University of New Mexico
UNM Digital Repository

Mathematics & Statistics ETDs Electronic Theses and Dissertations

1-28-2015

Performance Analysis and Optimization of Hermite Methods on NVIDIA GPUs Using CUDA

Evan T. Dye

Follow this and additional works at: https://digitalrepository.unm.edu/math_etds

Recommended Citation
Dye, Evan T.. "Performance Analysis and Optimization of Hermite Methods on NVIDIA GPUs Using CUDA." (2015). https://digitalrepository.unm.edu/math_etds/15

This Thesis is brought to you for free and open access by the Electronic Theses and Dissertations at UNM Digital Repository. It has been accepted for inclusion in Mathematics & Statistics ETDs by an authorized administrator of UNM Digital Repository. For more information, please contact disc@unm.edu.

Evan T. Dye

Candidate

Mathematics and Statistics

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Daniel Appelö, Chairperson

Stephen Lau

Jens Lorenz

Performance Analysis and Optimization of Hermite Methods on NVIDIA GPUs Using CUDA

by

Evan T. Dye

B.S., Mathematics, Oklahoma Panhandle State University, 2011
B.M., Music, Oklahoma Panhandle State University, 2011

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Mathematics

The University of New Mexico

Albuquerque, New Mexico

December, 2014

©2014, Evan T. Dye

Dedication

For all those who never give up on their dreams

“Learn from yesterday, live for today, hope for tomorrow.
The important thing is not to stop questioning.” – Albert Einstein

Acknowledgments

I would like to thank Dr. Daniel Appelö for his support and expertise throughout my graduate studies, for his seemingly infinite patience, for being my thesis advisor, and for being the chair of my thesis committee. It is my honor to have Dr. Stephen Lau and Dr. Jens Lorenz serve on my thesis committee, and to have had the opportunity to learn from them. Special thanks to Dr. Daniel Teske and Dr. Matthew Saunders for their instruction and advisement during my undergraduate study, and for pushing me towards graduate studies. Gratitude is to be given to Ana Parra Lombard for always being there to answer administrative questions, and making sure paperwork was turned in on time. Thanks to the University of New Mexico Department of Mathematics and Statistics for funding me in the form of a teaching assistantship, and to all those within the department that I have had the pleasure of working with. Thanks to Missy and Fabian for their invaluable friendship. Special thanks to my mother Rachel, for educating me at home from preschool through high school; to my sister Emily, for her support and assistance; to my sister Eva, for picking up the slack; to my brother Eric and his family, for their continued support and encouragement through difficult times; to my fiancée Bond, for her encouragement, love, and patience; and to my son Aiden, for always smiling. To all those who have supported and encouraged me, thank you.

Supported in part by NSF Grant DMS-1319054. Any conclusions or recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of NSF.

Performance Analysis and Optimization of Hermite Methods on NVIDIA GPUs Using CUDA

by

Evan T. Dye

B.S., Mathematics, Oklahoma Panhandle State University, 2011
B.M., Music, Oklahoma Panhandle State University, 2011

M.S., Mathematics, University of New Mexico, 2014

Abstract

In this thesis we present the first, to our knowledge, implementation and performance analysis of Hermite methods on GPU accelerated systems. We give analytic background for Hermite methods; give implementations of the Hermite methods on traditional CPU systems as well as on GPUs; give the reader background on basic CUDA programming for GPUs; discuss performance characteristics of GPUs; we give recommended design choices for GPU implementations of Hermite methods; and present and discuss examples which illustrate the effect these design choices have on performance. Lastly, we present areas of future research that may yield increased performance for Hermite methods on GPUs.

Contents

List of Figures **x**

List of Algorithms **xii**

List of Sample Code **xiii**

1 Introduction **1**

2 Hermite Methods **6**

2.1 1D Example 6

2.2 Linear Constant Coefficient Systems of Hyperbolic Equations 9

2.3 2D Example 10

3 Implementation of Hermite Methods **12**

3.1 1D Serial Implementation 12

3.2 2D Parallel Implementation 18

4 CUDA 25

4.1 Host, Device, and Kernel 26

4.2 Threads, Blocks, and Grids 26

4.3 Warps 29

4.3.1 Divergence 29

4.4 Memory Spaces 31

4.5 Special Performance Metrics 35

4.6 CUDA Implementation of Hermite Methods 36

5 Numerical Examples 40

5.1 Convergence Study 41

5.2 Performance Studies 44

5.2.1 Example 1: Performance at Maximal Number of Threads Per Block 44

5.2.2 Example 2: Performance at 64 Threads Per Block 47

5.2.3 Example 3: Performance with Maximal q 50

6 Summary and Outlook 52

Appendices 54

A Creation of $\widetilde{A}^{-1}$ 55

B Notes on Scaling **61**

References **65**

List of Figures

1.1 Floating-point operations per second for CPUs and GPUs through time, from [3]. 2

1.2 Memory bandwidth in GB/s for CPUs and GPUs through time, from [3]. 3

1.3 Architectural differences between the cache and control heavy CPU and the arithmetic logical unit heavy GPU, from [3]. 4

