Relativistic Magnetohydrodynamics on Graphics Processing Units

Forrest Wolfgang Glines

A senior thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Bachelor of Science

David Neilsen, Advisor

Department of Physics and Astronomy

Brigham Young University

April 2016

ABSTRACT

Relativistic Magnetohydrodynamics on Graphics Processing Units

Forrest Wolfgang Glines
Department of Physics and Astronomy, BYU
Bachelor of Science

Simulating binary star mergers in full relativistic magnetohydrodynamics with general relativity is computationally expensive, with production level simulations taking up to two months using traditional algorithms. These speeds are insufficient to explore the parameter space of binary star mergers. Following recent trends in chip manufacture, CPU speeds are unlikely to increase and speed up simulation times. In order to shorten simulation times new algorithms that take advantage of newer, faster computing architectures such as GPUs are required. This thesis presents GMHD, a relativistic magnetohydrodynamics code that runs on NVIDIA GPUs faster than other codes on CPUs. It implements a high-resolution shock-capturing algorithm using the piecewise-parabolic method and a total variation bounded method based on the Osher-Chakrabarthy method. The accuracy of the fluid methods are tested simulating the shock tube problem and the Kelvin-Helmholtz instability. Both methods accurately mode solution. This thesis also presents tests demonstrating the weak and strong scalability of the code tests to hundreds of GPUs. GMHD shows the viability and usefulness of using GPUs and forms a basis for future work on large scale simulations of binary mergers.

Keywords: RMHD, GPU,binary mergers

ACKNOWLEDGMENTS

I would like to thank my advisor, David Neilsen, for his help throughout the project. This research was supported by the NSF under the grant PHY-1308727 to Brigham Young University, by NASA under the grant NNX13AH01G, 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.

Contents

Chapter 1

Introduction

1.1 Neutron Star Mergers

Neutron stars are dense dead stars. They are the remnants of supernovae explosions after main-sequence stars with masses between 8-25 $M_\odot$ have exhausted their nuclear fuel [1]. During the supernova explosion, the core of the star is compressed to a radius of 10–15 km while still retaining up to 2.9 $M_\odot$, its density limited only by the neutron degeneracy pressure [2]. The star can be rapidly spinning and may also be extremely magnetized, making them energetic in every way.

The merger of two neutron stars is likely to be very hot and energetic. The energies and masses involved will usually lead to the formation of a black hole [3]. The merger and subsequent formation of a black hole that cuts off further radiation would appear as a quick, high intensity burst of light observable by space telescopes. These properties of the burst make neutron star merger prime candidates for the progenitors of short-hard gamma ray bursts (SGRBs).

In recent years space telescopes such as Swift and Fermi have observed SGRBs constantly occurring throughout the universe. Gamma-ray bursts are divided into two different types: short bursts, which last less than two seconds; and longs bursts, which are observed for longer than two

seconds. While it is accepted that longer and lower energy gamma ray bursts are probably caused by supernovae, the progenitors of these short-hard gamma ray bursts (SGRBs) are unconfirmed. The leading candidates are binary mergers of compact objects, either a neutron star binary merger or a black hole-neutron star merger. However, the high energy of these mergers and the quick formation of a black hole to end the flash make neutron star mergers probable progenitors of SGRBs.

The binary merger model for SGRBs can be explored through simulations. The evolution of the mergers lack analytical solutions, but numerical fluid simulations can model these systems. If SGRB-like radiation profiles can be calculated in computer simulations, then it would justify these mergers as SGRB progenitors. However, these simulations would require complex fluid models to capture all essential characteristics of the merger.

1.2 Relativistic Magnetohydrodynamics Simulations

Neutron stars can be modeled as a fluid of relativistic particles, making relativistic magnetohydrodynamics (RMHD) the ideal model. Because neutron stars are extremely dense and magnetized, they need to be modeled using RMHD. Due to the masses involved and possible formation of a black hole, simulations of neutron star mergers require general relativity for the gravitational model. Neutron stars are typically magnetized with magnetic field strengths on that vary between 10^4–10^8 G , the magnetic fields must be integrated in time with the fluid variables such as density and pressure. This requires the use of a magnetohydrodynamics fluid simulation. With these pieces together, the most basic simulation of a neutron star merger would be a relativistic magnetohydrodynamics simulation. Even without the addition of extra physical effects, such as neutrino radiation, radiation, and nuclear chemistry effects, the code has a heavy workload with just RMHD. Production level runs of binary mergers may take up to two months, which makes it

difficult explore many scenarios. For example, production simulations of binary mergers using the RMHD code HAD can take two months [4]. Faster codes are needed to more thoroughly study the parameter space of neutron star mergers.

The fluid equations can be solved in either a Lagrangian or Eulerian formulation. We use the Eulerian form, where the fluid equations are solved on a fixed coordinate system. A grid is first chosen to span the domain, and fluid variables such as pressure, density, and fluid velocity are initialized and tracked at each point. The relevant fluid equations are then evolved using typical numerical methods such as finite difference, finite volume, or pseudospectral methods. The formulation of the equations and the numerical methods are chosen based on the needs of the problem.

1.3 Graphical Processing Units

Due to increasing physical limitations on chip manufacture, traditional computing speeds will likely plateau in coming years. Central processing unit (CPU) clock rates, the number of operations a CPU can do in a second, have stagnated in the past decade because of power constraints. Past trends such as Moore's law, the doubling of transistor density every two years for the past 50 years, will likely end as the feature sizes of chips approach lengths of only a few atoms wide. As die sizes become smaller, these problems will become insurmountable, making future gains for traditional CPUs either too expensive or impossible. Instead, chip manufacturers increasing processor capability by creating dedicated many cored supplements to the CPU called coprocessors. These coprocessors save cost and power by packaging hundreds of parallel cores together.

Graphical Processing Units (GPUs) provide more processing power at a lower cost, higher density, and lower power consumption. GPUs are special coprocessors designed for graphics processing with hundreds more cores than traditional CPUs. Although they were originally made for

graphics, the large number of floating point cores makes them well suited for scientific computing. Recently the GPU manufacturers developed GPU like coprocessors specifically designed for high performance computing, with larger core counts and memory storage.

New supercomputers will use GPUs or other co-processors extensively. Titan at Oak Ridge National Lab, one of the largest supercomputers in the US for academic use, uses GPUs for a large portion of it's computer power. US initiatives for future supercomputers, such as the CORAL Collaboration, plan even larger machines using NVIDIA GPUs or Intel Xeon Phis. Moreover, these initiatives also include substantial efforts to migrate existing codes onto GPUs.

GPU programming can be more difficult compared to traditional programming due to limitations of the GPUs. As will be discussed, GPUs have very specific constraints on the parallelization of the algorithm to be effective. They also have optimal memory access patterns which require careful programming to utilize efficiently. Poor implementations can lead to nearly serial execution, leading to idle cores on the GPU and wasted processing power. While GPUs have more raw processing power, their limitations can make some algorithms nearly impossible to implement on GPUs. Fortunately, finite-difference-like methods like those used in this paper work well with the constraints of GPUs.

1.4 Overview

This paper will present the algorithm, performance, and results of the relativistic magnetohydrodynamics code GMHD implemented on GPUs [5]. Chapter 2 will discuss the RMHD fluid equations, the methods used to solve them, and their implementation on GPUs. In particular, Sections 2.2 and 2.3 present a Godunov type method and an Osher-Chakravarthy method for solving the RMHD fluid equations. Then in Section 2.4 the implementation of these methods will detailed. Chapter 3 presents results for fluid instability tests and scalability tests for the code. Specifically, Section

3.2 will present simulation results for the Riemann problem for both methods. Finally, Sections 3.5-3.7 present results of scaling tests and compare the performance of GMHD versus a traditional CPU code.

Chapter 2

Numerical Methods

This chapter presents the RMHD fluid equations, the fluid methods we use to solve them, and their implementation on GPUs. First the RMHD conservation equations and the variables involved will be described in Section 2.1. The two methods used to evolve the fluid equations will then be presented, the Godunov method in Section 2.2 and the Osher-Chakravarthy method in Section 2.3. Lastly, these methods and their implementation on GPUs are discussed in Section 2.4 and the communication strategy parallel execution is presented in Section 2.5.

2.1 Fluid Equations

The RMHD fluid equations express the conservation of mass, energy, and momentum for a magnetized relativistic fluid with perfect conductivity. Derivations of the RMHD equations are abundant in the literature, so we will only present the equations here. A more detailed presentation can be found in [6].

The fluid equations can be written as a conservation law as

$$\partial_t \mathbf{u} + \partial_k f^k(\mathbf{u}) = 0, \tag{2.1}$$

where $\mathbf{u}$ is the state vector consisting of the conserved variables and f^k is the flux in the k direction. Here ∂_t and ∂_k denote time and spatial partial derivatives respectively.

In our formulation of the RMHD equations, the conserved variables are the relativistic density D, the momentum in each coordinate direction S_i, the magnetic field in each direction B^k, and the energy τ. These are defined in terms of another set of variables called the primitive variables: the rest mass density ρ_0, the coordinate velocities v^i, and the pressure P. The magnetic field is included in both sets of conserved and primitive variables. The conserved variables are then defined, using the Einstein index convention, as

$$\begin{aligned} D &= W\rho_0 \\ S_i &= (h_e W^2 + B^2)v_i - (B^j v_j)B_i \\ \tau &= h_e W^2 + B^2 - P - \frac{1}{2}\left[(B^i v_j)^2 + \frac{B^2}{W^2}\right] - W\rho_0. \end{aligned} \tag{2.2}$$

Here $B^2 \equiv B_i B^j$ and the Lorentz factor W and fluid enthalpy h_e are defined as

$$\begin{aligned} W &= \frac{1}{1 - v_i v^i} \\ h_e &= \rho_0(1+\varepsilon) + P, \end{aligned} \tag{2.3}$$

and ε is the specific internal energy.

The fluid equations in flay space and Cartesian coordinates can be written in conservation form

as

$$\begin{aligned}
\partial_t D + \partial_i(Dv^i) &= 0 \\
\partial_t \tau + \partial_i(S^i - Dv^i) &= 0 \\
\partial_t S_x + \partial_i((\perp T)^i_x) &= 0 \\
\partial_t S_y + \partial_i((\perp T)^i_y) &= 0 \\
\partial_t S_z + \partial_i((\perp T)^i_z) &= 0 \\
\partial_t B^y + \partial_i(B^x v^i - B^i v^x) &= 0 \\
\partial_t B^x + \partial_i(B^y v^i - B^i v^y) &= 0 \\
\partial_t B^z + \partial_i(B^z v^i - B^i v^z) &= 0.
\end{aligned} \tag{2.4}$$

The projected stress tensor $(\perp T)^i_b$ is

$$(\perp T)^i_b = v^i S_b + P \cdot h^i_b - \frac{1}{W^2}\left[B^i B_b - \frac{1}{2} h^i_b \cdot B^2\right] - (Bv)\left[B^i v_b - \frac{1}{2} h^i_b \cdot (Bv)\right],$$

where h^i_b denotes the spacetime metric.

The fluid equations are first discretized using the method of lines. The spatial derivatives of the flux functions are calculated using high-resolution shock-capturing methods. The resulting set of ODEs are integrated in time using a 3rd order Runge-Kutta (RK3)integrator [7]:

$$\begin{aligned}
\mathbf{u}^* &= \mathbf{u}^n + \Delta t(\partial_t \mathbf{u}) \\
\mathbf{u}^{**} &= \frac{3}{4}\mathbf{u}^n + \frac{1}{4}\left[\mathbf{u}^* \Delta t(\partial_t \mathbf{u}^*)\right] \\
\mathbf{u}^{n+1} &= \frac{1}{3}\mathbf{u}^n + \frac{2}{3}\left[\mathbf{u}^{**} \Delta t(\partial_t \mathbf{u}^{**})\right].
\end{aligned} \tag{2.5}$$

Where u^* and u^{**} are temporary variables which can be be overlaid physically in memory, leading to efficient memory usage for this RK3 integrator.

