GPU ACCELERATION OF THE ISO–7 NUCLEAR REACTION NETWORK USING OPENCL

A Senior Scholar Thesis

by

DANIEL ALPHIN HOLLADAY

Submitted to Honors and Undergraduate Research
Texas A&M University
in partial fulfillment of the requirements for the designation as

UNDERGRADUATE RESEARCH SCHOLAR

May 2012

Majors: Nuclear Engineering
Physics

GPU ACCELERATION OF THE ISO–7 NUCLEAR REACTION NETWORK USING OPENCL

A Senior Scholar Thesis

by

DANIEL ALPHIN HOLLADAY

Submitted to Honors and Undergraduate Research
Texas A&M University
in partial fulfillment of the requirements for the designation as

UNDERGRADUATE RESEARCH SCHOLAR

Approved by:

Research Advisor:	Ryan McClarren
Associate Director, Honors and Undergraduate Research:	Duncan MacKenzie

May 2012

Majors: Nuclear Engineering
Physics

ABSTRACT

GPU Acceleration of the iso–7 Nuclear Reaction Network using OpenCL.
(May 2012)

Daniel Alphin Holladay
Department of Nuclear Engineering
Department of Physics
Texas A&M University

Research Advisor: Dr. Ryan McClarren
Department of Nuclear Engineering

We looked at the potential performance increases available through OpenCL and its parallel computing capabilities, including GPU computing as it applies to time integration of nuclear reaction networks. The particular method chosen in this work was the trapezoidal BDF-2 method using Picard iteration, which is a non-linear second order method. Nuclear reaction network integration by itself is a sequential process and not easily accelerated via parallel computation. However, in tackling a problem like modeling supernova dynamics, a spatial discretization of the volume of the star is necessary, and in many cases is combined with the computational technique of operator splitting. Every spatial cell would have its own reaction network independent of the others, which is where the parallel computation would prove useful. The particular reaction network analyzed is called the iso–7 reaction network that looks at the dynamics of 7 of the more dominant nuclides in supernovae. The computational performance was compared between the CPU and the GPU, in which the GPU showed performance increases of up to 8 times. This increase was realized on the small–scale because the computations were limited to running on a single device at any given time. However, these performance gains would only increase as the problem size was scaled up to the large–scale.

DEDICATION

This project is dedicated to my grandfather, former Master Chief Petty Officer Jerry Dan Alphin, who was recently diagnosed with AlzheimerÕs disease. Without him, I would not be where I am today.

ACKNOWLEDGMENTS

I would like to acknowledge Dr. Ryan McClarren for his assistance in learning the tremendous intricacies of OpenCL as well as graduate student Joshua Hansel for his LaTeX assistance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I
INTRODUCTION

It has been known for some time now that stars are limited in the size elements they can produce through nuclear fusion reactions. Not all processes that generate elements greater in size and proton number of iron 56 ($^{56}_{26}\text{Fe}$) are known. A known generator of such nuclides is via a supernova. Such an event is very complicated with hundreds of different large ions, charged particles, neutrons, and photons of all different energies interacting and reacting in a highly coupled manner [1]. These reactions are modeled through a system of coupled nonlinear ordinary differential equations (ODE) called nuclear reaction networks. The reaction networks can describe the gain and loss rates of all species from neutrons to heavy elements like uranium. These reaction networks are applicable to fuel burn–up in nuclear reactors as well. In many cases, it is not feasible or necessary to analyze networks that take into account all or even most of the different species. In fact, much of the relevant physics can be extracted by looking at a few of the dominant species and processes. This work will analyze these systems using such models that take into account a few dominant processes using a reaction network known as the iso–7 nuclear reaction network [2]. This network analyzes the coupling between 7 of the more dominant nuclides associated with supernovae dynamics, those being: $^{4}_{2}\text{He}$, $^{12}_{6}\text{C}$, $^{16}_{8}\text{O}$, $^{20}_{10}\text{Ne}$, $^{24}_{12}\text{Mg}$, $^{28}_{14}\text{Si}$, and $^{56}_{28}\text{Ni}$. Reaction networks are currently solved with numerical time integration techniques and equation linearization. Much work

This thesis follows the style of Journal of Computational Physics.

has been performed using first order linearization models and time integration of the ODE using standard computing architectures[3]. This research will look at the full non-linear system of ODE using similar time integration techniques, but with the addition of new and more efficient computing architectures.

Current high performance computers and supercomputers obtain their computational capabilities not from tremendously high clock speeds, but from using several processing units in parallel. It is not a trivial task to generate computer codes to optimize the use of multiple processors. For this reason, a group of corporations and organizations lead by Apple Inc. have created an open standard for massively parallel computing across many computational processing unit (CPU) and graphics processing unit (GPU) architectures called OpenCL. OpenCL stands for open computing language and is the culmination of tremendous efforts to bring parallel computing to non-computer scientists, and will be discussed in further detail later. The iso–7 reaction network will be integrated using OpenCL. The simplicity of this statement is deceptive, as doing so is not straightforward due to the fact that time integration of a system of differential equations is a sequential process, and therefore not parallelizable. Therefore the acceleration will be obtained in an indirect fashion. To numerically analyze supernova dynamics, one must not only discretize time, but also space since the star takes up non–zero volume. Acceleration will come from the fact that to accurately model a three dimensional star, there will be a unique iso–7 reaction network for each of the spatially discretized cells that make up the star, which could easily be in the millions and possibly in the billions.

Previous work

The dynamics of supernovae have been explored and analyzed thoroughly and much of these dynamics are understood, but improvement in these methods is always desired. The purpose of this work is not to break new ground with regard to reaction networks, but to show the performance increases achievable through the use of GPUs. This increase in performance will allow for cost effective computation of high fidelity, high spatial resolution, large–scale supernovae simulations, as well as similar allowable increases on smaller scale simulations.