2.1 Time stepping process where I represents the interpolation, T represents the stepping in time, solid points represent the primal-grid, and open points represent the dual-grid. 8

4.1 The CUDA thread hierarchy, from [3]. 28

4.2 The CUDA memeory hierarchy, from [3]. 34

5.1 Error convergence when $m = 3$, $q = 4m + 2$, and a CFL of 0.9. 42

5.2 Error convergence when $m = 4$, $q = 4m + 2$, and a CFL of 0.9. 43

5.3 Performance with a fixed block size of 1024, $q = 2m + 1$, and a CFL of 0.1. 45

5.4 Performance with a fixed block size of 1024, $q = 2m + 1$, and a CFL of 0.1. 46

5.5 Performance with a fixed block size of 64, $q = 2m + 1$, and a CFL of 0.1. 48

5.6 Performance with a fixed block size of 64, $q = 2m + 1$, and a CFL of 0.1. 49

5.7 Performance with a fixed block size of 64, $q = 4m + 2$, and CFL of 0.9. 51

List of Algorithms

3.1 1D Time Loop: First Half Time Step 14
3.2 1D Time Loop: Second Half Time Step 16
3.3 1D Time Loop: Boundary Conditions 17
3.4 2D Time Loop: Part 1 20
3.5 2D PDE: Method 1 21
3.6 2D PDE: Method 2 22
3.7 2D Time Loop: Part 2 23
4.1 CUDA: First Half Time Step 38

List of Sample Code

4.1 Example where divergence would likely occur. 29

A.1 Creation of the matrix $\widetilde{A}^{-1}$. 55

Chapter 1

Introduction

The Hermite method for numerically solving periodic hyperbolic partial differential equations works well in parallel, due to its large number of computations per communication. In addition, the Hermite method only requires data from neighboring points, unlike some methods which require several points in each direction. So when solving on a domain partitioned across processors, only the boundary data needs to be communicated. These characteristics lead to a good surface to volume ratio. Furthermore, between each communication, two half time steps and one interpolant need to be computed; thus, there is a relatively long period of time between communications. This delay between communications means it is relatively easy to partition the domain so that communication can start and finish while computations are carried out. All of these characteristics combined means the algorithms have good weak scaling on both distributed memory and shared memory systems.

The down side to the Hermite method is that it is not particularly easy to code, and it requires a structured grid. There is no way around the complexity of code, as each problem will require a certain amount of new code for the partial differential equation being solved. Complex geometries, however, can be handled by using the Hermite

method on the interior of the grid, while using another method for the boundaries. A common way of dealing with complex geometries is to use the discontinuous Galerkin method on the boundary, where these complex geometries exist, and then interface the discontinuous Galerkin method with the Hermite method; using, the Hermite method on the interior of the computational domain [1].

Due to the nature of real-time, high-definition 3D graphics, the programmable Graphics Processor Unit (GPU) has become a hugely parallel, multi-threaded, many-core processor with high computational horsepower with high memory bandwidth. Figure 1.1 shows the large discrepancy between a standard CPU and the GPU in terms of FLOPs, while Figure 1.2 shows the discrepancy in memory bandwidth. These figures show why it is important to consider the implementation of numerical methods on GPUs.

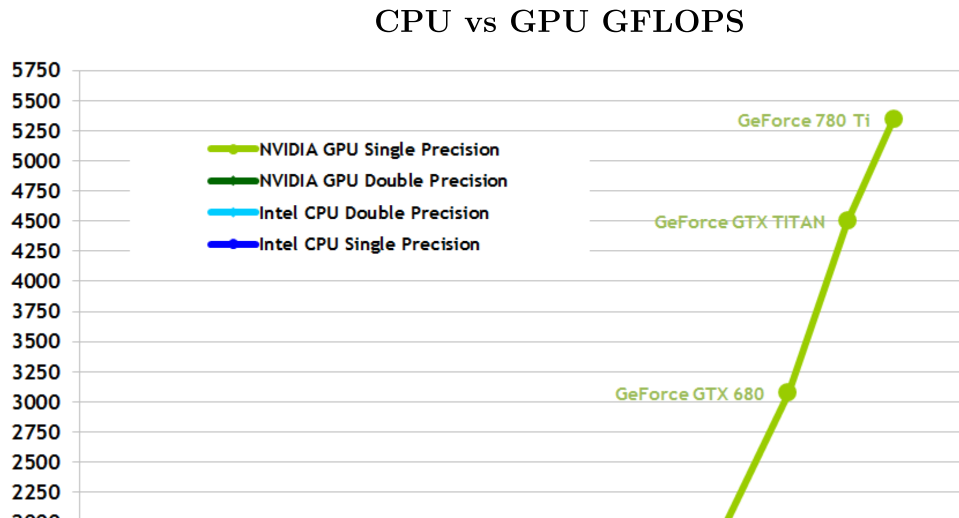

Figure 1.1: Floating-point operations per second for CPUs and GPUs through time, from [3].