As mentioned above, high-resolution shock-capturing (HRSC) methods are use to compute the spatial derivative of $\mathbf{f}(\mathbf{u})$. Simple finite difference approximations of the derivative will often

produce spurious oscillations around shocks, which are discontinuous solutions of the fluid equations [8]. Simple methods may also diffuse and dampen important features in the fluid evolution [9]. The two methods used here, a finite volume and a finite difference scheme, suppress these oscillations while still accurately modeling the shock.

Both methods place conditions on the total variation $TV(\mathbf{u})$, which is defined on a one dimensional grid as

$$TV(u) = \sum_j |u_j - u_{j-1}|,$$

where the sum over j is over all of space. The total variation is a simple measure of the variance between adjacent points. Regulating $TV(\mathbf{u})$, as many fluid methods do, can help regulate spurious oscillations that form. Fig. 2.1 demonstrates different conditions on regulating $TV(\mathbf{u})$. For example, Godunov based methods are total variation diminishing (TVD) methods. They guarantee that $TV(\mathbf{u})$ will never increase but can only decrease, or that

$$\frac{d}{dt}TV(\mathbf{u}) \leq 0.$$

This automatically kills any spurious oscillations but at the cost of accuracy. Osher-Chakravarthy methods are total variation bounded (TVB) methods. They loosen the condition to guarantee that $TV(\mathbf{u})$ has an upper bound b which may depend on time.

$$TV(\mathbf{u}(t)) \leq b(t).$$

The looser condition allows higher order methods with greater computational efficiency compared to TVD methods. TVD methods suppress most spurious oscillations, but some oscillations can still blow up. These methods and their advantageous will be presented below.

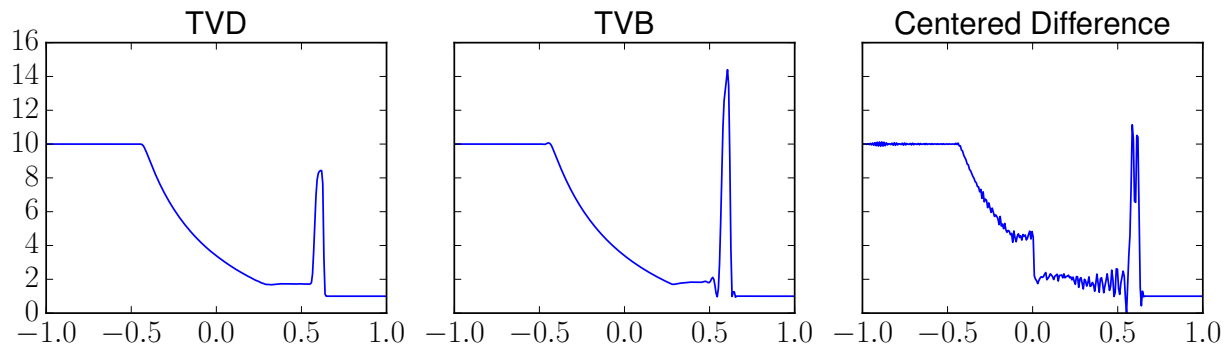


Figure 2.1 Here the different conditions on the total variation for each fluid method are demonstrated. All figures show the density of the evolution of the Riemann problem, specifically case I as is explained in section 3.2. The left figure shows the solution from a Godunov-like total variation diminishing method (TVD), the middle figure shows a Osher-Chakravarthy total variation bounded method (TVB), and the right figure show a simple central difference method without total variation preserving properties. The TVB method, having the strictest constraint on the total variation, has the least spurious oscillations. The TVD method has oscillations but still keeps them small. The central difference scheme, since it does not consider the total variation at all, lets the oscillations grow uncontrollably.

2.2 Godunov-type Method

Neutron star mergers are very dynamic with highly nonlinear fluid flows and strong shocks. Shocks can be problematic for fluid solvers as they cause numerical instabilities in the form of spurious oscillations. HRSC methods based on Godunov's method are well suited for modelling shock because they are TVD methods. They suppress the oscillations that would otherwise form at the discontinuities around shocks.

To achieve higher order accuracy than Godunov's method, we use reconstruction to compute fluid variables at cell interfaces. In a finite volume method, the fluid values are reconstructed at the grid cell interfaces, one reconstruction for the left hand side and another for the right hand side. Then the Riemann problem is solved at each interface for a flux of each fluid variable through the cell interface [10]. The resulting flux is integrated, here using a Runge-Kutta 3 (RK3) integrator,

to obtain new values for the fluid variables at the center of the cell.

We use the Parabolic Piecewise Method (PPM) [11] for the reconstruction step. Reconstruction is similar to careful interpolation to avoid spurious oscillations due to shocks. The method first performs a quartic polynomial interpolation at each cell boundary using the fourth-order Lagrangian interpolation formula. We set the left and right side values $u^{\ell}_{i+1/2}$ and u^{r}_{i+1} of the interface to be

$$u^{\ell}_{i+1/2} = u_{i+1/2} \qquad \text{and} \qquad u^{r}_{i+1} = u_{i+1/2}.$$

We then apply monotonicity conditions [11] so that the interpolating parabola is monotone. The entire PPM process is illustrated in Fig. 2.2.

The Riemann problem's solution is approximated using these fluid values to get a flux across cells using the Harten, Lax, van Leer and Einfeldt (HLLE) solver [12], which simplifies the structure of the solution and is based on conservation properties of the equations. The Riemann problem is a well studied problem in fluid dynamics consisting of two fluid regions of differing pressures, densities, and velocities separated by discontinuity. The HLLE solver gives an approximate solution to the flux of fluid variables through the interface. If u^{ℓ} and u^{r} are the fluid state variables on the left and right hand sides respectively, the numerical flux vector is given by

$$\hat{f} = \frac{\lambda_r^+ f(u^{\ell}) - \lambda_{\ell}^- f(u^r) + \lambda_r^+ \lambda_{\ell}^- \left(u^r - u^{\ell}\right)}{\lambda_r^+ - \lambda_{\ell}^-}, \tag{2.6}$$

where λ_{ℓ}^- and λ_r^+ are the fastest left and right moving characteristic speeds, respectively. As explained above this flux is used to compute $\partial_t \mathbf{u}$ used in the RK3 integrator in 2.5 to evolve $\mathbf{u}$.

The PPM method guarantees that $TV(\mathbf{u})$ is always diminishing. It also attempts to limit the dissipation inherent to TVD methods. However, in some cases it can add too much dissipation the solution, which can blur out interesting features from the real solution. Additionally, with a seven point stencil it is also only 2nd order accurate for smooth solutions. This is much more computationally expensive than the naive centered difference scheme. The Osher-Chakravarthy method has a much higher accuracy as it is a 5^{th} order accurate scheme for the same stencil.

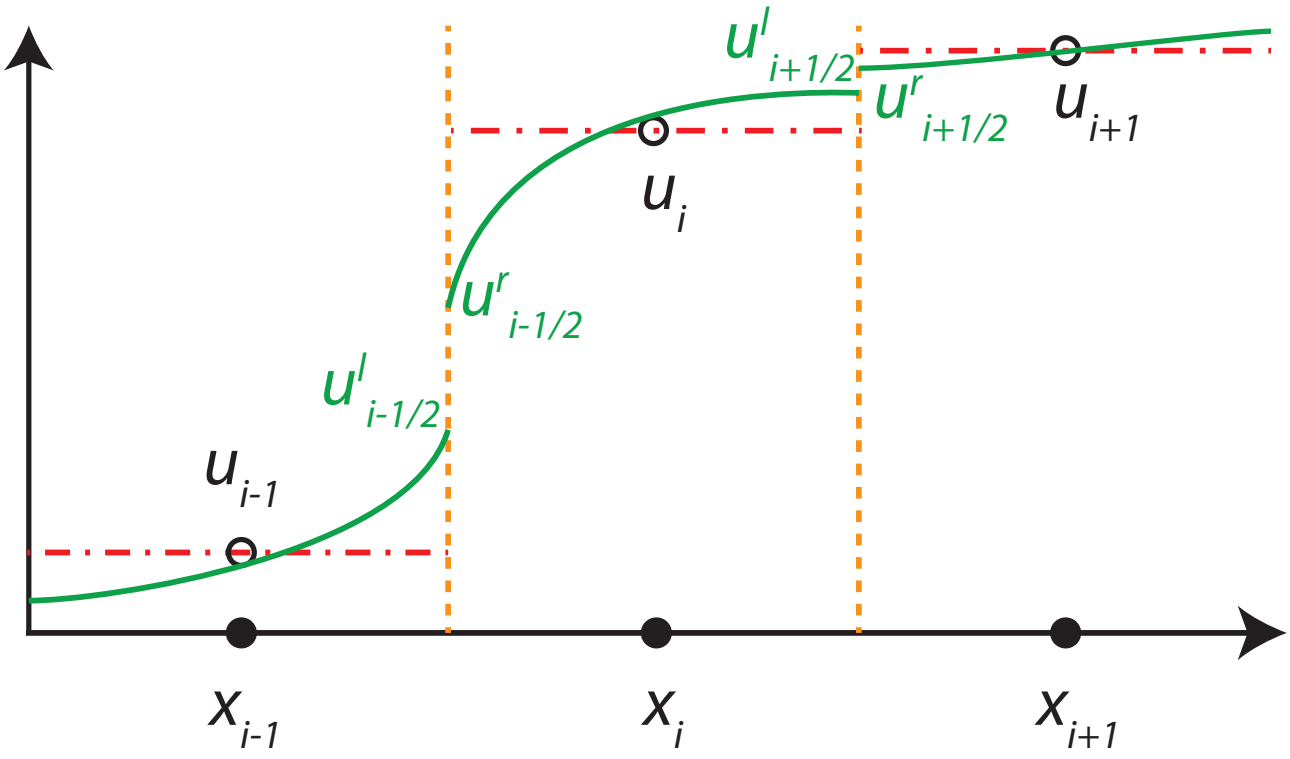


Figure 2.2 This illustrates the reconstruction of the fluid variable u. The average value of u of the i^{th} cell is u_i, which is indicated by the dot-dashed line (red). The variable u is reconstructed using piecewise parabolic functions, as indicated by the solid lines (green). The reconstructed values $u^{\ell}_{i+1/2}$ and $u^{r}_{i+1/2}$ at the interface between x_i and x_{i+1} are obtained from the piecewise functions at the interface on both sides. In Godunov-like methods, the Riemann problem is solved at each interface using $u^{\ell}_{i+1/2}$ and $u^{r}_{i+1/2}$.

2.3 Osher-Chakravarthy Finite Difference Method

Osher-Chakravarthy methods are a class of high order and accuracy TVB methods [13]. They approximate the spatial derivative of the flux using a simple centered derivative with a dissipation correction term to suppress numerical instabilities. It is shown in [14] that the centered $2m-1$ order Osher-Chakravarthy scheme can be written as finite difference scheme. In the Osher-Chakravarthy finite difference (OCFD), The time derivate $\partial_t u_j$ used for updating can be written as

$$\partial_t u_j = -C^{2m} f_j + (-1)^{m-1} \beta (\delta x)^{2m-2} D_+^m D_-^{m-1} (\lambda_{j-1} D_- u_j),$$

where C^{2m} is the $2m$ th order centered difference operator, D_+^m is the m th order forward difference operator, and D_-^{m-1} the backward difference operator, with $2m+1$, m, and $m-1$ point stencils respectively. Here λ_j is the spectral radius of the eigenvalues of the flux function. The positive coefficient β controls the strength of the dissipation term. In [14] it is also shown that for a fifth-order accurate scheme with $m=2$, the coefficient β has an optimal value of $2/75$.

Both the HLLE and OCFD methods use seven point stencils, meaning to evolve one point the algorithm needs 3 points on either side in each of the x, y, and z directions. The larger the stencil the more computation will be required. Although both use seven point stencils, the HLLE method is only 2^{nd} order accurate while the OCFD method is 5^{th} order accurate, giving much better results away from the shock for the same stencil. The simple finite difference operators are also much simpler to compute than the PPM algorithm (see Fig. 2.3). This in part allows the OCFD method to run much faster than the HLLE method.

