# GPUs and Einstein's Equations

Tim Dewey
Advisor: Dr. Manuel Tiglio

AMSC Scientific Computing–University of Maryland

May 5, 2011

# Outline

# Project Motivation

- LIGO is depending on reliable simulations to identify black hole detections in very noisy data
- Black hole simulations are computationally intensive
- GPUs are a reasonably flexible and efficient for large scale computations
- Using GPUs may reduce computation time and cost
- This project will focus on building up code that will be the groundwork for simulating black holes on GPUs

# Project Summary

Implement a spectral method PDE solver for Einstein's equations

1. Prototype solver in Matlab (Fall 2010)
2. Write and verify C code (Spring 2011)
3. Write and verify CUDA code (Spring 2011)
4. Compare CPU and GPU perfomance (Spring 2011)

# Einstein's Equations in 1-d

- Spherically symmetric black hole – coordinates are 1d in space instead of 3d in space
- Solve 6 coupled hyperbolic equations that are 1st order in space and time
- There are 6 variables $g_{rr}, g_T, K_{rr}, K_T, f_{rrr}, f_{rT}$ that describe a spherically symmetric metric on a Lorentzian manifold

# Building Blocks of Spectral Solver

## Chebyshev Collocation Points (Degree $N$)

$x_i = \cos(\pi \cdot i/N), \qquad i \in \{0, \ldots, N\},\ x_i \in [-1, 1]$

## Approximating the Spatial Derivative

$\mathbf{u} = \{u(x_0), \ldots, u(x_N)\}^T$
$\mathbf{D}$ is the $(N+1) \times (N+1)$ differentiation matrix relevant for the collocation points $x_i \in [-1, 1]$

$$\mathbf{u}' \approx \mathbf{D}\mathbf{u}$$

# Evolution in Time

## Fourth-Order Runge-Kutta (RK4)

$$y' = rhs(t, y), \qquad y(t_0) = y_0$$

$$t_{n+1} = t_n + h$$

$$k_1 = rhs(t_n, y_n)$$

$$k_2 = rhs(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}hk_1)$$

$$k_3 = rhs(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}hk_2)$$

$$k_4 = rhs(t_n + h, y_n + hk_3)$$

$$y_{n+1} = y_n + \tfrac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4)$$

# Boundary Conditions

- Event horizon of the black hole is at a radius of $2M$, where $M$ is the mass of the black hole
- Set inner boundary at 1.9
- Inner boundary is inside the black hole, so no boundary conditions need to be imposed explicitly
- After each step within RK4, adjust the outer boundary using the initial conditions

# RHS Computations

- Compute initial values of main variables

$$g_{rr}, g_T, K_{rr}, K_T, f_{rrr}, f_{rT}$$

- Use differentiation matrix to approximate the spacial derivatives (e.g., $g'_{rr} \approx \mathbf{D} g_{rr}$)
- For each collocation point $r_i$,

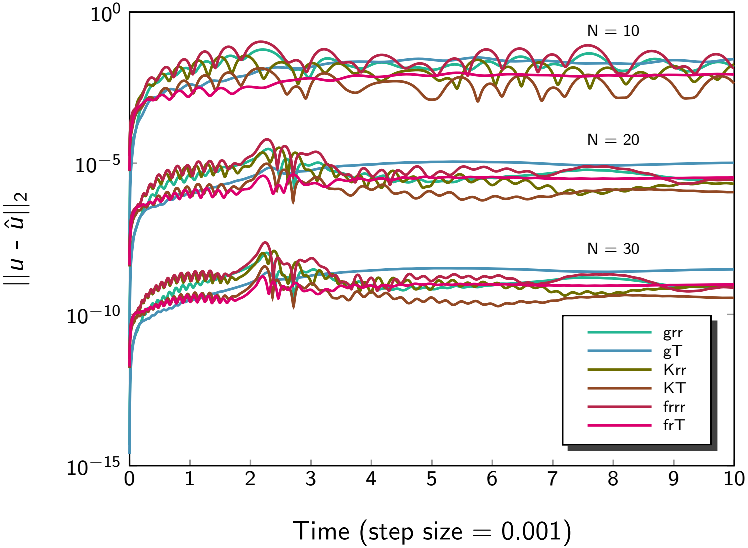
  Compute derivatives $\dot{g}_{rr}, \ldots, \dot{f}_{rT}$ at $r_i$, which depend on $g'_{rr}, \ldots, f'_{rT}$ at $r_i$
- For this special case, $\dot{g}_{rr}, \ldots, \dot{f}_{rT}$ should be 0

