Accelerating Geophysics Simulation using CUDA

Brandon Holt
University of Wisconsin–Eau Claire
105 Garfield Ave, Eau Claire, WI
holtbg@uwec.edu

Daniel Ernst
University of Wisconsin–Eau Claire
105 Garfield Ave, Eau Claire, WI
ernstdj@uwec.edu

ABSTRACT
CitcomS, a finite element code that models convection in the Earth's mantle, is used by many computational geophysicists to study the Earth's interior. In order to allow faster experiments and greater simulation capability, there is a push to increase the performance of the code to allow more computations to complete in the same amount of time. To accomplish this we leverage the massively parallel capabilities of graphics processors (GPUs), specifically those using NVIDIA's CUDA framework. We translated existing functions to run in parallel on the GPU, starting with the functions where the most computing time is spent. Running on NVIDIA Tesla GPUs, initial results show an average speedup of 1.8 that stays fairly constant with increasing problem sizes. Though many applications can see even orders of magnitude improvement from GPGPU acceleration, the potential improvement for CitcomS is limited by several factors, including being bound by MPI collective communication. With newer GPGPU frameworks such as Fermi, further performance improvements can be expected, though a more significant overhaul of the CitcomS code would be necessary for any significantly better speedup.

General Terms
Parallel programming, GPGPU, finite element

Keywords
CitcomS, Blue Waters Undergraduate Petascale Internship, CUDA

1. INTRODUCTION
Graphics processors are commodity hardware, found in nearly every modern personal computer, which are highly specialized to do all of the computations required to draw pixels on the screen. For intensive graphics applications such as high-resolution three-dimensional games, this means transforming, calculating lighting, and mapping and applying textures on millions of vertices and pixels. To do this, GPUs have banks of hundreds of small compute cores that process a stream of data together in parallel. By sacrificing some of their independence and flexibility, these cores save hugely on power consumption compared to more general-purpose CPUs. In addition, in order to keep all of those cores busy, GPUs have particularly high memory bandwidth.[8]

The high performance computing community has long been interested in using specialized *accelerators* to speed up certain parts of their computations, but these processors were often prohibitively expensive. In contrast, because of their commodity nature, GPUs are much cheaper. Instead of simply drawing graphics on the screen, we are interested in using these cards for more general purpose work. By using the massively parallel capabilities of GPUs for computation across large datasets, applications can see huge performance improvements. NVIDIA's CUDA parallel computing architecture is currently the most popular technology for general purpose programming of GPUs. Many high performance applications have seen order of magnitude speedups using CUDA, including NAMD (nanoscale molecular dynamics) with a 20x speedup, and Havok FX with 10x speedup for realtime physics simulation.[10]

In the field of geophysics, CitcomS is used by many to study convection problems in Earth's mantle. By accelerating the application we can enable simulations with finer detail or more time steps to complete in the same amount of time. At this time when single cores are not getting any faster, speedups come from parallelizing computation. CitcomS already uses Message Passing Interface (MPI) libraries to split up work across multiple compute nodes. However, the price of communicating boundary values limits the amount that the simulation can be split up in this way. To further parallelize the work done on each node, we use the massively parallel capabilities of graphics processors (GPUs). Using NVIDIA's CUDA programming model, we are able to parallelize each node's data-parallel computations using CUDA-capable GPUs if they are available. This report discusses the techniques used to accelerate CitcomS, describes

performance challenges and optimizations, and lists the results of our experiments showing speedup of the application as a whole.