Since both the Godunov and OCFD methods heavily use finite difference like operations, they are well suited for GPU programming. The OCFD method only needs simple stenciled derivative methods. The Godunov method builds on these methods with just a few modifications in the PPM algorithm to handle the monotonicity conditions. Both are able to vectorize well on the massively parallel GPU architectures, as will be discussed in the next section.

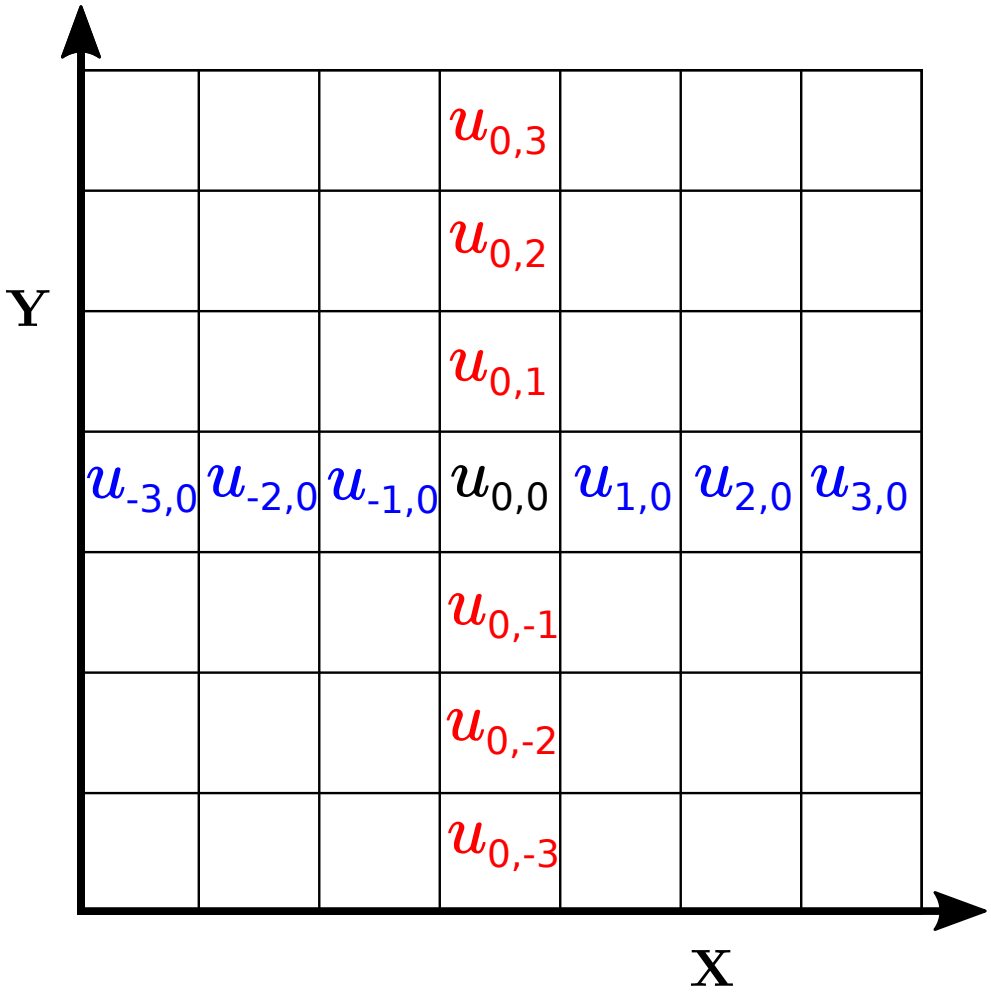


Figure 2.3 The seven point stencil used for both the HLLE and OCFD methods. The point in the center, $u_{0,0}$, is updated using the nearest three points in each direction. For example, in the OCFD method, to compute the central difference derivative in the x-direction at $u_{0,0}$) the blue values in the row of $u_{-2,0}$ through $u_{2,0}$ are summed using the appropriate weight. To compute the derivative in the y- direction the red values in the column through $u_{0,-2}$ through $u_{0,2}$ are used.

2.4 Implementation on GPUs

Programming for GPUs can be very different then programming for CPUs because of inherent differences in their architecture. Performance gains by GPUs over CPUs are achieved by putting hundreds of multiprocessors on the GPU while sacrificing their ability to operate completely independently like a CPU. Kernels, or functions to run on the GPU, must be written to perform the same function on hundreds of data points simultaneously or any performance advantage will be lost. GPUs also have very specific memory access constraints for optimal performance, both for memory access on the GPU and between the GPU and CPU. Memory stores on the GPU are very limited, which is the biggest constraint for the code. It is key to properly use the GPU memory hierarchy to fully exploit the GPU.

Fig. 2.4 details the memory for NVIDIA's Kepler architecture. We chose to work with NVIDIA's CUDA C to interface with the GPUs. Programming for other GPUs and many-cored processors uses the same principles discussed here. However, we will use CUDA terminology for some of the memory structures and details specific to NVIDIA's GPUs.

The main persistent memory of the GPU is limited to several GB of Dynamic Random Access Memory (DRAM), referred to as global memory. Fluid state variables and intermediary variables such as $d\mathbf{u}/dt$ and the fluxes are stored in global memory. Since these variables are not needed at each step of the fluid method, several variables are overlaid in memory. Without resorting to expensive paging algorithms to store memory on the host, the grid computed by each GPU is limited by the size of global memory. For the Godunov-type method, this allows for about a 162^3 grid. DRAM is also very slow, both for memory operations between the host and device and also on the device itself. It is also optimized for memory accesses of adjacent points in blocks of 32 threads. For this reason the GPU provides various levels of memory caches.

Above DRAM in the GPU memory sits the 64KB L1 Cache for each multiprocessor, normally used for shared memory, which is the memory shared by blocks of 32 threads. Recent CUDA com-

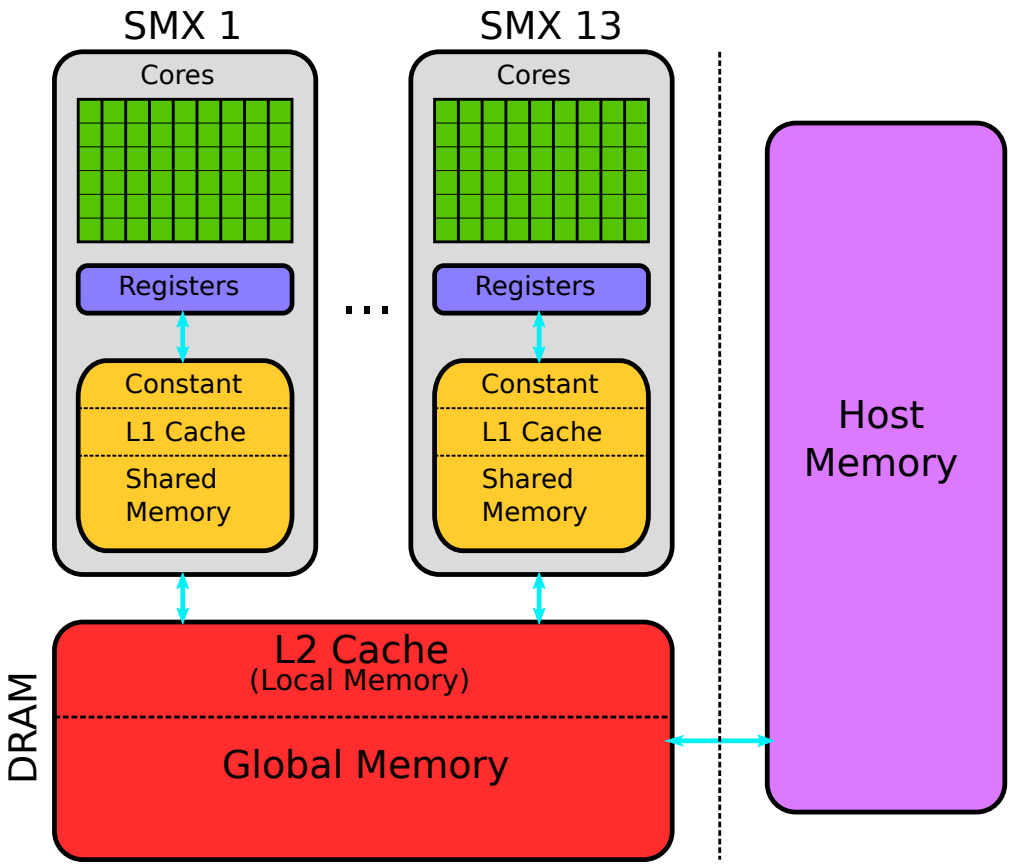


Figure 2.4 Here the memory hierarchy of GPU memory is illustrated are well as the general programming architecture. The GPU splits the DRAM space into global memory for persistent data and the L2 cache. Each streaming multiprocessor (SMX, the Tesla K20 has 13) has a small data storage split between shared memory for sharing data between threads on the SMX; the L1 cache; and constant memory for variables constant through kernel execution, such as function arguments. The GPU transfers memory to the host through global memory, which is a time-consuming operation.

pilers also provide the option to use the DRAM for extra shared memory if needed. However, this is undesirable as the DRAM is orders of magnitude slower than the L1 Cache. The shared memory's small size often limits the number of threads that can run concurrently. The main purpose of shared memory is to serve as a cache between global memory and the registers on the processor, where computation actual occurs.

Each multiprocessor on a GPU has a small memory space called the registers. Processor operations such as multiplication and addition work directly on data in this memory space. CUDA automatically handles data transfers to and from the registers. However, if too many registers are needed for a particular kernel, the registers spill into what is called local memory, which is stored in DRAM. Too many of these spill stores can severely limit performance. This can be avoided by splitting kernels and instructions into several simpler pieces. GMHD achieves this by performing functions such as reconstruction, computing the numerical flux, and spatial derivatives in separate kernels.

Memory transfer between host memory, or CPU memory, and DRAM is time consuming. In order to reduce these memory transfers, all steps of the algorithm are performed on the GPU. Data are only transferred to host memory to write to disk and to send to other processors. This is in contrast to many other GPU codes which use the CPU to perform a limited number of numerically intensive calculations such as root finding. Each of these limitations are addressed in the implementation of the fluid methods in order to fully exploit the GPU.

In finite difference or similar methods data points are used multiple times to compute derivatives and interpolations. Rather than load these points from global memory every time, they are first cached into shared memory. 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 is used, which is based on a tiling scheme for finite difference derivatives [15]. Within a tile, data are first loaded from DRAM into the shared memory cache on the chip. The threads in

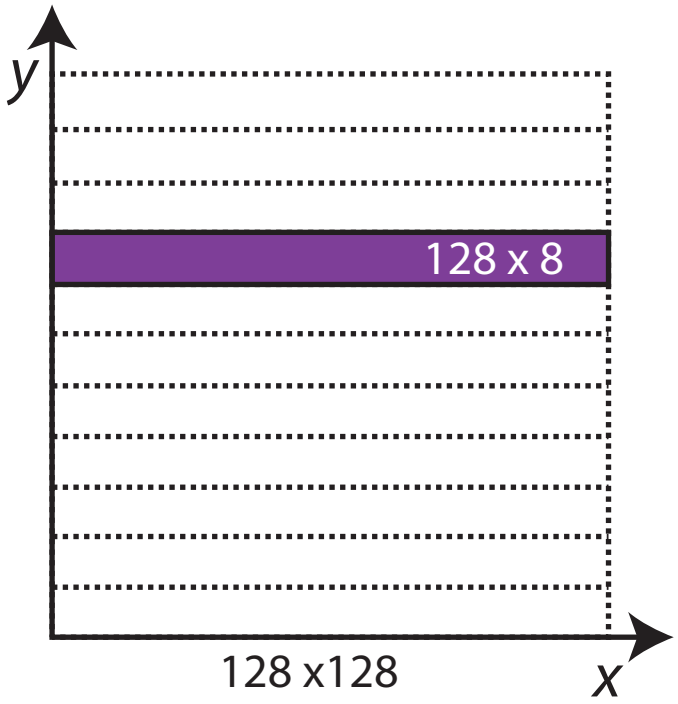


Figure 2.5 This figure illustrates the tiling scheme for a five point stencil in the x direction on a 128^3 grid. The purple region represents the derivatives that will be computed using a single block of shared memory. Unlike the tiling for y and z directions, tiling in the x direction will load each point into shared memory exactly once.

the tile are then synchronized. Data are then loaded from the shared memory cache into the local registers for computation. Because data are arranged on the GPU along the x-direction, tiling is different for y- and z-derivatives than for x-derivatives, as shown in Fig. 2.5 and Fig. 2.6.

