Parallelization of Livermore Solver for Ordinary Differential Equations Using NVIDIA's CUDA Architecture

Paul T. Thompson
Office of Science, Science Undergraduate Laboratory Internship Program

Western Michigan University, Kalamazoo

Princeton Plasma Physics Laboratory
Princeton, New Jersey

August 27, 2009

Prepared in partial fulfillment of the requirement of the Office of Science, Department of Energy's Science Undergraduate Laboratory Internship under the direction of Stephané Ethier in the Theory department at Princeton Plasma Physics Laboratory.

Participant: _____
Signature

Research Advisor: _____
Signature

PARALLELIZATION OF LIVERMORE SOLVER FOR ORDINARY DIFFERENTIAL EQUATIONS USING NVIDIA'S CUDA ARCHITECTURE

PAUL T. THOMPSON

INTRODUCTION

When a Tokamak, such as the National Spherical Tokamak Experiment (NSTX), ignites the brief flame of nuclear fusion, all manner of sensors and detectors are standing by, waiting to collect data. Monitored by physicists eager to eek out even the smallest improvement in confinement time, temperature, and electron pressure, these diagnostics are critical tools in developing and furthering our understanding of plasma physics. However, they are only the first step in the process. The data that's collected must still be processed. Until it is fed to the Black Box that is a data analysis program, it's all just numbers.

At the Princeton Plasma Physics Lab (PPPL), one of those all-important Black Boxes is called PIES, the Princeton Iterative Equilibrium Solver. At its core is a tried and true FORTRAN routine, the Livermore Solver for Ordinary Differential Equations (LSODE). LSODE does most of the heavy lifting for PIES, contributing a great amount to its required computation time.

Our task was to convert the LSODE routine from a regular, serially-executing Central Processing Unit (CPU) based function, to one that can be run on an NVIDIA Graphical Processing Unit (GPU). By delegating the LSODE calculations to the GPU, we hope to harness the massive parallel compute resources of modern graphics cards, and achieve significant speedups in PIES

run times. In this paper we discuss that process of delegation, our motivations, and the results we've achieved.

But before we get into the guts of the work, we first present some rudimentary background information on fusion and Tokamaks. This is to familiarize readers who may not have prior knowledge of the subject. From there, we move on to a brief description of the PIES program, including a discussion of why parallelization is both possible and desirable. Next, we discuss NVIDIA's C for CUDA Application Programming Interface (API), the underlying CUDA Architecture, and the hardware we used for development and testing. After that, we briefly discuss the "cuLsoda" library that we've developed, and present benchmark data comparing its performance running in parallel on a GPU, and running serially on a single CPU. Finally, we discuss the future of the library, and work that remains to be completed to integrate it with the PIES program.

Getting Up To Speed

This section is intended as a very brief overview of the basics of nuclear fusion, and Tokamak fusion reactors. Readers familiar with these topics are invited to skip ahead to the next section.

Roughly speaking, nuclear fusion occurs when two atoms collide with sufficient energy to make their nuclei stick together, fusing them into a single atom [1]. This goes beyond the simple construction of molecules through chemical bonding. In chemical bonding, we have one molecule, made up of several atoms which are linked together, but still identifiable as being

previous atoms. In the process of creating this single atom, which is no longer distinguishable as having previously been several, a great amount of energy is released. Our favorite example is to take two hydrogen atoms (two deuterium atoms, to be specific), and smack them together to make a single helium atom (a ^{3}He, to be even more specific). In the process, a very energetic neutron is released. This neutron can then be used to heat water, which drives turbines which, in turn, generate electricity. The actual process is, of course, a bit more complicated, but that's the gist of it. To make a rough comparison of the amount of energy that can be produced by fusion to what can be produced by burning fossil fuels, we note that a 16 ounce bottle of water can produce as much energy as a 55 gallon barrel of oil [2].

Now, one of our favorite ideas for creating and controlling fusion here on Earth is called a Tokamak. A Tokamak can be loosely (and comedically) described as "A Giant Electromagnetic, Hydrogen-Burning Doughnut." It consists of a large chamber, shaped like a doughnut (the proper term is 'torus', but doughnuts are more fun), which is then wrapped with many turns of wire. hydrogen gas is injected into the otherwise empty doughnut, and is heated up until the electrons begin to pop off, and go flying about (the hydrogen atoms 'ionize,' we say). At this point, the gas has become a plasma. Simultaneously, a current is passed through the coils wrapping the doughnut, creating a magnetic field. This magnetic field serves to contain the plasma, keeping it from leaking all over the place, burning up the walls of the chamber, etc. The plasma is then heated more and more, until the hydrogen ions start smacking into each other with enough energy to cause them to fuse. That's it.

