UNIVERSITY OF COPENHAGEN

MASTER THESIS

Porting DISPATCH MHD to GPU Using Directive-Based Programming

Author:
Michael HAAHR

Supervisors:
Troels Haugbølle
Åke Nordlund

A thesis submitted in fulfillment of the requirements for the degree of Master of Science in Physics

in the

Centre for Star and Planet Formation
Niels Bohr Institute

June 1, 2021

Declaration of Authorship

I, Michael HAAHR, declare that this thesis titled, "Porting DISPATCH MHD to GPU Using Directive-Based Programming" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: [illegible]

Date: 1-6-2021

UNIVERSITY OF COPENHAGEN

Abstract

Faculty of Science
Niels Bohr Institute

Master of Science in Physics

Porting DISPATCH MHD to GPU Using Directive-Based Programming

by Michael HAAHR

The use of directive-based programming for porting HydroDynamics (HD) and MagnetoHydroDynamics (MHD) computations to the GPU has been investigated. Both OpenMP and OpenACC HD implementations have been tested, but only OpenMP has been used for MHD. The use of asynchronous execution and pinned memory in OpenMP is not supported by the GCC compiler, and it was not possible to get a running version with the Cray or LLVM compiler. To hide overhead and keep the GPU busy it was found necessary to bunch together multiple patches. With bunching, comparing 1 GPU to 40 CPU cores gave a speedup of around 3 for the MHD implementation. The MHD implementation has been integrated into DISPATCH as a new type of solver. A module for bunching DISPATCH tasks has likewise been added. The new solver, MHD_Bunch, has been validated with a 1D MHD shock tube experiment and the Orszag-Tang Vortex experiment. The MHD_Bunch GPU implementation is around 1.7 times faster than the MHD_Bunch CPU implementation, and around 2.9 times faster than DISPATCH/RAMSES.

Overall, the goal of porting DISPATCH MHD solver to GPU has been a success and a production-ready solver and bunching module have been implemented into DISPATCH.

Acknowledgements

First and foremost, I would like to thank my supervisors, Troels Haugbølle and Åke Nordlund, for providing guidance and assistance throughout the project. Their help laid the foundation, both technical and theoretical, on which the thesis is built.

I would also like to give special thanks to Sven Karlson, for the invaluable insight into fine details of GPU hardware and programming.

I would also like to acknowledge the assistance of the Danish e-infrastructure Cooperation (DeIC) especially in the early phases of the project. Specifically, I would like to thank René Løwe Jacobsen and Tethys Svensson.

Lastly, I would like to acknowledge access through a PRACE preparatory allocation to Piz Daint at CSCS, Switzerland, which allowed me to test the code in a Cray environment.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

PDE	**P**artial **D**ifferential **E**quation
IC	**I**nitial **C**condition
MHD	**M**agneto**H**ydro**D**ynamics
HLL	**H**arton **L**ax and van **L**eer
HLLC	**H**arton **L**ax and van **L**eer **C**ontact
HLLD	**H**arton **L**ax and van **L**eer **D**iscontinuities
MUSCL	**M**onotonic **U**pstream-centered **S**cheme for **C**onservation **L**aws
EMF	**E**lectro**M**otive **F**orce
CPU	**C**entral **P**rocessing **U**nit
FLOPS	**F**loating **P**oint **O**perations **P**er **S**econd
NUMA	**N**on-**U**niform **M**emory **A**ccess
GPU	**G**raphics **P**rocessing **U**nit
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
HIP	**H**eterogeneous-Computing **I**nterface for **P**ortability
API	**A**pplication **P**rogramming **I**ntrface
GPGPU	**G**eneral **P**urpose **GPU**
SM	**S**treaming **M**ultiprocessor
SIMT	**S**ingle **I**nstruction **M**ultiple **T**hread
OpenMP	**Open** **M**ulti-**P**rocessing
OpenACC	**Open** **ACC**elerators
LB	**L**eft **B**ottom
LT	**L**eft **T**op
RB	**R**ight **B**ottom
RT	**R**ight **T**op

List of Symbols

ρ	density	$\mathrm{kg\,m^{-3}}$
u	x velocity	$\mathrm{m\,s^{-1}}$
v	y velocity	$\mathrm{m\,s^{-1}}$
w	z velocity	$\mathrm{m\,s^{-1}}$
px	x momentum	$\mathrm{kg\,m\,s^{-1}}$
py	y momentum	$\mathrm{kg\,m\,s^{-1}}$
pz	z momentum	$\mathrm{kg\,m\,s^{-1}}$
p	pressure	$\mathrm{kg\,m^{-1}\,s^{-2}}$
e	energy	$\mathrm{kg\,m^{2}\,s^{-2}}$
B_x or A	x magnetic field flux	$\mathrm{kg\,A^{-1}\,s^{-2}}$
B_y or B	y magnetic field flux	$\mathrm{kg\,A^{-1}\,s^{-2}}$
B_z or C	z magnetic field flux	$\mathrm{kg\,A^{-1}\,s^{-2}}$
E_x	x edge-averaged electromotive force	$\mathrm{kg\,m\,A^{-1}\,s^{-3}}$
E_y	y edge-averaged electromotive force	$\mathrm{kg\,m\,A^{-1}\,s^{-3}}$
E_z	z edge-averaged electromotive force	$\mathrm{kg\,m\,A^{-1}\,s^{-3}}$
c^+ or c^-	courant number	dimensionless
c_a	Alfvén velocity	$\mathrm{m\,s^{-1}}$
c_s	slow magneto acoustic velocity	$\mathrm{m\,s^{-1}}$
c_f	fast magneto acoustic velocity	$\mathrm{m\,s^{-1}}$

Chapter 1

Introduction

Numerical simulations have long been a key source of knowledge in astrophysics. Simulations have greatly increased our understanding of everything from cosmology scales down to planet formation. Particularly, the application of the equations of magnetohydrodynamics (MHD) has given key insight into non-trivial physical systems.

The methods used in MHD have become more and more sophisticated and have constantly pushed the limit of the available hardware at any given time. Different techniques have been employed to utilize the available compute resources. Particularly, the block-based mesh and block-based adaptive mesh refinement (AMR; Berger and Oliger, 1984) techniques are widespread. Both block-based mesh and AMR can be almost-trivially converted to run in parallel on different compute nodes by dividing the mesh into smaller blocks and distributing the blocks.

However as supercomputers are becoming ever larger, these methods face issues with scalability. DISPATCH is a task-based simulation framework that aims to solve the scalability problem, which most simulation frameworks face today. Unlike standard meshed, which has a global timestep, each `block` (or task) in DISPATCH has its own timestep. This means that large-timestep blocks are not slowed down by small-timestep blocks, increasing the overall performance. MHD may be solved near-independently in each block, with the only dependency being on neighboring blocks. This reduces overall communication and allows for better dynamic load balancing for both inter- and intra-node. DISPATCH has been shown to possess near-optimal weak and strong scaling (Nordlund et al., 2018).

With the exascale supercomputer about to come online, frameworks like DISPATCH will become essential for large simulations. In the next-generation supercomputers such as LUMI[1] and FRONTIER[2], the majority of the compute power lies in accelerators (GPU). To run efficiently on these machines DISPATCH must be adapted to use GPUs. GPUs have extremely high throughput but require a lot of work per memory transfer to be favored over standard CPU execution. MHD codes such as RAMSES (Teyssier, 2002) are computationally intensive and are therefore ideal for GPU execution.

Accelerating computations with the GPU is an active research topic. The GenASIS code (Budiardja and Cardall, 2019) was accelerated using a mix of Fortran and C. FARGO3d (Benitez-Llambay and Masset, 2016) is a fully GPU-based framework written solely in CUDA. Most recently several approaches for GPU accelerated were investigated in the GENE code (Germashewski et al., 2021), which ultimately decided on using CUDA C++.

Both GenASIS and GENE are written in Fortran, but use CUDA/C or CUDA/C++ to offload code to the GPU. GENE (Germashewski et al., 2021) did try using

[1]https://www.lumi-supercomputer.eu/

[2]https://www.olcf.ornl.gov/frontier/

only OpenACC and OpenMP but did not pursue this due to limited compiler support at the start of their project. Similar conclusions were made in the early stages of this thesis, but with the newest GCC version, OpenMP offload saw much better support. In this thesis, the use of OpenMP, and briefly OpenACC, is examined as a means to port the DISPATCH/RAMSES solver to GPUs. An almost complete refactoring of the code was found necessary as large and complicated functions are not well suited for GPU execution.

It was found necessary to bunch several tasks together and execute all at the same time for the GPU version to be favorable. In addition to the solver, a bunching module has been added to the DISPATCH framework. Simulations using both GPU solver and bunching-module have been tested and validated to produce the same result down to machine precision for a single timestep. Two experiments have been carried out the validate the solver: A 1D MHD shock tube (Ryu and Jones, 1995) and the Orszag-Tang Vortex (Orszag and Tang, 1979).

The thesis is divided as follows. Chapter 2 gives an overview of the physical theory and numerical methods of hydrodynamics and magnetohydrodynamics. Chapter 3 gives a detailed description of DISPATCH and its task-based structure, which separates it from other frameworks. In Chapter 4 and 5 the fundamentals of CPU and GPU architecture are explained, as well as the basics of directive-based programming. Chapter 6 goes through the various changes made to the DISPATCH/RAMSES MHD code to get it GPU ready. Chapter 7 goes through the results showing both correctness and speedup of the GPU version as compared to the existing code. Chapter 8 comments on the current implementation as well as possible future extensions and compares the results to other works. Lastly, chapter 9 gives the conclusion of the thesis.

Chapter 2

Theory

2.1 Hydrodynamics

Hydrodynamics is the study of liquids in motion. In the context of astrophysics, *liquid* is used rather broadly, to also include gas. This section focuses on the governing equations of hydrodynamics and the numerical methods used in the thesis. The description and figures are based largely on Toro (2009), and follows the notation given there.

A hydrodynamic system is, in the simplest sense, described by the Euler equations for conserved variables. The conserved variables are density, momentum (x,y,z), and energy:

$$\rho_t + (\rho u)_x + (\rho v)_y + (\rho w)_z = 0 \tag{2.1}$$

$$(\rho u)_t + (\rho u^2 + p)_x + (\rho u v)_y + (\rho u w)_z = 0 \tag{2.2}$$

$$(\rho v)_t + (\rho u + v)_x + (\rho v^2 + p)_y + (\rho v w)_z = 0 \tag{2.3}$$

$$(\rho w)_t + (\rho u + w)_x + (\rho v w)_y + (\rho w^2 + p)_z = 0 \tag{2.4}$$

$$E_t + [u(E+p)]_x + [v(E+p)]_y + [w(E+p)]_z = 0 \tag{2.5}$$

Where u, v, and w are the velocities in the x-, y-, and z-directions, E is the total energy, p is the pressure and ρ is the density. The subscripts denote derivatives of the respective variables; t, x, y, and z. The total energy is given by $E = \rho(\frac{1}{2}V^2 + e)$, where $V^2 = u^2 + v^2 + w^2$ and e is the internal energy per unit mass.

The laws in equations 2.1-2.5 relates the change over time in mass, momentum, and energy to the changes in space. The equations all sum to zero, as all changes in time must be caused by a displacement in space, e.i., they are conserved. In some cases, a `source` term might be added to the right side to factor in any work done by some outside force. This is however not considered in the thesis.

The equations are often written in vector form, which can simplify many expressions. The vector form is shown below and will be used in the rest of the section.

$$\mathbf{U_t} + \mathbf{F}(\mathbf{U})_\mathbf{x} + \mathbf{G}(\mathbf{U})_\mathbf{y} + \mathbf{H}(\mathbf{U})_\mathbf{z} = \mathbf{0} \tag{2.6}$$

Where:

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ E \end{bmatrix}, \mathbf{F}(\mathbf{U}) = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho u v \\ \rho u w \\ u(E+p) \end{bmatrix}, \mathbf{G}(\mathbf{U}) = \begin{bmatrix} \rho v \\ \rho v u \\ \rho v^2 + p \\ \rho v w \\ v(E+p) \end{bmatrix}, \mathbf{H}(\mathbf{U}) = \begin{bmatrix} \rho w \\ \rho w u \\ \rho w v \\ \rho w^2 + p \\ w(E+p) \end{bmatrix} \tag{2.7}$$

Throughout this chapter **F** and **F(U)** is used interchangeably.

2.1.1 Finite Volume Method

Finite volume methods are a class of methods used for evaluating partial differential equations with the use of volume and surface integrals. In order to use the finite volume method, equation 2.6 is put in integral form. For the 1D case, this can be written as:

$$\oint [\mathbf{U}dx - \mathbf{F}(\mathbf{U})dt] = \mathbf{0} \tag{2.8}$$

A control volume can be defined in space and time. If the control volume is defined as $V = [x_L, x_R] \times [t_1, t_2]$ the equation can be further simplified. The control volume given here is a rectangle as shown in figure 2.1. Under this assumption equation 2.8 becomes:

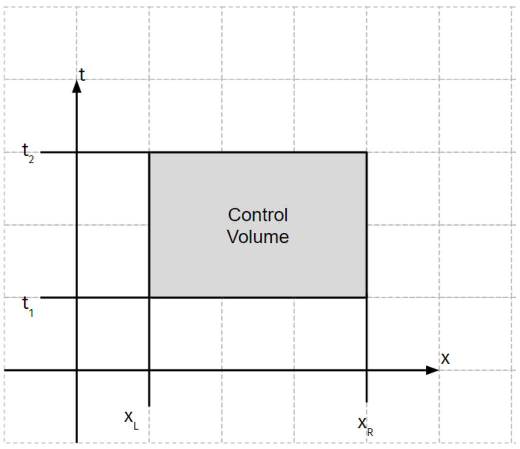


FIGURE 2.1: Control volume of 1D conservation laws

$$\int_{x_L}^{x_R} \mathbf{U}(x,t_2)dx = \int_{x_L}^{x_R} \mathbf{U}(x,t_1)dx + \int_{t_1}^{t_2} \mathbf{F}(\mathbf{U})(x_L,t)dt - \int_{t_1}^{t_2} \mathbf{F}(\mathbf{U})(x_R,t)dt \tag{2.9}$$

This equation states that any control volume can be updated by calculating the net flux over the surface area of the volume. The method holds in higher dimensions.

With equation 2.9 in mind, any 1D experiment can be updated in time by splitting the space into M finite volumes. Each grid point will hold the volume average value for mass, momentum, and energy. This allows us to simplify 2.9 to:

$$\mathbf{U}_x^{t+1} = \mathbf{U}_x^t + \mathbf{F}(\mathbf{U})_{x-\frac{1}{2}} - \mathbf{F}(\mathbf{U})_{x+\frac{1}{2}} = \mathbf{U}_x^t + \mathbf{F}(\mathbf{U})_L - \mathbf{F}(\mathbf{U})_R \tag{2.10}$$

The fluxes going into the volume elements are calculated at the interface of the grid cell. Thus, the i'th grid cell will have $\mathbf{F}(\mathbf{U})_L = \mathbf{F}(\mathbf{U}_{\mathbf{i}-\frac{1}{2}})$ and $\mathbf{F}(\mathbf{U})_R = \mathbf{F}(\mathbf{U}_{\mathbf{i}+\frac{1}{2}})$.

The basic idea of average values, cell center, and interfaces are shown in figure 2.2. The problem in the finite volume method is thus reduced to finding the fluxes in and out of each grid cell for each time step. The methods used for this are the so-called Riemann Solvers, described in the following sections

2.1.2 Riemann Problem

The setup shown in figure 2.2 gives rise to several Riemann problems. A Riemann problem is a special group of partial differential equations and initial conditions for which follows:

$$\begin{aligned} \mathbb{PDE}: \quad & \mathbf{U}_t + \mathbf{A}\mathbf{U}_x = \mathbf{0} \\ \mathbb{IC}: \quad \mathbf{U}(x,0) = \mathbf{U}^0(x) = & \begin{cases} \mathbf{U_L} & \text{if} \quad x < 0 \\ \mathbf{U_R} & \text{if} \quad x > 0 \end{cases} \end{aligned} \tag{2.11}$$

The structure of the Riemann problem is shown in figure 2.3 for a single PDE, whereas equation 2.11 is written for multiple PDEs. Figure 2.3b shows 1 wave emanating from the origin with a characteristic speed, a. The example here is that of

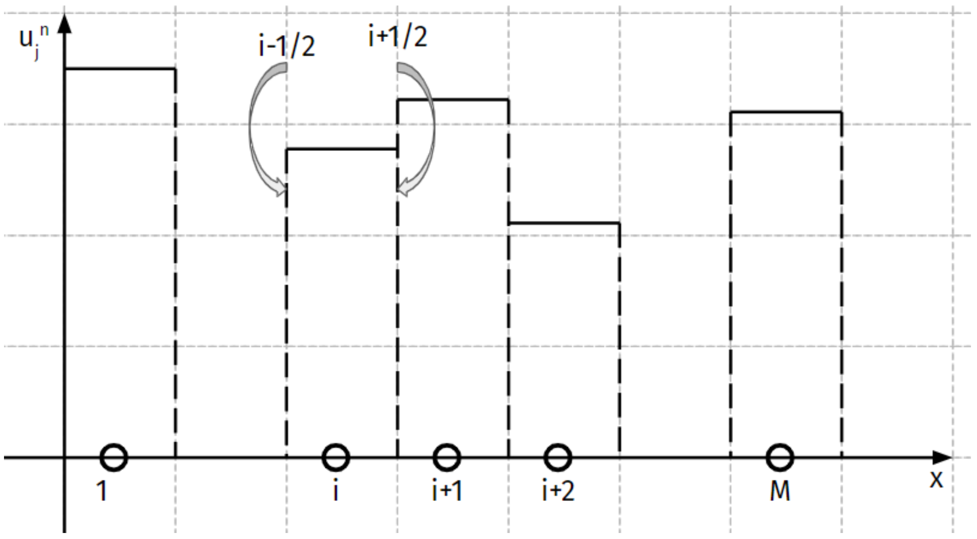
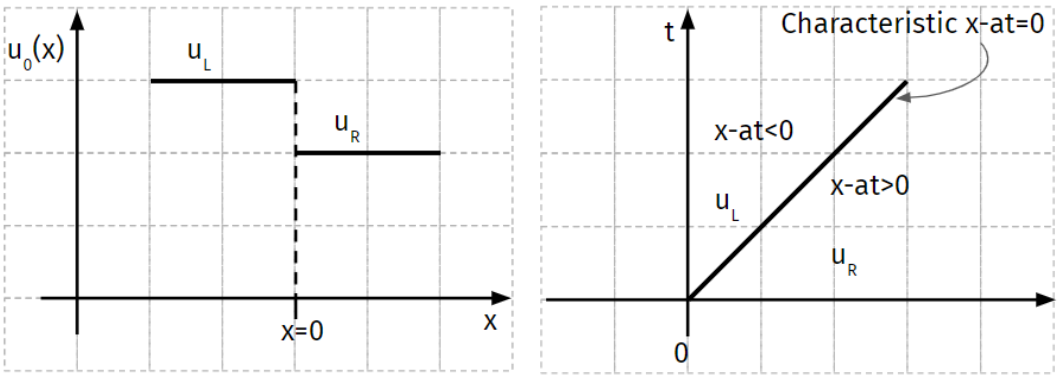


FIGURE 2.2: Average values for each grid point. Dashed line represent interface between cells.

linear advection where the characteristic speed is a constant and does not change depending on the value of **U**. In the general case, there will be multiple nonlinear waves propagating from the origin. The speed and nature of the waves are discussed below.



(A) Initial condition for the Riemann problem (B) Wave propagation for the Riemann problem

FIGURE 2.3: Riemann problem and characteristic wave

2.1.3 Wave Speed and Characteristic Variables

The matrix **A** in equation 2.11, which describes the type of waves can be expressed as:

$$\mathbf{A} = \mathbf{K}\mathbf{\Lambda}\mathbf{K}^{-1} \tag{2.12}$$

Where $\mathbf{\Lambda}$ is a diagonal matrix of the eigenvalues, λ_i, and the columns in **K** are the corresponding eigenvectors. For a system with m entries in **U**, this gives rise to m eigenvalues. Each eigenvalue represents a wave propagating from the origin with a speed of λ_i. The structure of the general solution is seen in figure 2.4a.

The introduction of **K** allows us to define a new set of variables, **W**, for which:

$$\mathbf{W} = \mathbf{K}^{-1}\mathbf{U} \quad \text{or} \quad \mathbf{U} = \mathbf{K}\mathbf{W} \tag{2.13}$$

Substituting this into equation 2.11 gives a new set of PDEs:

$$\mathbf{K}\mathbf{W}_t + \mathbf{A}\mathbf{K}\mathbf{W}_x = \mathbf{0} \tag{2.14}$$

By multiplication of $\mathbf{K}^{-1}$ this becomes:

$$\mathbf{W}_t + \mathbf{\Lambda}\mathbf{W}_x = \mathbf{0} \tag{2.15}$$

This equation is referred to as the *canonical* or *characteristic* form of the system. The characteristic form becomes a set of decoupled partial differential equations of the form:

$$\frac{\partial w_i}{\partial t} + \lambda_i \frac{\partial w_i}{\partial x} = 0 \tag{2.16}$$

The left and right initial conditions may be expanded as a linear combination of the eigenvectors. This can be used to write the general solution at any position. The left and right initial condition can be written as:

$$\mathbf{U_L} = \sum_{i=1}^{m} \alpha_i \mathbf{K}^{(i)}, \quad \mathbf{U_R} = \sum_{i=1}^{m} \beta_i \mathbf{K}^{(i)} \tag{2.17}$$

Initial data can be obtained from setting $w_i^{(0)}(x) = \alpha_i$ if $x < 0$ and $w_i^{(0)}(x) = \beta_i$ if $x > 0$. From this, the solution at a later time can be found using

$$w_i(x,t) = w_i^{(0)}(x - \lambda_i t) = \begin{cases} \alpha_i & \text{if} \quad x - \lambda_i t < 0 \\ \beta & \text{if} \quad x - \lambda_i t > 0 \end{cases} \tag{2.18}$$

Thus, for a given coordinate (x,t), there is an eigenvalue λ_I such that $\lambda_I < \frac{x}{t} < \lambda_{I+1}$. This means that for all $i \leq I$ it holds that $x - \lambda_i t < 0$ and for all $i > I$ it holds that $x - \lambda_i t > 0$. In terms of the original variables, α and β, and I the general solution be be written as:

$$\mathbf{U}(x,t) = \sum_{i=I+1}^{m} \alpha_i \mathbf{K}^{(i)} + \sum_{i=1}^{I} \beta_i \mathbf{K}^{(i)} \tag{2.19}$$

For a system with only 2 variables, the result corresponds to what is shown in figure 2.4b. Here $\mathbf{U_L}$ and $\mathbf{U_R}$ can be written in terms of α and β as:

$$\mathbf{U_L} = \alpha_1 K^{(1)} + \alpha_2 K^{(2)}, \quad \mathbf{U_R} = \beta_1 K^{(1)} + \beta_2 K^{(2)} \tag{2.20}$$

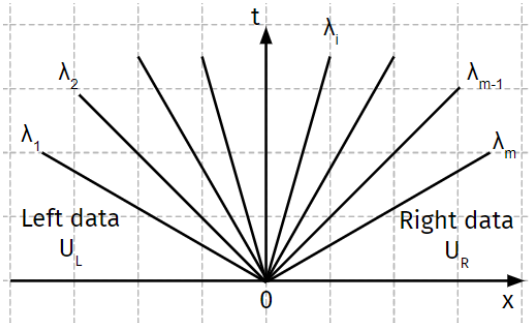
These can be combined to find the values in the star region

$$\mathbf{U}^*(x,t) = \beta_1 K^{(1)} + \alpha_2 K^{(2)} \tag{2.21}$$

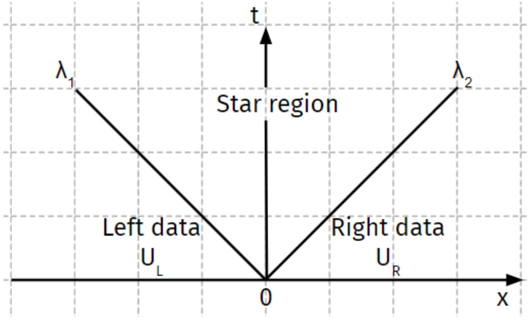
Given $\mathbf{U}$ and $\mathbf{A}$, $\mathbf{U}^*(x,t)$ can be uniquely found. A solution for any time at $x = 0$ can then be derived as:

$$\mathbf{U}(0,t) = \begin{cases} \mathbf{U_L} \text{ if} & \lambda_1 t > 0 \\ \mathbf{U}^* \text{ if} & \lambda_1 t < 0 < \lambda_2 t \\ \mathbf{U_R} \text{ if} & \lambda_2 t < 0 \end{cases} \tag{2.22}$$

The solution to the Riemann problem with 2 variables thus becomes a matter of calculating the wave speeds, and choosing the pre-defined solution ($\mathbf{U_L}$, $\mathbf{U}^*$ or $\mathbf{U_R}$) accordingly. This solution is the basic notion of the HLL-solver, discussed in section 2.1.5.



(A) Structure of the Riemann problem for m variables



(B) Riemann problem for a system of 2 variables. Riemann solution found at $x = 0$. the solution is here in the *Star region*

FIGURE 2.4: Riemann structure for M variables and solution for 2 variables.

2.1.4 Wave Solutions

So far, only the case of linear advection has been considered, where a wave travels at a constant speed, which does not change depending on the value of $\mathbf{U}$. In the non-linear case where the characteristic speed is a function of $\mathbf{U}$, distortions in the wave are produced. Consider the case of a single variable for which the characteristic speed is a convex function of the value $\mathbf{u}$. Here $\mathbf{u}$ is used to denote that it is only a single value and not a vector. Such a setup is illustrated in figure 2.5. As the figure shows, two distinct regions occur.

The expansive region from x_1 to x_4 occurs because the flux characteristic at x_2 is larger than at x_1, the flux characteristic at x_3 is larger than at x_2 and flux characteristic at x_4 is larger than at x_3. The value of u_4 will thus propagate faster than u_3, which will propagate faster than u_2, which will propagate faster than u_1. This will result in a broader and flatter region.

The compressive region from x_4 to x_7 occurs for the opposite reasons as the expansive region. In the compressive region, u_7 propagate slower than u_6, which propagate slower than u_5 etc. This region will become narrower and steeper.

The following subsections describe the three fundamental wave types. $\mathbf{u_L}$ and $\mathbf{u_R}$ are scalar variables with the flux assumed to be convex.

Shock Waves

Shock waves are small transition layers of very rapid change in physical quantities. Because the transition layers are physically small they are very well represented as mathematical/numerical discontinuities. Figure 2.6a shows the setup resulting in a shock wave. The values of $\mathbf{u_L}$ are higher than $\mathbf{u_R}$. $\mathbf{u_L}$ will thus propagate faster than $\mathbf{u_R}$. This is the extreme case of the compressive region discussed in the previous section. The overlap of characteristics will result in a shock wave. The shock wave propagates with a speed S. Shock waves satisfy the *Rankine-Hugoniot condition*. For the scalar case, this is

$$S = \frac{\Delta f}{\Delta u} \tag{2.23}$$

where $\Delta f = f(\mathbf{u_R}) - f(\mathbf{u_L})$ and $\Delta u = \mathbf{u_R} - \mathbf{u_L}$. Additionally, shock waves satisfy the *entropy condition*:

$$\lambda(\mathbf{u_L}) > S > \lambda(\mathbf{u_R}) \tag{2.24}$$

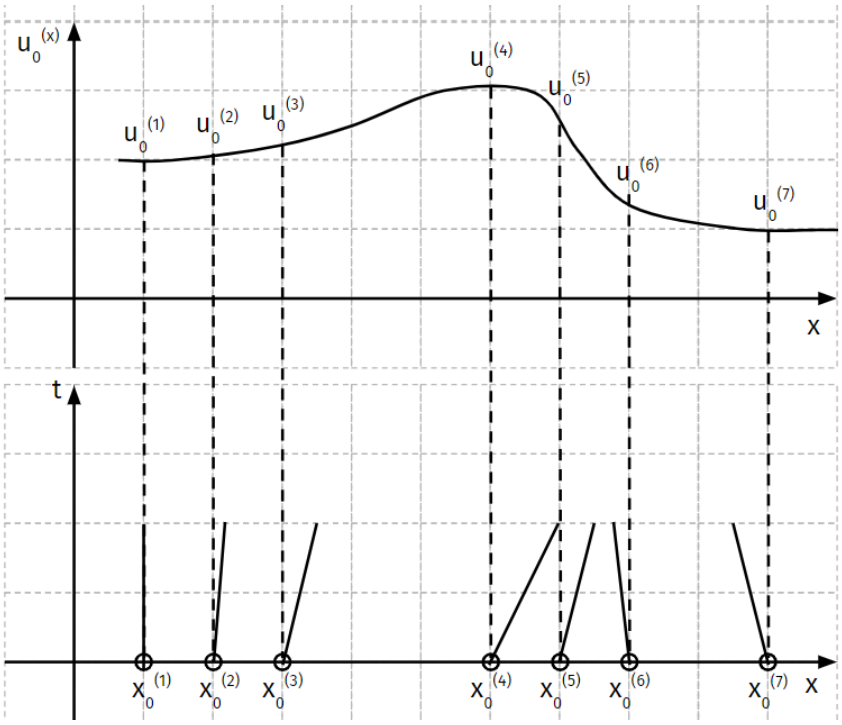


FIGURE 2.5: Relation between flux and value for a convex flux function. The upper part shows the initial condition and the lower part shows the corresponding flux characteristics for the different values. The region between $x_0^{(1)}$ and $x_0^{(4)}$ is expansive and the region between $x_0^{(4)}$ and $x_0^{(7)}$ is compressive.

That is, the characteristics to the left of the shock waves are larger than those to the right. The solution to a shock wave becomes

$$\mathbf{u}(x,t) = \begin{cases} \mathbf{u_L} & \text{if} \quad x - St < 0 \\ \mathbf{u_R} & \text{if} \quad x - St > 0 \end{cases} \tag{2.25}$$

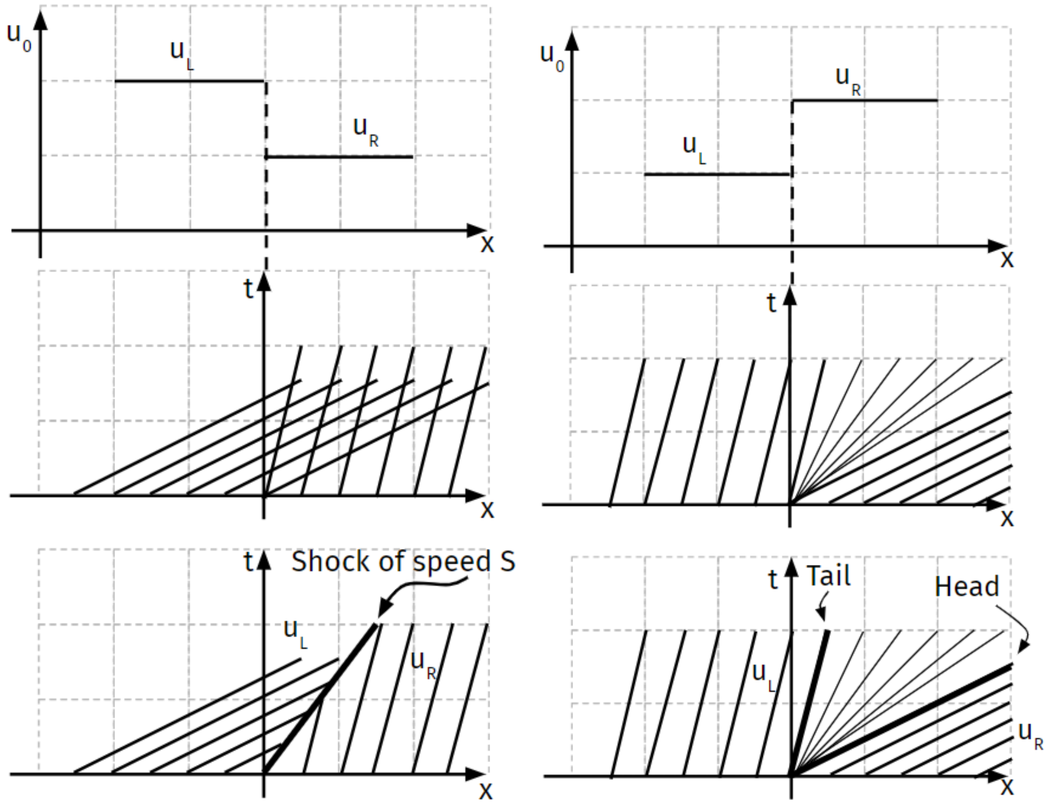
Rarefaction Waves

Rarefaction *shocks* occur when $\mathbf{u_L} < \mathbf{u_R}$ and the entropy condition from shock waves are violated. The setup can be seen at the top of figure 2.6b. It is mathematically correct to keep the same solution from equation 2.25, but the solution is not physically correct and is therefore not used.