For reconstruction and derivatives in the x-direction, blocks of threads span the width of the 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.

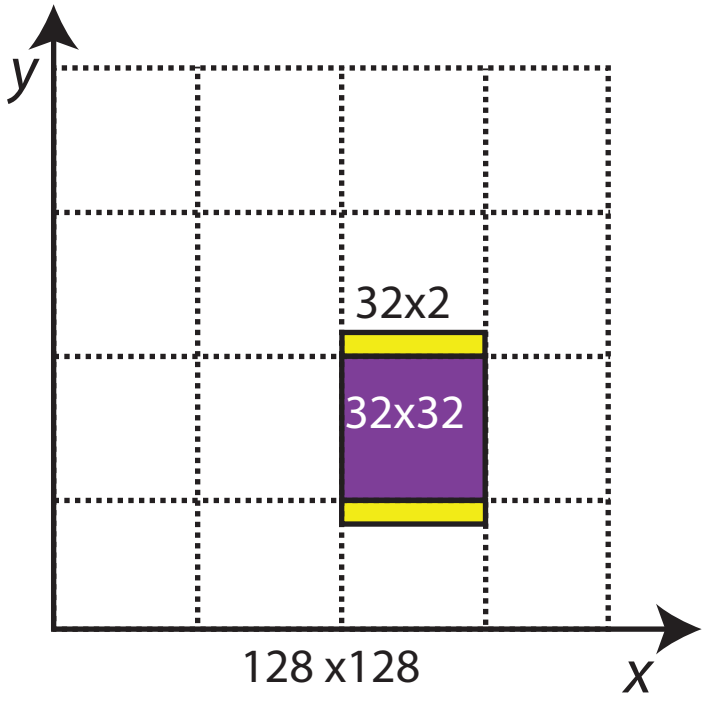


Figure 2.6 This figures illustrates the tiling scheme for a five point stencil in the y direction on a 128^3 point grid. The purple region represents the points for which the derivative is computed. The yellow region represents points which must be loaded into memory to compute the derivative for points on the boundary. These data points will be loaded in twice, once here and again to compute the blocks adjacent in the y- direction. Tiling in the z- direction also uses the same scheme, only transposed into the x-z plane.

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.

The Godunov method and RMHD equation implementation is split across 96 kernels. Within the kernels themselves, apart from the detailed memory treatment described above, GMHD 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.

2.5 Communication Using MPI

Problem sizes are often too large for a single processor or node. In this case we need to run on several distributed nodes, each with a separate GPU. Dividing a problem over multiple compute nodes can speed up run times. A single compute node is extremely limited in power and memory capacity. Using multiple compute nodes in a supercomputing setting gives much more computing power and allows larger simulations to be run. However, it requires careful programming to efficiently utilize these resources.

GMHD uses the Message Passing Interface (MPI) to communicate between compute nodes and GPUs. MPI is a standard and well established message passing system for high performance computing environments. It allows data to be transferred directly between RAM stores on different hosts. Recent implementations such as OpenMPI and MPICH have even included optimized message passing between GPUs on different hosts. Previously data needed to be transferred first from the device to the host, transferred to the other host (which involves an extra buffering step on the host), and then finally onto the other GPU. Now data can be directly buffered directly from the GPU into host memory, then transferred onto the other GPU. Overall this gives a small improvement for

GPU to GPU memory transfer times.

Currently GMHD only supports a static, uniform grid defined at run time. The large uniform grid is split into many smaller grids, one for each compute node with a GPU. However, in order to evolve points on the edge of the grid, the stencils require points outside of the grid. In order to accommodate the stencil, extra points called ghostzones are added to the grid borders. The number of ghostzones depends on the fluid method. For example, the OCFD scheme requires a seven point stencil, so three ghostzones are needed, as is shown in Fig. 2.7. Because the ghostzones now lie on the grid boundaries they are not evolved but are updated through other means. Ghostzones that lie on the domain boundaries are updated by applying boundaries conditions, such as periodic, vacuum, and outflow boundary conditions. Ghostzones on grid borders that touch other grids in the MPI layout are set up to overlap real points on the bordering grid. These points are updated by retrieving the evolved values from the other grid. Since the other grid will also require ghostzones, each grid overlaps its neighbor by twice the number of ghostzones, which is shown in Fig. 2.8.

Because MPI can only transfer contiguous blocks of memory, exporting and importing functions are used to prepare and unload blocks of boundary data. Since memory is already adjacent in the x direction, this is simple for the y and z boundaries. For a 162^3 grid with 3 ghostzones on each boundary, the y and z boundaries are $156 \times 3 \times 156$ and $156 \times 156 \times 3$ blocks. However, in the x direction, the border is $3 \times 156 \times 156$, which is strided in memory. We also cannot force memory accesses of 32 adjacent points since only three are needed. The inefficiency is inherit to the border dimensions, but realistically has little effect on performance since the exporting and importing functions take relatively little time. The extra overhead from communication does detract from performance gains. However, GMHD demonstrates good scaling properties, as will be shown in the next chapter.

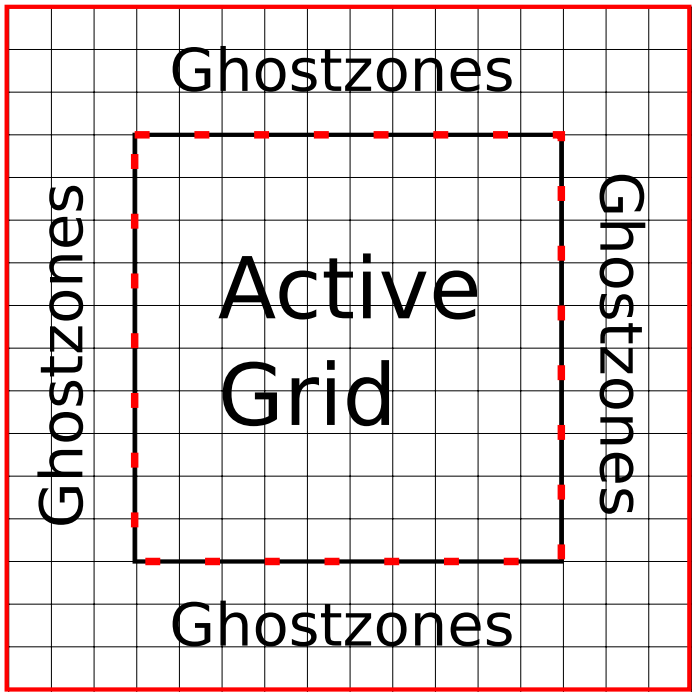


Figure 2.7 This figure illustrates the ghostzones for a grid using a method with a seven point stencil. The cells within the dotted lines grid comprise the active grid and have complete stencils and can be updated locally. The three cell borders for each grid represent the grids' ghostzones, which must be updated by either applying boundary conditions or retrieving data from another grid.

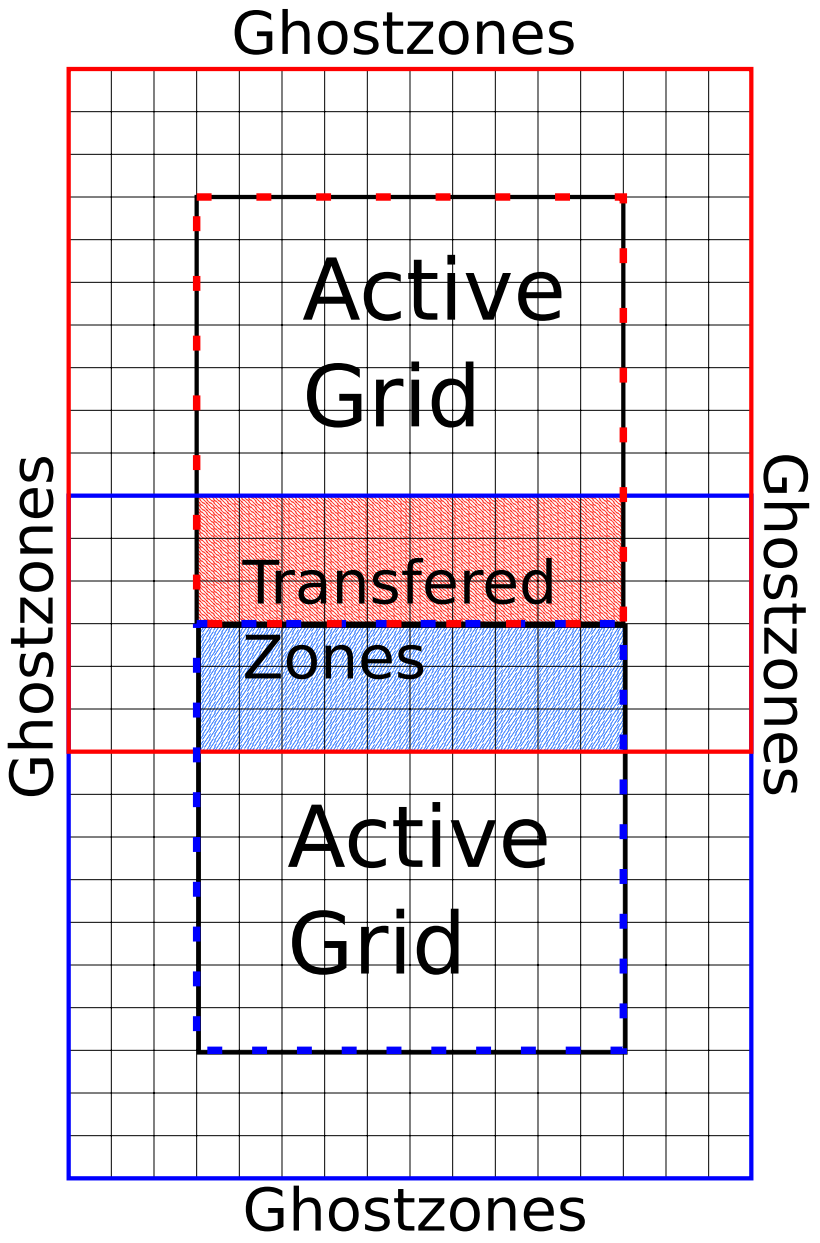


Figure 2.8 This figure illustrates the setup of two grids, outlined in red and blue, and their overlapping ghostzones for a method with a seven point stencil. The shaded region in the center represents data which will be transferred between the two grids using MPI.

Chapter 3

Results

This chapter presents tests of the fluid methods and the scalability of the code. After a brief overview of fluid tests in Section 3.1 below, I will present results for both numerical methods for solving the Riemann problem in Section 3.2 and the Kelvin-Helmholtz instability in Section 3.3. The parallel performance of the code is demonstrated for strong and weak scaling tests in Sections 3.5 and 3.6 and with a comparison to an existing CPU code, HAD, in Section 3.7. Finally, implications and future work using GMHDwill be discussed in Sections 3.8 and 3.9.

3.1 Fluid instability tests

Fluid codes need to be validated by checking their convergence and verifying that they reproduce well-known solutions. Some well-known instabilities are good test solutions because they have very dynamic fluid flows. The accuracy of solutions obtained using different methods can reveal strengths and weaknesses of the methods. For example, the HLLE method is a high-resolution shock-capturing algorithm that is very robust and can handle strong shocks, but it is also very dissipative. The OCFD method, conversely, is well suited for modeling smooth flows with high precision, such as required to study turbulence, but it performs less well at shock discontinuities.

