

# GPUs and Einstein's Equations

Timothy Dewey  
tdewey@math.umd.edu  
Advisor: Dr. Manuel Tiglio  
tiglio@umd.edu  
Physics Department, CSCAMM\*  
University of Maryland

May 17, 2011

## Abstract

This paper provides an outline of an implementation of a PDE solver for a simple 1-d case of Einstein's equations using CPUs and GPUs. We provide a brief introduction to the project and then provide an outline of the implementation and validation. Finally, we present a performance comparison between the GPU and CPU implementation.

## 1 Introduction

The numerical relatively community is depending on efficient large-scale simulations of binary black hole inspirals. We believe we can make significant time and power improvements over traditional CPU implementations by solving Einstein's equations with graphics processing units (GPUs). Einstein's equations are a good candidate for GPUs because of the computationally intensive right hand side (RHS) computations. Furthermore, the simulations are highly parallelizable. This type of application is where GPUs often show significant performance advantages over CPUs (see [1]).

The purpose of this project is to develop code to solve a simple 1-d case of Einstein's equations using spectral methods. The initial prototype code is written in MATLAB. However, the main goal is to develop high performance code that can be run on GPUs. Beyond just the learning experience that this project affords, the hope is that the performance of this code will inform decisions about pursuing GPU implementations of large-scale simulations in numerical relativity.

---

\*Center for Scientific Computation and Mathematical Modeling

## 2 The Problem

The goal is to solve (numerically) the Schwarzschild equations for a spherically symmetric black hole. These equations are coupled hyperbolic equations that are first order in space and time. Since this is a spherically symmetric black hole, the coordinates are 1-d in space ( $r$  is the radius) instead of 3-d in space. In order to solve these equations, we use a pseudo-spectral collocation method (we will say spectral method from now on). There are six main variables  $g_{rr}, g_T, K_{rr}, K_T, f_{rrr}, f_{rT}$ , that describe a spherically symmetric metric on a Lorentzian manifold. The equations being solved are listed below (see [3] for more detail). A MATLAB function that performs the RHS computations is found in Appendix A.

$$\dot{g}_{rr} = \beta g'_{rr} + 2g_{rr}\beta' - 2\tilde{\alpha}g_{rr}^{1/2}g_T K_{rr}, \quad (1)$$

$$\dot{g}_T = \beta g'_T + 2/r\beta g_T - 2\tilde{\alpha}g_{rr}^{1/2}g_T K_T, \quad (2)$$

$$\begin{aligned} \dot{K}_{rr} = & \beta K'_{rr} - 1/g_{rr}^{1/2}g_T\tilde{\alpha}f'_{rrr} + 2K_{rr}\beta' - 8/g_{rr}^{1/2}\tilde{\alpha}f_{rrr}f_{rT} - 6g_{rr}^{1/2}/g_T\tilde{\alpha}f_{rT}^2 - \\ & 1/g_{rr}^{1/2}g_Tf_{rrr}\tilde{\alpha}' + 2/g_{rr}^{1/2}g_T/rf_{rrr}\tilde{\alpha} - g_{rr}^{1/2}g_T\tilde{\alpha}'' + 4g_{rr}^{1/2}g_T/r\tilde{\alpha}' + \\ & 2g_{rr}^{1/2}\tilde{\alpha}K_{rr}K_T - 1/g_{rr}^{1/2}g_T\tilde{\alpha}K_{rr}^2 - 6g_{rr}^{1/2}g_T\tilde{\alpha}/r^2 + 2/g_{rr}^{3/2}g_T\tilde{\alpha}f_{rrr}^2, \end{aligned} \quad (3)$$

$$\begin{aligned} \dot{K}_T = & \beta K'_T - 1/g_{rr}^{1/2}\tilde{\alpha}g_Tf'_{rT} + 2/r\beta K_T - 1/g_{rr}^{1/2}g_T\tilde{\alpha}'f_{rT} - 2/g_{rr}^{1/2}\tilde{\alpha}f_{rT}^2 + \\ & g_{rr}^{1/2}g_T\tilde{\alpha}/r^2 + 1/g_{rr}^{1/2}\tilde{\alpha}K_TK_{rr}g_T, \end{aligned} \quad (4)$$

$$\begin{aligned} \dot{f}_{rrr} = & -\tilde{\alpha}g_{rr}^{1/2}g_TK'_{rr} + \beta f'_{rrr} - \tilde{\alpha}/g_{rr}^{1/2}K_{rr}f_{rrr}g_T - 10\tilde{\alpha}g_{rr}^{1/2}K_{rr}f_{rT} + \\ & 12g_{rr}^{3/2}/g_Tf_{rT}\tilde{\alpha}K_T + 3\beta'f_{rrr} - 4g_{rr}^{3/2}K_T\tilde{\alpha}' + 8g_{rr}^{3/2}/rK_T\tilde{\alpha} + \\ & g_{rr}\beta'' - 4g_{rr}^{1/2}\tilde{\alpha}K_Tf_{rrr} - g_{rr}^{1/2}g_TK_{rr}\tilde{\alpha}' + 2g_{rr}^{1/2}g_T/rK_{rr}\tilde{\alpha}, \end{aligned} \quad (5)$$

$$\begin{aligned} \dot{f}_{rT} = & -\tilde{\alpha}g_{rr}^{1/2}g_TK'_T + \beta f'_{rT} + 2/r\beta f_{rT} + \beta'f_{rT} - g_{rr}^{1/2}g_TK_T\tilde{\alpha}' - \\ & \tilde{\alpha}/g_{rr}^{1/2}K_Tf_{rrr}g_T + 2\tilde{\alpha}g_{rr}^{1/2}K_Tf_{rT}. \end{aligned} \quad (6)$$

Note that  $\tilde{\alpha} = 1$ ,  $\tilde{\alpha}' = 0$ ,  $\beta = \sqrt{2/r}$ ,  $\beta' = -1/\sqrt{2r^3}$ , and  $\beta'' = 3/\sqrt{8r^5}$ . The initial values of the main variables are computed using the analytic solution described in Section 3, and the spatial derivatives of the main variables,  $g'_{rr}, g'_T, K'_{rr}, K'_T, f'_{rrr}$ , and  $f'_{rT}$  are approximated by performing a matrix vector multiplication as described in the following section.