The reason why the GPU is so much faster than the CPU, is that the GPU is specialized for computationally intensive, parallel computations. Unlike the CPU

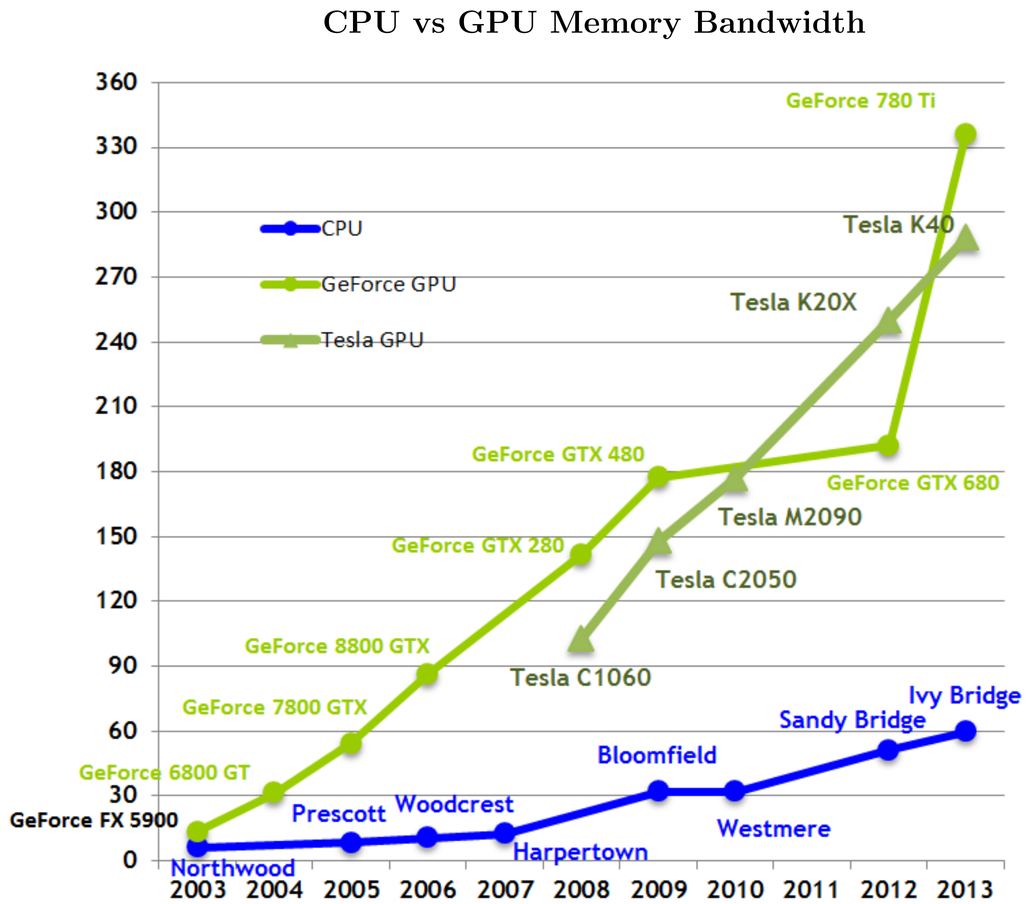

Figure 1.2: Memory bandwidth in GB/s for CPUs and GPUs through time, from [3].

which must run general purpose code. The GPU design is one of a much higher transistor count, of which the majority are devoted to data processing rather than the more standard data caching and flow control of a CPU. Figure 1.3 illustrates these architectural differences.

So in general, the GPU is well-suited to address problems which require large parallel computations. The GPU is designed to make the same computations on many data elements simultaneously, which is to say a high arithmetic to memory operation ratio. Because the same operations are carried out on each data element,

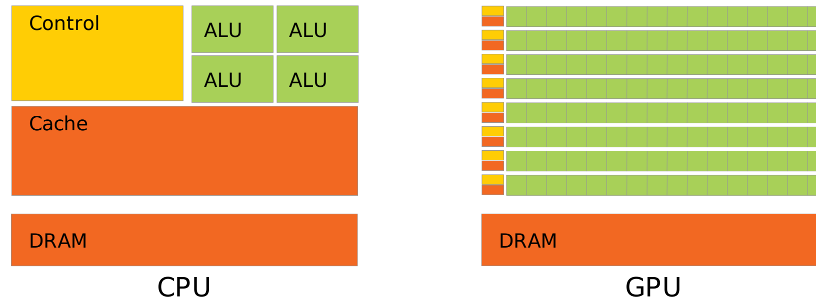

Figure 1.3: Architectural differences between the cache and control heavy CPU and the arithmetic logical unit heavy GPU, from [3].

there is a much lower need for advanced flow control. This is in contrast to the CPU, which makes heavy utilization of advanced flow control. In addition, memory access latency on the GPU can be hidden with calculations instead of the large data caches used on the CPU.