The setup here is the extreme case of the expansive wave, and one would therefore expect a more smooth transition, rather than a shock. This smooth region will slowly expand. The bottom of figure 2.6b shows the Tail and Head of the expansion. To the left of the Tail, the solution will be $\mathbf{u_L}$ and to the right of the Head to the solution will be $\mathbf{u_R}$. The speed of the Head will be given by the characteristics of the right side, λ_R. and the speed of the Tail will be given by the characteristics of the left side, λ_L.

In between the Head and Tail, there will be a family of waves that takes on all values between λ_L and λ_R. The solution in this region will thus depend on the position, $\mathbf{u}(x,t)$, between the Head and Tail. The general solution for inviscid flow with convex flux becomes:

$$\mathbf{u}(x,t) = \begin{cases} \mathbf{u_L} & \text{if} \quad \frac{x}{t} < \lambda_L \\ \frac{x}{t} & \text{if} \quad \lambda_L < \frac{x}{t} < \lambda_R \\ \mathbf{u_R} & \text{if} \quad \frac{x}{t} > \lambda_R \end{cases} \tag{2.26}$$



(A) From top to bottom: Initial condition, characteristics, and resulting shock wave.

(B) From top to bottom: Initial condition, characteristics, and resulting rarefaction waves.

FIGURE 2.6: Shock and rarefaction waves for convex flux function.

Contact Waves

Contact waves (or discontinuities) are waves for which both the Rankine-Hugoniot condition and the Generalised Riemann Invariants hold and that have parallel characteristics:

$$\lambda_L = \lambda_R = S \tag{2.27}$$

The Generalised Riemann Invariants are relations between the changes in w_i and the respective right eigenvector component k_i. Here w_i and k_i refer to the characteristic variables described in section 2.1.3. For the Generalised Riemann Invariants to hold the following must hold:

$$\frac{dw_1}{k_1^{(i)}} = \frac{dw_2}{k_2^{(i)}} = ... = \frac{dw_m}{k_m^{(i)}} \tag{2.28}$$

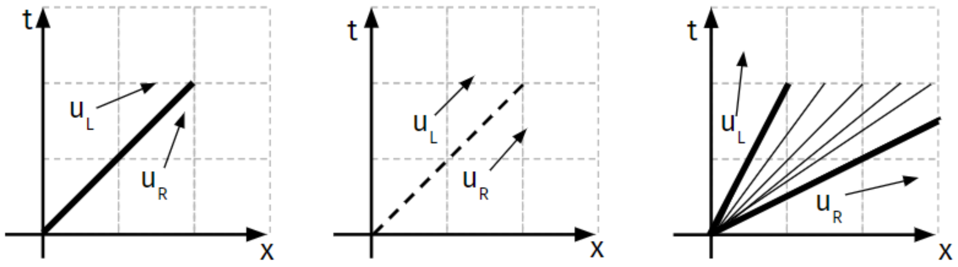


FIGURE 2.7: Shock wave, contact discontinuity, and rarefaction wave. Arrows indicate the direction of characteristics on either side of the wave.

Waves summary

Three different types of solutions representing different waves can occur in the Riemann Problem. The solutions are outlined in figure 2.7. In the previous sections, only the scalar case has been discussed. However, the wave types extend to vector cases with $\mathbf{U_L}$ and $\mathbf{U_R}$ where the characteristic of each wave is given by λ_i from equation 2.16. For a system of m variables, each of the m different waves may be any of the three types.

2.1.5 Riemann Solvers

The goal of Riemann Solvers is to provide a computationally feasible solution to the general Riemann problem. For the thesis, only the solvers based on the Harten Lax and van Leer (HLL) method are considered. These methods include the later extensions such as the HLLC and HLLD. The HLL, HLLC and HLLD are methods for estimating the flux at the interface (figure 2.2) between cells. This flux is referred to as the Godunov flux.

Godunov Method

Before introducing the Godunov method, a quick recap of the numerical first-order upwind scheme is needed, as the Godunov method is an extension to this scheme.

Partial derivatives of PDE's are often calculated numerically on either side of the cell:

$$\mathbf{u_x} = \frac{\mathbf{u_i^t} - \mathbf{u_{i-1}^t}}{\Delta x} \tag{2.29}$$

or

$$\mathbf{u_x} = \frac{-\mathbf{u_i^t} + \mathbf{u_{i+1}^t}}{\Delta x} \tag{2.30}$$

Each of these is a viable mathematical approach. Depending on the sign of the wave propagation, sign(a), one of the solutions will be unstable and the other will be stable. The stable solution will be the *upwind* scheme. For positive a equation 2.29 is the upwind scheme and for negative a, it will be equation 2.30. When updating a cell from time t to time $t+1$ the two equations can be combined into:

$$\mathbf{u_i^{t+1}} = \mathbf{u_i^t} - c^+(\mathbf{u_i^t} - \mathbf{u_{i-1}^t}) - c^-(-\mathbf{u_i^t} + \mathbf{u_{i+1}^t}) \tag{2.31}$$

where the Courant numbers c^+ and c^- are based on wave speeds:

$$c^+ = \frac{\Delta t a^+}{\Delta x}, \quad c^- = \frac{\Delta t a^-}{\Delta x} \tag{2.32}$$

where $a^+ = a$ and $a^- = 0$ if $a \geq 0$, $a^+ = 0$ and $a^- = a$ if $a < 0$. Choosing a time step such that $0 < |c^{\pm}| < 1$ ensures that the scheme is conditionally stable.

While the above upwind scheme is stable, it is not conservative. Equation 2.31 can be changed to instead use intercell flux, which makes it conservative. For a PDE,

$$\mathbf{U}_t + \mathbf{F}(\mathbf{U})_x = 0 \tag{2.33}$$

the conservative numerical method becomes

$$\mathbf{U}_i^{t+1} = \mathbf{U}_i^t + \frac{\Delta t}{\Delta x}(\mathbf{F}_{i-\frac{1}{2}} + \mathbf{F}_{i+\frac{1}{2}}) \tag{2.34}$$

The Godunov flux, $\mathbf{F}_{i-\frac{1}{2}}$, is an approximation of the physical flux. As explained in section 2.1.4, the flux is assumed to be dependent on the value of $\mathbf{U}$, i.e. $\mathbf{F}_{i-\frac{1}{2}} = \mathbf{F}(\tilde{\mathbf{U}}_{i-\frac{1}{2}})$. Here, $\tilde{\mathbf{U}}_{i-\frac{1}{2}}$ is the solution to the Riemann Problem with initial left and right states $\mathbf{U}_L = \mathbf{U}_{i-1}$ and $\mathbf{U}_R = \mathbf{U}_i$. In order to calculate the Godunov flux, the Riemann problem must be solved for both $i - \frac{1}{2}$ and $i + \frac{1}{2}$.

For the Euler equations, there will at each interface be 10 possible solutions to the Riemann problem as illustrated in figure 2.8. The Euler equations have 5 variables, which means 5 wave patterns. Here, only three are shown as the contact discontinuity wave has multiplicity 3.

HLL

The HLL Riemann solver was proposed in 1983 by Harten Lax and van Leer (Harten, Lax, and Leer, 1983). The purpose is to directly compute an approximate flux at the cell interface, $i \pm \frac{1}{2}$. The approximation is done by assuming that two waves separate the left and right states. Between the two waves the solution, $\mathbf{U}^*$, is assumed to be constant. The central idea is similar to the two-variable system sketched in figure 2.4b. The HLL solver approximates the fastest signal velocities S_L and S_R. How the values are approximated and how this affects the solver will be discussed in section 2.1.6. This creates a control volume for which the average value after a timestep can be calculated using integrals. This approach is sketched in figure 2.9. This is very similar to figure 2.1, but there are now 3 distinct regions instead of 1 at time T. These are the regions between $[x_L, TS_L]$, $[TS_L, TS_R]$ and $[TS_R, x_R]$. Note that a contact discontinuity is sketched but is not used to split the solution into additional regions.

The Godunov flux at $\frac{x}{t} = 0$ depends on which regions this falls under. If $S_L \geq 0$, the intercell flux is determined by the values in the left region,$[x_L, TS_L]$, or $\mathbf{F}(\mathbf{U_L})$. Similarly, if $S_R \leq 0$ the values lie in the $[TS_R, x_R]$ region and the flux becomes $\mathbf{F}(\mathbf{U_R})$. If $S_L < 0 < S_R$ the flux is determined by the variables in the star region, $[TS_L, TS_R]$. Putting this into a single equation the HLL flux, $\mathbf{F}^{hll}$, is given by:

$$\mathbf{F}^{hll} = \begin{cases} \mathbf{F}_L, & \text{if} \quad S_L \geq 0 \\ \frac{S_R\mathbf{F}_L - S_L\mathbf{F}_R + S_L S_R(\mathbf{U}_R \mathbf{U}_L)}{S_R - S_L}, & \text{if} \quad S_L < 0 < S_R \\ \mathbf{F}_R, & \text{if} \quad S_R \leq 0 \end{cases} \tag{2.35}$$

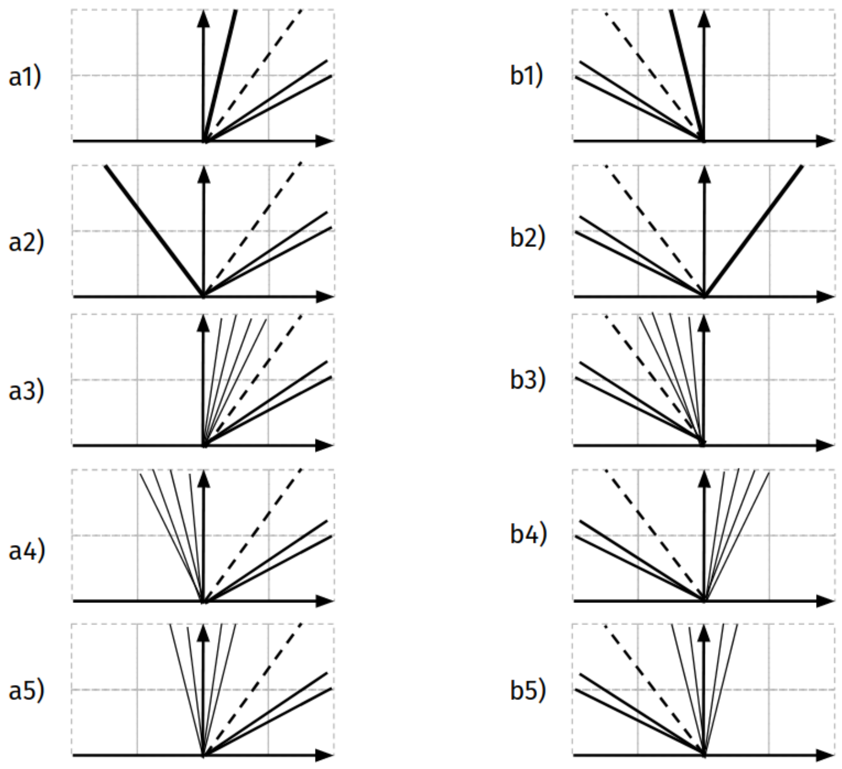


FIGURE 2.8: 10 different solutions for the Euler equations evaluated at $\frac{x}{t} = 0$ depending on the wave speeds. a1-5) shows solutions with positive particle speed and b1-5) shows solutions with negative particle speeds. Bold lines indicate shock, dashed lines indicate contact discontinuity, a fan of four lines indicates a rarefaction wave, and a pair of two lines indicates a wave of unknown character.

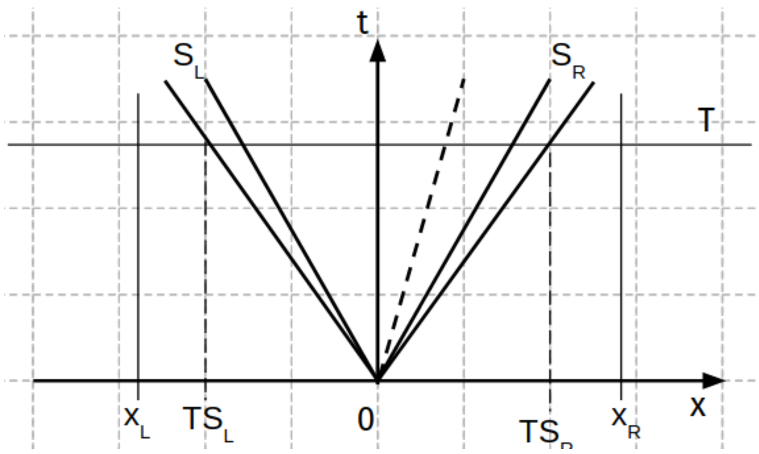


FIGURE 2.9: Control volume for HLL solver with signal velocities S_L and S_R. 3 distinct regions exists after time T: $[x_L, TS_L]$, $[TS_L, TS_R]$ and $[TS_R, x_R]$. Dashed lines indicate contact discontinuity and two lines indicate a wave of unknown characteristics.

where $\mathbf{F}_L = \mathbf{F}(\mathbf{U_L})$ and $\mathbf{F}_R = \mathbf{F}(\mathbf{U_R})$. The HLL flux can be calculated sequentially in each dimension, and the conservative property still holds.

The shortcoming of the HLL solver is in the assumption that the star region is constant. This assumption works well for shock waves, but does not properly evaluate rarefaction waves and tends to dissipate contact discontinuities.

HLLC

The HLLC is an extension to the HLL solver, which aims at solving the issue of the dissipation of contact discontinuities (Hence "C" for contact). In the HLLC approach, the star region is split into the left and right star region, $\mathbf{U}_{*L}$ and $\mathbf{U}_{*R}$. This creates 4 regions separated by 3 waves. The left and right most waves are referred to as S_L and S_R as in the HLL, and the middle wave is referred to as S_* or S_M. Equation 2.35 can be extended to

$$\mathbf{F}^{hllc} = \begin{cases} \mathbf{F}_L, & \text{if} \quad S_L \geq 0 \\ \mathbf{F}_{*L}, & \text{if} \quad S_L < 0 < S_* \\ \mathbf{F}_{*R}, & \text{if} \quad S_* < 0 < S_R \\ \mathbf{F}_R, & \text{if} \quad S_R \leq 0 \end{cases} \tag{2.36}$$

Where $\mathbf{F}_{*L} = \mathbf{F}(\mathbf{U}_{*L})$ and $\mathbf{F}_{*R} = \mathbf{F}(\mathbf{U}_{*R})$. For the Euler equations, the left and right fluxes in the star region in the x-direction are calculated by:

$$\mathbf{F}_{*L} = \frac{S_*(S_L\mathbf{U}_L - \mathbf{F}_L) + S_L(p_L + \rho_L(S_L - u_L)(S_* - u_L))D_*}{S_L - S_*} \tag{2.37}$$

$$\mathbf{F}_{*R} = \frac{S_*(S_R\mathbf{U}_R - \mathbf{F}_R) + S_R(p_R + \rho_R(S_R - u_R)(S_* - u_R))D_*}{S_R - S_*} \tag{2.38}$$

Where $D_* = [0, 1, 0, 0, S_*]$. D_* can be changed when calculating the differences in the y- and z-direction. The speed of the middle wave can be calculated only based on S_L and S_R and initial values:

$$S_* = \frac{p_R - p_L + \rho_L u_L(S_L - u_L) - \rho_R u_R(S_R - u_R)}{\rho_L(S_L - u_L) - \rho_R(S_R - u_R)} \tag{2.39}$$

2.1.6 Wave-Speed Estimates for HLL and HLLC

The estimates of S_L and S_R has a big impact on the performance of the algorithm. As described in Roe, 1981 the smallest, λ_1, and largest, λ_m, eigenvalues for the one-dimensional equivalent of equation 2.6 is:

$$S_L = \lambda_1 = \tilde{u} - \tilde{a}\,, \quad S_R = \lambda_m = \tilde{u} + \tilde{a} \tag{2.40}$$

where $\tilde{u}$ and $\tilde{a}$ is the Roe-averaged particle and sound speed:

$$\tilde{u} = \frac{\sqrt{\rho_L} u_L + \sqrt{\rho_R} u_R}{\sqrt{\rho_L} + \sqrt{\rho_R}}\,, \quad \tilde{a} = \left((\gamma - 1)(\tilde{H} - \frac{1}{2}\tilde{u})^2 \right) \tag{2.41}$$

where enthalpy $H = \frac{E+p}{\rho}$ may be approximated by $\tilde{H} = \frac{\sqrt{\rho_L} H_L + \sqrt{\rho_R} H_R}{\sqrt{\rho_L} + \sqrt{\rho_R}}$.

A simpler approach was suggested by Davis, 1988 where S_L and S_R are based solely on data values:

$$S_L = u_L - a_L\,, \quad S_R = u_R + a_R \tag{2.42}$$

or

$$S_L = min(u_L - a_L, u_R - a_R)\,, \quad S_R = max(u_R + a_R, u_L + a_L) \tag{2.43}$$

where a_α is the sound speed for ideal gas, calculated by $a_\alpha = \sqrt{\frac{\gamma p_\alpha}{\rho_\alpha}}$. Here the subscript, α, denotes either left or right subscript.

The best estimate of left and right wave speed depends on the experiment. Depending on the choice, some solver may fail and predict non-physical negative density. Einfeldt et al., 1991 introduced a subclass of solvers called *positively conservative*, which are guaranteed to produce physical results. Einfeldt et al., 1991 further argues that equation 2.43 is positivily conservative for the HLL solver. Similarly, Batten et al., 1997 argues that the following wave speed estimates make the HLLC solver positivity conservative:

$$S_L < u_L - \sqrt{\frac{\gamma - 1}{2\gamma}} a_L\,, \quad S_R > u_R - \sqrt{\frac{\gamma - 1}{2\gamma}} a_R \tag{2.44}$$

where again, $a_\alpha = \sqrt{\frac{\gamma p_\alpha}{\rho_\alpha}}$.

2.2 MUSCL Type Solvers

The previous sections described the hydrodynamical laws and how to update in a grid-based system. Each grid cell was assumed to have a constant physical value, which leads to the Riemann problem at the interface of each grid cell. In reality, the physical variables will not be constant over the entire cell. This is illustrated in figure 2.10. Here the true value of some variable is shown in red and the integral average is shown by horizontal lines at each grid cell. As the figure shows some interface values are poorly described by the cell average. Better values, therefore, need to be extrapolated to be used by the Riemann Solvers.

Leer, 1979 introduced the *Monotonic Upstream-centered Scheme for Conservation Laws* (MUSCL) scheme for extrapolating interface values and update cell-centered values. The scheme employs a leapfrog-like method. MUSCL type solvers make predictions at $t + \frac{1}{2}\Delta t$. These values are then used in the Riemann problem to find the flux at time $t + \frac{1}{2}\Delta t$. This flux is then used to update the values at time t to

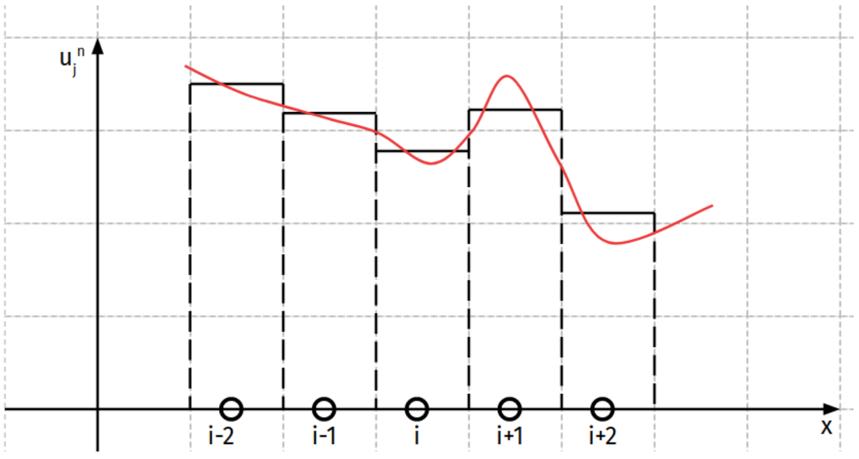


FIGURE 2.10: The true values for some variable is shown in red and the integral average shown as constant lines in each grid cell.

$t + \Delta t$. The generic steps of the MUSCL-type solver as used in DISPATCH/RAMSES are discussed in the following subsections.

2.2.1 Primitive Variables

As described in section 2.1.3, the conserved variables may be expressed with another set of characteristic variables. For the Euler equations with variables $\mathbf{U} = [\rho, \rho u, \rho v, \rho w, E]^T$ the characteristic variables are $\mathbf{W} = [\rho, u, v, w, p]^T$. The first variable, density, remains the same, but momentum and energy are replaced with velocity and pressure.

These variables are in the MUSCL scheme referred to as the primitive variables, which is also done in the rest of the thesis. The use of these makes the next steps in the scheme simpler as all steps can be done independently for each $\mathbf{W_i}$ in each dimension when solving in 3D.

2.2.2 Slope Limiters

From the primitive variables, left and right interface values may be extrapolated based on the gradient between two cells. Artificial numerical oscillations may occur in simulations, which can cause unrealistic interface values. To suppress this effect, slope limiters are used. The general idea is that instead of representing u_i as the integral value of a constant function, we may instead think of it as the integral of a linear function in the region. In other words, the red line in figure 2.10 are in each cell approximated by a linear function:

$$u_i(x) = u_i^n + \frac{x - x_i}{\Delta x} \Delta i, \tag{2.45}$$

where $\Delta_i / \Delta x$ is the slope of the cell. This slope is calculated from the physical variables by looking at the neighbor cells. The simplest estimate is based on the central derivative, which imposes no limitations on the slope and thus does not suppress the artificial oscillations. *minmod* (Yee, 1989) is one of the simpler slope limiters. It examines the left and right gradient. If the gradients have opposite signs, zero will be set as the gradient. Otherwise, if the left gradient is positive the minimum of the left and right gradient is chosen. If the left gradient is negative, the maximum of the left and right gradient is chosen. This method will often tend to smear sharp gradients. Other methods such as the *double minmod*, *superbee*, *van Albada* or *van Leer* may alternatively be used. Bai, Yang, and Zhou, 2018 compares the aforementioned limiters and concludes that "*The appropriate limiter should be selected based on actual cases.*".

2.2.3 Prediction

Once the gradient is found predictions are made for the time $t + \frac{1}{2}\Delta t$. This is done purely using the gradient and existing values and is based on a simple forward in time Euler discretization of the primitive variables:

$$\rho_t + u\rho_x + \rho u_x + v\rho_y + \rho v_y + w\rho_z + \rho u_z = 0 \tag{2.46}$$

$$u_t + uu_x + \frac{1}{\rho}p_x + vu_y + +wu_z = 0 \tag{2.47}$$

$$v_t + uv_x + +vv_y + \frac{1}{\rho}p_y + wv_z = 0 \tag{2.48}$$

$$w_t + uw_x + +vw_y + +ww_z + \frac{1}{\rho}p_z = 0 \tag{2.49}$$

$$p_t + up_x + u_x a^2 p + vp_y + v_y a^2 p + wp_z + w_z a^2 p \tag{2.50}$$

where a is corresponds to the values such that the eigenvalues becomes (for 1D in x-direction) $\lambda_1 = u - a$, $\lambda_2 = u$ and $\lambda_3 = u + a$. Based on these equations the predictions are then made as follows

$$\rho^{n+\frac{1}{2}} = \rho^n + (-u * \Delta\rho_x - \Delta u_x * \rho)\frac{1}{2}\frac{dt}{dx} + (-v\Delta\rho_y - \Delta v_y\rho)\frac{1}{2}\frac{dt}{dy} + (-w\Delta\rho_z - \Delta w_z\rho)\frac{1}{2}\frac{dt}{dz} \tag{2.51}$$

$$\begin{aligned} u^{n+\frac{1}{2}} = u^n &+ (-u\Delta u_x - (\Delta p_x + B\Delta B_x + C\Delta C_x)\frac{1}{\rho})\frac{1}{2}\frac{dt}{dx} \\ &+ (-v\Delta u_y + B\Delta A_y\frac{1}{\rho})\frac{1}{2}\frac{dt}{dy} \\ &+ (-w\Delta u_z + C\Delta A_z\frac{1}{\rho})\frac{1}{2}\frac{dt}{dz} \end{aligned} \tag{2.52}$$

$$\begin{aligned} v^{n+\frac{1}{2}} = v^n &+ (-u\Delta v_x + A\Delta B_x\frac{1}{\rho})\frac{1}{2}\frac{dt}{dx} \\ &+ (-v\Delta v_y - (\Delta p_y + A\Delta A_y + C\Delta C_y)\frac{1}{\rho})\frac{1}{2}\frac{dt}{dy} \\ &+ (-w\Delta v_z + C\Delta B_z\frac{1}{\rho})\frac{1}{2}\frac{dt}{dz} \end{aligned} \tag{2.53}$$

$$\begin{aligned} w^{n+\frac{1}{2}} = w^n &+ (-u\Delta w_x + A\Delta C_x\frac{1}{\rho})\frac{1}{2}\frac{dt}{dx} \\ &+ (-v\Delta w_y + B\Delta C_y)\frac{1}{2}\frac{dt}{dy} \\ &+ (-w\Delta w_z - (\Delta p_z + A\Delta A_z + B\Delta B_z)\frac{1}{\rho})\frac{1}{2}\frac{dt}{dz} \end{aligned} \tag{2.54}$$

$$\begin{aligned} p^{n+\frac{1}{2}} = p^n &+ (-u\Delta p_x - \Delta u_x \gamma p)\frac{1}{2}\frac{dt}{dx} \\ &+ (-v\Delta p_y - \Delta v_x \gamma p)\frac{1}{2}\frac{dt}{dy} \\ &+ (-w\Delta p_z - \Delta w_x \gamma p)\frac{1}{2}\frac{dt}{dz} \end{aligned} \tag{2.55}$$

In the equations above, the magnetic fields in x-, y-, and z-directions (A, B, and C) are included. For now, these can all be assumed to be 0, but they will be further discussed in the next section when discussing MHD.

2.2.4 Interface Values

The predictions made at time $t + \frac{1}{2}\Delta t$ are all cell centered. The face-centered values at $t + \frac{1}{2}\Delta t$ are then extrapolated by subtracting or adding the half gradient from the predicted values. It is important to note that the gradient is still the one evaluated at t while the predicted values are evaluated at $t + \frac{1}{2}\Delta t$. In DISPATCH/RAMSES these values are stored as arrays called `left` and `right`, which holds the values in each direction as sketched in figure 2.11.

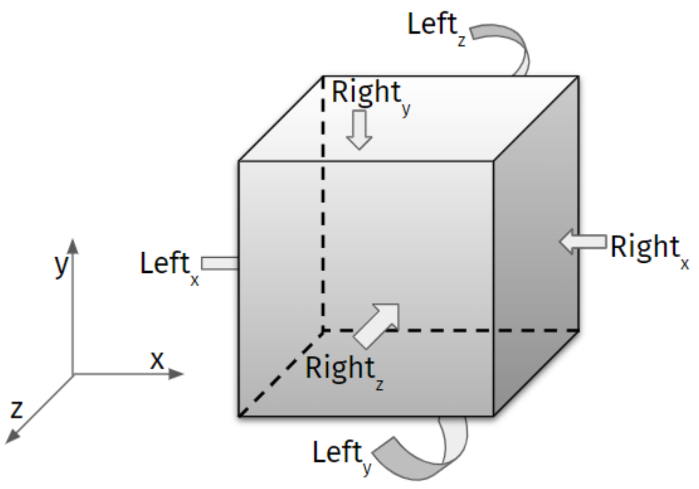


FIGURE 2.11: Left and right interface values in each direction

2.2.5 Riemann Solver

The interface values are then used to solve the 1D Riemann problem at each interface in each direction. The Godunov flux values are downstaggered, which correspond to the left interface in each direction. When solving the Riemann problem in the x-direction for cell (i,j,k), interface values `left(i,j,k)` and `right(i-1,j,k)` are required. An important thing to note here is that the `left(i,j,k)` actually corresponds to the right values in the Riemann problem and the `right(i-1,j,k)` values corresponds to the left values, as sketched in figure 2.12. The Riemann Solver is run similarly in the y- and z-directions.

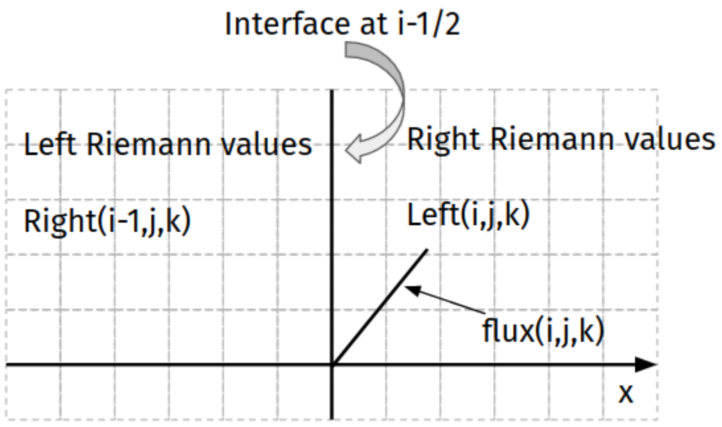


FIGURE 2.12: Left and right array values are used as right and left values in the Riemann problem respectively. Calculated flux is stored in index (i,j,k) (down-staggered).

2.2.6 Flux update

The last step in the MUSCL scheme is to use the Godunov flux to update each cell value. The update is rather simple and can be written as:

$$\begin{aligned} C_{val}^{n+1} = C_{val}^{n} &+ (flux_x^{n+\frac{1}{2}}{}_{(i,j,k)} - flux_x^{n+\frac{1}{2}}{}_{(i+1,j,k)})\frac{dt}{dx} \\ &+ (flux_y^{n+\frac{1}{2}}{}_{(i,j,k)} - flux_y^{n+\frac{1}{2}}{}_{(i,j+1,k)})\frac{dt}{dy} \\ &+ (flux_z^{n+\frac{1}{2}}{}_{(i,j,k)} - flux_z^{n+\frac{1}{2}}{}_{(i,j,k+1)})\frac{dt}{dz} \end{aligned} \tag{2.56}$$

Where C_{val} is any of the conserved variables. Once again, recall that the flux indices are face-centered.

2.3 Magneto Hydrodynamics

2.3.1 Governing Equations

The Euler equations (eq 2.6) can be extended to include the magnetic fields in each direction. For ideal MHD this becomes:

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ B_x \\ B_y \\ B_z \\ E \end{bmatrix}, \mathbf{F}(\mathbf{U}) = \begin{bmatrix} \rho u \\ \rho u^2 + p_T - B_x^2 \\ \rho u v - B_x B_y \\ \rho u w - B_x B_z \\ 0 \\ B_y u - B_x v \\ B_z u - B_x w \\ u(E + p_T) - B_x(\mathbf{v} \cdot \mathbf{B}) \end{bmatrix}, \mathbf{G}(\mathbf{U}) = \begin{bmatrix} \rho v \\ \rho u v - B_y B_x \\ \rho v^2 + p_T - B_y^2 \\ \rho w v - B_y B_z \\ B_x v - B_y u \\ 0 \\ B_z b - B_y w \\ v(E + p_T) - B_y(\mathbf{v} \cdot \mathbf{B}) \end{bmatrix},$$

$$\text{H(U)=} \begin{bmatrix} \rho w \\ \rho u w - B_z B_x \\ \rho v w - B_z B_y \\ \rho w^2 + p_T - B_z^2 \\ B_x w - B_z u \\ B_y w - B_z v \\ 0 \\ w(E + p_T) - B_z(\mathbf{v} \cdot \mathbf{B}) \end{bmatrix} \quad (2.57)$$

where $\mathbf{v} = (u, v, w)$, $\mathbf{B} = (B_x, B_y, B_z)$, $p = (\gamma - 1)(E - \frac{1}{2}\rho|\mathbf{v}|^2 - \frac{1}{2}|\mathbf{B}|^2)$ and $p_T = p + \frac{1}{2}|\mathbf{B}|^2$. The flux of the normal magnetic field component is always zero due to the divergence-free condition of the magnetic field. Note here the introduction of *total pressure*, which includes the pressure due to the magnetic fields. Similar to the Riemann problem without magnetic fields, the problem can be split up in each dimension and the 1D Riemann problem can be solved at each cell interface.