# Validating Numerical Solution

1. Choose a series of degrees $N$ of Chebyshev polynomials (e.g., $N \in \{10, 20, 30\}$)
2. Choose a fixed time step size (e.g. 0.001)
3. Evolve Einstien's equations $T$ time steps
4. At each time, determine the error of each component

   (e.g., for analytic solution $g_{rr}$ and approximation $\hat{g}_{rr}$, error $= ||g_{rr}(\mathbf{r}) - \hat{g}_{rr}(\mathbf{r})||_2$)
5. Verify that the error converges rapidly to 0 as $N$ increases

# Error in Solution to Einstein's Equations



# Implementation

- Wrote code for 1-d case in Matlab, C, and CUDA
- Left 2-d and 3-d cases as future work
- Replicated 1-d case in 2nd and 3rd dimension
- Added dummy components to simulate memory accesses:

  "3-d" code uses arrays that are
  $(N+1) \times (N+1) \times (N+1) \times 50$
- Increased computation to simulate RHS work:

  for loop around RHS (computes 1-d RHS 50 times)

# Usage and Options

Make : python make.py -g -d 3 -f

Usage : gpu_solver3 [options]

- -t tmax, tmax is final time
- -d dt, dt is the time step size
- -i r0, r0 is the inner boundary (default = 1.9)
- -o r1, r1 is the outer boundary (default = 11.9)
- -N deg, deg is the degree of the Chebyshev polynomial
- -a infile, infile is a file with a saved state
- -f filename, filename is name of the solution file to be written

# Outline of C/CUDA code

1. Initialize data structures and memory on CPU
2. Set initial conditions of PDE on CPU
3. Call GPU version of Runge-Kutta from CPU

## GPU Runge-Kutta

Allocate GPU memory and copy data to GPU

```
for t = 0:num_steps
   gpu_rhs <<< nBlocks, nThreads >>> ()
   gpu_update <<< nBlocks2, nThreads2 >>> ()
   ...  repeat 3x
```

4. Copy results to CPU
5. Write results to binary file

# Crash Course in GPU Computing

- Fermi GPU has 448 cores (at 1.147 GHz) on 14 multiprocessors
- Blocks run independently on a multiprocessor in warps of 32 threads (up to 1024 threads per block)
- CPU provides a "grid" that defines the number of blocks (nBlocks) and threads per block (nThreads)
- GPU executes a kernel called by the CPU

## GPU Kernel Call

```
gpu_rhs <<< nBlocks, nThreads >>> ()
```

- __syncthreads(), synchronizes threads across a block in a kernel

# Crash Course Continued

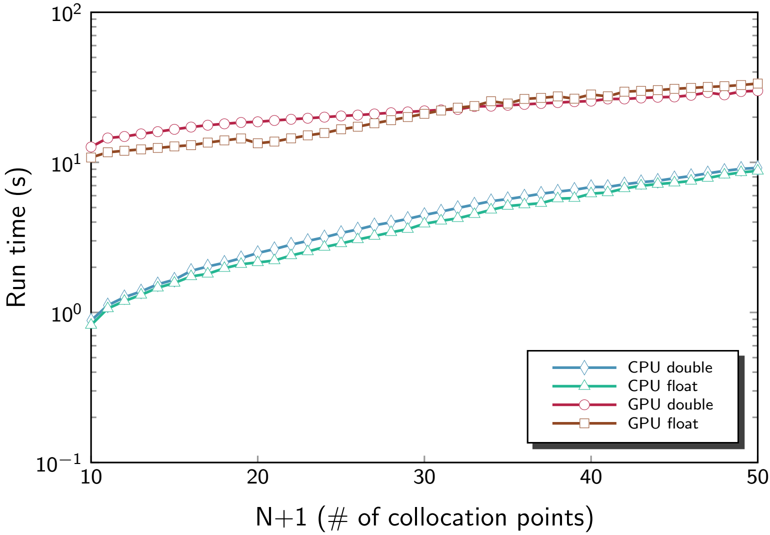
- Contiguous memory should be accessed across threads in a block
- Cannot have dependency across blocks (forces multiple of (N+1) threads per block with current kernel)
- Need a lot of parallelism to keep the GPU busy
- Do a lot of computation for each memory access

# CPU Info

What about the CPU?

