that may require tens of kilobytes of space each. Having additional space for these sorts of large data table on the GPU would assist in porting large-scale scientific codes.

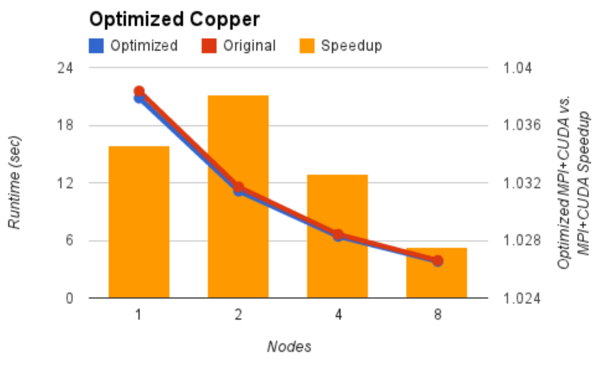
Another optimization option is using shared memory optimizations for common data arrays. In order to use these shared memory optimizations, we have to copy from global memory to shared memory in every thread block. When performing this copy, we take advantage of all of the threads in our kernel to do this in parallel. Results of CoMD using shared memory are shown in Figure 3. We see a substantial slowdown in performance due to a number of reasons. First, there is a large penalty in copying such a large amount of data, even when done in parallel. Second, the threads of our kernel were very light weight before using this additional shared memory, which requires every block of threads to store tens of kilobytes of data each. This could hamper context switches or prevent additional blocks from being scheduled due to memory size constraints, which would lower overall utilization of the SMs. Constant memory optimization suffer from similar slowdown as seen on Figure 3 but for slightly different reasons. Constant memory does not require explicit copy from within the kernel; instead, the problem arises due to the access pattern of CoMD. When every thread in a half-warp accesses a different value in constant memory, the accesses are serialized, which causes a slowdown relative to the parallel access in global memory. Additionally, there is a high degree of spatial and temporal locality in the memory accesses of CoMD, so the caches of the GPU are effective in speeding up global memory accesses.

Allocating additional memory to the cache is another possible optimization technique. On CUDA devices the L1 cache and shared memory use the same hardware resources. This can be set through the a call to `cudaDeviceSetCacheConfig` with the argument `cudaFuncCachePreferL1` to make the device prefer a larger L1 cache and smaller shared memory. We achieved no additional speedup when this was attempted due to the regular access pattern of CoMD. This regular access pattern means that the old data that a larger cache would allow us to store is of no utility since it will not be read again. Additionally, the working set of each thread was small enough to fit in the smaller cache, so increasing its size did not help. remains the same regardless of the increase of cahce. Results can be seen in Figure 3.

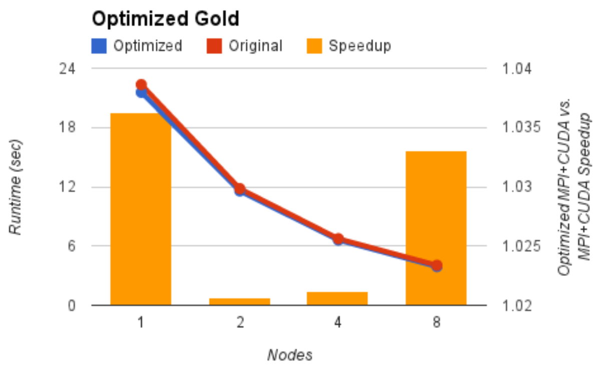
A small optimization that did gain some modest speedup was the use of registers instead of expensive global memory accesses. This type of optimization was very limited since it can only be applied to read only variables without incurring relatively costly writes to shared or global memory for other threads to read. Using this optimization for the applicable variables brought the runtime down to 94.5% of the baseline due to increased memory access speed for those variables.