2.3.2 HLLD: Riemann Solver for MHD

The HLL and HLLC can be extended to include the magnetic fields. However, by introducing magnetic fields to the equation, rotational discontinuities propagating with the Alfvén waves occurs. These are not well handled by neither HLL nor HLLC. Miyoshi and Kusano, 2005 suggested using the same assumptions as HLLC, but extending the Riemann fan with two additional intermediate states as illustrated in figure 2.13.

The eigenvalues (waves separating each state) of equation 2.3.1 are well known. In the 1D case along the x-axis, these are given by:

$$\lambda_{1,7} = S_M \pm c_f, \quad \lambda_{2,6} = S_M \pm c_a, \quad \lambda_{3,5} = S_M \pm c_s, \quad \lambda_4 = S_M \quad (2.58)$$

where SM is the normal velocity across the fan. This correspond to two Alfvén waves, c_a, four magneto-acoustic waves, c_s and c_f, and one entropy wave, SM. In the HLLD solver, c_s is not used. The Aflvén and magneto-acoustic waves in the

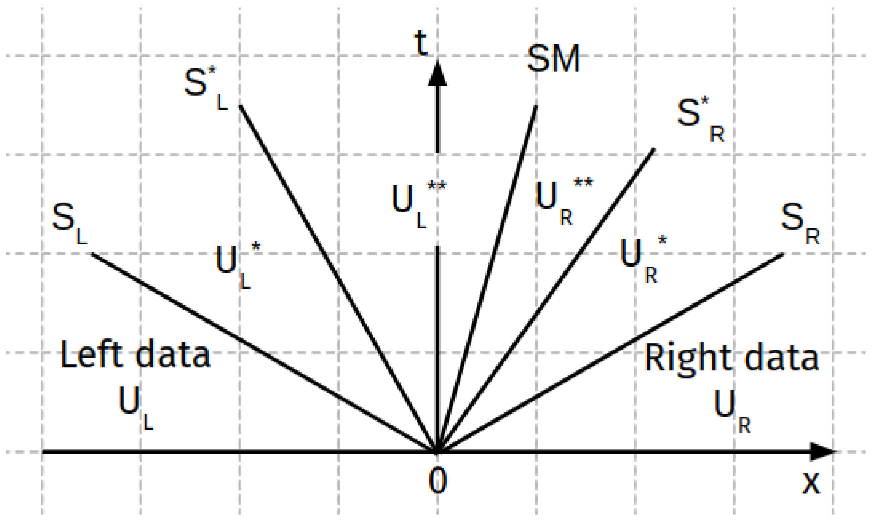


FIGURE 2.13: six-state HLLD solver for MHD.

x-direction are given by:

$$c_a = \frac{|B_x|}{\sqrt{\rho}}, \quad c_f = \left(\frac{\gamma p + |\mathbf{B}|^2 + \sqrt{(\gamma p + |\mathbf{B}|^2)^2 - 4\gamma p B_x^2}}{2\rho} \right)^{1/2} \tag{2.59}$$

For the rest of this chapter I will refer to $\lambda_1 = S_M - c_{fL}$ as S_L, $\lambda_7 = S_M + c_{fR}$ as S_R, $\lambda_2 = S_M - c_{aL}$ as S_L^*, $_6 = S_M + c_{aR}$ as S_R^* and $\lambda_4 = S_M$ simply as S_M.

Here c_{fL} and c_{fR} are calculated using the left and right region values respectively. Similarly, c_{aL} and c_{aR} will be calculated based on left and right star region values, which is also shown further below.

The normal velocity across the Riemann fan is similar to the HLLC, but with the total pressure:

$$S_M = \frac{(S_R - u_R)\rho_R u_R - (S_L - u_L)\rho_L u_L - p_{T_R} + p_{T_R}}{(S_R - u_R)\rho_R - (S_L - u_L)\rho_L} \tag{2.60}$$

The normal velocity and total pressure are assumed to be constant over the fan. The total pressure is given by:

$$p_T^* = \frac{(S_R - u_R)\rho_R p_{T_L} - (S_L - u_L)\rho_L p_{T_R} + \rho_L \rho_R (S_R - u_R)(S_L - u_L)(u_R - u_L)}{(S_R - u_R)\rho_R - (S_L - u_L)\rho_L} \tag{2.61}$$

In both HLL and HLLC the star region flux was directly linked to the left/right flux. In the HLLD scheme, intermediate values in the star and double-star region are instead calculated and the flux is then calculated according to equation 2.3.1.

In the following, the α subscript denotes either the left or right state. For 1D in the x-direction, the left and right star region values are given by:

$$\rho_\alpha^* = \rho_\alpha \frac{S_\alpha - u_\alpha}{S_\alpha - S_M} \tag{2.62}$$

$$v_\alpha^* = v_\alpha - B_x B_{y_\alpha} \frac{S_M - u_\alpha}{\rho_\alpha (S_\alpha - u_\alpha)(S_\alpha - S_M) - B_x^2} \tag{2.63}$$

$$w_\alpha^* = w_\alpha - B_x B_{z_\alpha} \frac{S_M - u_\alpha)}{\rho_\alpha(S_\alpha - u_\alpha)(S_\alpha - S_M) - B_x^2} \tag{2.64}$$

$$B_{y_\alpha}^* = B_{y_\alpha} \frac{\rho_\alpha(S_\alpha - u_\alpha)^2 - B_x^2}{\rho_\alpha(S_\alpha - u_\alpha)(S_\alpha - S_M) - B_x^2} \tag{2.65}$$

$$B_{z_\alpha}^* = B_{z_\alpha} \frac{\rho_\alpha(S_\alpha - u_\alpha)^2 - B_x^2}{\rho_\alpha(S_\alpha - u_\alpha)(S_\alpha - S_M) - B_x^2} \tag{2.66}$$

$$e_\alpha^* = \frac{(S_\alpha - u_\alpha)e_\alpha - p_{T_\alpha}u_\alpha + p_{T_\alpha} + p_T^* S_M + B_x(\mathbf{v}_\alpha \cdot \mathbf{B}_\alpha - \mathbf{v}_\alpha^* \cdot \mathbf{B}_\alpha^*)}{S_\alpha - S_M} \tag{2.67}$$

These values are then used to determine double-star region values as well as the Alfvén waves:

$$S_L^* = S_M - \frac{|B_x|}{\sqrt{\rho_L^*}}, \quad S_R^* = S_M + \frac{|B_x|}{\sqrt{\rho_R^*}} \tag{2.68}$$

Furthermore, the transverse velocities and magnetic field can be shown to be constant in the double star region, and we thus do not need to differentiate between left and right double star region values:

$$v^{**} = \frac{\sqrt{\rho_L^*}v_L^* + \sqrt{\rho_R^*}v_R^* + (B_{y_R}^* - B_{y_L}^*)sign(B_x)}{\sqrt{\rho_L^*} + \sqrt{\rho_R^*}} \tag{2.69}$$

$$w^{**} = \frac{\sqrt{\rho_L^*}w_L^* + \sqrt{\rho_R^*}w_R^* + (B_{z_R}^* - B_{z_L}^*)sign(B_x)}{\sqrt{\rho_L^*} + \sqrt{\rho_R^*}} \tag{2.70}$$

$$B_y^{**} = \frac{\sqrt{\rho_L^*}B_{y_R}^* + \sqrt{\rho_R^*}B_{y_L}^* + \sqrt{\rho_L^*\rho_R^*}(v_R^* - v_L^*)sign(B_x)}{\sqrt{\rho_L^*} + \sqrt{\rho_R^*}} \tag{2.71}$$

$$B_z^{**} = \frac{\sqrt{\rho_L^*}B_{z_R}^* + \sqrt{\rho_R^*}B_{z_L}^* + \sqrt{\rho_L^*\rho_R^*}(w_R^* - w_L^*)sign(B_x)}{\sqrt{\rho_L^*} + \sqrt{\rho_R^*}} \tag{2.72}$$

where $sign(B_x) = 1$ if $B_x > 0$ and $sign(B_x) = -1$ if $B_x < 0$. Finally, the energy of the double star regions can be found by:

$$e_\alpha^{**} = e_\alpha^* \pm \sqrt{\rho_\alpha^*}(\mathbf{v}_\alpha^* \cdot \mathbf{B}_\alpha^* - \mathbf{v}_\alpha^{**} \cdot \mathbf{B}_\alpha^{**}) \tag{2.73}$$

where the $\pm$ is minus for $\alpha = L$ and plus for $\alpha = R$.

The equations listed above make it possible to compute the flux for each different region. The final flux is then chosen based on

$$\mathbf{F}^{hlld} = \begin{cases} \mathbf{F}_L, & \text{if} \quad S_L > 0 \\ \mathbf{F}_L^*, & \text{if} \quad S_L \leq 0 \leq S_L^* \\ \mathbf{F}_L^{**}, & \text{if} \quad S_L^* \leq 0 \leq S_M \\ \mathbf{F}_R^{**}, & \text{if} \quad S_M \leq 0 \leq S_R^* \\ \mathbf{F}_R^*, & \text{if} \quad S_R^* \leq 0 \leq S_R \\ \mathbf{F}_R, & \text{if} \quad S_R < 0 \end{cases} \tag{2.74}$$

2.3.3 HLLD 2D - stable magnetic flux corrections

An issue with the HLLD equations as described in the previous section is that it does not enforce the divergence of the magnetic field to vanish at all times. Early MHD solvers fixed this by adding divergence cleaning steps or reformulating the

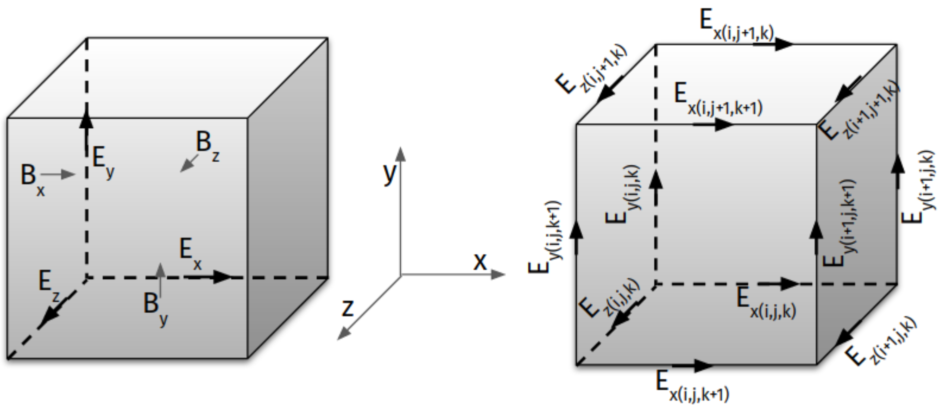


FIGURE 2.14: Magnetic field and EMF location are represented with array index values. x, y, and z and denoted with index i,j,k respectively.

MHD equations (see Ryu et al., 1998 or Brackbill and Barnes, 1980). Fromang, Hennebelle, and Teyssier, 2012 proposed extending the MUSCL variant of HLLD to include the induction equation in a constrained transport formulation. In this scheme, the density, momentum, and energy are updated based on the flux found in the 1-dimensional HLLD equations described in the previous section. From this point, this will be referred to as the HLLD 1D. The magnetic fields are updated based on the edge-averaged electromotive force (EMF) found by solving the Riemann problem in 2D. This will be referred to as HLLD 2D.

The magnetic fields are face-centered. In the x-direction the value of $B_x(i,j,k)$ thus actually represent the value at $B_x(i-\frac{1}{2},j,k)$ and similar for y- and z-direction. The change in magnetic field can be calculated from time n to $n+1$ based on the edge-centered EMFs calculated from the HLLD 2D as follows:

$$B_x^{n+1}{}_{(i-\frac{1}{2},j,k)} = B^n_{x\,(i-\frac{1}{2},j,k)} + \frac{E_z^{n+\frac{1}{2}}{}_{(i-\frac{1}{2},j+\frac{1}{2},k)} - E_z^{n+\frac{1}{2}}{}_{(i-\frac{1}{2},j-\frac{1}{2},k)}}{dy}dt - \frac{E_y^{n+\frac{1}{2}}{}_{(i-\frac{1}{2},j,k+\frac{1}{2})} - E_y^{n+\frac{1}{2}}{}_{(i-\frac{1}{2},j,k-\frac{1}{2})}}{dz}dt \tag{2.75}$$

with similar expressions for B_z and B_y. This expression can be rewritten in term of array indexes as:

$$B_x^{n+1}{}_{(i,j,k)} = B^n_{x\,(i,j,k)} + \frac{E_z^{n+\frac{1}{2}}{}_{(i,j+1,k)} - E_z^{n+\frac{1}{2}}{}_{(i,j,k)}}{dy}dt - \frac{E_y^{n+\frac{1}{2}}{}_{(i,j,k+1)} - E_y^{n+\frac{1}{2}}{}_{(i,j,k)}}{dz}dt \tag{2.76}$$

If not stated otherwise, the indexes based on array value will be used. An illustration of the location of the magnetic field and EMF based on array indices can be seen in figure 2.14. The rest of this chapter will focus on the actual extensions made to the MUSCL solver to get stable EMF values in the HLLD 2D.

2.3.4 MUSCL Extensions for MHD

Primitive Variables

The primitive variables as described in section 2.2 are extended to also include the magnetic field in each direction (A, B, and C). In DISPATCH/RAMSES two values are stored of the magnetic field. The face-centered values are stored in a separate

`BF` array. Interpolated cell-centered values are stored in the same array as the other primitive variables.

Slopes

The slope of the cell-centered magnetic field is calculated identically to the primitive variables. The slopes of the face-centered magnetic field variables are calculated in the transverse direction. Face-centered slopes in the same direction as the magnetic field are not needed.

Predict

Predictions at time $t + \frac{1}{2}\Delta t$ are made as described in section 2.2 for density, velocities and pressure. For the magnetic field, additional steps must be made. First the edge-centered EMFs as illustrated in figure 2.14 are calculated at time t. This is done by simple arithmetic averages of the magnetic field and velocities. For example the EMF $E^n_{z\,(i,j,k)}$ is calculated by:

$$E^n_{z\,(i,j,k)} = \bar{v}_x \bar{B}_y - \bar{v}_y \bar{B}_x \tag{2.77}$$

with

$$\bar{v}_x = \frac{1}{4}(v^n_{x\,(i,j,k)} + v^n_{x\,(i,j-1,k)} + v^n_{x\,(i-1,j,k)} + v^n_{x\,(i-1,j-1,k)}) \tag{2.78}$$

$$\bar{v}_y = \frac{1}{4}(v^n_{y\,(i,j,k)} + v^n_{y\,(i,j-1,k)} + v^n_{y\,(i-1,j,k)} + v^n_{y\,(i-1,j-1,k)}) \tag{2.79}$$

$$\bar{B}_x = \frac{1}{2}(B^n_{x\,(i,j,k)} + B^n_{x\,i,j-1,k}) \tag{2.80}$$

$$\bar{B}_y = \frac{1}{2}(B^n_{y\,(i,j,k)} + B^n_{y\,i-1,j,k}) \tag{2.81}$$

Here v_x and v_y are the cell centered values, and B_x and B_y are face-centered. The EMFs are then used to predict the face-centered magnetic fields at time $t + \frac{1}{2}\Delta t$. This is done exactly according to equation 2.76, but with the EMF values at t, and multiplied with $\frac{1}{2}dt$ instead of dt. This gives predictions for the face-centered values of each magnetic field along their respective direction, which corresponds to left/right array values. Note that these predictions means that `right(A,i,j,k)=left(A,i+1,j,k)`, and similar for B and C, further enforcing that the magnetic fields are constant across the cells.

Lastly, the transverse magnetic field is needed in each direction. This is done by interpolating the cell-centered magnetic fields and using the cell-centered gradient similar to how left/right values are calculated for the primitive variables.

2D predictions

To use the HLLD 2D Riemann Solver the full MHD state at each edge must be reconstructed at time $t + \frac{1}{2}\Delta t$. Each edge will have 4 values corresponding to the values predicted by each cell connected to the edge. As such, edge-centered corner values are interpolated in each direction. The corner values are labeled as LB, LT, RB, and RT denoting left/right and bottom/top. Here left/right is used to denote the first and second edge in the first transverse direction. Similar bottom/top denotes first and second edge in the second transverse direction. An illustration of the corner values can be seen in figure 2.15.

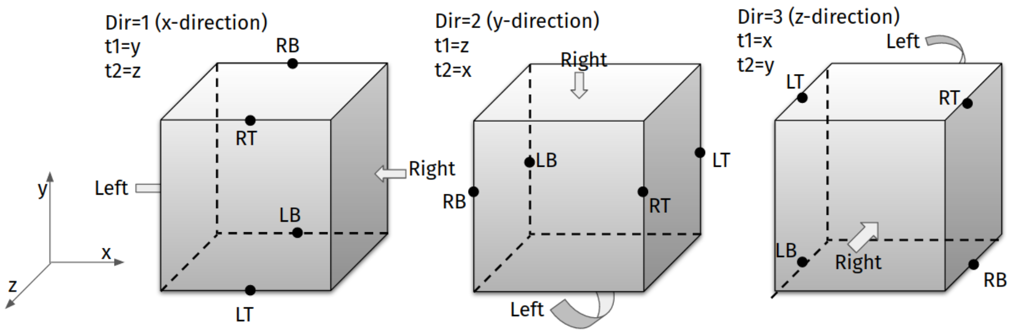


FIGURE 2.15: Corner values in each direction. t1 and t2 denote the first and second transverse direction.

The predictions for the primitive variables are calculated based on the cell-centered value and the transverse gradients as follows:

$$\begin{aligned} LB_\rho &= \rho - \Delta\rho_1 - \Delta\rho_2 \\ LT_\rho &= \rho - \Delta\rho_1 + \Delta\rho_2 \\ RB_\rho &= \rho + \Delta\rho_1 - \Delta\rho_2 \\ RT_\rho &= \rho + \Delta\rho_1 + \Delta\rho_2 \end{aligned} \tag{2.82}$$

Where subscripts 1 and 2 denote gradient in the first and second transverse direction. Velocities and pressure are calculated in the same way. The magnetic field in the normal direction (A for dir=1, B for dir=2, and C for dir=3) is calculated similarly with the cell-centered average of the left and right state along with the cell-centered gradient.

The transverse magnetic fields on the other hand are calculated using face-centered values and face-centered gradients. The magnetic field in the first transverse direction uses the gradient in the second transverse direction. For example, when looking in the x-direction the RB values are calculated as follows:

$$\begin{aligned} RB_B &= BR - \Delta BR_z \\ RB_C &= CL + \Delta CL_y \end{aligned} \tag{2.83}$$

Where BR is the right interface value of B, CL is the left interface value of C, ΔBR_z is the face-centered gradient of B at the right interface in the z-direction and ΔCL_y is the face-centered gradient of C at the left interface in the z-direction. Here left and right for B is in the y-direction while left and right for C is in the z-direction.

Calling the HLLD 2D

The corner values from the 2D predictions are then used in the HLLD 2D. The values required by the solver in each cell are the corner values from 4 neighboring cells at the edges where the EMF is defined. As such, when looking in the x-direction the HLLD solver requires 4 values in the same position as $E_{x(i,j,k)}$ in figure 2.14. When calling the HLLD 2D instead of calling the edge-centered values LB, LT, RB, and RT they are instead called the SW, NW, SE, and NE values. This is done to highlight the fact that they are not the same corner values as earlier described. When looking in the x-direction the LB value of cell (i,j,k) becomes the NE value. LT of cell (i,j,k-1) becomes SE, RT of cell (i,j-1,k-1) becomes SW and RB of cell (i,j-1,k) becomes NW.

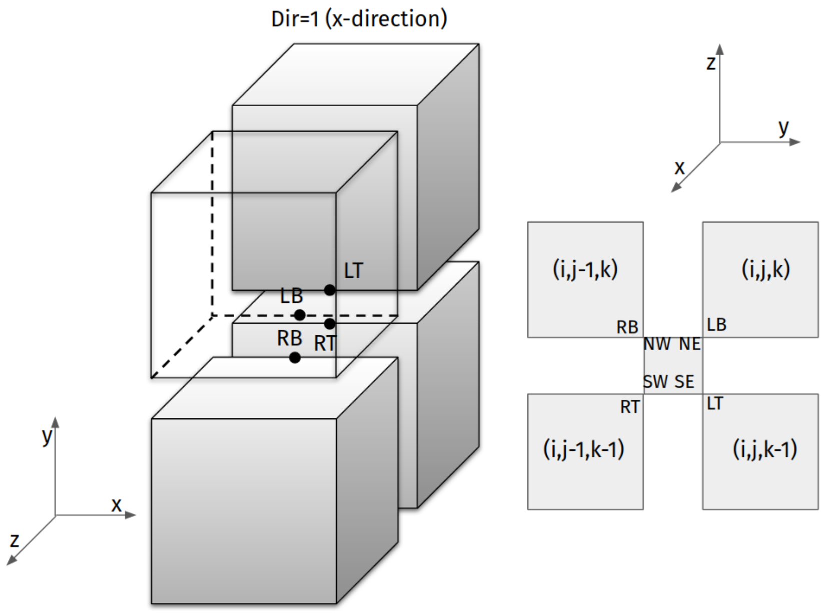


FIGURE 2.16: Cell corner values to edge-centered corner values in the x-direction. SW, SE, NW, NE values are used to solve the 2D Riemann problem and calculate EMF.

This is also sketched in figure 2.16. Similar figures for y- and z-direction can be found in figure A.1 in the appendix.

2.4 Summary

To summarise this chapter. The MUSCL method for solving HD and MHD equations has been presented. It is based in part on the volume integral for density, momentum and energy.

$$\begin{aligned} \int_V \mathbf{U}(x,y,z,t_1)dV = \int_V \mathbf{U}(x,y,z,t_2)dV+ \\ \int_{t_1}^{t_2}\int_{A_{LX}} \mathbf{F}(\mathbf{U})(x_1,y,z,t)dAdt - \int_{t_1}^{t_2}\int_{A_{RX}} \mathbf{F}(\mathbf{U})(x_2,y,z,t)dAdt+ \\ \int_{t_1}^{t_2}\int_{A_{LY}} \mathbf{G}(\mathbf{U})(x,y_1,z,t)dAdt - \int_{t_1}^{t_2}\int_{A_{RY}} \mathbf{G}(\mathbf{U})(x,y_2,z,t)dAdt+ \\ \int_{t_1}^{t_2}\int_{A_{LZ}} \mathbf{H}(\mathbf{U})(x,y,z_1,t)dAdt - \int_{t_1}^{t_2}\int_{A_{RZ}} \mathbf{H}(\mathbf{U})(x,y,z_2,t)dAdt \end{aligned} \tag{2.84}$$

where subsript V denotes volume integral, subscript A denotes surface integral and subscript LX, RX, LY, RY, LZ and RZ denotes left or right interface in x- y- and z-direction.

It is furthermore based on the integral form of the induction equation:

$$-\frac{\partial B}{\partial t}dA = \oint Edl \tag{2.85}$$

To find the required flux and EMF the full MHD state is reconstructed at the interface and at the edges of each cell. For interface values slope interpolation is used inside each cell and the Riemann problem is solved at the interface. The following steps describe the numerical implementation:

1. Convert conserved to primitive variables
2. Compute slopes
3. Predict values based on simple Euler time discretization of the primitve variables
4. Construct predicted states at interface and corners
5. Solve 1D and 2D Riemann problem in each direction
6. Update conserved variables

The following chapters will discuss in practice how the DISPATCH code work and how the MUSCL solver has been implemented using GPUs.

Chapter 3

DISPATCH

The goal of the thesis has been to convert the MHD code of the DISPATCH/RAMSES solver to GPU. This chapter aims at giving a brief overview of the DISPATCH framework.

3.1 Motivation and Key Ideas

There currently exist many different codes for astrophysics fluid simulation. Some examples are ZEUS (Stone and Norman, 1992), FARGO3D (Benitez-Llambay and Masset, 2016), BIFROST (Gudiksen et al., 2011), STAGGER (Nordlund, Galsgaard, and Stein, 1994), and GenASIS (Cardall et al., 2014). These codes are at their core grid-based. Traditional grid-based codes distribute equal-sized chunks of the overall grid to each available compute unit. Timesteps may be computed locally, but are applied globally, which both gives rise to extra communication and also waste of update cost since all cells must respect the most restrictive timestep constraint encountered. Additionally, the amount of computations needed to advance different parts of the grid may differ, especially in cases with Adaptive Mesh Refinement(AMR). All this means that traditional codes only scale well up to a limit on the number of cores used. One might partially compensate for this by custom fitting a code specifically to the system it is to be run on. However, as discussed in Dubey et al., 2014, modern codes are too large to be realistically modified to run optimally on a given machine/architecture. Dubey et al., 2014 therefore reasons, that existing codes must be refactored, and new frameworks must be designed, aimed specifically for scalability.

DISPATCH (Nordlund et al., 2018) is a relatively new framework, which aims to solve the scalability problems that traditional codes face. DISPATCH breaks with the traditional domain-decomposition strategy. Instead of statically splitting up a grid into relatively large chunks, smaller chunks are used and are represented as Fortran objects (referred to as *extended data types* in Fortran). These hold several *time slices* of the grid-values, and also meta-data, such as location, mesh information, time, timestep, etc. These chunks are in DISPATCH referred to as *patches*, which are one type of extension to a basic, underlying task concept.

The patch organization allows timestep constraints to be applied locally, which reduces update costs significantly in cases where the allowed time step varies a lot. It also enables dynamic load balancing as patches can freely be exchanged between compute nodes. Each patch is updated individually and only communicates with neighboring patches to exchange guard/ghost zones. Advancing patch values for hydrodynamic, MHD, or radiative transfer can be done locally in each patch. If multiple types of updates are to be made on the same patch separate tasks may be created for MHD and radiative transfer, for example. While typically not executing as a separately scheduled task, the exchange of ghost zones and similar services are handled by procedures that are independent of the solvers, which makes the development of new solvers, or porting of existing solvers to GPU much easier.

What tasks need to be updated and the order in which this happens is controlled by a central task scheduler, which *dispatches* the tasks inside each compute node.

3.2 Tasks

The concept of tasks is central to the DISPATCH framework. As mentioned above, *patches* are grid-based extensions of basic task objects in DISPATCH, with task ID, flags, etc.

As both DISPATCH and OpenMP are used in this thesis, it is important to note that *tasks* have different meanings depending on the context. When discussing OpenMP a task is best defined as "*a schedulable unit of work defined by a region of code plus a data environment*" (Mattson, He, and Koniges, 2019). When discussing DISPATCH a task is simply an extended data type, which carries information such as task number, readiness, and more.

In practice, a DISPATCH task is often implemented inside an OpenMP task region. For performance, multiple DISPATCH tasks may be scheduled together inside a single OpenMP region if there are direct dependencies or small DISPATCH service tasks that are better executed with other work to avoid scheduling overhead.

The basic task object in DISPATCH may be extended to represent many different things, such as single finite-size objects with complex interiors (stars, planets, ...), ensembles of millions of particles (e.g. dust), or bundles of rays of light, describing radiative energy transfer.

This creates a hierarchy for the different types of task-specific data commonly used. The *task object* holds the basic information about the task (ID fx.) as mentioned above, and related methods (such as giving unique ID numbers to tasks). Next in the hierarchy, the *patch object* adds information and methods related to the spatial properties of such as size, number of cells, guard zones, physical variables, and coordinate system. Further up is the *solver object*, which holds information and procedures related to the specific solver (fx. MHD, HD, or radiative transfer). And lastly, the *experiment object* holds experiment-specific and procedures, including boundary and initial conditions.

Tasks are furthermore organized into a task list, whose *nodes* (called *links* in DISPATCH to avoid confusion with compute nodes) contain pointers to the tasks. The link objects also contain information about the neighboring tasks on which the current task depends or vice versa. The DISPATCH main program calls the *execute* procedure in the tasks list object, which in turn calls *update* on each task in the *ready-queue* of tasks that are ready to be updated until all tasks are finished.

3.3 Task Scheduling

The task scheduler is the novel technique that separates DISPATCH from other codes. A simplified flowchart is shown in figure 3.1. The figure show only intra-node scheduling and ignore the *dispatching* of tasks between nodes.

As described above, each patch object has some meta-information about the patch. Part of this is the patch's *readiness*. *Readiness* is used to determine if the patch is ready to advance in time. This depends mainly on how far in time the neighboring patches have advanced. This readiness is checked in the pre-processing part for figure 3.1.

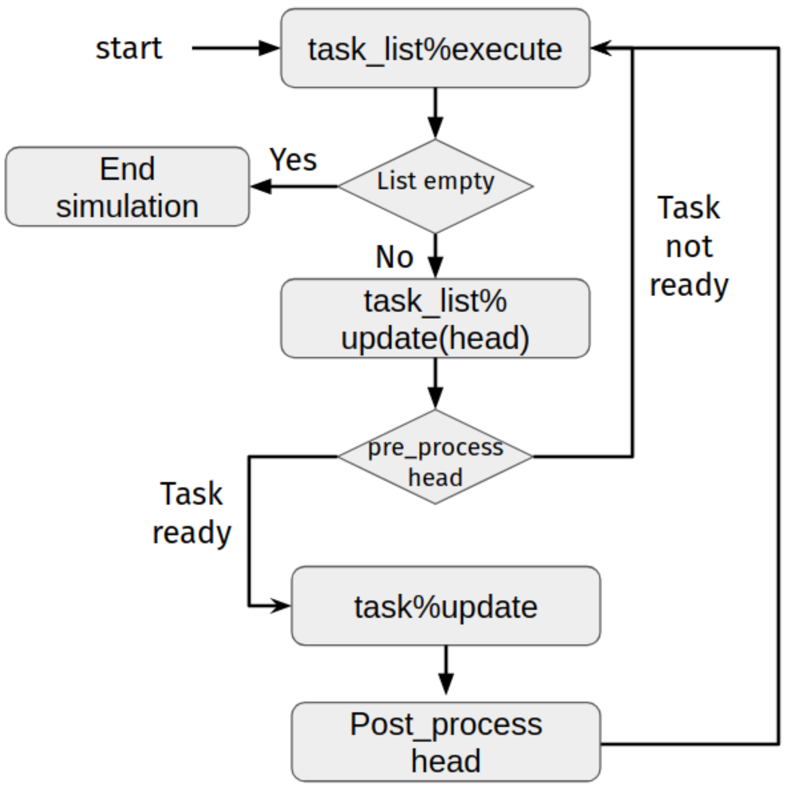


FIGURE 3.1: Flowchart showing a simplified execution flow of DISPATCH within a single node.

Once a patch is ready it is appended to the *ready-queue*, which is a different type of *task list*. The ready-queue is sorted by time, with the oldest task in front. DISPATCH offers several (currently 6) different modes for dispatching the tasks in the ready queue.

The first mode, which is also the default one, simply lets threads 'pop' the queue to access the oldest task. After updating a task, the thread checks the neighboring tasks to see if the update has made any neighbor ready. This is done in the post-process part of figure 3.1. If any ready tasks are found they are added to the queue. This approach is simple and scales very well. The reason for this is that the time it takes to pop a task from the queue is very small, and thus even if this needs to be protected by an OpenMP lock it does not cause significant waiting time.

The other modes were mainly used to evaluate other strategies for dispatching tasks; e.g. using a single thread exclusively for such work. None of these alternative strategies turned out to have significant advantages, given the typical range of cores-per-processor in use today. If much larger numbers of cores per processor become available in the future, it may well pay off to set aside one or more cores to exclusively do work related to task dispatching.

