Scalable Relativistic High-Resolution Shock-Capturing for Heterogeneous Computing

Forrest Wolfgang Glines, Matthew Anderson, David Neilsen

1Department of Physics and Astronomy, Brigham Young University, Provo, UT, 84602
2Center for Research in Extreme Scale Technologies, Indiana University, Bloomington, IN, 47404

Abstract—A shift is underway in high performance computing (HPC) towards heterogeneous parallel architectures that emphasize medium and fine grain thread parallelism. Many scientific computing algorithms, including simple finite-differencing methods, have already been mapped to heterogeneous architectures with order-of-magnitude gains in performance as a result. Recent case studies examining high-resolution shock-capturing (HRSC) algorithms suggest that these finite-volume methods are good candidates for emerging heterogeneous architectures. HRSC methods form a key scientific kernel for compressible inviscid solvers that appear in astrophysics and engineering applications and tend to require enormous memory and computing resources. This work presents a case study of an HRSC method executed on a heterogeneous parallel architecture utilizing hundreds of GPU enabled nodes with remote direct memory access to the GPUs for a non-trivial shock application using the relativistic magnetohydrodynamics model.

Keywords—GPU, relativistic MHD, heterogeneous computing

I. INTRODUCTION

The end of Dennard scaling and the approaching end of Moore’s law is helping precipitate a shift in emerging high performance computing (HPC) architectures towards emphasizing medium and fine grain thread parallelism in the context of heterogeneous (e.g., GPU) parallel architectures. Otherwise unachievable efficiencies in both time, energy, and scalability can result by mapping algorithms originally designed for coarse grain execution to medium and fine grain execution for heterogeneous systems. This work examines just such an algorithm: high-resolution shock-capturing (HRSC) for relativistic magnetohydrodynamics (MHD).

HRSC methods form a key scientific kernel in compressible inviscid solvers that appear in astrophysics and engineering applications. While the mapping of HRSC methods to heterogeneous computing is still not fully developed, significant algorithmic advances have been made that enable speedups on the order of 10 times or larger by placing nearly all of the computations on the GPU [2]. At the same time, the development of GPUDirect has reduced latency using remote direct memory access (RDMA) to the GPU and consequently significantly improved performance of MPISendRecv on GPU memory [2]. These advances suggest that the mappings and technology necessary for a highly scalable relativistic fluid code designed for heterogeneous parallel architectures are now in place. Further, due to the medium grain nature of HRSC algorithms, heterogeneous computing appears to be the very ideal architecture for this type of simulation. This work reports on the development of a HRSC method for relativistic MHD that is scalable to hundreds of GPUs.

Relativistic MHD is a model that describes a neutral plasma of relativistic charged particles in the limit of infinite conductivity [2]. While any electric fields in the rest frame of the plasma are eliminated by the spontaneous flow of electric currents, these currents can sustain magnetic fields. The relativistic MHD equations are used to model plasmas in a wide variety of astrophysical systems, including neutron stars, accretion disks around black holes, and gamma-ray bursts.

New laser interferometer detectors are currently being constructed at several sites around the world to directly detect gravitational waves from mergers of compact objects (black holes or neutron stars). In the United States, the Advanced LIGO detectors are expected to begin observing gravitational wave in the next few years [2]. Interpreting data from these detectors requires an accurate expected waveform obtained from numerical simulations of mergers, which has in turn motivated significant research into numerical simulations of compact object mergers. These simulations solve the relativistic MHD equations coupled to the Einstein equations of general relativity for the gravitational field. Computationally, these simulations of merging compact objects (black holes or neutron stars) typically require adaptive mesh refinement (AMR) and 300–10,000 cores in a conventional parallel distributed architecture; the simulations generally take a few months of wallclock time. Thus to support the international effort to detect and observe gravitational waves, there is a particular relevance in accelerating these types of simulations beyond what conventional parallel architectures have provided.

While not the key focus of this case study, another motivation for exploring HRSC methods in heterogeneous parallel architectures is because of their energy efficiency. Previous studies on the Tesla K20x, the GPU used in this case study, have reported a factor greater than two improvement in energy efficiency compared with a multi-core CPU [2]. With Exascale computing energy requirements currently estimated at the unfeasibly high amount of 200 megawatts [2], energy efficient heterogeneous parallel architectures will likely figure prominently in the future high performance computing resources that will be running the HRSC algorithm.