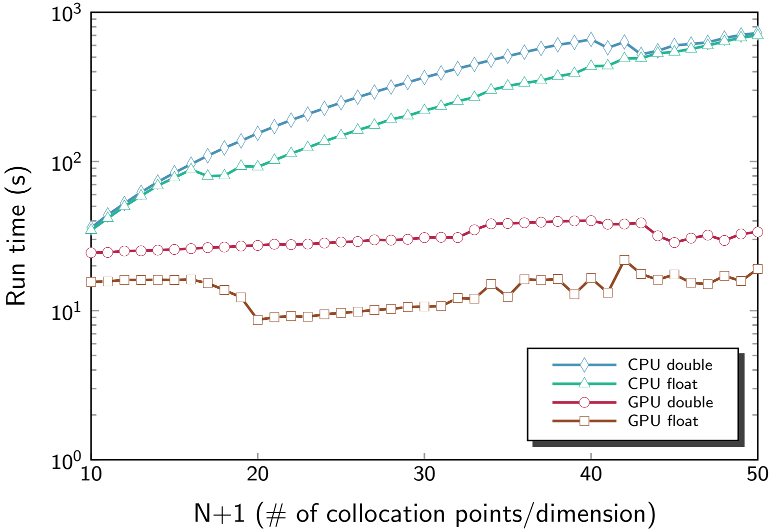
- Intel(R) Xeon(R) CPU X5550 @ 2.67GHz
- Cache size : 8192 KB
- 8 cores, but used only 1
- CPU code is not parallel code

# CPU vs GPU : 1-d



$10^2$
$10^1$
$10^0$
$10^{-1}$
Run time (s)
10
20
30
40
50
N+1 (# of collocation points)
CPU double
CPU float
GPU double
GPU float

# CPU vs GPU : 2-d

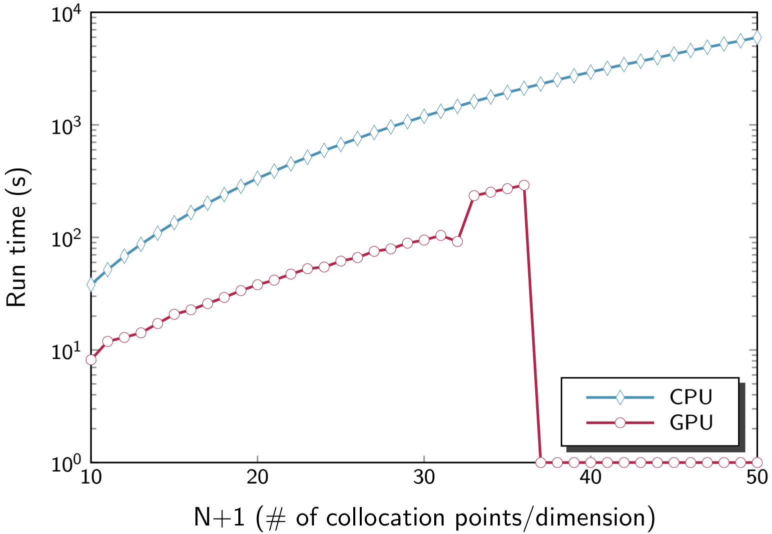


$10^3$
$10^2$
$10^1$
$10^0$
Run time (s)
10
20
30
40
50
N+1 (# of collocation points/dimension)
CPU double
CPU float
GPU double
GPU float