The above-described scheduling explains how tasks are scheduled within a single compute node. DISPATCH may be run on multiple compute nodes. Here, each node will have its task list, which is a subset of the entire grid. Non-blocking MPI calls are used to exchange ghost zones for boundary patches in each compute node. Boundary patches will have a special flag set.

Rather than sending only ghost zones DISPATCH transfers the entire neighbor patch. This simplifies package creation and also simplifies balancing work between nodes. This means that patches may be stored on multiple compute nodes. Only one node needs to update any given patch. This is handled by setting the boundary flag to *virtual* instead.

This also makes load balancing easier. Because entire patches are exchanged, load balancing requires no extra data exchange between nodes. If for example node 1 is overloaded, and node 2 has little work, the virtual patches on node 2 may simply change the flag from virtual to boundary. Similarly, these patches on node 1 will change status from boundary to virtual, and node 2 will have more work to do while node 1 has less.

Chapter 4

the Central Processing Unit (CPU)

To better understand the considerations needed to program for GPUs, a brief overview of the CPU is needed. The goal of this chapter is to highlight the most important features and potential problems of the CPU. Many of these features are hidden from the programmer, and as such are often not considered during development.

4.1 Purpose of the CPU

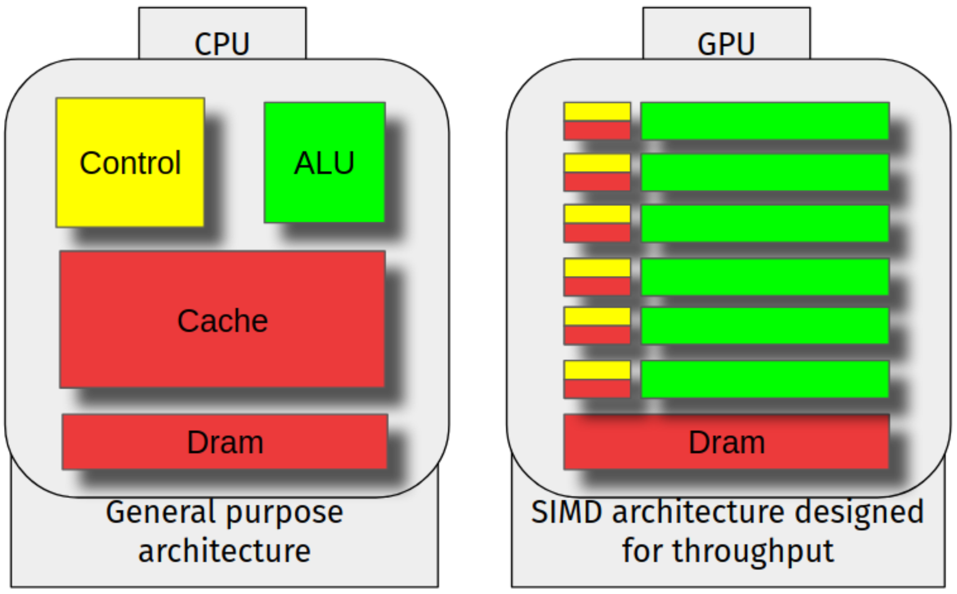

CPU
Control
ALU
Cache
Dram
General purpose architecture
GPU
Dram
SIMD architecture designed for throughput

FIGURE 4.1: Main differences in GPU and CPU architecture.

The main purpose of the CPU is to process data. The CPU needs to be able to handle branch prediction, arithmetic, as well as input/output operations. All this needs to happen with low latency.

To support such a general workload, the CPU needs to be able to support multiple different memory access patterns. This is most effectively done by having a large cache and a large portion of the CPU is therefore dedicated to memory. A large part of the CPU is also dedicated to the *control unit*, which maintains and handles what is being stored and what instruction to be executed next. This leaves only a small part of the CPU for actual computation. This structure is sketched in figure 4.1, which also shows the main differences between a CPU and GPU.

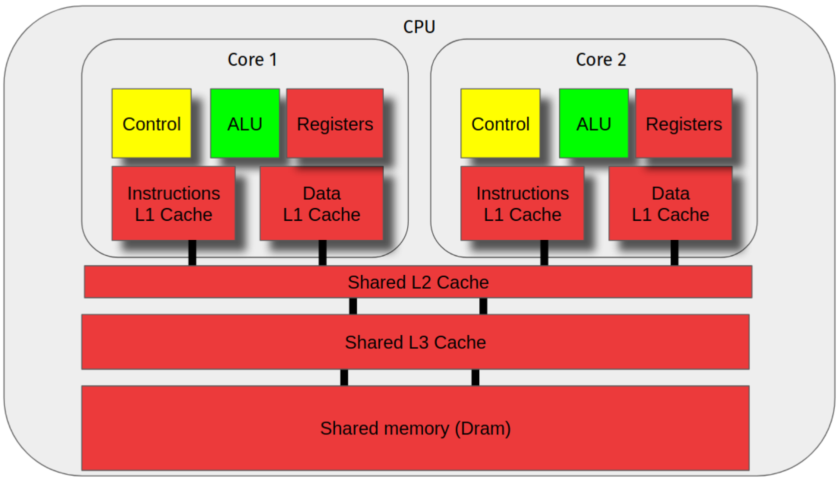


FIGURE 4.2: Simple memory hierarchy for a dual-core system. Only cache and main memory are shown in the figure.

When programming for the CPU, it is therefore common practice to try to eliminate redundant computations as much as possible. This is however not always advantageous as fetching from the L1 cache is still slower than a single computation. When the approach is taken, it often comes at the cost of having many local variables and storing intermediate values. On the CPU this is not a problem as there is plenty of storage for the variables. On the GPU storage is limited but computations are inexpensive and as such require a different programming approach.

The rest of the chapter will focus on the memory model, as optimizing memory transfer is the most important part of GPU programming. It is therefore necessary to know how memory is managed on the CPU compared to the GPU and how this can affect performance.

4.2 The Memory Model

Today, a state-of-the-art CPU may run at around 5Ghz ($5 * 10^9$ cycles per second). An operation such as a multiplication of two floating-point numbers takes around 1 clock cycle. This means that a typical CPU can theoretically run $5 \cdot 10^9$ Floating Point Operations Per Second (FLOPS) or five GFLOPS. This estimate does not take into account any form of instruction-level parallelism. Many modern CPUs are capable of executing over 16 instructions per clock cycle.

To perform arithmetic operations the CPU needs to have the values in registers. Moving data from main memory to registers takes a long time compared to a single clock cycle (see table 4.1 for reference). For this reason, modern computers come with a memory hierarchy, that not only consists of a disk and main memory, but also several smaller and smaller caches ending with the registers. Data is moved between system memory, main memory, and caches in chunks. Between system memory and main memory, these chunks are referred to as *blocks* and between main memory and cache are referred to as *cache lines*. Blocks are moved as programs often access memory stored in close proximity. This is true both for data and for instructions. For example, if one instruction is selected for execution there is a high probability that the next instruction in memory will be executed after. Similarly, if index i in an array is used in a for-loop, chances are that index $i + 1$ will be referenced soon thereafter.

This spatial locality is a key idea for the performance of the cache system. Programs are however not forced to adhere to spatial locality.

A simple overview of the memory hierarchy for a dual-core system can be seen in figure 4.2. Each core has its own L1 cache. The two cores share an L2 cache, which is connected to a larger L3 cache, which is then connected to the main memory. The main memory is in turn connected to some disk, and the computer will most likely be connected to other computers via the internet. The further away from the cores the longer it takes to read from the corresponding memory. As it can be seen in table 4.1, reading from main memory is on the order of 500 times slower than executing a typical instruction.

The CPU will generally try to keep things as close to the CPU (or core) as possible to avoid idle wait time. This works well for single-core CPUs but can have some negative impact on multi-core systems. If core 1 has a value in its L1 cache and core 2 needs access to that value, there is substantial overhead in the transfer. Thus, different cores may have different access times for the same memory. This is generally referred to as Non-Uniform Memory Access (NUMA).

Modern CPUs and compilers are very advanced, and for general-purpose computing, they can make very good predictions on how to best store and move data in the memory. However, a downfall to the generality of the modern CPU and compilers is that they handle everything well, but not optimally.

execute typical instruction	1/5.000.000.000 sec = 0.2 ns
read from L1 cache memory	0.5 ns
branch misprediction	5 ns
fetch from L2 cache memory	7 ns
Mutex lock/unlock	25 ns
fetch from main memory	100 ns
send 2K bytes over 1Gbps network	20.000 ns
read 1MB sequentially from memory	250.000 ns
fetch from new disk location (seek)	8.000.000 ns
read 1MB sequentially from disk	20.000.000 ns
send packet US to Europe and back	150 milliseconds = 150.000.000 ns

TABLE 4.1: Table of reference timings for different memory actions. Taken from `http://norvig.com/21-days.html#answers`

4.2.1 Virtual Memory

Virtual memory is a memory management technique employed by the operating system. It is an abstraction of the memory system, which simplifies memory management within each program. Each program will have its virtual memory. This memory will be seen as contiguous by the program. From the perspective of the program, it looks as if it has uncontested access to all main memory. The operating system and hardware will map each programs virtual memory to physical memory. Contiguous memory in a program's virtual memory is not guaranteed to be contiguous in physical memory. Having virtual memory also allows a program, which requires more memory than physically available to run without errors.

The translation between physical and virtual memory happens in the Memory Management Unit (MMU), which is located on the CPU in modern computers. Virtual memory is usually stored in chunks of 4096 bytes called pages. This is referred to as a memory frame when referencing the physical memory counterpart. Each process will have its page table, which holds information about address translation; going between virtual and physical memory. If a page is referenced and it is not in the main memory, it is called a page fault.

To reduce access time the MMU has a memory cache called the Translation Lookaside Buffer (TLB). The TLB is essentially a cache for the page table. Due to the large size of the page table typical miss rate is only around $0.01 - 0.1\%$ (Negrutn, 2013).

The distinction between virtual and physical memory becomes important for memory transfer between CPU and GPU, which will be discussed in more detail in section 5.3.6.

4.2.2 Cache Coherence

Cache lines are constantly moving between the different levels of the memory hierarchy. The CPU will have a cache coherence protocol, which makes sure that all cores see the same values in memory. This means that when a core updates a cache line, it will invalidate any other copies. The next use of the cache line will thus be forced to reload the most recent value. A detailed explanation of how this is handled is beyond the scope of this thesis.

4.2.3 False Sharing

False sharing refers to a severe decrease in performance due to multiple cores competing for access to the same cache line. As explained above, data is moved between caches in chunks called cache lines. Let two different variables X and Y be stored in the same cache line. If core1 tries to access X and core2 tries to access Y they will both request the cache line to be stored in their L1 cache. As explained above, only one core may have control over a cache line at a time and will as such invalidate the other core's copy of the cache line. If each core updates X and Y multiple times, they will keep on invalidating the other core's copy of the cache line, which can severely impact performance.

A typical cache line is 64 bytes (Mattson, He, and Koniges, 2019). When making a for-loop run in parallel over an array with 8-byte values one might want to distribute at least 8 values to each core if the cache size of typical size (8x8 = 64). There is however no guarantee that each cache line will be filled with 8 values used by the same core, as values may be stored staggered.

Chapter 5

Graphics Processing Units (GPU)

5.1 Accelerator Programming

As discussed in the last chapter, the CPU is a very good general-purpose machine. The CPU works incredibly well on a single core due to the complex control logic and data management, but only a small fraction of the transistors is dedicated are used for actual computations. GPUs (and other accelerators) devote much fewer transistors to control and memory and instead focuses on throughput as is shown in figure 4.1. The GPU is more restrictive and should not be used for all applications, but can vastly outperform the CPU when parallelism can be exploited.

Most simulations work on large arrays where each index must be updated as time progresses. If an update of an array index does not depend on other indices (or at least only locally), it can easily be parallelized to run on an accelerator. There exist different types of accelerators, but the Graphics Processing Unit (GPU) is by far the most common option. The high peak performance makes GPUs incredibly useful in computational physics, where the data access patterns are regular and algorithms can be executed with the single-instruction-multiple-data paradigm.

This chapter will describe the basic architecture of the GPU, how to program it low-level and slowly build up to the more abstract high-level directive-based programming. It will focus on CUDA and NVIDIA terminology. For example, NVIDIA refers to a collection of cores on the GPU as a Streaming Multiprocessor while AMD refers to this as a SIMD unit. The different vendors use slightly different terminology, but the basic ideas remain the same and should translate fairly easily.

The CUDA part of the chapter is largely based on the CUDA C Programming Guide (NVIDIA Corporation, 2020).

5.2 GPU Architecture

The GPU as separate computer hardware was invented in the 90s as a tool to solve the toughest computational problem in mainstream computers: Rendering Graphics (McClanahan, 2011). NVIDIA was the first to release a 'true' GPU in 1999 and came to define a GPU as "`a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second`"(*GPU: Changes Everything*). 8 years later in 2007 NVIDIA released its CUDA development environment, which allowed the complex graphics rendering hardware to be used for general-purpose computing. The corresponding hardware is commonly referred to as General Purpose GPU (GPGPU). To understand how this works, the basic architecture setup of a modern GPU will be explained, before explaining the development environment, CUDA, and the more high-level approaches of directive-based programming.

Because of the vast amount of cores on the GPU, cores are partitioned into groups. Each group is called a Streaming Multiprocessor (SM). The important distinction between CPUs and GPUs is that instructions are carried out by the SM rather than individual cores. SMs are thus the GPU equivalent of a core. A SM is sketched in figure 5.1. As the figure shows, there are multiple cores in a single SM. The overall structure of the GPU is sketched in figure 5.2.

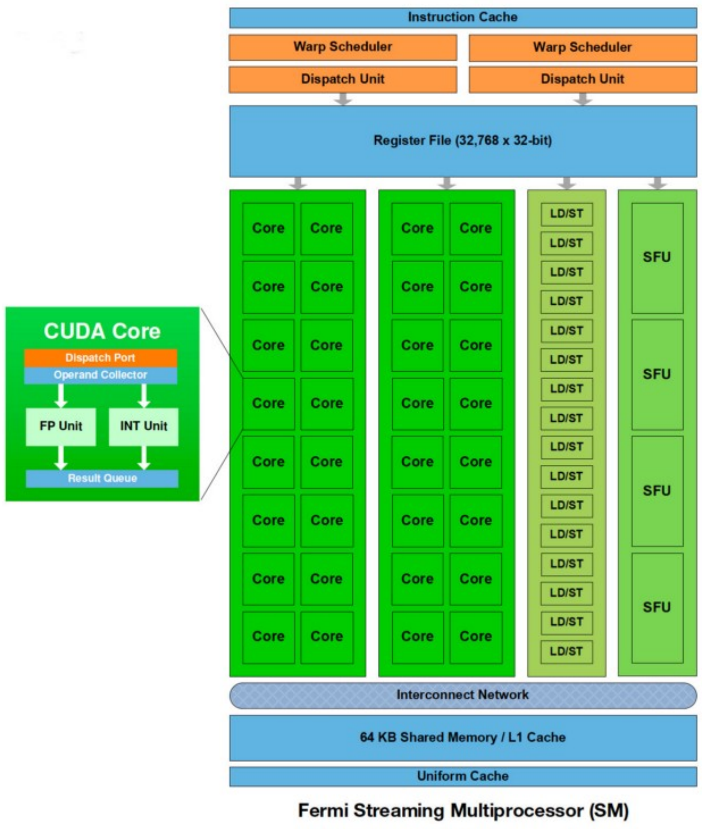


FIGURE 5.1: Streaming Multiprocessor overview (Negrutn, 2013)

Each SM has a dedicated amount of space for registers. This means that, unlike CPU cores, the cores here share register space. If one executes code that requires a lot of variables in registers per core, the cores can compete for register space, which heavily reduces the throughput. The register file on the new NVIDIA Ampere A100 GPU is 256KB, which corresponds to 32768 8-byte words per SM.

The cache/shared memory is much smaller than that of the CPU. This means that each core can only have a few hundred variables in L1 memory. If a variable is *simple* it will be better to recompute it when it is needed rather than persistently storing it.

When executing code on the GPU several threads will be spawned. Instructions to be carried out by each thread will then be mapped to individual cores. In the

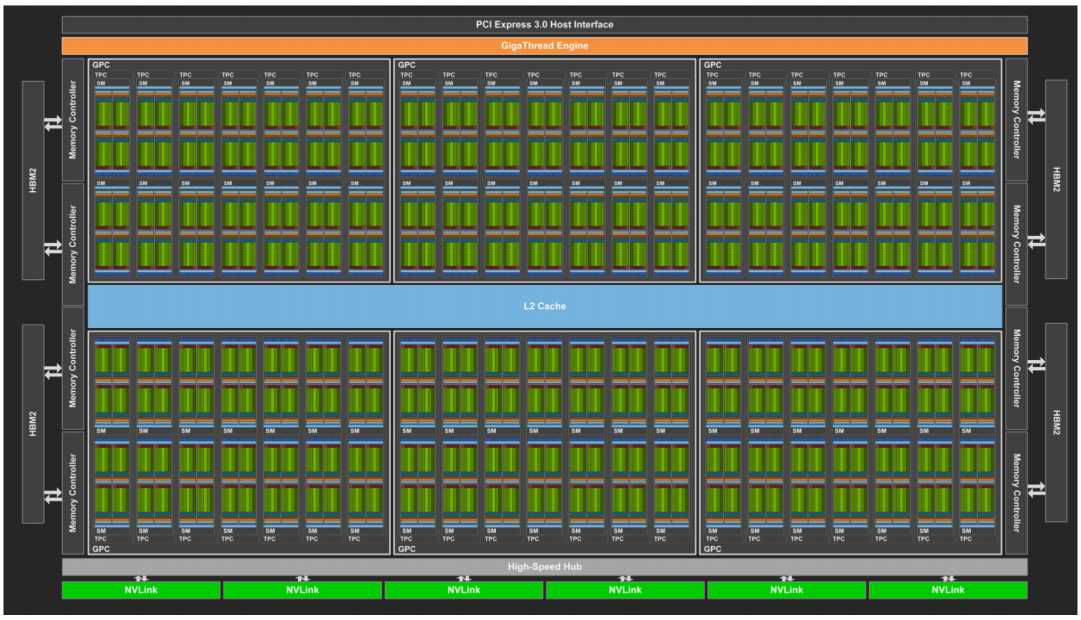


FIGURE 5.2: GPU achitecture (Dr. Momme Allalen, 2020)

remainder of the chapter, I will refer to threads rather than cores as the individual component carrying out instructions.

Threads a grouped for execution by the SM in chunks called `warps`. The size of a warp depends on the specific GPU architecture, but the standard is 32. A SM typically has 2 warp schedulers and the A100 may handle up to 64 warps at a time. This means that up to 2048 threads may be active in one SM. The warps are created by the warp scheduler and dispatched to hardware cores. When a warp is being executed 32 threads will execute the scheduled instructions before being terminated. Each execution will be referred to as "a thread being executed". An important thing to note here is that the warp will not terminate until ALL 32 threads are done with the work. if 31 threads take 1 second to complete, but the last takes 10 seconds, the 31 threads will remain idle until the last is done. Making sure that work is distributed roughly evenly is therefore very important when programming for GPUs.

Similar to the CPU, a warp may stall because it is waiting for data to be available. If only 1 warp was active this would have a huge impact on performance. However, since multiple warps may be active the latency can be hidden, by the warp scheduler picking up a different warp. Oversubscription is therefore important for good performance.

5.3 CUDA

CUDA or Compute Unified Device Architecture is an API model created by NVIDIA for targeting their GPUs for general purposes. While CUDA is not directly used in the project, the low-level CUDA functions are indirectly used when using directive-based programming. A basic understanding of the low-level features is therefore still useful when using high-level directive programming.

CUDA provides a level of abstraction, which allows the programmer to target the SM and cores of the GPU without close knowledge of the hardware. The overall structure of a CUDA program is shown in figure 5.3. To the left in the image is sketched the execution flow and to the right is the inner structure of the grid. In the

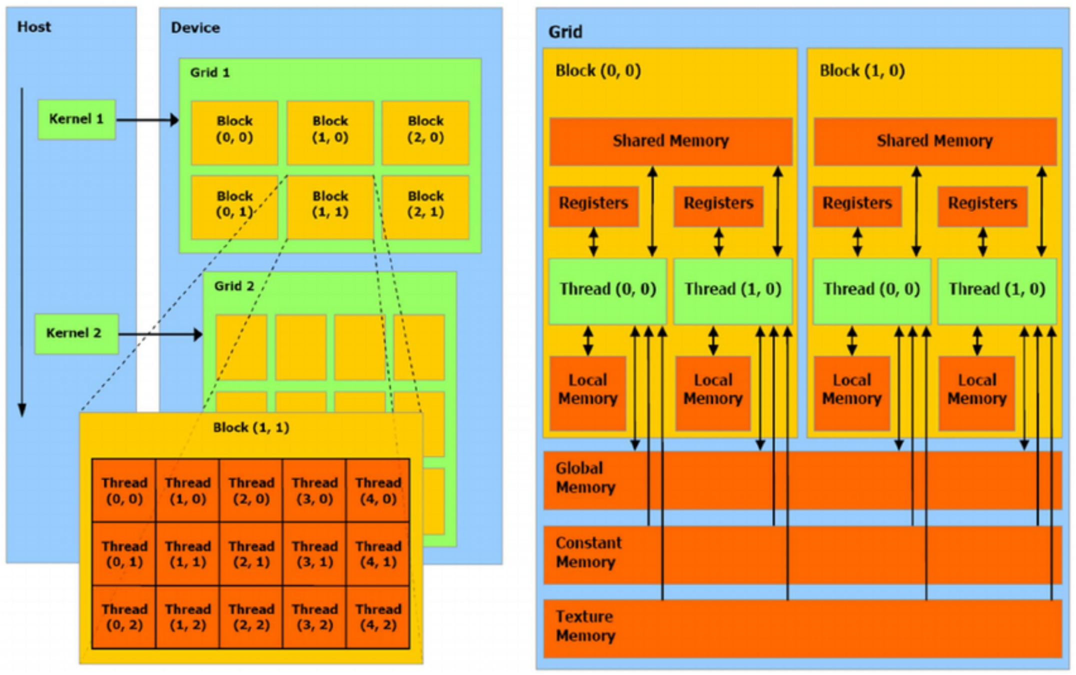


FIGURE 5.3: CUDA structure. Basic execution flow a CUDA program is shown to the left, and the inner structure of the grid is shown to the right. Processing units are divided into a grid of blocks, each with a unique block id. Blocks are further divided into threads with a thread id, which is unique in that block. Threads have shared memory in a block, while threads of different blocks can only communicate via global memory. Grid and block structure may be 1D, 2D, or 3D. (Nobile et al., 2014)

context of GPU programming, the `host` refers normally to the CPU. The host is the place from which `kernels` are dispatched. A kernel simply means 'A function that is run on the GPU'. The GPU (or any other accelerator) is referred to as the `device`. The left side of the image shows how a normal CUDA program is executed. The program will start on the host (CPU) and execute instructions. At some point, it will encounter a call to a kernel (GPU function). This will stop the execution on the host and move execution to the device (GPU). Once the kernel is completed the host will resume execution until another kernel is met or the program terminates. The host can continue in parallel with the kernel execution. This is achieved by running the kernels asynchronous, which will be discussed more below.

5.3.1 Loops in CUDA

CUDA follows the Single Instruction Multiple Thread (SIMT) execution model. This means that the same instruction will be performed by many threads. The same instruction is carried out by each thread on different data by using thread ID as an anchor for array indices. This means that CUDA programs will differ from regular programming. An example can be seen in listing 5.1. The example shows a simple function for initiating an array. In the normal Fortran function, a do-loop iterates over each index. In the CUDA version, the loop is removed. Instead, `i` is initialized as the `threadidx%x`, the thread number/index. By spawning 100 threads with a thread index from 1 to 100 the array will be initialized similarly to the normal Fortran program. The `attributes` keyword lets the compiler know that this is a device function. An important thing to note about the below example is that all data-movement-related CUDA steps are excluded, as this will be discussed later. The example serves only to show the syntax differences in a simple do-loop function and corresponding CUDA kernels.

```fortran
subroutine int_fortran(x)
    real, dimension(:) :: x
    integer :: i
    do i = 1,100
        x(i) = 0
end subroutine

attributes(global) subroutine int_cuda_fortran(x)
   real, dimension(:) :: x
    integer :: i
    i = threadidx%x
    x(i) = 0
end subroutine
```

LISTING 5.1: Fortran vs. CUDA format

5.3.2 Defining Grid and Blocks in CUDA

Making sure that the correct number of threads is spawned and data is distributed correctly is the next important part of CUDA programming. This is done by using the Grid-Block-Thread structure seen in the right part of figure 5.3. CUDA allows the programmer to define grids of blocks for each kernel. A grid can be 1D, 2D, or 3D and can contain several blocks. The blocks can similarly be 1D, 2D, or 3D and contain several threads. The thread id is local within a block. A thread can thus be uniquely referenced globally by using block id, grid size, thread id, and block size. A thread can also be locally referenced by using only thread id and potentially the block size. Each block cannot have more than 1024 threads.

The threads required for the above code can thus be initiated in multiple ways. For example, one could have a grid with a single block that has 100 threads in a 1D structure. It is also possible to have a grid with 10 blocks in 1D structure, each with 10 threads in 1D. The structure is specified in the call to the kernel as shown below in listing 5.2. Note that if the second option is used `int_cuda_fortran` should be changed to include the block id when specifying `i`. Another thing to note is that nothing is preventing the programmer from writing `int_cuda_fortran<<<1, 101 >>>`, which will result in an out-of-bound reference of x.

```fortran
!call int_cuda_fortran<<<grid, block >>>(function parameters)
call int_cuda_fortran<<<1, 100 >>>(x)
call int_cuda_fortran<<<(10,1), (10,1) >>>(x)
```

LISTING 5.2: Specifying grid-block structure

There is, assuming the thread/block id is used correctly, no semantic difference between the different ways of assigning block/thread id. The difference comes in performance. In the simple example above the difference will likely not be noticeable as an array of 100 values is small enough to be cached entirely in shared memory. For larger problems, this is not true and the difference can be significant.

As shown in figure 5.3 a block has a shared memory. On the hardware level, this corresponds to the shared memory / L1 cache of an SM. A CUDA block is thus restricted to have all threads running on the same SM. Since the shared memory is much faster than the global memory making sure that the data used within a block can fit in shared memory can have a big impact on performance. The programmer must adapt the grid-block structure to best fit the problem at hand. The programmer may choose the dimensions of the grid and block and as such the number of threads per block. The programmer can however not choose the number of blocks scheduled for each SM.

The best structure cannot be derived directly from the problem because the hardware plays an important role. A program optimized on the V100 GPU may for example not be optimal for the A100 GPU as the A100 has much larger available memory and cache, and thus would respond differently to the grid and block structure. Similarly, the number of blocks per SM may be differently scheduled on the two devices. When optimizing CUDA code it is, therefore, necessary to fine-tune the program to the target hardware.

As mentioned in section 5.2, the actual execution of threads happens in warps which, usually, consists of 32 threads. For this reason, it is custom that the number of threads within a block is some multiple of 32. Usually, far more threads than physical cores are spawned in a kernel. This allows the schedulers to keep multiple warps going on each SM at all times.

5.3.3 Handling GPU-CPU Memory Movement

The GPU comes with a large L2 cache and its main memory access is structured similar to the CPU memory model. This allows a large amount of data to be stored physically closer to the GPU cores making it much faster than if it had to rely on the CPU main memory. The GPU memory has its own distinct address space separate from the CPU main memory. When programming CPUs, the CPU and/or compiler handles all memory transfer from disk to main memory and caches. When programming CUDA code, the programmer has to explicitly handle the memory transfer from host to device. With CUDA 6 NVIDIA introduced a unified memory where host and device share address space. This functionality of CUDA has however not been used in the thesis as the simpler syntax can harm performance.

In Fortran, there are 2 ways to create device variables and arrays. The first more Fortran-like is to add `device` to the parameter specification. The second, more C like is to use the CUDA version of `malloc`, `cudaMalloc`. A simple example of this is shown in listing 5.3. The second option offers more control to the programmer about when exactly the arrays are initialized. There is however a bigger chance that something will go wrong, as we often see in C/C++ where pointers are not handled properly.

```fortran
subroutine host_func1()
    real, device,dimension(100) :: x,y
    call device_function<<<*,*>>>(x,y)
end subroutine

subroutine host_func2()
    real, device, allocatable :: x_ptr,y_ptr
    call cudaMalloc(x_ptr,100)
    call cudaMalloc(y_ptr,100)
    call device_function<<<*,*>>>(x_ptr,y_ptr)
    call cudaFree(x_ptr)
    call cudaFree(y_ptr)
end subroutine
```

LISTING 5.3: Initiating arrays/variables on the device

Memory can be transferred between host and device with the CUDA function, `cudaMemcpy`, which can move both from host to device and from device to host depending on the input parameters. A more advanced allocation function and memory movement function exists to target arrays or sub-arrays. It is also possible to make use of the constant or texture memory of the GPU, which is faster than both shared and global memory. These are however not used in the thesis and are therefore outside the scope of this discussion.

5.3.4 Latency Hiding

Memory movement from host to device is very time-consuming. Referring back to table 4.1, fetching memory from main memory on the CPU takes on the order of 100ns. Memory movement between host and device varies a lot but may be anything from the order of $10 - 10^3 \mu s$ (Lustig and Martonosi, 2013). Reducing memory reads is important in CPU programming. Taking into account the larger computational capacity of the GPU reducing transfer between CPU and GPU becomes even more important when programming for GPUs. Because of the large number of cores and the high percentage of transistors dedicated to computations, calculations on the GPU are almost instantaneous compared to the time spent waiting on the data to arrive. For this reason, the most important part of optimizing GPU code is to reduce the time spend waiting. This is done by so-called latency hiding.

Latency hiding is achieved by making sure the GPU is always busy. Instead of fetching data from main memory to GPU once the data is needed, the data should be moved before the kernel execution call. This way, when the kernel is called and data eventually needed, the data is already available on the GPU, and virtually no time is spent waiting for the data. In most programs, there will be multiple kernels. The CPU may continue without waiting on a kernel to terminate by using asynchronous execution and may continue executing other instructions. These instructions could include asynchronous memory transfer to the device, such that the data required by a second kernel is ready when the second kernel is called. Computations, memory transfer in, and memory transfer out of the GPU can proceed simultaneously.

5.3.5 Streams

A stream in CUDA is a sequence of instructions that execute in sequence. A stream will likely consist of some data movement and several kernels. Kernels and data movement can be put in a queue such that the execution on the host may continue while the stream is being executed asynchronously. Streams are used to handle concurrency. This is done by separating dependent kernels and data movement into the same stream. Kernels that have no dependencies can be put in separate streams and potentially be run concurrently on the device.

```fortran
!call int_cuda_fortran<<<grid, block,stream >>>(function parameters)
call int_cuda_fortran<<<1, 100,1 >>>(x)
call int_cuda_fortran<<<(10,1), (10,1),2 >>>(x)
```

LISTING 5.4: Adding stream to function call

Listing 5.4 shows the function calls from listing 5.2, but with the stream specified for both function calls. For the first function call stream one is used and for the second stream two is used. This means that both of these functions can be run concurrently by the two streams.

There is no software limit to the number of streams one can create, but there will be a hardware limit where additional software streams are mapped to the same hardware stream. This puts a limit on the number of concurrent streams. This will most likely not be a problem as most kernels take up a substantial portion of the GPU. Fx. if each kernel uses half of the available cores in the GPU, only 2 kernels

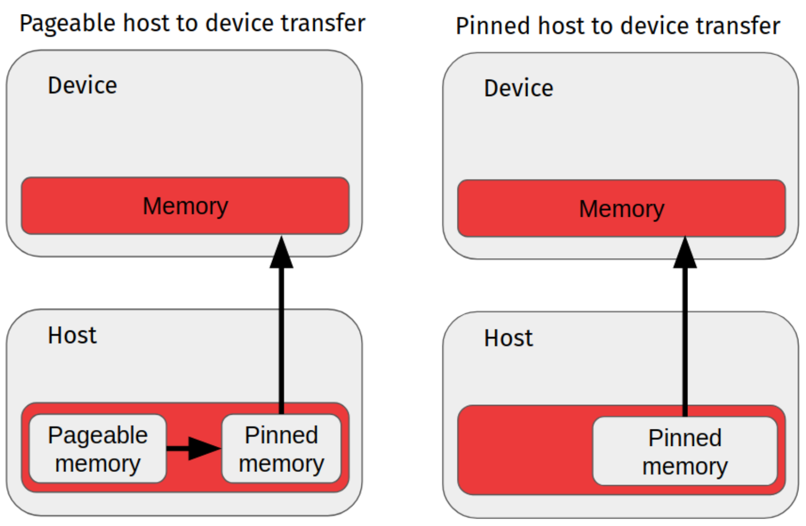


FIGURE 5.4: Difference between pinned and non-pinned memory transfer from host to device. More communication and data transfer are involved in non-pinned memory making it slower than pinned memory.

can be active at the same time even though the hardware might support 16 concurrent streams. In a scientific context with large simulations, the hardware limit on concurrent streams will not be the bottleneck.