### 2.1 Spectral Methods

In order to develop a spectral solver for Einstein's equations, we need to be able to approximate the derivative of a smooth function (i.e., a function with several derivatives that exist). Using spectral methods, the error in this approximation can converge to 0 faster than any power law convergence. In order to realize this "spectral convergence", we use the roots of orthogonal

polynomials (e.g., Legendre or Chebyshev) as collocation points. Specifically, we use Chebyshev polynomials with the Gauss-Lobatto points (see [4] for more details).

The Chebyshev collocation points are  $x_i = \cos(\pi \cdot i/N)$ , for  $i \in \{0, \dots, N\}$ . These points determine a dense  $(N+1) \times (N+1)$  differentiation matrix  $\mathbf{D}$ . The definition of  $\mathbf{D}$ , as found in [4], is as follows:

$$\mathbf{D}_{00} = \frac{2N^2 + 1}{6}, \quad (7)$$

$$\mathbf{D}_{NN} = -\frac{2N^2 + 1}{6}, \quad (8)$$

$$\mathbf{D}_{jj} = \frac{-x_j}{2(1-x_j^2)}, \quad j = \{1, \dots, N-1\}, \quad (9)$$

$$\mathbf{D}_{ij} = \frac{c_i (-1)^{i+j}}{c_j (x_i - x_j)}, \quad i \neq j, \quad i, j = \{0, \dots, N\}, \quad (10)$$

where  $c_i = 2$  for  $i = 0$  or  $N$  and  $c_i = 1$  otherwise. MATLAB code that implements the above equations is found in Appendix A.

Define a column vector of the collocation points  $\mathbf{x} = \{x_0, \dots, x_N\}^T$ , and let

$$\mathbf{u}_{\mathbf{x}} = \{u(x_0), \dots, u(x_N)\}^T.$$

We can approximate the derivative of a function  $u(x)$  at the collocation points by simply performing the matrix vector multiplication  $\mathbf{D}\mathbf{u}_{\mathbf{x}} \approx \mathbf{u}_{\mathbf{x}}'$ . This computation is fast and requires little memory for small  $N$ . Typically,  $N \leq 70$  for large-scale simulations using a multiple-domain approach, and the total number of points across all domains is  $N^3$ . As the problem is broken into multiple domains, the minimum distance between collocation points gets smaller and it scales as  $O(N^2)$ . This means for large  $N$  this minimum distance becomes small and, in turn, the CFL limit becomes small. In order to keep a reasonable size for the time step, we want to keep  $N$  small. Furthermore, because we have spectral convergence, the error in our approximations will typically approach machine precision with  $N < 40$ .

Technically,  $\mathbf{x}$  is a vector of values from the interval  $[-1, 1]$ ; recall the Chebyshev collocation points are  $x_i = \cos(\pi \cdot i/N)$ , for  $i \in \{0, \dots, N\}$ . However, we want our collocation points to represent values from some other interval of the real line. The solution for this is to scale and shift the interval  $[-1, 1]$  to fit the interval of the spatial domain of interest. For example, we might choose the interval  $[1.9, 11.9]$  as our spatial domain. The units on the radius are in terms of the mass of the black hole (i.e.,  $M = 1$ ), and the event horizon of the black hole is at  $r = 2M$ . The choice of 1.9 as in the inner boundary is not arbitrary; 1.9 lies within the event horizon of the black hole. Because the inner boundary is inside the black hole, only the outer boundary requires enforcement of boundary conditions. This is discussed later.

Now, we shift and scale the interval  $[-1, 1]$  and get the new collocation points  $r_i = x_i \cdot ((r_{max} - r_{min})/2) + ((r_{max} + r_{min})/2)$ , where  $r_{min}$  is the inner boundary and  $r_{max}$  is the outer boundary. Let  $\mathbf{r} = \{r_0, \dots, r_N\}^T$ . To compute the approximation of a derivative at the points  $\mathbf{r}$ , we use:

$$\mathbf{u}_{\mathbf{r}}' \approx \mathbf{D}\mathbf{u}_{\mathbf{r}} \cdot \frac{dx}{dr} = \mathbf{D}\mathbf{u}_{\mathbf{r}} \cdot \frac{2}{r_{max} - r_{min}}. \quad (11)$$

## 2.2 Runge-Kutta and Boundary Conditions

In order to evolve Einstein's equations in time, we use a standard 4th order Runge-Kutta method as described in [5]. The method is outlined in Algorithm 1.