3.2 Shock tube Riemann Problem

The Riemann problem in fluid dynamics is one of the few nonlinear problems in fluid dynamics with an analytical solution [16]. Thus it is a standard test to gauge how well a fluid method can model shocks. The initial configuration for the Riemann problem consists of two arbitrary constant states that are joined at a discontinuity. The solution general consists of difference constant states that are joined by shock waves, rarefaction waves, or a contact discontinuity. We can gauge the accuracy of a shock capturing method by how well it models the velocity and high pressure of the shock.

We tested both methods for several shock tube conditions. Table 3.1 shows the initial conditions for each test case. In all cases the fluid was evolved on a 300 one-dimensional point grid with physical dimensions $x \in [-1,1]$ out to time $t = 0.8$. A Courant-Friedrichs-Lewy (CFL) condition of .25 was used to determine the time step. The adiabatic gas constant γ for all cases is $4/3$.

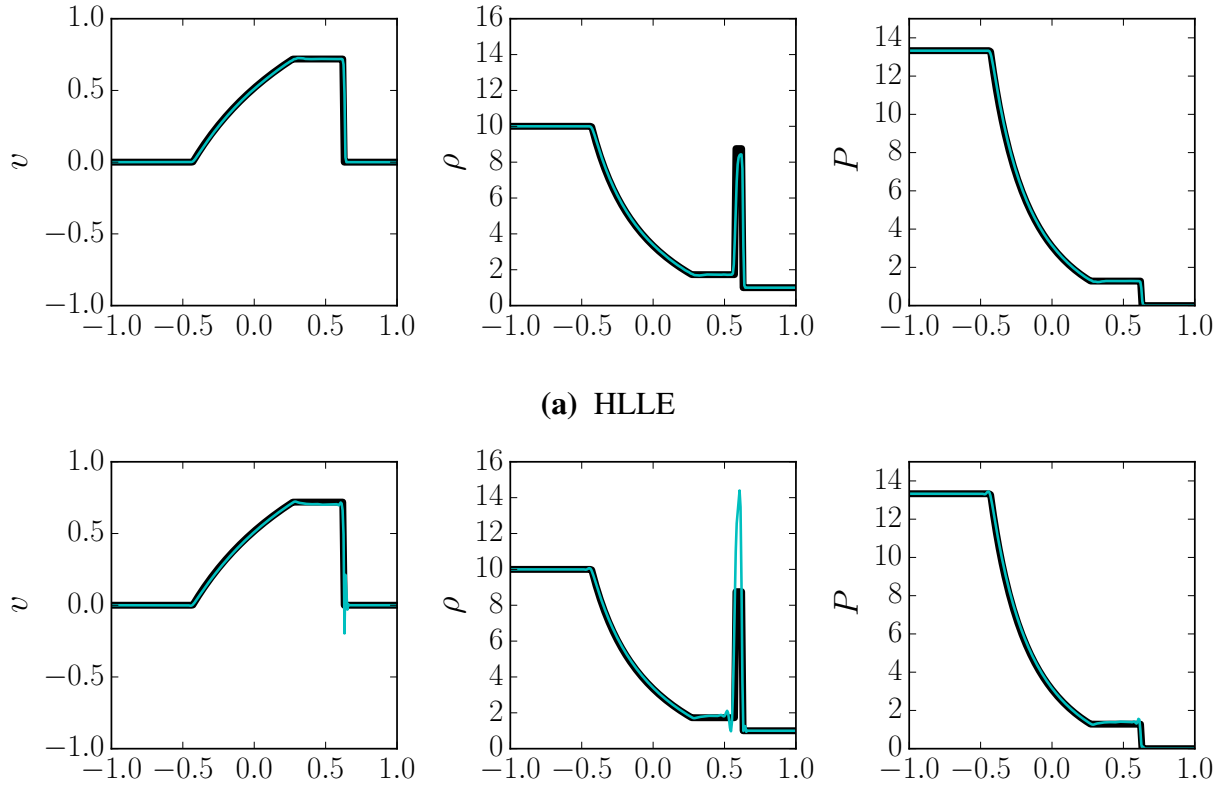
Figures 3.1– 3.5 compare the performance of the HLLE and OCFD methods for the shock tubes shown in Table 3.1. Overall the HLLE method is very robust, it suppresses spurious oscillations about the discontinuities, but it is more dissipative. The OCFD method has faster convergence, but there are spurious oscillations near the discontinuities. These oscillations do not grow in time, but they can introduce numerical problems. In testing, the OCFD method is less robust than the HLLE method.

3.3 Kelvin-Helmholtz Instability

The Kelvin-Helmholtz instability is an instability of the shearing interface between two regions of fluid that move at different speeds. We perturb the boundary to initiate the instability. The velocity shear at the interface causes turbulent flow in both regions. This test can be useful for determining how well a fluid method can model turbulent flows. Because the OCFD method is a higher order

Case		v	ρ	P
I	$x<0$	0	10	13.33
	$x>0$	0	1	10^{-6}
II	$x<0$	0	1	10^{-6}
	$x>0$	0	10	13.33
III	$x<0$	0.2	.1	0.05
	$x>0$	-0.2	.1	0.05
IV	$x<0$	0.99999	0.001	3.333×10^{-9}
	$x>0$	-0.99999	0.001	3.333×10^{-9}
V	$x<0$	0	1	1000
	$x>0$	0	1	0.01

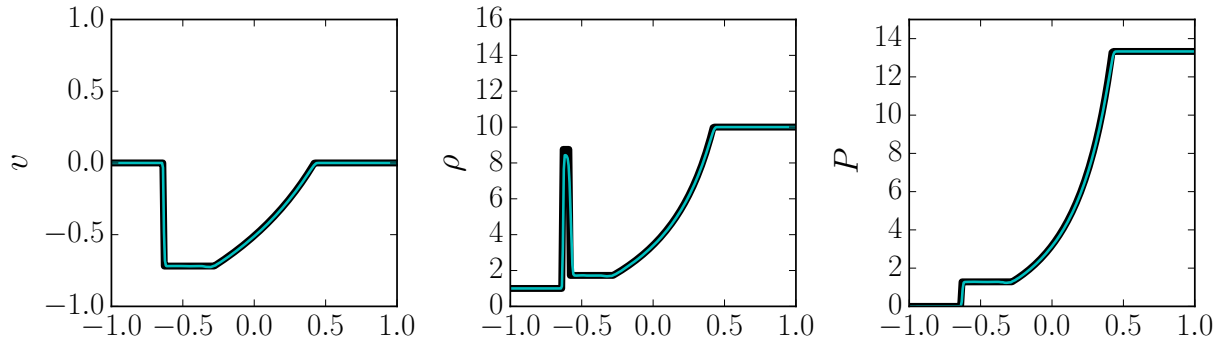
Table 3.1 Initial conditions for the velocity v, density ρ, and pressure P for the four Riemann problem test cases. Case I tests two initial at rest regions with differing densities and pressures. Case II is identical to case I but reversed. It serves as a sanity check on the code to show that it is symmetric. Case III has two regions of equal pressure and density but that have opposing velocities. Case IV is similar but tests two regions of higher velocity moving towards each other. Case V is similar is case I except the regions only differ in pressure by not density, which produces a difficult shock to capture. All units are relative except for v, which is in terms of the speed of light c.



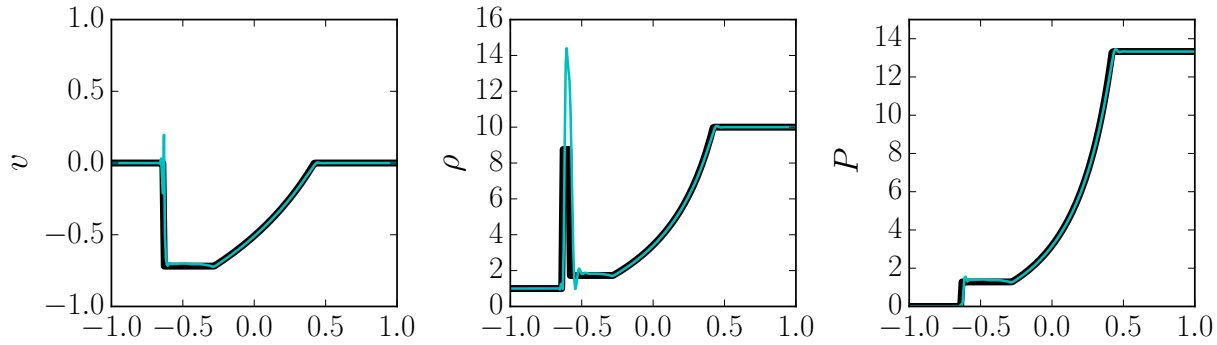
(a) HLLE

(b) OCFD

Figure 3.1 The solution of the Riemann problem for case I. The thin line represents the solution given by each method while the thick line represents the exact solution. Velocity is in terms of c while density and pressure are in relative units. Case I initially has two stationary regions of differing pressures and densities. As time advances a shock should propagate through the lower pressure fluid with a sharp peak in density behind it. Both methods perform well. The HLLE method has no spurious oscillations at the shock. The OCFD method overestimates the density behind the shock and oscillations are evident behind both the shock and around the contact discontinuity.

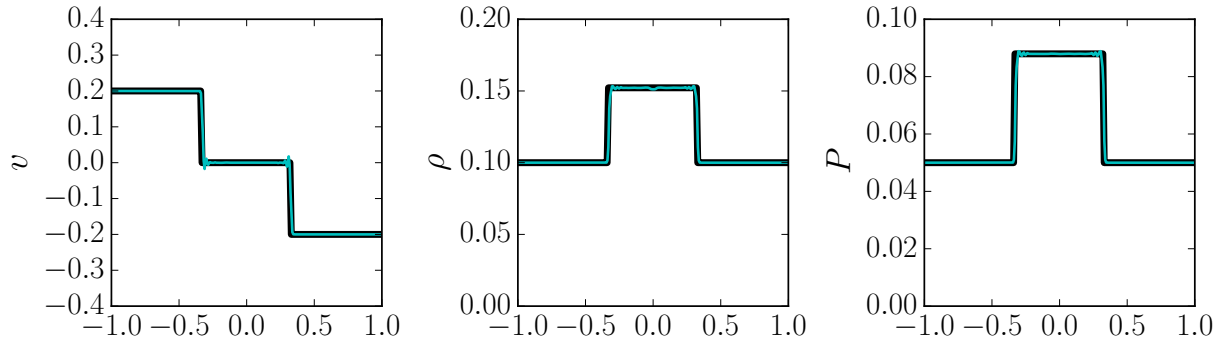


(a) HLLE

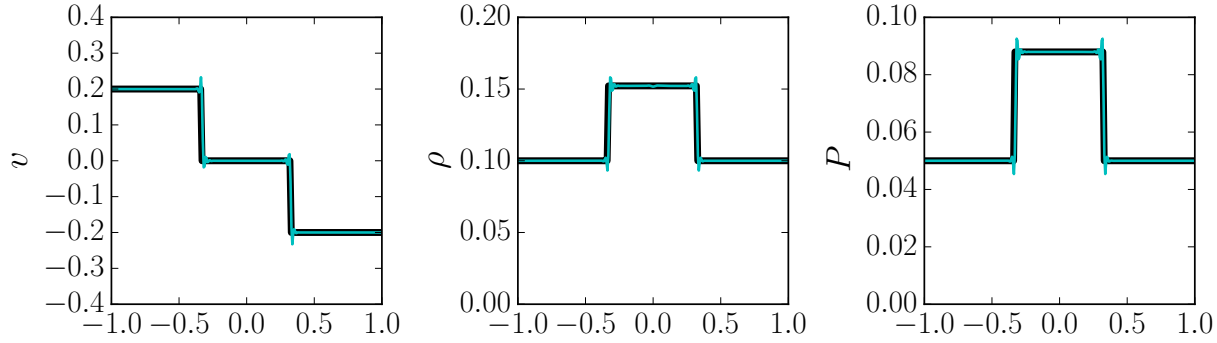


(b) OCFD

Figure 3.2 The solution of the Riemann problem for case II. The thin line represents the solution given by each method while the thick line represents the exact solution. Velocity is in terms of c while density and pressure are in relative units.Case II is identical to case I, except spatially reversed. Since the methods are symmetric this should give reversed solutions to case I. Both methods perform as expected.

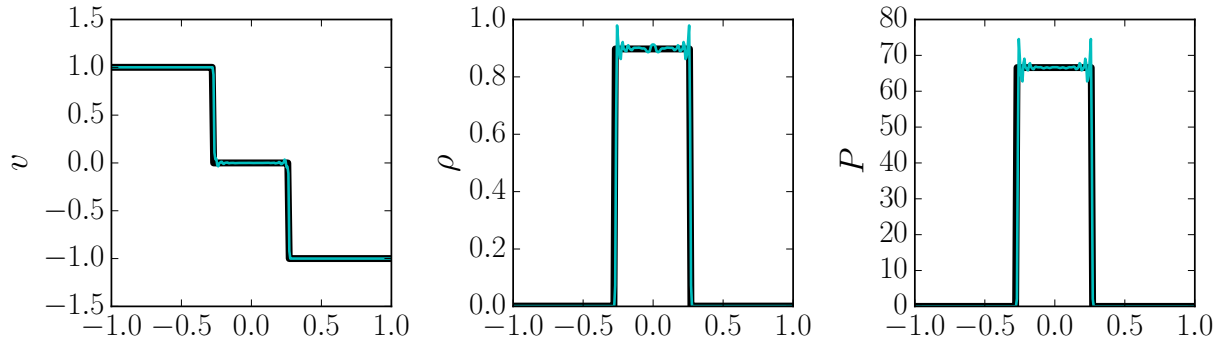


(a) HLLE

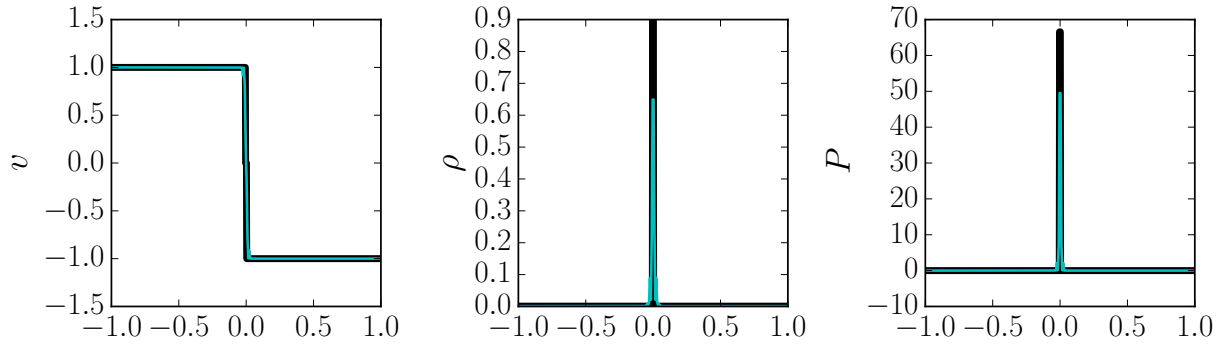


(b) OCFD

Figure 3.3 The solution of the Riemann problem for case III. The thin line represents the solution given by each method while the thick line represents the exact solution. Case III has the two regions initially with opposite velocities. A high pressure shock should form in the middle. Both methods are able to model this, although with oscillations around the shocks. The HLLE method manages to dampen many of these while the OCFD experiences larger oscillations.

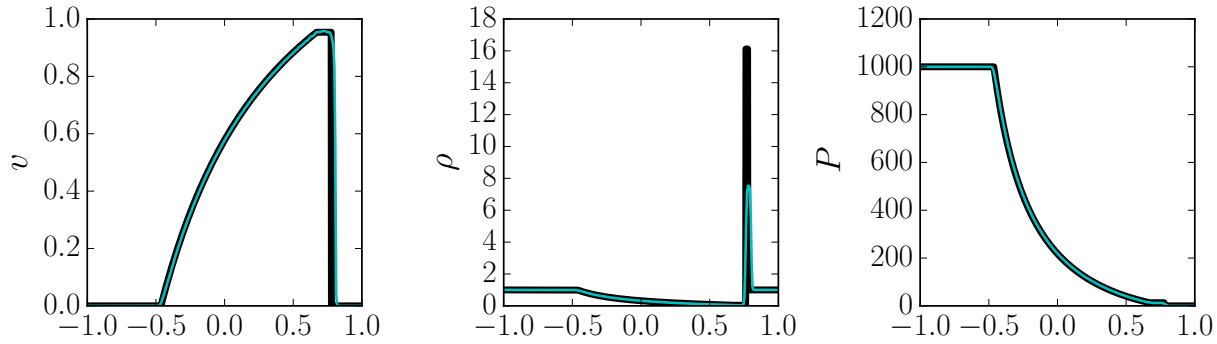


(a) HLLE $t = 0.8$

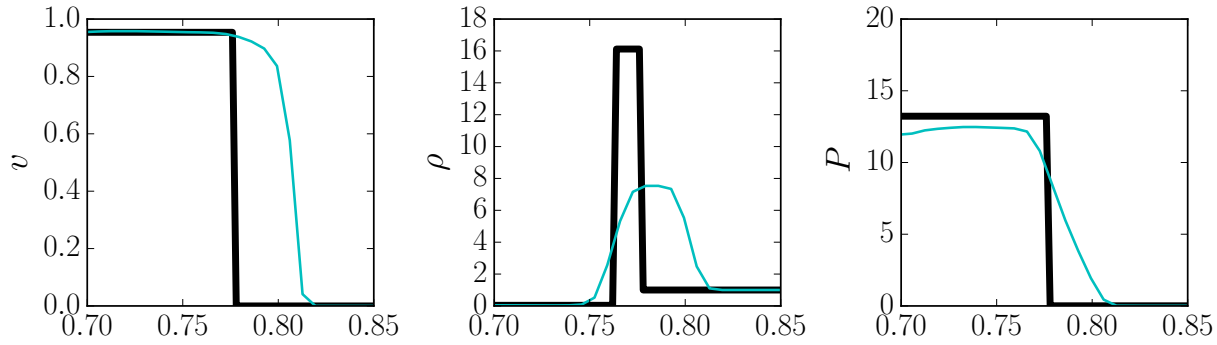


(b) OCFD $t \approx 0.01$

Figure 3.4 Results for case IV. The thin line represents the solution given by each method while the thick line represents the exact solution. Case IV is similar to case III but with higher velocities. Here the oscillations in OCFD cause the method to completely fail after several iterations, as shown in 3.4b. The HLLE method reaches $t = 0.8$ albeit with small oscillations in 3.4a.



(a) HLLE



(b) HLLE

Figure 3.5 Results for case V. The thin line represents the solution given by the HLLE method while the thick line represents the exact solution. The lower graph shows a zoomed in view of the shock. The TVB solution is not shown because it fails during the first time step. Case V is similar to case I that it begins with stationary regions however they differ in pressure instead of density. In the correct solution a well defined shock should move through the lower pressure fluid. However, the highly dissipative HLLE method smears the high pressure behind the shock and overestimates its value.

method for the same stencil as the HLLE method, we expect it to better model the instability.

We used both methods to produce solutions to the Kelvin Helmholtz instability. The test was run on a two dimensional $[-0.5, 0.5] \times [-1.0, 1.0]$ domain with periodic boundary conditions, meaning the top and bottom edges were connected as well as the left and right. This gave 2 regions, an outer and inner region, with 2 interfaces. The regions were given equal pressures, slightly differing densities, and an equal but opposing v_x. A small perturbation was introduced at the boundary with a small sine wave in v_y. The tanh function was used in place of a step function to give a smoother transition between the two regions. The initial conditions for interfaces at $y = \pm 0.5$ are described as follows:

If $y \leq 0$

$$\begin{aligned} \rho(x,y) &= \rho_0 - \rho_1 \tanh\left(\frac{y+0.5}{w}\right) \\ v_x(x,y) &= -v_s \tanh\left(\frac{y+0.5}{w}\right) \\ v_y(x,y) &= -a_0 v_s \sin(2\pi x) \exp\left(\frac{(y+0.5)^2}{\sigma}\right) \end{aligned} \tag{3.1}$$

If $y \leq 0$

$$\begin{aligned} \rho(x,y) &= \rho_0 + \rho_1 \tanh\left(\frac{y-0.5}{w}\right) \\ v_x(x,y) &= v_s \tanh\left(\frac{y-0.5}{w}\right) \\ v_y(x,y) &= a_0 v_s \sin(2\pi x) \exp\left(\frac{(y-0.5)^2}{\sigma}\right), \end{aligned} \tag{3.2}$$

where ρ_0 and ρ_1 are the densities of the two regions, w is the shear layer width, v_s is the velocity shear, a_0 is the amplitude of the perturbation, and σ controls the width of the perturbation. Table 3.2 gives values for each of these constants.

Figure 3.6 shows the solutions given by each method. Both methods were able to capture some of the turbulent flow at the fluid interface. The OCFD method produced unphysical swirls between the physical curls of the turbulent flow, which is characteristic of higher order methods. However,

ρ_0	0.505
ρ_1	0.495
w	0.01
v_s	0.5
a_0	0.1
σ	0.1

Table 3.2 Values for constants used in the initial conditions for the Kelvin-Helmholtz instability from Eq. 3.1 and 3.2.

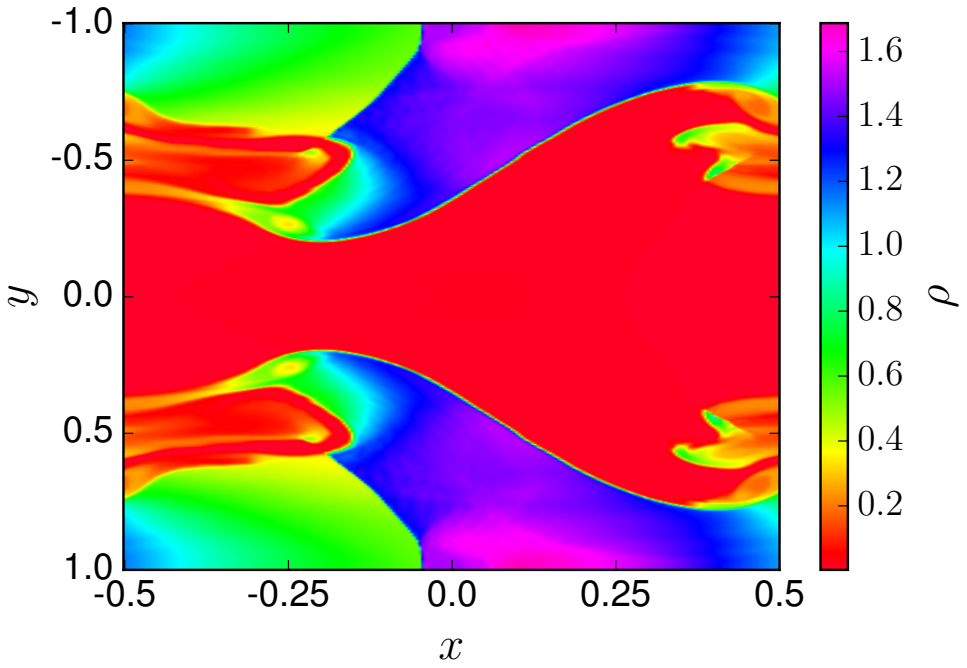
the OCFD method captured the smooth flow near the end of the curl of the turbulent flow better than the HLLE method.