In order to take advantage of the computational power of GPUs for computations beyond graphics, NVIDIA introduced CUDA® in November of 2006. CUDA is a general purpose parallel computing platform and programming model for NVIDIA GPUS. An alternative to CUDA is OpenCL, which was initially developed by Apple Inc.; however, it is now developed by the Khronos Compute Working Group. The technical details of the the OpenCL 1.0 specification were finished in November of 2008.[1] OpenCL is able to run on various GPUs, FPGAs and CPUs, and is based on C99. Most of the discussions and conclusions in this paper can be translated to OpenCL.

Because of the way GPUs perform computational tasks, the Hermite method is an excellent candidate for implementation on GPUs. We began by developing a naive GPU implementation of the Hermite method, which yielded relatively meager

[1] http://www.khronos.org/opencl/

performance gains over the CPU implementation. However, we have found some simple design patterns for coding Hermite methods on the GPU, specifically regarding memory usage and block size, which increased performance significantly from the naive GPU implementation. These results show that significant performance gains can be attained on GPUs, without the need of designing highly complex code. This is particularly important when working with heterogeneous systems where many performance considerations must be made during code design.

To our knowledge, this thesis presents the first implementation and performance analysis of Hermite methods on GPU accelerated systems. To better understand Hermite methods, the method will be discussed from an analytic perspective in Chapter 2. Implementation of the Hermite methods on traditional CPU systems will be discussed in Chapter 3. Chapter 4 introduces CUDA programming on GPUs, the performance characteristics of GPUs, and an implementation of the Hermite method on GPUs. Performance examples are presented and discussed in Chapter 5. Lastly, Chapter 6 closes this thesis with summary and outlook.

Chapter 2

Hermite Methods

In this chapter we will introduce arbitrary-order Hermite methods for the numerical solution of periodic hyperbolic partial differential equations, given initial data. Hermite methods differ from standard difference methods because derivative data is carried at each grid point, in addition to the values of the solution. It has been proved in [2] that these methods, using all derivatives up to order m in each coordinate direction, are stable with m-independent CFL condition; furthermore, the method converges at order $2m+1$. To explain the Hermite method, one-dimensional scalar, linear constant coefficient system, and two-dimensional scalar examples will be explored.

2.1 1D Example

We begin to illustrate the Hermite method with the following simple problem:

$$u_t = au_x,\ u(x,0) = f(x),\ x \in [x_L, x_R],\ u(x_L,t) = u(x_R,t),\ t \geq 0, \tag{2.1}$$

where $f(x)$ is a smooth function, and a is a real constant. For the initial data, $f(x)$, to be compatible with the periodic boundary conditions, it must be $(x_R - x_L)$-periodic.

Define the primal-grid as follows:

$$X_P = \{x_i = x_L + hi : i = 0, 1, \ldots, N\}, \quad h = \frac{x_R - x_L}{N}. \tag{2.2}$$

Similarly, the dual-grid is defined as follows:

$$X_D = \left\{ x_{i+\frac{1}{2}} = x_L + \frac{h(2i+1)}{2} : i = 0, 1, \ldots, (N-1) \right\}. \tag{2.3}$$

We begin by constructing polynomial approximations to the solution, one for each point on the dual-grid. These polynomials are centered at the points on the dual-grid, and satisfy the initial data upto m derivatives at their neighboring primal-grid points; thus, they are Hermite interpolants. At time $t = 0$, the resulting spacial polynomials are denoted as follows:

$$p_{i+\frac{1}{2}}(x, 0) = \sum_{j=0}^{2m+1} c_{j0} \left(x - x_{i+\frac{1}{2}} \right)^j, \quad i = 0, 1, \ldots, (N-1) \tag{2.4}$$

Now that the data has been interpolated, the coefficients c_{j0} are expanded as a temporal polynomial in order to step forward in time. Precisely, we expand $p_{i+\frac{1}{2}}(x, 0)$ in a Taylor series in time centered around $t_0 = 0$, which we write in the following manner:

$$p^0_{i+\frac{1}{2}}(x, t) = \sum_{j=0}^{2m+1} \sum_{k=0}^{2m+1-j} c_{jk} \left(x - x_{i+\frac{1}{2}} \right)^j (t - t_0)^k, \quad i = 0, 1, \ldots, (N-1). \tag{2.5}$$

From the spatial and temporal derivatives of the PDE, we know that the following relations hold for the exact solution:

$$\frac{\partial^k u}{\partial t^{k-j} \partial x^j} = a \frac{\partial^k u}{\partial t^{k-j-1} \partial x^{j+1}}. \tag{2.6}$$

Enforcing these relations on $p^0_{i+\frac{1}{2}}(x, t)$ at time $t = 0$ yields the following recursion relation for c_{jk}:

$$c_{jk} = a \frac{(j+1)}{k} c_{j+1,k-1}, \quad k = 1, 2, \ldots, 2m+1-j, \; j = 0, 1, \ldots, 2m. \tag{2.7}$$

We then use this recursion relation to compute all c_{jk} directly from c_{j0}, $j = 0, 1, \ldots, 2m+1$.