---

### Algorithm 1 Runge-Kutta

---

Let  $du/dt = f(t, u)$ .  
1: Set  $t_0 = 0$ ,  $u_0 = u(t_0)$ ,  $h = \Delta t$ .  
2: **for**  $n = 0$  to num\_steps  
3:    $t_{n+1} = t_n + h$   
4:    $d_1 = f(t_n, u_n)$   
5:    $d_2 = f(t_n + \frac{1}{2}h, u_n + \frac{1}{2}hd_1)$   
6:    $d_3 = f(t_n + \frac{1}{2}h, u_n + \frac{1}{2}hd_2)$   
7:    $d_4 = f(t_n + h, u_n + hd_3)$   
8:    $u_{n+1} = u_n + \frac{1}{6}h(d_1 + 2d_2 + 2d_3 + d_4)$

---

For each intermediate value in this algorithm, we must enforce the boundary conditions. As we mentioned before, the enforcement of boundary conditions is only required at the outer boundary (e.g.,  $r_{max} = 11.9$ ). We enforce the boundary conditions on the incoming characteristic variables (i.e., characteristic variables with positive speed). The characteristic variables  $v$  are as follows:

$$v_1 = g_{rr}, \quad (12)$$

$$v_2 = g_T, \quad (13)$$

$$v_3 = K_{rr} - g_{rr}^{-1/2} f_{rrr}, \quad (14)$$

$$v_4 = K_T - g_{rr}^{-1/2} f_{rT}, \quad (15)$$

$$v_5 = K_{rr} + g_{rr}^{-1/2} f_{rrr}, \quad (16)$$

$$v_6 = K_T + g_{rr}^{-1/2} f_{rT}. \quad (17)$$

We set the time derivatives  $\dot{v}_i$  to zero for  $i = \{1, \dots, 4\}$  (these are the characteristic variables with positive speed), then we solve for the time derivative of the main variables and assign this adjusted value to be the time derivative at the outer boundary. Hence we have:

$$\dot{g}_{rr} \leftarrow \dot{v}_1 = 0, \quad (18)$$

$$\dot{g}_T \leftarrow \dot{v}_2 = 0, \quad (19)$$

$$\dot{K}_{rr} \leftarrow (\dot{v}_3 + \dot{v}_5)/2 = (\dot{K}_{rr} - \dot{g}_{rr}^{-1/2} \dot{f}_{rrr} + \dot{K}_{rr} + \dot{g}_{rr}^{-1/2} \dot{f}_{rrr})/2, \quad (20)$$

$$\dot{f}_{rrr} \leftarrow \frac{\sqrt{g_{rr}}}{2}(\dot{v}_5 - \dot{v}_3) = \frac{\sqrt{g_{rr}}}{2}(\dot{K}_{rr} + \dot{g}_{rr}^{-1/2} \dot{f}_{rrr} - (\dot{K}_{rr} - \dot{g}_{rr}^{-1/2} \dot{f}_{rrr})), \quad (21)$$

$$\dot{K}_T \leftarrow (\dot{v}_4 + \dot{v}_6)/2 = (\dot{K}_T - \dot{g}_{rr}^{-1/2} \dot{f}_{rT} + \dot{K}_T + \dot{g}_{rr}^{-1/2} \dot{f}_{rT})/2, \quad (22)$$

$$\dot{f}_{rT} \leftarrow \frac{\sqrt{g_{rr}}}{2}(\dot{v}_6 - \dot{v}_4) = \frac{\sqrt{g_{rr}}}{2}(\dot{K}_T + \dot{g}_{rr}^{-1/2} \dot{f}_{rT} - (\dot{K}_T - \dot{g}_{rr}^{-1/2} \dot{f}_{rT})). \quad (23)$$

Note that in equations 21 and 23, the value  $\sqrt{g_{rr}}$  is the square root of the analytic value of  $g_{rr}$  as computed for the initial conditions. The outer boundary conditions are enforced after each approximation of the time derivative (i.e., lines 4-7 in Algorithm 1).

### 3 Validation

The final step is to validate the approximate solution that is computed numerically by the PDE solver. In order to validate the implementation, we rely on the Schwarzschild solution. This is an analytic solution to Einstein's equations for a spherically symmetric black hole (see [6], Section II.D.2). Below is the Schwarzschild solution given in [6] using the Painleve-Gullstrand coordinate system.

$$g_{rr} = 1, \tag{24}$$

$$g_T = 1, \tag{25}$$

$$K_{rr} = -\sqrt{\frac{M}{2r^3}}, \tag{26}$$

$$K_T = -\sqrt{\frac{2M}{r^3}}, \tag{27}$$

$$f_{rrr} = \frac{4}{r}, \tag{28}$$

$$f_{rT} = \frac{1}{r}, \tag{29}$$

$$\tilde{\alpha} = 1, \tag{30}$$

$$\beta^r = \sqrt{\frac{2M}{r}}. \tag{31}$$

To test the validity of the code, we compare the analytic solution of Einstein's equations above to the numerical solution of equations 1 to 6. In this special case of a spherically symmetric black hole, the solution is in a steady state. That is, the exact time derivative (RHS) of the solution is 0 in each component of the solution. The accuracy of the approximation to the solution depends on the degree of the Chebyshev polynomials and/or the precision of the floating point numbers used in the computation.

The arguments to the PDE solver include boundary points in space (e.g., [1.9, 11.9]), the initial and final times (e.g., [0, 10]), the size of the time step  $\Delta t$  (e.g.,  $\Delta t = 0.001$ ), and the degree of the Chebyshev polynomials  $N$  (e.g.,  $N = 30$ ). The size of the domain, the number of time steps, the size of each time step ( $\Delta t$ ), and the degree of the polynomials all influence the global error of the approximation. For a fixed domain and number of times steps, the error approaches 0 (up to machine precision) as we increase  $N$  and decrease  $\Delta t$ . Furthermore, as we increase  $N$ , we see spectral convergence; the error rapidly decreases to machine precision as  $N$  increases. This spectral convergence is how we validate our code against the known solution.