5.3.6 Pinned Memory

Another important aspect in optimizing GPU code is the use of pinned memory. The general differences between a pinned and non-pinned data transfer are shown in figure 5.4. When transferring data to the device without the use of pinned memory, the requested data will first be moved in host memory to a pinned buffer. From here it will then be transferred to the device.

The reason for this extra step is that the memory controller on the device can only transfer memory for which the address space is contiguous in physical memory. As mentioned earlier, processes on the CPU will use virtual memory. The arrays (or other data types) will be contiguously located only in virtual memory, as discussed in section 4.2.1, but may be split into pieces in physical memory or perhaps even stored on disk. For this reason, the host must first move it to the pinned buffer. The pinned buffer is a portion of the main memory where the virtual address is also stored sequentially in physical space.

If pinned memory is used a larger portion of the memory is allocated as pinned and data is stored directly in the pinned buffer. When transferring data to the device, data no longer have to be moved to pinned memory. This can make the data transfer faster by orders of magnitude (Lustig and Martonosi, 2013).

Provided a large enough main memory, the use of pinned memory can speed up any GPU code for which the data transfer takes up a large portion of the total time. Since data transfer is often a bottleneck, this means that the use of pinned memory will in almost all cases give better results.

5.3.7 Bank Conflict

The shared memory that is available locally on each SM is divided into equally sized chunks called *banks*. This affects threads within a block, which may use the shared memory for fast memory access. Each bank can handle one request at a time. If n threads access n different banks all memory read/write can happen concurrently. Banks are not consecutive in memory. If there are 32 banks in shared memory, bank 0 may hold for example address 0,32,64,96 and so on. If multiple threads access the same bank, the requests will be serialized, which is called a bank conflict. Having the bank memory being non-consecutive makes bank conflicts less likely to occur. The problem with bank conflicts is similar to false sharing within multi-core CPUs. Bank size and structure differ from hardware and CUDA versions. Mei and Chu, 2017 discusses in more detail how bank conflicts occur and how to avoid them.

5.3.8 Recap of Optimal GPU Performance

When writing optimal GPU programs, the most important part is to hide the latency by transferring data before it is needed by kernels and programming the kernels in a cache-friendly way. Once this is achieved, pinned memory may be implemented to make the required memory transfers even faster. Finally, if kernels are not large enough to utilize the entire device, multiple streams can be used by non-dependent kernels. As a rule of thumb, you should always oversubscribe the device by having at least 4-10 times more active threads than available hardware cores.

5.4 Directive-Based Programming

CUDA offers great flexibility and control, but can be tedious to program and requires specifying separate functions for CUDA application, which cannot be run without a CUDA-enabled GPU. An alternative to CUDA is to use directive-based APIs such as OpenMP and OpenACC. These APIs are more abstract and high-level, which builds on top of the CUDA model without explicit use of CUDA.

Directive-based APIs work by adding special comments to the regions you wish to parallelize. When compiling, the compiler will then choose how to translate this into run-able device machine code. For example, a comment could be added to a for-loop specifying that you want the compiler to make the for-loop run in parallel. On a computer with a GPU, the compiler might choose to make GPU code for executing the for-loop. On a computer with no GPU, the compiler might choose to parallelize the loop using multiple CPU threads.

This means that the code remains portable to different computer architectures so that a single code base can be maintained, and the code does not have to be as fine-tuned just to run on the target device. It is also far easier to write a comment instructing the compiler to run something in parallel than to explicitly re-write the code to using CUDA thread/block id as the index.

Some compilers have today reached a state where they can optimize code almost as well as an intermediate programmer would be able to in CUDA (Li and Shih, 2018). For these reasons, it is highly favorable to make GPU implementations using directive programming.

The two following sections of this chapter will give a brief overview of OpenMP and OpenACC, which are the two directive-based APIs investigated in the thesis.

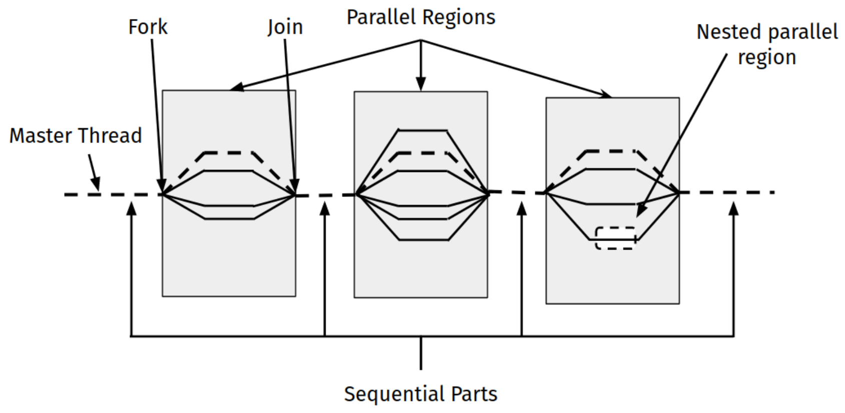


FIGURE 5.5: Fork-join execution of OpenMP program. Parallel regions may be nested within parallel regions.

5.4.1 OpenMP

OpenMP (Open Multi-Processing) is an API developed in the 90s to transform a sequential program into a parallel program. The original idea was for the parallel program to be run on a shared memory multiprocessor computer. Another key idea was that it should be done with minimal changes to original sequential code and keep both parallel and sequential code semantically equivalent.

```
!$omp parallel [clause,clause]
...
!$omp end parallel
```

LISTING 5.5: Parallel directive of a structured block (Mattson, He, and Koniges, 2019)

The basic structure of an OpenMP parallel region can be seen in listing 5.5. The code in this region will be run by every thread spawned by the compiler. The `clause` directive gives the programmer control over how the parallel region behaves. The clause could specify if a variable is private or shared among threads, or specify how many threads should be spawned. Some clauses, like the number of threads spawned, are only compiler suggestions and may be ignored by the compiler. The execution follows a fork-join pattern shown in figure 5.5. This approach is very similar to that shown in figure 5.3.

As mentioned above, certain clauses may control how data is stored for a parallel region. How data behaves in OpenMP is referred to as the OpenMP Data Environment. The three most important clauses for this is the `shared(list)`, `private(list)` and `firstprivate(list)` clauses. The `shared(list)` clause specifies that the variables in the `(list)` should be shared among threads. These variables will thus exist before and after the parallel region and changes made will persists. In OpenMP, variables defined before parallel regions are shared by default. Variables defined within the parallel region are `private` by default. If a variable is set as private in the parallel data clause, each thread will have a new variable with that name and type. If the variable is defined before the parallel construct, the value of the variable will not be copied to the thread. Variables defined before the parallel construct, which should be copied to each thread should use the `firstprivate` clause. This works similar to `private` but copies the value to each thread. For both `firstprivate` and `private` the changes made to the variables does not persist after the parallel region ends. It is good practice to set the `default(none)` clause that forces the programmer to explicitly set the scope of each variables as either `shared`, `private` or `firstprivate`. This makes sure that no private variables are accidentally treated as shared, or the other way around.

Since its development in the 90s multiple iterations of the API have been made with the newest being OpenMP 5.0. From OpenMP 4.5, which was released in 2015, the `target` directive has been supported. The target directive is added to the parallel directive to denote that this should be *offloaded to the target*. This is OpenMP jargon for *executed on a device separate from the CPU*. At the same time, the data environment now has a separate directive such that multiple parallel regions may share the same data environment. An example can seen in listing 5.6. The clauses for the data region, [to,from,tofrom], tells the compiler if an array or variable should be moved to the GPU, from the GPU, or moved to GPU and back after the data region has ended. The example also show that not only has the `target` directive been added, but also `teams` and `distribute`. As mentioned, the `target` directive tells the compiler that the region has to do with the device. The `teams` directive tells the compiler to create a number of thread 'teams'. The `distribute` tells the compiler that the corresponding for loop should be distributed among the teams. The directive "`target teams distribute parallel do`" thus tells the compiler that it should create a league of teams on the device, distribute the for-loop in chunks across the teams, and execute those chunks in parallel on the threads within a team.

```
!$omp target data [to,from,tofrom]
!$omp target teams distribute parallel do
do i=1,N
    ...
end do
!$omp end target teams distribute
!$omp end target data
```

LISTING 5.6: Target and target data directive for OpemMP

The `teams` clause roughly corresponds to the blocks in CUDA. Each team will be a block. With the `distribute` clause the workload in the parallel region will be distributed across the blocks. The `parallel do` within a target region corresponds to the parallelism within a block. That is, `teams distribute` determines the `grid` variable in listing 5.2 and `parallel do` determines the `block` variable in the same listing.

When parallelizing loops it is important to consider how the clauses translate into CUDA (or CUDA-like) code. If multiple nested loops are present, it is usually best practice to surround the outermost loop with the `target teams distribute` statement. The innermost loop should, if applicable, be parallelized with the `simd` statement. Intermediate loops should be collapsed and parallelized as shown in listing 5.7.

```
!$omp target data [to,from,tofrom]
!$omp target teams distribute
do i=1,N
    !$omp parallel do collapse(2)
    do j=1,N
       do k=1,N
            !$omp simd
            do l=1,N
                ...
            end do
        end do
    end do
end do
!$omp end target teams distribute
!$omp end target data
```

LISTING 5.7: Optimal parallel structure of loops

5.4.2 OpenACC

Unlike OpenMP, which in the beginning was concerned only with multi-core parallelism, OpenACC has since its beginning been focused on directive-based programming for heterogeneous computing (computing on more than one type of system). OpenACC 1.0 was announced in 2011 and has supported offloading code to an accelerator since the beginning. The syntax of directives is very similar to that of OpenMP. Listing 5.8 shows the OpenACC version of listing 5.6.

```
!$acc data [to,from,tofrom]
!$acc parallel do
do i=1,N
    ...
end do
!$acc end parallel
!$acc end data
```

LISTING 5.8: OpenACC parallel and data region

As the example shows there is nothing to differentiate between accelerator parallelism and CPU parallelism in OpenACC. How the directive should be interpreted is handled by a compiler flag that tells the compiler what the 'target' is for offloading. The compiler flag `-ta:tesla::cc75` will write GPU code for an NVIDIA(Tesla) GPU with compute capability 7.5. The compile flag `-ta:multicore` will instead create parallelism on the CPU, similar to OpenMP without `target` directives.

The fact that OpenACC does not have the `target` directive can somewhat simplify the code, but it does give the user less control. The simplicity of OpenACC has also been one of the main goals of the API. The lack of `target` directive, means that OpenACC cannot be used to achieve both host and device parallelism within the same program. Both host and device parallelism can still be achieved by specifying host parallelism with OpenMP and device parallelism with OpenACC.

Similar to OpenMP's `teams`, OpenACC has different levels of device parallelism, which may be put in front of the parallel statement in listing 5.8. OpenACC has 3 levels of parallelism: `gang`, `worker` and `vector`. Additionally `seq` may be used to explicitly specify sequential execution. The `gang` directive have an almost one-to-one correspondence with the `teams` directive from OpenMP, and specifies parallelism on a CUDA block level. `worker` specifies parallelism on the warp level and `vector` on the threads level.

Comparing OpenMP and OpenACC

While OpenACC as a whole is much younger than OpenMP, its support of accelerator offloading is older. As a result, most accelerator compiler implementation issues have been solved in OpenACC, while many remain in OpenMP. OpenACC offers a more fine-grained control, as one can control both block, warp, and thread-level parallelism with OpenACC. In OpenMP, only block and thread-level parallelism can be directly used.

OpenACC was originally developed by PGI and was intended as a multi-platform standard. But since PGI was acquired by NVIDIA it has been primarily developed for NVIDIA devices. While it is possible to compile OpenACC for other devices using other compilers, it may not perform as well as with NVIDIA GPUs. OpenMP on the other hand has always had broad industry support and is less dependent on the device on which it runs. In recent years more and more vendors are starting to use AMD. In addition, many next-generation supercomputers use AMD GPU, which makes OpenMP the more favorable choice.

Chapter 6

Implementation

6.1 DISPATCH Mockup

To test the OpenMP and OpenACC features a mockup of DISPATCH was created by my supervisors. The mockup consisted of three different files. The content and purpose of these files are here briefly described.

6.1.1 Mockup.f90

This file consisted of the main program to be executed. In addition, a linked list was implemented. The linked list was used as a substitute for the task-scheduling found in DISPATCH. In the earliest implementation, each task was popped from the list, updated, and then pushed back onto the list. Each list item contained a pointer to a solver-type object as in DISPATCH. There was no parallelism on the CPU.

6.1.2 Muscl.f90

This file consisted of a simplified version of the MUSCL-type HD solver. An `update` subroutine based on the MUSCL routine was created as seen in listing 6.1.

```
SUBROUTINE update (self)
  class(solver_t):: self
  call alloc   (self)
  call ctoprim (self)
  call slopes  (self)
  call predict (self)
  call riemn3d (self)
  call divflux (self)
  call dealloc (self)
  self%time = self%time + self%dtime
END SUBROUTINE update
```

LISTING 6.1: Update subroutine in early mockup

In this subroutine everything is still object oriented. `alloc` allocates the needed temporary arrays. `ctoprim` converts the conserved variables to primitive variables. `slopes` performs the slope limiter step, `predict` makes prediction for the values at time $t + \frac{1}{2}\Delta t$. `riemn3d` calculates left/right values and subsequently call the Riemann solver in each direction. `divflux` then uses the calculated flux to update the conserved variables, and lastly `dealloc` deallocates all temporary arrays.

6.1.3 Riemann.f90

This file consisted of the simple HLL solver, which was used in the early phases of the project.

6.2 Proof of Concept

The first implementations using both OpenACC and OpenMP focused solely on the feasibility of using only directive-based approaches for targeting GPUs. As such the correctness of the code was not investigated, and many positivity preserving steps were left out. Only the core-subroutines from listing 6.1 was used. Initially, OpenACC was investigated as OpenMP did not have sufficient support. Later in the thesis, GCC released an update, and the focus switched to OpenMP, which remained the main focus throughout the thesis. In this section, the iterative steps towards an optimized HD code are explained.

Throughout this chapter, meta-results of the implementations and compilers are also reported. This includes for example, how easy compilers are set up and made ready or how compiler implementations differ from OpenMP or OpenACC specifications.

6.2.1 OpenACC

The OpenACC implementations were compiled with the Portland Group Compiler (PGI). Early in the project the community edition 19.10 was used. As PGI got integrated into NVIDIA HPC SDK the 20.9 and 21.2 versions were used instead. No differences in performance were found between version 19.10 and 20.9, but version 21.2 that used cuda 11.2 performed significantly worse, and version 20.9 was therefore used. A step-by-step guide to setting up the two versions can be found in the appendix B.1.

OpenACC v0.1

The first implementation aimed to examine the basic challenges of directive-based approaches in modern Fortran. `!$acc parallel loop collapse(n)` was added to each loop where `n` is the number of nested loops. This approach was by no means optimal but was an easy way to get the majority of the calculations ported to the GPU. In the `update` subroutine, a data-region was created covering all subroutine calls. This reduced the required memory transfer between host and device between each kernel calls.

An important issue was discovered. OpenACC does not handle object-oriented programming well. The `self` object may be moved to the device, but internal subroutines and arrays are not properly moved, and/or pointers still refer to the CPU subroutine/array. To circumvent this limit, arrays and variables used within each subroutine were instead passed directly to the subroutine. Arrays were passed as pointers, as these seem to be better handled in OpenACC (this was later found to be redundant).

The `update` subroutine was therefore changed to become slightly more complex. A snippet of the update subroutine can be seen in listing 6.2. The full update subroutine can be found in appendix C. As shown, a pointer to each array is created and new copies of all variables are stored. These are then passed directly to the individual MUSCL solver subroutines. The data regions ensure that the arrays are only moved between host and device once per update.

```fortran
subroutine update(self)
        class(solver_t),target :: self
        integer:: i3, i2, i1
        real, dimension(:,:,:,:,:,:), pointer:: mem
        real, dimension(:,:,:,:), pointer :: prim
```

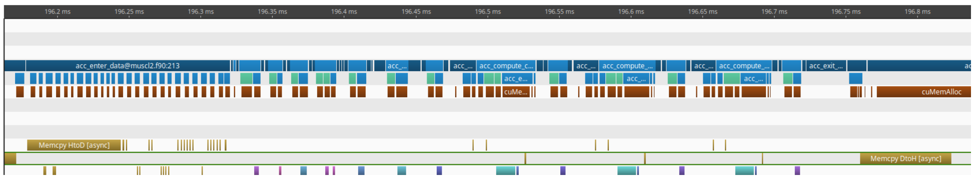


FIGURE 6.1: Profiling of the first OpenACC implementation with the use of NVIDIA's visual profiler, nvvp. There is substantial overhead in each kernel call and delay between Riemann kernels(light blue) because of host-to-device communication.

```
        ...
        mem => self%mem
        prim => self%prim
        ...
        !$acc data copyin(new,...,prim, grad, left, rght, flux, mem)
        call ctoprim (self, mem, new, it, prim, lb, ub, g2, u2_max)
        call slopes  (self, prim, grad, nv,  l, u)
        ...
        !$acc end data
end subroutine update
```

LISTING 6.2: Update subroutine in first OpenACC version.

A program consisting of a single update was profiled and a snapshot can be seen in figure 6.1. The profiler shows significant delay, or overhead, between each kernel call. In addition, some kernels have host-to-device transfer, which should all be handled at the beginning of the update. As a result, the GPU is not fully utilized. The next iteration tried to solve this overhead issue.

OpenACC v0.2

This version aimed at reducing the overhead by combining kernels. As described earlier, each loop was simply run as an individual kernel. Within a single subroutine, multiple loops may instead be run within the same parallel region. However, to run loops in different subroutines in the same kernel the subroutines had to be compiled as device subroutines. This was achieved by adding `!$acc routine` at the start of each subroutine call. The directive can be called with the clauses `gang`, `worker`, `vector`, `seq`. Each clause defined the level of parallelism for which the routine is intended as explained in section 5.4.2. An example of how this was used in the implementation is shown in listing 6.3.

Contrary to initial findings in v0.1 it was found that some object-oriented features were supported, and thus the subroutines could be called with the `self` variables. All subroutines in the `muscl.f90` file except `riemann3d` was listed as device subroutines. The loops in `riemann3d` had pragma statements similar to v0.1 and the call to the HLL solver was likewise made on the CPU.

```
SUBROUTINE predict (self)
    !$acc routine gang
    ...
    !$acc loop gang
    do ..
    end do
end subroutine predict
```

LISTING 6.3: Compiling subroutines with loops for device execution

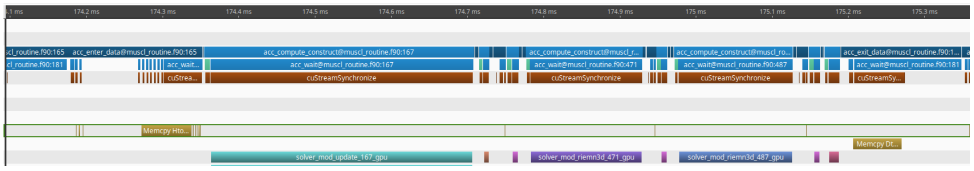

174.2 ms
174.3 ms
174.4 ms
174.5 ms
174.6 ms
174.7 ms
174.8 ms
174.9 ms
175 ms
175.1 ms
175.2 ms
175.3 ms
acc_enter_data@muscl_routine.f90:165
acc_compute_construct@muscl_routine.f90:167
acc_compute_construct@muscl_r...
acc_compute_construct@muscl_ro...
acc_exit_data@muscl_routine.f90:1...
acc_wait@muscl_routine.f90:167
acc_wait@muscl_routine.f90:471
acc_wait@muscl_routine.f90:487
acc_wait@muscl_routine.f90:181
cuStreamSynchronize
Memcpy Hto...
Memcpy Dt...
solver_mod_update_167_gpu
solver_mod_riemn3d_471_gpu
solver_mod_riemn3d_487_gpu

FIGURE 6.2: Profiling of the second OpenACC implementation with the use of NVIDIA's visual profiler, `nvvp`. Much of the idle time has been cut off, but the individual kernels take longer.

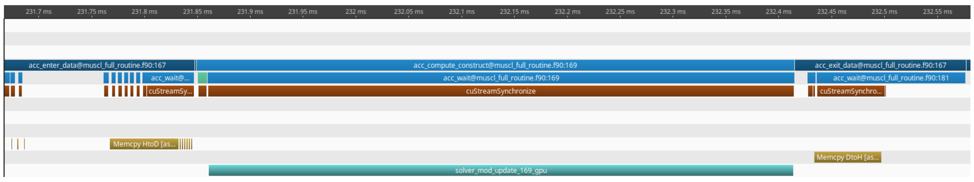


FIGURE 6.3: Profiling of the third OpenACC implementation with the use of NVIDIA's visual profiler

Similar to v0.1 a profiler was run on a single update. The snapshot of the profiler can be seen in figure 6.2. Fewer kernels are executed, and as a result, the kernel launch overhead is reduced. However, the execution time of the kernels is higher.

Combining multiple subroutines into one kernel was later found to produce race conditions. Race conditions occur when fx. some threads would begin executing the loops in `predict` before all threads had finished executing the loops in `slopes`. As OpenACC provides no way to enforce that the same thread executes the same index in different loops, the only alternative is to have barriers between each subroutine.

OpenACC v0.3

In the third implementation, all subroutine calls in `update` were moved to one kernel. This was done by creating a new subroutine in the `riemann.f90` file, which was not object-oriented. As seen in figure 6.3 this removed all host-to-device communication during the update, but the kernel itself had a longer execution time. This approach also further increased how often race conditions could occur, and it was decided to not pursue this approach any longer.

OpenACC v0.4

This implementation took its starting point in v0.2. Instead of having multiple subroutines contained in the same kernel, each subroutine was called in separate parallel regions. In the parallel region, the pragma statement `async(self%task_number)` was added. Each patch has its unique ID, which was referenced with `self%task_number`. Each parallel region in an `update` thus had the same task number/ID. The subroutines were therefore placed in the same `async` queue. Each queue corresponds to a stream in CUDA jargon.

Because `async` is used the CPU will send the kernel corresponding to the parallel region to the GPU and continue. With a parallel region around each subroutine, each kernel will be sent to the GPU before anything has been computed. At the end of the `update`, `!$acc wait(self%task_number)` was added to make sure all kernels in the queue had completed before finishing the update.

The use of the `async` queue almost eliminated the overhead between kernel calls. This is seen in the profiler snapshot in figure 6.4. Note here that the HLL Riemann

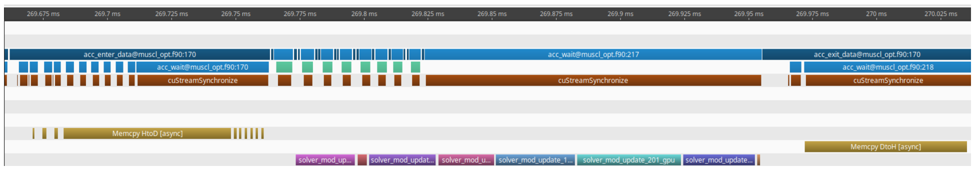

acc_enter_data@muscl_opt.f90:170
acc_wait@muscl_opt.f90:217
acc_exit_data@muscl_opt.f90:170
acc_wait@muscl_opt.f90:170
acc_wait@muscl_opt.f90:218
cuStreamSynchronize
cuStreamSynchronize
cuStreamSynchronize
Memcpy HtoD [async]
Memcpy DtoH [async]

FIGURE 6.4: Profiling of the fourth OpenACC implementation with the use of NVIDIA's visual profiler. The kernel launch overhead is largely gone, and the GPU is active a larger percentage of the time. There is still no latency hiding.

solver was only run in the x-direction. The duration of each kernel is roughly the same as for v0.1.

Although `async` is used to send kernels asynchronously to the GPU, no kernels are being executed in parallel on the GPU. This was the focus of the next iteration.

OpenACC v0.5

The main focus of this implementation was to add parallelism both on CPU and GPU. No changes were made to the file containing the MUSCL update routine or the HLL solver. In the previous implementations, each task was popped from the list, updated, and pushed back if the time did not exceed the end time set by the experiment.

To have multiple threads accessing the same list the mockup had to be changed. A pseudocode showing the general idea is shown in listing 6.4. A parallel region was created with a `single` region inside. The thread executing the single region will be referred to as the master thread.

The master thread is responsible for generating tasks that can be executed by other threads. A nested while-loop pops n tasks and stores them in a temporary list. This is referred to as a *bunch*. After this an `OpenMP` task is created for which the temporary list is given as a `firstprivate` variable

Two nested loops are required as the master thread will finish generating tasks before the first task is finished. If only a single loop is used the global list will then be empty and the loop will terminate. If the `taskwait` clause was within the inner loop only one task would be generated before waiting for all tasks to finish. This implementation does create artificial wait latency between updates for different task times. The idea was to get closer to the task-based scheduler currently implemented in DISPATCH.

Each task will pop patches from the temporary list and call the `update` routine. If the task time remains below a given end time they will be pushed back to the temporary list. After all tasks have been updated the temporary list is pushed back onto the global list. This is done rather than to push each task to the global list individually, to reduce the overhead generated by the critical region.

A profiler was run on this implementation and a snapshot is shown in figure 6.5. The test was done using 5 threads, so only 5 streams are active at the same time. The setup allows kernels and data transfers to be performed concurrently. While not visible in the snapshot, the latency caused by the `taskwait` statement was visible and had a noticeable impact on performance.

```
!$omp parallel
!$omp single
do while (full_list not empty)
    do while (full_list not empty)
```

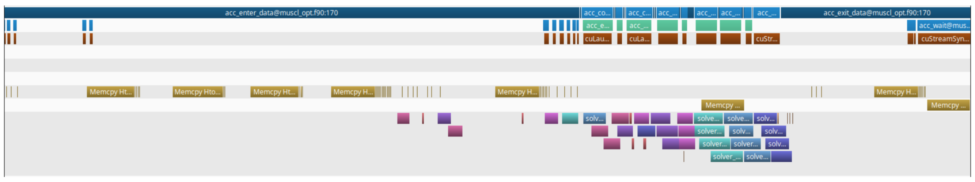

acc_enter_data@muscl_opt.f90:170
acc_exit_data@muscl_opt.f90:170
acc_wait@mus...
cuStreamSyn...
Memcpy HtoD
Memcpy ...
solve...

FIGURE 6.5: Profiling of the fifth OpenACC implementation with the use of NVIDIA's visual profiler. Multiple kernels are being executed at the same time, and data transfer is happening while some kernels are executing.

```
        !$omp critical
        temp_list = pop n elements from full_list
        !$omp end critical
        !$omp task firstprivate(temp_list)
            do while (temp_list not empty)
                link = pop temp_list
                call link%task%update
                if link&task%time < end_time
                    append to temp_list
            end do
            !$omp critical
            append temp_list to full_list
            !$omp end critical
        !$omp end task
    end do
    !$omp taskwait
end do
!$omp end parallel
!$omp end single
```

LISTING 6.4: Pseudocode for parallel use of linked list

OpenACC v0.6

This implementation improved v0.5 by using pinned memory. Nothing was changed in the files as compared to v0.5 but the compiler flag `-ta=tesla:pinned` was added.

6.2.2 OpenMP

Once the initial OpenACC implementations had shown that porting to GPU and hiding much of the memory latency is possible, investigation of OpenMP implementation took place. This was originally not deemed feasible, but with the release of GCC 10.1 in May and 10.2 in July 2020, new offload features were added. In addition, the announcement that LUMI would feature AMD GPUs made OpenMP a priority. A step-by-step guide to installing GCC with offload can be found in appendix B.2.

OpenMP v0.1

This followed a similar approach to OpenACC v0.5. The `mockup.f90` file was set up to pop a `bunch` of tasks and the tasks were updated one by one. Unlike OpenACC, OpenMP does not handle object-oriented features at all. For example `!$omp target data map(to:self%mem)` does not compile with GCC. Instead, temporary pointers had to be created for each array similar to OpenACC v0.1. This was a common approach used through the rest of the OpenMP implementations, which allowed a somewhat indirect use of "object-arrays". Each subroutine was called on the CPU, and a target region was put around each loop.

Despite using almost the same setup as the OpenACC version, this implementation was much slower. There was a much larger kernel execution overhead and the kernels themselves were much slower. Each kernel launch was accompanied by a host to device transfer. This was not the case for OpenACC, where all memory transfer could be done in the beginning of the update. In addition, asynchronous execution is not supported in GCC. This meant, that even with multiple CPU threads running, only one stream was used. Using multiple CPU threads did hide some of the kernel execution overhead.

OpenMP v0.2

This approach attempted to remove the overhead by executing the entire bunch in one kernel. The target and target data region was moved to the `mockup.f90` file. The target region was placed around a loop, calling `update` for each patch in the bunch. `update` was changed to be fully imperative and now took the required arrays and variables as input.

This approach however never got fully functional as several issues were discovered. Initially, `target teams distribute` was placed around the loop. This meant that a `team` of threads would be associated with each patch, and the teams distributed across the available SM's. Although much effort was put into bug-fixing, memory errors kept occurring. These errors seem to arise when a single kernel is either using too many registers or too many separate arrays within a single kernel.

Another approach was tested, where the `teams distribute` clause was removed in `mockup.f90` and instead put together with the parallel clause. While this compiled and ran without errors it did not produce the correct result. In fact, with this implementation, nothing was done in the kernels. This approach also showed another flaw in GCC. Although the result was not correct, the approach was run for several iterations. After running for some time, out-of-memory errors would occur. It was discovered that the arrays allocated with OpenMP statements inside OpenMP tasks were not properly deallocated, and memory usage would build up. This was confirmed by also running OpenMP v0.1 for an extended period of time.

OpenMP v0.3

The issues from v0.2 were solved in two different ways. First, the issue regarding memory errors was solved by going back to having a kernel for each subroutine inside `update`. However, each loop was extended to also loop over all patches in the bunch. In this way, primitive variables for all patches were calculated in `ctoprim`, before any patch started calculating the slopes. This approach allowed full utilization of the GPU in each kernel but required more memory. Additionally, with large bunch sizes, the overhead became insignificant, and the drawbacks of having a kernel for each loop were removed.

Secondly, the issue regarding memory leak was fixed by having static arrays. All arrays were extended to now have two additional indices. One bunch index and one thread index. Once an OpenMP task began executing its thread id could be used to only access and move part of the global static arrays.

6.3 HLLD

As the HD implementations indicated that a purely directive-based approach was feasible with both OpenACC and OpenMP the focus of the project shifted to the more complex MHD implementation. Initially, the implementation was done analogous to the simpler HD implementation. However, several positivity preserving steps had been left out and these now had to be incorporated into the mockup. In DISPATCH/RAMSES, most of the preparation happens in the `trace3D` subroutine. `trace3D` consists of a large loop that calculates the required variables for calculating the fluxes in HLLD 1D and the EMFs in HLLD 2D. This large loop uses many different arrays and around 70-80 loop private variables. This is not an issue on the CPU, but several issues arose on the GPU. The A100 GPU has around 255 available registers per thread. While this is larger than the 80 used loop variables, the compiler may choose to use more registers. As mentioned in the previous section, it was found that using this many variables in combination with many different arrays causes runtime memory errors.

The subroutine, therefore, had to be re-written by splitting it up into multiple smaller subroutine, which could be run as separate kernels, similar to the HD implementation. This imposed several new problems. In `trace3D` each iteration of the loop is independent of other iterations. By splitting up the loop dependencies across kernels arises. In addition, as discovered during HD implementation OpenMP comes with a large overhead when creating kernels. Splitting one loop into multiple kernels thus created additional overhead. In the later stages of implementation, it was found possible to recombine the kernels, eliminating almost all overhead from kernel execution. The rest of this subsection will explain the steps taken to port the existing DISPATCH/RAMSES MHD solver to the GPU.