Next, we evaluate $p^0_{i+\frac{1}{2}}(x,t)$ and its derivatives at the half time-step $t_{\frac{1}{2}} = \frac{\Delta t}{2}$ and dual-grid point $x_{i+\frac{1}{2}}$. This yields the following approximation:

$$\frac{1}{j!}\frac{\partial^j u}{\partial x^j}\left(x_{i+\frac{1}{2}}, t_{\frac{1}{2}}\right) \approx \sum_{k=0}^{2m+1-j} c_{jk}\left(\frac{\Delta t}{2}\right)^k, \quad j = 0, \ldots, m. \tag{2.8}$$

Now that there is data for the function and m derivatives on the dual-grid at time $t = t_{\frac{1}{2}}$, the above process can be repeated in order to the produce the data on the primal-grid at a time $t_1 = \Delta t$. Note that at the edges of the domain, we use the periodic boundary conditions in order to construct the interpolant. This entire process can be repeated until we reach the final time $t_n = n\Delta t$. Figure 2.1 illustrates the full time stepping process.

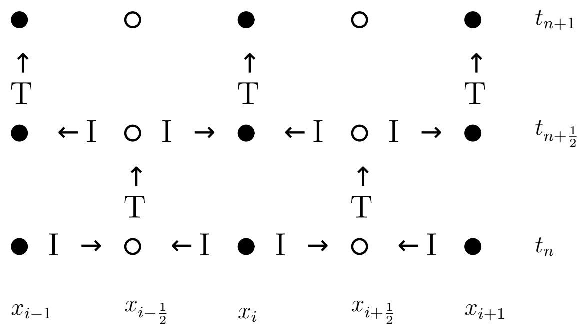

Figure 2.1: Time stepping process where I represents the interpolation, T represents the stepping in time, solid points represent the primal-grid, and open points represent the dual-grid.

2.2 Linear Constant Coefficient Systems of Hyperbolic Equations

We next describe the case of linear constant coefficient systems of hyperbolic equations. For these systems, the interpolation process is unchanged from the scalar example in Section 2.1; although, it is applied to each field. The only change occurs in the recursion relation, which is based on the partial differential equation.

Again, consider a simple example with a constant coefficient system in r-equations:

$$\begin{aligned} u_t = Au_x, \qquad A = A^T \in \mathbb{R}^{r\times r}, \\ u(x,0) = f(x), \; x \in [x_L, x_R], \; u(x_L,t) = u(x_R,t), \; t \geq 0. \end{aligned} \tag{2.9}$$

For this case, data consists of approximate scaled derivative and function values for each component of the r components. The interpolation, recursion relation, and time stepping are carried out component wise, thus solving linear constant coefficient systems of hyperbolic equations with initial data is almost identical to the scalar case.

The polynomial p, is now a $\mathbb{R}^r$-valued function with each component consisting of a polynomial of degree $2m+1$ as before. Furthermore, the polynomial coefficients $c_{jk} \in \mathbb{R}^r$, are computed by requiring that p satisfies the following relation:

$$\frac{\partial^k u}{\partial t^{k-j}\partial x^j} = A\frac{\partial^k u}{\partial t^{k-j-1}\partial x^{j+1}}. \tag{2.10}$$

This in term gives the vector recursion relation:

$$c_{jk} = \frac{(j+1)}{k}Ac_{j+1,k-1}. \tag{2.11}$$

Again, the entire process can be repeated until $t_n = n\Delta t$.

2.3 2D Example

For the two-dimensional case, consider the following problem:

$$
\begin{aligned}
&u_t = a\left(u_x + u_y\right),\ u\left(x,y,0\right) = f\left(x,y\right),\ x \in \left[x_L, x_R\right],\ y \in \left[y_L, y_R\right], \\
&u\left(x_L,y,t\right) = u\left(x_R,y,t\right),\ u\left(x,y_L,t\right) = u\left(x,y_R,t\right),\ t \geq 0,
\end{aligned} \tag{2.12}
$$

where $f\left(x,y\right)$ is a smooth function, and a is a real constant. For the initial data, $f\left(x,y\right)$, to be compatible with the periodic boundary conditions, it must be $(x_R - x_L)$-periodic in x and $(y_R - y_L)$-periodic in y. Defining X_P, Y_P and X_D, Y_D as in (2.2) and (2.3), but with $h_x = \frac{x_R - x_L}{N_x}$ and $h_y = \frac{y_R - y_L}{N_y}$; then the points on the primal-grid and dual-grid can be denoted by (x_i, y_j) and $\left(x_{i+\frac{1}{2}}, y_{j+\frac{1}{2}}\right)$ respectively.