2. BACKGROUND

2.1 CitcomS

CitcomS is a finite element code developed and supported by the Computational Infrastructure for Geodynamics (CIG). Written in C, it has support for MPI to allow it to be run in parallel on shared and distributed memory platforms. It is designed to solve compressible thermochemical convection problems, but it also support variable viscosity and so can be used to study the movement of plates. The model consists of a grid of points arranged in a spherical shell (either a full sphere or a restricted region). Starting with a set of initial values including temperature, pressure, and velocity at each point, it repeatedly solves the momentum and advection-diffusion equations, giving the new state of the system at each successive time step. An iterative relaxation scheme is used to solve the partial differential equations for velocity and pressure across the grid domain, using either a conjugate gradient or multi-grid solver to find solutions for the equations which are represented as discrete matrices.[12]

As a typical finite element code, with its regular grid and high spatial locality, CitcomS is characterized as a *structured grid* problem as defined by Asanovic and colleagues in their report. Because CitcomS does not do adaptive mesh refinement, this means that it should be relatively simple to parallelize by simply breaking up the grid onto different nodes and sharing updated boundary values between iterations.[3] In fact, this is precisely what is done using MPI already, and because each point only relies on its immediate neighbors in the grid, calculations on chunks distributed to each node should be able to be parallelized even further.

2.2 CUDA Programming Model

While CUDA aims to make GPUs programmable like CPUs, the mindset is quite different. Instead of one primary thread (or a minimal number of software threads) on CPUs, the idea behind the CUDA programming model is to have hundreds or thousands of hardware threads running the same *kernel* function on different pieces of a large data set. In general, each GPU thread individually runs slower than on the CPU, but together, hundreds of threads have a much higher throughput. Because of the less flexible nature of GPU cores, these threads also have extremely limited branching options: no function calls are allowed (except inline) within a kernel.

In the CUDA model, there is a reference hierarchy to threads with different localities: a *block* of threads all run on the same *streaming multiprocessor (SM)* which has a number of compute cores, a bank of registers local to each thread, and a fast *shared memory* cache. Thread blocks are organized in a *grid*, all running the same kernel and sharing the *global memory* bank on the GPU. Only threads within a block can be synchronized using primitive barrier calls. Because global synchronization is impossible within a kernel, the algorithm must have work that can be completely de-coupled.

Porting existing CPU code to CUDA primarily involves finding which parts do a lot of computation on a large set of data in a serial loop and splitting up that work onto many threads. However, to get optimum performance, the programmer must know the intricate details of the hardware and how they affect performance. Optimization practices include: managing the severely limited shared memory cache, coalescing global memory accesses, minimizing in-warp divergence, avoiding bank conflicts, and maximizing thread occupancy.[9]

Table 1: Profiling Results
(Conjugate Gradient Solver)

% Time	Function
78.82	`n_assemble_del2_u`
4.70	`conj_grad`
4.06	`global_vdot`
3.49	`assemble_div_u`
2.82	`get_elt_k`
2.12	`assemble_grad_p`
1.17	`regional_exchange_id_d`
...	...

3. IMPLEMENTATION

We used the current release of CitcomS, version 3.1.1, as the basis of our CUDA accelerated version.

3.1 Profiling

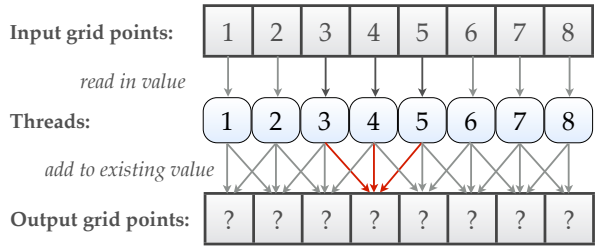
Translating the entire application to CUDA would be neither practical nor desirable. CitcomS consists of a large codebase, with many different functions for calculating each parameter of the simulation. Many of these operations, particularly input and output tasks, are simply not conducive to massively parallel execution. Therefore, in order to maximize our impact on the overall speed of the code, we carefully profiled it to see where in the code most of the time was being spent.