This work is structured as follows. Related work is given in Section 2, followed by a description of the high-resolution shock-capturing algorithm and its GPU mapping in Section 3. Details on the MPI implementation for multiple GPUs as well

Keywords—GPU, relativistic MHD, heterogeneous computing

I. INTRODUCTION

The end of Dennard scaling and the approaching end of Moore’s law is helping precipitate a shift in emerging high performance computing (HPC) architectures towards emphasizing medium and fine grain thread parallelism in the context of heterogeneous (e.g., GPU) parallel architectures. Otherwise unachievable efficiencies in both time, energy, and scalability can result by mapping algorithms originally designed for coarse grain execution to medium and fine grain execution for heterogeneous systems. This work examines just such an algorithm: high-resolution shock-capturing (HRSC) for relativistic magnetohydrodynamics (MHD).

HRSC methods form a key scientific kernel in compressible inviscid solvers that appear in astrophysics and engineering applications. While the mapping of HRSC methods to heterogeneous computing is still not fully developed, significant algorithmic advances have been made that enable speedups on the order of 10 times or larger by placing nearly all of the computations on the GPU [2]. At the same time, the development of GPUDirect has reduced latency using remote direct memory access (RDMA) to the GPU and consequently significantly improved performance of MPISendRecv on GPU memory [2]. These advances suggest that the mappings and technology necessary for a highly scalable relativistic fluid code designed for heterogeneous parallel architectures are now in place. Further, due to the medium grain nature of HRSC algorithms, heterogeneous computing appears to be the very ideal architecture for this type of simulation. This work reports on the development of a HRSC method for relativistic MHD that is scalable to hundreds of GPUs.

Relativistic MHD is a model that describes a neutral plasma of relativistic charged particles in the limit of infinite conductivity [2]. While any electric fields in the rest frame of the plasma are eliminated by the spontaneous flow of electric currents, these currents can sustain magnetic fields. The relativistic MHD equations are used to model plasmas in a wide variety of astrophysical systems, including neutron stars, accretion disks around black holes, and gamma-ray bursts.

New laser interferometer detectors are currently being constructed at several sites around the world to directly detect gravitational waves from mergers of compact objects (black holes or neutron stars). In the United States, the Advanced LIGO detectors are expected to begin observing gravitational wave in the next few years [2]. Interpreting data from these detectors requires a library of expected gravitational waveforms obtained from numerical simulations of mergers, which in turn motivated significant research into numerical simulations of compact object mergers. These simulations solve the relativistic MHD equations coupled to the Einstein equations of general relativity for the gravitational field. Computationally, these simulations of merging compact objects (black holes or neutron stars) typically require adaptive mesh refinement (AMR) and 300–10,000 cores in a conventional parallel distributed architecture; the simulations generally take a few months of wallclock time. Thus to support the international effort to detect and observe gravitational waves, there is a particular relevance in accelerating these types of simulations beyond what conventional parallel architectures have provided.

While not the key focus of this case study, another motivation for exploring HRSC methods in heterogeneous parallel architectures is because of their energy efficiency. Previous studies on the Tesla K20x, the GPU used in this case study, have reported a factor greater than two improvement in energy efficiency compared with a multi-core CPU [2]. With Exascale computing energy requirements currently estimated at the unfeasibly high amount of 200 megawatts [2], energy efficient heterogeneous parallel architectures will likely figure prominently in the future high performance computing resources that will be running the HRSC algorithm.

This work is structured as follows. Related work is given in Section 2, followed by a description of the high-resolution shock-capturing algorithm and its GPU mapping in Section 3. Details on the MPI implementation for multiple GPUs as well
as the benchmark fluid tests are found in Section ??, while Section ?? presents the results of the study. Our conclusions and directions for future work are given in Section ??.

II. RELATED WORK

Several efforts at mapping HRSC methods to heterogeneous computing have been made, nearly all of which focus on Newtonian methods for very small GPU cluster configurations. Nevertheless, these studies have contributed enormously to this work and have strongly influenced the many attempts to broaden the impact of heterogeneous computing to new algorithms and methods. Wang et al. [?] provided the first fluid adaptive mesh refinement implementations for one to four GPUs, implementing the HLL (Harten-Lax-van Leer) Riemann solver and finding speedups of a factor of 10 over CPU equivalents. Zink [?] presented the first relativistic fluid solver for one or two GPUs, mapping the HLLE (Harten, Lax, van Leer, Einfeldt) solver to GPU computing and also found a factor of 10 improvement over a quad-core CPU simulation control case. Weak scaling tests in this result were limited to just two GPUs, although a strong case for an MPI implementation was made.

In addition to relativistic fluids, other numerical relativity research is currently being done using heterogeneous computing. Khanna et al. [?] examined extreme mass ratio inspiral gravitational wave sources using OpenCL on the Tesla GPU. Later, Khanna [?] studied higher than double-precision floating-point accuracy simulations for spinning black hole tails using CUDA. Choudhary et al. [?] explored the use of OpenCL for a hyperbolic partial differential equation using finite differencing. All of these studies, however, have been limited to very small numbers of GPU nodes and have examined benchmark problems that have low memory requirements. In contrast, this work reports on a case study that examines a highly scalable GPU-based HLLE implementation using a memory intensive benchmark problem.