In Figure 1, we see the error in the solution computed using various values of  $N$  with  $\Delta t = 0.001$ . For the analytic solution  $u$  and approximate solution  $\hat{u}$ , Figure 1 has the value  $\|\mathbf{u}_r - \hat{\mathbf{u}}_r\|_2$  on the vertical axis. Recall,  $\mathbf{u}_r = \{u(r_0), \dots, u(r_N)\}^T$  for  $r_i = x_i \cdot ((r_{max} - r_{min})/2) + ((r_{max} + r_{min})/2)$ , where the  $x_i$  are the collocation points in the interval  $[-1, 1]$ .

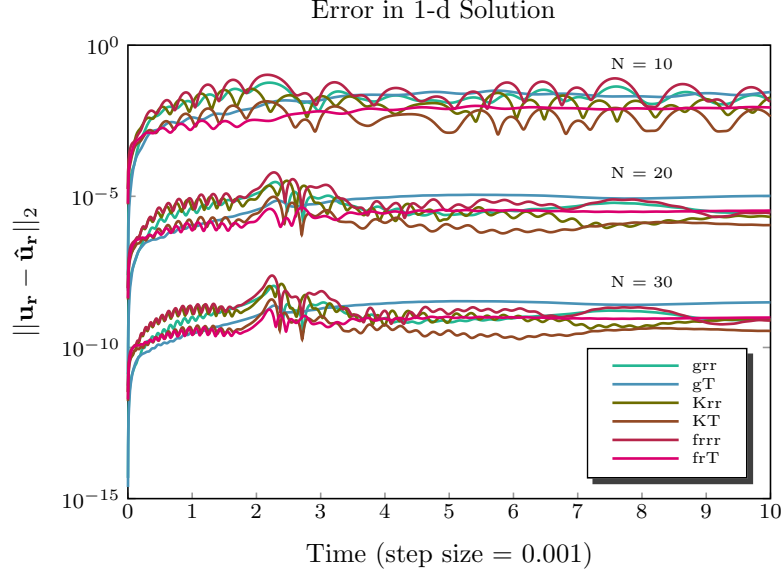


Figure 1: Error in the components of the numerical solution  $\hat{u}$  for multiple values of  $N$ .

For the three values of  $N$  shown in Figure 1, the errors differ by a few orders of magnitude. We stop at  $N = 30$  in the figure, but for  $N = 40$  the error improves as one would expect from the figure. This demonstrates the spectral convergence, and in turn, validates our code. Another important observation is that the error stays bounded as a function of time. In Figure 1, from time 4 to time 10, the error is approaching a steady state. With  $N = 10$  the solution converges to a steady state by time 1000 (i.e.,  $10^6$  steps) using double precision floating point.

## 4 Implementation

The initial prototype of the code is the MATLAB spectral solver. To compare between the CPU performance and the GPU performance we use C and CUDA code. Much of the C code written for the CPU implementation is reused in the GPU implementation. The host CPU code is nearly identical for the CPU and GPU solvers, but the computationally intensive RHS computations are implemented in a CUDA kernel that runs on the GPU.

Solving the full 2-d or 3-d system of Einstein's equations requires a significant amount of work that is beyond the scope of this project. While a functional high-performance numerical spectral solver is interesting, numerically solving the equations for a system of 1-d equations with a known analytic solution is not terribly interesting. Also, the 1-d case requires only a small amount of computation. In order to get some useful performance data from this project, we replicate the 1-d solution in the 2nd and 3rd spatial dimension. This gives us our “2-d” and “3-d” code, and enables us to study the relative performance of the CPU and GPU on data arrays that are the size and shape that we are interested in. Furthermore, we increase the

RHS computation for the 2-d and 3-d code. The 1-d formulation is roughly 200 floating point operations (flops) per collocation point. In a 3-d simulation of inspiral black holes, this is on the order of 10,000 flops per collocation point. In order to simulate the computational cost, we have a `for` loop around the RHS computation in the 2-d and 3-d RHS functions. In both cases, the RHS computation from the 1-d case is computed 50 times, which brings us close to 10,000 flops per collocation point.

## 4.1 Target Hardware

The C and CUDA code was written to compile and run on `hpd.cscamm.umd.edu`. This machine has two Quad-core Intel Xeon processors that operate at 2.67 GHz, an x86\_64 architecture, and 8 MB of shared L3 cache. These processors are the host processors for two Tesla GPUs with the Fermi architecture. Each GPU has 448 cores; 14 multiprocessors with 32 cores each. The cores operate at 1.15 GHz. The a peak performance for this GPU is 1.03 Tflops for single precision floating point and 515 Gflops for double precision floating point. The disparity between the performance in single and double precision should be contrasted with the analogous performance on the CPU. The single precision implementation is only about 10% faster than the double precision implementation on the CPU. With that said, the Fermi architecture is a significant improvement in double precision computation over previous generations of GPUs. In Section 5, we give performance results for CPU and GPU in both double and single precision, but we suspect that double precision will be required for any large-scale simulation of inspiral black holes.

## 4.2 Computational Organization

One of most important aspects of optimizing code is organizing the data in memory so it can be accessed efficiently. C is a row-oriented language, which means we want the 6 main variables (see equations 24 to 29) in the last dimension of the array. These are the values that will be accessed often when evaluating the RHS of equations 1 to 6.

In both the C and CUDA code, we allocate the large multi-dimensional arrays as one contiguous block of memory. This allows the Runge-Kutta function, which acts pointwise on each array, to be indifferent to the dimensionality of the problem. It only needs to know the number of elements in each array and the size of each element. In the 3-d code we have multiple arrays that are allocated for  $(N + 1)^3 \cdot 50 \cdot \text{sizeof}(\text{efloat})$  bytes. These arrays are treated as 1-d where the code can be made general enough to handle problems of different dimensions (e.g., the Runge-Kutta function). When it is convenient to index this as a 4-d array (e.g., the RHS function), we cast the array to `(efloat (*) [N+1] [N+1] [50])`. For example, if we cast the array `u`, then we can access `u[i] [j] [k] [p]`, for  $i, j, k \in \{0, \dots, N\}$  and  $p \in \{0, \dots, 49\}$ . Having the 50 components as the final index is good for the row-oriented CPU code. However, this is not what we want for the GPU.

In the CUDA code, we have a kernel that computes the RHS. The kernel is called using a grid that defines the number of blocks and the number of threads per block. For the GPU we were using a block can have at most 1024 threads (32 threads run at one time), and each block must be independent of the rest. The kernel uses multiple threads in a block to cooperate on the computation of the RHS for each collocation point. Ideally, contiguous memory should

be accessed across threads in a block. This means we want the  $N + 1$  collocation points to be contiguous in memory. This is why main variables are in the first index of the arrays in the GPU computations. The result is that the GPU code uses arrays that are indexed in the reverse order of the CPU code (i.e., `u[p][k][j][i]`), and the 4-d casting is now `(efloat(*)[N+1][N+1][N+1])`.

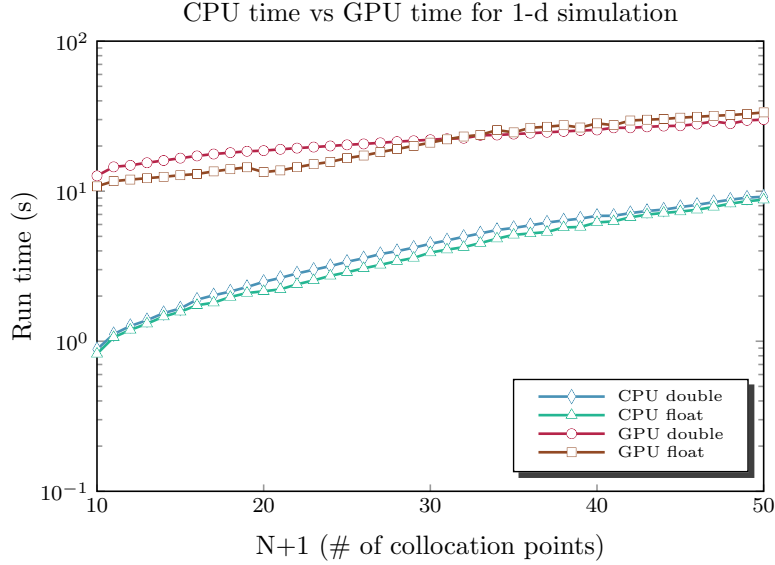


Figure 2: CPU run times vs. GPU run times for 1-d code with various  $N$ .

## 5 Results

The MATLAB prototype is not meant to be fast, but it does use matrix and vector operations instead of `for` loops the computations are column-oriented when possible. The MATLAB code takes about 13.2 seconds to compute 10,000 time steps of the 1-d case for 32 collocation points. With the CPU code, we were able to do the same computation in about 0.5 seconds (about 26 times faster).

In order to compare the (single core) CPU performance with the GPU performance, we recorded the run time of the CPU and GPU code for the 1-d, 2-d, and 3-d code, in both single and double precision floating point (float and double respectively). In all cases, we want to take enough time steps that the time setting up the problem is very small in comparison with the time spent evolving the equations in time. For the 1-d, 2-d, and 3-d code we took 100,000; 10,000; and 1000 steps respectively.

In Figure 2, we see times for the 1-d case. The GPU code is slower than the CPU code for both single and double precision. This is due to the fact that we can only use a small portion of the GPU for this code. Furthermore, there is not a lot of computation being done for



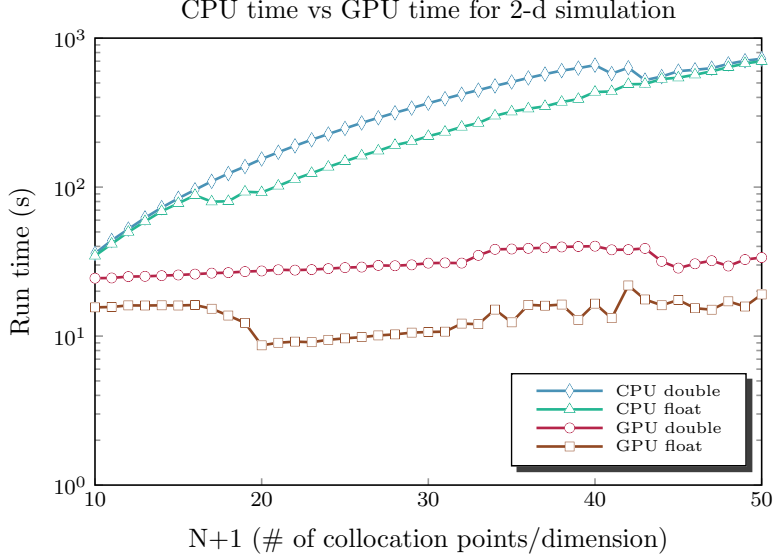


Figure 3: CPU run times vs. GPU run times for 2-d code with various  $N$ .

each collocation point. The overhead of setting up and calling the GPU kernel dominates the computation, and much of this overhead happens within the time step loop. Notice that the gap gets smaller as  $N$  increases since we are introducing more parallelism in the GPU as we increase  $N$ . We expect this trend to continue until the full GPU is busy.

The run times for the 2-d code are in Figure 3. We see that as  $N$  increases, we start to get significant gains over the CPU. As  $N$  increases we have more blocks for the GPU to work on. Still, notice that the GPU time does not increase much as we increase  $N$ . This is due to the fact that we are not using the full power of the GPU for smaller  $N$ . The timings in Figure 3 show some strange behaviors of the code, not all of which are understood. The CPU float timings get better around 17 on the  $x$ -axis, and similarly, the times improve around 41 on the  $x$ -axis for the CPU double timings. These times seem to be repeatable within a small relative error, and it is interesting to note the 17 and 41 are the points at which the error goes to 0 (i.e., falls below the respective machine precision) for several of the components of the solution. How or why this affects the run time is not clear.

Figure 4 shows the run times for the 3-d code. For the GPU, the single precision code is about twice as fast as the double precision code. For the CPU, the difference is roughly 10%. There is an interesting stair-step effect in the GPU run times. This is not surprising from the way we implemented the GPU kernel for the RHS computation. Since we are using a matrix vector multiplication  $\mathbf{D}\mathbf{u}$  to approximate  $\mathbf{u}'$ , we need to have the complete vector  $\mathbf{u}$  within each block of the kernel that is computing this derivative. The GPU RHS kernel performs both the RHS computations for each collocation point and this derivative calculation. The end result is that we must have a multiple of  $(N + 1)$  threads per block. It is plausible that for some values of  $N$  a greater percentage of the GPU is active than for other values of  $N$ . As more GPU

resources get allocated, the time increase is minimal for an incremental change in  $N$ , but when we increment  $N$  and obtain a poor allocation of the GPU resources, we see a sudden jump in the run time.

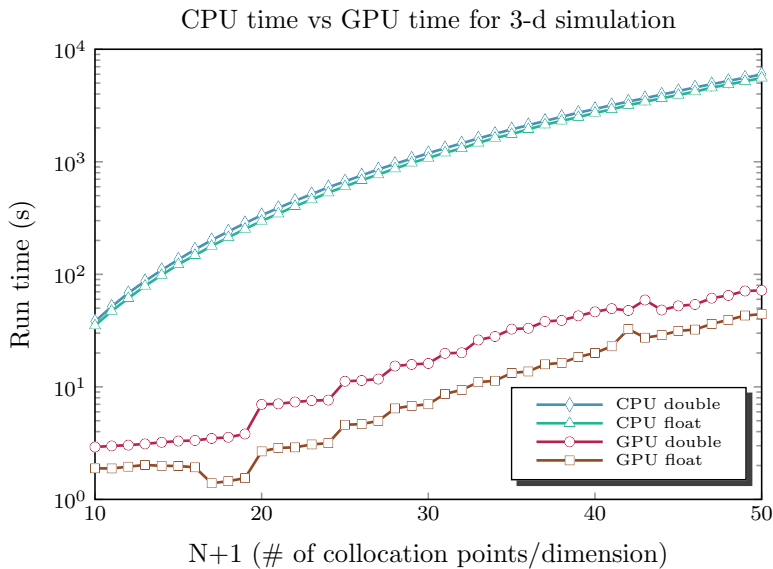


Figure 4: CPU run times vs. GPU run times for 3-d code with various  $N$ .

Figure 5 shows the performance gain of the GPU over the CPU for the 2-d and 3-d code for both single and double precision floating point. As expected, the GPU affords very large performance gains over a single core of a CPU. The double precision code runs about 80 times faster on the GPU than the CPU. Run time is an important factor, but comparing a single core to a full GPU is a little tricky using just run times. Perhaps a better comparison is the energy expended by each device to complete the full computation. According to [7], the Tesla GPU consumes 238 W, while one Quad-core Intel Xeon X5550 processor uses 95 W (see [8]). We make the assumption that the CPU and GPU are running at full power for the duration of the computations. If we do the arithmetic, we see that the GPU takes almost exactly 10 times the power as a single core of the CPU. On the right  $y$ -axis of Figure 5 we mark the ratio of the energy (in kWh) used by the CPU and GPU.<sup>1</sup> In [7], NVIDIA claims the Tesla “Delivers cluster performance at 1/10th the cost and 1/20th the power of CPU-only systems based on the latest quad core CPUs.” Our results seem to be in line with NVIDIA’s claims.

<sup>1</sup>Of course, the ratio is unitless, so the energy units are arbitrary. Perhaps J are a more standard scientific unit, but we use the standard billing unit of kWh to appeal to those concerned with the paying electric bill.