Using GNU gprof[4], we ran a series of simulations with different grid sizes and input parameters. Shown in Table 1 are results for a typical run using the conjugate gradient solver. Actual numbers for each function fluctuated slightly between different simulations, but the trend was consistent. It was clear that the function `n_assemble_del2_u`, at 78% of the overall time, was where we should focus our efforts to improve the conjugate gradient solver. According to Amdahl's Law, the theoretical speedup of an algorithm from parallelization is a function of the speedup of the parallelized part (S) and the proportion of the computation that this part represents (P):

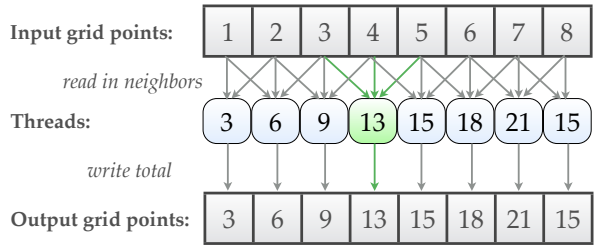
$$\frac{1}{(1-P)+\frac{P}{S}}$$

Speedup here simply refers to the ratio of serial execution time to the parallelized execution time ($S = T_{serial}/T_{parallel}$). Speeding up only `n_assemble_del2_u` with $P = 79\%$, assuming perfect speedup ($S = \infty$), the maximum theoretical speedup overall would be 4.72.

Further inspection of the code showed that this function is used to calculate the Laplacian on the discretized temperature, pressure, and velocity matrices, which correspond to the size of the spherical grid. As we expected for this structured grid code, this matrix computation corresponds



(a) In the original algorithm, each point added its own contribution to each of its neighbors' new values. Concurrently writing to the same spot in memory from multiple threads like this leads to nondeterministic behavior because the reads and writes could happen in any order.



(b) To eliminate conflicting read/writes, the algorithm was restructured so that each thread calculates its own new value by computing what each of its neighbors would have contributed based on their current values and storing just its own value.

Figure 1: Reordering Reads and Writes

to an operation being executed across all points in the spherical grid, marking this as a good candidate for GPU execution and making this our primary target for translation to CUDA.

3.2 Translation

The first part of translating `n_assemble_del2_u` was deciding how to break up the work onto GPU threads. For CUDA, using as many threads as possible is often the best option. Each streaming multiprocessor can execute several blocks of threads. If a large number of threads in the same block make memory requests, they can queue up waiting for their data while other threads are executed. Also, if these threads access memory local relative to each other, their accesses can be coalesced into a single global load. Therefore, the more threads the CUDA scheduler has available in each thread block, the more opportunities it has to hide memory latencies. On the other hand, once each thread gets its memory, it ought to have a significant amount of work to do with it, otherwise all the time used getting the memory there was wasted.

Taking those factors into consideration, we decided to make each thread handle a single grid point. This is justified because the calculation of each point requires aggregating values from several matrices for each of its neighbors. In the original CPU code for `n_assemble_del2_u`, there was a primary loop that iterated over each point in the grid, so we simply refactored it into a CUDA kernel, using the thread and block IDs in place of the loop variable.

However, running iterations of a loop concurrently introduces problems that did not exist when run serially. Threads running on the GPU have extremely limited synchronization options. CUDA cards with compute capability of at least 2.0 introduced some atomic store operations but at a significant performance hit from serialization. Without synchronizing, it is unsafe for concurrent threads to write to the same spot in memory, so kernel functions should be designed such that each thread will have "ownership" of a subset of memory that only it stores results into. In the case of our kernel, each thread was straightforwardly responsible for calculating its own grid point's future state. However, each iteration of the original loop had one loop over its neighbors where it incremented each neighbor's values based on its own value. This would have led to multiple threads concurrently incrementing the same spots in memory, resulting in nondeterministic behavior. Fortunately it proved possible to reverse the second loop so that each thread pulled together all the values that were previously being incremented, resulting in a working, deterministic CUDA translation, calculation for calculation. See Figure 1 for an illustration of this.

3.3 Verification

Because our goal is a functionally equivalent refactoring, it was essential that we verified our function's output with the original at every step of development. This was accomplished by simply executing both versions each time `n_assemble_del2_u` was called. A single array constitutes the output for the function, so both original and CUDA output were stored, with the following condition used to flag an error:

$$\frac{|x_i - y_i|}{y_i} > 5.0 \times 10^{-15}$$

where y_i is an element in the original function result and x_i is the corresponding CUDA result. Even though it is expected that all of the same calculations will be done, some small floating point precision error is expected due to some the reordering of calculations and minor differences in how GPU cores implement floating point operations, but those errors should be relatively insignificant.