This overview is, of course, a very rough one, to the point that any physicist reading it would not be blamed for breaking out in uncontrollable laughter. But it is nevertheless conceptually sufficient for the average reader to be able to appreciate the rest of this paper. We strongly encourage anyone who is interested in fusion and Tokamak reactors to make a visit to that often shunned, but now ubiquitous source of almost right information, Wikipedia [3]. The Wikipedia articles on fusion and on Tokamaks provide an excellent starting point for those who are eager to learn.

All About PIES

Now, with the basics under our belts, we move on to a brief discussion of PIES. PIES is used to calculate magneto-hydrodynamic equilibrium states of the plasma inside the torus [4]. One of the most computationally intensive steps in achieving this is the following of magnetic field lines. PIES does this by making a call to the LSODE routine to solve an Ordinary Differential Equation (ODE) for each of the field lines. Here is where our opportunity for parallelization shows itself. Because the number of field lines that must be followed is large, and because they are all independent of each other, PIES is a prime candidate for parallelization. Rather than doing the calculations serially, one after another, if we can compute the field lines simultaneously, we should be able to see a significant speedup.

Nevertheless, modifying existing code takes a lot of effort, and the general rule of thumb in production of tools like PIES is, ‘If it ain’t broke, don’t fix it’. So there must be a compelling reason to make changes. Fortunately (or unfortunately?), this reason comes in the form of run times. For a typical job sent to PIES, the time required to finish the calculations can range

and the nature of the data being analyzed [5]. In particular, the presence of stochastic (chaotic) regions in the plasma can extend computation time. Hence, any speedup is desirable, particularly if it can allow the code to run fast enough for an analysis to complete in between experimental shots. At this time, PIES has been partially parallelized, and is capable of running on multiple server nodes at once. However, the prospect of additional speed gains, by making use of the parallel resources of GPUs, provides the motivation for our work. So let's talk a bit about GPUs in general, and also the specific hardware that we used.

MATERIALS AND METHODS

GPGPU and NVIDIA's CUDA

General Purpose computing on GPUs (GPGPU) is a quickly developing sub-field of scientific computing [6]. It provides a new tool, a miniature parallel computer system, packed into the convenient form of a common graphics card. For example, NVIDIA's current top of the line gaming card, the GeForce GTX 295, contains 480 parallel processing cores [7]. This raw processing power, when combined with an appropriate problem (one that has a high ratio of arithmetic operations to memory operations), allows for greatly reduced computation times, and for large amounts of data to be processed very quickly [8].

NVIDIA's C for CUDA API enables programmers to make use of GPU resources, and presents them with a relatively small extension to the C programming language. Because it's based on C, the basics can be learned fairly quickly by programmers already familiar with C. However, due to some characteristics of GPU architecture, particularly the kinds of memory that available and how it's accessed, optimizing code to run on the GPU can be very challenging [9]. The

The basic way that CUDA works is that it takes a single function, and uses the GPU to execute it simultaneously on many different pieces of data. This is akin to shooting 100 arrows at once at 100 different targets, rather than shooting one arrow at each target sequentially. We say that the job has been split into several 'threads' of execution. For the arrow example, we would have created 100 threads, each of which shoots a target.

The programmer's job, then, is to frame their computational task in such a way that it can be divided up into multiple threads. Then, using the commands provided by the C for CUDA API, send these threads off to the GPU to be executed.

This is what we have done with our 'cuLsoda' library, which we'll talk about in just a moment.

The Hardware

First, we should say a word about the actual hardware we used. Testing and development of the cuLsoda library was conducted on two linux server nodes, each equipped with two quad-core Intel Xeon E5345 processors, running at 2.33GHz, and 16GB of RAM. Each node was also equipped with four NVIDIA Tesla C1060 computing processors. Tesla C1060s contain 240 processor cores distributed across 30 multiprocessors, and 4GB of on-board RAM.

The cuLsoda Library

To build our library, we started out with the FORTRAN version of the LSODA variant of LSODE. LSODA is LSODE, but with the added feature of being able to automatically switch methods, depending on whether the problem it's solving is 'stiff' or not [10]. The difference is minor, and the interfaces are similar enough that it is an easy substitution.

We then converted the FORTRAN code into C, and from there adapted the code to run on the GPU using CUDA.

Once the conversion was complete, and the code had been verified to produce correct output, we built two versions of a benchmark program. Both programs used the cuLsoda library to solve a sample system of ODEs. But where one was run on the GPU, the other used a version of the library which had been converted back to run exclusively on the CPU. The system of ODEs being solved was,

$$y_1'(t) = \cos(t)$$
$$y_2'(t) = \cos(2 * t)$$
$$y_3'(t) = \cos(3 * t)$$
$$y_4'(t) = \cos(4 * t)$$

The integration was done starting at t = 0, with output at t = 10 000.