We also contemplated leveraging concurrency in our application by using CUDA streams. Before starting what could be potentially a complex process of modifying our code to take advantage of streams, we decided to use profiling tools on our kernels to see whether it was a worthwhile endeavor. Table 2 shows the profile information of our two kernels, as given by the `nvprof` tool. The highlight of this table is the time it takes to copy data to and from the GPU. Communication is such a small amount of time compared to the total time taken by the two kernels that using streams to overlap computation with memory operation across the processor and GPU would only give us marginal speedup at best. Due to this, we decided against implementing this possible optimization.

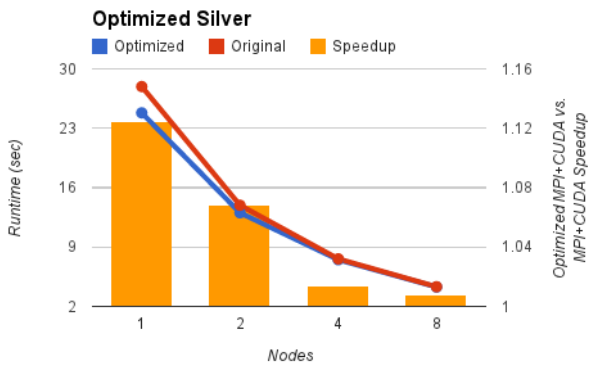
In conclusion, we implemented our kernels with the register allocated variable optimization and with atomic operations. Results of the optimized version are presented in Figure 4. Overall, we achieve upwards of 12% speedup compared to our baseline MPI+CUDA implementation.



(a) Results for Optimized Copper



(b) Results for Optimized Gold



(c) Results for Optimized Silver

Figure 4: Scaling Results for optimized Copper, Gold, and Silver simulation using MPI+CUDA

5. CONCLUSIONS

Time(%)	Time	Calls	Avg	Min	Max	Function Name
55.47	12.71s	101	125.86ms	116.48ms	134.45ms	eamForce(float*, [...])
43.93	10.07s	101	99.67ms	94.22ms	112.65ms	eamForce_2(float*, [...])
0.34	77.20ms	3030	25.48us	704ns	232.70us	[CUDA memcpy HtoD]
0.18	41.21ms	505	81.60us	2.08us	225.40us	[CUDA memcpy DtoH]
0.09	19.90ms	606	32.84us	1.02us	53.22us	[CUDA memset]

Table 2: `nvprof` Results

In this report, we highlight the importance of leveraging accelerator technology, such as GPUs, to speedup HPC applications. We show that in order to effectively do so, a mixture of MPI and CUDA code must be used. We highlight numerous difficulties when trying to write MPI+CUDA hybrid code that must be taken into account when writing such programs, and possible solutions to overcome them. We profile CoMD, a well known proxy mini-app developed by The Exascale Co-Design Center for Materials in Extreme Environments at Los Alamos National Laboratory, and speedup its computationally intense functions using GPU accelerators. We discuss the computation involved in doing EAM force calculations and the difficulties of parallelizing it using GPUs and, specifically, the MPI+CUDA paradigm. We also go over our design choices when offloading the hot functions. Finally, we run a 256,000 atom simulation of different metals on Blue Waters and compare our MPI+CUDA implementation against the original MPI code. Blue Waters is representative of the shift to accelerators occurring in the supercomputing domain for performance and efficiency reasons. We ran strong scaling experiments from one to eight processors and show speedups of over 12x. We performed a suite of common optimizations and some more complex optimizations that take advantage of underlying hardware characteristics of GPU hardware.

6. REFERENCES

[1] NVIDIA Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110.*

[2] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, February 2014.

[3] K. Sajjapongse, X. Wang, and M. Becchi. A preemption-based runtime to efficiently schedule multi-process applications on heterogeneous clusters with gpus. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 179–190, New York, NY, USA, 2013. ACM.

[4] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Addison-Wesley Professional, 1st edition, 2010.

[5] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, et al. Anton, a special-purpose machine for molecular dynamics simulation. *Communications of the ACM*, 51(7):91–97, 2008.

[6] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of computational chemistry*, 28(16):2618–2640, 2007.

[7] J. P. Walters, V. Balu, V. Chaudhary, D. Kofke, and A. Schultz. Accelerating molecular dynamics simulations with gpus.