Background

The iso–7 Reaction Network

As stated above, the iso–7 reaction network assumes 7 different groups of reactant types. In general, there are hundreds of different types of nuclides that are produced during supernovae, and the iso–7 approximates this large network with a much smaller one.

The Equations

Traditionally, this analysis begins by defining the mass of the i^{th} nuclide per unit volume to be ρ_i, such that:

$$\sum_{i=1}^{7} \rho_i = \rho \tag{1.1}$$

Where ρ is the total density of the volume under analysis. The mass fraction of nuclide i is the ratio of its density to the total density and is given by X_i. From this and equation (1.1), conservation of mass dictates that:

$$\sum_{i=1}^{7} \frac{\rho_i}{\rho} = \sum_{i=1}^{7} X_i = 1 \tag{1.2}$$

The mass fractions, X_i, are the desired quantities as they describe how much any given nuclide exists. However, reaction rates are based upon the number of particles and not by mass. The number density is the most useful quantity in terms of performing the analysis, while the mass fraction tends to be the desired quantity at the end of the analysis. The number density of the i^{th} nuclide is $n_i = \rho_i N_A / A_i$ where N_A is Avogadro's number and A_i is the atomic number of the i^{th} nuclide. Now, we define the dimensionless molar abundance for the i^{th} nuclide to be:

$$Y_i \equiv \frac{n_i}{\rho N_A} = \frac{X_i}{A_i} \tag{1.3}$$

The continuity equation for the dimensionless molar abundances is given by [2]:

$$\frac{\partial Y_i}{\partial t} + \frac{\partial}{\partial \vec{x}}\left(Y_i \vec{v}_i\right) + \frac{\partial}{\partial \vec{v}}\left(Y_i \vec{a}_i\right) = \dot{R}_i \tag{1.4}$$

where $\vec{v}_i$ is the velocity of the i^{th} nuclide dimensionless molar abundance, $\vec{a}_i$ is the acceleration of the i^{th} nuclide dimensionless molar abundance, and $\dot{R}_i$ is given by [2]:

$$\dot{R}_i = \sum_{j,k} Y_\ell Y_k \lambda_{kj}(\ell, \rho, T) - Y_i Y_j \lambda_{jk}(i, \rho, T) \tag{1.5}$$

where λ_{kj} is the gain rate of type i nuclides and λ_{jk} is the loss rate of type i nuclides. The partial derivative terms in equation (1.4) represent loss of mass from a given control volume due to advection via bulk flow. The other gain and loss rates are combined into $\dot{R}_i$ which is a function the λ. These coefficients are a function of the thermodynamic state of the region under analysis. The temperature and density specify this, and empirical formulas were used to compute these rates as a function of $\vec{Y}$, ρ, and T [3]. Equation (1.4) represents a system of partial differential equations (PDE), which are in principle much more difficult to solve than a system of ODE. In order to simplify the governing equations, this work will assume that operator splitting is employed. Operator splitting is the process of breaking down a problem into different components, such as spatial and temporal components, and allows for computer codes to solve each component separately. This decoupling results in a lower fidelity model, but is computationally much cheaper. The iso–7 reaction network seeks to solve only the temporal component of this problem, leaving out the spatial components. This means that all spatial and therefore velocity related terms won't be considered, and instead of needing to solve equation (1.4), the following equation will be considered:

$$\frac{\mathrm{d}Y_i}{\mathrm{d}t} = \dot{R}_i \tag{1.6}$$

This is the final form of the equation that will be considered in the analysis. This equation is of the form:

$$\frac{\mathrm{d}\vec{y}}{\mathrm{d}t} = \vec{f}\,(\vec{y}) \tag{1.7}$$

This equation is therefore a first order, nonlinear, system of ODE. There are many ways to solve a system of this form. These will be discussed in the methods section.

OpenCL

OpenCL stands for open computing language and lives up to its name, in that it is very open. The standard is managed by the Khronos group and has several partners that assist in improving OpenCL. Its main purpose is to make parallel computing using CPUs and GPUs easy to do for non-computer scientists as well as to create a universal standard so that any program written in OpenCL can be run in parallel on any machine capable of parallel computing [4].

Devices

One of the many benefits to OpenCL is its portability, or its ability to run on a number of different machines. This is achieved through the language's ability to query the machine as to the available computing devices [5]. These devices are typically either standard CPUs (processors) or GPUs (graphics cards). In any given machine, there could be several processors and graphics cards, all of which can be used simultaneously for computation. In addition, each device could have several cores, with each core capable of handling multiple threads. A thread is the most basic unit of computation capable of being sent to a processing unit, and it is merely a set of instructions for the processor. In executing a thread, the processor obeys the set of instructions delivered to to it. OpenCL is thus able to take advantage (hardware permitting) of multiple thread execution in a given core, for multiple cores in a given device, and multiple devices in a given machine.

Kernels

With the multithreading capabilities of OpenCL defined, the question of implementation must be answered. Once the available devices have been defined, a context for the devices to operate must be created, as well as a way to send and receive information from the devices [6]. Perhaps the most important component of implementation is the OpenCL kernel. A kernel is very similar to a function or a subroutine in that the kernel requires inputs and returns outputs. There can be several kernels in a given program, and each kernel can be executed an arbitrary number of times, each instance with its own set of inputs. The only limit to the number of simultaneous executions is set by the hardware. Each instance of the kernel and its associated input is known as a work item. A collection of work items is called a work group, and these structures can be very useful in breaking down a large problem into smaller constituents to be solved in an efficient manner [6]. Almost any problem, with the exception of purely sequential tasks can realize tremendous benefits from this multi-threading capability if run on conventional multicore CPUs. This is not necessarily true for GPUs.