6.3.1 Code Refactoring

Dimension Representation

In the HD setup, the main arrays containing data about the fluid variables and derivatives are indexed as `mem(x,y,z,nv,ii,tid)` and `grad(x,y,z,nv,dir,ii,tid)`. Here (x,y,z) references patch dimension, nv references variables (density, momentum etc.), ii references bunch index and tid references thread index as explained in the HD section. `prim` was stored similar to `mem` and `[flux,left,right]` was stored similar to `grad`. Fortran stores arrays column-wise. As density, momentum and energy are often used in the same iteration, this way of storing the loops is not cache friendly.

A small test program was created to test several permutations of the dimension representation. The test program simply loaded a value from an array, added an integer, and stored the new value back into the array. This was done for each index in the array multiple times and the total time for each permutation was stored. Based upon the result, dimension was instead stored as `mem(x,nv,y,z,ii,tid)` and `grad(x,nv,dir,y,z,ii,tid)`. When running the innermost loop (over the first dimension) as SIMD, this setup performed much better.

In addition the scheduling clause `schedule(static,1)` was changed to `schedule(static)`, as this too is more cache-friendly. These changes allowed an entire bunch update to be performed by a single kernel.

The arrays storing left and right values were also merged into one array with the structure `left_right(x,nv,lr,dir,y,z,ii,tid)`, where `lr` is 1 for left and 2 for right. This made sure that both left and right variables were always close in memory as they are always used together. The corner values were stored in a similar array.

Furthermore, instead of storing left and right interface values in index (x,y,z), the left and right Riemann variables were stored. This simplified the access to the left and right variables in the HLLD 1D subroutine.

Refactoring Trace3D

In the current version of DISPATCH/RAMSES HD and MHD solvers, the conversion to primitive variables and calculation of slopes happens before `trace3d` is called. Everything else needed in the Riemann Solvers happens in `trace3d`. First, EMF is calculated based on time t, which is used to predict magnetic fields at time $t + \frac{1}{2}dt$. For convenience, slopes are expressed as differences. Magnetic field and primitive variables are then predicted at time $t + \frac{1}{2}dt$. From these predicted values, differences are used to infer the face-centered values stored in the `left` and `right` arrays (`qp` and `qm` in `DISPATCH/RAMSES`). Lastly, corner values are calculated.

All steps described above happen in one single loop. The first step towards a working GPU code was to split this into several semi-independent steps. Each step below corresponds to a subroutine:

1. Convert conservative to primitive values and calculate u_{max}.
2. Calculate slopes.
3. Calculate EMFs for time t.
4. Use EMFs to predict magnetic fields at time $t + \frac{1}{2}dt$.
5. Infer left and right magnetic field values.
6. Predict primitive variables at $t + \frac{1}{2}dt$.
7. Infer left and right primitive variables.
8. Predict corner values for primitive variables.
9. Predict corner values for magnetic fields.
10. Calculate fluxes in HLLD 1D based on left/right values.

11. Update conservative variables based on fluxes.
12. Calculate EMFs in HLLD 2D based on corner values.
13. Update magnetic fields based on EMFs.
14. Convert conservative to primitive values and calculate u_{max}.

In DISPATCH/RAMSES conversion to primitive variables and slopes are not called in `trace3d`, but these are calculated on the GPU and therefore included in the above steps. Furthermore, primitive variables are calculated twice: both in the beginning and in the end. The reason for this is that the timestep is normally calculated after conversion to primitive variables. However, in the GPU version, the timestep has to be calculated before using existing CPU subroutines. Therefore u_{max} must be calculated at the end of each update so the newest value is used when calculating the next timestep.

Some of these subroutines could be fused. For example, corner values could be predicted in a single subroutine. However, sometimes memory errors would occur. As explained in section 6.2.2, these errors seem to arise when using too many variables or arrays in the same subroutine. Therefore calculations on primitive variables and magnetic field were separated as much as possible. This allowed as few arrays as possible to be used in any given subroutine.

Originally, a target region was created around each subroutine, and the subroutine calculated for several patches. This allowed the GPU to have plenty of data and made sure each step had been completed on all patches before moving to the next. This did however introduce overhead to each kernel call.

The overhead was removed by having one single target region covering all steps. Each subroutine was changed to only update one patch based on a patch index. This also allowed the subroutines to be called separately on the CPU without bunching. All subroutines were placed inside a loop. The `omp distribute` clause was used to distribute each patch update across the available SMs. This approach was originally found to not work in OpenMPv0.2, but with the new array representation, it no longer produced memory errors.

DISPATCH/Ramses Errors

During implementation, some minor algorithmic mistakes were discovered in the MHD solver used in both DISPATCH and RAMSES. This is interesting, since the MHD solver has existed in RAMSES since 2002, and has been used in more than 1000 articles.

First, when calculating corner values and transverse slopes for the magnetic fields the first and second transverse directions are flipped in the y-direction. If a right-handed coordinate system is maintained, the z-direction should be the first transverse direction and the x-direction the second. However, in DISPATCH/RAMSES it is the other way around. For the magnetic field, it has no effect as the values are stored and loaded in the same way, so the correct values are still used. However, it has an effect when calculating corner values. Here, the RB and LT values are flipped in the y-direction, and this is not properly accounted for when transforming LB, LT, RB, and RT to SW, NW, SE, and NE values for the HLLD 2D.

Second, for the EMF found in HLLD 2D the values are not multiplied with the correct dx, dy, dz. In DISPATCH/RAMSES EMF_x is multiplied with dt/dy, EMF_y is multiplied with dt/dz and EMF_z is multiplied with dt/dx. This is done as the cell size is assumed to be the same in each direction. However, if the number of cells/patches is not identical in each direction small numerical differences will arise. These errors will have an effect over time.

Symmetry in Trace3d

The `trace3d` subroutine uses separate code for each direction effectively increasing code length by a factor of three. This avoids advanced index manipulation by coding explicitly for each direction. However, this makes the code more difficult to read and increases the likelihood of errors as future improvements on the code will have to be added in three different places.

The GPU implementation has in many places taken the opposite approach and instead relied on index manipulation to reduce repeats in the code. This has mainly been done using six small arrays: `j1(1,0,0)`, `j2(0,1,0)`, `j3(0,0,1)`, `v1(2,3,4)`, `v2(3,4,2)` and `v3(4,2,3)`. The j-arrays are used when ± 1 is added to the x-, y- or z-dimension. For example, `prim(x+j1(dir),nv,y+j2(dir),z+j3(dir),ii)` may be used to add 1 to the x dimension, if $dir = 1$, or 1 to y direction if $dir = 2$ and 1 to z dimension if $dir = 3$.

The v array's are used to keep track of the velocities and magnetic field. The array values are assumed to be stored as: $(\rho, vx, vy, vz, p, Bx, By, Bz)$. `v1` always hold the index of the normal velocity. `v2` always hold the index of the first transverse velocity, and `v3` the index of the second transverse velocity. As such if $dir = 1$ the used values will be $v1 = 2, v2 = 3, v3 = 4$. If $dir = 2$ the values will be $v1 = 3, v2 = 4, v3 = 2$ and if $dir = 3$ the values will be $v1 = 4, v2 = 2, v3 = 3$. Similarly, magnetic field indices are found by just adding $+4$.

Optimizing HLLD 1D for GPU

The inner loop of the HLLD 1D solver remains more or less unchanged. There are only two differences between the GPU implementation and DISPATCH/RAMSES. First, the DISPATCH/RAMSES version takes as input a 1D array whereas the GPU version is run on the entire patch. This is done to avoid having to copy left and right variables into temporary 1D arrays. Instead, the direction is added as a loop. In each direction the HLLD solver expects the "normal" velocity to be located at index `iu`. In the x-direction this means U is stored in `iu`, in the y-direction V is stored in `iu` and in the z-direction W is stored in `iu`. Similarly first and second transverse direction are always stored in `iv` and `iw` respectively. Magnetic field components are stored likewise. Here the normal, first transverse and second transverse component are stored in `ia`, `ib` and `ic` respectively. This index manipulation happens when the left and right variables are stored using the array and not in the HLLD 1D subroutine.

Optimizing HLLD 2D for GPU

Similar to the HLLD 1D, the HLLD 2D was implemented to run on the entire patch in each direction. In each direction, indices were similarly defined, such that the "normal" direction was always stored at index `iu` for velocities and `ia` for magnetic fields. Unlike DISPATCH/RAMSES, this version of the HLLD 2D has U and A as the normal velocity/magnetic field, V and B as the first transverse, and W and C as the second transverse.

The inner loop was changed such that almost all the required computations happened outside the if-statements. This was done to reduce the execution time of the

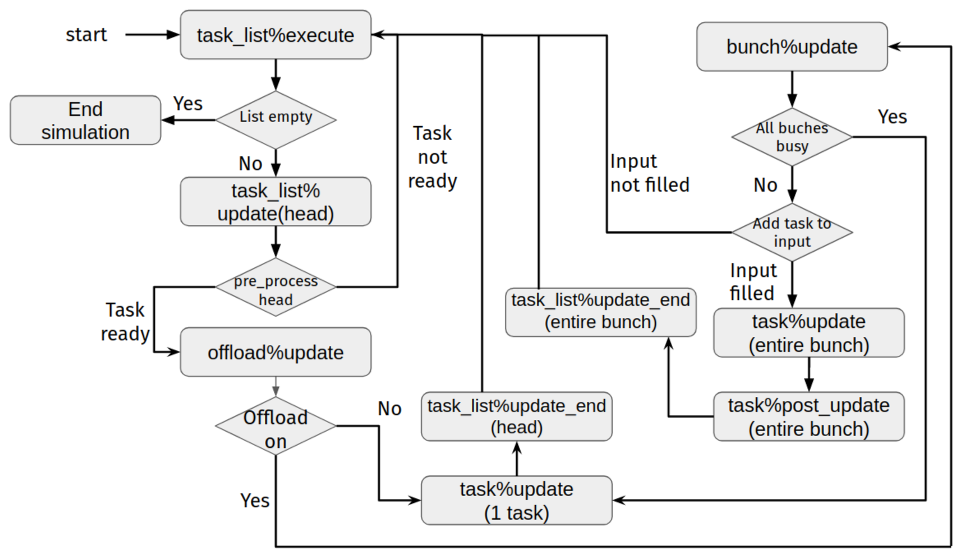


FIGURE 6.6: DISPATCH execution flow with the added bunch module.

divergent part of the code. Although this may lead to some unnecessary computations by some threads, all threads will have roughly the same execution time in each loop, which is preferable in SIMT.

6.4 DISPATCH Integration

6.4.1 offload_mod

Similar to the original mockup code, a skeleton code with the basic module setup was provided by Åke Nordlund. The `dispatcher` data type remains unchanged. The `task_list` has been slightly altered. `task_list%update` is now split into two separate subroutines: `task_list%update` and `task_list%update_end`. The `task_list%update_end` consists of all post-update handling of the `link`. The flowchart shown in chapter 3 is shown in figure 6.6 with the new execution flow.

If the offload module is used, `task_list%update` will call `offload%update` and return. If the offload module is not used, `task%update` is called as usual and `task_list%update_end` is called immediately.

`offload%update` will call `bunch%update` on one of the available bunches. An `offload` type is initiated at the start of the program and is globally available. Upon initialization `offload_params` are read from the input file. These parameters define if offload is on/off, size of bunches, number of bunches, the interval between forced updates, and if updates may be run on the CPU when all bunches are occupied.

The offload data type has a list of bunch types. The length of this list is determined by the "number of devices" input, but cannot exceed the actual number of available devices. The intent is to have one bunch per available device, but currently, multiple devices are not yet implemented. Multiple devices can still be used by having multiple MPI ranks per node.

6.4.2 bunch_mod

This module is in charge of scheduling bunches on a single device. To avoid confusing terminology these bunches are referred to as bunch data, and the bunch scheduler itself is just a bunch. As such, each device has a bunch tied to it. Each bunch may have multiple bunch data, which is stored in a list. The "number of bunches" input parameter determines how many active bunch data a scheduler will control.

When bunch update is called it will check if all bunch data types are already filled and busy. If this is the case and use_cpu is set to true, the task is updated on the CPU. If this is not the case the task will be placed in the `input` bunch data. `input` is a pointer to one of the bunch data in the list. When placed in `input`, `mht_t%pre_update` is called to determine timestep and afterward, the task data will be copied to the static arrays. After the task has been added the bunch scheduler will check if the `input` bunch data is filled or a certain wall clock time interval has passed. Once either of these is true, the tasks in the `input` bunch data will be updated. When this happens, the `input` bunch data pointer will be updated to the next bunch data in the list in a round-robin fashion. This makes it possible to schedule and fill bunch data while one or more bunch data is being updated.

After all GPU computing on a bunch data is done, updated values from the static arrays are copied back to the task and `task_list%update_end` is called for each task to return it to the task list. The bunch module is responsible for the right part of figure 6.6.

The GPU implementation can easily be compiled to run on the CPU as well. The implementation will therefore from now on be referred to as the MHD_Bunch implementation. When the MHD_Bunch implementation is referenced, it is implied that it is run on the GPU. Note that some individual tasks may still be run on the CPU when bunches are compiled for GPU. When running only on the CPU it will be referred to as the MHD_Bunch CPU implementation or MHD_Bunch implementation (CPU).

Chapter 7

Results

7.1 Experimental Setup

Experiments and runtime measurements were run both on a desktop system and the HPC compute cluster used at the Center for Star and Planet Formation[1] referred to as STENO. Unless otherwise stated all runtime measurements are averaged over 3 separate runs.

7.1.1 Desktop System

Part of the experiments and runtime measurements were performed on my desktop. The desktop is running Linux Mint 19.1 Cinnamon, with the following hardware:

- `Intel(C) Core(TM) i7-4790U CPU @ 4.00GHz`(4cores and 8 hardware threads)
- `NVIDIA GeForce GTX 1660 Ti` (1770 MHz, 6 GB GDDR6, 24SMs)
- 2x(`Kingston KHX1600C10D3/8G 8GB DDR3 1600MHz`)

7.1.2 STENO

The experiments run on STENO have been run either on the `astro2_gpu`, `astro2_long` or `astro2_short` partition.

The `astro2_gpu` is equipped with 4 NVIDIA A100 GPU's and 2x16-core AMD EPYC 7302 at 3GHz. The NVIDIA A100 GPUs each have a FP32 peak performance of 19.5TFLOPS, and the CPU has a peak performance of around 3TFLOPS.

`astro2_short` and `astro2_long` runs on the same machines. They are running on nodes with 2x24 core Intel Cascade-Lake CPUs (24-core Xeon 6248R @ 3.0GHz) with a theoretical peak performance of just above 4TFLOPS. As only 40 cores are used in the experiments, the peak performance is a bit below 4TFLOPS.

The peak performance is roughly 5 times higher on a single NVIDIA A100 compared to the CPUs on `astro2_short`. If we include the performance of the CPU on `astro2_gpu` the speedup is roughly 6 times higher.

7.2 Proof of Concept Runtimes

The initial runtime measurements for the HD implementations are shown in figure 7.1. These measurements were performed on the desktop system comparing 1 GPU with 5 CPU threads. Only 5 threads were used as multiple background processes were running, which reduced performance when using all 8 available hardware threads. The first two OpenMP implementations are not shown as they ran out

[1] `https://starformation.hpc.ku.dk/`

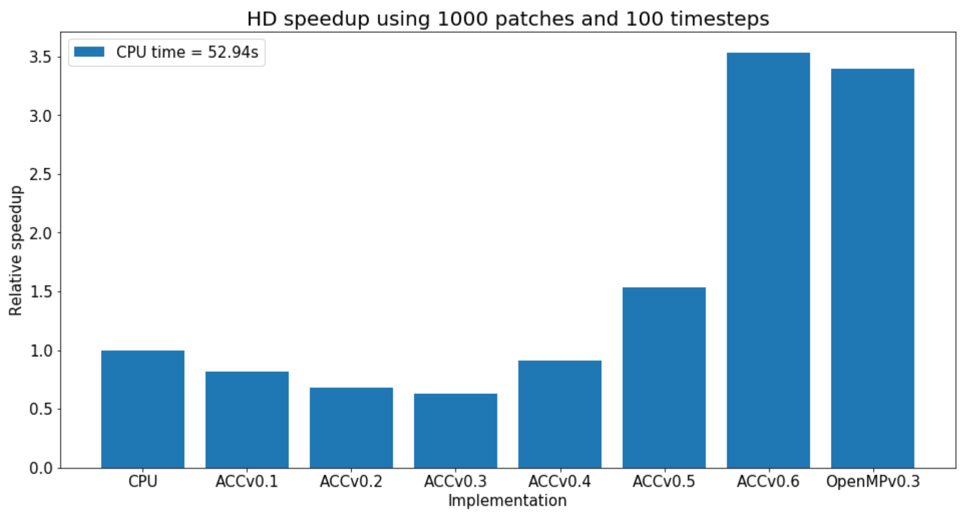


FIGURE 7.1: Relative speedup of GPU implementations compared to CPU.

of memory when using 1000 patches or simply didn't run properly. As the figure shows the simplest implementations gave little to no speedup.

The speedup reached during OpenACC implementation is around 3.5 when having multiple threads calling GPU kernels and hiding overhead with asynchronous execution. As profiling also shows (figure 6.5), there is in this implementation still much idle time on the GPU.

The OpenACC v0.6 speedup is calculated based only on 1 time measurement. The reason for this is that pinned memory did not work properly with some of the newer NVIDIA driver updates on the desktop system. Only 1 runtime measurement was made in the early phases of the project.

For the OpenMP implementations, only the third implementation could run on 1000 patches. As a result only OpenMPv0.3, which uses static array and bunching, is shown in the figure. The runtime is very similar to OpenACC v0.6. The OpenMP implementation used neither asynchronous execution nor pinned memory. The figure shows that the performance from bunching and using static arrays is roughly the same as hiding latency with asynchronous execution and using pinned memory. If bunching and static array's were added to the OpenACC implementation, it would most likely be much faster than the OpenMP version.

There is roughly a factor 2 in speedup between OpenACC v0.5 and OpenACC v0.6. A similar speedup is to be expected in the OpenMP version if pinned memory becomes available.

7.3 Optimal Dimension Layout

Several tests were carried out to determine the optimal way of storing and accessing variables. The tests was done by shuffling the possible ways of storing `nv` (primitive/conservative variables) and `dir` (direction). For example, one permutation could be to store the array as (nv,x,y,dir,z,patches) and another could be (x,nv,dir,y,z,patches). In the setup, patch index is always stored last, and x,y, and z are always stored in that order. A simple nested loop was set up to evaluate performance. In the innermost loop, the array value is stored in a temporary variable, added a constant, and stored back into the array. Since the real solver often accesses values in neighboring cells, tests were also carried out where the value is loaded from the cell with x-1, y-1, or z-1. The basic structure of the loop can be seen in listing 7.1. The inner loop was run with `!$omp simd` for both CPU and GPU. When running on the GPU, the patches were distributed across SMs with `!$omp target teams distribute` and the intermediate loops were collapsed and parallelized. On the CPU only the `!$omp simd` clause on the innermost loop was present. The loops were run a hundred times, to smooth out any fluctuations in runtime.

```
!$omp target teams distribute
do i6=1,patches
    !$omp parallel do schedule(static,1) &
    !$omp shared(arr,i6,dims) private(test,i1) collapse(4)
    do i5=1,dims(5)
        do i4=1,dims(4)
            do i3=1,dims(3)
                do i2=1,dims(2)
                    !$omp simd
                    do i1=1,dims(1)
                        test = arr(i1,i2,i3,i4,i5,i6) + 5
                        arr(i1,i2,i3,i4,i5,i6) = test
                    end do
                end do
            end do
        end do
    end do
end do
```

LISTING 7.1: loop structure to determine optimal storing

In figure 7.2 the results running on the CPU can be seen. Except when `dir` is stored in the first index, there isn't much difference in runtime when loading and storing from the same cell. Interestingly, there seems to be a large and consistent jump in runtime when accessing the x-neighboring cell when `nv` is not stored in the first index. The cause of this was investigated in collaboration with Sven Karlson. By inspection of the assembly code, it was determined that several different versions of the loop were compiled. The program would then at runtime choose what version to run.

When accessing the neighbor in the x-direction and x-dimension were stored in the first index no instruction-level parallelism could be achieved. This happens as each iteration of the inner loops depends on the prior iteration.

In the MHD_Bunch implementation, this is not the case as read and write mostly happens to different arrays. Another test was therefore made where read and write operations were done on two different arrays. The result is seen in figure 7.3. As it can be seen, there is no longer a spike in runtime when accessing the x-neighbor. In this case, there is no noticeable difference in runtime except when storing `dir` in the first index. However, some of the kernels are about 50% slower, despite no extra computations are added.

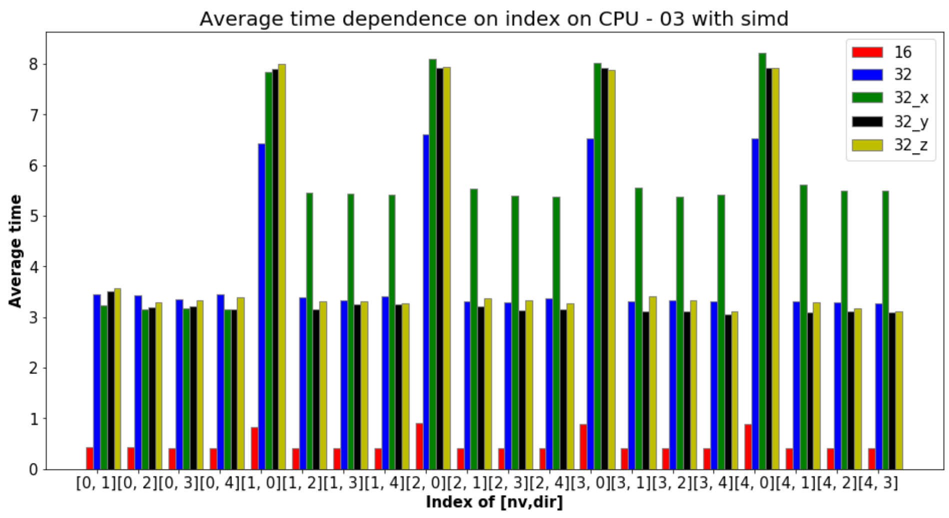


FIGURE 7.2: Optimal index test on CPU

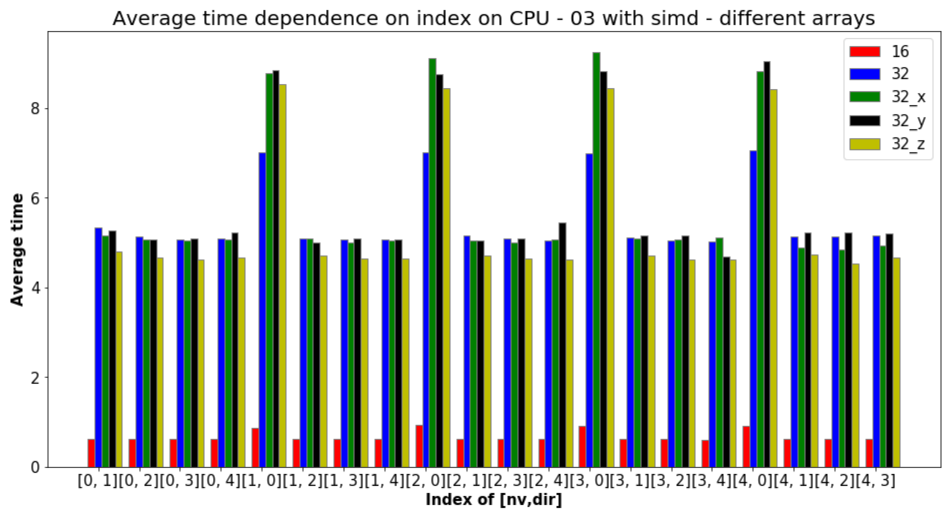


FIGURE 7.3: Optimal index test on CPU using a different array for reading and writing.

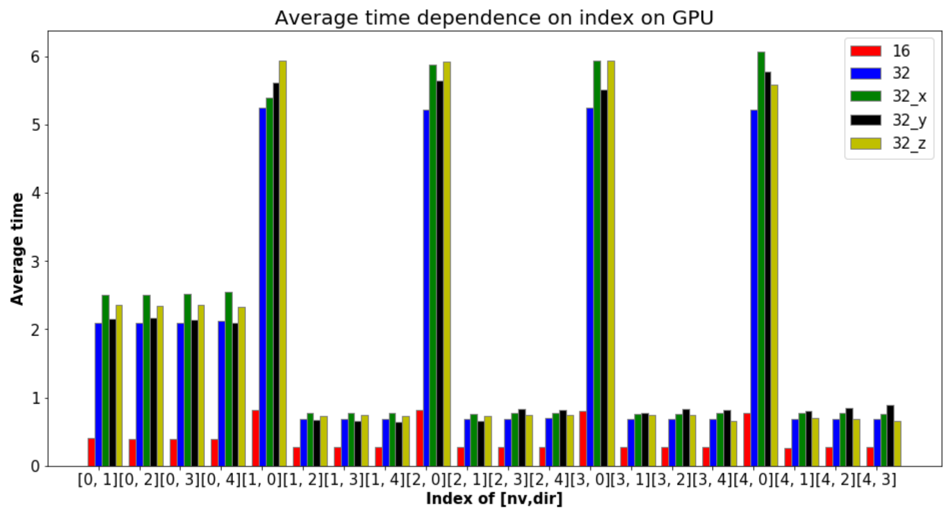


FIGURE 7.4: Optimal index test on GPU

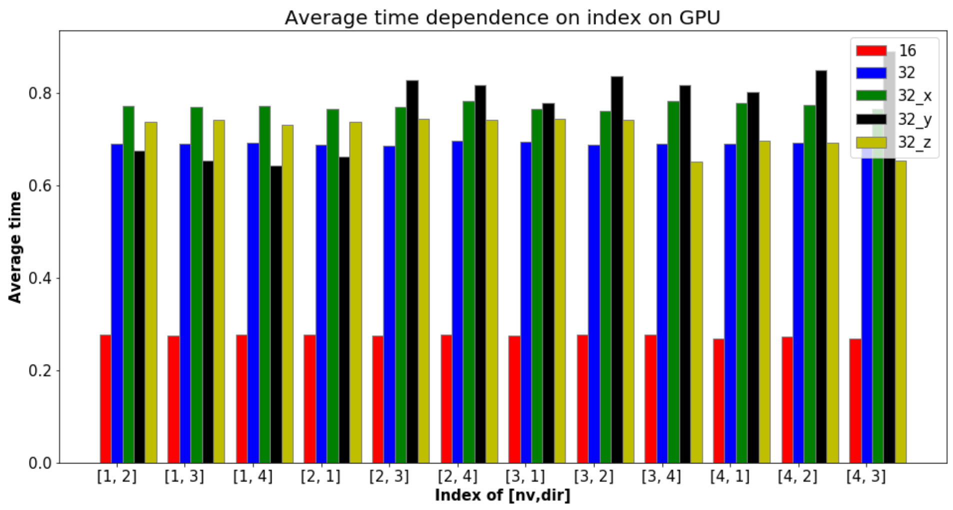


FIGURE 7.5: Optimal index test on GPU, ignoring largest runtimes.

Figure 7.4 shows the experiment using the same array to load and store values on the GPU. A large increase in runtime is seen when `dir` is stored in the first index. However, unlike the CPU, storing `nv` in the first index gives a much worse performance. Figure 7.5 shows the GPU runtimes, but with the largest runtimes ignored. There is no significant difference between storing `nv` in the second, third, fourth of the fifth index. There is no difference when accessing and storing in the same cell. There also seems to be no impact when accessing the x-neighbor cell. There is however a small, but noticeable, difference when accessing the y- and z-neighbor cells.

Since some temporary arrays in DISPATCH/RAMSES are already stored as (x,nv,dir,y,z,patches) it was therefore decided to use this layout.

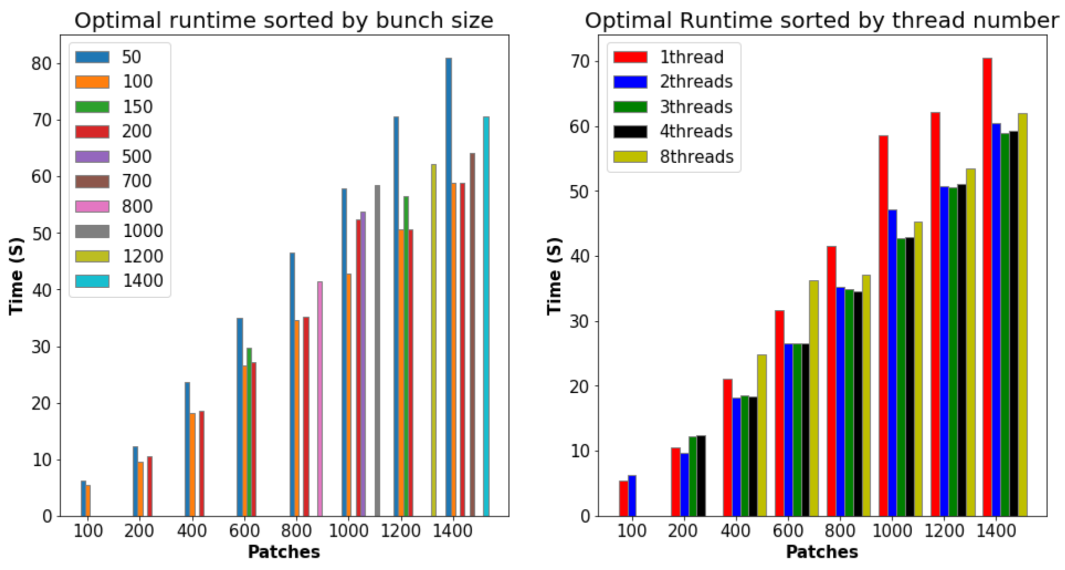


FIGURE 7.6: Optimal runtimes sorted by bunch size and number of threads for different numbers of total patches.

7.4 HLLD Runtime

7.4.1 GPU Dependency of Bunch Size and Threads

The runtime of the HLLD mockup was measured on STENO for different bunch sizes and number of threads. The experiment was run using 26x26x26 patches with 3 guard cells in each direction for a total of 32x32x32 cells per patch. The experiment examines up to 1400 patches. This is roughly the number of 32x32x32 patches that can be stored on the GPU. This corresponds to an effective volume of 290^3. Notice that the total volume is limited by CPU memory since different patches can be shuffled in and out of the GPU. The result is shown in figure 7.6.

In the left figure, the runtime is shown sorted by bunch size and in the right, it is sorted by the number of threads. In the left figure, the optimal runtime for each bunch size is shown. Optimal runtime is to be understood as the smallest runtime for the 5 different number of threads. Similarly in the right figure, for each number of threads, the optimal bunch size was chosen. Due to an issue in the mockup-scheduler, only bunch sizes that were divisible by the total number of patches could be run.

The results consistently show that a bunch size of somewhere between 100-500 and using between 2-4 threads is optimal. The A100 has 108 SMs. For full utilization of the GPU, all SMs needs to be working. Each patch in a bunch is distributed to an SM with `!$omp teams distribute`. It is therefore expected that at least 108 patches per bunch are needed for optimal performance. The experiment indicates that a bunch size between 1-4 times the number of SMs gives optimal performance. More than 1 thread should also be used to hide kernel overhead.

7.4.2 CPU Comparison

Once the optimal bunch/thread combination for the GPU was found, the experiment could be run on the CPU. The experiment is shown in figure 7.7 and 7.8. Figure 7.7 shows the result going up to 1400 patches. The best performing GPU and CPU runtime for each number of patches is shown in figure 7.7a. The relative runtime (compared to GPU) for 4 different numbers of threads is shown in 7.7b. In all cases, using 40 threads performs best, but the relative speedup gained from doubling the number of threads diminishes, and there is very little difference between using 20 and 40 threads.

Figure 7.7a shows a slightly sublinear trend for the GPU and a slightly superlinear trend for the CPU. The speedup increases quickly up to around 800 patches. After this, it continues to increase, but at a slower rate. This indicates that at least around 800-1000 patches are needed for the GPU to be saturated.