3.4 Timing Benchmarks

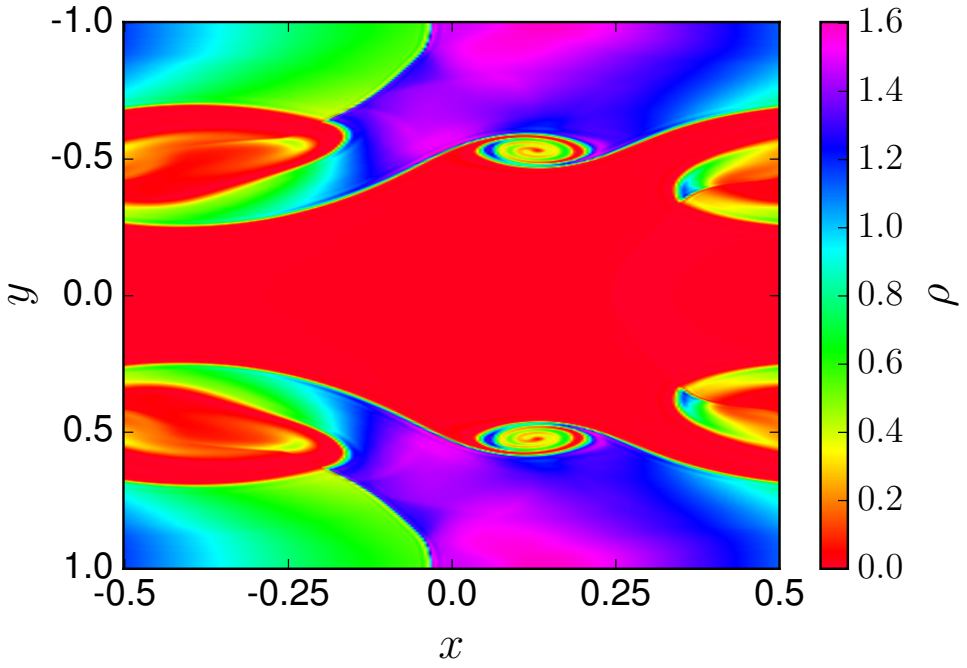
We investigated the scalability of the code with strong and weak scaling tests. Strong scaling measures the ability to split a problem up between processors. Weak scaling measures how well the code can handle larger problem sizes and the number of compute nodes is increased. In both aspects the code scaled well. We also compared the GPU code to the mature CPU-based code HAD [4]. The GMHD code outperformed the HAD code by a factor of 2.5. The details of these tests follow.

In all timing tests we used a Gaussian pulse, analogous to an explosion in space, as the initial conditions. The Gaussian pulse provides a simple test case that gives a roughly equal workload for all processors within the blast radius. Table 3.3 gives the dimensions and parameters for the initial Gaussian pulse used for each scaling test.

Indiana University's Big Red II super computer with 676 Tesla K20 GPU nodes was used for scaling tests. NVIDIA's Tesla K20 GPU is a Kepler architecture GPU coprocessor specifically



(a) HLLE



(b) OCFD

Figure 3.6 Density of the solutions of the Kelvin-Helmholtz instability given by the HLLE and OCFD methods. Both methods capture the large curl across the periodic bounds of the domain but the OCFD method resolves the flow better. The OCFD solution also has the unphysical curls near $x \approx 0.1$ that are typical of higher-order fluid methods.

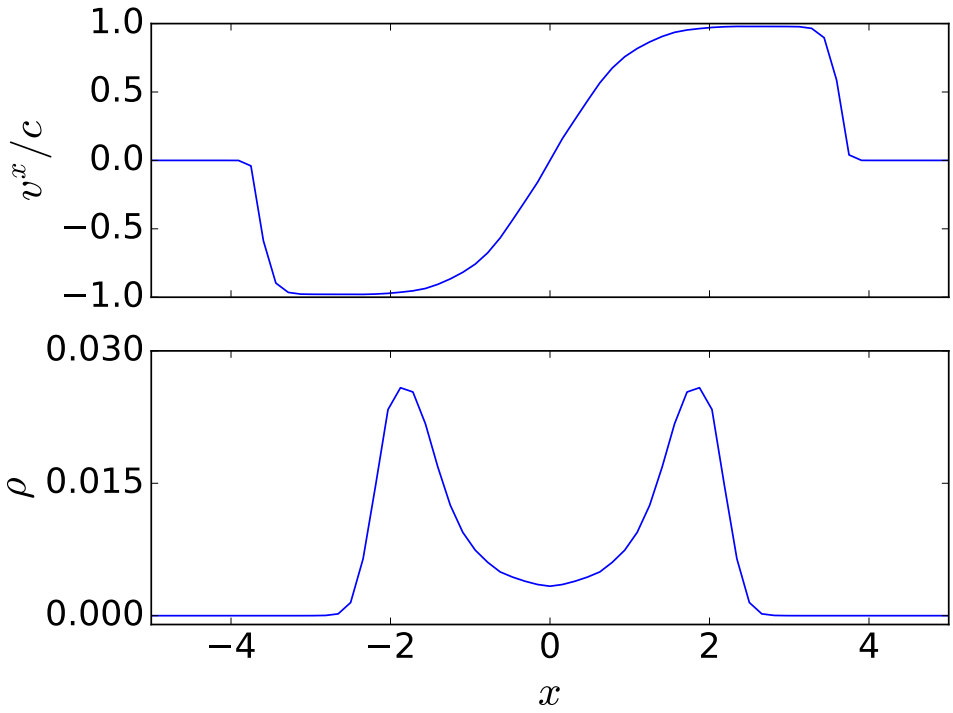


Figure 3.7 The velocity component v^x and density of an evolved Gaussian pulse plotted along the x direction. The initial Gaussian has spread out into two advancing blast fronts.

Parameters	GPU Weak Scaling	GPU Strong Scaling	HAD Scaling
Num. of points	$160 \times 162 \times 160$	$160 \times 162 \times 160$	161^3
$x, y, z \in$	$[-50, 50]$	$[-50, 50]$	$[-5, 5]$
A	100	10	10^{-3}
σ	5	1	1
κ	1	1	0.1
δ	10^{-7}	10^{-7}	10^{-9}

Table 3.3 The grid sizes and the parameters for the initial Gaussian pulse used for the scaling tests. The grid sizes define the dimensions per node for the strong scaling test and the dimensions for the entire domain for the weak scaling and HAD tests. x, y and z specify the physical size of the domain. The parameters σ and A give the deviation and mean of the Gaussian pulse for the density. The pressure is proportional to the density by κ while δ sets a floor value for the density.

built for scientific computing. Table 3.4 show the specifications for the NVIDIA Tesla K20 GPU. The most relevant value is the 5GB DRAM, which determined the $160 \times 162 \times 160$ grid size as the largest that could fit in memory.

3.5 Strong Scaling test

Strong scaling demonstrates how much faster a code can run when given more resources. If a particular job takes too long to finish, a code with good strong scaling can use more nodes to finish the job much faster. Typically only strong scaling out to about 10 times as many nodes is important, since computing resources are usually too constrained to allow for more nodes.

In a strong scaling test a fixed problem is divided over an increasing number of processors. The problem size is chosen such that the grid size can be evenly divided in each dimension several times. The simulation is then run on an increasing number of compute nodes with smaller work

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

Table 3.4 Specifications for the NVIDIA Tesla K20 GPU used for all scaling tests.

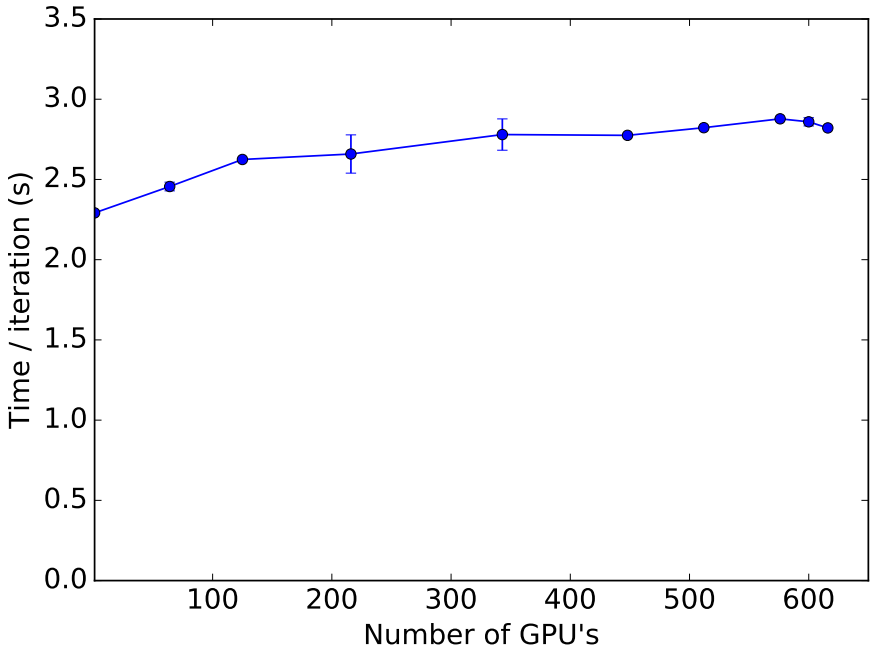


Figure 3.8 This figure shows the average execution time per iteration of the GPU code in a weak scaling test. Times are calculated from the average run time of five runs, each with 150 iterations and $160 \times 162 \times 160$ points per GPU. Error bars are calculated as 2σ. Simulations were performed on Indiana University's Cray XE6/XK7 (Big Red II).

loads per node. Ideally, the runtime will be inversely proportional to the number of nodes: doubling the node count should halve the runtime. Communication overhead and the increased number of ghostzones, however, impede these speed ups. 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. For a code to exhibit good strong scaling the runtimes should follow the inverse curve as closely as possible.

GMHD exhibited good strong scaling, with diminished returns for node counts larger than 40. The strong scaling test, shown in Figure 3.9, used the largest problem size that could fit on a single GPU node. The performance improvement asymptotes at slightly more than 40 GPU nodes with

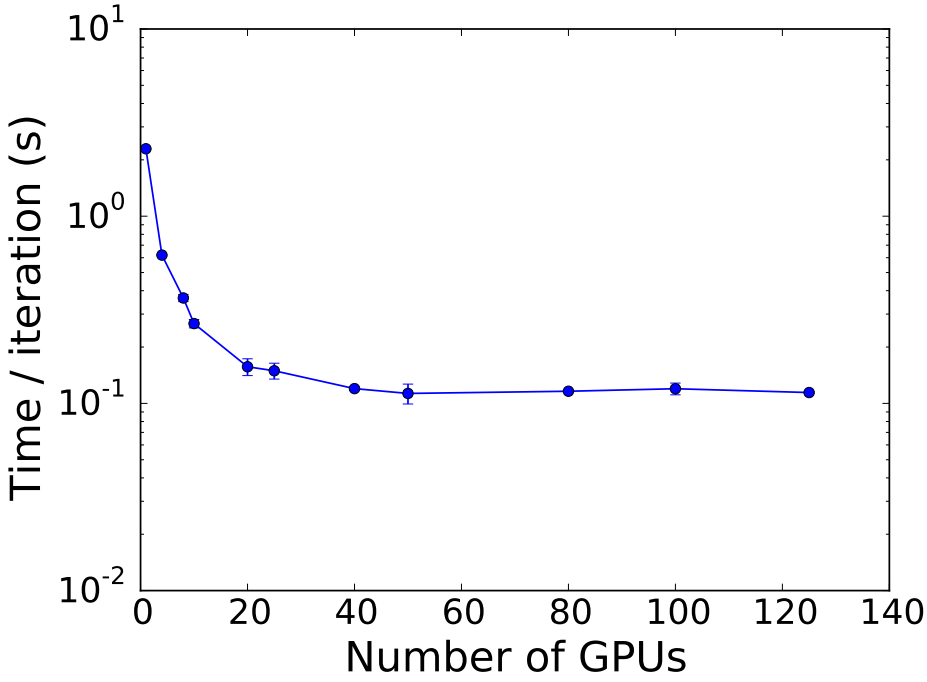


Figure 3.9 This figure shows the average execution time for a single iteration of the fluid code in a strong scaling test. Times are calculated from the average run time of five runs, each with 150 iterations and 160^3 points. Error bars are calculated as 2σ.

a 20 times speed up compared to one node. 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. However, the speed ups for node counts below 40 are sufficient to show that increasing the node count by a factor of 10 will reduce runtimes, if the resources are available.