GPU acceleration

Almost any program can be converted to OpenCL and run on a GPU, but not every program should be converted [4]. A GPU is designed to tackle very few types of problems with tremendous efficiency. These problems, as the name implies, are typically graphics intensive tasks. Graphics intensive tasks tend to involve a lot of computation on a very small set of memory. Because of the nature of these tasks, GPUs are designed with lots of low power processor cores in a small space, and with very little memory allotted to each core. Tasks that are highly parallelizable can be performed with significantly less power than that which is necessary in conventional CPUs. It turns out that power consumption plays a significant role in limiting supercomputer performance.

The power problem

Power is a requirement to perform computations, not only to push electrons through the circuits in the processors, random access memory (RAM), and other components, but also to keep these systems cool. As electrons flow through the circuits, they collide with the material, thus depositing some of their kinetic energy into the material in the form of heat. These complex systems have very specific nominal operating conditions, and therefore significant cooling is needed, which consumes even more power. To illustrate this, the Terascale Simulation Facility (TSF) at the Lawrence

Livermore National Laboratory (LLNL) will be studied. TSF houses several different supercomputing systems, including Purple and BlueGene/L, capable of 460 trillion operations per second, and consumes 25 Megawatts of power [7]. Assuming a linear relationship, it would require over 50 GW of power to break the exascale barrier, or to reach 1 quintillion operations per second. This is called the exascale barrier for this very reason, it simply requires too much power and is not feasible to perform computations on the exascale using supercomputing systems similar to this one.

Conversely, the Nuclear Engineering department has recently purchased a graphics cluster capable of 8 trillion operations per second, yet consumes $\sim$ 2 kW of power. It would require around 200 MW of to break the exascale barrier with this level of performance scaling, which is a significant improvement and the main reason why low power consumption is so important when discussing large-scale computation.

CHAPTER II

METHODOLOGY

Solution techniques

The iso–7 reaction network takes the form of equation (1.7). This is a non-linear system of ODE. Every worthwhile method to solve these systems of ODE is going to have its advantages, as well as its shortcomings; no method is significantly better than any other. In previous work, a linearization technique was used, which will be outlined below.

Previous work

In general, for a function $f : \mathbb{R}^n \to \mathbb{R}^n$, a first order Taylor expansion of this function at $\vec{y}_{i+1}$ about $\vec{y}_i$ is given by:

$$\vec{f}\,(\vec{y}_{i+1}) \approx \vec{f}\,(\vec{y}_i) + \mathbf{J} \cdot (\vec{y}_{i+1} - \vec{y}_i) \tag{2.1}$$

Where $\mathbf{J}$ is the Jacobian matrix, in which the element occupying the i^{th} row and j^{th} column is given by:

$$J_{ij} = \frac{\partial f_i}{\partial y_j} \tag{2.2}$$

Where i and j are specifying the element of the vectors $\vec{f}$ and $\vec{y}$. Using the standard first order finite difference with time step size h, equation (1.7) can be approximated by:

$$\left(\frac{\delta_{ij}}{h} - J_{ij}\right)\Delta_j = f_i\left(\vec{y}\right) \tag{2.3}$$

$$\vec{\Delta} \equiv \vec{y}_{i+1} - \vec{y}_i \tag{2.4}$$

Where h is the time step size and δ_{ij} is the Kronecker delta. This equation is a simple linear system which can be solved by standard means, such as Gaussian elimination or LU decomposition. This is the linearized backward Euler method, also known as the BDF-1 method, which will be discussed in more detail later. Other previous methods have used this same linearization technique but a different method to approximate the time derivative, such as implicit Runge-Kutta. Despite the high accuracy with respect to time that these methods can attain, they all linearize the function $\vec{f}$, which is not a good approximation if $\vec{f}$ is highly non-linear. Additionally, the Jacobian matrix, $\mathbf{J}$ must be stored and evaluated at every time step, which requires much more memory than just storing vectors. As previously discussed, for GPU acceleration, methods requiring the least amount of memory are preferred. The method chosen is discussed below and does not require storing the Jacobian matrix. For the iso–7 network, this saves 42 variables that do not have to be stored for every network. As already stated, each spatially discretized cell would have its own network to solve. This means that a method that does not require the Jacobian matrix saves 42 times the number of spatial cells, which could easily be in millions or billions. The IEEE standard 754 for Binary Floating-Point Arithmetic states that all compliant single precision numbers consume 4 bytes of memory, which could potentially free hundreds of gigabytes of memory that could be used for other things.

Backward difference methods

Backward difference formula (BDF) methods are methods that solve a system of ODEs by evaluating the function of $\vec{f}\,(\vec{y})$ at the $i+1$ time–step, as follows:

$$\frac{\mathrm{d}\vec{y}^{\;i+1}}{\mathrm{d}t} = \vec{f}\left(\vec{y}^{\;i+1}\right) \tag{2.5}$$

The nomenclature associated with BDF methods is based upon the order accuracy to which the time derivative is approximated. The BDF-1 method is first order accurate with respect to the discretized time–step size, Δt, via the first order finite difference approximating the time derivative. The order of accuracy of a finite difference refers to the size of the error as a function of the discretized step size. If a method is said to be n^{th} order accurate, the error associated with the approximation will vary as Δt^n. This means that if the step size is cut in half, the error in the approximation will drop by approximately a factor of 2^n. The BDF-1 method is given below:

$$\frac{\vec{y}^{\;i+1} - \vec{y}^{\;i}}{\Delta t} = \vec{f}\left(\vec{y}^{\;i+1}\right) \tag{2.6}$$

As previously stated, this method is more commonly known as the backward Euler method. A higher order method is known as the BDF-2 method, which is second order accurate with respect to the time–step size. This method approximates the time derivative evaluated at the $i+1$ time–step in the following manner:

$$\frac{\mathrm{d}\vec{y}^{\;i+1}}{\mathrm{d}t} = \frac{1}{2}\left(\frac{\mathrm{d}\vec{y}^{\;i+3/2}}{\mathrm{d}t} + \frac{\mathrm{d}\vec{y}^{\;i+1/2}}{\mathrm{d}t}\right) \tag{2.7}$$

Similar averaging for the $i+1/2$ time–step yields [8]:

$$\frac{\mathrm{d}\vec{y}^{\;i+3/2}}{\mathrm{d}t} = 2\frac{\mathrm{d}\vec{y}^{\;i+1/2}}{\mathrm{d}t} - \frac{\mathrm{d}\vec{y}^{\;i-1/2}}{\mathrm{d}t} \tag{2.8}$$

Using these two equations, and first order differencing of the derivatives, one arrives at the final form of the BDF-2 method:

$$\frac{\mathrm{d}\vec{y}^{\,i+1}}{\mathrm{d}t} = \frac{3}{2}\left(\frac{\vec{y}^{\,i+1} - \vec{y}^{\,i}}{\Delta t}\right) - \frac{1}{2}\left(\frac{\vec{y}^{\,i} - \vec{y}^{\,i-1}}{\Delta t}\right) = \vec{f}\left(\vec{y}^{\,i+1}\right) \tag{2.9}$$

First order terms with respect to the time–step sum to zero, and the method is second order, as the name implies. However, this method requires that $\vec{f}\,(\vec{y}^{\,i+1})$ is a known quantity, and unfortunately it is not. In addition, the BDF-2 method requires that both $\vec{y}^{\,i}$ and $\vec{y}^{\,i-1}$ are known, which is undesirable from a computational perspective. The BDF-2 method can be modified by taking a half time step, and this is called the trapezoidal BDF-2, or T/BDF-2 method, in which the successive value of $\vec{y}$ relies only on the previous iterate [8]. The trapezoidal aspect comes from the way in which $\vec{y}$ at the $i + 1/2$ is approximated:

$$\frac{\mathrm{d}\vec{y}^{\,i+1/2}}{\mathrm{d}t} = \frac{1}{2}\left(\vec{f}\left(\vec{y}^{\,i+1/2}\right) + \vec{f}\left(\vec{y}^{\,i}\right)\right) \tag{2.10}$$

Using the standard first order finite difference for the time derivative, and solving for $\vec{y}^{\,i+1/2}$:

$$\vec{y}^{\,i+1/2} = \vec{y}^{\,i} + \frac{\Delta t}{4}\left(\vec{f}\left(\vec{y}^{\,i+1/2}\right) + \vec{f}\left(\vec{y}^{\,i}\right)\right) \tag{2.11}$$

The standard BDF-2 method is then applied using the half time step by replacing $\vec{y}^{\,i}$ with $\vec{y}^{\,i+1/2}$, $\vec{y}^{\,i-1}$ with $\vec{y}^{\,i}$, and Δt with $\Delta t/2$ in equation (2.9).

$$\vec{y}^{\,i+1} = \frac{1}{3}\left(4\vec{y}^{\,i+1/2} - \vec{y}^{\,i} + \Delta t\vec{f}\left(\vec{y}^{\,i+1}\right)\right) \tag{2.12}$$

Note that equations (2.11) and (2.12) are correct, but are not in closed form, and in general are transcendental. By very definition, they cannot in general be put into closed form.

Nonlinear solution methods

Methods for solving nonlinear equations include Newton's method, fixed point iteration, and bisection, among others. Newton's method would require the computing, storing, and inverting of the Jacobian matrix which, as already stated, is not suitable for GPU implementation. Bisection does not require the Jacobian matrix, but does require knowledge of the solution domain. A variant of fixed point iteration, called Picard iteration, was the method chosen for use in this implementation. This method simply guesses a value for $\vec{y}^{\,i+1}$ and $\vec{y}^{\,i+1/2}$, and then improves upon the guess by iteration [8]. This method is described by two indices, an iteration index ℓ, and a timing index i. Equations (2.11) and (2.12) are modified to achieve a closed form as follows:

$$\vec{y}^{\,i+1/2,\ell} = \vec{y}^{\,i} + \frac{h}{4}\left(\vec{f}\left(\vec{y}^{\,i+1/2,\ell-1}\right) + \vec{f}\left(\vec{y}^{\,i}\right)\right) \tag{2.13}$$

$$\vec{y}^{\,i+1,\ell} = \frac{1}{3}\left(4\vec{y}^{\,i+1/2,\ell} - \vec{y}^{\,i} + h\vec{f}\left(\vec{y}^{\,i+1,\ell-1}\right)\right) \tag{2.14}$$

This procedure is seeded by letting $\vec{y}^{\,i+1/2,0} = \vec{y}^{\,i}$ and $\vec{y}^{\,i+1,0} = \vec{y}^{\,i+1/2,1}$ This iteration process will continue until either the maximum number of allowed iteration steps has been reached or the relative difference between successive iterates is less than some tolerance, ε [8]. The maximum number of steps used was 1000, and in every case, the tolerance was met before the maximum number of iteration steps. The convergence criterion is given by:

$$\frac{\left|\left|\vec{y}^{\,i+1,\ell} - \vec{y}^{\,i+1,\ell-1}\right|\right|_2}{\left|\left|\vec{y}^{\,i+1,\ell}\right|\right|_2} < \varepsilon \tag{2.15}$$

This was implemented for $\varepsilon = 5 \times 10^{-15}$. The double bar in this case represents the L–2 norm, which is equivalent to the root mean square.