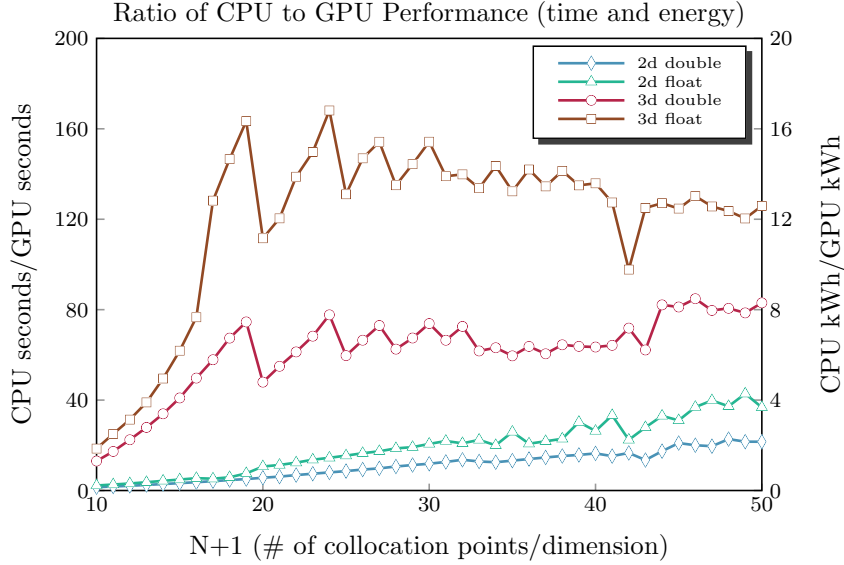


Figure 5: Run time and power performance ratio (CPU/GPU).

## 6 Conclusion

Based on the results for double precision, the GPU implementation is about 80 times faster than a single core of a CPU. What is probably more important is that the GPU implementation is about 8 times more energy efficient than a CPU implementation. This is a direct comparison of the GPU and CPU performance; there is no dependence the number of cores. A spectral method solver that is 8 times more energy efficient than current implementations would represent a significant cost savings. There is still much work to be done to realize these gains, and more work is required to show these gains can come to fruition for a large-scale simulation of inspiral black holes. Still, these results are encouraging and warrant further work in developing code to solve Einstein's equations in a more complex 3-d domain.

## References

- [1] NVIDIA. <http://www.nvidia.com/object/why-choose-tesla.html>. 2010.
- [2] NVIDIA CUDA C Programming Guide. Version 3.2. November 9, 2010.  
[http://developer.download.nvidia.com/compute/cuda/3.2-prod/toolkit/docs/CUDA\\_C\\_Programming\\_Guide.pdf](http://developer.download.nvidia.com/compute/cuda/3.2-prod/toolkit/docs/CUDA_C_Programming_Guide.pdf)
- [3] Gioel Calabrese, Luis Lehner, and Manuel Tiglio. “Constraint-preserving boundary conditions in numerical relativity.” *Phys. Rev. D* 65, 104031. May 10, 2002.
- [4] L. Trefethen. “Spectral Methods in MATLAB.” SIAM, 2000.
- [5] “Runge-Kutta methods.” [http://en.wikipedia.org/wiki/Runge-Kutta\\_methods](http://en.wikipedia.org/wiki/Runge-Kutta_methods).
- [6] Lawrence E. Kidder, Mark A. Scheel, Saul A. Teukolsky, Eric D. Carlson, and Gregory B. Cook. “Black hole evolution by spectral methods.” *Phys. Rev. D* 62, 084032. September 26, 2000.
- [7] “TESLA™ M2050/M2070 GPU COMPUTING MODULE.” <http://www.nvidia.com/tesla>.
- [8] “Intel® Xeon® Processor 5500 Series: An Intelligent Approach to IT Challenges.” [https://images01.insight.com/media/pdf/Intel\\_5500\\_Product\\_Brief\\_111209.pdf](https://images01.insight.com/media/pdf/Intel_5500_Product_Brief_111209.pdf).

## A Matlab Code

### cheby.m

```
function [D,x] = cheby(N)
% cheby  Chebyshev Differentiation Matrix and Grid
%
% Syntax:
% [D,x] = cheby(N)
%
% Summary:
% This function computes the Chebyshev Differentiation Matrix of degree N.
% The output matrix D is a N+1 x N+1 square differentiation matrix, and
% the Chebyshev grid x has is a vector of length N+1. This function gives
% the same output (up to rounding error) as cheb.m from [1].
%
% Input:
% N = Degree of the polynomials used to differentiate.
%
% Output:
% D = The differentiation matrix.
%
% x = The Chebyshev grid points.
%
% References:
% [1] Lloyd N. Trefethen. "Spectral Methods in Matlab." SIAM, 2000.
% http://www.comlab.ox.ac.uk/oucl/work/nick.trefethen.
%
% Author:
% Timothy D. Dewey (September 20, 2010)

% Define the grid points x.
x = cos(pi*(N:-1:0)'/N);

D = zeros(N+1);
c = [2; ones(N-1,1); 2];
for i = 1:N+1
    for j = [1:i-1,i+1:N+1]
        D(i,j) = c(i) * (-1)^(i-1+j-1) / (c(j) * (x(i) - x(j)));
    end
    % Use the opposite of the sum of off diagonals for better stability as
    % suggested in [1].
    D(i,i) = -sum(D(i,:));
end
```