III. MAPPING HRSC TO HETEROGENEOUS PARALLEL ARCHITECTURES

HRSC methods are key components in compressible inviscid fluid solvers and have some resemblance to the finite-difference methods that have already been successfully implemented on GPUs. This section reviews our HRSC method for relativistic MHD. This section concludes with a brief discussion on the time integrator and the mapping of the HRSC algorithm to the GPU.

A. HRSC methods

HRSC methods are based on Godunov’s method for conservation laws

\[
\frac{\partial u}{\partial t} + \sum_{k=1}^{3} \frac{\partial f^k(u)}{\partial x^k} = 0. \tag{1}
\]

(For a general discussion of HRSC methods, see e.g., Refs. [?], [?]). A basic outline of the method is as follows. The computational domain is divided into cells with a finite volume discretization, and the average value of each fluid variable is computed in the cell. A reconstruction procedure is done to give a higher-order representation of the fluid in the cell, and then the Riemann problem is solved at each interface between cells. The resulting solution is integrated to obtain new average values of the fluid variables in the cell.

Our numerical method is three-dimensional for generic fluid flow. However to simplify the discussion, this section reviews the method in only one dimension. The extension to three dimensions is done by simply summing the derivatives taken independently in each dimension. The fluid cells are uniformly distributed and centered at the locations \( x_i \equiv x_{\text{min}} + i \Delta x \). The solution is calculated at the discrete times \( t^n \equiv n \Delta t \). We adopt the shorthand notation that a function \( u^n_i \) represents the average value of the function \( u(x,t) \) in the cell centered at \( x_i \).

\[
u^n_i = \frac{1}{\Delta x} \int_{x_i-\Delta x/2}^{x_i+\Delta x/2} u(x,t) \, dx. \tag{2}
\]

We first discretize the spatial derivative in the conservation equation (??), giving the unsplit semi-discrete equations

\[
\frac{d u_i}{dt} = -\frac{\hat{f}_{i+1/2} - \hat{f}_{i-1/2}}{\Delta x}, \tag{3}
\]

Here \( \hat{f} \) is the numerical flux function that is calculated from a (usually) approximate solution of the Riemann problem at each cell interface.

The Riemann problem consists of two constant fluid states connected by a single discontinuity, which we will label the left state \( u_{i+1/2}^l \) and the right state \( u_{i+1/2}^r \). At the cell interface \( x_{i+1/2} \), we could choose the left and right states to be simply the average values \( u_{i+1/2}^l = u_i \) and \( u_{i+1/2}^r = u_{i+1} \). This results in a numerical method that is first order in \( \Delta x \). To achieve higher order methods (for smooth flow), we reconstruct the fluid variables using interpolation schemes that avoid oscillations near discontinuities. We use the Piecewise Parabolic Method (PPM) [?] to reconstruct the fluid states at the cell boundaries, and this reconstruction method is illustrated in Fig. ?? Finally, after reconstructing the fluid states, we use an approximate solution to the Riemann problem using the HLLE numerical flux

\[
\hat{f} = \lambda^+_r f(u^r) - \lambda^-_r f(u^l) + \lambda^+_r \lambda^-_l \left( u^r - u^l \right), \tag{4}
\]

where \( \lambda^+ \) and \( \lambda^- \) represent the largest characteristic speeds at the interface in the right and left directions, respectively, and \( f \) is the flux term in the conservation equation.

PPM gives a parabolic reconstruction of the fluid states at the cell boundaries. Reconstruction is inherently an interpolation procedure, which must be done carefully to avoid spurious oscillations at shocks in the fluid. This method begins by calculating a quartic interpolating polynomial at the cell boundary where for smooth functions this value reduces to the familiar fourth-order Lagrangian interpolation formula. We then set values for the functions on the left and right sides of the interface as

\[
u_{i+1/2}^l = u_{i+1/2}; \quad u_{i+1/2}^r = u_{i+1/2}. \tag{5}
\]

A set of monotonicity conditions (See [?], Eqn 1.10) are then applied to the interpolating function so that the interpolation parabola is monotone.
HRSC methods differ from finite difference methods in several key areas, including the reconstruction of the finite volume cell average. While HRSC methods are more complicated than simple finite difference methods, they share enough similarities to motivate a heterogeneous implementation of HRSC schemes. In order to demonstrate a heterogeneous implementation of the HRSC method, a set of equations must be chosen for the case study. Given their relevance to current research in astrophysics, we choose the relativistic MHD equations for this purpose.

B. Relativistic MHD equations

The heterogeneous implementation of the relativistic MHD equations described here will eventually be used in large scale simulations. The implementation has been designed, for example, to solve the fluid equations on a general, curved spacetime for extremely relativistic flows. It has also been designed to run on a large number of GPUs now and to incorporate AMR in the future.

The MHD fluid equations are conservation laws for mass (conservation of baryons), energy, and momentum. These equations are written in terms of two different complete sets of variables: the (1) \textit{conserved variables}, which are the conserved quantities (relativistic density \(D\), momentum \(\mathbf{S}\), and energy \(\tau\)) and the magnetic field \(\mathbf{B}\), \(\mathbf{u} = (D, \mathbf{S}, \tau, \mathbf{B})\); and the (2) \textit{primitive variables}, which are the rest-mass density \(\rho\), velocity \(\mathbf{v}\), pressure \(P\), and the magnetic field, \(\mathbf{v} = (\rho, \mathbf{v}, P, \mathbf{B})\). (The magnetic field is included in both sets of variables for convenience in working with curved spacetimes, where the conserved magnetic field includes an additional geometric term.) The bold symbols \(\mathbf{v}\), \(\mathbf{S}\), and \(\mathbf{B}\) are 3-dimensional vectors, giving a total of eight scalar functions in both sets of variables.

Given an equation of state, the primitive and conservative variables are each complete, meaning that both uniquely specify the state of the fluid. The transformation from primitive to conserved variables involves simple algebraic relationships, however, the inverse transformation does not have an analytic expression. The calculation of primitive variables from conserved variables requires solving a nonlinear, transcendental equation. Without a simple expression of \(v\) in terms of \(u\), both sets of variables are required in the MHD equations. For relativistic MHD, the solution of this nonlinear transcendental equation adds more floating point computations to each collocation point and decreases the communication to work ratio in a parallel implementation over its classical counterpart.

We solve the relativistic MHD equations in a flat (Minkowski) spacetime with Cartesian coordinates. With these assumptions the ideal MHD equations have a particularly simple form, and they can be written in conservation form as Eq. (??). Finally, we choose the ideal gas equation of state to close the system of equations.

C. Integration in Time

Numerically integrating differential equations using Runge-Kutta is well studied on heterogeneous architectures [7]. In this case study, a small variation of Runge-Kutta is used for integrating the resulting differential equations in time. The semi-discrete equations (??) are integrated with the optimal third-order Runge-Kutta scheme that preserves the total variance diminishing (TVD) condition:

\[
\mathbf{u}^{(1)} = \mathbf{u}^n + \Delta t L^{(1)}(\mathbf{u}^n),
\]

\[
\mathbf{u}^{(2)} = \frac{3}{4} \mathbf{u}^n + \frac{1}{4} \mathbf{u}^{(1)} + \frac{1}{4} \Delta t L(\mathbf{u}^{(1)}),
\]

\[
\mathbf{u}^{n+1} = \frac{1}{3} \mathbf{u}^n + \frac{2}{3} \mathbf{u}^{(2)} + \frac{2}{3} \Delta t L(\mathbf{u}^{(2)}).
\]

This variation from the standard third-order Runge-Kutta implementation does not alter the performance on the GPU.

After calculating each of the states \(\{\mathbf{u}^{(1)}, \mathbf{u}^{(2)}, \mathbf{u}^{(n+1)}\}\), the nonlinear transcendental equation referred to in Section ?? is solved to find the corresponding primitive variables \(\{v^{(1)}, v^{(2)}, v^{(n+1)}\}\). A GPU based Newton-Raphson solver is used to solve the transcendental equation.

The entire algorithm to evolve the fluid one complete step is summarized here

each Runge-Kutta stage: each direction \(\{x, y, z\}\) performs PPM reconstruction of \(v_{i+1/2}^{(1)}\), \(v_{i+1/2}^{(2)}\) calculate \(u_{i+1/2}^{(1)}, u_{i+1/2}^{(2)}\) compute fluxes \(f(u^{(1)}), f(u^{(2)})\) compute the speeds \(\lambda^{(1)}(u_{i+1/2})^{(1)}, \lambda^{(2)}(u_{i+1/2})^{(2)}\), compute the HLLE numerical flux \(f^{+}\) compute contribution to the right hand side of Eq. (??) compute update to Runge-Kutta \(\{u^{(1)}, u^{(2)}, u^{(n+1)}\}\),

communicate ghostzones (if a parallel simulation) apply boundary conditions solve for updated primitive variables

D. GPU Implementation

The GPU implementation of the HRSC method and the relativistic MHD equations was targeted for the Cray XK7, consisting of two 16-core AMD Opteron CPUs and one NVIDIA Kepler GPU with Cray’s Gemini network interconnect. This choice is motivated in part by the high floating point
operations per watt reported for this platform. It also reflects the trend in recent heterogeneous HPC systems including Titan at Oak Ridge National Laboratory, Blue Waters at the National Center for Supercomputing Applications at Illinois, and Todi at the Swiss National Supercomputing Centre. The mapping of the HRSC method and the relativistic MHD equations to this architecture, however, is easily generalized to other heterogeneous systems.

Following the approach of [?], the GPU implementation of the HRSC method avoids bus latency by placing nearly all of the computation on the GPU DRAM. NVIDIA’s CUDA was chosen for the implementation and all computations were performed using double precision. As noted in Table ??, the number of registers per thread is limited to 255, making smaller kernels (or simpler functions) more optimal so that the kernel code can fit multiple times in the registers. CUDA does have the ability to store instructions in DRAM, in what are called spill stores, but at a large drop in performance. We found execution times to be much slower with higher register counts per kernel and register counts of around 40 to be optimal for occupancy and execution time. This led us to split functions into many separate calls at the expense of loading data from DRAM more often. Data is stored in DRAM between function calls, as it is the only memory that is large enough and persists throughout the lifetime of the program. Variables that are used in multiple functions for the computation of the flux and eigenvalues, such as the sound speed and the inner product of the fluid velocity, are computed before hand and stored for each point. This also reduces function sizes and improves device occupancy at the trade off of more transactions with DRAM. Other variables such as the fluxes for the left and right hand sides, conservative, and primitive variables are also kept in DRAM. In total, 175 different variables are tracked; however, due to overflowing in memory, the total memory usage is 132 variables for each point. The maximum problem size that could fit on the Tesla K20, which has 5 GB DRAM, is a $160 \times 162 \times 160$ grid with 132 double variables at each point occupying about 4.4 GB. The K20’s constant memory, which is a section of DRAM optimized for simultaneous access to the same memory by multiple threads, is used for constants determined at run time such as the timestep size $\Delta t$.

The majority of optimizations to the code are centered on efficient memory usage. On the K20, both memory and arithmetic operations are executed simultaneously in blocks of 32. Operating on smaller chunks of data is allowed but the left over spaces are wasted. We optimize our use of the GPU by operating on blocks of 32 threads as often as possible, and thus we require the number of grid points in the $x$-dimension to be a multiple of 32.

Care is also taken to efficiently use the several tiers of memory on the card. Data is stored in the 5 GB of DRAM so that it will persist throughout the lifetime of the program. However, memory access to DRAM is relatively slow compared to the small 64 KB data cache on each multiprocessor, which is split between the explicitly managed shared memory cache and software managed L1 cache. Shared memory can occupy a maximum of 48 KB of this space. For functions that access points in DRAM multiple times but in different threads, data are loaded into the shared memory cache to minimize DRAM accesses. The data cache is shared amongst a block of at most 1024 threads. For the final computation, data are finally loaded into registers local to each thread. Both DRAM and the shared data cache are accessible in contiguous chunks of 32. Strided memory access and collisions between threads when accessing memory both result in serialized requests. To avoid this performance hit, memory accesses are done in contiguous blocks of 32 wherever possible.

Like finite-difference methods on GPUs, when computing the reconstruction of fluid variables and derivatives in HRSC methods, points in memory are used multiple times in different threads. Rather than load these points from global memory every time, they are first loaded into the shared memory cache. Threads within a block can then efficiently load points into the registers for computation. In order to meet these restrictions and maximize performance, a system of tiling used, which is based on a tiling scheme for finite difference derivatives [?]. Within a tile, data are first loaded from DRAM into the shared memory cache on the chip. The threads in the tile are then synchronized. Data are then loaded from the shared memory cache into the local registers for computation. Because data are arranged along the $x$-direction, tiling is different for derivatives in the $x$-direction than for the $y$- and $z$-directions, as shown in Figure ??.

For reconstruction and derivatives in the $x$-direction, blocks of threads span the width of domain and have a height subject to the 1024 thread limit and to the size of the shared cache. For example, a domain 128 points wide blocks would be $1024/128 = 8$ tall. The domain width is also fixed to be a multiple of 32 to ensure arithmetic and memory operations are in blocks of 32. These blocks are contiguous in the $x$-direction and so also in memory, allowing optimal access. Each point in global memory is only accessed once.