$$||\vec{v}||_2 = \sqrt{\sum_{i=1}^{n} v_i^2} \tag{2.16}$$

Expectations

One is not able to simply state that either device, either the CPU or GPU, will in general outperform the other. There are several parameters that one must look at. If only one spatial cell exists in the problem to be solved, both devices will perform extremely well as only a small portion of the available resources would be used in the calculation. In fact, the GPU would probably perform worse in that case due to high memory on and off loading time. At this point, additional spatial cells will negligibly effect the computation time, so the CPU will continue to outperform the GPU for small number of spatial cells. However, eventually all of the CPU resources will be used and computations will have to wait on previous computations before they can proceed. At this point, the computation time will start to increase approximately linearly with the number of cells, as in, if I double the number of cells, I will double the computation time. Most of the time, this will occur much earlier in the CPU than in the GPU because the GPU typically has at least a hundred or more processor cores available than does the CPU. Even so, the CPU will have to increase in compute time significantly before the GPU will surpass the CPU in performance. However, due to more resources, the GPU should win out in a big way in the limit of very large numbers of spatial cells. This crossing point depends not only on the number of spatial cells and the particular devices being used to execute the computation, but also on the amount of computation that occurs in each spatial cell. Evolving the abundances of the nuclides by 10^4 time–steps requires significantly more computation per spatial cell than does two time steps. Memory on and off loading will be a significant portion of overall compute time if there are few time steps to compute, and therefore, the number of spatial cells required for the GPU to surpass the CPU will increase. These are the expected results, but only the actual results matter, which will be discussed next.

CHAPTER III

RESULTS

This section outlines the results obtained by running OpenCL on two different machines. The first machine is a macbook pro with an Intel® Core™ i7-2820 QM for the CPU and an AMD ATI Radeon HD 6750M graphics card for the GPU. The second machine is a mac pro running an Intel® Xeon™ X5650 for the CPU and an AMD ATI Radeon HD 5770 graphics card for the GPU.

Test problem

In order to solve any given system of ODE, the initial conditions must be known. This not only includes the initial dimensionless molar abundances of all of the nuclides, but also the initial thermodynamic state of the system. In this case, the system is one of the spatial cells in the star. In previous analyses, there were two different ways of treating the thermodynamic quantities, T and ρ. The first case is the adiabatic, or "explosive" regime, in which the temperature and density have reached a maximum value, and adiabatically relax into a lower temperature and lower density state. The second regime is called the hydrostatic burning regime and assumes constant temperature and density [3]. This is the particular problem that was solved in this case. More specifically, an initially pure silicon volume at constant temperature $T = 6 \times 10^9\ K$ and constant density ρ was allowed to evolve with time–step size 1×10^{-11} seconds. To semi–verify that the code was working properly, a density of $1 \times 10^7\ \frac{\mathrm{g}}{\mathrm{cm}^3}$ was chosen to be compared with previous results [3]. The code was executed for 10^4 time–steps and compared with previous results. Unfortunately, the comparison was riddled with inaccuracies because the results were not tabulated, but graphed on a log-log scale. Additionally, the problems were solved in different ways as the test problem was solved in previous work by linearizing the equations. Despite these differences, the results of the two methods were almost identical. Another hydrostatic problem was looked at and the code results were correct for early times, but there was an issue with the production of ${}^{56}_{28}\mathrm{Ni}$. Future work should focus on a more rigorous verification study of this code, as well as the issue associated with ${}^{56}_{28}\mathrm{Ni}$ production. Despite these issues, due to low time, the parallel study was conducted with the code as it was.

Parallel issues

Unfortunately, there were some unforeseen issues executing the code on the graphics cards. The issue was traced to a single line of code that computed a portion of the reaction rate of ${}^{24}_{12}\mathrm{Mg}$. The reason this particular line of code is causing problems is still unclear. To get around this issue for the parallel study, that line of code was commented out. This meant that the results obtained in these studies are incorrect. However, the amount of computation performed by this modified code is almost identical to the previous problem, but with slightly different results. Since the results are not the key aspect of the parallel study, this should not effect the results. Once the problem is resolved, it is very reasonable to assume that both codes will exhibit similar, if not identical scaling with respect to increased number of spatial cells. In execution of the scaling, each spatial cell was given a different density ranging between $1 \times 10^7\ \frac{\mathrm{g}}{\mathrm{cm}^3}$ and $1 \times 10^9\ \frac{\mathrm{g}}{\mathrm{cm}^3}$ to show that parameters could be easily varied between the spatial cells.