## rhs.m

```
function u_dot = rhs(u, r, D)
% rhs Right hand side of Einstein's equations (Vectorized).
%
% Syntax:
%   u_dot = rhs(u, r, D)
%
% Summary:
%   Compute the right-hand side of Einstein's equations using for a
%   spherically symmetric black hole using Painleve-Gullstrand coordinates.
%
% Input:
%   u = [grr, gT, Krr, KT, frrr, frT] is an (N+1) x 6 matrix where (N+1) is
%   the of number collocation points used.
%
%   r = the collocation points in the scaled interval.
%
%   D = the differentiation matrix.
%
% Output:
%   u_dot = [grr_dot, gT_dot, Krr_dot, KT_dot, frrr_dot, frT_dot] is an
%   N x 6 matrix where N is the number collocation points used.
%
% Example:
%
%   rmin = 1.9;
%   rmax = 11.9;
%   t0 = 0;
%   t1 = 1.5;
%   num_steps = 1000;
%
%   norm_err = zeros(6, num_steps + 1);
%   N = 20;
%   [D, x] = cheby(N);
%   r = shiftInterval(x, rmin, rmax);
%   [grr, gT, Krr, KT, frrr, frT] = schwarzschild(r);
%   u0 = [grr, gT, Krr, KT, frrr, frT];
%   [y,t] = rk(@rhs, [t0 t1], u0, r, D, num_steps);
%   err = bsxfun(@minus, y, u0);
%   for i = 1:6
%       for j = 1:num_steps + 1
%           norm_err(i,j) = norm(err(:,i,j),2);
%       end
%   end
```

```

%      end
%
%      figure, semilogy(t, norm_err)
%
%
% References:
% [1] Lawrence E. Kidder, Mark A. Scheel, and Saul A. Teukolsky. "Black
%      hole evolution by spectral methods."
%      http://arxiv.org/pdf/gr-qc/0005056v1
%
% [2] G. Calabrese, L. Lehner, M. Tiglio. "Constraint-preserving boundary
%      conditions in numerical relativity." arXiv:gr-qc./0111003v1.
%      November 2001. http://arxiv.org/pdf/gr-qc/0111003v1
%
% Author:
% Tim Dewey, (November, 2010)

% Break out u components.
% u = ([grr, gT, Krr, KT, frrr, frT]);
grr = u(:,1);
gT = u(:,2);
Krr = u(:,3);
KT = u(:,4);
frrr = u(:,5);
frT = u(:,6);

alphaT = ones(size(u(:,1)));
dalphaT = zeros(size(u(:,1)));
d2alphaT = zeros(size(u(:,1)));

beta = ones(size(u(:,1))) .* sqrt(2./r);
dbeta = ones(size(u(:,1))) .* -1./sqrt(2.*r.^3);
d2beta = ones(size(u(:,1))) .* 3./sqrt(8.*r.^5);

% Derivative in space. d./dx .* dx./dr f -> d./dr f
dr = 2/(max(r) - min(r));

dgrr = D * grr * dr;
dgT = D * gT * dr;
dKrr = D * Krr * dr;
dKT = D * KT * dr;
dfrrr = D * frrr * dr;
dfrT = D * frT * dr;

```

```

grr_dot = beta .* dgrr + 2 .* grr .* dbeta - ...
          2 .* alphaT .* grr.^(0.5) .* gT .* Krr;

gT_dot = beta .* dgT + 2./r .* beta .* gT - ...
          2 .* alphaT .* grr.^(0.5) .* gT .* KT;

Krr_dot = beta .* dKrr - 1./grr.^(0.5) .* gT .* alphaT .* dfrrr + ...
           2 .* Krr .* dbeta - 8./grr.^(0.5) .* alphaT .* frrr .* frT - ...
           6 .* grr.^(0.5)./gT .* alphaT .* frT.^2 - ...
           1./grr.^(0.5) .* gT .* frrr .* dalphaT + ...
           2./grr.^(0.5) .* gT./r .* frrr .* alphaT - ...
           grr.^(0.5) .* gT .* d2alphaT + 4.*grr.^(0.5) .* gT./r .* dalphaT + ...
           2 .* grr.^(0.5) .* alphaT .* Krr .* KT - 1./grr.^(0.5) .* gT .* ...
           alphaT .* Krr.^2 - 6.*grr.^(0.5) .* gT .* alphaT./r.^2 + ...
           2./grr.^(1.5) .* gT .* alphaT .* frrr.^2;

KT_dot = beta .* dKT - 1./grr.^(0.5) .* alphaT .* gT .* dfrT + ...
           2./r .* beta .* KT - 1./grr.^(0.5) .* gT .* dalphaT .* frT - ...
           2./grr.^(0.5) .* alphaT .* frT.^2 + grr.^(0.5) .* gT .* ...
           alphaT./r.^2 + 1./grr.^(0.5) .* alphaT .* KT .* Krr .* gT;

frrr_dot = -alphaT .* grr.^(0.5) .* gT .* dKrr + ...
            beta .* dfrrr - alphaT./grr.^(0.5) .* Krr .* frrr .* gT - ...
            10 .* alphaT .* grr.^(0.5) .* Krr .* frT + ...
            12 .* grr.^(1.5)./gT .* frT .* alphaT .* KT + ...
            3 .* dbeta .* frrr - 4 .* grr.^(1.5) .* KT .* dalphaT + ...
            8 .* grr.^(1.5)./r .* KT .* alphaT + grr .* d2beta - ...
            4 .* grr.^(0.5) .* alphaT .* KT .* frrr - ...
            grr.^(0.5) .* gT .* Krr .* dalphaT + 2 .* grr.^(0.5) .* ...
            gT./r .* Krr .* alphaT;

frT_dot = -alphaT .* grr.^(0.5) .* gT .* dKT + ...
            beta .* dfrT + 2./r .* beta .* frT + dbeta .* frT - ...
            grr.^(0.5) .* gT .* KT .* dalphaT - alphaT./grr.^(0.5) .* ...
            KT .* frrr .* gT + 2 .* alphaT .* grr.^(0.5) .* KT .* frT;

u_dot = [grr_dot, gT_dot, Krr_dot, KT_dot, frrr_dot, frT_dot];

```