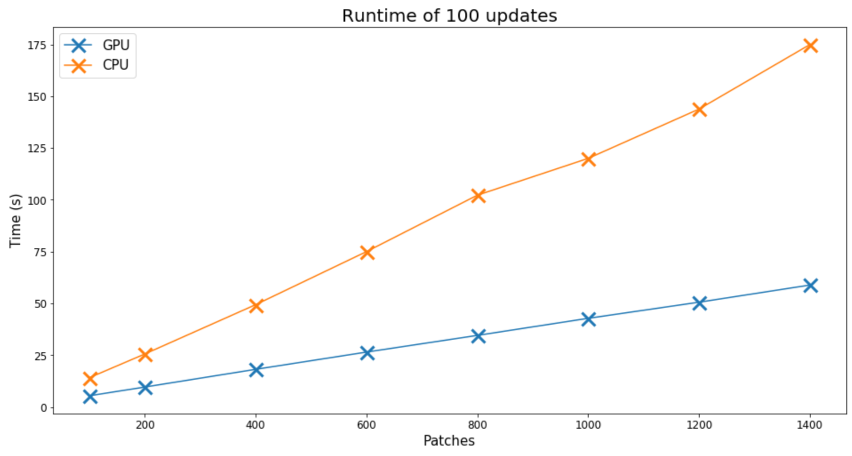
The speedup using 5 or 10 threads increases smoothly, and continues to increase as more and more patches are added. This is not the case for 20 and 40 threads were the speedup is almost constant. The reason for this is most likely that the added overhead of using so many threads still accounts for a significant portion of the runtime.

Figure 7.8 shows the same experiment, but going up to 10.000 patches. In this part of the experiment, only 40 threads were used on the CPU. Like the result for smaller patch sizes, the relative speedup increases rapidly up to around 800. Then there is a small dip in performance, which is quickly restored when using more patches.

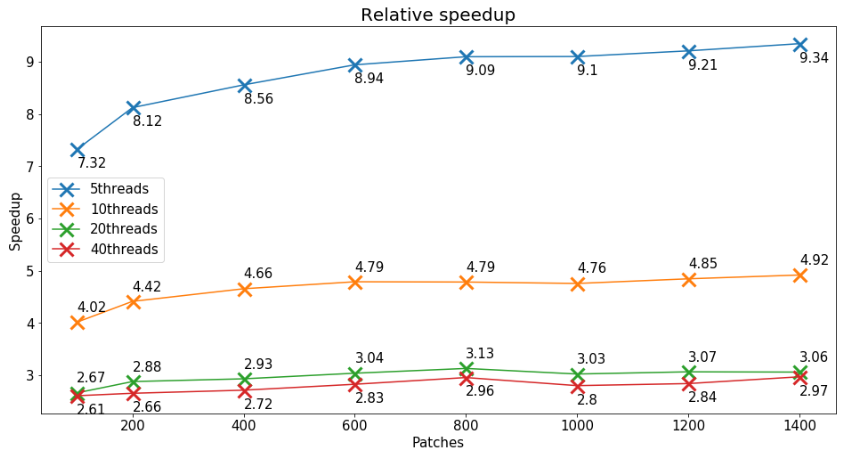
This dip is most like caused by a sub-optimal choice of bunch size in this range. A broader range of bunch sizes was not used because only bunch sizes divisible by the number of patches could be used. This is not the case in the MHD_Bunch implementation.

For 5000 and 10000 patches, the speedup increases slightly, most likely due to the GPU being fully saturated in both cases, but the CPU implementation having superlinear scaling. The speedup is in the range of 2.6-3.0. This is somewhat lower than the theoretical speedup of just under 6.

A profiler was run to determine any obvious causes for the non-optimal speedup. As the profiler could not be run on the server, it was run for a smaller setup on the desktop system. The setup ran 240 patches for 10 updates using 2 threads and a bunch size of 48. The profiler output can be seen in figure 7.9. From the start of the first HtoD transfer to the end of the last DtoH transfer 2731ms passes. Of these 426ms is spent on HtoD transfer, 471ms on DtoH, 1426ms on kernel execution, which leaves around 408ms idle time. If the transfer and idle time could be hidden the performance would increase by roughly a factor of 2, which would bring the speedup compared to using 40 CPU threads to around 5-6 bringing it close to the theoretical maximal speedup.

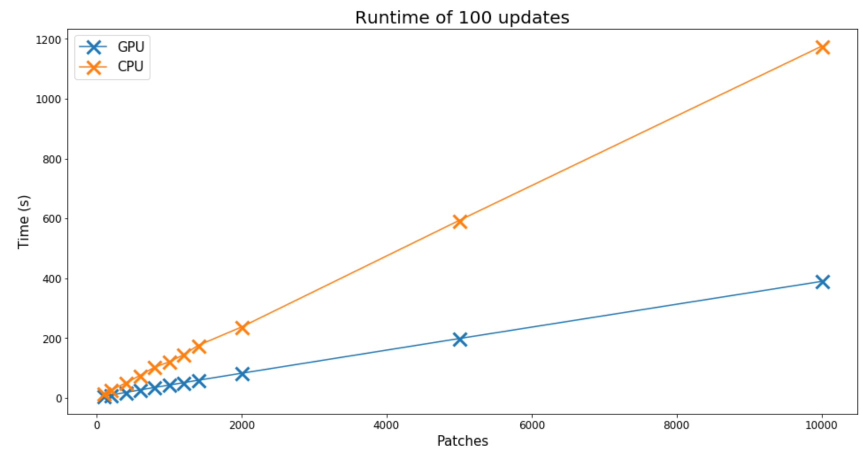


(A) Runtime

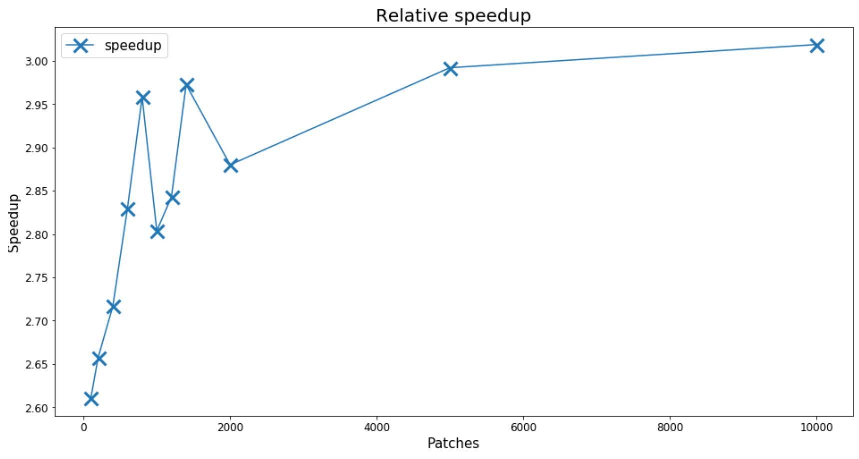


(B) Relative Speedup

FIGURE 7.7: Runtime and relative speedup using different numbers of CPU threads. For each patch size and thread count, only the time of the best performing bunch size is shown. For the GPU only the time of the best bunch/thread combination is stored



(A) Runtime



(B) Relative Speedup

FIGURE 7.8: Runtime and relative speedup for a large number of patches.

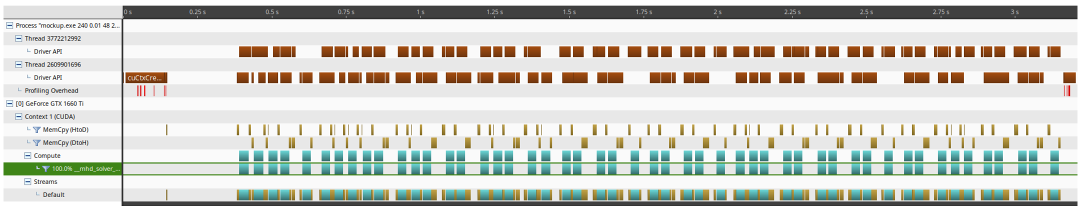


FIGURE 7.9: Profiler of MHD implementation for 240 patches, running 10 updates with a bunch-size of 48 and using 2 threads.

7.5 Integration Validation

Validation was first performed by writing out all arrays (prim, grad, flux, etc.) after 1 update on a single patch with a fixed timestep. This was done for both the MHD_Bunch implementation and for DISPATCH/RAMSES. This method was useful in the early stages of integration when serious errors were still present in the GPU implementation. However, as the MHD_Bunch implementation approached correctness this was not a robust validation for two reasons.

First, because a different approach was taken numerical differences were bound to occur even with a physically correct solution. For FP32 one might expect around 6-9 significant digit precision. Errors such as sign errors may in some cases (where the value is small) not produce much difference in the result after one update. This difference was often in the 6-9 significant digit range. As such it was not possible to discern between correct or incorrect solutions based on this approach.

Secondly, as stated in section 6.3.1 some minor errors were discovered in DISPATCH/RAMSES. This meant that strict enforcement of the same values would not prove the correctness of the MHD_Bunch implementation.

For these reasons more complex experiments was set up for validation. Two experiments were chosen. First, the 1D MHD shock experiment from Ryu and Jones, 1995 was chosen. The initial conditions is a standard Riemann problem with left and right starting states. To the left of 0.5 the initial conditions are $(\rho_L = 1, v_{xL} = 0, v_{yL} = 0, v_{zL} = 0, E_l = 1, B_{yL} = 1, B_{zR} = 0)$ and to the right $(\rho_R = 0.21, v_{xR} = 0, v_{yR} = 0, v_{zR} = 0, E_R = 1, B_{yR} = 1, B_{zR} = 0)$ with $B_x = 1$ on both left and right states. The experiment is run in the range $x = [0, 1]$ using 512 grid point. The MHD_Bunch CPU and DISPATCH/RAMSES uses a single patch with 512 cells. The MHD_Bunch GPU experiment uses 8 patches each with 64 cells as the bunching scheduler crashes with only 1 patch. The results can be seen in figure 7.10. As the figure shows, the results are identical for all three solvers and matches the results in figure 4a in Ryu and Jones, 1995. The second experiment chosen was the Orszag-Tang vortex (Orszag and Tang, 1979). This experiment was chosen as it features a multitude of waves that interacts and cross one another. This results in a complex system that covers a wide range of conditions. As such any errors in a solver will quickly become noticeable in the experiment. The experiment has the initial conditions:

$$V = -sin(y)\hat{x} + sin(x)\hat{y} \tag{7.1}$$

$$B = -B_0 sin(2\pi y)\hat{x} + B_0 sin(4\pi x)\hat{y} \tag{7.2}$$

The experiment was run with both DISPATCH/RAMSES and the MHD_Buch implementation (CPU and GPU) for a setup with 32x32x1 cells per patch and 16x16x1 patches (512x512 cells) and with 32x32x1 cells per patch and 32x32x1 patches (1024x1024 cells). The "mistake" in DISPATCH/RAMSES in the EMF should not matter in this case, if the size of the z-direction is chosen to be $1/(32 * 16)$ in the first case and $1/(32 * 32)$ in the second case.

The results can be seen in figure 7.11 and 7.12. The solutions are identical up until 0.5s. At this point, the inner part starts forming small vortices that are seen at t=0.6 in both figures. Three vortexes are seen in the 512x512 experiment. Two small on either side and one in the middle. Five is seen in the 1024x1024 experiment. Two very small on either side, two somewhat larger closer to the center and one in the center. It may be hard to discern from the figures, but these vortices are slightly displaced in the two different runs, which causes a large effect at time t=1. It should be noted that multiple runs were performed for both DISPATCH/RAMSES and MHD_Bunch

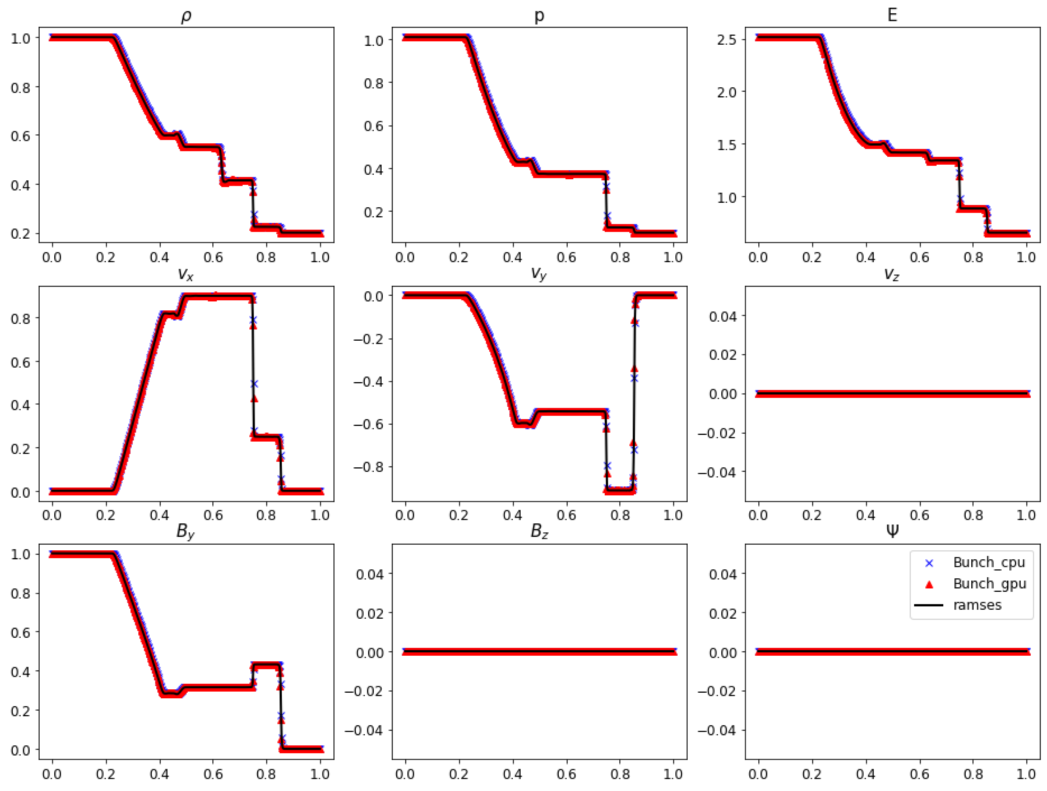


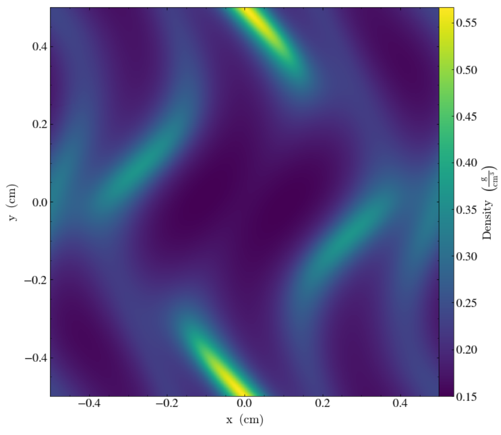
FIGURE 7.10: 1D shock tube test with initial condition $(\rho_L = 1, v_{xL} = 0, v_{yL} = 0, v_{zL} = 0, E_l = 1, B_{yL} = 1, B_{zR} = 0)$ and $(\rho_R = 0.21, v_{xR} = 0, v_{yR} = 0, v_{zR} = 0, E_R = 1, B_{yR} = 1, B_{zR} = 0)$ for the left and right states.

implementation. The results of some of these varied. Some examples are shown in figure D.1 in the appendix.

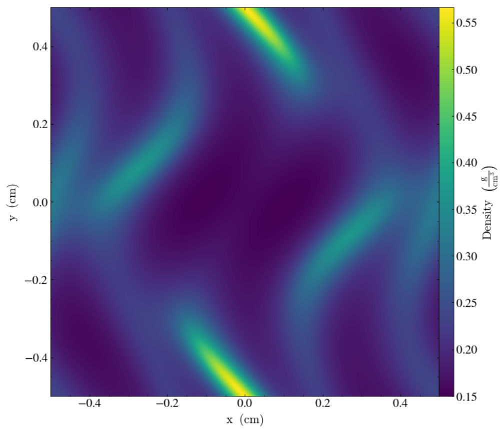
The reason for this variation is not in the solvers themselves but the exchange and interpolation of ghost zones. The Orszag-Tang experiment is a completely symmetric setup, and any small deviation from symmetry might set off a domino effect that disrupts the system later in the simulation. Patches are picked up and updated in a semi-random order. This means that neighboring cells may be updated in a different order when running the simulation multiple times. Because of this, the patches will not have the same time and ghost zone interpolation will therefore differ. This is enough to cause the small disturbance at around t=0.6 that leads to the larger and more visible differences at time t=1. This only happens because of the strict symmetry. As such, in realistic scenarios, the result will be the same. Validation of MHD_Bunch implementation was performed both when compiling for GPU and CPU bunch execution.

A larger Orszag-Tang experiment was run on STENO to measure the runtime of the different versions. This experiment used 26x26x26 cells per patch (32x32x32 with ghost zones) and 32x32x3= 3072 patches. This experiment was only run for about 100 iterations. The MHD_Bunch GPU implementation ran in 180.4s, MHD_Bunch CPU implementation ran in 305.8s and DISPATCH/RAMSES ran in 527.4s. The MHD_Bunch GPU implementation was run for 4 different bunch sizes: 54, 108, 216, and 432 where 108 gave the lowest runtime.

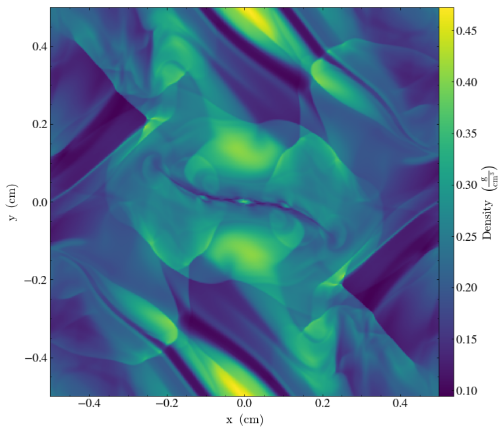
The speedup comparing MHD_Bunch GPU is thus around 1.7 compared to MHD_Bunch CPU. This is a somewhat smaller speedup than the mockup because bunching is still not optimal. The speedup comparing MHD_Bunch GPU to DISPATCH/RAMSES is 2.9, and the speedup comparing MHD_Bunch CPU to DISPATCH/RAMSES is 1.7.



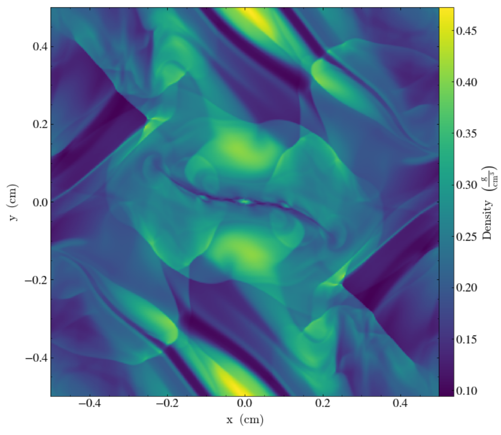
(A) DISPATCH/RAMSES solution at time 0.1



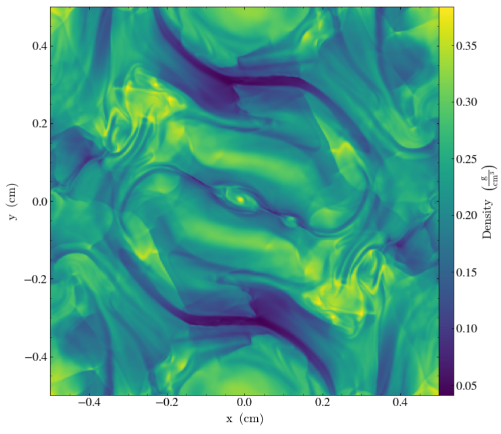
(B) MHD_Bunch solution at time 0.1



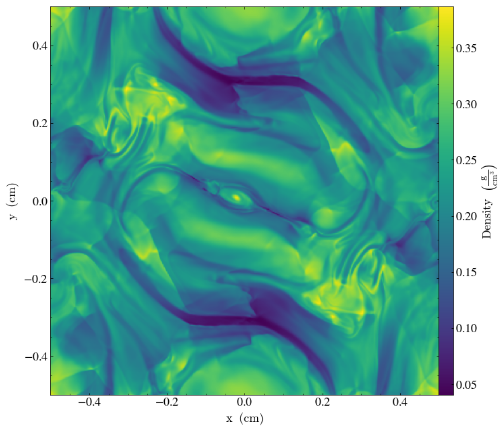
(C) DISPATCH/RAMSES solution at time 0.6



(D) MHD_Bunch solution at time 0.6

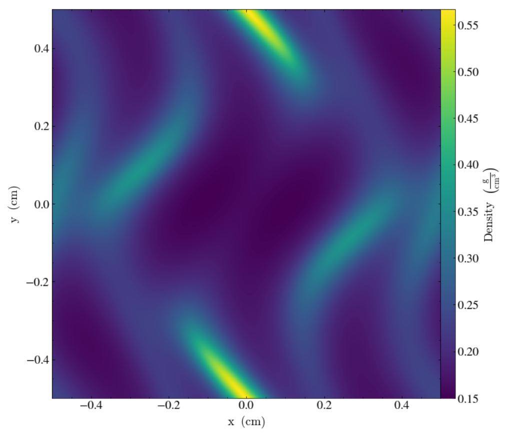


(E) DISPATCH/RAMSES solution at time 1.0

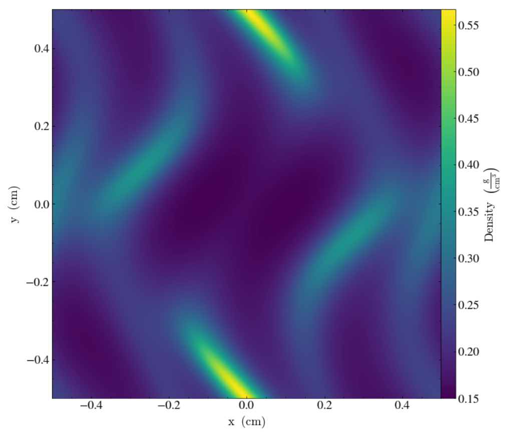


(F) MHD_Bunch solution at time 1.0

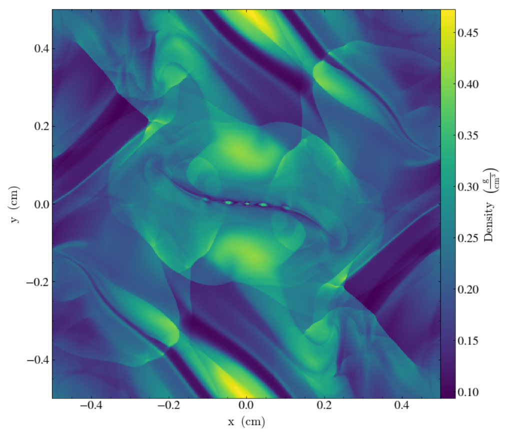
FIGURE 7.11: Side by side snapshots of DISPATCH/RAMSES solution and MHD_Bunch implementation running the Orszag-Tang experiment on a 512x512 grid. There is a slight mismatch in the later time steps



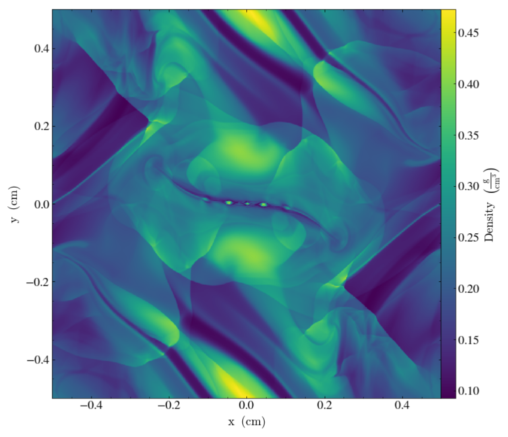
(A) DISPATCH/RAMSES solution at time 0.1



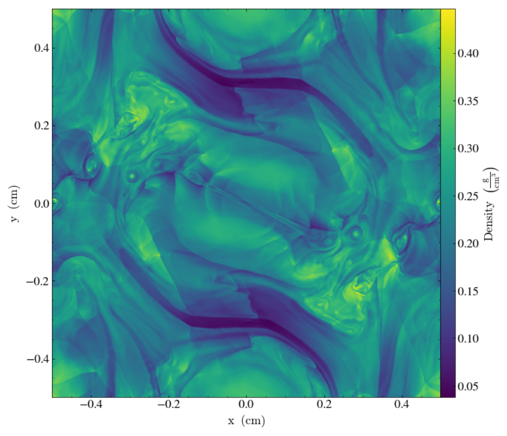
(B) MHD_Bunch solution at time 0.1



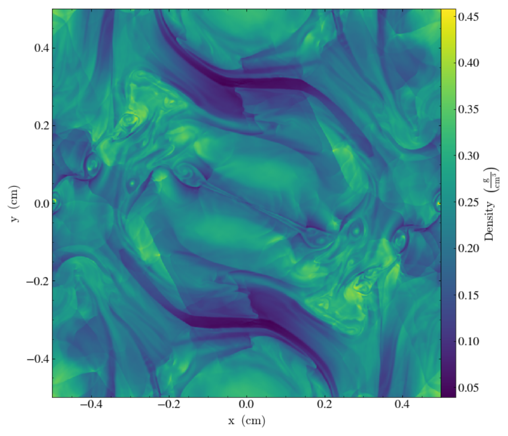
(C) DISPATCH/RAMSES solution at time 0.6



(D) MHD_Bunch solution at time 0.6



(E) DISPATCH/RAMSES solution at time 1.0



(F) MHD_Bunch solution at time 1.0

FIGURE 7.12: Side by side snapshots of DISPATCH/RAMSES solution and MHD_Bunch implementation running the Orszag-Tang experiment on a 1024x1024 grid. There is a slight mismatch in the later time steps

Chapter 8

Discussion

8.1 Current State

The MHD_Bunch implementation has been integrated into DISPATCH and a bunching module added for scheduling of bunches. The implementation has been validated with a 1D MHD shock tube and the Orszag-Tang vortex experiments. The implementation only has support for the use of 1 device per MPI rank, but the code is prepared for possible future implementation of multiple devices. The way bunching is scheduled (size, interval, etc.) is statically set as user input.

The mockup shows a speedup of around 3.0 comparing one A100 GPU to 40 CPU threads. If idle time and memory transfer can be hidden, this speedup is expected to roughly double to around 6. As 4 GPUs are available an additional factor 4 could be added for a speedup of 24. The long-term goal is to target the LUMI system, which features a slightly slower socket. LUMI will also have the AMD Instinct^{TM} GPUs[1]. The current AMD Instinct GPU, MI100, is just under 20% faster than the A100, and LUMI will likely use a successor to the MI100. This would bring the speedup to around 30. Although this estimate is based on the current speedup of the mockup, it is expected that DISPATCH could see the same speedup on large experiments.

8.2 Remaining Issues

There currently persists some issues in the MHD_Bunch implementation. They are briefly listed below:

- The bunching module will crash in very small experiments, such as the 1D shock tube.
- Stalling sometimes causes a lot of small bunches to be scheduled, which can harm performance.
- When finishing a simulation the bunching module will sometimes stall for 30s-1min especially in smaller experiments.
- The static array allocation still assumes that each dimension (x,y,z) has the same number of cells, which causes it to sometimes allocate unnecessarily large arrays.
- The current implementation has required a change of the basic task data type. This may cause an issue when integrated into production as many other solvers rely on this type.

[1]https://www.lumi-supercomputer.eu/lumi-one-of-the-worlds-mightiest-supercomputers/

8.3 Extensions to the Bunching Module

8.3.1 Dynamic Bunching

Optimizing the bunch-scheduling remains the largest potential for improving performance. The scheduler does not properly ensure that a bunch is always filled and ready to be sent, executed, and returned to the CPU and as a result, more idle time occurs than in the mockup.

This could be improved by implementing a dynamic bunch scheduler. For example, the `interval` between forced updates could be adjusted throughout a simulation. This would also supersede the `interval scaling` parameter.

Currently, there is an option for updating a single task on the CPU if all bunches are busy. This could likewise be extended to updating a small bunch on the CPU. For example, if the bunch size is set to 200, but only 2 tasks have been added to the bunch. After a while, it might be preferable to just run these on the CPU.

The choice of whether or not to run a partly filled bunch on the GPU might be driven by the number of tasks that will become available after the update. DISPATCH routinely interacts with the neighboring patches to exchange ghost zones and checks if neighbors have become ready after an update. This could be altered to instead check before updating how many neighbors rely on the patch. If a large number of patches are waiting for a single patch or a small bunch to be updated it might be better to update on the CPU immediately.

8.3.2 Other Solvers

The current bunching module only supports the MHD_Bunch solver. An obvious next step in the development of DISPATCH would be to port other solvers to GPU. Higher-order MHD-solvers would be good candidates. Alternatively, ray-tracing radiative transfer or self-gravity could for example be ported. Just like MHD, it will most likely be the case that the best approach for the GPU version of these solvers will be to bunch tasks together.

Depending on the implementation only a little (or perhaps nothing) would have to be changed in the bunching module to support these solvers.

8.4 Reducing Memory Footprint

The DISPATCH/RAMSES solver has been refactored but the basic structure of the solver remains the same. Several temporary arrays are used to store intermediate values of the entire patch. In the MHD_Bunch implementation, these intermediate arrays are statically stored both on the GPU and on the CPU, which requires a lot of memory.

Each cell requires in principle only knowledge about its neighbors. Each cell can thus be updated by only storing a 3x3x3 region of temporary variables. On the CPU, it might provide better results to not store the temporary arrays for the entire patch. Instead, only a 3x3x3 region could be stored. Each cell could then be looped over in sequence. The extra memory requirement of each patch only would be reduced to O(1).

This would probably not be the best approach on the GPU as there would not be enough work for each thread and/or not enough threads. An alternative approach could be to only look at a slice of the patch. At any given time, the temporary variables needed to be stored would only be NxNx3. This would reduce the memory requirement from O(N^3) to O(N^2), with N being the patch size in each dimension.

8.5 Other Compilers

Only the GCC and PGI have been used in this thesis. As earlier discussed, these both have their nondesirable quirks. The PGI compiler works great for OpenACC on NVIDIA GPUs, but it is not well optimized for OpenMP or running on other GPUs. GCC on the other hand does not suffer a vendor bias. However, GCC is far behind in many of the offload features in OpenMP for Fortran.

Specifically, the support of asynchronous execution is completely missing in GCC. Other compilers (Cray, LLVM, and IBM) do have advertised support for asynchronous execution. The Cray compiler was briefly investigated during this thesis. In extension to my work, René Løwe Jacobsen and Tethys Svensson from DeIC investigated the use of other compilers to run kernels asynchronously in OpenMP. However, the effort proved unsuccessful for the GCC, PGI, and Cray compilers.

8.6 Different approaches

8.6.1 OpenACC/OpenMP Hybrid

With the announcement that LUMI was going to feature AMD GPUs, OpenMP took priority over the OpenACC. As a result, the OpenACC HD implementation was never fully integrated with a bunch solution. Furthermore, no OpenACC implementation was done for MHD. As shown during HD implementation asynchronous execution is fully working and easy to implement once a working solution is implemented in OpenACC. For these reasons, it could be an interesting future project to implement the MHD with an OpenACC approach.

8.6.2 CUDA/C

As stated multiple times during the thesis, one of the largest obstacles has been the limited compiler support for offloading with OpenMP in Fortran, specifically the memory transfer. Similar to the GenASIS (Budiardja and Cardall, 2019) and GENE code(Germashewski et al., 2021) interoperability with CUDA/C or CUDA/C++ could be used as the compiler support for C and C++ is much better. The AMD non-vendor-specific API HIP could be used as an alternative to CUDA.

8.7 Comparison With GenASIS and GENE Codes

The GenASIS and GENE codes are in some ways very similar to DISPATCH. They are both Fortran-based and make use of the object-oriented features introduced in Fortran 2003. Comparing the results achieved when porting these two code bases to GPU is therefore a good proxy for the quality of the MHD_Bunch implementation.