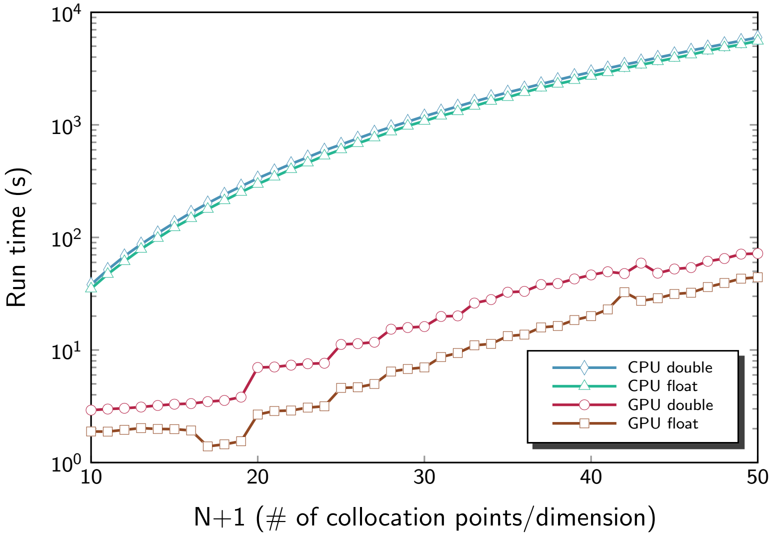
# One interesting limitation

- Recall $\mathbf{Du} \approx \mathbf{u}'$
- Full $\mathbf{u}$ vector is required to compute $\mathbf{u}'$
- GPU block must contain full $\mathbf{u}$ vector to compute $\mathbf{u}'$
- nThreads must be a multiple of $N + 1$
- 3-d case has $(N + 1)^3$ RHSs, so try nBlocks $= (N + 1)^2$ and nThreads $= (N + 1)$
- Need more threads per block

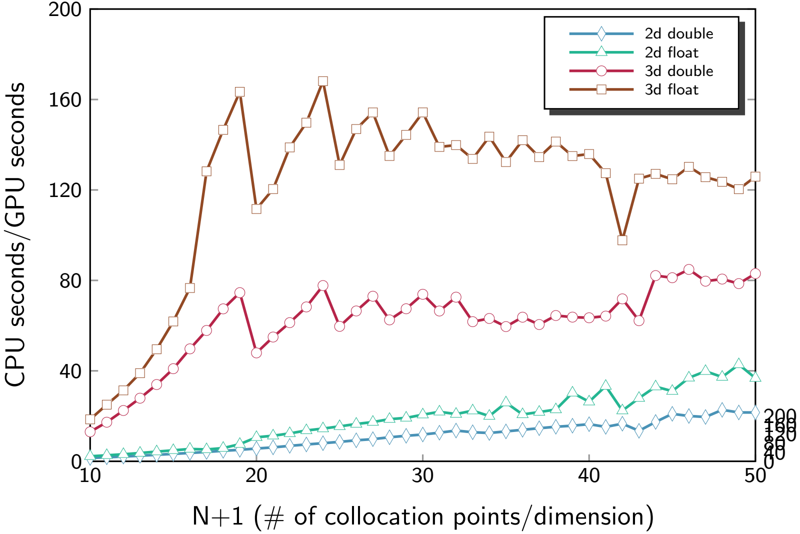
# CPU vs GPU : 3-d double precision



# CPU vs GPU : 3-d


Run time (s)
$10^4$
$10^3$
$10^2$
$10^1$
$10^0$
10
20
30
40
50
N+1 (# of collocation points/dimension)
CPU double
CPU float
GPU double
GPU float

# CPU runtime/GPU runtime



# Schedule

- $\surd$ February 10, 1-d C code verified on test data
- $\surd$ March 15, 1-d CUDA code verified on test data
- $\surd$ April 15, Optimized CUDA code
- *In progress :* May 1, Complete writeup and deliverables
- *Future work :* 2-d and 3-d versions, time permiting

# References


1 Lloyd N. Trefethen. "Spectral Methods in Matlab." SIAM, 2000. http://www.comlab.ox.ac.uk/oucl/work/nick.trefethen.

2 G. Calabrese, L. Lehner, M. Tiglio. "Constraint-preserving boundary conditions in numerical relativity." arXiv:gr-qc/0111003v1. November 2001.

3 Lawrence E. Kidder, Mark A. Scheel, and Saul A. Teukolsky. "Black hole evolution by spectral methods." http://arxiv.org/abs/gr-qc/0005056v1.

4 "Runge–Kutta methods." http://en.wikipedia.org/wiki/Runge-Kutta_methods."

5 "TESLA™ M2050/M2070 GPU COMPUTING MODULE." http://www.nvidia.com/tesla.