For the CPU version of the benchmark, we solved this system n times, iteratively, for n = 1 to 1 024, and tracked the time, T, it took the program to complete for each value of n. The time ended up scaling linearly with n, and so we were able to easily extrapolate values for T all the way out to n = 2 048.

For the GPU version, we instead created n threads on the graphics card, for n = 1 to 2 048, and, again, tracked the completion time, T. It should be noted that the GPU benchmarking was done utilizing only a single Tesla graphics card, even though we had access to the two server nodes mentioned above.

The data was then plotted, along with the speedup (defined as T_{GPU}/T_{CPU}). These results are shown below in Figure 1.

RESULTS

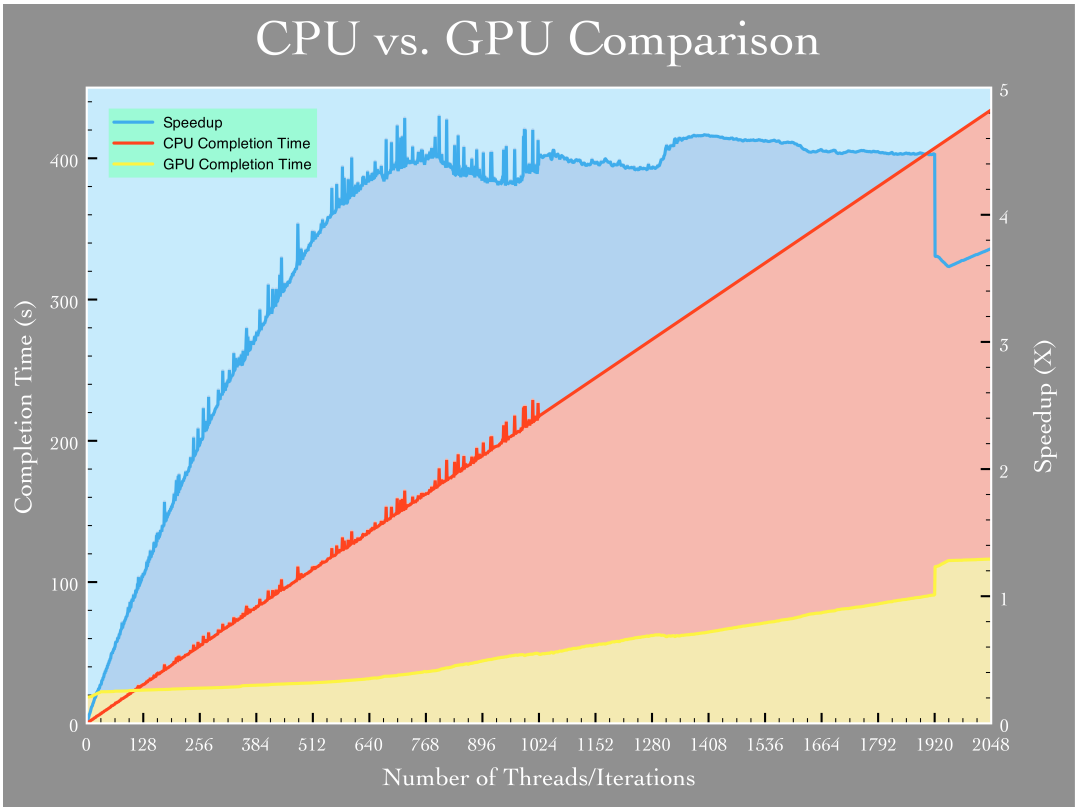

Figure 1. Benchmark results comparing CPU and GPU task completion times, and resulting speedup (T_{GPU}/T_{CPU}).

DISCUSSION AND CONCLUSION

The interesting features to note in the graph are, first, that the CPU completion time scales very linearly with n. This is expected, and is due to the fact that the CPU is simply iterating the call to the solver over and over.

Secondly, the GPU line initially scales exponentially (with a trend line of $T = 21.864 * e^{0.0005*n}$ seconds), up to about n = 580, or so. At that point, it transitions to a linear scaling (with trend line of T = 0.0379 * n + 7.3997 seconds). Our interpretation of this result is that with fewer than 580 threads, the GPU cores are underutilized. Up to this point, cores will have clock cycles

where they do nothing, as they wait for memory accesses to finish. We should note that in the CUDA architecture, NVIDIA minimizes these effects by having the processing cores do computations on one thread while waiting for another to finish accessing memory [11]. At the 580 thread mark, it appears that all cores have reached full efficiency, with no more wasted processor cycles. As we add threads after this, each contributes the same amount to the completion time. This is also reflected in the leveling out of the speedup curve.

The third feature of note is closely related to the second. Namely, the dramatic change that occurs at n = 1 921. Here, T jumps by more than 10s, with a corresponding crash in speedup. This is the result of the GPU reaching it's maximum capacity for simultaneous threads. Note that 1 921/240 = 8. This indicates that each processor core is able to handle eight threads at a time, before it must start queueing additional work. So the first 1 920 threads all begin execution approximately simultaneously, while the 1 921st gets stuck waiting for one of threads in the first batch to complete. Likewise for all subsequent threads. Examining the GPU line between 1 921 and 2 048, and comparing its shape to the region between 1 and 128, we get a hint that the same exponential-first-then-linear pattern may replicate as n increases.

The fourth feature of note is the difference between the CPU and GPU case in the value of T at n = 1. For a single iteration, it takes the CPU 0.21s to complete. However, the GPU requires 18.55s for a single thread. This dramatic difference is the result of slow memory accesses on the GPU. Because we did not optimize the GPU code, it fails to take advantage of many of the features and coding tricks which must be used to achieve good performance on the hardware. This is why it takes the GPU 76 times longer than the CPU to complete a single call to the cuLsoda library. Fortunately, this can be remedied, as will be discussed in the next section.

Future Work

A number of tasks remain in the development of the cuLsoda library. Because the code was converted directly from FORTRAN, it retains the majority of the GOTO statements that were in the original. Consequently, it is difficult to follow the flow of control, and debugging is particularly challenging. Though tedious, priority must be given to rewriting sections of the code where these statements appear. Additionally, the commenting and documentation included in the library is leftover from the FORTRAN version, and must be updated to reflect the changes we've made.

CUDA specific optimizations also need to be made to the code. These include making use of the fast 'shared' memory on the GPUs, as well as restructuring to reduce unnecessary duplication of variables, and possible integration of on-chip Basic Linear Algebra Subroutines (BLAS) contained in the CUBLAS library provided by NVIDIA. When optimizations are complete, we expect the single-thread runtime to be reduced from the 18 seconds mentioned above, to less than a second.

Lastly, and perhaps most importantly, a front-end remains to be built which will allow integration of the new cuLsoda library with PIES. Such an interface would ideally scale to take advantage of multiple GPUs located on separate nodes, and return metrics to the user to allow the monitoring of performance.

Development of the cuLsoda library will continue, and the project is being hosted by Google Code at http://code.google.com/p/culsoda/. Readers interested in contributing are invited to contact one of the project owners.

ACKNOWLEDGEMENTS

Dr. Stephané Ethier

This work was conducted from June through August of 2009 at the PPPL, under the guidance of Dr. Stephané Ethier, with significant assistance and contributions from Dr. Don Monticello, both of the Theory Department at PPPL. Thanks is due to the U.S. Department of Energy, and to PPPL for creating, organizing, and funding the SULI program.

REFERENCES

[1] R. Serway and J. Jewett, "Applications of Nuclear Physics," in *Physics for Scientists and Engineers with Modern Physics,6*th ed. Belmont, CA: Brooks/Cole--Thomson Learning, 2004, ch. 45, sec. 45.4, pp. 1487–1495.

[2] M. Mauel, "Fusion Energy in a Non CO_2 Emitting Energy Portfolio", http://fire.pppl.gov/aaas05_mauel_fusion.pdf (accessed August 18, 2009).

[3] http://www.wikipedia.org/.

[4] D. A. Monticello, e-mail message to author, August 28, 2009.

[5] S. Ethier, e-mail message to author, July 16, 2009.

[6] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, March 2007.

[7] NVIDIA, *NVIDIA CUDA Programming Guide*, ver. 2.3, NVIDIA Corporation, p. 101, July 1, 2009.

[8] NVIDIA, *NVIDIA CUDA Programming Guide*, ver. 2.3, NVIDIA Corporation, p. 3, July 1, 2009.

[9] NVIDIA, *NVIDIA CUDA Programming Guide*, ver. 2.3, NVIDIA Corporation, pp. 71–100, July 1, 2009.

[10] K. Radhakrishnan and A. Hindmarsh, *Description and Use of LSODE, the Livermore Solver for Ordinary Differential Equations*, NASA Reference Publication 1327, Lawrence Livermore National Laboratory Report UCRL-ID-113855, December 1993.

[11] NVIDIA, *NVIDIA CUDA Programming Guide*, ver. 2.3, NVIDIA Corporation, p. 77, July 1, 2009.