Scaling results

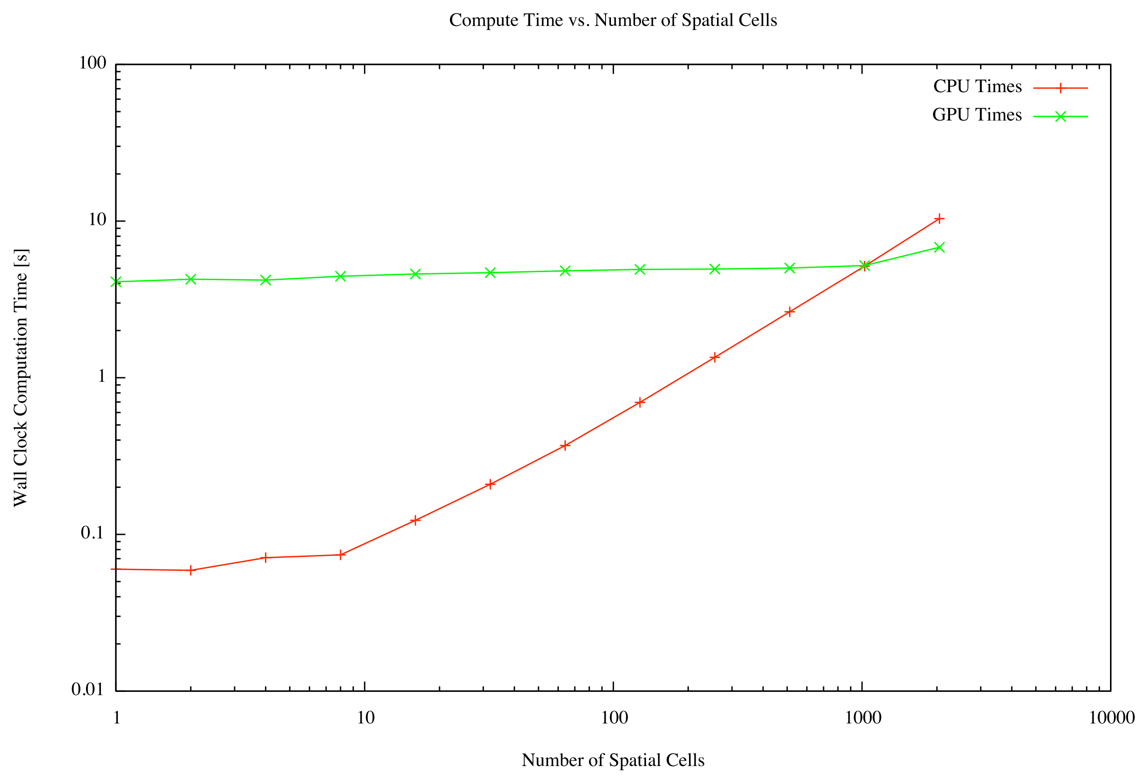

Fig. 1. GPU–CPU comparison with 10^4 steps. GPU compute time drops below CPU compute time just after 1000 spatial cells given 10^4 time–steps

Results for both machines using different numbers time–steps are given. The macbook pro results with 10^4 time–steps only go up to 2048 spatial cells because beyond this number, the GPU would crash. The mac pro would also crash given enough spatial cells, but it was able to handle many more spatial cells. The particular code used was only able to be executed on a single device at a time. However, OpenCL is capable of executing kernels on several devices simultaneously.

Macbook pro

Below are scaling results showing compute time vs. number of spatial cells for two different problems. In the first problem, at each cell, the code computed 10^4 time–steps, which was the highest number of time–steps analyzed in the scaling analysis. In the second problem, the code computed 10^2 time–steps per cell.

10^4 time–steps

With 10^4 time steps, we expect that the GPU will perform better with the fewest number of spatial cells of all of the comparisons because it has the most time–steps per cell.

In **Fig. 1**, notice that the CPU computation time remained approximately constant when the number of spatial cells was $\lesssim 10$. This is the region in which not all of the available resources are being taken advantage of, and will from now on be referred to as the flat region. After this point, the computation time increases swiftly with the number of spatial cells, which is the region in which all available computational resources are saturated, and from now on will be called the saturated region. These two regions are very important in describing the conditions for which a GPU will outperform a CPU. In the case displayed in **Fig. 1**, the GPU had just reached the saturated region when it surpassed the CPU, and even though there is only one data point, it appears that the computation time will not increase as swiftly with respect to the number of spatial cells, which means that it will perform even better when compared to a CPU when the number of spatial cells is large. From now on, the number of spatial cells in which the GPU surpasses the CPU in performance will be called the GPU crossover point.

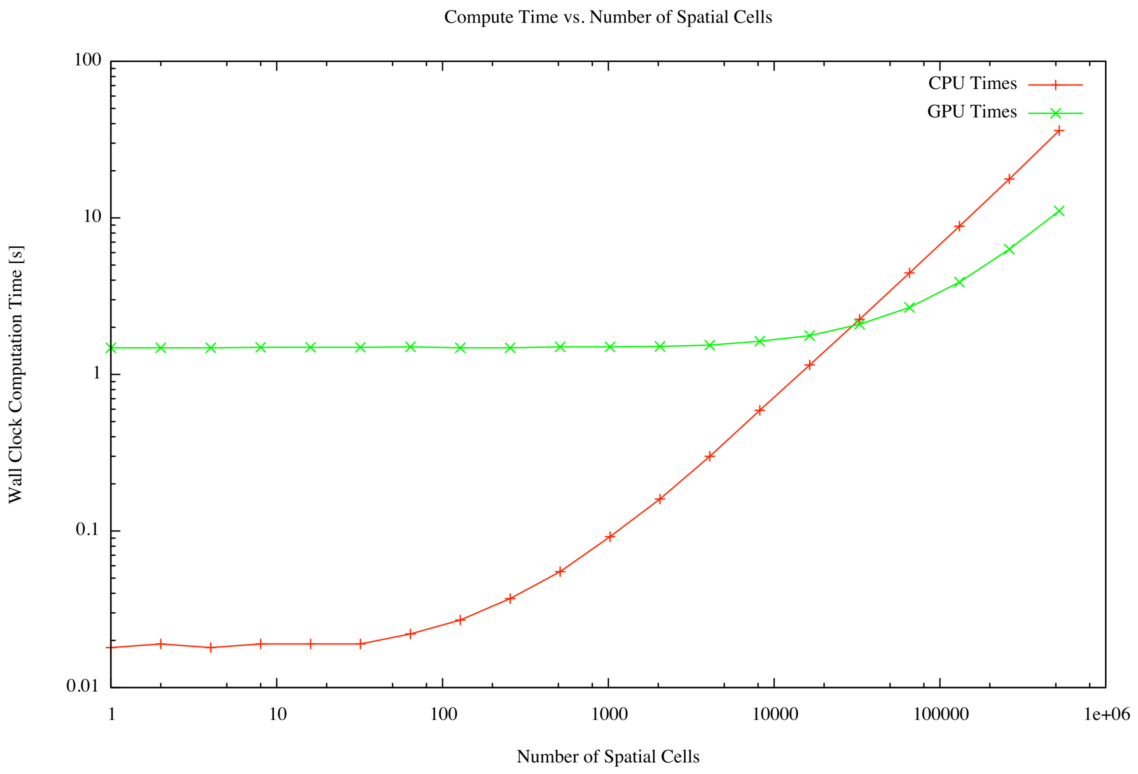

Fig. 2. GPU–CPU comparison with 10^2 steps. GPU compute time drops below CPU compute time just after 50000 spatial cells.

