Accelerating a Simulation of Type-I X-ray Bursts from Accreting Neutron Stars

Mark Mackey
Professor Alexander Heger

The goal of my project was to develop an optimized linear system solver to shorten the runtime of a computer simulation. The simulation was created to study the thermonuclear burning of Type I X-ray Bursts in low-mass X-ray binary star systems (LMXBs)[1]. To study these systems, thousands of X-ray bursts need to be simulated. Currently, the simulation can complete around one X-ray burst per day. I worked on accelerating the main bottleneck of the simulation by using very powerful graphics cards (GPUs) instead of the central processing unit (CPU) to perform the necessary calculations. I made steady progress over the summer towards this goal and eventually developed a solution that was four to five times faster than the original code.

The computer code that modeled the X-ray bursts is named KEPLER[2]. It models the nuclear burning using an extended nuclear reaction network. This requires solving very large systems of linear equations resulting from the linearization and discretization of a very stiff coupled partial differential equation system. The problem can be written as a matrix equation of the form

\[ Ax = b, \]

where A is a matrix of coefficients, b is a known vector, and x is the unknown solution vector that must be found to satisfy the equality. Due to the nature of nuclear reaction networks, the systems generated by the simulation are sparse; a majority of the elements in the matrix A are zero. Ideally, the algorithm chosen to compute the solution would be designed to ignore the zero elements to save valuable computing time. Figure 1 illustrates a typical matrix generated by the simulation. Currently, the simulation uses a package called MA28[3] to solve these systems using a highly optimized implementation of a sparse algorithm. My plan was to replace this package with one that

![Figure 1: A 1700x1700 matrix with 20505 non-zero elements distributed along the diagonal](image)
used both of the server's NVIDIA Tesla C2050 GPUs to compute the solution faster.

The first decision I had to make was what programming language to use. There are currently two languages that allow programmers to write code that runs on these graphics cards: OpenCL, and NVIDIA CUDA[4]. Initially I had decided to use OpenCL[5] because it would allow the code to run on many different types of GPUs unlike CUDA which only runs on NVIDIA cards. I eventually decided to switch to CUDA after it became clear that it had a much larger support base.

The second decision I had to make was what algorithm to use. I needed to find algorithms that were optimized for sparse systems so that the solution could be computed as fast as possible. There are several algorithms designed to do this, but many of them were ruled out because they required the matrices to be symmetric. I was left with two options: the biconjugate gradient stabilized method (BiCGSTAB)[6], and the generalized minimum residual method (GMRES)[7]. I found a library called “cusp” containing implementations of both these algorithms and tested them on some of the matrices generated by the simulation. The results were far from satisfactory as the runtime of the algorithms were several orders of magnitude larger than they needed to be.

I decided to meet with University of Minnesota professor Yousef Saad to ask for advice. He had co-developed the GMRES algorithm I tested earlier and his paper entitled “GPU-accelerated Preconditioned Iterative Linear Solvers” was the most useful publication I had found while researching algorithms for this project. His advice was to try implementing a non-sparse algorithm and see what performance I could achieve since GPUs are much better with dense computations and the size of the matrices were too small to benefit fully from the GMRES or BiCGSTAB algorithms.

I took Professor Saad’s advice and began developing a solver based on a parallel implementation of Gaussian elimination. I developed three different designs before I achieved one that would correctly solve all of the systems I tested it with. My fourth design improved the runtime by 20% by taking special care to align memory. This implementation was very fast, taking only 2 milliseconds to solve a 1024x1024 sized system (for comparison the same algorithm running on a
single core took 4.4 seconds). Figure II shows the runtime of this implementation vs the order of the matrix. Unfortunately it was still 8 times slower than the MA28 package that uses a sparse algorithm running on the CPU.

My next step was to optimize the algorithm to try and cut down the runtime. My best chance to do this was to optimize the program’s global memory access patterns. I spent time researching how to do this and discovered the CUDA profiler. The profiler is used to collect statistics about CUDA applications and identify possible bottlenecks. I experimented with the profiler and tried to optimize the memory access pattern of a simple program. Unfortunately the profiler continuously reported that the global memory access pattern was inefficient even though I had followed all the rules to make it perfectly efficient. I contacted a CUDA developer at NVIDIA about my problem and we eventually learned that I had found a bug in the profiler. I continued to develop the latest implementation of my algorithm while I waited for the bug to be fixed. It was at this point that I ran out of funding for my UROP.

Over the next few months I continued to explore other options in my spare time. I was looking into a package called CUDA_ITSOL[8] developed by Professor Saad. It is written in CUDA and contains several
iterative solvers for sparse systems. All I had to do was write an interface and test it. The results showed that the solver based on the GMRES algorithm was on average four to five times faster than MA28 on the same sized matrix. Figure III on the previous page shows a plot of the runtime of this package. From this plot it is clear that the time taken to copy the matrix and result vector to and from the GPU is greater than the time taken to solve the system. One possible way to decrease this time is to modify to code to use page-locked host memory. This increases the bandwidth between host memory and device (GPU) memory and should bring the copy time down.

Though I was successful in developing an algorithm that was faster than the existing code, there is still a need for further performance increases. A factor of four is a good improvement but to make significant progress the simulation needs to be around two orders of magnitude faster. There are still many options that I have not yet pursued to help reach this goal. The code can be further optimized to use page-locked host memory for faster data transfers. This would also allow data transfers to be performed concurrently with kernel execution so that the elements of the next matrix could be transferred to the GPU while the previous system is being solved. The code can also be extended to use both of the systems GPUs instead of just one.

Before this experience I had never attempted to solve a real world problem that was this large. Over the course of the project I learned many things I didn't know before like how to develop applications in CUDA and how to compile applications written in multiple programming languages. This was also my first experience designing code that executes in parallel instead of sequentially. I am proud of what I accomplished in the time I was given and I know this experience will help me in my future endeavors in graduate school.
REFERENCES