For reconstruction and derivatives in the $y$- and $z$-directions, blocks are 32 wide with a height that is constrained by the 1024 thread limit and the size of the shared cache. Because the tile in the $y$- (or $z$-) direction doesn’t span the entire domain, extra points on the borders of the tile have to be loaded in. These points get loaded from DRAM twice into two different tiles. An alternative would be for threads to compute more than one point along $y$ or $z$, surpassing the 1024 thread limit, so a block can span the entire $y$- or $z$-direction. However, for grid domains larger than 64, the data cache is often too small to fit the entire $y$- or $z$-dimension, limiting it to one point per thread. Because the block is 32 wide in the $x$-direction, memory accesses are contiguous blocks of 32.

As mentioned above, the HRSC method and relativistic MHD equation implementation is split across 96 kernels. Within the kernels themselves, apart from the detailed memory treatment described above, the code largely matches the CPU implementation thereby minimizing the code porting effort. The entire evolution of the fluid, with the exception of the MPI ghostzone communication, is performed on the GPU. This avoids the low bandwidth transfer of data to the CPU and host RAM. The only data transfer between the device and host is for output and to communicate grid ghostzones between nodes. The parallelization details and the performance benchmark test problem are discussed in the following section.
Fig. 2: This figure shows the tiling used for PPM reconstruction (requiring a five-point stencil) on a $128^3$ grid. The left panel shows the tiling for reconstruction in the $x$-direction, while the right panel shows the tiling for reconstruction in the $y$-direction. The gray region represents the points for which the reconstruction is computed and where data are loaded into the cache. The hatched region represents extra data off the edge of the tile that must be loaded into cache for computation of derivatives in the interior. These points will be loaded in cache twice, once for each of two different tiles. Tiling for the reconstruction in the $z$-direction is similar to the tiling in the $y$-direction.

TABLE I: Specifications for NVIDIA Tesla K20 GPU

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of processor cores</td>
<td>2496</td>
</tr>
<tr>
<td>Number of multiprocessors</td>
<td>13</td>
</tr>
<tr>
<td>Processor core clock</td>
<td>706 MHz</td>
</tr>
<tr>
<td>Memory clock</td>
<td>2.6 GHz</td>
</tr>
<tr>
<td>DRAM size</td>
<td>5 GB</td>
</tr>
<tr>
<td>Max Shared Memory Size per block</td>
<td>48 KB</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>208 GB/sec</td>
</tr>
<tr>
<td>Max CUDA blocks per kernel</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>Max CUDA threads per kernel</td>
<td>1,024</td>
</tr>
<tr>
<td>Max CUDA threads per multiprocessor</td>
<td>2,048</td>
</tr>
<tr>
<td>Max 32 bit registers per thread</td>
<td>255</td>
</tr>
<tr>
<td>Memory I/O</td>
<td>320-bit GDDR5</td>
</tr>
<tr>
<td>Memory configuration</td>
<td>20 pieces of 128M × 16 GDDR5 SDRAM</td>
</tr>
<tr>
<td>System interface</td>
<td>PCI Express Gen2 × 16</td>
</tr>
</tbody>
</table>

IV. MPI IMPLEMENTATION AND BENCHMARK

The MPI implementation follows a standard nonblocking, two-sided Isend/Irecv approach without a communication progress thread running in the background. After each RK step, ghostzones need to be communicated between separate grids. The HRSC method as described in Section ?? with PPM limiter requires three ghostzones. These ghostzones are exchanged in the standard way with the exception of grid corners and edges. For simplicity, the ghostzone exchange for these points are serialized in each of the $\{x, y, z\}$ directions.

Data are first prepared for communication in a buffer on the GPU. When using GPUDirect, data can be sent and received directly between the GPU DRAM using Isend/Irecv. If not using GPUDirect, the data must first be copied from the GPU onto the host before using Isend. While those requests are running, grids on the edges of the domain compute the outflow boundary conditions, which do not require data from other processors. Data are then received using Irecv directly into a buffer on the GPU if using GPUDirect or else into a buffer on the host. Once the data are received, the boundary points are imported from the buffer onto the grid. The code then waits for all communication requests to complete before moving onto the next dimension.

The MPI implementation is summarized in the following pseudo code: $x, y, z$ direction low side and high side prepare data in buffer on GPU not using GPUDirect copy buffer on GPU into buffer on host send buffer using MPI_ISend compute boundary conditions for boundary grids low side and high side wait for receive request to finish using MPI_Wait receive data into buffer using MPI_IRecv not using GPUDirect copy buffer from host onto GPU import data from buffer into border points low side and high side wait for send requests to complete using MPI_Wait

For the non-GPUDirect runs unpinned memory was used (allocated with malloc) as the buffer for ghostpoints on the CPU. CUDA provides a type of memory allocated on the host called pinned memory (allocated with cudaMallocHost) which allows optimized transfers of data from the GPU to CPU. However, pinned memory is much more expensive to allocate compared to unpinned memory and so only gives speed ups when the transfers between the GPU and CPU are on the order of gigabytes. While the transfer of ghostzones may not reach this level, pinned memory might make a difference for future runs when writing data from the GPU to disk. This will be included in future tests on optimization of the code.