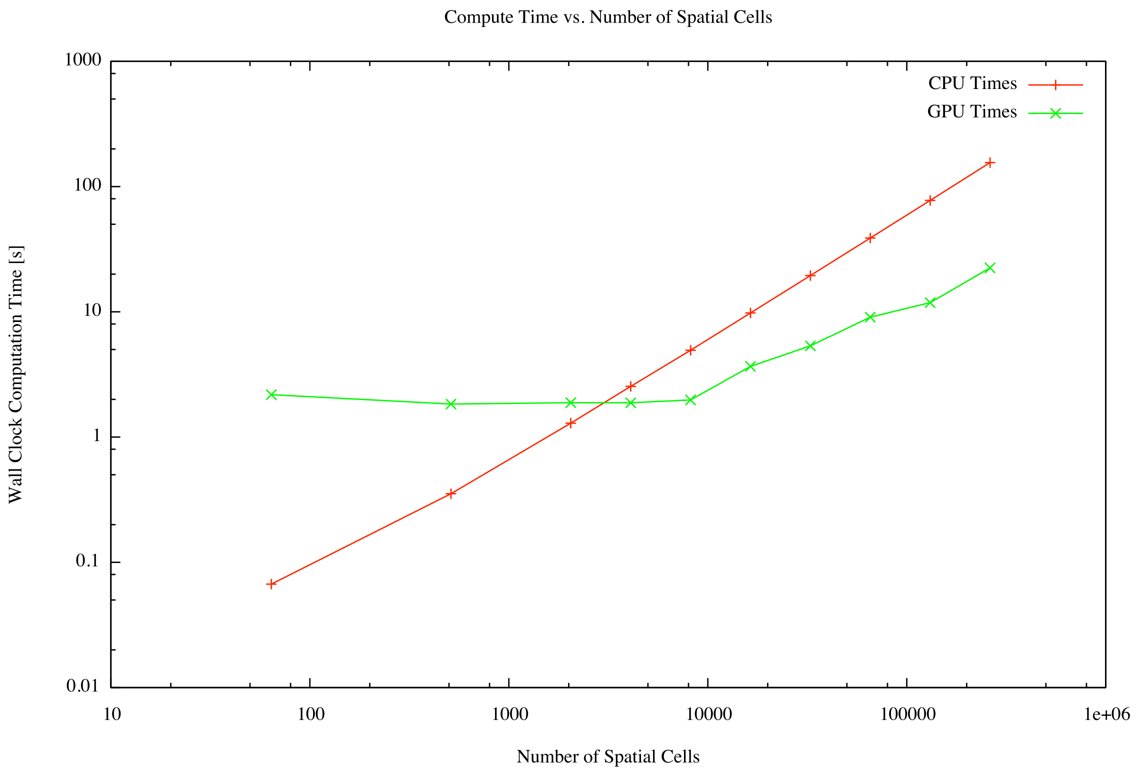

Fig. 3. GPU–CPU comparison with 10^3 steps. GPU compute time drops below CPU compute time just after 5000 spatial cells given 10^3 time–steps

10^2 time–steps

With 10^2 time steps, we expect that the GPU crossover point will occur at a higher number of spatial cells than in the previous comparison.

With less computation occurring per spatial cell, the flat region of both the CPU and the GPU were larger and thus the GPU crossover was shifted to the right. This shift is clearly seen in **Fig. 2** as the crossover occurred after 50000 spatial cells, especially compared to the 2000 cells needed for crossover in the previous of 10000 time–steps per spatial cell.

Mac pro

Below are scaling results showing compute time vs. number of spatial cells for two different problems. In the first problem, at each cell, the code computed 10^3 time–steps. In the second problem, the code computed 2 time–steps at each cell. Additionally, since this machine has higher performance devices, we expect that the flat regions will be larger for both devices, which will shift the GPU crossover point further to the right (i.e. higher spatial cell number).

10^3 time–steps

With 1000 time–steps, the GPU crossover will occur relatively early and this should show how a GPU can not only outperform a CPU, but do so by a significant factor.

It it especially important to note that all figures are plotted on a log-log scale, but it is especially important in the case of **Fig. 3**. At around 10^5 spatial cells, the GPU is performing almost an order of magnitude faster than the CPU, and trending to even better performance with more spatial cells. This set of data was able to show the saturated region of the GPU more fully than **Fig. 1**. Additionally, the mac pro used was equipped with 2 6–core processors and 2 graphics cards, but only one at a time was used. If both were used, the flat regions would be even larger (by a factor of 2 or so) for both the CPU and the GPU, which would probably mean even better performance gains by the GPU.