3.6 Weak Scaling Test

Weak Scaling shows how well a code can handle larger problem sizes in the same amount of time given more computing resources. This allows simulations on larger and more refined grids. It is

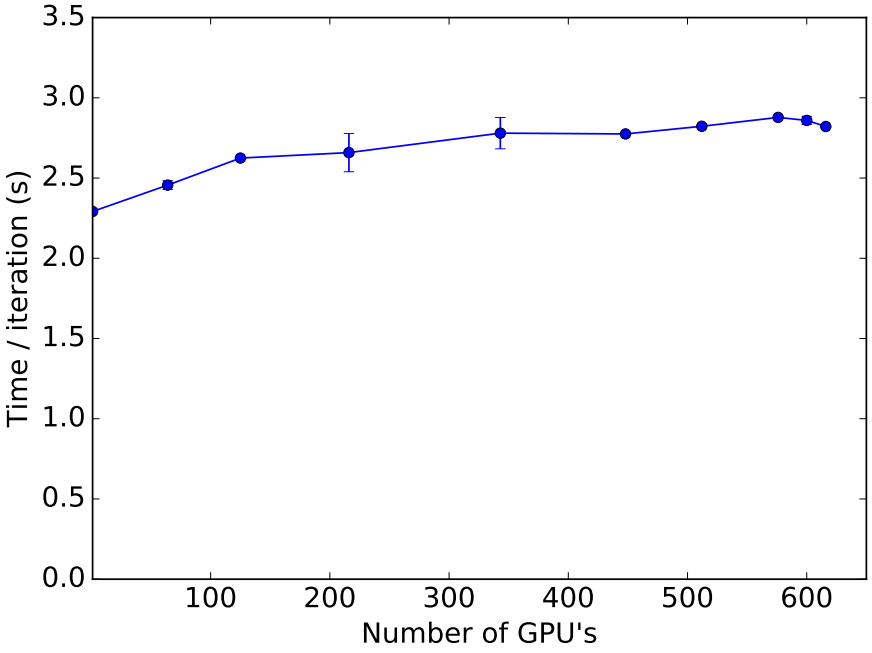


Figure 3.10 Results of the weak scaling test showing the time taken per iteration of the algorithm vs. number of GPU's in the test. In this test the simulation work load per GPU remains constant while the total problem size increase with the total number of GPU's. Ideally, the time per iteration should remain constant as the GPU's count is increased, but communication overhead interrupts this. Because runtimes stay relatively constant, the code exhibits good weak scaling. The data was produced by running a 162^3 grid on each processor using a Gaussian pulse as the initial conditions. For each GPU count the test was run five times for an average. The error bars are calculated as 2σ. Simulations were performed on Indiana University's Cray XE6/XK7 (Big Red II)

also more representative of normal use since limited computing resources are usually used at full memory capacity.

In a weak scaling test, the same amount of work is given to an increasing number of processors, increasing the total problem size. A work load per node is first chosen, usually at the maximum memory capacity of the node. The simulation is then run for increasing number of processors. Given perfect scaling, the runtime will be constant as the number of processors and problem size are increased. However, the overhead of communicating between nodes inevitably slows down the simulations for high processor counts. If the runtimes remain relatively constant, the code is said to have good weak scaling.

The results for the weak scaling test are shown in Figure 3.10. GMHD showed very good weak scaling, running at 1.5× total run time for 676 nodes compared to 1 node. The result is nearly optimal with a slight positive slope in the scaling curve as anticipated due to the serialization of communication for edges and corners. The runtime stays relatively constant as more nodes are added. Due to these results, GMHD can run larger problem sizes without much longer turn around times.

3.7 CPU Comparison Test

The code was also compared to an established traditional CPU code HAD [4]. HAD is a grid based AMR code that implements the same HLLE and PPM algorithm presented here. It scales well up to hundreds of CPU nodes but suffers slow downs around one thousand nodes.

The GPU code ran 2.5× times faster on a single node compared to HAD on a single node with 32 CPU cores. Figure 3.11 compares a CPU-based HAD implementation [4] of the HLLE method with the GPU 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. [17]. This may be due, in part, to the very small CPU core counts that were available for comparison testing in that research.

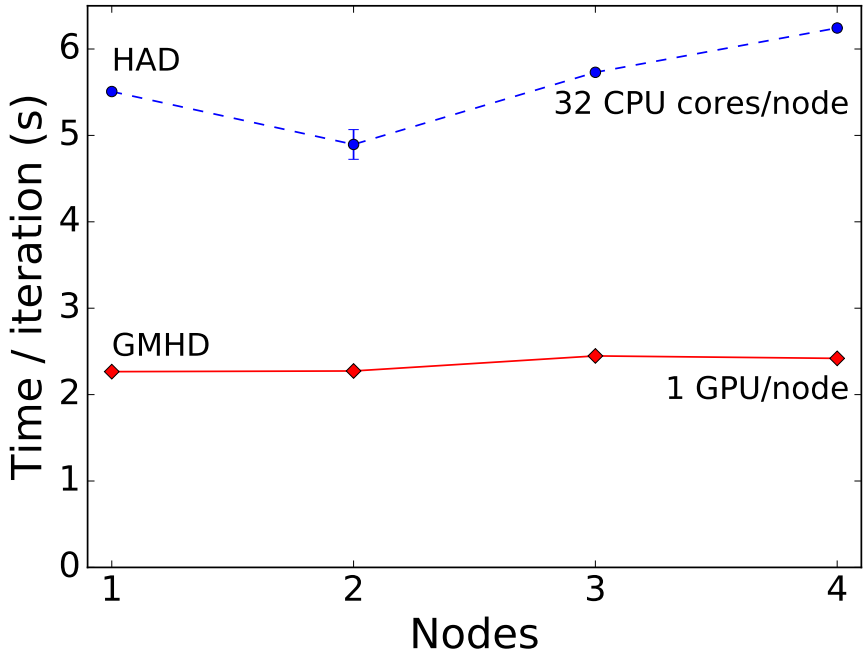


Figure 3.11 This figure compares the performance for the HAD code (optimized for CPUs) and our code optimized for GPUs. The plot shows the average execution time per iteration in a weak scaling test on four nodes. The HAD code was run on CPU-only compute nodes of Indiana University's Cray XE6/XK7 (Big 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 2σ.

3.8 Conclusion

I developed a new hydrodynamics code for compressible fluids that runs efficiently on GPUs. This code, which is designed for relativistic astrophysics applications, can use two different numerical methods. The HLLE scheme is very robust and can evolve strong shocks. The OCFD method is a higher-order method that can be used in simulations with smooth flows, such as turbulent fluid flows. The numerical methods were verified using exact solutions of the Riemann problem and a calculation of the Kelvin-Helmholtz instability.

The performance of the GMHD code was tested by timing the evolution of a Gaussian pulse on Indiana's University's Big Red II. The code displayed very good weak and strong scaling. It also outperformed the established CPU based code HAD with 2.5× faster runtimes. Overall the code demonstrates that GPUs can significantly accelerate calculations such as the HRSC algorithms considered here. Several month long runtimes for simulations of binary star mergers can be reduced to weeks. As GPUs improve the performance gap between GPUs and CPUs is likely to increase. GPUs will be a vital component to exploit for future codes.

3.9 Future Work

The GMHD code was developed to test numerical methods and strategies for large scale simulations of problems in relativistic astrophysics. These simulations require adaptive mesh refinement (AMR) to resolve the many different length scales, from the neutron star interior to the gravitational wavezone. To use the GMHD code for these computations, an efficient AMR algorithm will need to be added to GMHD.

This work also investigated the OCFD algorithm for relativistic fluid simulations. This method is not appropriate for strong shocks, but it gives 5^{th} order accuracy for very little computational cost. Future work will explore coming the HLLE and OCFD methods to efficiently provide high

accuracy in regions with smooth flow and robust HRSC for strong shocks.

One limitation running GMHD was the small 5GB DRAM on the GPU that restricted grid sizes per node. However future GPUs will very likely have more DRAM onboard. For example, the latest NVIDIA Tesla P100 GPU has 16GB of memory, while the NVIDIA Pascal architecture will allow up to 32GB per GPU. Future machines with these GPUs will allow the GMHD code to run on larger problem sizes without slowing down computation. Supercomputers will also implement faster communication links, such as NVIDIA's NVLink, to transfer memory from the GPU to the CPU at a much higher bandwidth than is currently possible, minimizing the time penalty for transferring data off the GPU. This will allow us to use the CPU for calculations that tune poorly to the GPUs. It would also make using the CPU memory for swap space feasible, meaning we could store and transfer grids off the GPU to the CPU, allowing several grids to functionally occupy one GPU.

Overall the GMHD code demonstrated the viability of using GPUs for RMHD. Future additions to the code will allow the full simulation of binary neutron star mergers much faster than existing codes. From these simulations possible gamma ray burst light curves paired with gravitational wave chirps can be computed. If these match gamma ray bursts and gravitational waves observed, then binaries would be confirmed as short hard gamma ray burst sources.

Bibliography

[1] J. M. Lattimer and M. Prakash, "The physics of neutron stars," Science **304,** 536–542 (2004).

[2] V. Kalogera and G. Baym, "The maximum mass of a neutron star," The Astrophysical Journal Letters **470,** L61 (1996).

[3] I. Bartos, P. Brady, and S. Marka, "How Gravitational-wave Observations Can Shape the Gamma-ray Burst Paradigm," Class. Quant. Grav. **30,** 123001 (2013).

[4] M. Anderson, E. Hirschmann, S. Liebling, and D. Neilsen, "Relativistic MHD with Adaptive Mesh Refinement," Class. Quantum Grav. **23,** 6503–6524 (2006).

[5] F. W. Glines, M. Anderson, and D. Neilsen, "Scalable Relativistic High-Resolution Shock-Capturing for Heterogeneous Computing," In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pp. 611–618 (2015).

[6] D. Neilsen, E. W. Hirschmann, and R. S. Millward, "Relativistic MHD and black hole excision: Formulation and initial tests,", Class.Quant.Grav. 23 (2006) S505, 2005.

[7] C.-W. Shu and S. Osher, "Efficient implementation of essentially non-oscillatory shock-capturing schemes," Journal of Computational Physics **77,** 439–471 (1988).

[8] R. MacCormack, "The effect of viscosity in hypervelocity impact cratering," Journal of spacecraft and rockets **40,** 757–763 (2003).

[9] P. Lax and B. Wendroff, "Systems of conservation laws," Communications on Pure and Applied mathematics **13,** 217–237 (1960).

[10] E. F. Toro, *Riemann solvers and numerical methods for fluid dynamics : a practical introduction* (Springer, Berlin, New York, 1997).

[11] P. Colella and P. R. Woodward, "The Piecewise Parabolic Method (PPM) for gas-dynamical simulations," Journal of Computational Physics **54,** 174–201 (1984).

[12] B. Einfeldt, "On Godunov-type methods for gas dynamics," SIAM Journal on Numerical Analysis **25,** 294–318 (1988).

[13] S. Osher and S. Chakravarthy, "Very high order accurate TVD schemes," Oscillation Theory, Computation, and Methods of Compensated Compactness, IMA Vol. Math. Appl **2,** 229–274 (1986).

[14] C. Bona, C. Bona-Casas, and J. Terradas, "Linear high-resolution schemes for hyperbolic conservation laws: TVB numerical evidence," Journal of Computational Physics **228,** 2266–2281 (2009).

[15] M. Harris, 2013.

[16] R. J. LeVeque, *Finite volume methods for hyperbolic problems*, *Cambridge texts in applied mathematics* (Cambridge University Press, Cambridge, New York, 2002), autres tirages : 2003, 2005, 2006.

[17] P. Wang, T. Abel, and R. Kaehler, "Adaptive mesh fluid simulations on GPU," New Astronomy **15,** 581 – 589 (2010).