The benchmark tests consist of simulating a Gaussian initial data profile for the rest-mass density, $\rho$, with a polytropic equation of state for the pressure, $P$,

$$\rho = \delta + A e^{-(r/\sigma)^2} , \quad P = \kappa \rho^\Gamma .$$

Here $r = \sqrt{x^2 + y^2 + z^2}$, $A$ and $\sigma$ are constants specifying the amplitude and width of the Gaussian, and $\delta$ represents an ‘atmosphere’ to keep the density non-zero. $\Gamma$ is the adiabatic index of the fluid, and $\kappa$ is a constant related to the entropy of the initial state of the fluid.

Evolutions are performed in geometric units with the speed of light, $c = 1$. These initial data are quite simple, but some of the computational challenges in evolving these data, e.g., evolving a fluid with large velocities in rarefied regions, are also encountered in evolutions of neutron star binaries. The fluid is evolved on a uniform domain in three dimensions.

TABLE II: Parameters for code tests

<table>
<thead>
<tr>
<th>Parameters</th>
<th>GPU Weak Scaling</th>
<th>GPU Strong Scaling</th>
<th>HAD Scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num. of points</td>
<td>$160 \times 162 \times 160^a$</td>
<td>$160 \times 162 \times 160^a$</td>
<td>$161^b$</td>
</tr>
<tr>
<td>$x, y, z \in$</td>
<td>$[-50, 50]$</td>
<td>$[-50, 50]$</td>
<td>$[-5, 5]$</td>
</tr>
<tr>
<td>$A$</td>
<td>100</td>
<td>10</td>
<td>$10^{-3}$</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\kappa$</td>
<td>1</td>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>$\Gamma$</td>
<td>4/3</td>
<td>4/3</td>
<td>4/3</td>
</tr>
<tr>
<td>$\delta$</td>
<td>$10^{-7}$</td>
<td>$10^{-7}$</td>
<td>$10^{-9}$</td>
</tr>
</tbody>
</table>

$a$ number of points per node. $b$ total number of points for all nodes.
for 150 timesteps with a fixed timestep size, $\Delta t$. The fluid initially has zero velocity, but during the benchmark runs the fluid speed quickly approaches the speed of light. Figure ? shows the density and velocity of the fluid along $x$ axis of the computational domain. The initial data parameters for all of the code tests are summarized in Table ??, while the parameter values are unfortunately not all identical, the average run time is not really affected by changes in the initial data parameters.

The final state of the fluid variables of the new GPU code was also compared to equivalent runs from the HAD code. The fluid variables differed by less than $10^{-14}$ in code units, consistent with floating point error.

![Graph](image)

Fig. 3: This figure shows a sample solution from the time evolution of a Gaussian profile at $t \approx 2$, plotted along the $x$-axis. The top frame shows the $x$-component of the velocity divided by the speed of light. The bottom frame shows the corresponding density profile.

V. RESULTS

We investigated the scalability of our heterogeneous HRSC method by conducting both weak and strong scaling tests of our GPU code. We also compared the performance of the GPU code to HAD [?], a mature research code for solving the relativistic MHD equations that is optimized for CPUs. While HAD is a generic AMR code for general relativistic astrophysics, it is was configured as closely as possible to match the algorithm used by the GPU code for these tests. Each test used Gaussian initial data with parameters given in Table ??, while the timings reported in Figures ??–?? have been averaged over five runs with error bars calculated at $2\sigma$. The HRSC implementation is highly scalable to hundreds of GPU nodes and shows a considerable speedup over the optimized CPU implementation.

A weak scaling test using a local grid size of $160 \times 162 \times 160$ grid for each GPU node is shown in Figure ??, used the largest problem size that could fit on a single GPU node. While the performance improvement asymptoted at slightly more than 40 GPU nodes, the real speedup was greater than an order of magnitude. For runs with 40 or more GPUs, the communication overhead overshadows the return of distributing the work. The ratio of the size of the boundaries that are transferred through communication to the size of the grid per node grows, meaning a larger percentage of time is spent in communication rather than computation. Beyond some node count, the grids on each node become small enough that all points can be computed nearly simultaneously on the GPU’s multiprocessors. At this point, smaller grid sizes would only cause cores on the GPU to be wasted for the iteration. While an improved parallelization scheme would be able to optimize further and overcome some of these obstacles, the parallel efficiency for such unigrid simulations is usually already high enough that implementing a radically different scheme is not justified.