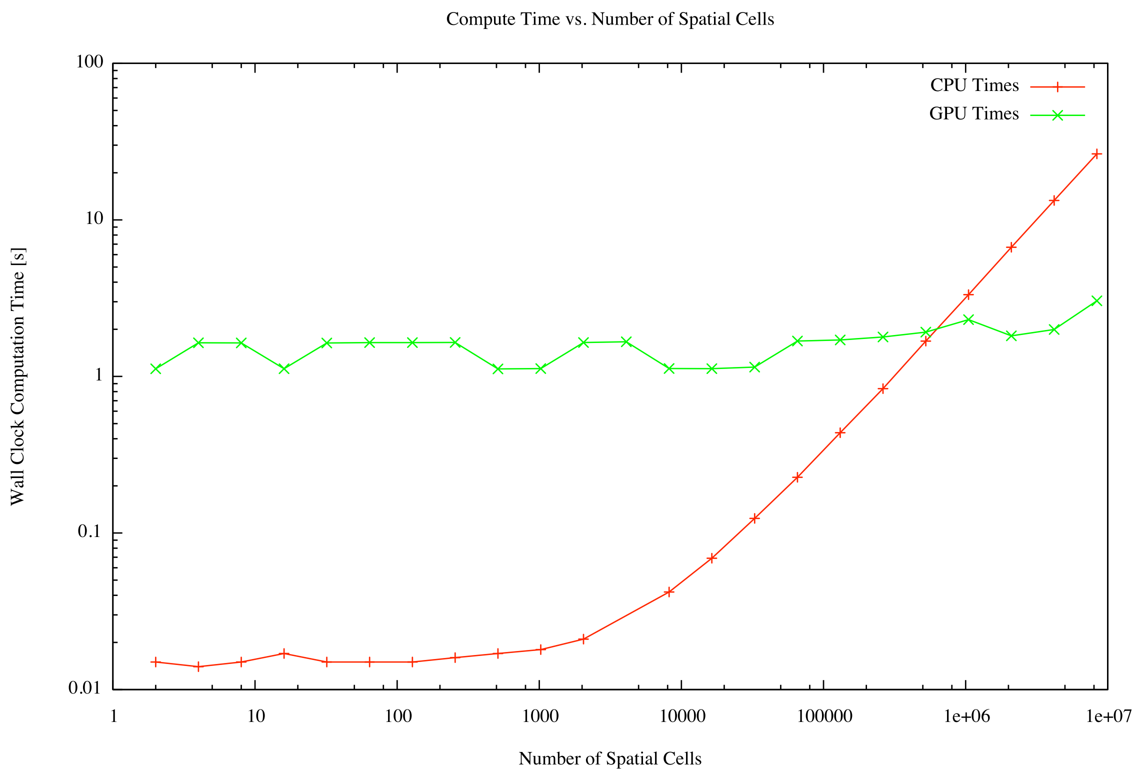

Fig. 4. GPU–CPU comparison with 2 steps. GPU compute time drops below CPU compute time just after 500000 spatial cells given 2 time–steps

2 time–steps

Fig. 4 shows that with 2 time–steps, the GPU crossover did not occur as early as in the case with 1000 time–steps per cell, but is still quite convincing.

The flat regions are quite visible in **Fig. 4** and because the CPU flat region is so large, the GPU crossover point is shifted far to the right. However, even with only 2 time–steps per cell, the GPU is outperforming the CPU by almost an order of magnitude at just under 10 million spatial cells. This analysis had the highest crossover point, yet the GPU would still easily outperform the CPU given the number of spatial cells necessary in a large–scale supernova simulation.

Quantitative performance increases

Table 1. Quantitative GPU speedup results

Machine	Number of time–steps	Number of cells	Factor speedup
Macbook pro	10^4	2048	1.52
Macbook pro	10^2	524288	3.26
Mac pro	10^3	262144	6.92
Mac pro	2	8388608	8.68

Again, code execution could only take place on a single device, which greatly limits the performance increases. Even so, nearly order of magnitude speedups were obtained, as seen in **Table 1**.

CHAPTER IV
CONCLUSION

The iso–7 nuclear reaction network was able to highlight the benefits of OpenCL and its applications to parallel programming on GPUs. There are several unsolved issues, like the $^{24}_{12}$Mg production rate issue, and the solutions to such problems are not easy to find due to the fact that GPU programming is still in its infancy. Despite these setbacks, high levels of performance increases were attained as a result of the proper choice of reaction network and the time integration strategy used to solve the system ODE. The T/BDF-2 method not only saved on the amount of memory required, which is extremely important when working with GPUs, but also allowed for the realization of the full nonlinear solution. This solution method did not require linearization of the reaction rates, and is also second order with respect to the time–step size. The T/BDF-2 method in conjunction with the use of OpenCL highlighted the significant performance increases attainable through smart usage of GPUs, with performance increasing by up to a factor of 8 in small–scale problems. However, this implementation will not only scale to larger systems very efficiently with even better performance, but said transfer to large–scale systems will be seamless because of OpenCL. In the future, we will add the ability to execute the iso–7 network and other such OpenCL codes on multiple devices simultaneously which will allow for the study of large–scale problems.

REFERENCES

[1] S. E. Woosley, R. D. Hoffman, The α–process and the r–process, The Astrophysical Journal 395 (1992) 202–239.

[2] F. X. Timmes, Integration of nuclear reaction networks for stellar hydrodynamics, The Astrophysical Journal Supplement 124 (1999) 241–263.

[3] F. X. Timmes, R. D. Hoffman, S. E. Woosley, An inexpensive nuclear energy generation network for stellar hydrodynamics, The Astrophysical Journal Supplement 129 (2000) 377–398.

[4] D. W. Gohara, Opencl: Episode 1 – introduction to opencl, `http://www.macresearch.org/files/opencl/Episode_1.mov` (Aug. 2009).

[5] D. W. Gohara, Opencl: Episode 2 – opencl fundamentals, `http://www.macresearch.org/files/opencl/Episode_2.pdf` (Aug. 2009).

[6] D. W. Gohara, Opencl: Episode 3 – building an opencl project, `http://www.macresearch.org/files/opencl/Episode_3.pdf` (Sep. 2009).

[7] D. Sprouse, Terascale simulation facility: Built for flexibility, `https://www.llnl.gov/str/JanFeb05/Atkinson.html` (Jan. 2005).

[8] J. E. Morel, Lecture 8: Time discretization, Texas A&M University Lecture, nuclear Engineering 430 (2010).

CONTACT INFORMATION

Name:	Daniel Alphin Holladay
Professional Address:	c/o Dr. Ryan McClarren Department of Nuclear Engineering Texas A&M University 3133 TAMU College Station, TX 77843-3133
Email Address:	dholladay00@gmail.com
Education:	B.S., Nuclear Engineering, B.S., Physics, Texas A&M University, May 2012 Undergraduate Research Scholar