3.4 Optimization

As mentioned before, the CUDA architecture can provide massive amounts of parallel computation with particularly high memory bandwidth. However, achieving that maximum performance is not always possible, and approaching that limit is what the majority of the novel work for this project was devoted to. Here we will step through several major iterations of the CUDA kernel, highlighting the most relevant performance aspects of each.

The CUDA profiler is a very simple tool which comes bundled with the CUDA framework, but it was particularly useful for diagnosing each of the bottlenecks described below. The profiler is very simple and supports the output of only four variables from a list of over 40 options. On systems with CUDA installed, profiling can be enabled by setting an environment flag (`CUDA_PROFILE=1`) before executing the CUDA program. Values of each variable for every kernel invocation is sent to a file (`./cuda_profile0.log` by default).[7] Below we will highlight profiling options we used to obtain information we needed about our kernel's performance.

Table 2: Kernel Optimizations
Speedup shown is for the ratio of the original CPU function's execution time to the time spent executing the `n_assemble_del2_u` *kernel itself.*

	Time (ms)	Speedup
Original:	2.00	1.00
Kernel 1:	880	0.0023
Kernel 2:	21.78	0.09
Kernel 3:	0.44	4.55
Kernel 4:	0.29	6.90

Kernel 1: The first working implementation of the kernel was 400 times slower than the original CPU version. On the CPU there are several layers of caches that make sure that successive reads and writes to a chunk of memory are as fast as possible, which are automatically handled in hardware on most systems. Because access patterns are potentially much more complicated to predict for thousands of threads, automatic caching is only available for the newest CUDA architectures. Therefore, in this first kernel, when each thread accesses its nearest neighbors' values, it must go all the way to global GPU memory, an operation that takes on the order of milliseconds. The GPU memory manager can *coalesce* memory accesses, that is load multiple threads' values all together, if the accesses are sequential and regular. However, because each thread is accessing each of its neighbors' values, which are not contiguous, these accesses are not able to be coalesced, even on newer architectures where caching is handled automatically. The CUDA Profiler options `gld_incoherent`, `gld_coherent`, `gst_incoherent` and `gst_coherent` show the number of uncoalesced (incoherent) loads and stores compared to coalesced ones. Ideally, there should be no incoherent loads or stores, which was achieved in subsequent kernels.

Kernel 2: Luckily there are closer and faster memory banks available on each streaming multiprocessor that can be shared among threads in a block. Moving the most heavily accessed data in our kernel into this *shared memory* can easily be done in a coalesced fashion, and once in this cache, accesses to the data are comparable to working with registers. In our kernel, the most heavily used array, which we determined by counting the number of reads from it in our code, is `Node_map`, which holds a map of each grid point to indices for all of its neighboring points. By explicitly loading all of the maps for each thread in a block into shared memory at the beginning of the kernel, the global accesses are regular and sequential, allowing the loads to be coalesced, and all subsequent accesses are essentially free. We were able to observe this on the CUDA cards with compute capability of 1.3 by observing a significant drop in the number of global loads (`gld_incoherent` and `gld_coherent`) from millions of loads to 40,000-80,000 per invocation (varying depending on the amount of branching, actually). Note: newer cards with compute capability greater than 2.0 have the option of outputting shared memory loads and stores directly.[7] Just doing this sped up the kernel by 40 times. Clearly even minor changes to CUDA kernels can either severely damage performance or greatly improve it.