Both codes were tested on Summit at the Oak Ridge Leadership Computing Facility[2]. Summit has two IBM Power9 CPUs with 22-cores per node leading to a total of 44 cores per node. 1 core on each socket is reserved for the operating system, so

[2]https://www.olcf.ornl.gov/summit/

a total of 42 cores is available. Each socket is connected to 3 NVIDIA Volta V100 GPUs. A total of 6 GPUs is thus available on each node. This also gives 7 CPU cores for each GPU.

The V100 has FP32 peak performance of around 15.7 TFLOPS making the total peak performance around 94.2 TFLOPS per node. This is around 1.2 times higher than the peak performance per node on STENO, which features 4 A100 GPU each with 19.5 TFLOPS peak performance for a total of 78 TFLOPS. The Power9 CPU has a peak performance of around 2.161 TFlops per node or 344 GFLOPS per 7 cores[3]. The relative theoretical peak performance between 7 cores and 1 GPU is therefore around 45.

The GenASIS code achieved a speedup of around 6 when comparing 1 GPU with 7 CPU threads, and double that with pinned memory. The GENE code likewise saw a speedup of roughly 15 when comparing 6 GPUs to 42 cores (1 GPU per 7 threads). Clearly, this is way below the theoretical peak performance in both cases. As noted by Germashewski et al., 2021 the speedup of 15 corresponds better to the ratio between the GPU and CPU memory bandwidth of around 16 within a single socket. However, this speedup of 15 only holds for single-node and drops to around 6 when using multiple nodes.

The mockup was similarly run using only 7 threads for a better comparison. The result is seen in figure 8.1. As the figure shows, the speedup is about a bit higher than what was found for GenASIS without pinned memory. GenASIS reduced runtime by 50% when using pinned memory. As earlier discussed the speedup is expected to roughly double if memory transfer and idle time can be hidden. This would bring the speedup to 13-15.

In addition, the CPUs on STENO is about twice as fast as on Summit, so the relative theoretical peak performance between 7 cores and 1 GPU is therefore around 30 on STENO. The peak memory transfer on STENO comparing 1 A100 to 1 CPU node is around 5.5. This is again very close to the expected speedup comparing 1 GPU with 40 cores. This was expected to be around 6 if memory transfer and idle time could be hidden

Considering all this, the results achieved in this project are on par with the results reached in the GenASIS and GENE code.

As shown in Nordlund et al., 2018, DISPATCH has near-optimal scaling. As the newly added bunch module does not interfere with the inter-node communication the scalability of DISPATCH remains the same. The speedup achieved in this project will therefore not suffer the same decrease in performance as the GENE code when running on multiple nodes.

It should be noted however that DISPATCH, GenASIS, and GENE are in many ways very different codes and the experiments were run of different systems. The comparison is meant only to give a superficial look at the current state and potential improvements in the MHD_Bunch implementation.

[3] https://fuse.wikichip.org/news/1351/ornls-200-petaflops-summit-supercomputer-has-arrived-to-become-worlds-fastest/

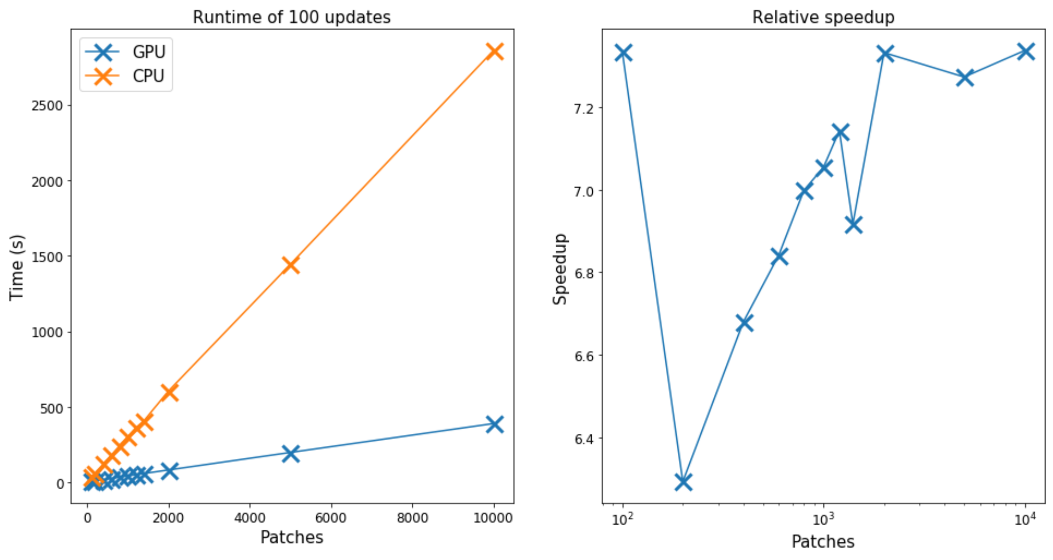


FIGURE 8.1: Runtime and speedup comparing 1 GPU to 7 CPU thread.

Chapter 9

Conclusion

It has been shown that porting DISPATCH to GPUs is possible using only directives and without any C/CUDA interoperability. Both OpenACC and OpenMP versions have been implemented and tested successfully on a hydrodynamic mockup using the HLL solver. Memory transfer is the main bottleneck in the HLL solver. Statically allocating work arrays and *bunching* tasks together significantly increase performance when running on the GPU, as it reduces the overhead and reduces latency, by oversubscribing the GPU work. Asynchronous execution and memory transfer have also been shown to be possible with OpenACC, which further improves GPU performance. Asynchronous execution of kernels with OpenMP is currently not possible with the GCC compiler. Asynchronous execution is in principle supported by the LLVM and Cray compilers, but it has not been possible to get a working version.

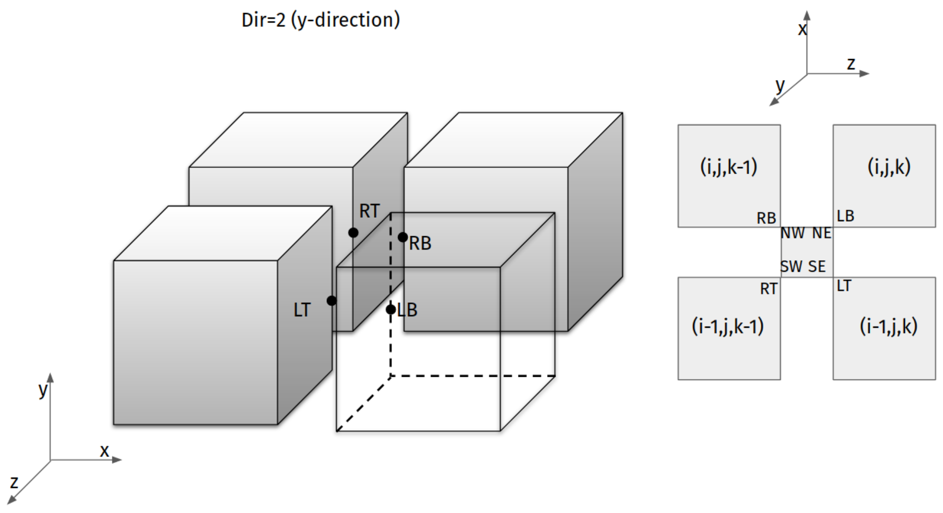
An HLLD solver has similarly been tested in a mockup and then implemented in DISPATCH. Memory transfer time is similar to the kernel execution time. The current speedup using 1 GPU per 40 cores is around 3 for the mockup. If memory transfer and idle time can be hidden, this is expected to be increased to be around 30 for 4 GPU per 42 cores on the LUMI system.

Porting the HLLD MHD solver required a complete refactoring of the code. A new solver, MHD_Bunch, has been implemented together with a bunching module. The code has been tested and validated with a 1D MHD shock tube and the Orszag-Tang Vortex experiments. Two issues in the DISPATCH/RAMSES code regarding the symmetry of the code were discovered and fixed in the MHD_Bunch implementation. The kernels have not been fine-tuned and there is still some room for improvement. The bunching module does not perfectly schedule the bunches, which causes idle time on the GPU. The speedup comparing MHD_Bunch GPU to MHD_Bunch CPU is around 1.7. The speedup comparing MHD_Bunch GPU to DISPATCH/RAMSES is 2.9, and the speedup comparing MHD_Bunch CPU to DISPATCH/RAMSES is 1.7.

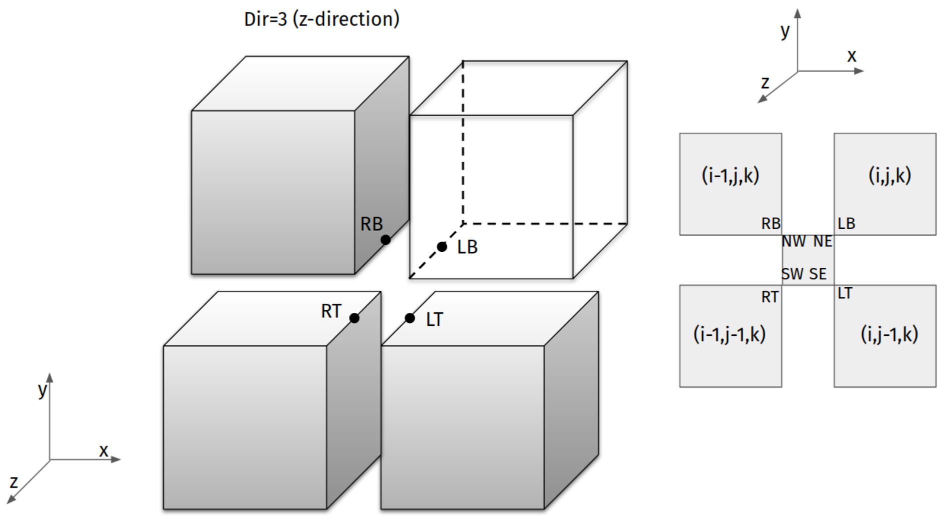
Overall, the project has been a success and the MHD_Bunch implementation outperforms the existing code both when running on the GPU and the CPU. More significantly, the thesis has demonstrated the viability of porting complex physics modules, using exclusively directives with satisfactory performance. I hope that the MHD_Bunch solver will be used in production and that the results and experiences of this thesis can be used as a template for porting more physics modules to GPUs. This would make it possible to execute non-trivial science applications at scale on LUMI in the near future.

Appendix A

Corner values, y- and z-direction



(A) Cell corner values to edge-centered corner values in the y-direction. SW,SE,NW,NE values are used to solve 2D Riemann problem and calculate EMF.



(B) Cell corner values to edge-centered corner values in the z-direction. SW,SE,NW,NE values are used to solve 2D Riemann problem and calculate EMF.

FIGURE A.1: Corner values for y- and z-direcitions

Appendix B

Compiler installation

B.1 PGI

Installation of PGI before HPC SDK integration:

- Make sure to have `gfortran` installed. At least v. 7.4.0 or newer
- Download the install file.
- Unpack tar file "tar xpfz <tarfile>.tar.gz"
- navigate to install folder and type "sudo ./install"
- accept and choose single network install (use default installation folder)
- use links - install cuda and Mpi components
- Install license key
- Update the `.bashrc` with the following (may need to change depending on the version)

```
export PGI=/opt/pgi;
export PATH=/opt/pgi/linux86-64/19.10/bin:$PATH;
export MANPATH=$MANPATH:/opt/pgi/linux86-64/19.10/man;
export LM_LICENSE_FILE=$LM_LICENSE_FILE:/opt/pgi/license.dat;
```

- Install openjdk-8-jre

Installaion of PGI with HPC SDK:

- Download install package from https://developer.nvidia.com/nvidia-hpc-sdk-downloads
- Follow install instruction
- After install add the following to `.bashrc`

```
NVARCH=‘uname -s‘_‘uname -m‘; export NVARCH
NVCOMPILERS=/opt/nvidia/hpc_sdk; export NVCOMPILERS
MANPATH=$MANPATH:$NVCOMPILERS/$NVARCH/20.9/compilers/man; export MANPATH
PATH=$NVCOMPILERS/$NVARCH/20.9/compilers/bin:$PATH; export PATH
export PATH=$NVCOMPILERS/$NVARCH/20.9/comm_libs/mpi/bin:$PATH
export MANPATH=$MANPATH:$NVCOMPILERS/$NVARCH/20.9/comm_libs/mpi/man
export LD_LIBRARY_PATH="/opt/nvidia/hpc_sdk/Linux_x86_64/20.7/cuda/11.0/
        lib64:$LD_LIBRARY_PATH"
export PATH="/opt/nvidia/hpc_sdk/Linux_x86_64/20.7/cuda/11.0:$PATH"
export PATH="/opt/nvidia/hpc_sdk/Linux_x86_64/20.7/cuda/11.0/bin:$PATH"
```

B.2 GCC

- Version 10.2 has issues with CUDA 11 so install cuda 10.2
- To allow profiler:
  - install java with `sudo apt-get install openjdk-8-jre`
  - set as default with: `sudo update-alternatives -config java` - select java-8
  - Create a file (e.g profile.conf) in the folder `/etc/modprobe.d`
  - Open file and insert the following:

    ```
    options nvidia "NVreg_RestrictProfilingToAdminUsers=0"
    ```

  - Close and restart
- once cuda is installed and profiler added add the following to the `.bashrc`

  ```
  export PATH=/usr/local/cuda-10.2:$PATH"
  export PATH=/usr/local/cuda-10.2/bin:$PATH"
  export LD_LIBRARY_PATH=/usr/local/cuda-10.2/lib64:$LD_LIBRARY_PATH
  ```

- Run GCC-offload script (see below). Change cuda-variable to denote your local cuda folder

```
#!/bin/bash
# Build GCC with support for offloading to NVIDIA GPUs.
set -o nounset -o errexit
# Location of the installed CUDA toolkit
cuda=/usr/local/cuda-10.2
# directory of this script
MYDIR="$( cd -P "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
work_dir=$MYDIR/gcc-10-2-offload
install_dir=$work_dir/install
rm -rf $work_dir
# Build assembler and linking tools
mkdir -p $work_dir
cd $work_dir
git clone https://github.com/MentorEmbedded/nvptx-tools
cd nvptx-tools
./configure \
   --with-cuda-driver-include=$cuda/include \
   --with-cuda-driver-lib=$cuda/lib64 \
   --prefix=$install_dir
make
make install
cd ..
# Set up the GCC source tree
git clone https://github.com/MentorEmbedded/nvptx-newlib
wget -c https://github.com/gcc-mirror/gcc/archive/releases/gcc-10.2.0.tar.gz
tar xf gcc-10.2.0.tar.gz
ln -s gcc-releases-gcc-10.2.0 gcc-src
```

```
cd gcc-src
contrib/download_prerequisites
ln -s ../nvptx-newlib/newlib newlib
target=$(./config.guess)
cd ..
# Build nvptx GCC
mkdir build-nvptx-gcc
cd build-nvptx-gcc
../gcc-src/configure \
   --target=nvptx-none \
   --with-build-time-tools=$install_dir/nvptx-none/bin \
   --enable-as-accelerator-for=$target \
   --disable-sjlj-exceptions \
   --enable-newlib-io-long-long \
   --enable-languages="c,c++,fortran,lto" \
   --prefix=$install_dir
make -j4
make install
cd ..
# Build host GCC
mkdir build-host-gcc
cd  build-host-gcc
../gcc-src/configure \
   --enable-offload-targets=nvptx-none \
   --with-cuda-driver-include=$cuda/include \
   --with-cuda-driver-lib=$cuda/lib64 \
   --disable-bootstrap \
   --disable-multilib \
   --enable-languages="c,c++,fortran,lto" \
   --prefix=$install_dir
make -j4
make install
cd ..
```

Appendix C

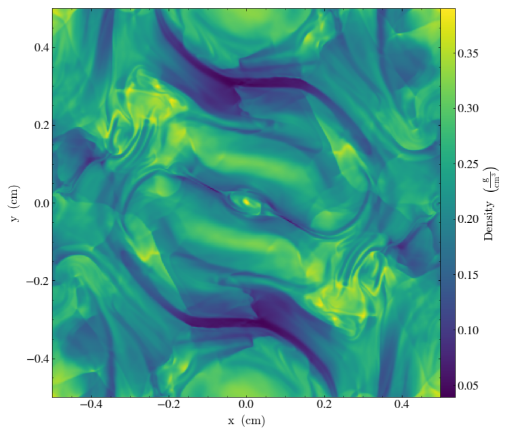
OpenACC v0.1 update

```fortran
subroutine update(self)
    class(solver_t),target :: self
    integer:: i3, i2, i1
    real, dimension(:,:,:,:,:,:), pointer:: mem
    real, dimension(:,:,:,:), pointer :: prim
    real, dimension(:,:,:,:), pointer :: left
    real, dimension(:,:,:,:), pointer :: rght
    real, dimension(:,:,:,:,:), pointer :: grad
    real, dimension(:,:,:,:,:), pointer :: flux
    real, dimension(:,:), pointer :: ff
    integer, dimension(:), pointer :: gn
    integer, dimension(3) :: lb, ub, l, u
    integer :: new, it, nv
    real(8) :: gamma, ds(3), dtime, g1
    real :: u2_max, g2, dtd1, dtd2, dtd3, dt2
    integer:: iv, uo(3), lo(3), li(3), ui(3)
    ff => self%ff
    grad => self%grad
    flux => self%flux
    mem => self%mem
    gn => self%gn
    prim => self%prim
    left => self%left
    rght => self%rght
    nv = self%nv
    lb = self%lb
    ub = self%ub
    new = self%new
    it = self%it
    gamma = self%gamma
    dtime = self%dtime
    ds = self%ds
    g1 = self%g1
    g2 = self%g2
    uo = self%uo
    lo = self%lo
    ui = self%ui
    li = self%li
    l = lb+1
    u = ub-1
    if (gamma == 1d0) then
      g2 = 1.0
    else
      g2 = (gamma-1d0)*gamma
    end if
    u2_max = 0.0
    dt2 = dtime*0.5
    dtd1 = dt2/ds(1)
    dtd2 = dt2/ds(2)
    dtd3 = dt2/ds(3)
    !$acc data copyin(new,it,lb,ub, g2, u2_max, &  !ctoprim variables
    !$acc             l, u, &                      !slopes variables
    !$acc             dtd1, dtd2, dtd3, dt2, g1, lo, uo, ff, &
    !$acc             li, ui, nv, &
    !$acc             prim, grad, left, rght, flux, mem)
    call ctoprim (self, mem, new, it, prim, lb, ub, g2, u2_max)
    call slopes  (self, prim, grad, nv,  l, u)
    call predict (self, prim, grad, mem, new, g1, lo, uo, dtd1, dtd2,
   dtd3, dt2, ff)
    call riemn3d(self, lo, uo, left, rght, prim, grad, flux, nv, mem)
    call divflux(self, mem, flux, li, ui, new, nv, dtd1, dtd2, dtd3)
    call sources(self, mem, prim, gamma, new)
    !$acc end data
    self%time = self%time + self%dtime
    self%it = mod(self%it,5) + 1
    self%new = mod(self%new,5) +1
end subroutine update
```

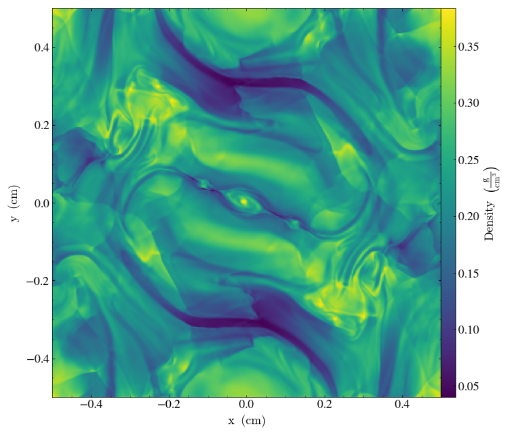
LISTING C.1: Update function in first OpenACC version.

Appendix D

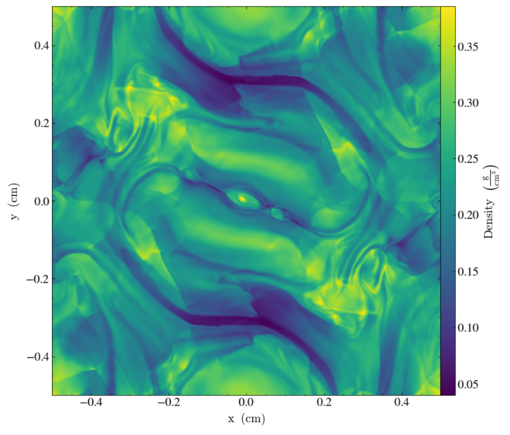
Orszag-Tang experiments



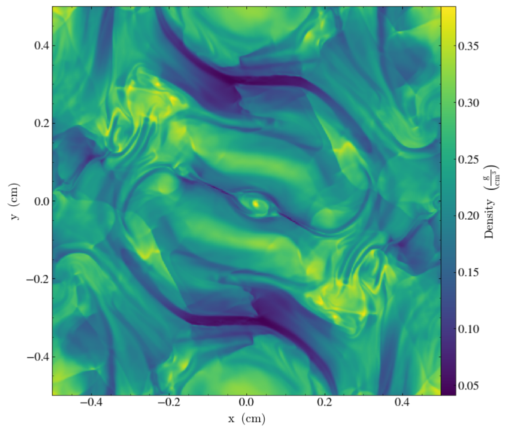
(A) MHD_Bunch at t=1 for grid-size 512x512



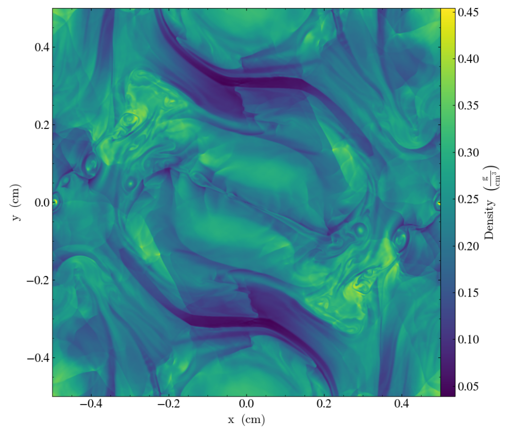
(B) MHD_Bunch at t=1 for grid-size 512x512



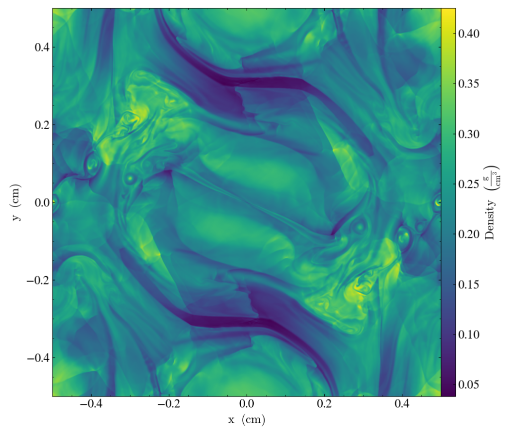
(C) MHD_Bunch at t=1 for grid-size 512x512



(D) DISPATCH/RAMSES solution at t=1 for grid-size 512x512



(E) MHD_Bunch at t=1 for grid-size 1024x1024



(F) DISPATCH/RAMSES solution at t=1 for grid-size 1024x1024

FIGURE D.1: Different outcomes for the Orszag-Tang experiments.

Bibliography

Bai, Feng peng, Zhong hua Yang, and Wu gang Zhou (2018). “Study of total variation diminishing (TVD) slope limiters in dam-break flow simulation”. In: *Water Science and Engineering* 11.1, pp. 68–74. ISSN: 1674-2370. DOI: https://doi.org/10.1016/j.wse.2017.09.004. URL: https://www.sciencedirect.com/science/article/pii/S1674237018300255.

Batten, P. et al. (Nov. 1997). “On the Choice of Wavespeeds for the HLLC Riemann Solver”. In: *SIAM J. Sci. Comput.* 18.6, 1553–1570. ISSN: 1064-8275. DOI: 10.1137/S1064827593260140. URL: https://doi.org/10.1137/S1064827593260140.

Benitez-Llambay, Pablo and Frédéric S. Masset (2016). “FARGO3D: A NEW GPU-ORIENTED MHD CODE”. In: *The Astrophysical Journal Supplement Series* 223.1, p. 11. ISSN: 1538-4365. DOI: 10.3847/0067-0049/223/1/11. URL: http://dx.doi.org/10.3847/0067-0049/223/1/11.

Berger, Marsha J. and Joseph Oliger (Mar. 1984). “Adaptive mesh refinement for hyperbolic partial differential equations”. English (US). In: *Journal of Computational Physics* 53.3, pp. 484–512. ISSN: 0021-9991. DOI: 10.1016/0021-9991(84)90073-1.

Brackbill, J.U and D.C Barnes (1980). “The Effect of Nonzero · B on the numerical solution of the magnetohydrodynamic equations”. In: *Journal of Computational Physics* 35.3, pp. 426–430. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(80)90079-0. URL: https://www.sciencedirect.com/science/article/pii/0021999180900790.

Budiardja, Reuben D. and Christian Y. Cardall (2019). “Targeting GPUs with OpenMP directives on Summit: A simple and effective Fortran experience”. In: *Parallel Computing* 88, p. 102544. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2019.102544. URL: http://www.sciencedirect.com/science/article/pii/S0167819119301358.

Cardall, Christian Y. et al. (2014). “GENASIS: GENERAL ASTROPHYSICAL SIMULATION SYSTEM. I. REFINABLE MESH AND NONRELATIVISTIC HYDRODYNAMICS”. In: *The Astrophysical Journal Supplement Series* 210.2, p. 17. ISSN: 1538-4365. DOI: 10.1088/0067-0049/210/2/17. URL: http://dx.doi.org/10.1088/0067-0049/210/2/17.

Davis, S. F. (1988). “Simplified Second-Order Godunov-Type Methods”. In: *SIAM Journal on Scientific and Statistical Computing* 9.3, pp. 445–473. DOI: 10.1137/0909030.

Dr. Momme Allalen (June 17, 2020). *Fundamentals of Accelerated Computing with CUDA C/C++*. PRACE. URL: https://doku.lrz.de/display/PUBLIC/PRACE+Course\%3A+Deep+Learning+and+GPU+Programming+Workshop (visited on 12/22/2020).

Dubey, Anshu et al. (2014). “A survey of high level frameworks in block-structured adaptive mesh refinement packages”. In: *Journal of Parallel and Distributed Computing* 74.12. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3217 –3227. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.07.001. URL: http://www.sciencedirect.com/science/article/pii/S0743731514001178.

Einfeldt, B et al. (1991). “On Godunov-type methods near low densities”. In: *Journal of Computational Physics* 92.2, pp. 273 –295. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(91)90211-3. URL: http://www.sciencedirect.com/science/article/pii/0021999191902113.

Fromang, Sebastien, P. Hennebelle, and Romain Teyssier (Dec. 2012). “A high order Godunov scheme with constrained transport and adaptive mesh refinement for astrophysical magnetohydrodynamics”. In: *Astronomy and Astrophysics*. DOI: 10.1051/0004-6361:20065371.

Germashewski, K. et al. (2021). In:

Gudiksen, Boris et al. (July 2011). “The stellar atmosphere simulation code Bifrost”. In: *Astronomy Astrophysics - ASTRON ASTROPHYS* 531. DOI: 10.1051/0004-6361/201116520.

Harten, Amiram, Peter Lax, and Bram van Leer (Jan. 1983). “On Upstream Differencing and Godunov-Type Schemes for Hyperbolic Conservation Laws”. In: *SIAM Rev* 25, pp. 35–61.

Leer, Bram van (July 1979). “Towards the Ultimate Conservative Difference Scheme V. A Second-order Sequel to Godunov’s Method”. In: *Journal of Computational Physics* 32, pp. 101–136. DOI: 10.1016/0021-9991(79)90145-1.

Li, Xuechao and Po-Chou Shih (Jan. 2018). “An Early Performance Comparison of CUDA and OpenACC”. In: *MATEC Web of Conferences* 208, p. 05002. DOI: 10.1051/matecconf/201820805002.

Lustig, Daniel and Margaret Martonosi (2013). “Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization”. In: *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. HPCA ’13. USA: IEEE Computer Society, 354–365. ISBN: 9781467355858. DOI: 10.1109/HPCA.2013.6522332. URL: https://doi.org/10.1109/HPCA.2013.6522332.

Mattson, T.G., Y. He, and A.E. Koniges (2019). *The OpenMP Common Core: Making OpenMP Simple Again*. Scientific and Engineering Computation. MIT Press. ISBN: 9780262538862. URL: https://books.google.dk/books?id=e6beyAEACAAJ.

McClanahan, Chris (2011). “History and Evolution of GPU Architecture A Paper Survey”. In:

Mei, X. and X. Chu (2017). “Dissecting GPU Memory Hierarchy Through Microbenchmarking”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.1, pp. 72–86. DOI: 10.1109/TPDS.2016.2549523.

Miyoshi, Takahiro and Kanya Kusano (2005). “A multi-state HLL approximate Riemann solver for ideal magnetohydrodynamics”. In: *Journal of Computational Physics* 208.1, pp. 315 –344. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2005.02.017. URL: http://www.sciencedirect.com/science/article/pii/S0021999105001142.

Negrutn, Dan (2013). *Primer: Elements of Processor Architecture. The Hardware/Software Interplay*. URL: https://www.sciencedirect.com/science/article/pii/S1674237018300255.

Nobile, Marco et al. (Mar. 2014). “cuTauLeaping: A GPU-Powered Tau-Leaping Stochastic Simulator for Massive Parallel Analyses of Biological Systems”. In: *PloS one* 9, e91963. DOI: 10.1371/journal.pone.0091963.

Nordlund, Aake et al. (May 2018). “DISPATCH: A numerical simulation framework for the exa-scale era - I. Fundamentals”. In: *Monthly Notices of the Royal Astronomical Society* 477. DOI: 10.1093/mnras/sty599.

Nordlund, Åke, Klaus Galsgaard, and R. F. Stein (1994). “Magnetoconvection and Magnetoturbulence”. In: *Solar Surface Magnetism*. Ed. by Robert J. Rutten and

Carolus J. Schrijver. Dordrecht: Springer Netherlands, pp. 471–498. ISBN: 978-94-011-1188-1. DOI: 10.1007/978-94-011-1188-1_37. URL: https://doi.org/10.1007/978-94-011-1188-1_37.

NVIDIA. *GPU: Changes Everything*. https://web.archive.org/web/20160408122443/http://www.nvidia.com/object/gpu.html. Accessed: 2020-12-22.

NVIDIA Corporation (2020). *NVIDIA CUDA C Programming Guide*. Version 11.0.

Orszag, S. A. and C. M. Tang (Jan. 1979). "Small-scale structure of two-dimensional magnetohydrodynamic turbulence". In: *Journal of Fluid Mechanics* 90, pp. 129–143. DOI: 10.1017/S002211207900210X.

Roe, P.L (1981). "Approximate Riemann solvers, parameter vectors, and difference schemes". In: *Journal of Computational Physics* 43.2, pp. 357 –372. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(81)90128-5. URL: http://www.sciencedirect.com/science/article/pii/0021999181901285.

Ryu, Dongsu and Tom Jones (Mar. 1995). "Numerical Magnetohydrodynamics in Astrophysics: Algorithm and Tests for One-Dimensional Flow". In: *Astrophysical Journal* 442.

Ryu, Dongsu et al. (July 1998). "A Divergence-Free Upwind Code for Multidimensional Magnetohydrodynamic Flows". In: *Astrophysical Journal* 509. DOI: 10.1086/306481.

Stone, James M. and Michael L. Norman (June 1992). "ZEUS-2D: A Radiation Magnetohydrodynamics Code for Astrophysical Flows in Two Space Dimensions. I. The Hydrodynamic Algorithms and Tests". In: 80, p. 753. DOI: 10.1086/191680.

Teyssier, R. (Apr. 2002). "Cosmological hydrodynamics with adaptive mesh refinement. A new high resolution code called RAMSES". In: 385, pp. 337–364. DOI: 10.1051/0004-6361:20011817. arXiv: astro-ph/0111367 [astro-ph].

Toro, Eleuterio F. (2009). "Riemann Solvers and Numerical Methods for Fluid Dynamics. A Practical Introduction". In: Springer-Verlag. DOI: 10.1007/978-3-540-49834-6.

Yee, H. C. (1989). "A class of high resolution explicit and implicit shock-capturing methods". In: