An Implementation of the Discontinuous Galerkin Method on Graphics Processing Units

by

Martin Fuhry

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Applied Mathematics

Waterloo, Ontario, Canada, 2013

Abstract

Computing highly-accurate approximate solutions to partial differential equations (PDEs) requires both a robust numerical method and a powerful machine. We present a parallel implementation of the discontinuous Galerkin (DG) method on graphics processing units (GPUs). In addition to being flexible and highly accurate, DG methods accommodate parallel architectures well, as their discontinuous nature produces entirely element-local approximations.

While GPUs were originally intended to compute and display computer graphics, they have recently become a popular general purpose computing device. These cheap and extremely powerful devices have a massively parallel structure. With the recent addition of double precision floating point number support, GPUs have matured as serious platforms for parallel scientific computing.

In this thesis, we present an implementation of the DG method applied to systems of hyperbolic conservation laws in two dimensions on a GPU using NVIDIA's Compute Unified Device Architecture (CUDA). Numerous computed examples from linear advection to the Euler equations demonstrate the modularity and usefulness of our implementation. Benchmarking our method against a single core, serial implementation of the DG method reveals a speedup of a factor of over fifty times using a USD $500.00 NVIDIA GTX 580.

Acknowledgements

First and foremost, thank you to my advisor, Lilia Krivodonova, for her expertise and encouragement. Her patience and dedication have made this thesis a reality. I would like to thank my committee members, Justin Wan and Hans De Sterck. Special thanks also to Noel Chalmers for his integral assistance in this work, especially during late nights squashing bugs in my code. I would like express my appreciation to David Fuhry for his assistance with code design and implementation decisions. Finally, I would like to acknowledge the support provided by my family during the prepartion of this thesis.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Numerical simulations can produce profoundly accurate predictions of what happens in the real world. In his essay "The Unreasonable Effectiveness of Mathematics", [16] Richard Hamming recalls computing his first real numerical approximations.

> My first real experience in the use of mathematics to predict things in the real world was in connection with the design of atomic bombs in the Second World War. How was it that numbers we so patiently computed on the primitive relay computes agreed so well with what happened on the first test shot at Almagordo? There were, and could be, no small-scale experiments to check the computations directly. Later experience with guided missiles showed me that this was not an isolated phenomenon – constantly what we predict from the manipulation of mathematical symbols is realized in the real world.

Mathematical models were born as we captured the fundamental laws of nature and confined them to paper. Many of these models describe physical systems responding to changes in themselves over space and/or time. These mathematical systems, called partial differential equations (PDEs), are used to model a considerable amount of physical phenomena from the heat distribution of a room to air flow around an airplane wing. They are modeled by functions, which typically depend on a coordinate variable x, or in multiple dimensions $\mathbf{x}$, and/or a time component t.

This thesis is exclusively concerned with solving a specific subset of PDEs called hyperbolic systems of conservation laws [13]. These types of PDEs model wave-like physical behavior of a system whenever the flow of its variables is conserved. The prototypical one-dimensional hyperbolic conservation law is the linear advection equation

$$\partial_t u(x,t) + a\partial_x u(x,t) = 0, \tag{1.1}$$

where the exact solution consists of the initial profile of the wave $u(x,0)$ traveling with velocity a along a coordinate direction x over some time t.

We aim to numerically approximate solutions to systems of hyperbolic conservation laws. Computing high-order accurate numerical approximations is incredibly expensive, often taking days or even weeks of computing time on a single machine or else requiring parallelization and supercomputers. Furthermore, complex hyperbolic conservation laws produce difficult to model physical phenomena such as discontinuities and turbulent fluid flow. Numerical inaccuracies in modeling these physical phenomena may lead to instabilities in the numerical method, rendering the approximation useless.

Creating efficient, highly accurate, and robust numerical methods for approximating PDEs is therefore very difficult work. Choosing the correct numerical method to use is problem dependent. Nonlinear hyperbolic conservation laws, in particular, are notoriously difficult to approximate.

Discontinuous Galerkin (DG) methods combine features of finite element methods and finite volume methods [30, 21, 9, 8, 6, 20]. With strong mathematical foundations, DG methods have a plethora of attractive properties. They are robust and high-order accurate, able to model the difficult to capture physical phenomena common to hyperbolic conservation laws. They use arbitrarily high-order approximations without increasing the stencil size. This feature is entirely due to the discontinuous element-local nature of the method, a key for parallelization techniques. They are able to run on unstructured meshes which can be adaptively refined during computation to assist in capturing moving physical phenomenon such as shock waves. Furthermore, each element can use a different order approximation, allowing the solution over problematic areas to be cheaply computed by low-order approximations while approximations over non-troublesome areas can be more accurately captured using very high-order approximations.

DG methods are especially open to parallel implementations for the following reasons. First, due to their discontinuous nature, they are element local, requiring only information from their immediate neighboring elements to advance the solution to the next time level, even for arbitrarily high-order approximations. Each element, therefore, may be thought of as an independent approximation. Furthermore, they can be paired with an explicit time stepping method, able to step forward in time using only previous information. As parallelization is one of DG's most enticing features, a vast amount of parallelization techniques and implementations have preceded this one; see [5, 4, 3]. DG methods presented in most parallel implementations take a nodal approach [19] while this implementation uses a modal approach [32].

Our DG method is implemented on graphics processing units (GPUs). This factor distinguishes our work from other parallel implementations, as scientific computing on GPUs has become popular only very recently. Indeed, the first dedicated GPUs emerged as recently as the 1980s. General purpose GPU computing was not well adopted until 2006, when the advent of dedicated, easy to use programming models for GPUs such as NVIDIA's Compute Unified Device Architecture (CUDA) emerged. With support for double-precision computing added to CUDA in 2008, GPUs have since become a serious tool for scientific computing. With cheap, consumer NVIDIA GPUs reaching teraflop double-precision performance, GPU computing is now, more than ever, a formidable force.

Even though scientific computing on GPUs is still in its infancy, several GPU implementations of DG methods have been tried with great success. Klöcker, et al., [23] achieved nearly four teraflops in single precision using an NVIDIA Tesla S1070 workstation for a 3D Maxwell problem. Further, they show in [22] a factor of forty to sixty times speed improvement over serial implementations of a 3D problem on an NVIDIA GTX 280. Hesthaven, et al., [18] achieved a twenty-five times speedup in performance versus a serial CPU implementation in 2009. On an NVIDIA Tesla S1070 workstation, Goedel, et al., [15] achieved a speedup of over seventy times a serial implementation.

This work presents a parallel implementation of DG methods to approximate two-dimensional systems of hyperbolic conservation laws on GPUs. In Chapter 2, we derive and present the DG method in one and two dimensions. In Chapter 3, we discuss parallel computing, presenting an introduction to CUDA. In Chapter 4, we show how DG methods may be parallelized, introducing our parallel implementation. Chapter 5 presents a series of computed examples and finishes with benchmarks of our implementation over a CPU implementation. We conclude in Chapter 6, listing this implementation's limitations and describing possible and planned future work.

Chapter 2

The Discontinuous Galerkin Method

2.1 One-Dimensional Hyperbolic Conservation Laws

Consider the one-dimensional hyperbolic conservation law

$$\partial_t u + \partial_x f(u) = 0, \tag{2.1}$$

for a sufficiently smooth flux function $f(u)$ over a domain $\Omega \subseteq \mathbb{R}$ with the initial condition

$$u(x, 0) = u_0(x),$$

and appropriate boundary conditions.

As solutions to (2.1) may be discontinuous, the spatial derivative may be ill-defined. To allow for discontinuous solutions, we restate the conservation law in the weak formulation. To this end, we multiply each side of (2.1) by a smooth test function with compact support, $v \in C_0^\infty(\Omega)$, and integrate over the entire domain to obtain

$$\int_\Omega \partial_t uv \, dx + \int_\Omega \partial_x f(u) v \, dx = 0. \tag{2.2}$$

Integrating the second term of equation (2.2) by parts and using the fact that v has compact support gives the weak formulation

$$\int_\Omega \partial_t uv \, dx - \int_\Omega f(u) v' \, dx = 0. \tag{2.3}$$

We say that u in (2.3) is a weak solution to the conservation law (2.1) if equation (2.3) is satisfied for all $v \in C_0^\infty(\Omega)$.

To build our computational model, we partition our domain into a finite mesh of N elements

$$\Omega = \bigcup_{i=1}^{N} \Omega_i, \tag{2.4}$$

where each $\Omega_i = [x_i, x_{i+1}]$ is a distinct, non-overlapping element of the mesh for $i = 1, \ldots, N$. The mesh is the collection of elements Ω_i and may be represented by the collection of endpoints x_i. We call $h_i = x_{i+1} - x_i$ the length of element Ω_i.

We enforce the weak formulation (2.3) over each element Ω_i independently. That is, we multiply equation (2.1) by a smooth test function v and integrate over Ω_i by parts to obtain

$$\int_{\Omega_i} \partial_t u v \, dx - \int_{\Omega_i} f(u) v' \, dx + f(u) v|_{x_i}^{x_{i+1}} = 0. \tag{2.5}$$

We approximate u over Ω_i in equation (2.5) with a function U_i in a finite-dimensional subspace of $C_0^\infty(\Omega_i)$. We call this subspace the finite element space and denote it by $V(\Omega_i)$. Suppose this finite element space has dimension $p+1$ with a basis

$$\Phi = \{\phi_j\}_{j=0}^{p}. \tag{2.6}$$

Then, we may represent $U_i \in V(\Omega_i)$ as a linear combination of basis functions

$$U_i = \sum_{j=0}^{p} c_{i,j} \phi_j, \tag{2.7}$$

for coefficients $c_{i,j}(t)$. The global approximation U is then represented by a direct sum of the local approximations U_i from equation (2.7)

$$U = \bigoplus_{i=1}^{N} U_i. \tag{2.8}$$

Note that each $V(\Omega_i)$ may have a different dimension, allowing the flexibility of approximating the solution over specific elements with higher- or lower-order accuracy than other local approximations We assume for simplicity that each element uses a function space of the same dimension for the approximating solution U_i, as is common in DG implementations.

To uniquely determine the coefficients $c_{i,j}$ for U_i from (2.7), we must enforce the weak formulation of the conservation law on Ω_i for $p+1$ linearly independent test functions v. The standard Galerkin formulation tells us to choose our test functions v from the same finite element space as our approximation U_i. That is, we want equation (2.5) to hold for every $v \in V(\Omega_i)$. Since Φ is a basis for $V(\Omega_i)$, we can choose $v = \phi_k, k = 0, \ldots, p$ and solve for U_i in

$$\int_{\Omega_i} \partial_t U_i \phi_k \, dx - \int_{\Omega_i} f(U_i)\phi_k' \, dx + f(U_i)\phi_k|_{x_i}^{x_{i+1}} = 0. \tag{2.9}$$

Writing U_i as in (2.7), we have

$$\int_{\Omega_i} \partial_t \sum_{j=0}^{p} c_{i,j}\phi_j\phi_k \, dx - \int_{\Omega_i} f\left(\sum_{j=0}^{p} c_{i,j}\phi_j\right)\phi_k' \, dx + f\left(\sum_{j=0}^{p} c_{i,j}\phi_j\right)\phi_k \,|_{x_i}^{x_{i+1}} = 0. \tag{2.10}$$

Rather than considering a separate basis for each element, we consider only a single basis over a canonical element by mapping Ω_i to a canonical $\Omega_0 = [-1, 1]$. Equation (2.10) is then computed over the canonical element, where each ϕ_i is mapped to a basis Φ of the canonical function space $V(\Omega_0)$. As the canonical function space $V(\Omega_0)$ is the same for each element, the finite element space for each Ω_i may use the same basis functions. We represent U_i mapped over Ω_0 as a linear combination of these new basis functions

$$U_i = \sum_{j=0}^{p} c_{i,j}\phi_j. \tag{2.11}$$

where $\phi_j \in \Phi$ is now a basis function of the canonical function space $V(\Omega_0)$. Computation is then done over the canonical element using the coefficients $c_{i,j}$ and basis elements ϕ_j in (2.11).

2.2 The DG Method With Legendre Polynomials

A commonly chosen finite element space is $V(\Omega_0) = \mathbf{P}_p(\Omega_0)$, the space of polynomials over Ω_0 with degree at most p. A simple choice of a basis for this space is

$$\Phi = \{1, x, x^2, x^3, ..., x^p\}. \tag{2.12}$$

Then, to recover the coefficients $c_{i,j}$ for the approximation U_i from the initial conditions, we can solve the system of equations

$$\int_{\Omega_i} u_0 \phi_k \, dx = \sum_{j=0}^{p} c_{i,j} \int_{\Omega_i} \phi_j \phi_k \, dx, \tag{2.13}$$

for each $\phi_k \in \Phi$. However, this particular problem is known to be very poorly-conditioned. Fortunately, using an orthogonal basis transforms this problem into a well-conditioned one.

Orthogonal bases are often used in computation because they tend to produce well-conditioned, simplified problems. We define the inner product between $v, w \in V(\Omega_0)$ as

$$(v, w) = \int_{\Omega_0} vw \, d\xi. \tag{2.14}$$

Using the Gram-Schmidt orthogonalization algorithm [35], any finite-dimensional basis can be made orthogonal. Applying the Gram-Schmidt orthogonalization algorithm, e.g., to the monomial basis $\{1, x, x^2, \ldots, x^p\}$ of $\mathbf{P}_p(\Omega_0)$ for the canonical element $\Omega_0 = [-1, 1]$ produces the Legendre polynomials. The first five Legendre polynomials ,

$$P_0 = 1 \tag{2.15}$$
$$P_1 = x \tag{2.16}$$
$$P_2 = \frac{1}{2}(3x^2 - 1) \tag{2.17}$$
$$P_3 = \frac{1}{2}(5x^3 - 3x) \tag{2.18}$$
$$P_4 = \frac{1}{8}(35x^4 - 30x^2 + 3), \tag{2.19}$$

are shown in Figure 2.1.

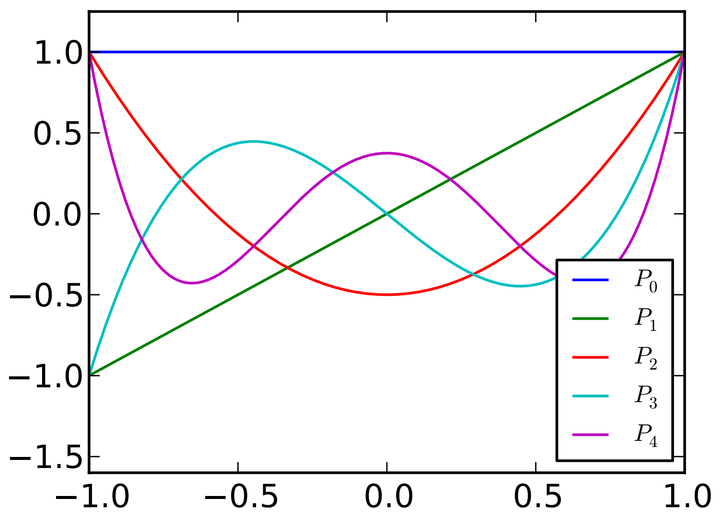
We now show how using this orthogonal basis simplifies the DG implementation. We approximate u over Ω_i with the Legendre basis functions P_j with

$$U_i = \sum_{j=0}^{p} c_{i,j} P_j. \tag{2.20}$$

We map each Ω_i to the canonical element with the bijection

$$x = \frac{x_{i+1} + x_i}{2} + \frac{h_i}{2}\xi, \tag{2.21}$$

Figure 2.1: The first five Legendre polynomials



where $\xi \in \Omega_0$. Then using each polynomial in our basis $v = P_k$, where $k = 0, \cdots, p$, we obtain $p+1$ equations

$$\frac{h_i}{2} \sum_{j=0}^{p} \frac{d}{dt} c_{i,j} (P_j, P_k) - (f(U_i), P_k') + f(U_i) P_k \,|_{-1}^{1} = 0. \tag{2.22}$$

The Legendre polynomials are normalized such that $P_k(1) = 1$. With this normalization,

$$(P_k, P_k) = \frac{2}{2k+1}. \tag{2.23}$$

We now obtain from (2.22) the ODE

$$\frac{h_i}{2k+1} \frac{d}{dt} c_{i,k} = (f(U_i), P_k') - f(U_i) P_k \,|_{-1}^{1}, \tag{2.24}$$

for $k = 0, \ldots, p$.

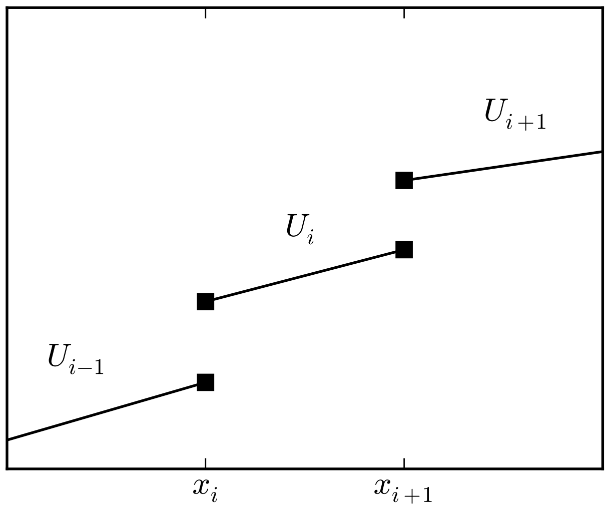
In vector form, we may then rewrite (2.24) as a system of ordinary differential equations

$$\frac{d}{dt} \mathbf{c} = L(\mathbf{c}), \tag{2.25}$$

where $\mathbf{c}$ is a global vector of solution coefficients and L is a vector function whose components are given by the right-hand side of equation (2.24).

2.3 Riemann Solvers

Figure 2.2: The discontinuities between local solutions over elements



The global approximation U is not well-defined at the boundaries of each element. As Figure 2.2 demonstrates, at each point x_i, both a left element Ω_{i-1} and a right element Ω_i evaluate the value of the approximation U. Since we do not enforce continuity over the boundaries of the elements as in standard finite element methods, we are not guaranteed that $U_{i-1}(x_i) = U_i(x_i)$, and thus $U(x_i)$ is twice-defined. To resolve these ambiguities, we use a Riemann solver.

The Riemann solver uses information from both the left and right elements to approximate the flux $f(U)$ at the boundary. At the left endpoint of element Ω_i, the flux $f(U_i)$ is approximated by the numerical flux function $f_n(U_{i-1}, U_i)$. Similarly, at the right endpoint of element Ω_i, the flux $f(U_i)$ is approximated by the numerical flux function $f_n(U_i, U_{i+1})$.

In the weak form of our one-dimensional scheme (2.3), the evaluation at the endpoints x_i, x_{i+1} of element Ω_i is given by

$$f(U_i)\phi_k \,\Big|_{x_i}^{x_{i+1}} = f_n(U_i, U_{i+1})\Big|_{x_{i+1}} \phi_k(x_{i+1}) - f_n(U_{i-1}, U_i)\Big|_{x_i} \phi_k(x_i). \tag{2.26}$$

We resolve the Riemann states in equation (2.26) using a numerical flux function.

Perhaps the simplest Riemann solver averages the values at the endpoints of two elements, as in

$$f_n(U_{i-1}, U_i) = \frac{f(U_{i-1}(x_i)) + f(U_i(x_i))}{2}. \tag{2.27}$$

This is known as the central flux solver. While the central flux solver is certainly easy to implement, it creates instabilities for nonlinear problems, and is not often used.

A more realistic numerical flux exploits the wave-like behavior of the solution. This flux, called the upwinding flux, uses information from "upwind" of a wave's direction, propagating information in the direction of the wave's motion. Assuming a positive wave velocity, the discontinuities at each element's interfaces are resolved by simply picking the left element's value, i.e., by setting

$$f_n(U_{i-1}, U_i) = f(U_{i-1}(x_i)). \tag{2.28}$$

For scalar conservation laws, resolving the ambiguities at the discontinuities is fairly straightforward through upwinding. For more complicated models, such as nonlinear problems and especially systems, the physical idea of wave direction becomes more obfuscated. In these cases, approximate Riemann solvers, such as Roe linearizations [33] or Lax-Friedrichs fluxes described in Section 2.7 are used.

2.4 The CFL Condition

The system of ODEs in equation (2.25) is solved using a numerical ODE solver. This involves stepping the solution forward in time by an appropriately small timestep Δt. Explicit ODE solvers step forward in time by using information from previous times.

Certain classes of numerical ODE solvers use multiple stages in computing the function approximation at the next time step. The classical fourth-order Runge-Kutta method solves the initial value problem

$$y'(t) = f(y, t), \quad y(t) = y_0, \tag{2.29}$$

using the value of y at the previous time step $y_n = y(t_n)$ to compute multiple stages

$$k_1 = \Delta t f\left(t_n, y_n\right), \tag{2.30}$$

$$k_2 = \Delta t f\left(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}k_1\right), \tag{2.31}$$

$$k_3 = \Delta t f\left(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}k_2\right), \tag{2.32}$$

$$k_4 = \Delta t f\left(t_n + \Delta t, y_n + k_3\right), \tag{2.33}$$

and combining them to obtain the function approximation at the next time

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \tag{2.34}$$

To remain stable, the numerical ODE solver must choose a sufficiently small value of Δt. For stability in the DG method, a Runge-Kutta time integration scheme of order $p+1$ requires

$$\Delta t_n \leq \frac{h_{\min}}{\lambda_{\max}(2p+1)}, \tag{2.35}$$

where $\lambda_{\max}$ is the maximum characteristic velocity at t_n of the system being modeled and $h_{\min}$ is the minimum element size. For more on numerical ODE solvers, see [27].

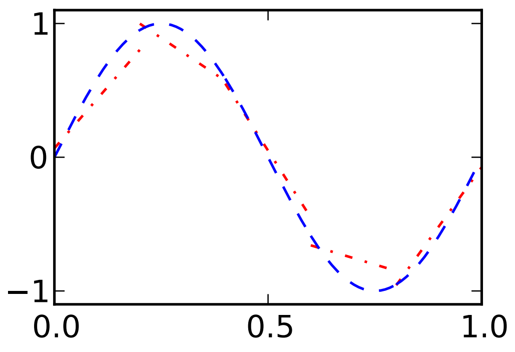
2.5 Refinement and Convergence

The DG method computes more accurate approximations of the exact solution through a process called refinement. We may approach refinement from two directions. Creating smaller elements in the computational domain will create more local approximations to the solution. This is called h-refinement, as the element lengths h_i becomes smaller. Increasing the dimension of the finite element space $V(\Omega)$, called p-refinement, creates higher order local approximations, as more basis functions are used to approximate the solution.

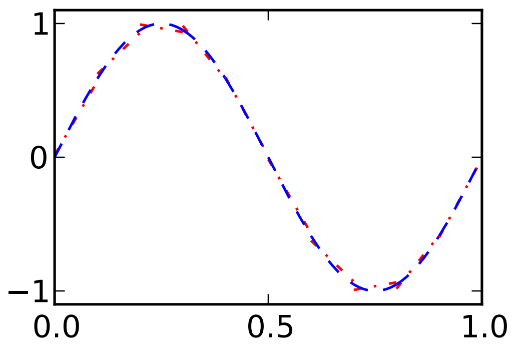
We will now demonstrate how h- and p-refinement of the DG method applied to a particular one-dimensional conservation law produces more accurate approximations of the exact solution. To demonstrate the effects of h-refinement, we hold $p = 1$ fixed and solve the linear advection equation

$$\partial_t u + \partial_x u = 0, \tag{2.36}$$

Figure 2.3: h-refinement with $p = 1$ for the linear advection problem in one dimension

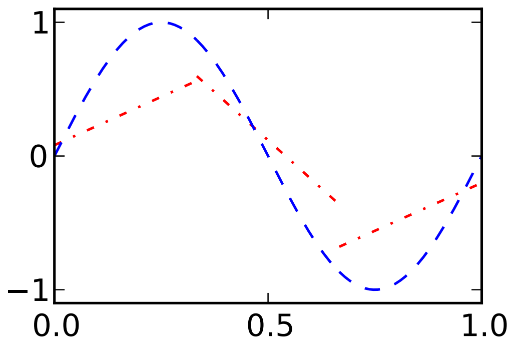


(a) $N = 5$

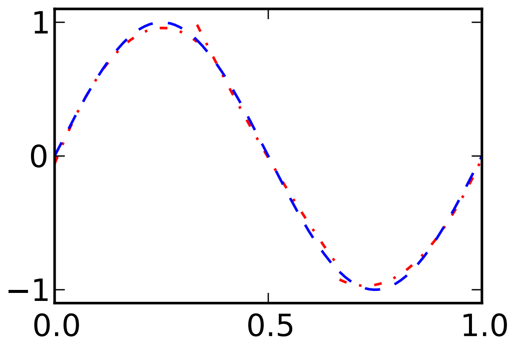


(b) $N = 10$

Figure 2.4: p-refinement with $N = 3$ applied to the linear advection equation



(a) $p = 1$



(b) $p = 2$

with sine wave initial conditions and periodic boundary conditions. After one period, we plot the computed solution in dash-dotted red over the analytical solution in dashed blue

$$u(x, t) = \sin(2\pi(x - t)), \tag{2.37}$$

in Figure 2.3 with two levels of h-refinement. Clearly, adding more elements to Ω has produced a better looking approximation.

To demonstrate the effects of p-refinement on the computed solution, we hold $N = 3$ elements fixed and compute the linear advection problem for one period. Figure 2.4 shows the computed solution in dash-dotted red plotted over the analytical solution in dashed blue.

Table 2.1: L^2 error and rate of convergence r for h- and p-refinement in the linear advection test problem

	$p = 1$		$p = 2$		$p = 3$		$p = 4$	
N	Error	r	Error	r	Error	r	Error	r
10	3.069E−2	-	1.211E−3	-	4.650E−5	-	1.060E−6	-
20	6.605E−3	2.238	1.513E−4	3.001	2.920E−6	3.993	3.337E−8	4.990
40	1.535E−3	2.084	1.891E−5	3.000	1.826E−7	4.000	1.054E−9	4.984
80	3.776E−4	2.023	2.364E−6	3.000	1.141E−8	4.000	3.325E−11	4.987

For $p = 2$, the approximation is nearly too similar to the analytical solution to distinguish. Even with a very coarse mesh, p-refinement produces good looking approximations.

To demonstrate the effectiveness of combining h-refinement and p-refinement, we compute the L^2 error

$$||u - U|| = \sum_{i=1}^{N} \sqrt{\int_{\Omega} (u - U_i)^2 dx}, \tag{2.38}$$

between the computed solution and the analytical solution after one period. The errors for different levels of h- and p-refinement are reported in Table 2.1 along with the convergence rate. While p-refinement seems to offer much greater improvement in accuracy over h-refinement, it is also more expensive computationally. On the other hand h-refinement demands more memory than p-refinement. Combining h-refinement with p-refinement yields a powerful tool to increase global accuracy.

The convergence and stability of the DG method has been analyzed extensively for one dimension in [6]. The theoretical order of convergence for one-dimensional DG methods is $O(h^{p+1})$, where h is the largest element size in the mesh, which nearly matches our convergence rate shown in Table 2.1.

2.6 The One-Dimensional DG Method For Systems

We now consider the one-dimensional conservation law for a system of equations

$$\partial_t \mathbf{u} + \partial_x \mathbf{f}(\mathbf{u}) = \mathbf{0}, \tag{2.39}$$

where $\mathbf{u} = (u_1, \ldots, u_n)$ is now a vector of n variables, over a computational domain $\Omega \in \mathbb{R}$, with a sufficiently smooth flux function $\mathbf{f}$, initial conditions

$$\mathbf{u}(x, 0) = \mathbf{u}_0(x), \tag{2.40}$$

and appropriate boundary conditions.

Few aspects of the derivation of the DG method change for systems of conservation laws. Hence, we proceed similarly to the scalar case (2.1). We choose a test function space $V(\Omega_i)$ and let $v \in V(\Omega_i)$. Multiplying both sides of equation (2.39) by v and integrating over Ω_i, integrating the second term by parts, yields the weak formulation

$$\int_{\Omega_i} \partial_t \mathbf{u} v \, dx - \int_{\Omega_i} \mathbf{f}(\mathbf{u}) v' \, dx + \mathbf{f}(\mathbf{u}) v \Big|_{x_i}^{x_{i+1}} = 0, \tag{2.41}$$

for each element Ω_i.

We map each Ω_i to the element $\Omega_0 = [-1, 1]$ and approximate $\mathbf{u}$ over the element Ω_i with a vector $\mathbf{U}_i = (U_i^1, \ldots, U_i^n)$ where each $U_i^m \in V(\Omega_0)$. Letting $\Phi = \{\phi_j\}_{j=0}^p$ be an orthogonal basis for $V(\Omega_0)$, we write each $\mathbf{U}_i$ as

$$\mathbf{U}_i = \sum_{j=0}^{p} \mathbf{c}_{i,j} \phi_j, \tag{2.42}$$

where each $\mathbf{c}_{i,j}$ is now a vector of coefficients $\mathbf{c}_{i,j} = (c_{i,j}^1, \ldots, c_{i,j}^n)$ so that each U_i^m satisfies

$$U_i^m = \sum_{j=0}^{p} c_{i,j}^h \phi_j. \tag{2.43}$$

Using $\mathbf{U}_i$ as our approximation of u over Ω_i in equation (2.41) and exploiting the orthogonality of basis Φ yields

$$\frac{h_i}{2} (\phi_k, \phi_k) \frac{d}{dt} \mathbf{c}_{i,j} = (\mathbf{f}(\mathbf{U}), \phi_k') - \mathbf{f}(\mathbf{U}) \phi_k \Big|_{-1}^{1}. \tag{2.44}$$

We resolve discontinuities at the element interfaces using a numerical flux function $\mathbf{f}_n$. The flux function $\mathbf{f}$ along the boundaries of each element x_i is approximated by a numerical flux function $\mathbf{f}_n(\mathbf{U}_i, \mathbf{U}_{i+1})$. We approximate the flux function with an approximate Riemann solver, such as a local Lax-Friedrichs Riemann solver, described in Section 2.7.

Equation (2.44) can be written as a system of ODEs

$$\frac{d}{dt}\mathbf{c} = L(\mathbf{c}), \tag{2.45}$$

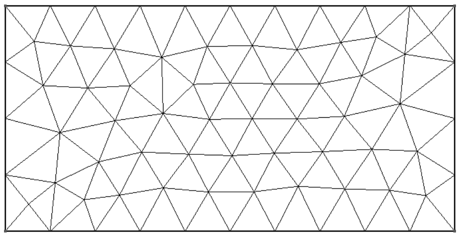
where $\mathbf{c}$ is a flattened vector of the coefficients for each U_i^m. The flattened vector for $\mathbf{c}$ may be created by flattening the matrix

$$\mathbf{C} = \begin{pmatrix} c_{1,0}^0 & c_{1,1}^0 & \cdots & c_{1,p}^0 \\ c_{2,0}^0 & c_{2,1}^0 & \cdots & c_{2,p}^0 \\ \vdots & \vdots & \ddots & \vdots \\ c_{N,0}^0 & c_{N,1}^0 & \cdots & c_{N,p}^0 \\ \vdots & \vdots & \ddots & \vdots \\ c_{1,0}^n & c_{1,1}^n & \cdots & c_{1,p}^n \\ c_{2,0}^n & c_{2,1}^n & \cdots & c_{2,p}^n \\ \vdots & \vdots & \ddots & \vdots \\ c_{N,0}^n & c_{N,1}^n & \cdots & c_{N,p}^n \end{pmatrix}, \tag{2.46}$$

row by row where each $c_{i,j}^m$ is the leading coefficient for the j-th basis function of the m-th variable in the system over element Ω_i.

2.7 The Two-Dimensional DG Method For Systems

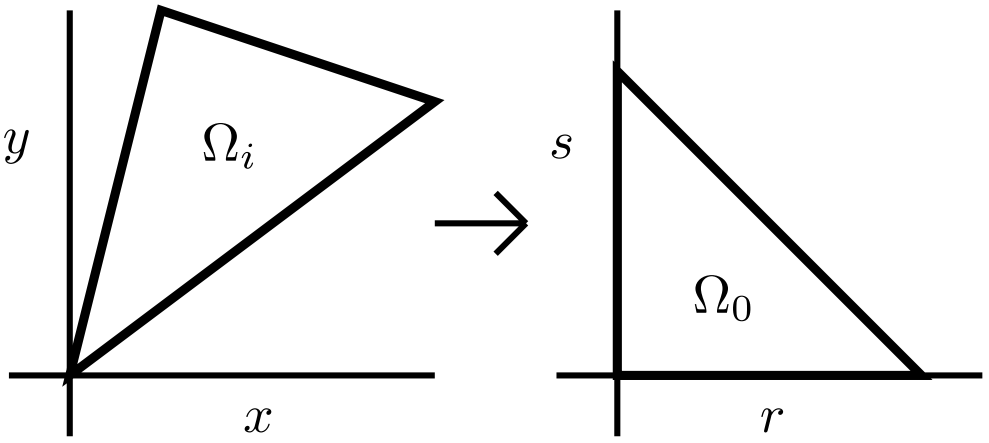
Figure 2.5: A two-dimensional domain Ω partitioned into a mesh of triangular elements Ω_i



We now consider the two-dimensional system of equations

$$\partial_t \mathbf{u} + \nabla \cdot \mathbf{F}(\mathbf{u}) = \mathbf{0}, \tag{2.47}$$

Figure 2.6: Mapping Ω_i to the canonical triangle Ω_0 with vertices (0,0), (1,0), and (0,1)



for a vector $\mathbf{u} = (u_1, u_2, \ldots, u_n)$ of n variables, over a computational domain $\Omega \subseteq \mathbb{R}^2$, with a sufficiently smooth flux function $\mathbf{F} = [F_1, F_2]$ where F_1 and F_2 are the fluxes in the x and y directions, respectively. We assume initial conditions

$$\mathbf{u}(x, y, 0) = \mathbf{u}_0(x, y), \tag{2.48}$$

and appropriate boundary conditions. We partition the two-dimensional domain Ω into a mesh of triangles Ω_i as in Figure 2.5.

We proceed as before and multiply (2.47) by a function $v \in V(\Omega_i)$, and integrate by parts to get the weak formulation

$$\frac{d}{dt}\int_{\Omega_i} vu\, d\mathbf{x} + \int_{\Omega_i} \nabla v \cdot \mathbf{F}(u)\, d\mathbf{x} - \int_{\partial\Omega_i} v\mathbf{F}(u) \cdot \mathbf{n}_i\, ds = 0, \tag{2.49}$$

where $\mathbf{n}_i$ is the outward facing unit vector along element i's edges.

The weak formulations for the one-dimensional system (2.41) and the two-dimensional system (2.49) are identical. The differences in implementation are due to the higher space dimension. Since Ω_i is a triangle, the two-dimensional integral becomes a volume integral instead of a one-dimensional line integral. Also, the boundary $\partial\Omega_i$ no longer consists of two endpoints; it is now the three edges of the triangle Ω_i. The integral over the boundary, then, becomes a surface, i.e., line integral in two dimensions.

Again, we map each element Ω_i in the domain to a canonical element Ω_0. As element Ω_i is a triangle, we choose the canonical element Ω_0 to also be a triangle. The mapping, shown in Figure 2.6, maps each coordinate $\mathbf{x} = (x, y) \in \Omega_i$ to a new coordinate $\mathbf{r} = (r, s) \in \Omega_0$. The canonical triangle used in this implementation has vertices at $(0, 0), (1, 0)$, and $(0, 1)$. The mapping to the canonical triangle in this case is bijective and is given by

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 - r - s \\ r \\ s \end{pmatrix}, \tag{2.50}$$

where (x_i, y_i) are the vertices of the given element. The Jacobian matrix for this mapping (2.50) is constant for straight-sided triangles, and is given by

$$J_i = \begin{bmatrix} x_r & y_r \\ x_s & y_s \end{bmatrix}. \tag{2.51}$$

Using this mapping to Ω_0, the volume integral in the weak formulation (2.49) over element Ω_i becomes

$$\int_{\Omega_i} \nabla v \cdot \mathbf{F}(\mathbf{u})\, d\mathbf{x} = \int_{\Omega_0} (J_i^{-1} \nabla v) \cdot \mathbf{F}(\mathbf{u}) |\det J_i|\, d\mathbf{r}. \tag{2.52}$$

We also map the edges of each element to the canonical interval $I_0 = [-1, 1]$. This mapping is given by

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \begin{pmatrix} \frac{1}{2}(1 - \xi) \\ \frac{1}{2}(1 + \xi) \end{pmatrix}, \tag{2.53}$$

where (x_i, y_i) are the endpoints of the given edge and $\xi \in I_0$. This mapping produces an additional factor in the surface integral due the chain rule, as

$$ds = \frac{1}{2}\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}\, d\xi. \tag{2.54}$$

This additional term is simply half the length of each edge. For each element Ω_i, define the constant

$$l_{i,q} = \frac{1}{2}\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}, \tag{2.55}$$

for each edge q of Ω_i. Here, (x_j, y_j) denote the vertices of the specific edge. Using the mapping (2.53) to I_0, the surface integral in the weak formulation (2.49) for element Ω_i becomes

$$\int_{\partial\Omega_i} v\mathbf{F}(\mathbf{u}) \cdot \mathbf{n}_i\, ds = \sum_{q=1}^{3} \int_{I_0} v\mathbf{F}(\mathbf{u}) \cdot \mathbf{n}_{i,q} l_{i,q}\, d\xi, \tag{2.56}$$

where $\mathbf{n}_{i,q}$ denotes the outward facing normal vector along edge q.

A basis for $V(\Omega_0)$ consisting of polynomials of degree p or less contains

$$N_p = \frac{1}{2}(p+1)(p+2), \tag{2.57}$$

basis functions. Let $\Phi = \{\phi_j\}_{j=1}^{N_p}$ be an orthogonal basis of N_p basis functions for $V(\Omega_0)$. Let our approximation U_i over element Ω_i be a linear combination of basis functions ϕ_j

$$\mathbf{U}_i = \sum_{j=1}^{N_p} \mathbf{c}_{i,j}\phi_j, \tag{2.58}$$

where $\mathbf{c}_{i,j}$ is again a vector of coefficients as in the one-dimensional system (2.42). Using this orthogonal basis and choosing each test function $\phi_j \in \Phi$ produces a system of ODEs

$$\frac{d}{dt}\mathbf{c}_{i,j} = \frac{1}{|\det J_i|}\left(\int_{\Omega_0} \mathbf{F}(\mathbf{U}_i)\cdot(J_i^{-1}\nabla\phi_j)\,|\det J_i|d\mathbf{r} - \sum_{q=1}^{3}\int_{I_0}\phi_j\mathbf{F}(\mathbf{U}_i)\cdot\mathbf{n}_{i,q}l_{i,q}\,d\xi\right). \tag{2.59}$$

Equation (2.59) can then be written as a system of ODEs

$$\frac{d}{dt}\mathbf{c} = L(\mathbf{c}), \tag{2.60}$$

for a flattened vector of coefficients $\mathbf{c}$. This flattened vector may be created by flattening the matrix

$$\mathbf{C} = \begin{pmatrix} c_{1,1}^0 & c_{1,2}^0 & \cdots & c_{1,N_p}^0 \\ c_{2,1}^0 & c_{2,2}^0 & \cdots & c_{2,N_p}^0 \\ \vdots & \vdots & \ddots & \vdots \\ c_{N,1}^0 & c_{N,2}^0 & \cdots & c_{N,N_p}^0 \\ \vdots & \vdots & \ddots & \vdots \\ c_{1,1}^n & c_{1,2}^n & \cdots & c_{1,N_p}^n \\ c_{2,1}^n & c_{2,2}^n & \cdots & c_{2,N_p}^n \\ \vdots & \vdots & \ddots & \vdots \\ c_{N,1}^n & c_{N,2}^n & \cdots & c_{N,N_p}^n \end{pmatrix}, \tag{2.61}$$

row by row.

The order p orthogonal basis functions over the canonical element used in this implementation can be found in [24]. They are given by a product of Jacobi polynomials $P_j^{\alpha,\beta}$ and Legendre polynomials P_j

$$\psi_j^k = P_{k-j}^{0,2j+1}(1-2r)(1-r)^j P_j\left(1 - \frac{2s}{1-r}\right), \tag{2.62}$$

where $k = 0, \dots, p$ denotes the degree of the polynomial and $j = 0, \dots, k$. The first six polynomials ψ_j^k for $k = 0, 1$ over this canonical triangle are shown in Figure 2.7.

As we do not impose continuity on the edges of each element, the value of the approximation is twice-defined along every edge. At the vertices of an element, the value of the approximation may even be defined more than twice. As in the one-dimensional case, the flux function $\mathbf{F}$ must then be approximated with a numerical flux function $\mathbf{F_n}$ to resolve the discontinuity along the edges of neighboring elements. When elements Ω_l and Ω_r share an edge, the numerical flux function will evaluate $\mathbf{F_n}(\mathbf{U}_l, \mathbf{U}_r)$ at integrations points on that edge. This numerical flux function represents the only communication between elements, making the DG method in two dimensions element local.

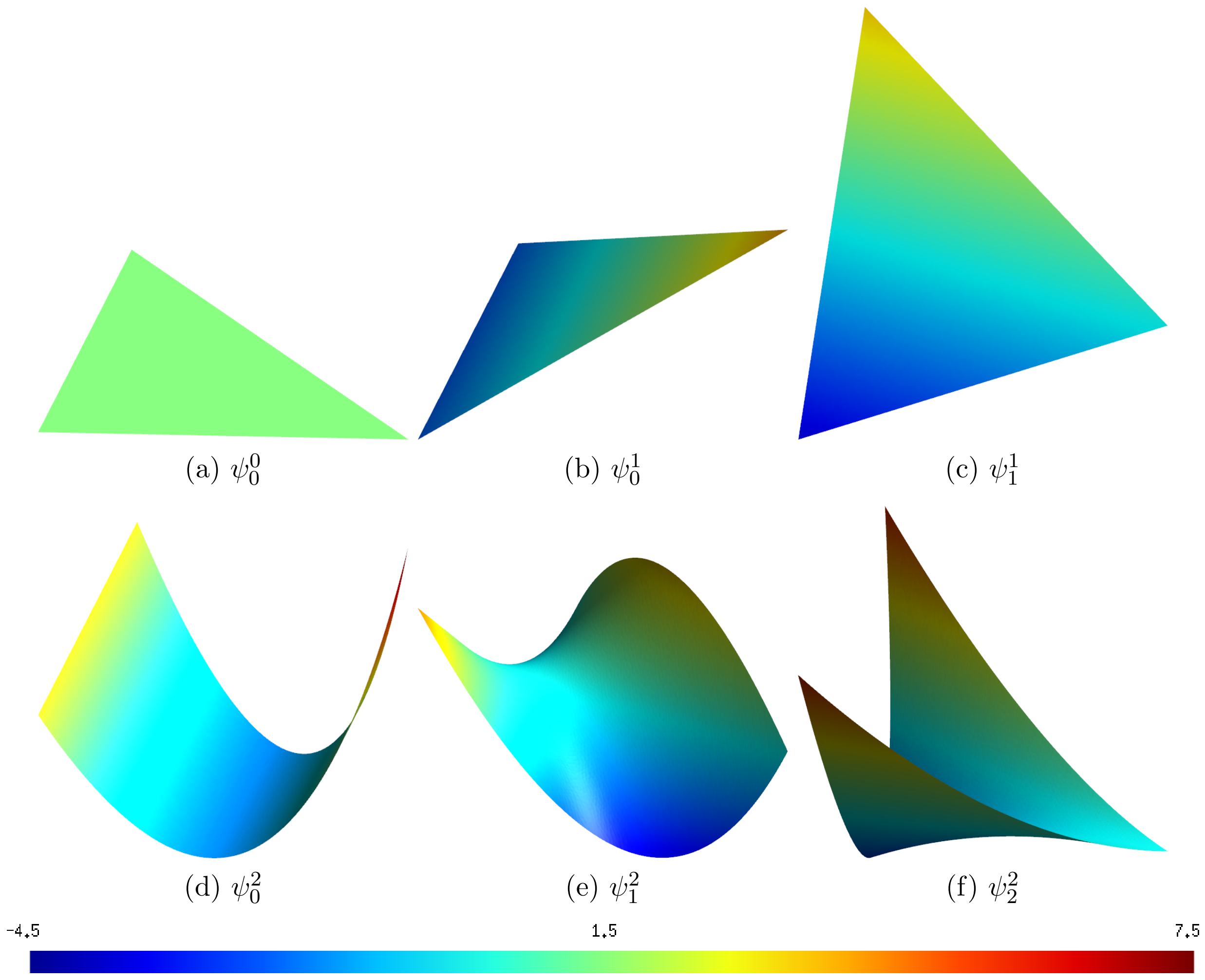
The local Lax-Friedrichs Riemann solver mimics the central flux (2.27), but contains an additional diffusive term to stabilize the numerical method. This Riemann solver, while not extremely accurate, is easy to implement and cheap to compute. We approximate the flux along an edge with the numerical flux function

$$\mathbf{F_n}(\mathbf{U}_l, \mathbf{U}_r) = \frac{1}{2}\left[(\mathbf{F}(\mathbf{U}_l) + \mathbf{F}(\mathbf{U}_r)) \cdot \mathbf{n} + |\lambda|(\mathbf{U}_l - \mathbf{U}_r)\right], \tag{2.63}$$

where λ represents the largest in magnitude eigenvalue of the Jacobian of $\mathbf{F}$ and $\mathbf{n}$ is the outward facing normal vector.

The convergence of the DG method for two-dimensional systems of equations is discussed in [6]. In practice, the convergence rate is the same as in one dimension, $O(h^{p+1})$, where here h is the radius of the largest inscribed circle of all elements in the mesh.

Figure 2.7: Orthogonal polynomials over the canonical triangle Ω_0



(a) ψ_0^0 (b) ψ_0^1 (c) ψ_1^1

(d) ψ_0^2 (e) ψ_1^2 (f) ψ_2^2

Chapter 3

Parallel Computing

The traditional paradigm in computing has been serial instructions carried out sequentially on a single central processing unit. Single processing units became more powerful due to frequency scaling, that is, by making the single processing unit compute instructions faster. Unfortunately, heat generation, among other physical constraints, prohibits unbound frequency scaling. This, among other factors, greatly restricted high performance serial computing.

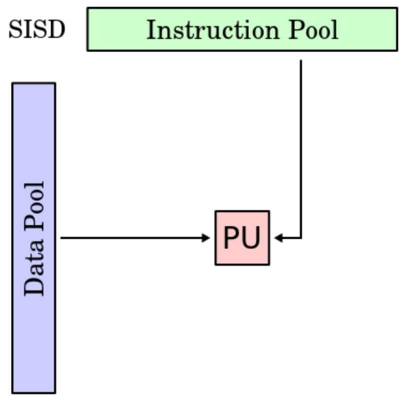
Parallel computing avoids the need for a strong single central processing unit idea by distributing computation among many processing units. This increases theoretical computation power without introducing the same physical constraints as frequency scaling in a single central processing unit. Indeed, theoretical computation power is doubled by simply doubling the number of processors.

Unfortunately, parallel computing introduces new constraints of its own. Computation can no longer be thought of as sets of serial instructions to compute. Large scale problems must be divided into smaller ones to take full advantage of parallel computing. The smaller problems must then be distributed evenly to the many individual processing units.

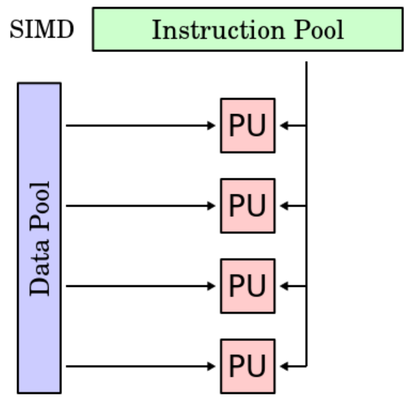
Problem division and distribution introduce new problems of their own. Communicating information between independent processors requires passing data from some global memory source to processor-local memory sources. Concurrent global memory data access introduces race conditions, as two processors cannot write information to the same location at the same time. Separated processors must be synchronized to keep data in the correct state.

3.1 Flynn's Taxonomy

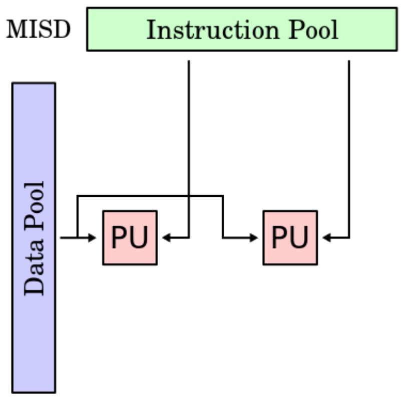
Figure 3.1: Flynn's taxonomy



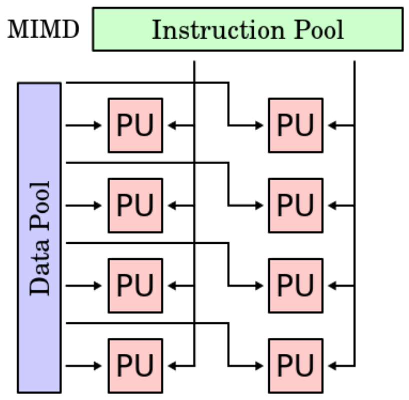
(a) Single instruction, single data



(b) Single instruction, multiple data



(c) Multiple instruction, single data



(d) Multiple instruction, multiple data

Flynn's taxonomy classifies computing architectures by dividing them into four classes based on the combination of distinct instruction sets and data sources. Each instruction set is computed by a thread, an individual member of a sequence of instructions. A computer architecture is said to be either single-threaded, operating entirely in serial, or multi-threaded, with some degree of parallelism as multiple threads execute concurrently. A computer architecture has either a single set of instructions, or multiple sets of instructions which operate on either a single data source or many data sources. Thus, Flynn's taxonomy identifies four unique combinations of computer architectures: single instruction, single data (SISD), single instruction, multiple data (SIMD), multiple instruction, single data (MISD), and multiple instruction, multiple data (MIMD).

Figure 3.1 outlines the four computer architectures introduced by Flynn's taxonomy [36]. Figure 3.1*a* demonstrates the SISD architecture, used in traditional serial computing whenever a single processing unit manipulates a single data source in serial. Figure 3.1*b* demonstrates the SIMD architecture, a popular parallel computing model used whenever multiple processing units manipulate a data source using identical instructions. Figure 3.1*c* demonstrates the MISD architecture, a less popular parallel computing model used when different instruction sets are distributed among the processors to operate on the same block of global data. Figure 3.1*d* demonstrates the MIMD architecture, a parallel programming model often used in distributed systems whenever data is entirely or nearly entirely localised to the individual processing units.

3.2 Graphics Processing Units

Dedicated graphics processing units were created with a single purpose in mind: quickly manipulate and display images. Early GPU hardware accelerated 2D performance, enabling operating systems to use graphical user interfaces over the standard terminal. GPU hardware then accelerated 3D performance, allowing 3D images to be rendered in real-time in high resolutions. The consumer demand for more realistic graphics rendered more quickly in even higher resolutions drove the industry to create even more powerful GPUs.

As graphics rendering is a highly parallelizable activity, GPUs have a highly parallel structure. They are a dedicated piece of hardware, able to compute independently from the CPU. They compute their own machine code on their own processing units with their own dedicated video memory, making them almost a completely independent machine.

Traditionally, programming for GPUs strictly involved commanding the GPU to render graphics, as graphics rendering was their original purpose. Programming languages such as OpenGL were created strictly for programming on GPUs. Moreover, they restricted GPU computation to mainly visualization and image manipulation. In order to perform other computations on GPUs, programmers needed to rewrite their general purpose instructions as graphics related instructions. As such, these early languages were not suitable for general purpose computing on GPUs.

With the advent of general purpose GPU computing languages such as the Compute Unified Device Architecture (CUDA) and OpenCL, computer programmers no longer needed to use arcane languages such as OpenGL for general purpose computing. These new general purpose languages made non graphics-related problems more approachable and even practical. GPUs, having a highly parallel structure, were then used to solve so-called embarrassingly parallelizable problems, such as matrix vector multiplications and brute-force searches in cryptography. As GPUs began to support double-precision floating point computations, implementations of more complex parallelizable problems in scientific computing such as computational fluid dynamics materialized. The latest GPU supercomputers, such as the NVIDIA Tesla K20, offer over a teraflop of computing power for only USD $3,500. This low cost and high performance have made GPUs a formidable tool for scientific computing.

Most graphics processing units use the SIMD parallel computing architecture described by Flynn's taxonomy. A global data set in dedicated video memory is manipulated by many GPU processing units running the same instruction set. Programming may be done from a thread perspective, as each thread reads the same instruction set, using it to manipulate different data. Threads have a unique numeric identifier, such as a thread index, allowing each thread to read from a different data location, or execute a different set of instructions based on its index.

3.2.1 CUDA

As mentioned above, this project uses CUDA, a proprietary GPU programming language and architecture developed by NVIDIA for use on NVIDIA GPUs. CUDA adopts the SIMD parallel computing architecture with programming done from a thread perspective. Threads manipulate data from dedicated video memory, performing computations on CUDA cores, multicore processors located on the GPU. For a more in-depth overview, including a tutorial, of the CUDA programming language, see [31].

Extensive programmer support is provided through CUDA-accelerated software libraries, such as cuBLAS, a basic linear algebra subroutine library, and Thrust, a flexible library of various parallelizable algorithms such as sorts and reductions. Further CUDA support is included in recent versions of scientific computing software such as MATLAB, which allows users to speed up many computations by offloading them to the GPU using their Parallel Computing Toolbox [28]. NVIDIA also provides compilers for programmers wanting to develop their own code using CUDA. These compilers can use code written in either C/C++ or Fortran.

The basic unit of computation in CUDA is the thread. Each thread has a unique identifier to it: a thread index. Threads are grouped into multiples of 32, called warps.

Every thread in the same warp runs the same set of instructions and is executed on the same GPU processing unit, called a Streaming Multiprocessor (SM). Each thread running on an SM shares a small amount of fast, cached register memory. An SM unit is assigned a collection of warps, called a block. These blocks are then further divided among grids. As execution of every warp in a block is completed on an SM, a new block from the grid is then assigned to that SM.

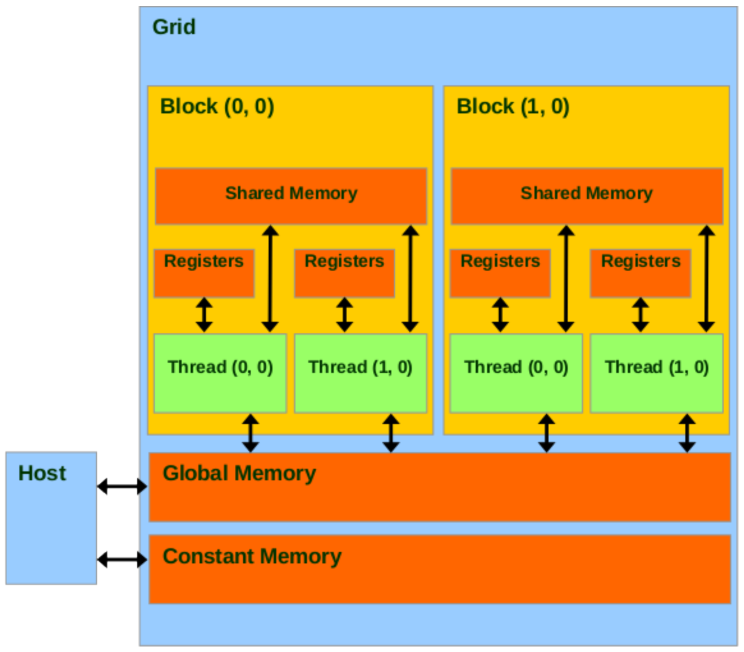
Each warp must execute the same instruction set. Boolean expressions pose problems, as two threads in the same warp may evaluate a boolean condition to different values. When this happens, the warp splits into branches, where each branch of threads executes a different path of code split by the boolean. This branching, called warp divergence, harms performance as both instructions must be run for this warp. On the other hand, when every thread in the same warp evaluates the boolean condition to the same value, they avoid warp divergence by taking the same path.

CUDA machine code is written in kernels - parallel code to be executed on the GPU. Typically, all threads on the GPU run the same kernel at the same time. Kernels run exclusively on the GPU and may therefore only use data located on the GPU's video memory. Any data created on the CPU must therefore be transfered to the GPU in order for threads to access it. Data is transfered between the GPU and CPU through the PCIe bus, which introduces latency. Furthermore, PCIe data transfer is limited in bandwidth; PCIe 3.0 has a maximum of 32GB/s throughput. Every data transfer between the GPU and CPU increases computation time, and should be avoided whenever possible.

CUDA has several hierarchies of memory, detailed in Figure 3.2, taken from [37]. GPU memory must be allocated separately from host (CPU) memory. The largest available storage on the GPU is the global memory. Global memory on the GPU is both readable and writable, is very large, but is uncached and very slow. Every memory access of GPU global memory impacts performance greatly due to latency and memory lookup time. Texture memory is a type of constant global memory which can sometimes be used to decrease these latencies.

While global memory is large, slow, and uncached, constant memory is small, quick, and cached. Each SM has access to a limited amount of fast, cached register memory. Register memory provides the fastest access for threads, but is only local to their specific thread. Whenever a thread requests more register memory than the SM can provide, local memory is used. Local memory, while allowing each thread to use gratuitous amounts of memory, has a very long memory access time, nearing that of global memory. Threads in the same block also have access to shared memory. This shared memory is also fast, but is limited in size and difficult to properly take advantage of.

Figure 3.2: GPU Memory Hierarchy



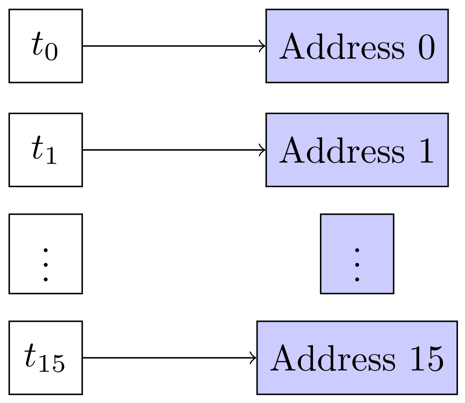
Load balancing is done entirely by CUDA's warp scheduler. In order to take full advantage of the processing power available, enough warps must be created to fully saturate the device. Full device utilization will not occur until each SM is simultaneously processing a different warp. The warp scheduler will attempt to fully utilize each SM by assigning ready warps to idle SMs. When memory access requests introduce latency, e.g., global memory access requests, the warp scheduler will attempt to hide the latency by swapping out the idle warps and replacing them with ready warps.

The number of active threads able to run on a single SM depends on two factors. First, the architecture of the SM determines the maximum allowable number of threads. For example, the NVIDIA Fermi GPU architecture may have up to 48 active warps on each SM. Second, the total amount of memory requested by every thread must not exceed the available memory on the SM.

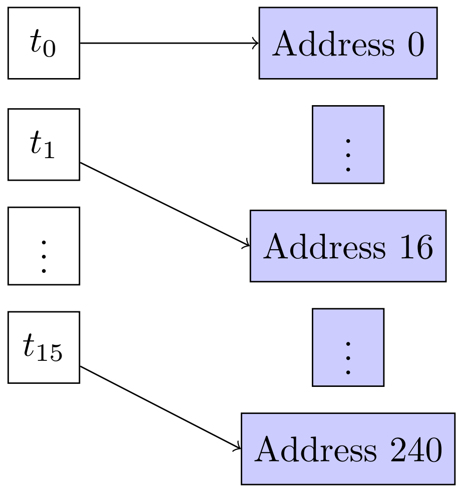
Thread occupancy measures the proportion of currently active warps to the maximum allowable number of active warps. When the combined memory requirement of every thread in a block exceeds the amount of available register memory, the SM will not run every thread in the block at once. It will run only as many threads as it has memory for them to share. As a result, either less threads must run at once in order to share the SM memory, or the SM must use local memory storage rather than register memory. When less threads run at once, thread occupancy, and thus, performance decreases. On the other hand, local memory introduces a significant amount of latency compared to register memory. The programmer must therefore maximize thread occupancy while resorting to local memory usage only as a last resort.

The arithmetic intensity, which we define as the ratio of computation time to memory access time, greatly influences the overall effectiveness of any implementation in CUDA. Problems with low arithmetic intensity spend most of their run time waiting during latencies in memory transactions. Problems with high arithmetic intensity, on the other hand, allow CUDA's warp scheduler to hide memory latency behind computation by scheduling ready warps ahead of those waiting on memory transactions.

Figure 3.3: Data coalescion in a half warp



(a) Coalesced data access pattern



(b) Uncoalesced data access pattern

Whenever every thread in a half-warp accesses the same region of global memory at the same time, this memory transaction may be coalesced. Figure 3.3 shows an example of coalesced data access versus uncoalesced data access. In the coalesced data access example, thread t_i accesses memory location i, thread t_{i+1} accesses memory location $i+1$, and so on for this warp. In this case, these data access operations are coalesced into a single access operation rather than 16 separate ones. In the uncoalesced data access example, thread t_i accesses memory location $16 \times i$. The data at address $16 \times i$ is not located nearby the data at address $16 \times (i-1)$, and is thus not local from a thread perspective. In this case, the SM must perform sixteen separate data access operations. As GPU computing is dominantly memory bound, data coalescion is of utmost importance for improving computation time. Later versions of CUDA allow permutations, among other things, of these transactions, so long as data locality is preserved; i.e., so long as all threads in a half-warp access the same nearby data locations.

3.2.2 CUDA Example

To further understand parallel programming with CUDA, we present a simple example. We will write a GPU kernel to add two arrays together.

```
// declare GPU memory
__global__ int *gpu_result;
__global__ int *gpu_A;
__global__ int *gpu_B;
```

We first declare three GPU variables, residing in GPU global memory. They may be accessed and modified by any thread.

```
// addition kernel
__global__ void add(int *A, int *B, int *result, int size) {
    // get unique thread identifier
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // only run if the thread index doesn't exceed the array size
    if (idx < size) {
        result[idx] = A[idx] + B[idx];
    }
}
```

Next we declare the GPU kernel `add`. This kernel adds the data from `A` to `B` and stores the sum in `result`. The thread running this kernel first finds its unique thread index `idx`. If the index is less than the size of the array, then this thread should add `A[idx]` to `B[idx]` and store the result in `result[idx]`. Note that each read of `A[idx]` and `B[idx]` will be coalesced, as will the write to `result[idx]` as the data is aligned along with the thread index.

```
int main(void) {
    // the size of A and B
    int size   = 1000;
    // number of threads to run per block
    int threads_per_block = 128;
    // number of blocks to run to reach size
    int blocks = size / threads_per_block + 1;
```

Next we create the CPU code which will allocate and initialize the GPU memory and then call the GPU kernel function. In this case, we want to add 1,000 integers, so we set `size` to 1,000 and create at least 1,000 threads to run our kernel. As CUDA requires the number of threads grouped to a block to be a power of two, we create 128 threads to run in each block. As a result, we are not able to create exactly 1,000 threads, so we create 1,024 threads spread across 8 blocks, with each block containing 128 threads. The last twenty-four threads with index 1,000 through 1,024 will do no computation, as their indices are larger than `size`.

```
    // allocate memory on the GPU
    cudaMalloc((void **) &gpu_result, size * sizeof(int));
    cudaMalloc((void **) &gpu_A,   size * sizeof(int));
    cudaMalloc((void **) &gpu_B,   size * sizeof(int));

    // allocate memory on the CPU
    int *result = (int *) malloc(size * sizeof(int));
    int *A     = (int *) malloc(size * sizeof(int));
    int *B     = (int *) malloc(size * sizeof(int));

    // fill A and B with ones
    memset(A, 1, size * sizeof(int));
    memset(B, 1, size * sizeof(int));

    // copy A and B over to the GPU
    cudaMemcpy(gpu_A, A, size * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(gpu_B, B, size * sizeof(int), cudaMemcpyHostToDevice);
```

We allocate the three GPU global variables on the GPU to be large enough to hold our arrays of integers. Then, we create three CPU variables of the same size for transferring data back and forth. We fill `A` and `B` with ones and transfer them over to GPU global memory `gpu_A` and `gpu_B`.

```
    // run the GPU kernel on the GPU
    add<<<threads, blocks>>>(gpu_A, gpu_B, gpu_result, size);

    // copy back the result
    cudaMemcpy(result, gpu_result, size * sizeof(int), cudaMemcpyDeviceToHost);

    return 0;
}
```

Finally, we run the kernel `add` with 128 threads and 8 blocks. The GPU global variable `gpu_result` is then copied back to the CPU, containing the sums of `gpu_A` and `gpu_B`.

Chapter 4

Implementation

We now detail our implementation of the DG method on a GPU using CUDA. Our implementation solves two-dimensional scalar and systems of hyperbolic conservation laws, as we described in Section 2.7. This implementation consists of three main components. The first component transforms our computational domain into a mesh and creates the mappings for elements and edges of that mesh. In addition, it allocates an appropriate amount of storage in GPU memory, and transfers the mesh mappings from CPU memory to GPU memory. The second component sets the specific parameters for the problem being modeled; namely, it describes the initial conditions, boundary conditions, and, most importantly, the flux function **F**. The third component contains the kernels tasked with computing equations (4.1) and (4.2) described in Section 2.7. Additionally, the third component combines the surface integral contributions with the volume integral contributions and steps forward in time using a numerical ODE solver.

4.1 Parallel Computing with DG Methods

Because the DG method uses independent local approximations of the solution U_i over each element in the mesh Ω_i, it is especially open to parallelization. Recall the weak formulation of the PDE in two dimensions as described in equation (2.59). Each equation for advancing the coefficient $\mathbf{c}_{i,j}$ in time can be expressed as a combination of a surface integral contribution

$$\int_{\partial\Omega_i} \phi_j \mathbf{F}(\mathbf{U}_i) \cdot \mathbf{n}_i \, ds, \tag{4.1}$$

and a volume integral contribution

$$\int_{\Omega_i} \mathbf{F}(\mathbf{U}_i) \cdot \nabla \phi_j \, d\mathbf{x}. \tag{4.2}$$

We map each element Ω_i to the canonical triangle Ω_0 with vertices (0,0), (1,0), and (0,1). Each edge $e_q, q = 1, 2, 3$ in $\partial\Omega_i$ is mapped to a side of the canonical triangle, and that side is then mapped to the canonical interval $I_0 = [-1, 1]$. As a result, (4.2) and (4.1) remain the same for each element. Thus, the same instruction set may be used to compute each element's volume and surface integral contributions. This suggests that the DG method may be implemented in parallel using a SIMD programming architecture.

The volume integral (4.2) may be computed independently from the surface integral (4.1), as it requires only the local coefficients $\mathbf{c}_{i,j}, j = 1, \ldots, N_p$ for element Ω_i and the inverse Jacobian matrix J_i^{-1}. The surface integral (4.1) requires information from the local element Ω_i and the three neighboring elements in order to solve the Riemann problem along each edge by using the numerical flux function $\mathbf{F}_\mathbf{n}$.

Two parallelization techniques are commonly implemented here. In one technique, each thread first computes the volume integral contributions for one element, then computes the surface integral contributions from each edge for that same element. The two contributions are summed and added to a right-hand side variable for each coefficient $\mathbf{c}_{i,j}$ and test function $\phi_j, j = 1, \ldots, N_p$. We refer to this technique as the single kernel implementation.

A second parallelization technique separates the volume integral computation from the surface integral computation. Surface integral (4.1) is a sum of three separate edges' line integrals. We thus consider this surface integral as the sum

$$\begin{aligned}\int_{\partial\Omega_i} \phi_j \mathbf{F}(\mathbf{U}_i) \cdot \mathbf{n}_i \, ds = &\int_{s_1} \phi_j \mathbf{F}(\mathbf{U}_i) \cdot \mathbf{n}_{i,1} l_{i,1} \, d\xi \\ &+ \int_{s_2} \phi_j \mathbf{F}(\mathbf{U}_i) \cdot \mathbf{n}_{i,2} l_{i,2} \, d\xi \\ &+ \int_{s_3} \phi_j \mathbf{F}(\mathbf{U}_i) \cdot \mathbf{n}_{i,3} l_{i,3} \, d\xi \end{aligned} \tag{4.3}$$

where s_q is a side of the canonical triangle, $\mathbf{n}_{i,q}$ is the corresponding outward facing normal vector at edge e_q, and $l_{i,q}$ is half of the corresponding edge e_q's length as discussed in Section 2.7. By considering each line integral in equation (4.3) as a separate, independent problem, we allow a greater degree of parallelism. A thread t_i may compute the volume integral contribution over element Ω_i for each $j = 1, \ldots, N_p$ coefficient $\mathbf{c}_{i,j}$ and store each result. A separate thread t_k computes the surface integral contribution over one edge e_k of an element as in (4.3). The volume integral contributions and surface integral contributions are later combined by yet another thread to form the full right-hand side from equation (2.59). We refer to this technique as the multiple kernel implementation.

An advantage of the single kernel implementation over the multiple kernel implementation is memory requirements. In the single kernel implementation, no additional memory is required to store volume integral contributions independently from surface integral contributions. As each thread computes one element's volume integral and one edge of that element's surface integral, it simply adds the result to a right-hand side variable.

Two main disadvantages of the single kernel implementation are the redundant computations of surface integrals and the loss of parallelization when compared to the multiple kernel implementation. An edge e_i sharing elements Ω_l and Ω_r will have the surface integral over this edge computed twice, as both threads t_l and t_r will compute the surface integral over e_i. Furthermore, as mentioned before, the surface integral calculations for each edge may be handled independently from the volume integral calculations. Therefore, a loss of parallelization results by grouping the surface integral calculations into the same kernel as the volume integral calculations.

In testing, we achieved nearly two times the performance using the multiple kernel implementation over the single kernel implementation. We suspect that this is mainly due to the loss of parallelization rather than the redundant computations of surface integrals, as computation time is rarely a bottleneck in GPU computing. We therefore use the multiple kernel technique in our implementation.

4.2 Numerical Quadrature

We use two-dimensional Gaussian quadrature rules [12] of appropriate order accuracy q_2 to approximate the volume integral

$$\int_{\Omega_0} \mathbf{F}(\mathbf{U}_i) \cdot (J_i^{-1} \nabla \phi_j) |\det J_i| \, d\mathbf{r} \approx \sum_{k=1}^{q_2} \mathbf{F}(\mathbf{U}_i(\mathbf{r}_k)) \cdot (J_i^{-1} \nabla \phi_j(\mathbf{r}_k)) |\det J_i| w_k \tag{4.4}$$

where $\mathbf{r}_k$ and w_k are the integration points and weights of the quadrature rule, respectively. Recall that $\mathbf{U}_i$ and ϕ_j are polynomials of degree at most p in our implementation; that is, $\nabla \phi_j \mathbf{U}_i$ is a polynomial of degree at most $2p - 1$. When the flux function $\mathbf{F}$ is linear, the integrating function is a polynomial of degree at most $2p - 1$, and so Gaussian quadrature rules of degree $q_2 = 2p-1$ should be used. Nonlinear flux functions require an extra degree of accuracy, setting $q_2 = 2p$. As we also implement nonlinear flux functions, we set $q_2 = 2p$ in this implementation.

We also use one-dimensional Gaussian quadrature rules of appropriate degree accuracy q_1 to approximate each component of the surface integral

$$\int_{s_q} \phi_j \mathbf{F}(\mathbf{U}_i) \cdot \mathbf{n}_i \, ds \approx \sum_{k=1}^{q_1} \phi_j(\mathbf{r}_k) \mathbf{F}(\mathbf{U}_i(\mathbf{r}_k)) \cdot \mathbf{n}_{i,l} w_k \tag{4.5}$$

for one-dimensional integration points translated to the correct side s_q of the canonical triangle $\mathbf{r}_k$ and weights w_k. Here, $\mathbf{U}_i$ and ϕ_j are both polynomials of degree at most p, and so Gaussian quadrature rules of degree $q_1 = 2p$ should be used for linear flux functions, and $q_1 = 2p+1$ for nonlinear flux functions. Once again, as we implement nonlinear flux functions, we set $q_1 = 2p+1$ in this implementation.

We precompute the evaluations of $\phi_j(\mathbf{x}_k)$ and $\nabla\phi_j(\mathbf{x}_k)$ at each of the two-dimensional integration points in Ω_0. We also precompute the evaluations of $\phi_j(\mathbf{x}_k)$ at the one-dimensional integration points mapped to $\partial\Omega_0$. These precomputed basis functions evaluations reduce computation cost immensely in (4.4) and (4.5). Indeed, whenever we evaluate $\mathbf{U}_i$ at an integration point $\mathbf{r}_k$, we compute

$$\mathbf{U}_i(\mathbf{r}_k) = \sum_{j=0}^{p} \mathbf{c}_{i,j} \phi_j(\mathbf{r}_k). \tag{4.6}$$

By precomputing the value of our basis functions at our integration points $\phi_j(\mathbf{r}_k)$, we may compute $\mathbf{U}_i$ at any integration point with only $n \times N_p$ multiplications and additions.

4.3 Algorithms

Algorithm 1 provides a high-level description of our implementation. Meshes are generated using GMSH [14], an open source finite element mesh generator. A Python script reads the mesh created by GMSH and generates a list of data structures representing the mappings between edges and elements. These data structures are then transfered to GPU memory. All precomputed data, such as the basis functions ϕ_j at their integration points, the inverse Jacobian matrices J_i^{-1} for each mesh element, and the normal vectors $\mathbf{n}_i$ for each element's edges, are stored in GPU memory. In practice, we compute and store the matrix $J_i^{-1}|\det J_i|$.

Algorithm 1 The DG Method on a GPU

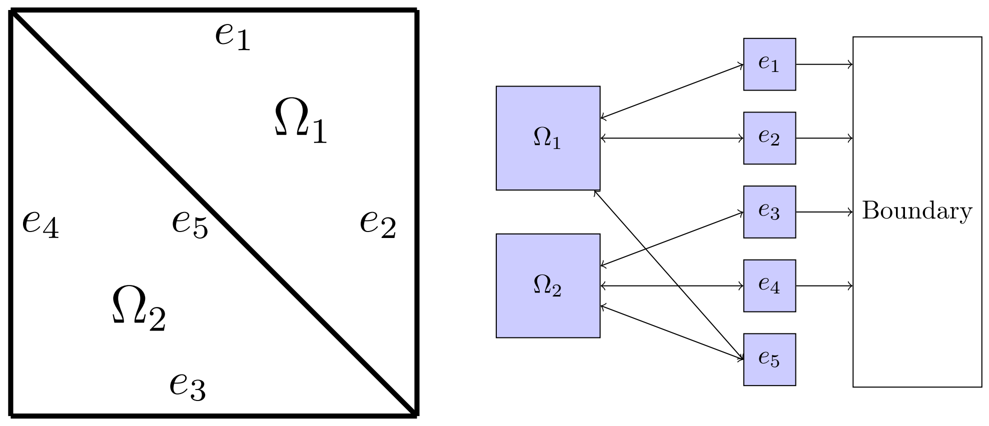
```
1. Read the mesh to CPU memory from a mesh file.
2. Transfer the data structures read from the mesh from CPU memory to GPU memory.
3. Precompute any reusable computations and store in GPU memory.
4. Compute the initial projection of $\mathbf{U}$ at $t = 0$ using initial conditions $\mathbf{u}_0$.
5. For each stage / step of a numerical ODE solver:
    (a) Calculate the CFL number.
    (b) Compute the surface integral for each edge.
    (c) Compute the volume integral for each element.
    (d) Combine the surface and volume integrals for each element in a numerical ODE
        solver.
6. Write the solution to file.
```

Using the initial conditions $\mathbf{u}_0$, we obtain the coefficients for the initial approximation $\mathbf{U}(\mathbf{x}, 0)$ by an L^2 projection on the finite element space. A numerical ODE solver approximates equation (2.59) using a stable timestep. To this end, we first compute the surface integral using one-dimensional numerical quadrature. Then, we compute the volume integral using two-dimensional numerical quadrature. These evaluations are stored separately and later recombined by a right-hand side evaluator. After a suitable number of timesteps, the solution is computed and written to file.

Our mesh consists of N_e edges e_k and N elements Ω_i. Each element is mapped to the canonical triangle Ω_0 with vertices at (0,0), (0,1) and (1,0) as described in Section 2.7. Our finite-element space $V(\Omega_0)$ consists of polynomials of degree at most p over Ω_0. The corresponding orthogonal basis elements used for $V(\Omega_0)$ are the orthogonal polynomials over Ω_0 described in Section 2.7, shown in Figure 2.7. For degree p polynomial approximation, N_p basis functions are required, defined in (2.57) Thus, for each element, we store $n \times N_p$ coefficients to represent the approximated solution over that element.

This implementation makes use of double precision floating point numbers whenever possible. Computing in double-precision is many times slower than computing in single precision on certain older versions of CUDA. In NVIDIA Fermi GPU architectures used

Figure 4.1: The mapping for a simple mesh.



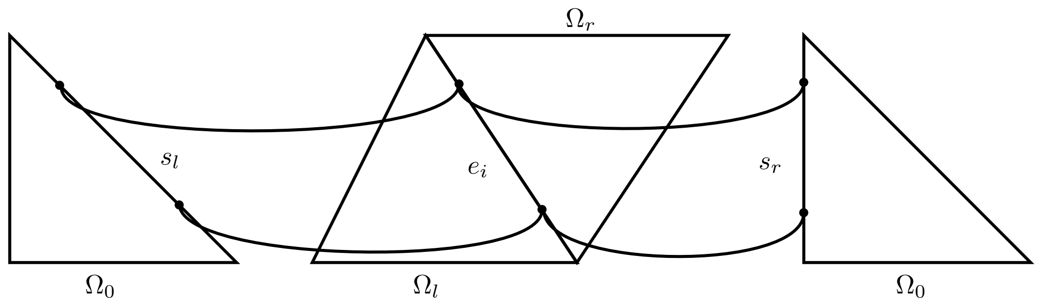
in this implementation, double precision computing is one quarter of the speed of single precision computing. See [31] for more information.

4.3.1 Mesh Generation

While one-dimensional meshes consist of only a collection of endpoints, two-dimensional meshes require significantly more data to represent the connectivity between elements. Each element is defined by its three vertices, which consequently define that element's three edges. In addition to storing these vertices, all elements must store three pointers to locate those edges. As every non-boundary edge connects two elements, these edges must be able to locate and differentiate between these elements. Edges along a boundary must be handled separately from non-boundary edges, and must be marked according to the type of boundary conditions assigned to them.

Figure 4.1 shows an example of a simple mesh and the associated mappings required for that mesh. Elements Ω_1 and Ω_2 map to their respective edges; edges $e_1, \ldots, e_5$ map back to their respective elements. Elements Ω_1 and Ω_2 share edge e_5. As such, edge e_5 must differentiate between elements Ω_1 and Ω_2. It will arbitrarily assign elements Ω_1 and Ω_2 as either left or right, e.g., Ω_1 is left and Ω_2 is right. In this case, the normal vector will point from element Ω_1 to elemnt Ω_2, as the normal vector always points from left to right by our convention. As edges e_1, e_2, e_3, and e_4 lie on the boundary of the domain, they store an identifier describing the type of boundary conditions assigned to them instead of a pointer to a right element.

Figure 4.2: The integration points on Ω_0 differ for edge e_i between elements Ω_l and Ω_r



The two variables `left_elem` and `right_elem` store the mapping from each edge to its left and right elements. In the case of Figure 4.1, edge e_5 may have `left_elem[4]` $= 0$ and `right_elem[4]` $= 1$ to indicate that element Ω_1 is left and element Ω_2 is right.

The elements store pointers to their edges in `elem_s1`, `elem_s2`, and `elem_s3`. As each element maps its three edges to the canonical triangle independently of how other elements map their edges, the shared edge e_5 may be mapped to different sides of the canonical triangle by elements Ω_1 and Ω_2. Each edge must then store a parameter indicating which side of the canonical triangle its left and right elements map it to. We call these `left_side_number` and `right_side_number` and store them in global memory.

Figure 4.2 demonstrates this behavior for any general Ω_l and Ω_r sharing an edge e_i. In this case, the one-dimensional integration points along that edge will differ for elements Ω_r and Ω_l.

Mesh generation is computed using both GMSH and a Python script. GMSH converts a geometry (.geo) file into a mesh (.msh) file containing an unstructured triangular mesh [14]. The resulting .msh file is read by the Python script and converted into a collection of edge to element and element to edge mappings. Mesh generation and connectivity is performed in serial and need only be performed once for each mesh created by GMSH.

4.3.2 Surface Integration Kernel

We first consider the evaluation of the surface integral (4.5). Suppose elements Ω_l and Ω_r share the edge e_i. A thread t_i will be assigned to this edge e_i to compute the surface integral on that edge using one-dimensional numerical quadrature rules, described in Section 4.2.

This computation is demonstrated from a thread perspective in Figure 4.3. Each thread reads the coefficients for its left and right elements $\mathbf{c}_{l,j}$ and $\mathbf{c}_{r,j}$ for $j = 1, \ldots, N_p$. The thread computes $\mathbf{U}_l$ and $\mathbf{U}_r$ at the integration points $\mathbf{r}_k$. If an edge e_i lies on a domain boundary, a ghost state $\mathbf{U}_g$ is created and assigned to $\mathbf{U}_r$, as described in Section 4.4. Boundary edges are marked with a negative index for a pointer to their right element to differentiate them from non-boundary elements. Determining if an edge is a boundary edge or not is as simple as looking at whether or not the pointer in `right_elem` is negative or positive.

Next, using $\mathbf{U}_l$ and $\mathbf{U}_r$ evaluated at their integration points, thread t_i computes the numerical flux $\mathbf{F}_\mathbf{n}(\mathbf{U}_l, \mathbf{U}_r)$. The final component to the numerical quadrature rule in (4.5) is the evaluated basis function. We multiply the computed values by the normal vector $\mathbf{n}_{i,q}$ and $l_{i,q}$, choosing the appropriate q for this edge and taking note that the normal vector always points from Ω_l to Ω_r. Finally, we multiply the result by both a basis function ϕ_j evaluated at the one-dimensional integration point $\mathbf{r}_k$ and a weight w_k.

We enforce a strictly positive Jacobian determinant in order to ensure that each element Ω_i is mapped to Ω_0 with the same orientation. Then the order of the integration points on that edge are reversed between elements Ω_l and Ω_r, shown in Figure 4.2

When the one-dimensional integration points differ, so too will the basis functions evaluated at those points. As such, the surface integral contributions for edge e_i may not be the same for each element. Therefore, we must store separate surface integral contributions for the left and right elements. Fortunately, the most expensive computation, $\mathbf{F}(\mathbf{U}_l, \mathbf{U}_r)$ may be reused in this computation. We store the surface integral contribution for Ω_l in `rhs_surface_left` and the surface integral contribution for Ω_r in `rhs_surface_right`.

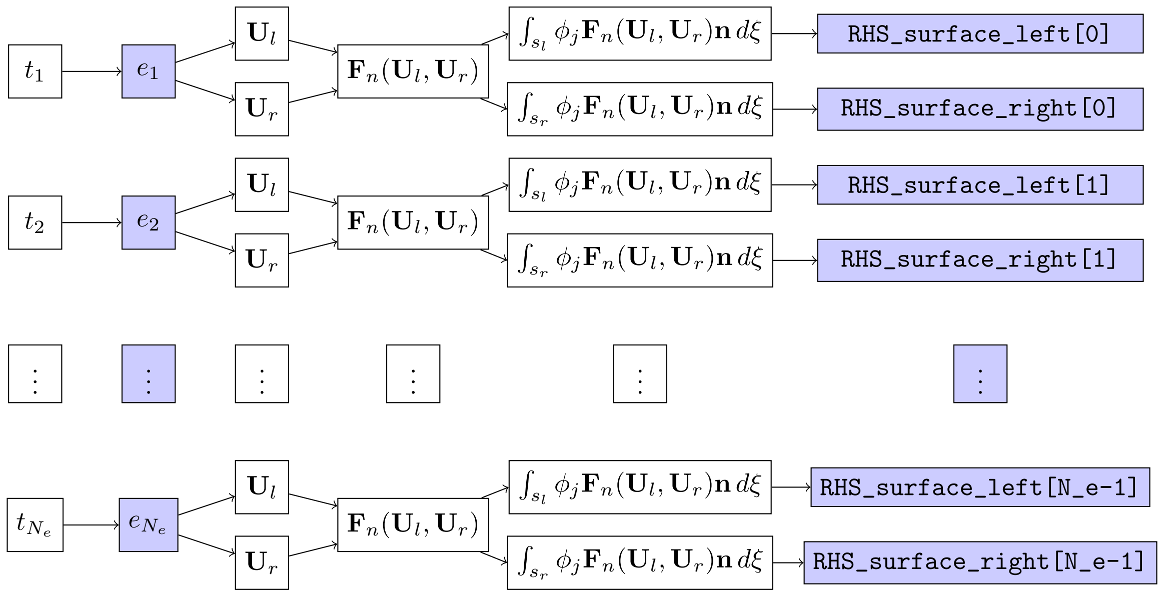
Each thread must read $2 \times N_p \times n$ coefficients to compute the approximation over Ω_l and Ω_r. This local storage requirement proves restrictive, as each SM on the GPU has access to only a limited supply of register memory. In performance tests, we found that using local memory and thereby increasing thread occupancy provided a performance boost of nearly thirteen times over the method of exclusively using register memory to hold these coefficients.

When an element Ω_i recombines its three separate edges' surface integral contributions as per (4.3), the element must know whether an edge considers it a left or a right element. If an edge considers Ω_i a left element, then the correct location for this edge's surface integral contribution is located in `rhs_surface_left`. Otherwise, the edge's surface integral contribution is located in `rhs_surface_right`.

Below we display the `eval_surface` kernel in full.

```
__global__ void eval_surface(
  double *C,                 // coefficients
  double *rhs_surface_left, // storage for left element, initialized to 0
  double *rhs_surface_right, // storage for right element, initialized to 0
```

Figure 4.3: Thread structure for the surface integral kernel



t_1
e_1
$\mathbf{U}_l$
$\mathbf{U}_r$
$\mathbf{F}_n(\mathbf{U}_l, \mathbf{U}_r)$
$\int_{s_l} \phi_j \mathbf{F}_n(\mathbf{U}_l, \mathbf{U}_r)\mathbf{n}\, d\xi$
$\int_{s_r} \phi_j \mathbf{F}_n(\mathbf{U}_l, \mathbf{U}_r)\mathbf{n}\, d\xi$
RHS_surface_left[0]
RHS_surface_right[0]
t_2
e_2
RHS_surface_left[1]
RHS_surface_right[1]
t_{N_e}
e_{N_e}
RHS_surface_left[N_e-1]
RHS_surface_right[N_e-1]

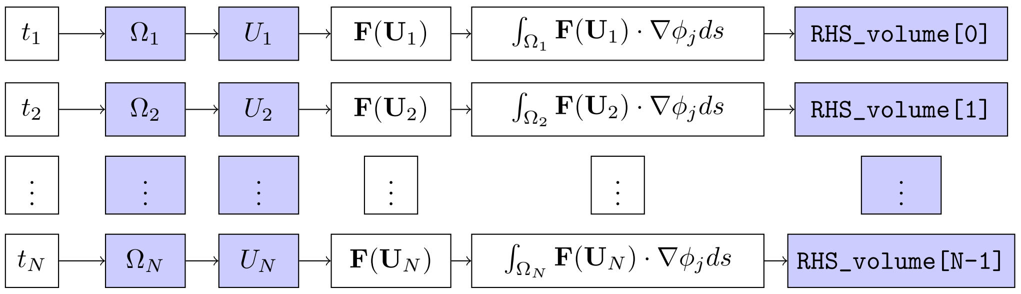
```
double *side_length,      // side lengths
double *V,                // element verticies
int *left_elem_pointer,   // pointer to left element
int *right_elem_pointer,  // pointer to right element
int *left_side_mapping,   // side s_l of Omega_0 that Omega_l maps e_i to
int *right_side_mapping,  // side s_r of Omega_0 that Omega_r maps e_i to
double *Nx, double *Ny,   // normal vector for this edge
double t) {               // time

int idx = blockIdx.x * blockDim.x + threadIdx.x;

if (idx < num_sides) {
  int i, j, n;     // indecies for looping
  int stride;      // used for data access
  int left_side;   // side s_l Omega_l maps this edge to
  int right_side;  // side s_r Omega_r maps this edge to
  int left_elem;   // index to Omega_l
  int right_elem;  // index to Omega_r

  double left_basis,     // psi_j at left integration point
  double right_basis;    // psi_j at right integration point
  double U_left[N_MAX];  // stores evaluation of U_l
  double U_right[N_MAX]; // stores evaluation of U_r
```

```
    double F_n[4];       // stores flux evaluation F_n
    double result_left;  // evaluation of Omega_l's result
    double result_right; // evaluation of Omega_r's result
    double nx, ny;       // the normal vector

    // locally stored coefficients
    double C_left[N_MAX * P_MAX];
    double C_right[N_MAX * P_MAX];

    // read coefficients to local memory
    for (i = 0; i < n_p; i++) {
        for (n = 0; n < N; n++) {

          // compute data access pattern for read
          stride = num_elem * n_p * n + i * num_elem;

          // read to local memory
          C_left[n*n_p + i] = C[stride + left_idx];
        }
    }

    // read right coefficients if this is not a boundary
    if (right_idx >= 0) {
        for (n = 0; n < N; n++) {
          // compute data access pattern for read
          stride = num_elem * n_p * n + i * num_elem;

          // read to local memory
          C_right[n*n_p + i] = C[stride + right_idx];
        }
    }

    // read index of Omega_l and Omega_r
    left_elem = left_elem_pointer[idx];
    right_elem = right_elem_pointer[idx];

    // read which s_l and s_r this edge is mapped to
    left_side = left_side_mapping[idx];
    right_side = right_side_mapping[idx];

    // read the normals for this edge
    nx = Nx[idx];
    ny = Ny[idx];

    // loop through each integration point
    for (j = 0; j < n_quad1d; j++) {

      // calculate U_left and U_right at integration point j
      eval_left_right(C_left, C_right, U_left, U_right,
        nx, ny, V, j, left_side, right_side,
        left_idx, right_idx, t);

      // compute F_n(U_left, U_right)
      riemann_solver(F_n, U_left, U_right, V, t, nx, ny, j, left_side);

      // multiply by phi_i at this integration point
      for (i = 0; i < n_p; i++) {

        // evaluate psi_j at the left integration point
        left_basis = basis_side[left_side * n_p * n_quad1d + i * n_quad1d + j];

        // evaluate psi_j at the right integration point
        right_basis = basis_side[right_side * n_p * n_quad1d + i * n_quad1d + n_quad1d -
             1 - j]);

        // loop through each variable U^n in the system
        for (n = 0; n < N; n++) {

          // compute data access pattern for coallesced write
          stride = num_sides * n_p * n + i * num_sides;

          // compute the result at this integration point
          result_left = -len/2*w_oned[j]*F_n[n]*left_basis;
          result_right = len/2*w_oned[j]*F_n[n]*right_basis;

          // write the result
          rhs_surface_left [stride + idx] += result_left;
          rhs_surface_right[stride + idx] += result_right;
        }
      }
    }
  }
}
```

4.3.3 Volume Integration Kernel

We now consider the evaluation of the volume integral (4.4). Each thread t_i in the volume integral kernel computes the volume integral contribution for an element Ω_i using numerical

Figure 4.4: Thread structure for the volume integral kernel


t_1
Ω_1
U_1
$\mathbf{F}(\mathbf{U}_1)$
$\int_{\Omega_1} \mathbf{F}(\mathbf{U}_1) \cdot \nabla\phi_j ds$
RHS_volume[0]
t_2
Ω_2
U_2
$\mathbf{F}(\mathbf{U}_2)$
$\int_{\Omega_2} \mathbf{F}(\mathbf{U}_2) \cdot \nabla\phi_j ds$
RHS_volume[1]
t_N
Ω_N
U_N
$\mathbf{F}(\mathbf{U}_N)$
$\int_{\Omega_N} \mathbf{F}(\mathbf{U}_N) \cdot \nabla\phi_j ds$
RHS_volume[N-1]

quadrature rules of an appropriate degree, as described in Section 4.2. Unlike the surface integral computations, the volume integral computation is element local. As a result, each thread t_i only reads the coefficients $\mathbf{c}_{i,j}$, $j = 1, \dots, N_p$ along with the inverse Jacobian matrix J_i^{-1} for element Ω_i.

Figure 4.4 shows the volume integral computations in this kernel from a thread perspective. Thread t_i evaluates $\mathbf{U}_i$ and the flux $\mathbf{F}(\mathbf{U}_i)$ at each two-dimensional integration point. The inverse Jacobian matrix J_i^{-1} is multiplied by the basis function gradient $\nabla\phi_j$ for each $j = 1, \dots, N_p$ evaluated at this integration point along with the appropriate quadrature weight w_k. The result is added to a right-hand side storage variable `rhs_volume`.

The surface integral and volume integral contributions may be computed independently from each other, meaning that this kernel and the surface integral kernel may run concurrently, a feature available in later versions of CUDA. This technique improves performance for particularly small meshes by increasing device saturation.

As each thread must read $N_p \times n$ coefficients, this storage requirement again proves restrictive for the GPU SMs. Letting threads use local memory increases thread occupancy, and greatly increases performance over requesting only register memory in our implementation.

Below we describe the `eval_volume` kernel in full.

```
__global__ void eval_volume(
  double *C,           // coefficients
  double *rhs_volume, // storage for result, intialized to 0
  double *J,           // inverse jacobian matrix
  double *V,           // verticies
  double t) {          // time

  int idx = blockIdx.x * blockDim.x + threadIdx.x;

```

```
if (idx < num_elem) {
  int i, j, k, n; // indecies for looping
  int stride;     // used for coallesced data access

  double U[N_MAX];      // evaluations of U
  double flux_x[N_MAX]; // flux x vector
  double flux_y[N_MAX]; // flux y vector
  double j[4];          // inverse jacobian matrix
  double result_x;      // invsere jacobian times grad(phi_j)_x
  double result_y;      // invsere jacobian times grad(phi_j)_y

  // locally stored coefficients
  double local_C[N_MAX * P_MAX];

  // read coefficients to local memory
  for (i = 0; i < n_p; i++) {
      for (n = 0; n < N; n++) {

        // compute data access pattern for coallesced read
        stride = num_elem * n_p * n + i * num_elem;

        local_C[n*n_p + i] = C[stride + idx];
      }
  }

  // read jacobian j = [y_s, -y_r, -x_s, x_r]
  for (i = 0; i < 4; i++) {
      j[i] = J[i*idx];
  }

  // loop through each integration point
  for (j = 0; j < n_quad; j++) {

    // initialize U to zero
      for (n = 0; n < N; n++) {
        U[n] = 0.;
    }

    // calculate U at the integration point
    for (k = 0; k < n_p; k++) {
      for (n = 0; n < N; n++) {

        // evaluate U at the integration point
        U[n] += local_C[n*n_p + i] * basis[n_quad * k + j];
      }
      }

      // evaluate the flux
      eval_flux(U, flux_x, flux_y, V, t, j, -1);

      // multiply by grad(phi_i) at this integration point
      for (i = 0; i < n_p; i++) {
        for (n = 0; n < N; n++) {

          // compute data access pattern for coallesced read
          stride = num_elem * n_p * n + i * num_elem;

          // compute the inverse jacobian times the basis gradient
          result_x = basis_grad_r[n_quad*i+j] * j[0]
                   + basis_grad_s[n_quad*i+j] * j[1];
          result_y = basis_grad_r[n_quad*i+j] * j[2]
                   + basis_grad_s[n_quad*i+j] * j[3];

          // compute the flux dotted by the result
          rhs_volume[stride + idx] += flux_x[n]*result_x + flux_y[n]*result_y;
        }
      }
    }
  }
}
```

4.3.4 Right-Hand Side Evaluator Kernel

The right-hand side evaluator kernel combines data from the three temporary storage variables `rhs_surface_left`, `rhs_surface_right`, and `rhs_volume` for each element to compute the right-hand side of equation (2.59). Each thread t_i combines the contributions from the surface and volume integrals for coefficients $\mathbf{c}_{i,j}$, $j = 1, \ldots, N_p$. For each edge, the thread must determine if that edge considers element Ω_i a left or right element. The thread first reads the value of $\det J_i$. It adds the volume integral contribution results from `rhs_volume` for each j. Next, the thread must locate element Ω_i's three edges. If it considers this element a left element, it will add every j component of `rhs_surface_left` to $\mathbf{c}_{i,j}$. If, on the other hand, it considers this element a right element, it will add every j component of `rhs_surface_right` to $\mathbf{c}_{i,j}$.

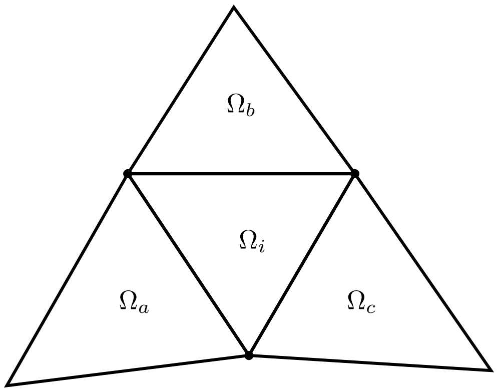
Serial implementations may simply add the computed surface integral and volume integral contributions to a single right-hand side variable as they are computed. This data access pattern would be impossible in a multiple kernel parallel implementation due to race conditions, that is, multiple threads attempting to write to the same location in memory simultaneously. During the surface integral computation, all three edges may attempt to add to the same data location at once and would corrupt memory.

Below we describe the `eval_rhs` kernel in full. While some warp divergence is inevitable in this kernel, by nesting the for loops inside of the boolean statements, we evaluate only three booleans per thread. Compared to performing the boolean evaluations inside of looping over each j, this optimization yields a tremendous increase in performance, especially for large j in higher-order polynomial approximation. This optimization greatly increased performance.

```
__global__ void eval_rhs(
  double *c,                  // coefficients initialized to 0
  double *rhs_volume,         // the volume integral contributions
  double *rhs_surface_left, // left surface integral contributions
  double *rhs_surface_right, // right surface integral contributions
  int *elem_s1,    // mapping to element's edge 1
  int *elem_s2,    // mapping to element's edge 2
  int *elem_s3,    // mapping to element's edge 3
  int *left_elem, // used to determine left or right
  double *detJ,    // jacobian determinant
  double dt) {     // timestep size

  int idx = blockDim.x * blockIdx.x + threadIdx.x;

  if (idx < num_elem) {
    int i, j, n;      // indecies for looping
    int S[3];         // store edge indecies
    int stride;       // used for coalesced data access
    int stride_edge; // used for coalesced data access

    // read jacobian determinant
    j = detJ[idx];

    // get the edge indecies
    S[0] = elem_s1[idx];
    S[1] = elem_s2[idx];
    S[2] = elem_s3[idx];

    // add volume integral
    for (i = 0; i < n_p; i++) {
      for (n = 0; n < N; n++) {

        // compute data access pattern for coallesced read
        stride = num_elem * n_p * n + i * num_elem;

        // add the volume integral contributions
        c[stride + idx] += dt / j * rhs_volume[stride + idx];
      }
    }

    // for each edge, add either left or right surface integral
    for (j = 0; j < 3; j++) {

      // if this is a left element, use rhs_surface_left
      if (idx == left_elem[S[j]]) {

        // add the surface integral
        for (i = 0; i < n_p; i++) {
          for (n = 0; n < N; n++) {

            // compute data access pattern for coallesced read
            stride = num_elem * n_p * n + i * num_elem;

            // edge access pattern
            stride_edge = num_sides * n_p * n + i * num_sides;

            // add the surface integral contribution
            c[stride + idx] += dt/j*rhs_surface_left[stride_edge + S[j]];
          }
        }

      // if this is a right element, use rhs_surface_right
      } else {

        // add the surface integral
        for (i = 0; i < n_p; i++) {
          for (n = 0; n < N; n++) {

            // compute data access pattern for coallesced read
            stride = num_elem * n_p * n + i * num_elem;

            // edge access pattern
            stride_side = num_sides * n_p * n + i * num_sides;

            // add the surface integral contribution
            c[stride + idx] += dt/j*rhs_surface_right[stride_side + S[j]];
          }
        }
      }
    }
  }
}
```

4.3.5 Limiters

Figure 4.5: To limit the solution over Ω_i, we evaluate the centroid values of surrounding elements Ω_a, Ω_b, and Ω_c



Discontinuities such as shocks are difficult to approximate numerically. Spurious oscillations, referred to as Gibbs phenomenon [17], occur near the discontinuity. These oscillations can produce nonphysical values in the model and result in numerical instability. We employ slope limiters to remove these oscillations.

We implement the popular Barth-Jespersen limiter [26, 29] for linear $p = 1$ approximations. We aim to limit the maximum slope in the gradient of

$$U_i(\mathbf{r}) = \bar{U}_i + \alpha_i(\nabla U_i) \cdot (\mathbf{r} - \mathbf{r}_c), \tag{4.7}$$

by selecting a limiting coefficient α_i. In (4.7), $\bar{U}_i$ is the average value of U_i over Ω_i and $\mathbf{r}_c$ is the centroid coordinate.

Suppose that element Ω_i is surrounded by elements Ω_a, Ω_b, and Ω_c, as in Figure 4.5. We choose α_i so that U_i introduces no new local extrema relative to these three surrounding elements. As such, this limiter uses a small stencil of only the three immediate surrounding elements.

We first evaluate U_i, U_a, U_b and U_c at their centroids. Define the maximum centroid value

$$U_i^{\max} = \max\left\{U_i(\mathbf{r}_c), U_a(\mathbf{r}_c), U_b(\mathbf{r}_c), U_c(\mathbf{r}_c)\right\} \tag{4.8}$$

and minimum centroid value

$$U_i^{\min} = \min\left\{U_i(\mathbf{r}_c), U_a(\mathbf{r}_c), U_b(\mathbf{r}_c), U_c(\mathbf{r}_c)\right\}. \tag{4.9}$$

We want to ensure that the maximum value of U_i does not exceed $U_i^{\max}$ and the minimum value of U_i does not fall below $U_i^{\min}$. Thus, at each two-dimensional integration point $\mathbf{r}_k$, we determine

$$\alpha_{i,k} = \begin{cases} \min\left\{1, \frac{U_i^{\max}-\bar{U}_i}{U_i(\mathbf{r}_k)-\bar{U}_i}\right\}, & U_i(\mathbf{r}_k) - \bar{U}_i > 0 \\ \min\left\{1, \frac{U_i^{\min}-\bar{U}_i}{U_i(\mathbf{r}_k)-\bar{U}_i}\right\}, & U_i(\mathbf{r}_k) - \bar{U}_i < 0 \\ 1, & U_i(\mathbf{r}_k) - \bar{U}_i = 0 \end{cases} . \tag{4.10}$$

Choose

$$\alpha_i = \min_k \left\{\alpha_{i,k}\right\}, \tag{4.11}$$

to limit the slope coefficients $\mathbf{c}_{i,1}$ and $\mathbf{c}_{i,2}$ for $p = 1$. For systems of equations, we choose a separate α_i for each variable m in the system.

Our implementation of this limiter operates element-wise. Each thread t_i limits the slopes for a single element Ω_i. Thread t_i first computes $U_i^{\max}$ and $U_i^{\min}$ as in (4.8) and (4.9). Then, at each two-dimensional integration point $\mathbf{r}_k$, thread t_i computes $U_i(\mathbf{r}_k)$ in order to compute $\alpha_{i,k}$. The smallest of the $\alpha_{i,k}$ values becomes the limiting constant α_i. Finally, the coefficients modifying the slope is multiplied with this α_i. This is repeated for each variable in the system.

Each evaluation of α_i requires a significant number of boolean comparisons. As such, unavoidable warp divergence certainly inhibits performance.

4.4 Numerical Boundary Conditions

Each edge on the boundary of the computational domain must be handled differently from non-boundary edges. As boundary edges have only a left element, Ω_l, they are assigned a negative index for their `right_elem` mapping. This negative index identifies the boundary type. To compute the surface integral along these boundaries, the Riemann solver must somehow compute $\mathbf{U}_r$. As no Ω_r is available to compute $\mathbf{U}_r$, a ghost state $\mathbf{U}_g$ is created to satisfy the boundary conditions and treated as $\mathbf{U}_r$.

The method used to compute the ghost state $\mathbf{U}_g$ depends on the boundary condition. Along inflow boundaries, the ghost state is simply assigned the values of the inflow conditions. Along outflow boundaries, the ghost state must be assigned an appropriate value which will not reflect waves back into the domain. Along reflecting boundaries, the ghost state must reflect velocities while leaving other conserved variables unchanged. As these boundary conditions are extremely problem dependent, they are handled individually for each problem.

We index our edges so that all boundary edges appear sorted and first in our edge list. This avoids warp divergence, as the edges with boundaries of the same type will be grouped in the same warp. Each thread in these warps will all assigned a similar ghost state, depending on the boundary condition.

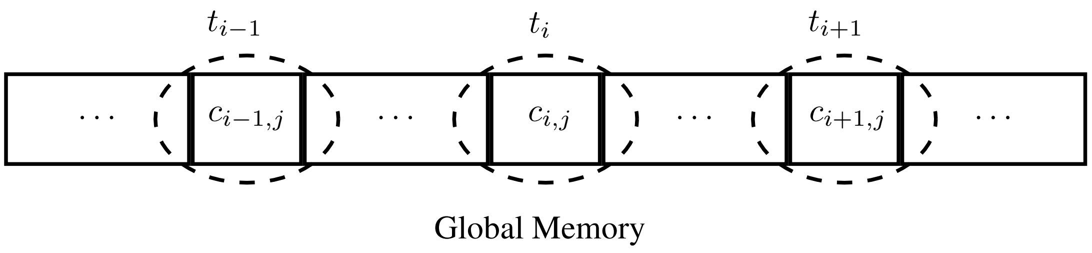
4.5 Data Coalescion

As high performance computing turns compute-bound problems into memory-bound problems, efficient memory management becomes essential. Below we describe the data ordering and memory access patterns in our implementation of the DG method.

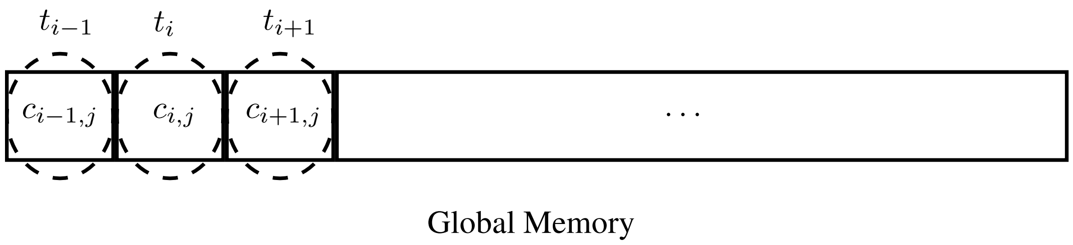
Data coalescion in this implementation is done wherever possible to reduce the number of read and write operations per thread. Unstructured meshes do not allow for predictable memory access patterns, making memory access patterns to allow data coalescion very difficult to implement. We store the coefficients $\mathbf{c}_{i,j}$ in a manner so that element-wise reads and writes are coalesced. That is, thread t_i may access coefficients $\mathbf{c}_{i,j}, j = 1, \ldots, N_p$ as a coalesced read. Unfortunately, this ordering does not allow coalesced reads in the surface integral kernel, as thread t_i needs to access coefficients $\mathbf{c}_{l,j}$ and $\mathbf{c}_{r,j}, j = 1, \ldots, N_p$.

To demonstrate this data ordering, we consider only a scalar problem ($n = 1$), as the coefficient data access pattern remains the same for higher n. Figure 4.6 illustrates two orderings of coefficients. The ordering demonstrated in 4.6a results in an uncoalesced memory access pattern. Here, threads t_{i-1}, t_i, and t_{i+1} simultaneously accesses memory locations far from one another. On the other hand, the ordering demonstrated in 4.6b results in a coalesced memory access pattern. In this case, threads t_{i-1}, t_i, and t_{i+1} access coefficients $c_{i-1,j}$, $c_{i,j}$, and $c_{i+1,j}$ which reside next to each other in memory. This data

Figure 4.6: Two orderings and memory access patterns for the coefficients



(a) An uncoalesced memory access pattern



(b) A coalesced memory access pattern

ordering may be done by flattening the coefficient matrix **C** in equation (2.61) column by column instead of row by row, as usually done in CPU implementations.

The two-dimensional array of coefficients **C** from equation (2.61) is flattened and stored in the device variable `C` in global memory. Accessing coefficient $c^m_{i,j}$ is done by

```
C[num_elem*n_p*m + j*num_elem + i]
```

where `num_elem` $= N$ and `n_p` $= N_p$.

4.6 Precomputing

Precomputing trades computing time for memory storage requirements and additional memory lookup time. On GPUs, this trade off is not always optimal, as computing time typically pales in comparison to memory lookup time. Furthermore, the scarcity of GPU video memory restricts storing superfluous amounts of precomputed data. With that said, we are able to take advantage of precomputing data in our implementation by using GPU constant memory.

By far, the most important precomputed values are the basis functions evaluated at both one- and two-dimensional integration points and the gradients of the basis functions evaluated at two-dimensional integration points. For the one-dimensional integration points, we require $N_p \times (2p+1)$ precomputed basis function evaluations along each side of the canonical triangle. Then, for each side s_i of the canonical triangle and corresponding integration points $\mathbf{r}_1, \ldots, \mathbf{r}_q$, $q_1 = 2p+1$, we must store the two-dimensional array

$$\texttt{basis\_si} = \begin{bmatrix} \phi_1(\mathbf{r}_1) & \phi_1(\mathbf{r}_2) & \cdots & \phi_1(\mathbf{r}_{q_1}) \\ \phi_2(\mathbf{r}_1) & \phi_2(\mathbf{r}_2) & \cdots & \phi_2(\mathbf{r}_{q_1}) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{N_p}(\mathbf{r}_1) & \phi_{N_p}(\mathbf{r}_2) & \cdots & \phi_{N_p}(\mathbf{r}_{q_1}) \end{bmatrix}, \tag{4.12}$$

as a flattened one-dimensional array

```
    basis_side = [basis_s1, basis_s2, basis_s3]
```

where `si` corresponds to side s_i of the canonical triangle. We then store `basis_side` in GPU constant memory. Each edge can look up its own preevaluated basis function by using its side mapping index as an offset. Doing this avoids using boolean evaluations to determine which side of the canonical triangle to use, which thereby avoids warp divergence. The gradients of the basis functions $\nabla\phi_j$ are precomputed at the two-dimensional integration points and stored in GPU constant memory in the arrays `basis_grad_r` and `basis_grad_s` in a similar fashion. GPU constant memory is easily able to accommodate these evaluations as even for large p, the number of basis function evaluations remains relatively small, and is independent of the mesh size.

To compute the m-th variable of $\mathbf{U}_l$ at the j-th one-dimensional integration point, we evaluate

```
U_l += C[num_elem*N_p*m + i*num_elem + l] * basis_side[left_side*N_p*n_quad1d + i*
    n_quad1d + j]
```

for each basis function index i. To compute $\mathbf{U}_r$, then, we must reverse the indexing of `basis_side`. That is, to compute $\mathbf{U}_r$ at the j-th one-dimensional integration point, we reverse the indexing of `basis_side`, evaluating

```
U_r += C[num_elem*N_p*m + i*num_elem + r] * basis_side[right_side*N_p*n_quad1d + i*
    n_quad1d + n_quad_1d - 1 - j]
```

for each basis function index i.

Whenever data grows with the size of the mesh, that data must be stored in global memory. We precompute the inverse Jacobian matrix J_i^{-1} and the determinant $|\det J_i|$

for each element, as both remain constant. Each edge stores a normal unit vector $\mathbf{n}_{i,q}$, $q = 1, 2, 3$ for the q-th edge of element i pointing from edge q's left to right element. In addition, the lengths $l_{i,q}$ of each edge are precomputed and stored. All precomputed data is sorted to allow coalesced reads.

The total memory required for computation depends on four factors. First, the size of the mesh determines the number of elements and edges. Second, the degree of the polynomial approximation determines the number of coefficients required to approximate the solution. Third, the size of the system of equations requires an extra vector of coefficients for each variable in the system. Finally, the ODE solver typically needs extra storage variables for intermediate steps or stages, which must be stored in global memory. Unfortunately, this current version of our implementation does not yet copy these extra storage variables from GPU memory back to CPU between timesteps when necessary, restricting the maximum mesh size we are able to handle.

Chapter 5

Computed Examples

Table 5.1: GPU Specifications

	NVIDIA GTX 580	NVIDIA GTX 460
Memory	3 GB GDDR5	1 GB GDDR5
CUDA Cores	512	336

We now present computed examples from this implementation of the DG method. Each example demonstrates the computation of a conservation law in two dimensions. Our simulations ran on two different graphics cards on separate workstations, detailed in Table 5.1. All tests were run on Ubuntu Linux using CUDA 4.0. Code for this implementation is located at `https://github.com/martyfuhry/DGCUDA`.

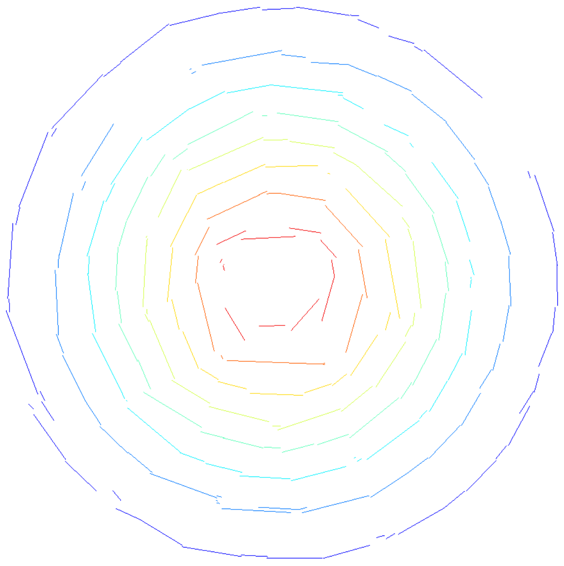
Mesh generation and postprocessing was done using GMSH and a Python script. All solutions displayed in GMSH were plotted using linear interpolation with no smoothing applied. The discontinuous nature of the numerical solution allows sharp jumps at isolines whenever solution values differ greatly between elements.

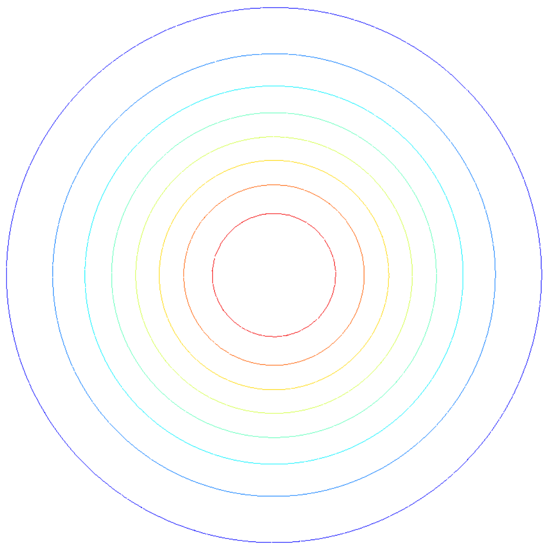
5.1 Linear Advection

We will first show a simple advection problem to verify the accuracy and convergence of our implementation. Linear advection is by far the simplest hyperbolic conservation law. It is a scalar law given by

$$\partial_t u + a(x,y)\partial_x u + b(x,y)\partial_y u = 0, \tag{5.1}$$

Figure 5.1: Isolines of the solution to the rotating hill problem with $p = 1$


(a) Mesh A


(b) Mesh D

where functions $a(x, y)$ and $b(x, y)$ determine the velocity of the flow of the initial profile u_0.

5.1.1 Rotating Hill

We consider the classical rotating hill advection [10] problem

$$\partial_t u - (2\pi y)\partial_x u + (2\pi x)\partial_y u = 0, \tag{5.2}$$

over a domain $\Omega = [-1, 1] \cup [-1, 1]$ with a single Gaussian pulse centered at (x_0, y_0) as our initial condition. The analytical solution

$$u(x, y, t) = \alpha \exp(-((x \cos(2\pi t) + y \sin(2\pi t) - x_0)^2 + \tag{5.3}$$
$$(-x \sin(2\pi t) + y \cos(2\pi t) - y_0)^2)/(2r^2)), \tag{5.4}$$

with constants α and r, rotates around the origin. We enforce this exact solution along the boundaries of our domain.

We simulate the rotating hill problem on four meshes, A, B, C, and D. The first mesh A contains 1,264 elements and each subsequent mesh was created through refinement by

Table 5.2: L^2 error and convergence rate r for levels of h- and p-refinement for the rotating hill test problem

	$p=1$		$p=2$		$p=3$		$p=4$	
Mesh	Error	r	Error	r	Error	r	Error	r
A	5.570E−2	-	3.704E−3	-	3.214E−4	-	2.236E−5	-
B	9.516E−3	2.549	3.284E−4	3.496	1.268E−5	4.664	6.452E−7	5.115
C	1.782E−3	2.417	3.648E−5	3.170	9.197E−7	3.785	2.214E−8	4.865
D	3.940E−4	2.177	4.438E−6	3.039	4.867E−8	4.240	6.325E−10	5.129

splitting each element into four smaller ones, quadrupling the mesh size with each successive refinement. We compute the simulation with $r = 0.15$ and $(x_0, y_0) = (0.2, 0)$ and report the results after one full rotation at $t = 1.0$. We use a Runge-Kutta fourth order time integrator and an appropriate stable CFL number (see Section 2.4). Using $p = 1, \cdots, 4$, we perform a convergence analysis using the L^2 norms for of h- and p-refinement in Table 5.2. The rate of convergence r nearly matches the theoretical $p + 1$ convergence rate expected for our scheme.

5.2 Maxwell's Equations

Maxwell's equations describe the interactions of electric and magnetic fields and are used to model problems in electrodynamics, optics, and circuits. These equations combine Faraday's law, Gauss's law, Ampére's circuital law, and Gauss's law for magnetism into a linear system describing wave movement through an electromagnetic field. We assume a transverse magnetic mode; that is, we assume that the electromagnetic field points orthogonally to the domain and that the magnetic field and wave propagation occur on the plane. Assuming zero initial current density, Maxwell's equations can be written in conservative form as the linear system of equations

$$\partial_t \begin{pmatrix} H_x \\ H_y \\ E_z \end{pmatrix} = \partial_x \begin{pmatrix} 0 \\ \frac{1}{\mu} E_z \\ \frac{1}{\epsilon} H_y \end{pmatrix} + \partial_y \begin{pmatrix} -\frac{1}{\mu} E_z \\ 0 \\ -\frac{1}{\epsilon} H_x \end{pmatrix} = \mathbf{0}, \tag{5.5}$$

where $\mathbf{H} = (H_x, H_y)$ is the magnetic field, E_z is the electric field, and the variables μ and ϵ represent the permittivity and the magnetic permeability of the medium. Assuming a zero initial current density forces our initial conditions to be initialized to zero. The current density may then be added in through use of a source term, which we will prescribe as inflow boundary conditions.

5.2.1 Circular Mirror

This simulation demonstrates the reflection of light off of a circular mirror [11]. Our computational domain shown in Figure 5.3a consists of a fully reflecting curved boundary on the right, reflecting top and bottom boundaries, and an emitter on the left boundary. The x domain for this problem ranges from 0 to 1.5, with the curved boundary beginning at $x = 0.5$. The y domain ranges from 0 to 2. We use a 68,258 element mesh with 102,695 edges.

The initial conditions are set to $\mathbf{u}_0 = \mathbf{0}$ throughout the domain. We prescribe an emitter along the left boundary, producing a single planar pulse wave

$$\begin{pmatrix} H_x \\ H_y \\ E_z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \sin(20\pi t) \end{pmatrix}, \tag{5.6}$$

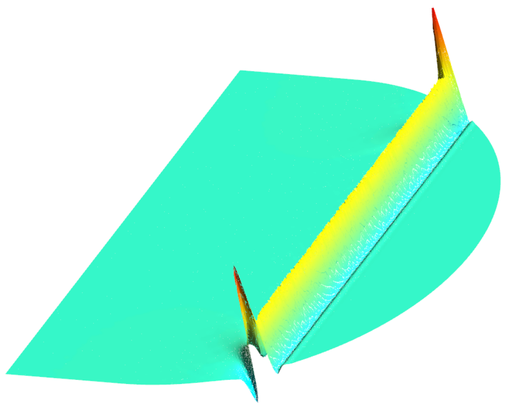
for one half-period of the sine wave and $\mathbf{0}$ afterwards. Figure 5.2 shows the pulse wave traveling along the mirror at four different times. The wave moves through the medium, reflecting off the mirror, and creating a caustic with a focal point near $(1, 1)$. The exact location of the focal point is difficult to determine, as small contributions from non-paraxial waves, that is, waves reflecting far from the center of the mirror slightly perturb the location of the exact focal point.

Also of interest is the intensity

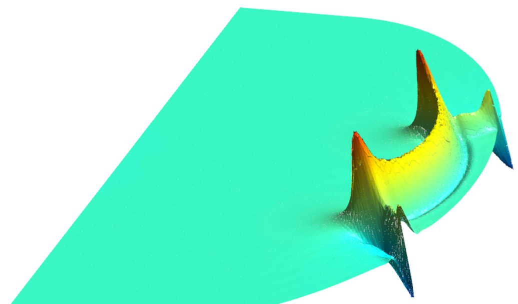
$$I = |E_z|^2, \tag{5.7}$$

averaged over time. Figure 5.3b shows the time averaged intensity of a single planar pulse with $p = 2$. We run the single pulse simulation for forty periods of the source wave (5.6) and average the intensity over time. This figure shows the focal point of the caustic in our simulation occurring at approximately $(x, y) = (1.06, 1.00)$. Indeed, our simulated focal point is not located at $(1, 1)$, as the simplified model would suggest. This suggests that our simulation is able to capture the many small contributions of non-paraxial waves which slightly modify the location of the focal point.

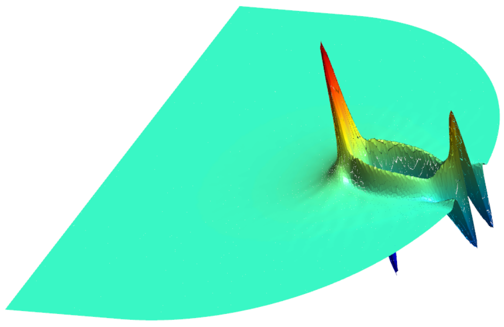
Figure 5.2: The electromagnetic field of the circular mirror test problem at various t



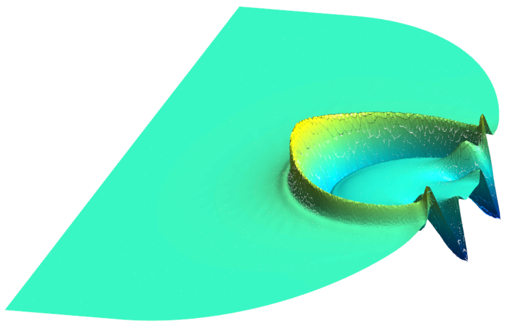
(a) $t = 1.05$



(b) $t = 1.65$

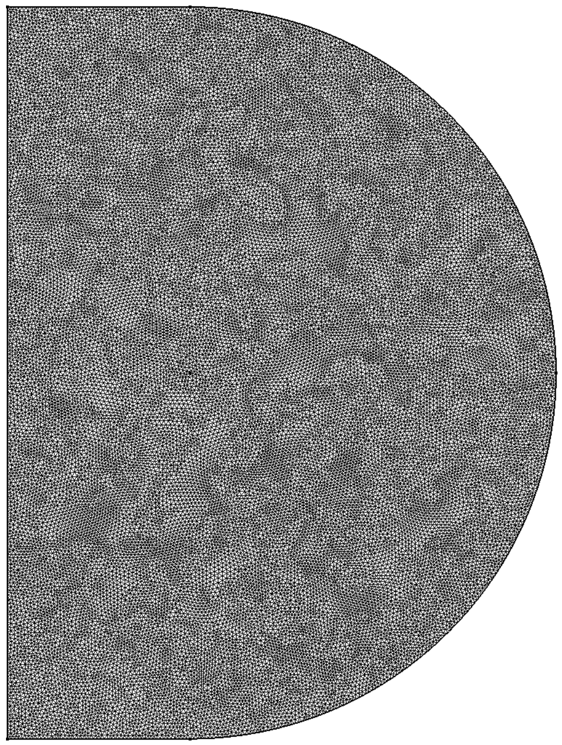


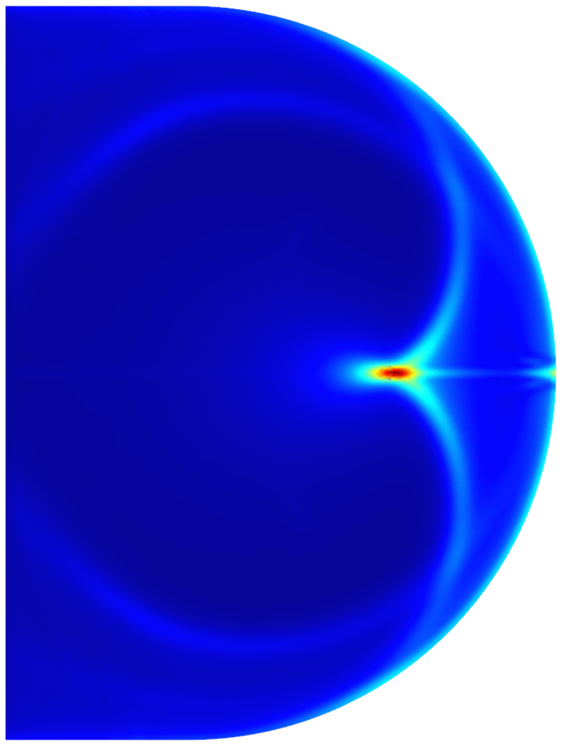
(c) $t = 2.00$



(d) $t = 2.20$

Figure 5.3: The mesh and time averaged intensity of a single pulse wave for the circular mirror test problem


(a) The mesh


(b) Time averaged intensity with $p = 2$

5.3 Shallow Water Equations

The shallow water equations describe the flow of a fluid when the size of the surface of the fluid is much larger than its height. They are given by

$$\partial_t \begin{pmatrix} h \\ uh \\ vh \end{pmatrix} + \partial_x \begin{pmatrix} uh \\ u^2h + \frac{1}{2}gh^2 \\ uvh \end{pmatrix} + \partial_y \begin{pmatrix} vh \\ uvh \\ v^2h + \frac{1}{2}gh^2 \end{pmatrix} = \mathbf{0}, \tag{5.8}$$

where h is the height of the fluid, and u and v are the velocity components. The constant g in (5.8) is the acceleration from gravity. We store the coefficients for the approximation to the conserved variables $\mathbf{u} = (h, uh, vh)$ and recompute the primitive variables u and v as needed.

To demonstrate the flexibility and ease of use of this implementation, we will provide a brief tutorial detailing how to implement the shallow water equations. The relevant files are found in the `shallowwater/` directory in the source code.

Suppose we wish to solve the shallow water equations with a Gaussian pulse as the initial conditions

$$h(x, y, 0) = h_0 + \alpha \exp(-((x - x_0)^2 + (y - y_0)^2)/(2r^2)). \tag{5.9}$$

In (5.9), the Gaussian is centered at (x_0, y_0) with amplitude α and radius r and shifted upwards by some positive initial height h_0. We set the initial velocities $u(x, y, 0) = v(x, y, 0) = 0$. We use a square computational domain $\Omega = [-1, 1] \cup [-1, 1]$ and reflecting boundary conditions.

Below, we describe the most important function in our implementation: the `eval_flux` function. Both kernels `eval_surface` and `eval_volume` call this device function at every integration point.

```
__device__ void eval_flux(
  double *U,      // the evaulation of U
  double *flux_x, // flux x vector to return
  double *flux_y, // flux y vector to return
  double *V,      // verticies
  double t,       // time
  int j,          // integration point
  int left_side) { // left side mapping

  // define the conserved variables
  double h, uh, vh;

  // read system variables
  h  = U[0];
  uh = U[1];
  vh = U[2];

  // compute flux_x vector
  flux_x[0] = uh;
  flux_x[1] = uh*uh/h + 0.5*G*h*h;
  flux_x[2] = uh*vh/h;

  // compute flux_y vector
  flux_y[0] = vh;
  flux_y[1] = uh*vh/h;
  flux_y[2] = vh*vh/h + 0.5*G*h*h;
}
```

This function takes, as arguments, the value of **U** at a given point and computes the flux vectors `flux_x` and `flux_y`. As the flux vectors do not depend on space or time in this case, we do not use the other arguments.

Below, we show our implementation of the local Lax-Friedrichs Riemann solver, described in Section 2.7.

```
__device__ void riemann_solver(
  double *F_n,        // numerical flux function to return
  double *U_left,     // U evaluated over Omega_l
  double *U_right,    // U evaluated over Omega_r
  double *V,          // verticies
  double t,           // time
  double nx, double ny, // the normal vectors
  int j,              // integration point
  int left_side) {    // left side mapping

  // for indexing
  int n;

  // define the flux vectors
  double flux_x_l[3], flux_x_r[3];
  double flux_y_l[3], flux_y_r[3];

  // calculate flux for U_left and U_right
  eval_flux(U_left, flux_x_l, flux_y_l);
  eval_flux(U_right, flux_x_r, flux_y_r);

  // compute the largest in magnitude eigenvalue
  double lambda = eval_lambda(U_left, U_right, nx, ny);

  // calculate the riemann problem at this integration point
  for (n = 0; n < N; n++) {
    // use the local lax-friedrichs riemann solver
    F_n[n] = 0.5 * ((flux_x_l[n] + flux_x_r[n]) * nx
                  +  (flux_y_l[n] + flux_y_r[n]) * ny
                  + lambda * (U_left[n] - U_right[n]));
  }
}
```

The device function `eval_lambda` computes the absolute value of the largest in magnitude eigenvalue of the Jacobian of **F**. In the shallow water equations, the eigenvalues are [1]

$$\lambda_1 = un_x + vn_y \tag{5.10}$$

$$\lambda_2 = un_x + vn_y - c \tag{5.11}$$

$$\lambda_3 = un_x + vn_y + c, \tag{5.12}$$

where $c = \sqrt{gh}$ is the speed of sound and $\mathbf{n} = (n_x, n_y)$ denotes the normal vector along the edge where the eigenvalues are calculated. Our implementation of this function is omitted.

We then include the initial conditions.

```
__device__ void U0(
  double *U, // to store the initial condition
  double x,  // x coordinate
  double y) { // y coordinate

  // define the center and radius of our pulse
  double x0, y0, r;
  x0 = 0.5;
  y0 = 0.5;
  r  = 0.1

  // create gaussian pulse
  U[0] = 10 + 5*exp(((x-x0)*(x-x0) + (y-y0)*(y-y0))/(2*r*r));
  U[1] = 0;
  U[2] = 0;
}
```

Our boundary conditions should reflect the velocities and keep the height h the same.

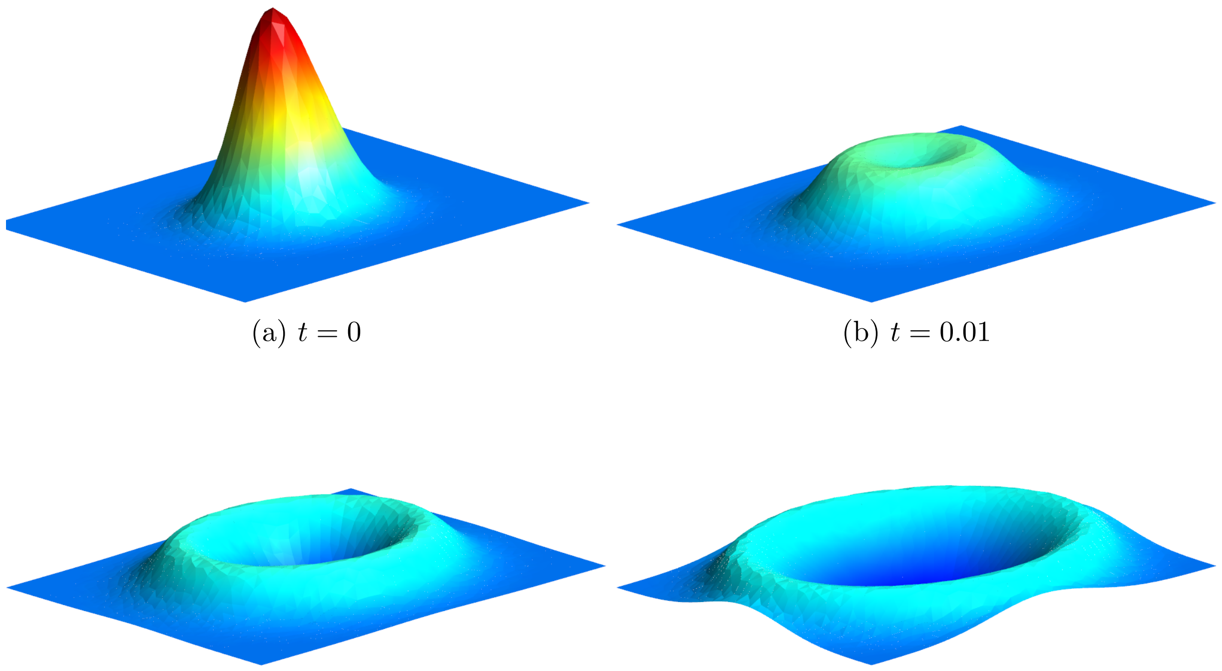
```
__device__ void U_reflection(
  double *U_left,       // the left evaluation
  double *U_right,      // the right evaluation to return
  double x, double y,   // coordinates
  double nx, double ny) { // normal vectors on this edge

  // for the dot product
  double dot;

  // set h to be the same
  U_right[0] = U_left[0];

  // and reflect the velocities
  dot = U_left[1] * nx + U_left[2] * ny;
  U_right[1] = U_left[1] - 2*dot*nx;
  U_right[2] = U_left[2] - 2*dot*ny;
}
```

We compile and run this example using $p = 2$ polynomial approximation to time $t = 0.01$ from the following command.

```
$ ./dgshallowtest -T 0.01 -p 2 test/mesh/testmesh.pmsh
```

The CPU will read the mesh file `testmesh.pmsh` and send the data structures and any precomputed variables to the GPU. The initial projection will be made using the `U0` device function. The time integrator will then compute the next time step using a stable CFL condition until time $t = 0.01$. The resulting solution approximations are turned into a .msh file to be read by GMSH. Figure 5.4 shows a linear interpolation of the approximation at various times t for a fairly coarse mesh of 2,984 elements with reflecting boundary conditions.

Figure 5.4: The shallow water equations at various t



(a) $t = 0$ (b) $t = 0.01$

(c) $t = 0.02$ (d) $t = 0.03$

5.4 Euler Equations

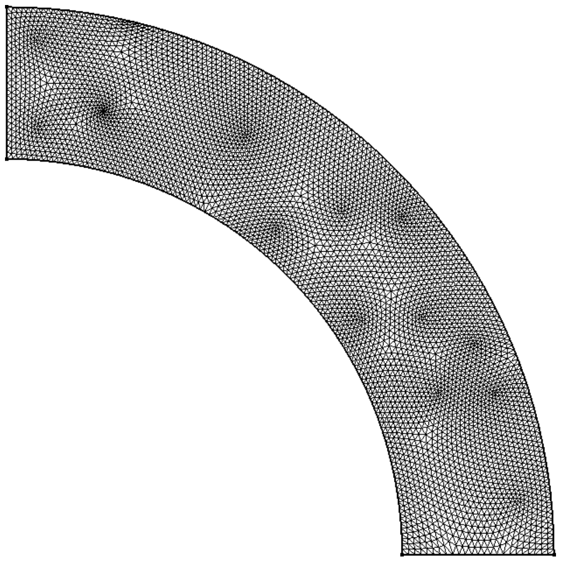
The Euler equations describe the flow of an inviscid, isotropic, compressible fluid. These nonlinear equations are derived from the physical conservations of mass, momentum and energy. In two dimensions, they are given by

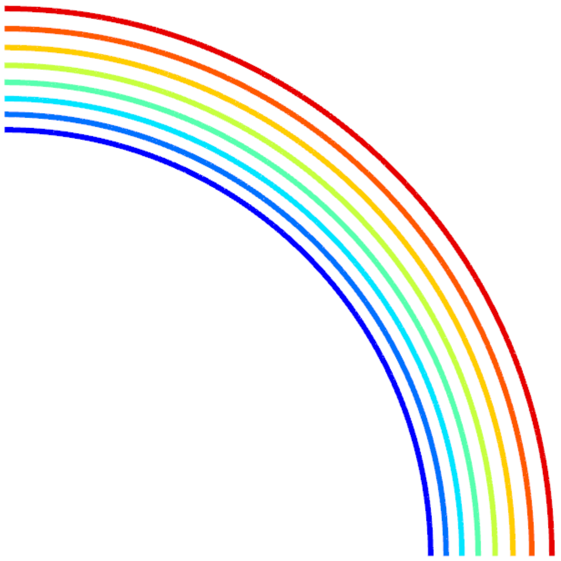
$$\partial_t \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ E \end{pmatrix} + \partial_x \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho u v \\ u(E+p) \end{pmatrix} + \partial_y \begin{pmatrix} \rho v \\ \rho u v \\ \rho v^2 + p \\ v(E+p) \end{pmatrix} = \mathbf{0}, \tag{5.13}$$

where ρ is the density, u and v are the velocity components, and E is the energy. The variable p in equation (5.13) is the pressure given by an equation of state, which we choose to be

$$p = (\gamma - 1)\left(E - \frac{\rho||\mathbf{v}||^2}{2}\right), \tag{5.14}$$

Figure 5.5: The mesh and approximated solution of the supersonic vortex test problem at a numerical steady state


(a) Mesh D


(b) Density isolines over mesh D with $p = 4$. Isolines along the outer surface represent higher densities than isolines along the inner surface.

for an adiabatic constant γ and velocity vector $\mathbf{v} = (u, v)$. For air, we take $\gamma = 1.4$.

We store the coefficients for the approximation to the conserved variables for $\mathbf{u} = (\rho, \rho u, \rho v, E)$ and recompute the primitive variables u, v, and p as needed. Computations in this section are performed until a numerical steady state is reached when the maximum absolute change in coefficient values

$$\max\left\{|\mathbf{c}_{i,j}^{n+1} - \mathbf{c}_{i,j}^{n}|\right\} \leq \texttt{TOL} \tag{5.15}$$

is less than an appropriate tolerance $\texttt{TOL}$.

Table 5.3: L^2 error in density and convergence rate r for levels of h- and p-refinement for the supersonic vortex test problem

	$p = 1$		$p = 2$		$p = 3$		$p = 4$	
Mesh	Error	r	Error	r	Error	r	Error	r
A	4.934E−3	-	3.708E−4	-	8.695E−6	-	4.719E−7	-
B	1.226E−3	2.009	6.003E−5	2.627	5.598E−7	3.957	1.887E−8	4.644
C	3.267E−4	1.908	8.077E−6	2.894	3.237E−8	4.645	6.925E−10	4.766
D	8.695E−5	1.910	1.043E−6	2.953	1.904E−9	4.086	2.189E−11	4.983

5.4.1 Supersonic Vortex

The supersonic vortex test problem models supersonic fluid flow through a curved quarter cylinder tube, like that shown in Figure 5.5a. This problem has a known smooth analytical solution, which we prescribe as the initial conditions. We use curved reflecting boundary conditions, detailed in [25], along the curved edges and inflow and outflow boundary conditions at the inlet and outlet. We run the simulation until numerical convergence occurs with tolerance `TOL` $= 10^{-14}$.

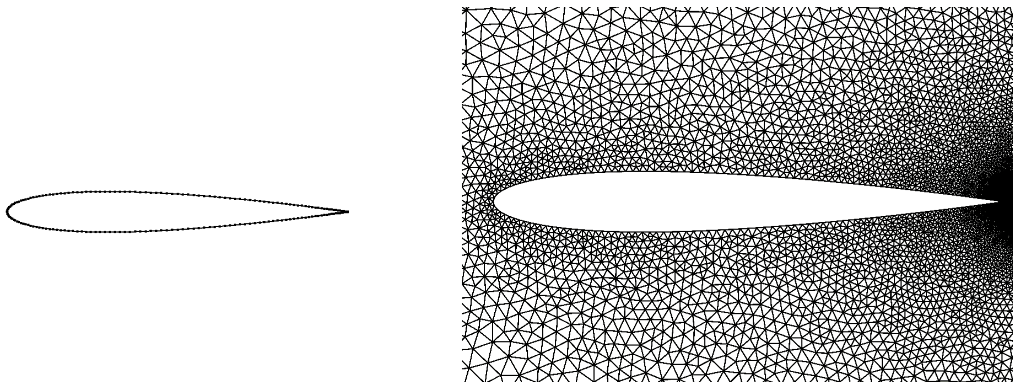
To verify the convergence of this implementation for nonlinear problems, we perform a similar convergence analysis as in Section 5.1.1. Our test meshes range from A to D with mesh A containing 180 elements. Meshes B through D were created by successive refinement of the previous mesh by splitting each triangle into four triangles, quadrupling the total number of elements with each refinement.

The L^2 error between the numerical steady state and analytical steady state are compared for each combination of $h-$ and p-refinement in Table 5.3. These convergence rates match theoretical convergence rates, verifying the accuracy of our implementation for nonlinear problems. Figure 5.5b shows the density isolines of the approximation at numerical convergence of our most accurate test. Our results look both quantitatively and qualitatively similar to those found elsewhere; e.g., [25].

5.4.2 Flow Around an Airfoil

We now present an example of air flow around an airfoil. This test simulates the air flow around a two-dimensional cross section of an airplane wing or a helicopter blade. One goal of airfoil design is to produce lift by modifying either the shape of the airfoil or the angle of the flow, known as the angle of attack. We therefore run several simulations with varying angles of attack and flow speeds.

Figure 5.6: The NACA0012 airfoil and a corresponding mesh



(a) NACA0012 airfoil (b) Zoomed view of the mesh

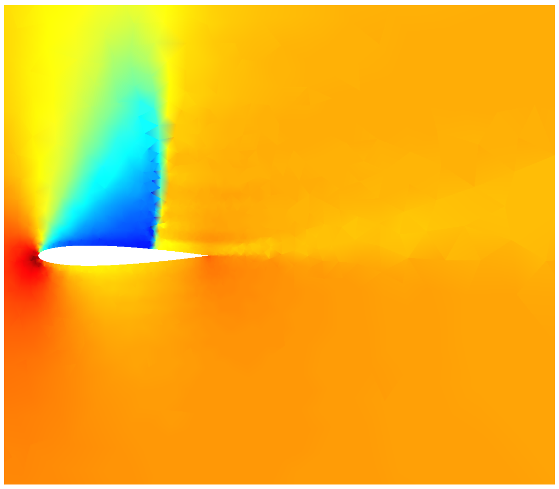
We consider free flow around the NACA0012 airfoil design [34] shown in Figure 5.6a at various angles of attack α and Mach numbers M. We set the free flow speed of sound and pressure to be equal to one. This amounts to setting initial conditions to

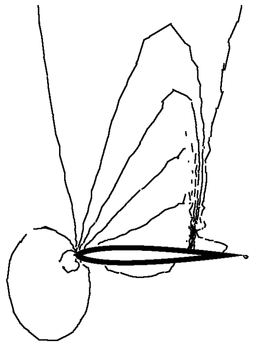
$$\begin{pmatrix} \rho \\ \rho u \\ \rho v \\ E \end{pmatrix} = \begin{pmatrix} \gamma \\ \rho M \cos \alpha \\ \rho M \sin \alpha \\ \frac{1}{2}\rho M^2 + \frac{1}{\gamma - 1} \end{pmatrix}, \tag{5.16}$$

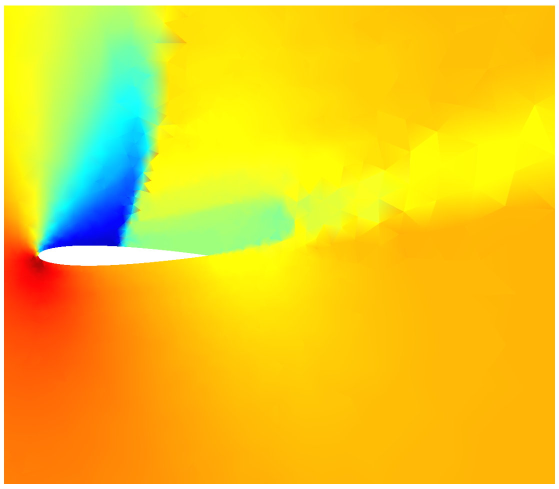
and identical free flow boundary conditions along the outer boundaries of our computation domain Ω. Our mesh, shown zoomed up in Figure 5.6b, contains 5,846 elements.

Figures 5.7 and 5.8 show the density and density isolines at a numerical steady state, computed with a tolerance of $\texttt{TOL} = 10^{-11}$, of four simulations using different values of Mach numbers M and angles of attack α. For all simulations, we use $p = 1$ with the Barth-Jespersen limiter, described in Section 4.3.5. Of interest is the classical example, shown in Figure 5.7a, with Mach number $M = 0.78$ and angle of attack $\alpha = 5°$. A shock wave forms along the top of this airfoil, which we are able to better capture by using a limiter. We also show supersonic flow around the airfoil in Figure 5.8c by using free flow with a Mach number $M = 4$, and a $\alpha = 0°$ angle of attack, creating a strong shock wave at the head of the airfoil.

Figure 5.7: Steady states of the airfoil test problem with $p = 1$ with different Mach numbers and angles of attack


(a) Density with $M = 0.78$, $\alpha = 5°$


(b) Density isolines with $M = 0.78$, $\alpha = 5°$


(c) Density with $M = 0.78$, $\alpha = 15°$

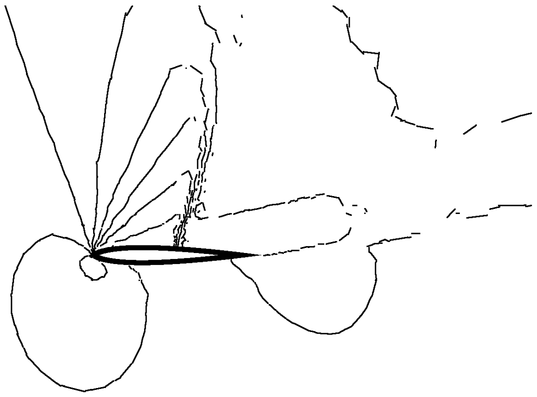
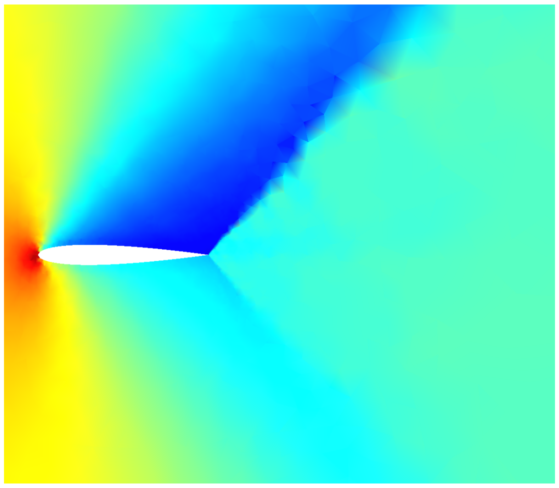
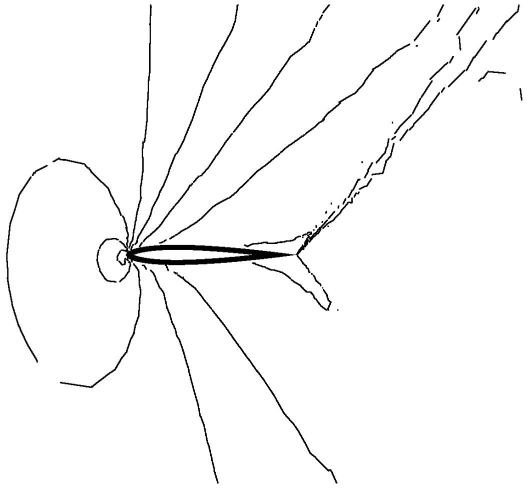
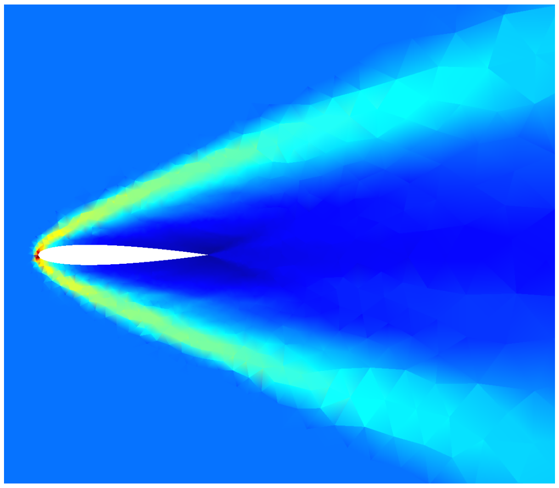

(d) Density isolines with $M = 0.78$, $\alpha = 15°$

Figure 5.8: Steady states of the airfoil test problem with $p = 1$ with different Mach numbers and angles of attack


(a) Density with $M = 0.98, \alpha = 5^\circ$


(b) Density isolines with $M = 0.98$, $\alpha = 5^\circ$


(c) Density with $M = 4, \alpha = 0^\circ$

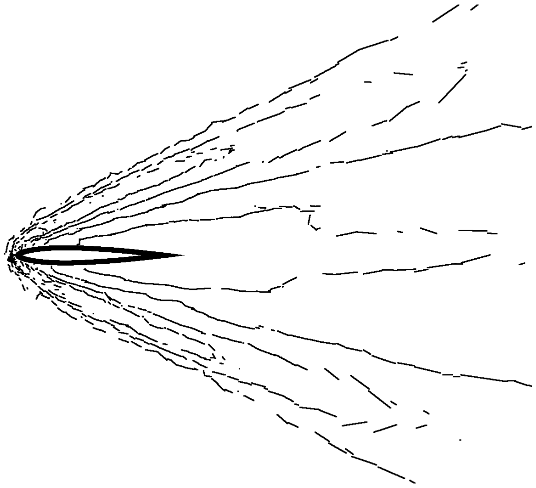
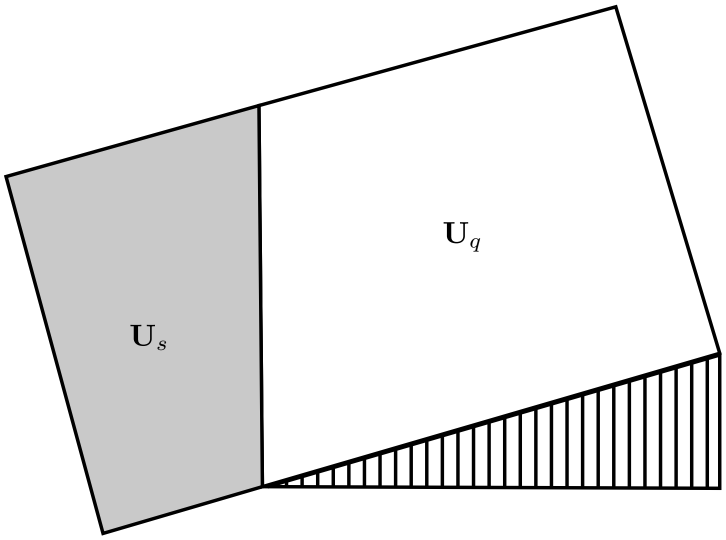

(d) Density isolines with $M = 4$, $\alpha = 0^\circ$

Figure 5.9: Computational domain Ω for the double mach reflection test problem. The striped triangle represents the reflecting wedge. The shaded region on the left is the shock region $\mathbf{U}_s$ while the region on the right is the pre-shock condition $\mathbf{U}_q$.



5.4.3 Double Mach Reflection

The double Mach reflection test problem models a planar shock wave over a reflecting angled wedge. This is equivalent to modeling an angled shock wave moving over a straight reflecting wall. The reflections of the initial shock wave create additional shock waves and contact discontinuities during this simulation.

We begin by sending a down- and right-moving shock wave across our domain, shown in Figure 5.9. The computational domain Ω is defined by $x = [0, 4], y = [0, 1]$. The lower boundary in our domain models a reflecting wedge beginning at $x_0 = \frac{1}{6}$.

We begin with an incident Mach 10 shock wave with propagation angle $\theta = 60°$ from the x-axis; i.e., we assume the wedge has a half-angle of 30°. We assume that the unperturbed flow's density and pressure are equal to 1.4 and 1 respectively. The after shock values are calculated to satisfy the Rankine-Hugonoit condition [2]. They are given by

$$\mathbf{U}_s = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ E \end{pmatrix} = \begin{pmatrix} \rho_s \\ 8.25\cos(\theta)\rho_s \\ -8.25\sin(\theta)\rho_s \\ \frac{1}{2}\rho_s 8.25^2 + \frac{P_s}{\gamma-1} \end{pmatrix}, \tag{5.17}$$

Table 5.4: Performance of the double Mach reflection test problem

Mesh	Elements	Memory	GTX 460 Runtime (min)	GTX 580 Runtime (min)
A	68,622	70.91 MB	2.94	2.07
B	236,964	225.73 MB	36.62	17.72
C	964,338	995.55 MB	-	139.46

where ρ_s is given by

$$\rho_s = \frac{\gamma(\gamma+1)M^2}{(\gamma-1)M^2+2}, \tag{5.18}$$

and the pressure is given by $P_s = 116.5$. The region to the right of the shock wave has the following initial conditions

$$\mathbf{U}_q = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ E \end{pmatrix} = \begin{pmatrix} \gamma \\ 0 \\ 0 \\ \frac{1}{\gamma+1} \end{pmatrix}. \tag{5.19}$$

The left boundary condition sets an inflow with $\mathbf{U}_s$ as the parameter values. The boundary condition along the top of the domain keeps up with the speed of the incident shock wave to simulate the effects of an infinitely long wave. Along the top boundary, at integration points to the left of the shock wave, the exact values from $\mathbf{U}_s$ are prescribed, while points to right of the shock wave use the values from $\mathbf{U}_q$. The x-location of the shock wave along the top boundary ($y = 1$) is given by

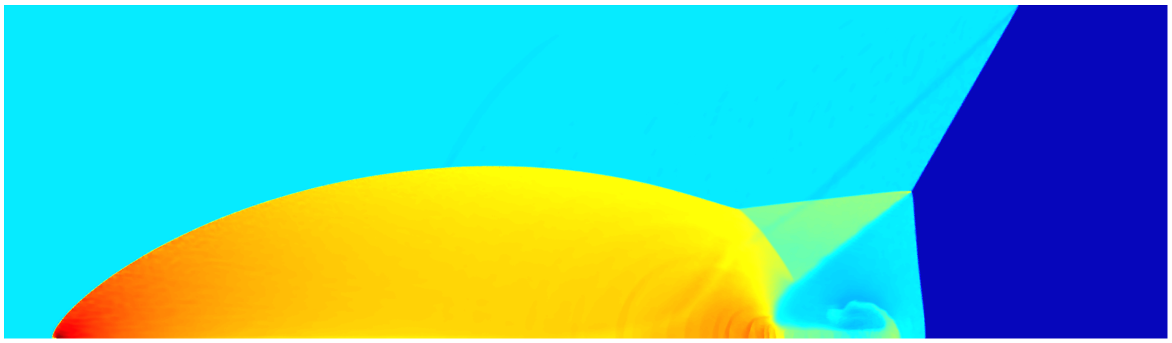
$$x_s = x_0 + \frac{1+20t}{\sqrt{3}}. \tag{5.20}$$

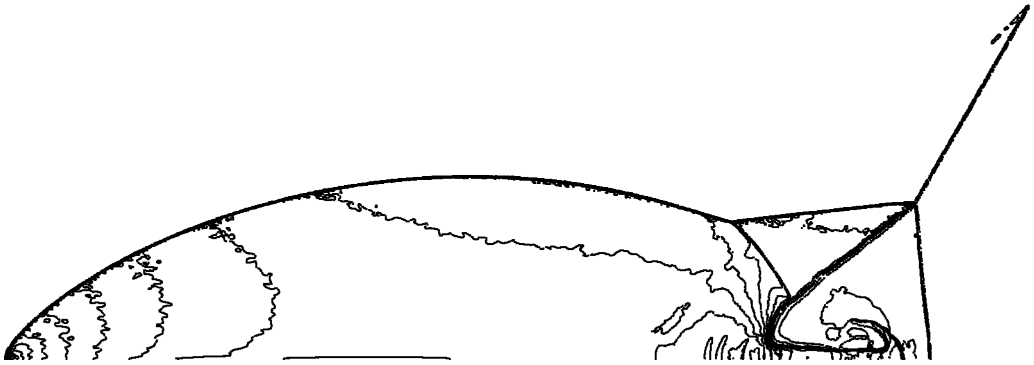
The lower boundary condition prescribes the values of the shock $\mathbf{U}_s$ at $x \leq x_0$ and uses reflecting boundary conditions beyond to simulate a perfectly reflecting wedge.

Our test set runs over three meshes of varying $h-$refinement, reported in Table 5.4. We compute the solution until $t = 0.2$ when the shock has moved nearly across the entire domain. Our solution is computed using $p = 1$ linear polynomials with the slopes limited using the Barth-Jespersen limiter. Mesh refinement is done by setting a smaller maximum edge length and creating a new mesh with GMSH.

The density and density isolines at $t = 0.2$ for the most refined mesh, C, are plotted in Figure 5.10. Our jet stream travels and expands as in similar simulations by [7]. The exact

Figure 5.10: Density for the double Mach reflection problem using $p = 1$


(a) Density for mesh C


(b) Density isolines for mesh C

boundary condition at the top edge of our domain introduces small numerical artifacts which can be seen at the front and top of the shock.

Table 5.4 also reports total runtime for both GPUs used and memory costs for these simulations. As mesh C requires more memory than the 1GB of available video memory on the GTX 460, we run the test for this mesh on only the GTX 580. The simulation time, even for the very large meshes, is not prohibitive. Meshes of C's size are typically too large to be run in serial implementations, usually requiring supercomputing time. In contrast, the GTX 580 completed this simulation in less than two and a half hours.

Table 5.5: Mesh sizes for the supersonic vortex test problem used for benchmarking

Mesh	A	B	C	D	E	F
Elements	180	720	2,880	11,520	46,080	184,320
Edges	293	1126	4,412	17,464	69,488	277,216

5.5 Benchmarks

After demonstrating the accuracy and robustness of our implementation, we now measure the performance. Three factors completely determine the performance of this implementation: the number of elements in the mesh N, the degree p polynomial approximation, and the number of variables n in the system. We therefore attempt to isolate each factor to measure overall performance. In all tests, we used the fourth order Runge-Kutta time integrator with a stable timestep.

5.5.1 Serial Comparison on a CPU

We compare the performance of this implementation with a CPU implementation computing in serial. To create the serial implementation, we rewrote the GPU implementation in C, making the following necessary adjustments. We replace every parallel computation run by a thread with a for loop. We also replace all GPU memory allocation with CPU memory allocations. In addition, we change the memory access pattern in the CPU implementation so that the coefficients are located nearby each other from an element perspective as opposed to a thread perspective. Finally, we remove all unnecessary memory copies as we no longer transfer data between the GPU and CPU. The surface integral contributions are still computed separately from the volume integral contributions and later recombined, maintaining the same basic structure as the GPU implementation.

We will compare the computation run time between the CPU and GPU implementations. Our first test problem will compare the run time of a computation of the Euler equations over different $h-$ and $p-$ refinement. Both the GPU and CPU will compute the same number of timestep iterations for the supersonic vortex test problem described in Section 5.4.1 for each combination of $h-$ and $p-$refinement. The first test mesh, A is created with 180 elements. Each successive mesh is obtained from refinement of the previous mesh. Mesh sizes used for this benchmark are reported in Table 5.5.

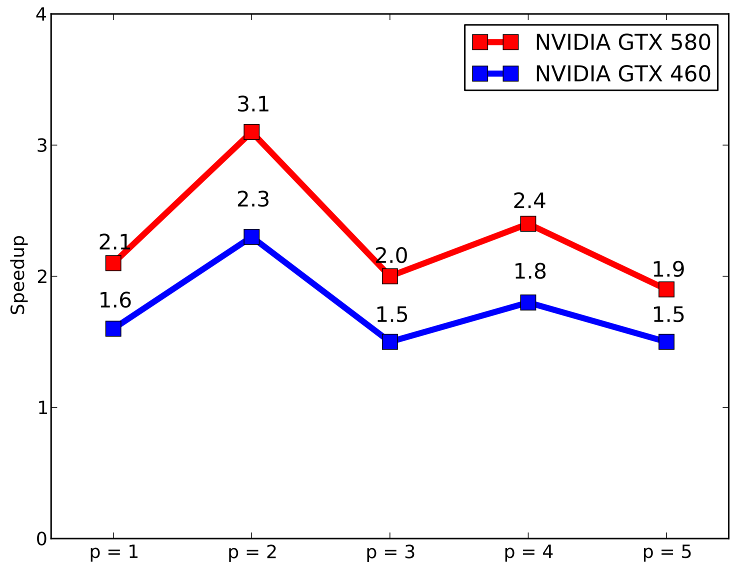
The CPU implementation runs on a single core on an Ubuntu 12.04 machine with an Intel Q6600 CPU running at 2.4GHz with 4GB of RAM. The GPU implementation runs on each of the two NVIDIA GPUs described in Table 5.1. The number of threads per block may be different for each test; only the best results are reported. A sufficient number of timesteps was computed for each of the tests to allow the simulation to run for a nontrivial amount of time on each device. Measured computation time does not include any precomputations, mesh loading time, or the computation of the initial projection from the initial conditions. Each test uses a fourth order, four stage Runge-Kutta time integrator. Measured speedup is simply computed using

$$\text{Speedup} = \frac{\text{Serial Execution Time}}{\text{Parallel Execution Time}}. \tag{5.21}$$

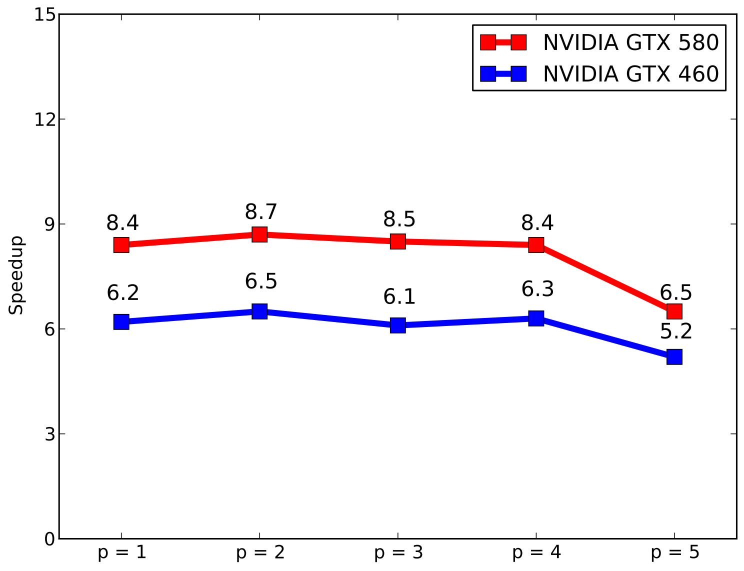
Measured speedup factors over each of the test meshes are reported in Figures 5.11 through 5.13. We see that for smaller meshes, the speedup is much less impressive than it is for larger meshes. Additionally, lower polynomial degrees p seem to offer a greater speedup advantage than higher p. We suspect this is due to a lack of increased parallelism for higher p, as the same number of threads must complete more work as p becomes larger. An alternative parallel implementation may find ways to increase thread granularity by increasing the total number of threads as p increases.

The GTX 460 performed quite well, considering its relative cost to the GTX 580. Unfortunately, the $p = 5$ test on mesh F requires more memory to run than the GTX 460 has available. The speedup factors tend to decrease for higher p, although this is not strictly uniform; e.g., $p = 3$ over mesh C outperforms $p = 2$ on the same mesh. The overall result, however, is clear. As mesh sizes increase, device saturation increases, revealing impressive speedup factors. As p increases, thread work increases without any respective change in parallelism, decreasing our overall speedup. This implementation performs, at best, 52.5 times faster on the GTX 580 and 31.8 times faster on the GTX 480 than on a single core CPU implementation.

Figure 5.11: GPU Speedups

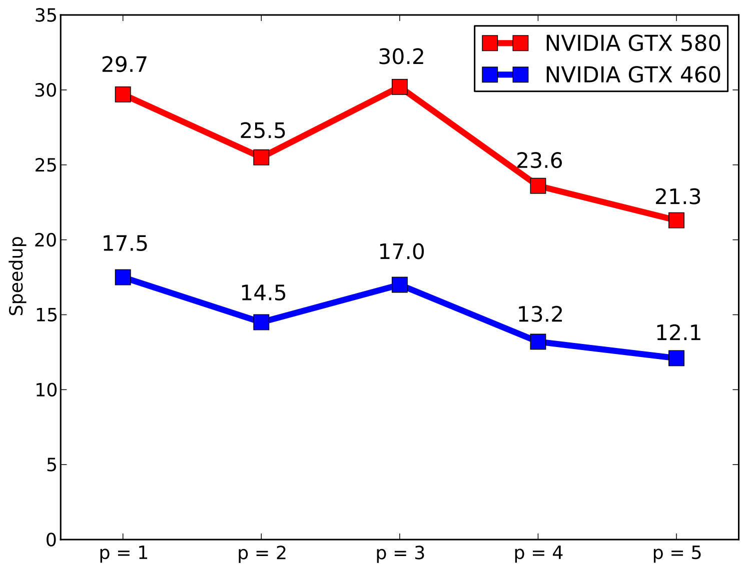


(a) Speedup over mesh A (180 elements)

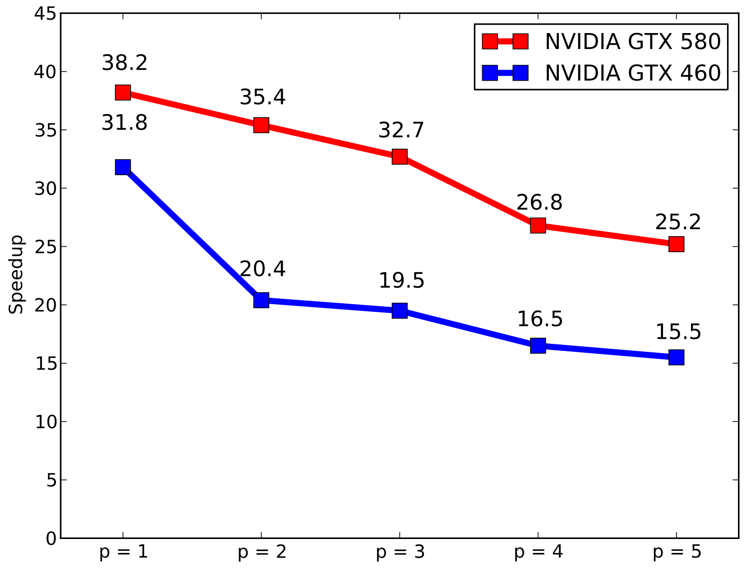


(b) Speedup over mesh B (720 elements)

Figure 5.12: GPU Speedups

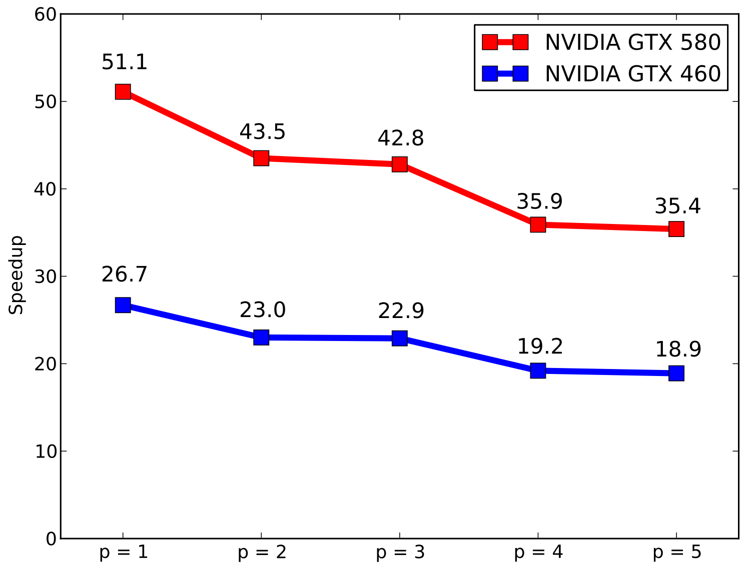


(a) Speedup over mesh C (2,880 elements)

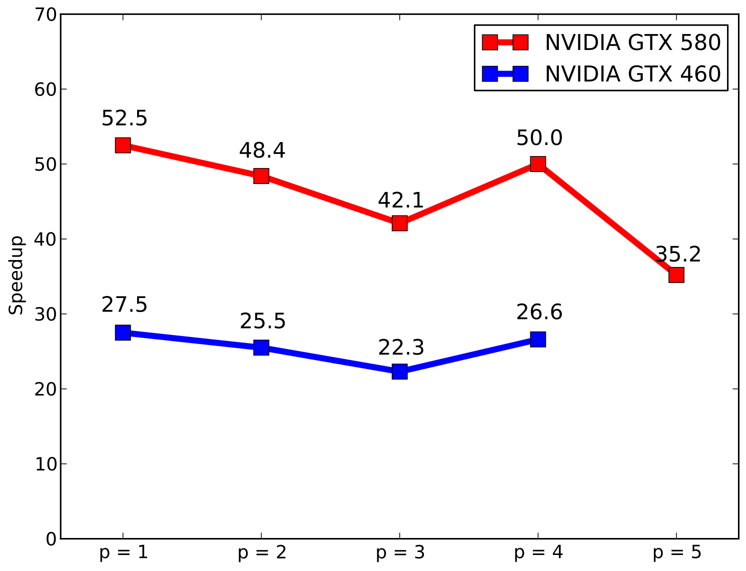


(b) Speedup over mesh D (11,520 elements)

Figure 5.13: GPU Speedups



(a) Speedup over mesh E (46,080 elements)



(b) Speedup over mesh F (184,320 elements)

Table 5.6: TPE for the supersonic vortex test problem

(a) GTX 460 (s)

Mesh	A	B	C	D	E	F
$p = 1$	6.63E−06	1.84E−06	7.22E−07	5.30E−07	4.80E−07	4.62E−07
$p = 4$	4.61E−05	1.42E−05	5.85E−06	4.82E−06	4.38E−06	4.26E−06

(b) GTX 580 (s)

Mesh	A	B	C	D	E	F
$p = 1$	5.09E−06	1.34E−06	4.21E−07	2.98E−07	2.54E−07	2.41E−07
$p = 4$	3.44E−05	1.05E−05	3.25E−07	2.77E−06	2.35E−06	2.26E−06

5.5.2 Scaling

Our final benchmark demonstrates the scalability of our implementation by measuring performance at device saturation. Device saturation requires a sufficiently large number of threads to be running simultaneously. As the total number of simultaneously running threads depends on the size of the mesh, we reach device saturation by computing over larger meshes.

We first fix $p = 1$ and compute the supersonic vortex test problem described in Section 5.4.1 over 10,000 timesteps on meshes A through F from Section 5.5.1. Next, we fix $p = 4$ and repeat the test using 1,000 timesteps over meshes A through F, described in Table 5.5. The computation run times for these two tests are displayed in Figure 5.14.

From this, we can measure the computation run time per element (TPE) for $p = 1$ and $p = 4$. The TPE is defined as

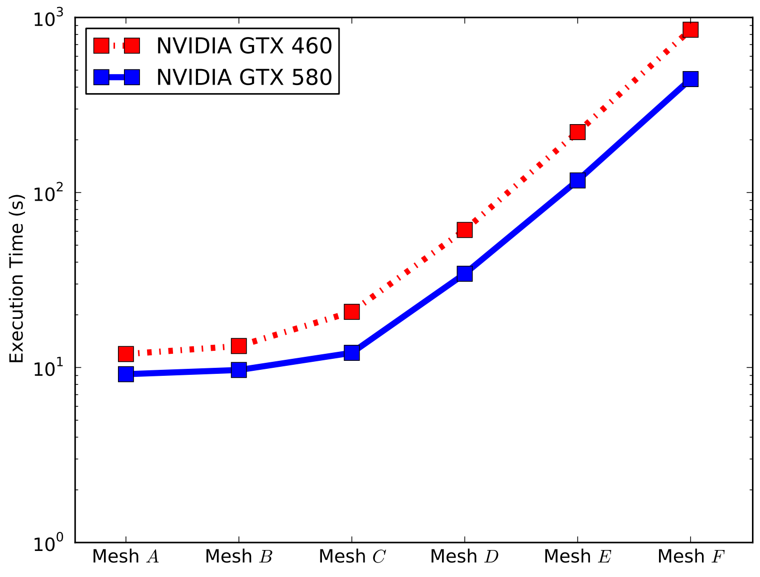
$$\text{TPE} = \frac{\text{Time to Run } T \text{ Timesteps}}{T \times \text{Number of Elements}}. \tag{5.22}$$

The TPE number computed for meshes A through F is reported in Table 5.6. Until the device is fully saturated, the TPE number remains large. This is simply due to the nonoptimal device utilization before device saturation. As the device becomes fully saturated, the TPE number shrinks and then remains nearly the same.

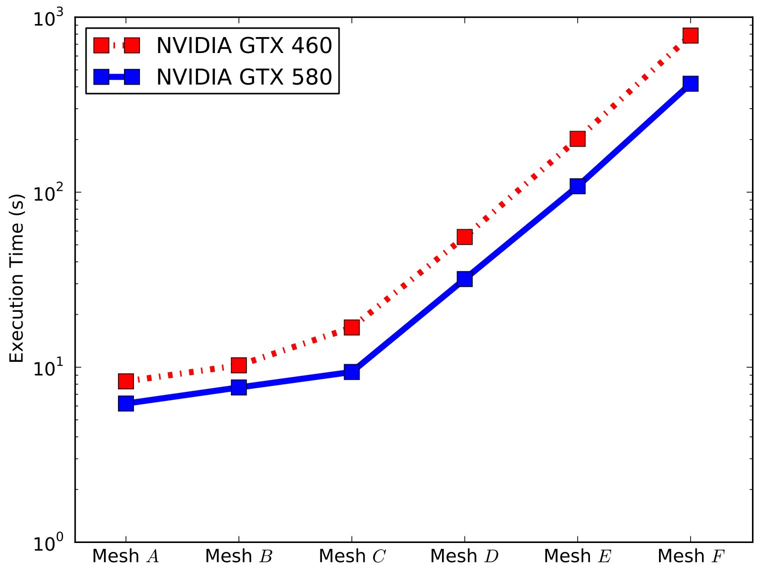
The TPE number remains around $4 \cdot 10^{-7}$ for $p = 1$ and $4 \cdot 10^{-6}$ for $p = 4$ on the GTX 460, and close to half of these values on the GTX 580. The GTX 580, indeed, computes the same test problems at around twice the speed of the GTX 460. Our implementation scales well with $h-$refinement, as more threads are created to compute each element. Unfortunately, $p-$refinement does not introduce more parallelism, as thread count only increases with $h-$refinement. Indeed, the TPE for $p = 4$ is roughly an order of magnitude greater than the TPE for $p = 1$. This is a limitation in our implementation, as higher-order methods, even on small meshes, are computed slowly.

We finally calculate the scalability for $h-$refinement by increasing mesh size and computing 10,000 timesteps of the supersonic vortex test problem for $p = 1$ and 1,000 timesteps of the same test problem for $p = 4$. We compute over meshes A through F and report the total computation time in Figure 5.14. The computation time increases exponentially for each level of $h-$refinement and increases an order of magnitude between $p = 1$ and $p = 4$. Again, until we compute over mesh C with 2,880 elements, we do not reach device saturation, and thus, execution time does not increase exponentially until enough threads are created to saturate the device.

Figure 5.14: GPU execution times



(a) GPU execution time for $p = 1$ and 10,000 timesteps



(b) GPU execution time for $p = 4$ and 1,000 timesteps

Chapter 6

Conclusion

Our implementation is both effective and versatile, able to quickly compute numerical solutions to many hyperbolic conservation laws, including problems from linear advection, the Euler equations, Maxwell's equations, and the shallow water equations. In addition, our implementation performs over fifty times faster on the GTX 580 than a CPU implementation and over thirty times faster on the GTX 460 on some meshes. Not just for toy problems, our implementation easily computes the double Mach reflection test problem with nearly one million elements on the GTX 580 in a very reasonable amount of time.

Our implementation does, however, have the following major limitations. First, we do not include support for partitioned meshes, meaning that only problems with meshes small enough to fit on the limited video memory may be computed. This limitation proves enormously restrictive, as large meshes with high-order polynomial approximation used for more interesting problems require prohibitive amounts of memory. Second, thread count, and thereby parallelism, does not increase with p. High-order polynomial approximation requires more work and computation time than low-order approximation while offering no advantages in parallelism. This explains the loss in speedup over CPU implementations as we increase p. We note, however, that our implementation still offers a significant running time speedup, nonetheless. Third, the surface integral contributions are stored twice for each edge, using valuable memory. Fourth, we do not support mesh adaptation.

We plan to address these issues in future work. By supporting partitioned meshes, problems with arbitrarily large meshes and very high-order polynomial approximation may be divided up. The solution over each partition of the mesh may be computed individually on a single GPU. Furthermore, the solutions over each partition of a mesh may be computed simultaneously, meaning our implementation would scale with multiple GPUs. This scalability would allow our implementation to compute problems of incredible size.

Bibliography

[1] K. Anastasiou and C. T. Chan. Solution of the 2D shallow water equation using the finite volume method on unstructured triangular meshes. *International Journal for Numerical Methods in Fluids*, 24:1225–1245, 1997.

[2] J. D. Jr. Anderson. *Fundamentals of Aerodynamics*. McGraw-Hill Science/Engineering/Math, 2001.

[3] A. Baggag, H. Atkins, and D. Keyes. Parallel implementation of the discontinuous Galerkin method. *NASA Technical Report*.

[4] K. S. Bey, J. T. Oden, and A. Patra. A parallel hp-adaptive discontinuous Galerkin method for hyperbolic conservation laws. *Applied Numerical Methods*, 20:321–336, 1996.

[5] R. Biswas, K. D. Devine, and J. Flaherty. Parallel, adaptive finite element methods for conservation laws. *Applied Numerical Mathematics*, 14(1-3):225–283, 1994.

[6] B. Cockburn, S. Hou, and C. W. Shu. The Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws IV: The multidimensional case. *Mathematics of Computation*, 54(190):545–581, 1990.

[7] B. Cockburn, George E. Karniadakis, and C. W. Shu. The development of discontinuous Galerkin methods. 1999.

[8] B. Cockburn, S. Y. Lin, and C. W. Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws III: One dimensional systems. *Journal of Computational Physics*, 84(1):90–113, 1989.

[9] B. Cockburn and C. W. Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws II: General framework. *Mathematics of Computation*, 52(186):411–435, 1989.

[10] B. Cockburn and C. W. Shu. Runge-Kutta discontinuous Galerkin methods for convection-dominated problems. *Journal of Scientific Computing*, 16(3), 2001.

[11] D. Connor. The discontinuous Galerkin method applied to problems in electromagnetism. Master's thesis, University of Waterloo, Waterloo, 2012.

[12] D. A. Dunavant. High degree efficient symmetrical Gaussian quadrature rules for the triangle. *International Journal for Numerical Methods in Engineering*, 21(6):1129–1148, 1985.

[13] L. C. Evans. *Partial differential equations*. American Mathematical Society, Providence, second edition, 2010.

[14] C. Geuzaine and J. F. Remade. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. `http://geuz.org/gmsh/`.

[15] N. Goedel, T. Warburton, and M. Clemens. GPU accelerated discontinuous Galerkin fem for electromagnetic radio frequency problems. *Antennas and Propagation Society International Symposium*, pages 1–4, 2009.

[16] R. W. Hamming. The unreasonable effectiveness of mathematics. *The American Mathematical Monthly*, 87(2), 1980.

[17] M. Hazewinkel. Gibbs phenomenon. *Encyclopedia of Mathematics*, 2001.

[18] J. S Hesthaven, J. Bridge, N. Goedel, A. Klöckner, and T. Warburton. Nodal discontinuous Galerkin methods on graphics processing units (GPUs).

[19] J. S. Hesthaven and T. Warburton. *Nodal Discontinuous Galerkin Methods*. Springer, New York, 2008.

[20] G. Jiang and C. W. Shu. On cell entropy inequality for discontinuous Galerkin methods. *Mathematics of Computation*, 62(206):531–538, 1994.

[21] C. Johnson and J. Pitkrata. An analysis of the discontinuous Galerkin method for a scalar hyperbolic equation. *Mathematics of Computation*, 46:1–26, 1986.

[22] A. Klöckner, T. Warburton, and J. S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics*, 228(21):7863–7882, 2009.

[23] A. Klöckner, T. Warburton, and J. S. Hesthaven. High-order discontinuous Galerkin methods by GPU metaprogramming. *GPU Solutions to Multi-scale Problems in Science and Engineering*, 2012.

[24] T. Koornwinder. *Two-variable analogues of the classical orthogonal polynomials. In theory and applications of special functions*. Academic Press, 1975.

[25] L. Krivodonova and M. Berger. High-order accurate implementation of solid wall boundary conditions in curved geometries. *Journal of Computational Physics*, 211:492–512, 2006.

[26] D. Kuzmin. A guide to numerical methods for transport equations. 2010.

[27] R. J. Leveque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. SIAM, Philadelphia, 2007.

[28] MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010.

[29] K. Michalak and C. Olliver-Gooch. Limiters for unstructured higher-order accurate solutions of the Euler equations. *Forty-Sixth Aerospace Sciences Meeting*, 2008.

[30] K. W. Morton and D. F. Mayers. *Numerical Solution of Partial Differential Equations : An Introduction*. Cambridge University Press, 2005.

[31] NVIDIA. CUDA C Best Practices Guide, 2011.

[32] D. Pietro, D. Antonio, and A. Ern. *Mathematical Aspects of Discontinuous Galerkin Methods*. Springer, 2012.

[33] P.L. Roe. Approximate Riemann solvers, parameter vectors and difference schemes. *Journal of Computational Physics*, (43):357–372, 1981.

[34] R. E. Shidahl and P. C. Klimes. Aerodynamic characteristics of seven symmetrical airfoil sections through 180-degree angle of attack for use in aerodynamic analysis of vertical axis wind turbines, 1981.

[35] L. N. Trefethen and D. Bau III. *Numerical Linear Algebra*. SIAM, Philadelphia, 1997.

[36] Wikipedia. Flynn's taxonomy. `http://en.wikipedia.org/wiki/Flynn%27s_taxonomy`, 2013. [Online; accessed 23-March-2013].

[37] R. Yang and Christopher Zach. GP-GPU: General purpose programming on the graphics processing unit. `http://cvg.ethz.ch/teaching/2011spring/gpgpu/`.