Figure ?? also shows the impact of using RDMA for the GPU (GPUDirect) in conjunction with MPI calls. GPUDirect slightly outperforms the control case until reaching 100 GPU nodes where there is a noticeable negative impact. This negative impact occurs where the strong scaling performance has largely asymptoted and where the slightly higher latencies in the control case could be mitigating some resource contention. In general, the use of RDMA is expected to be very important to heterogeneous implementations of the HRSC method on large numbers of processors in production simulations.

Figure ?? compares the CPU-based HAD implementation [?] of the HRSC method with the heterogeneous implementation explored here. The comparison is a weak scaling test using one to four nodes of a Cray XE6 (32 cores/node) with one to four nodes of a Cray XK7 (16 cores and 1 Kepler GPU / node). The GPU implementation easily outperforms the optimized CPU implementation by over a factor of two, although this is much less than the order of magnitude improvements reported by Wang et al. [?] and Zink [?]. This may be due, in part, to the very small CPU core counts that were available for comparison testing in that research.

The average occupancy of the GPU implementation was calculated to be 55%. This was done by taking the approximate occupancy of each kernel as measured by the NVIDIA command line profiler for a single node run and weighing against the percent of the total execution time each kernel took. This means that on average, the multiprocessors on the GPUs were evaluating instructions at 55% if their full capacities. Occupancies higher than 50% are often no better or can even degrade overall performance. In order to raise occupancy, kernels would have to be split into small kernels (to reduce register usage per kernel) or shrink the blocks of threads that share cache memory. Both options can increase the number of memory access and the work load for the multiprocessors. Although the occupancy may increase the total execution time may still be worse. Occupancy around 50% balances memory latency with processor usage.

VI. CONCLUSIONS

This work has presented a highly scalable heterogeneous implementation of the HLLE HRSC method using the relativistic MHD equations as a case study. Using NVIDIA’s CUDA,
strong and weak scaling simulations of a relativistic outflow were presented with comparison against an optimized CPU implementation. A detailed CUDA-agnostic memory layout for the algorithm was provided to facilitate reproduction of the implementation. The heterogeneous implementation outperformed the optimized CPU version by over a factor of two. A nontrivial benchmark showing scalability to hundreds of GPUs was also presented. We conclude that heterogeneous approach presented is viable for full-scale, end-science production runs.

In the future we plan to augment this heterogeneous approach with adaptive mesh refinement and active load balancing, such that the code can efficiently refine interesting features of the fluid evolution and still take advantage of all allocated resources. Relativistic gravitational interactions will also be added in the future so that a larger variety of problems can be simulated.

Some kernels in the code are not optimal and, thus, do not utilize the entire GPU. Part of future work will include improving these kernels while still ensuring that the occupancy is balanced with latency from memory transactions. Occupancy for specific functions can be increased by reducing the number of registers the function uses through simplifying functions, removing local variables, and shrinking the sizes of thread blocks.

The current parallel implementation is fairly optimal for this regular grid, but that will not be the case for an AMR simulation. Future work will investigate overlapping more phases of computation. For example, the computation of primitive variables on the interior grid could be overlapped with communication of ghostzones between nodes.

ACKNOWLEDGMENT

It is a pleasure to thank our long-term collaborators Eric Hirschmann, Luis Lehner, Steven L. Liebling, Patrick Motl, and Carlos Palenzuela, with whom we have had many discussions on HRSC methods for relativistic MHD. We acknowledge many discussions with Eric Holk on heterogeneous implementations. This research was supported by the NSF under the grant PHY-1308727 to Brigham Young University, and a grant from the BYU Office of Research & Creative Activities. Research on Big Red II was supported in part by Lilly Endowment, Inc., through its support for the Indiana University Pervasive Technology Institute, and in part by the Indiana METACyt Initiative.
Fig. 6: This figure compares the performance for the HAD code (optimized for CPUs) and our code optimized for GPUs. The plot shows the average execution time per iteration in a weak scaling test on four nodes. The HAD code was run on CPU-only compute nodes of Indiana University’s Cray XE6/XK7 (Big Red II) with dual AMD Opteron 16-core Abu Dhabi x86_64 CPUs, or 32 CPU cores per node. The GPU code ran on CPU/GPU compute nodes of Big Red II, with one NVIDIA Tesla K20 GPU accelerator per node. Times are calculated from the average run time of five runs, each with 150 iterations and $160 \times 162 \times 160$ points per node. Error bars are calculated as $2\sigma$. 