Kernel 3: Using an array of indices to map into another array is a common method of saving computation time on the CPU because it allows for all the neighbors to be calculated once and then simply looked up every subsequent time they are needed. This made sense on the CPU where there was a deep cache and only one compute unit. On the GPU, however, it often makes sense to recompute some values if it would take less time than waiting on extremely slow un-cached memory accesses. Once an (x, y, z) location in the grid is calculated from the thread index, three levels of nested loops can iterate over all the neighbors by simply oscillating on each side of each dimension. This was observed again as a drastic decrease in the number of global loads in our profiling results. Eliminating `Node_map` in this way improved the performance of the kernel nearly another 50 times.

Kernel 4: With `Node_map` no longer taking up space in shared memory, other data was able to take its place. However, the Tesla generation of CUDA cards have only 16 kilobytes of shared memory per multiprocessor.[8] Even just the primary input array will not fit completely in shared memory for typical grid sizes. Because we have such limited synchronization ability, we must ensure that all values of the array that might need to be accessed by a thread in a block are pre-loaded into shared memory. By loading in a 3-dimensional tile of points surrounding the points referenced by the current block of threads, we are able to maximize the amount of caching we can do.

Working off of the final kernel design, there were a number of minor changes that we tried to maximize our usage of all of the GPU's resources. Each multiprocessor can have up to eight resident blocks, which allows it to hide memory latencies by scheduling warps from other blocks to run while some are waiting. This *occupancy* is determined by the amount of registers allocated per thread, the number of threads, and the amount of shared memory that each block uses. Using the verbose `ptxas` compiler option, the programmer is able to see these parameters, and the CUDA profiler displays the precise occupancy of each kernel invocation as well. Using the CUDA Occupancy Calculator, a spreadsheet available from NVIDIA's website, developers can put in the various parameters for their run, such as shared memory and register usage and it will show what the limiting factor for occupancy is.[6] Using this and the output of the CUDA profiler, it was clear to us that the major limiting factor was the number of registers being used by each thread, which we were unable to minimize. Instead, we adjusted the number of threads per block, observing the performance of each kernel and settling on the best configuration. We found that 192 threads per block best balanced the tradeoff between using too many registers and losing shared memory benefits by having thread blocks that are too small. However, these results would not be optimal for other CUDA architectures. It would have to be retested and optimized for each architecture to achieve maximum performance.

At this point, even with that primary array taking up all available space in shared memory, there are still several more global arrays that are accessed regularly during the computation. Several large floating point arrays that represent the stiffness matrix are much too large to for us to cache. As a result, our threads are constantly waiting on slow global memory loads instead of making full use of the GPU com-

pute units. There are a number of other performance issues that affect kernels at the instruction level, such as branch divergence when threads executing in lock-step on an SM take different paths and must be serialized, and bank conflicts when shared memory accesses are strided incorrectly. However, examining and eliminating these kinds of issues is only useful when the majority of the time is already being spent executing instructions. Because we are bound by memory accesses, not by compute resources, there is little more that can be done to further optimize the kernel.[5]

3.5 Automatic Integration

Not all users will have NVIDIA GPUs available, so we provide users of CitcomS with the option to use the CUDA-accelerated version or not. `Autoconf` and `automake` are two tools already being used in the code to automatically find libraries and configure the build. A simple addition to the configure script adds an option to build with CUDA support. However, we also did not want to require that everyone using CitcomS on a particular cluster would be required to use the CUDA version, nor did we want to force two versions to be built for such a small difference in code. When built with CUDA support, our `use_cuda` option is added to the configuration file that is used to specify each simulation, which can be used to select the CUDA-accelerated version at runtime with little to no overhead.

4. RESULTS

In the section on Optimization and in Table 2, our speedups only referred to the computation of the `n_assemble_del2_u` function. To see how these results translated into overall system acceleration, we ran a series of simulations and measured their runtimes. Each simulation was run twice per trial, once using the original CPU-only version and once with the CUDA version. Shown in Figure 2 are average times for three runs of a typical simulation with varying grid sizes and number of MPI processes, run using Tesla-generation cards that were available on Earlham College's Al-Salam cluster[1] and NCSA's Lincoln cluster[2]. Our timings for each setup had standard deviations of less than 5% of the average times after 3 trials, with most times varying less than 2.0 seconds between trials. Because of this consistency, the three trials should be accurate enough to make judgements about. Similar results were observed for several other simulations using different physical parameters (varied initial conditions and viscosity parameters), which was expected because this function is an integral part of both solvers, so its usage pattern is highly regular.

Seen in Figure 2a, as the problem size was increased, better speedups were observed. This is because as we increase the number of grid points, we were able to better take advantage of the massive parallelism provided by the GPU. However, the speedup tapered off around 1.8x, which we hypothesize is because the amount of memory that needs to be transferred to and from the GPU increases as well. From Figure 2b, it can be seen that the speedups we observe in the one-node case do not scale perfectly with increasing number of MPI processes. However, larger problem sizes still showed greater speedups. This is to be expected because for the same problem size, increasing the number of MPI processes gives each CUDA kernel smaller pieces of the grid to work with, which was just shown in Figure 2a to make speedup worse. Therefore overall, the best performance is still to be had with larger grids because it will give the best CUDA performance and allow more divisions into MPI processes.

Based on the actual speedup of `n_assemble_del2_u`, we can revisit Amdahl's Law, this time with $S = 6.9$ and $P = 78.82\%$ still. This results in a predicted overall speedup of 3.07. This is obviously less than the theoretical best we computed of 4.72 which is to be expected because perfect speedup is obviously impossible. However, it is significantly better than the 1.8x we actually achieved. This can be partly explained by the time needed to allocate and copy all of the memory onto the GPU, which was not included in the kernel timings because the memory copies are done in several different places in the code. Despite the high bandwidth to the GPU, the entire set of data used within the function must be transferred both ways every time it is called.

5. FUTURE WORK

As the CUDA-accelerated version of CitcomS stands right now, one function, `n_assemble_del2_u`, has been successfully translated and optimized for NVIDIA Tesla GPUs.

5.1 Multi-grid Solver

As previously stated, `n_assemble_del2_u` played the greatest role in the conjugate gradient solver. While it also was used in the multi-grid solver, another function, `gauss_seidel`, was the most time-intensive function there. We spent some time attempting to translate this function as well so that both solvers could be significantly accelerated. However, it proved significantly more difficult than the previous translation. For `n_assemble_del2_u`, all of the calculations simply read from the original array and stored their new values in the output array. The Gauss-Seidel method is an iterative relaxation method for solving a linear set of equations. It is based on the simpler Jacobi method which averages its nearest neighbors's current values to compute its own new value. In order to converge in fewer iterations, Gauss-Seidel makes use of any new values that are available in calculating each new value.[11] This is perfectly acceptable in a serially executed loop, but when loop iterations are run concurrently, this causes problems. This meant that we could not do a functionally equivalent translation of `gauss_seidel`. We made several attempts to write a Jacobi function that would operate in the same way as `gauss_seidel` but simply required more iterations to converge. However, everything we tried caused issues when run as part of the multi-grid solver, causing the solutions to never converge or to end up at infinity. At this time, the code has been left out of the production version of our CUDA-accelerated CitcomS. Perhaps with more experience with numerical methods we would be able to find the places where our Jacobi method failed to duplicate what the Gauss-Seidel function was doing. It might be that future work could be done to re-implement the multi-grid solver from the ground up in CUDA.

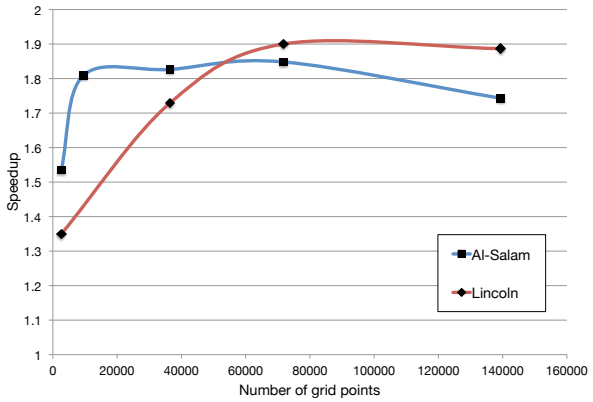
5.2 MPI Communication Barriers

Beyond the Gauss-Seidel/Jacobi issues, the most obvious course for future work would be to continue translating more of CitcomS's code to CUDA. However, there is a limit to the amount that patchwork translation such as this can be used to improve overall performance. The existing MPI commu-

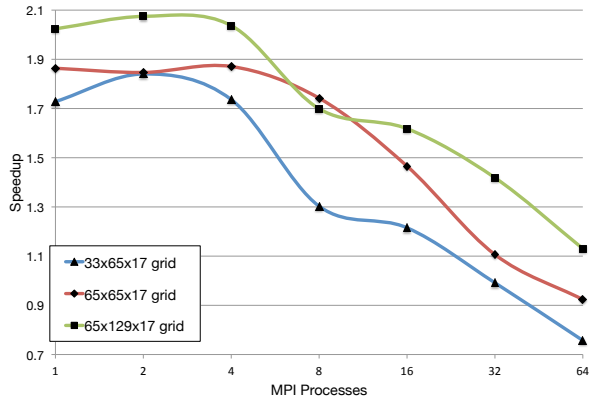
Figure 2: Speedup Results

Timing results used for speedup calculations are each averages of at least 3 separate runs.

(a) Scaling Problem Size



(b) Scaling MPI Processes



Grid Size	Orig. (s)	CUDA (s)	Speedup
2601	26.79	19.856	1.3492
36465	745.68	431.23	1.7292
71825	1609.46	847.04	1.9001
139425	5116.58	2712.16	1.8865

MPI	33x65x17	65x65x17	65x129x17
1	1.728	1.864	2.024
2	1.840	1.846	2.075
4	1.736	1.871	2.037
8	1.302	1.742	1.699
16	1.215	1.466	1.618
32	0.992	1.106	1.418
64	0.757	0.924	1.130

nication pattern in CitcomS frequently does collective operations sharing updated boundary values among all of the processes to keep the individual pieces of the grid synchronized. These MPI calls are an absolute barrier to what can be computed uninterrupted on the GPU. Prior to communication, the kernel must complete and the values must be copied back to main memory from the GPU's memory. Once communication is complete, the data can be copied back to the GPU and the kernel resumed. With these hard barriers in place, even if all data-parallel computations were done by the GPU, the limiting factor would be all the memory movement. A more complete rewrite of the CitcomS codebase might be able to minimize the amount of communication needed and perhaps coalesce it into a single update per step. However, with current cluster architectures where all inter-node communication goes through CPU nodes, the communication will likely still be a major limiting factor.

5.3 Other Directions

Future work on accelerating CitcomS could go several directions. If a version that will work across different vendors is desired, an OpenCL version of the current translation should be straightforward because the basic kernel model remains largely unchanged between the two. Along the same vein, the current version has certain parameters, such as the number of threads per block, hand-optimized and hard-coded into the source. Running optimally on even the newer Fermi generation of GPUs, which have larger shared memory caches among many other improvements, would require adjusting these parameters based on experimental results. For other GPU architectures similar adjustments would need to be made. Perhaps some future work could be done to automate this optimization task, for CitcomS or for CUDA/GPGPU applications in a more general sense. Because GPU acceleration is a highly popular area of research right now, it is likely that many new tools will soon be available to potentially be applied to CitcomS.

6. REFLECTIONS

This project was not only a scientific venture attempting to enhance geophysicists' tools. As part of the Undergraduate Petascale Internship Program, the goal is also to enhance undergraduate education, so here I will reflect on my own experience and how it can be duplicated for other undergraduates.

Working on this project has given me a great deal of experience working on a number of different high performance clusters. The two-week workshop at the beginning of the summer kickstarted my work, but through the process of studying, translating, and optimizing, I have gotten much more comfortable with it all. I have become adept at debugging all kinds of issues with building, installing, and running all manners of programs. In working with CUDA for the last year, I have come to understand the architecture in great depth, and I have a feeling for a wide variety of programming problems and performance issues. Because of my experience with CUDA and connections through the UPEP instructors, I have gotten the opportunity to help as an assistant instructor at an intermediate parallel programming workshop, as well as at this year's Blue Waters UPEP Institute. These teaching experiences have further reinforced my understanding of all of the parallel programming techniques that are taught there. I think getting undergraduates to

spend a summer digging down into real scientific code and get their hands dirty writing real high performance code will give them experience that will be invaluable to them in pursuing a research career later.

While working at the extremely low-level of CUDA optimization has helped me understand the architecture and programming model quite well, I have recognized that the amount of time it took me to get better at this is simply not efficient for the majority of programmers to do. In the coming age of computing, it is likely that, in order to continue to scale in performance without excessive power expenditure, computing will become increasingly heterogeneous, with specialized processors such as GPUs playing an important role. It is already infeasible to expect every programmer to become an expert in all of the different kinds of accelerator hardware that are available now. My work hand-tooling CUDA code and optimizing it by experimentation and often guesswork has made it obvious to me that new programming models need to be explored.

I am interested in finding out how to separate the mechanical processes, such as finding the optimal number of threads per block or managing limited shared memory space, from the creative ones, such as thinking of the best way to parallelize the algorithm. The mechanical tasks could be accomplished in many different ways, either by the compiler at build time, by a runtime system, or one of many other techniques. Another interesting area of research would be to better facilitate the creative part so that programmers can effectively represent their ideas in a way that compilers and the rest can take advantage of the available compute resources.

In the interest of pursuing these research directions, I will be attending graduate school at the University of Washington. My planned research goal stems from these issues with programming in CUDA: I am interested in applying aspects of programming languages, compilers, runtime systems, and software engineering tools to assist programmers in developing applications for the heterogeneous parallel computers, smart phones, or other ubiquitous computing devices that will need software in the future.

7. ACKNOWLEDGEMENTS

This research was conducted through the Blue Waters Undergraduate Petascale Education Program (BW-UPEP). In an effort to broaden and diversify the high performance community, this program funds a number of undergraduate students to do computational science research with a faculty advisor for a summer, as well as a stipend to continue their research during the school year. To prepare these students for their projects, they attend a two week workshop on parallel programming, scientific computing, and high performance architectures. Thanks to Shodor and the National Computational Science Institute (NCSI) for their financial support of this internship.

This research was supported in part by the National Science Foundation through TeraGrid resources provided by the National Center for Supercomputing Applications (NCSA) under grant number TG-CCR100014.

Thanks also to Charlie Peck from the Computer Science Department at Earlham College for the use of their Al-Salam cluster. Thanks also to the University of Wisconsin–Eau Claire Chemistry Department and Professor Christine Morales for use of their cluster, EB-Wilson.

8. REFERENCES

[1] Earlham College Cluster Computing Group. `http://cluster.earlham.edu/`.

[2] NCSA Scientific Computing: Intel 64 Tesla Linux Cluster Lincoln. `http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64TeslaCluster/`.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[4] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17:120–126, June 1982.

[5] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors*. Morgan Kaufman Publishers, 2010.

[6] NVIDIA. Cuda occupancy calculator. `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`.

[7] NVIDIA. *CUDA Profiler README, Version 3.0*. NVIDIA, 2009.

[8] NVIDIA. Nvidia's next generation cuda compute architecture: Fermi, white paper. Technical report, NVIDIA Corporation, 2009.

[9] NVIDIA. *CUDA C Programming Guide, Version 4.0*. NVIDIA, March 2011.

[10] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[11] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1988.

[12] E. Tan, M. Gurnis, L. Armendariz, L. Strand, and S. Kientz. *CitcomS: User Manual*. Computational Infrastructure for Geodynamics (CIG), 3.1.1 edition, July 2009. `http://geodynamics.org/cig/software/citcoms`.