In a similar manner to the one-dimensional case, we can construct the following polynomial:

$$
\begin{aligned}
&p_{i+\frac{1}{2},j+\frac{1}{2}}\left(x,y,0\right) = \sum_{k=0}^{2m+1}\sum_{l=0}^{2m+1} c_{kl0}\left(x - x_{i+\frac{1}{2}}\right)^k\left(y - y_{j+\frac{1}{2}}\right)^l, \\
&i = 0,1,\ldots,\left(N_x - 1\right), \quad j = 0,1,\ldots,\left(N_y - 1\right).
\end{aligned} \tag{2.13}
$$

As before, c_{kl0} for $i = 0,1,\ldots,\left(N_x - 1\right)$ and $j = 0,1,\ldots,\left(N_y - 1\right)$ are computed from the initial data on the primal grid. Here we insist that the derivatives

$$
\frac{\partial^{k+l} p_{i+\frac{1}{2},j+\frac{1}{2}}\left(x,y,0\right)}{\partial x^k \partial y^l}, \quad k = 0,1\ldots,m,\ l = 0,1,\ldots,m
$$

agree with the initial data at the four adjacent nodes on the primal grid.

Now $p_{i+\frac{1}{2},j+\frac{1}{2}}$ can be expanded into a Taylor series in time, which for exact evolution need to be of degree $2\left(2m+1\right)$. However, this is of a higher degree than needed to attain $2m+1$ order accuracy. Consequently, we can choose the order q, in

time, such that $2m+1 \leq q \leq 2(2m+1)$, noting that no additional spatial data is required. This brings about the following polynomial:

$$p^0_{i+\frac{1}{2},j+\frac{1}{2}}(x,y,t) = \sum_{k=0}^{2m+1}\sum_{l=0}^{2m+1}\sum_{s=0}^{q} c_{kls}\left(x - x_{i+\frac{1}{2}}\right)^k\left(y - y_{j+\frac{1}{2}}\right)^l (t-0)^s, \quad (2.14)$$
$$i = 0,1,\ldots,(N_x-1), \quad j = 0,1,\ldots,(N_y-1).$$

Recall that for the solution u, the following must hold:

$$\frac{\partial^{k+l+s}u}{\partial t^s \partial x^k \partial y^l} = a\left(\frac{\partial^{k+l+s}u}{\partial t^{s-1}\partial x^{k+1}\partial y^l} + \frac{\partial^{k+l+s}u}{\partial t^{s-1}\partial x^k \partial y^{l+1}}\right). \quad (2.15)$$

Thus by imposing the relations on $p^0_{i+\frac{1}{2},j+\frac{1}{2}}(x,y,t)$ at $\left(x_{i+\frac{1}{2}}, y_{j+\frac{1}{2}}, 0\right)$, we arrive at the following recursion relation for c_{kls}

$$c_{kls} = \frac{a}{s}\left((k+1)\,c_{k+1,l,s-1} + (l+1)\,c_{k,l+1,s-1}\right). \quad (2.16)$$
$$k = 0,1,\ldots,2m,\ j = 0,1,\ldots,2m,\ s = 1,2,\ldots,q.$$

Using (2.16) we can evaluate $p^0_{i+\frac{1}{2},j+\frac{1}{2}}(x,y,t)$ and its derivatives at the half time-step $t_{\frac{1}{2}} = \frac{\Delta t}{2}$ and dual-grid point $\left(x_{i+\frac{1}{2}}, y_{j+\frac{1}{2}}\right)$, yielding the following approximation:

$$\frac{1}{(k!)\,(l!)}\frac{\partial^{k+l}u}{\partial x^k \partial y^l}\left(x_{i+\frac{1}{2}}, y_{j+\frac{1}{2}}, t_{\frac{1}{2}}\right) \approx \sum_{s=0}^{q} c_{kls}\left(\frac{\Delta t}{2}\right)^s, \quad k = 0,\ldots,m, \quad (2.17)$$
$$l = 0,\ldots,m \quad i = 0,1,\ldots,(N_x-1), \quad j = 0,1,\ldots,(N_y-1).$$

Since there is now data for the function and m derivatives on the dual-grid at time $t = t_{\frac{1}{2}}$, the above process can be repeated in order to the produce the data on the primal-grid at a time $t_1 = \Delta t$. As before, we use the periodic boundary conditions to construct the interpolant at the edges of the domain. As in the previous examples, the entire process can be repeated until $t_n = n\Delta t$.

Chapter 3

Implementation of Hermite Methods

In this chapter we will discuss a one-dimensional serial implementation, and a two-dimensional parallel implementation of the Hermite method. The Hermite method can be broken down into three main sections: computation of coefficients of a Hermite interpolating polynomial from initial data, computation of remaining coefficients using a recursion relation, and evolution in time via Taylor time stepping. Pseudocode is presented for all three of these steps.

3.1 1D Serial Implementation

Consider the following problem:

$$u_t = u_x,\ u(x,0) = f(x),\ x \in [x_L, x_R],\ u(x_L,t) = u(x_R,t),\ t \geq 0, \tag{3.1}$$

where $f(x)$ is a smooth function. For better numerical conditioning, the implementation differs from Section 2.1 in that the initial data, along with the approximate solution, is scaled in the following way (see Appendix B for more details.) We store the coefficients of $p_i(x,0)$ in an array u, of dimension $(m+1)\times N$, such that:

$$p_i(x,t) = \sum_{j=0}^{2m+1} \sum_{k=0}^{2m+1-j} c_{jk} \left(\frac{x-x_i}{h}\right)^j \left(\frac{t-0}{\Delta t}\right)^k, \quad i = 0,1,\ldots,(N-1) \tag{3.2}$$

$$u[j,i] = \frac{h^j}{j!}\frac{\partial^j f(x_i)}{\partial x^j}, \quad j = 0,1,\ldots,m,\ i = 0,1,\ldots,(N-1)\,. \tag{3.3}$$

Finding the coefficients of the Hermite interpolating polynomial on the dual-grid amounts to solving a linear system $\tilde{A}\vec{\tilde{c}} = neighborData$. In our implementations, $\widetilde{A}^{-1}$ is computed explicitly and stored for later use (see Appendix B for an explanation, or Appendix A for an implementation for computing $\widetilde{A}^{-1}$.)

In Algorithm 3.1, the approximate solution on the dual-grid at a half time step is computed. Then in Algorithm 3.2 the approximate solution on the interior of the primal-grid at a full time step is computed. Lastly, the approximate solution is found on the boundary of the primal-grid at a full time step in Algorithm 3.3.

Algorithm 3.1 1D Time Loop: First Half Time Step

```
1:  for i ← 1 to N − 1
2:  Compute c⃗:
3:      tcofs ← 0
4:      neighborData[0 : m] ← u[:, i − 1]
5:      neighborData[m + 1 : 2m + 1] ← u[:, i]
6:      tcofs[:, 0] ← Ã^{-1} neighborData
7:  Recursion Relation:
8:      for idt ← 1 to 2m + 2
9:          for idx ← 0 to 2m + 1 − idt
10:             tcofs[idx, idt] ← Δt/(h*idt) (idx + 1) tcofs[idx + 1, idt − 1]
11:         end for
12:     end for
13: Taylor Stepping:
14:     for idx ← 0 to m
15:         uh[idx, i − 1] ← tcofs[idx, 0]
16:         for l ← 1 to 2m + 2
17:             uh[idx, i − 1] ← uh[idx, i − 1] + 2^{-l} * tcofs[idx, l]
18:         end for
19:     end for
20: end for
```

In Algorithm 3.1, at each dual-grid point all coefficients of the approximating polynomial are found through the recursion relation and then the solution at the next half step is found via Taylor time stepping. For each grid point, the initial coefficients are computed on lines 3 though 6. Then the remaining coefficients are computed via the recursion relation on lines 8 through 12. Because of the scaling of the initial data, an additional factor of $\frac{1}{h}$ is needed in the recursion relation on line 10; see Appendix

B. Finally the Taylor stepping is done in lines 14 though 19. Since there is a factor of Δt applied at line 10, no such multiplication is need on line 17; thus, only powers of 2 need to be computed here. After stepping through each grid point, similarly to equation (3.3), uh now stores scaled data such that:

$$uh[j,i] \approx \frac{h^j}{j!}\frac{\partial^j}{\partial x^j} u\left(x_{i+\frac{1}{2}}, t_{\frac{1}{2}}\right), \qquad (3.4)$$
$$j = 0, 1, \ldots, 2,\ i = 0, 1, \ldots, N-1.$$

Algorithm 3.2 1D Time Loop: Second Half Time Step

```
21: t ← t + Δt/2
22: for i ← 1 to N − 2
23: Compute c̃⃗:
24:     tcofs ← 0
25:     neighborData[0 : m] ← uh[:, i − 1]
26:     neighborData[m + 1 : 2m + 1] ← uh[:, i]
27:     tcofs[:, 0] ← Ã^{−1} neighborData
28: Recursion Relation:
29:     for idt ← 1 to 2m + 2
30:         for idx ← 0 to 2m + 1 − idt
31:             tcofs[idx, idt] ← Δt/(h∗idt) (idx + 1) tcofs[idx + 1, idt − 1]
32:         end for
33:     end for
34: Taylor Stepping:
35:     for idx ← 0 to m
36:         u[idx, i] ← tcofs[idx, 0]
37:         for l ← 1 to 2m + 2
38:             u[idx, i] ← u[idx, i] + 2^{−l} ∗ tcofs[idx, l]
39:         end for
40:     end for
41: end for
```

In Algorithm 3.1, the solution at the interior points of the primal-grid are advanced. For each point, we again compute the initial coefficients, lines 24 through 27, then apply the recursion relation, lines 29 through 33, and do the Taylor time stepping, lines 35 through 41.

Algorithm 3.3 1D Time Loop: Boundary Conditions

42: *Compute* $\vec{\vec{c}}$:

43: $tcofs \leftarrow 0$

44: $neighborData[0:m] \leftarrow uh[:, N-2]$

45: $neighborData[m+1:2m+1] \leftarrow uh[:, 0]$

46: $tcofs[:, 0] \leftarrow \widetilde{A}^{-1}\, neighborData$

47: *Recursion Relation*:

48: **for** $idt \leftarrow 1$ **to** $2m+2$

49:     **for** $idx \leftarrow 0$ **to** $2m+1-idt$

50:         $tcofs[idx, idt] \leftarrow \frac{\Delta t}{h*idt}\,(idx+1)\, tcofs[idx+1, idt-1]$

51:     **end for**

52: **end for**

53: *Taylor Stepping*:

54: **for** $idx \leftarrow 0$ **to** m

55:     $u[idx, 0] \leftarrow tcofs[idx, 0]$

56:     **for** $l \leftarrow 1$ **to** $2m+2$

57:         $u[idx, 0] \leftarrow u[idx, 0] + 2^{-l} * tcofs[idx, l]$

58:     **end for**

59: **end for**

60: $u[:, N-1] \leftarrow u[:, 0]$

61: $t \leftarrow t + \frac{\Delta t}{2}$

In Algorithm 3.3 the periodic boundary conditions are used to compute $u[:, 0]$ and $u[:, N-1]$. We begin by computing the initial coefficients for the grid point x_0 on lines 43 through 46. Then apply the recursion relation for this point on lines 48 through 52, and Taylor stepping on lines 54 through 59. Finally we copy the results into $u[:, N-1]$. Note that $u[:, N-1]$ could be directly computed instead of $u[:, 0]$. All these steps are repeated until final time has been reached.

The above pseudocode does not represent the strict way in which the algorithms should be programmed. For instance, line 57 in Algorithm 3.3 would not evaluate 2^{-l} at each step, but rather a variable, initially set to 1.0 would be multiplied by 0.5 at the beginning of the **for** loop in line 54; then, this variable would be multiplied by $tcofs[idx, l]$ in line 57. This method of computing 2^{-l} is much faster, and will result in less round-off.

3.2 2D Parallel Implementation

There is little change from the serial implementation to this parallel implementation, which uses **parallel for** loops; furthermore, it is a straight forward change from the one- to two-dimensional implementations discussed. Thus, we skip any intermediate implementations and will discuss a two-dimensional parallel implementation in this section.

Consider the following problem:

$$\begin{aligned} &u_t = u_x + u_y,\ u\left(x, y, 0\right) = f\left(x, y\right),\ x \in \left[x_L, x_R\right],\ y \in \left[y_L, y_R\right], \\ &u\left(x_L, y, t\right) = u\left(x_R, y, t\right),\ u\left(x, y_L, t\right) = u\left(x, y_R, t\right),\ t \geq 0, \end{aligned} \tag{3.5}$$

where $f\left(x, y\right)$ is a smooth function.

For the two-dimensional case, the scaling is done in each direction. The resulting scaled data is stored in an array u, of dimension $(2m+2) \times (2m+2) \times N_x \times N_y$:

$$u[k, l, i, j] = \frac{h_x^k h_y^l}{(k!)\,(l!)} \frac{\partial^{k+l} f}{\partial x^k \partial y^l} \left(x_i, y_j\right). \tag{3.6}$$

We create matrices $\widetilde{A_x}^{-1}$ and $\widetilde{A_y}^{-1}$ using the same method as in the one-dimensional case; thus, there is a matrix created for each coordinate direction.

These matrices will later be used to compute the initial coefficients, just as in the one-dimensional case. See appendix A for an implementation for computing these matrices.

In Algorithm 3.4 the initial coefficients are computed. Then the recursion relation is evaluated by using either Algorithm 3.5 or Algorithm 3.6. Finally, Taylor time stepping is done in Algorithm 3.7.

For this example, assume that uh, u, $\widetilde{A_x}$, $\widetilde{A_y}$, m, q, and Δt are known globally across CPU cores, while all other variables are local to each CPU core.

Algorithm 3.4 2D Time Loop: Part 1

1: **parallel for** $j \leftarrow 1$ **to** $N_y - 1$
2: **for** $i \leftarrow 1$ **to** $N_x - 1$
3: *Compute Coefficients for Hermite Interpolation*:
4: $tcofs \leftarrow 0$
5: **for** $idy \leftarrow 0$ **to** m
6: $neighborData[0:m] \leftarrow u[:, idy, i-1, j-1]$
7: $neighborData[m+1:2m+1] \leftarrow u[:, idy, i, j-1]$
8: $smcofs[:, idy] \leftarrow \widetilde{A_x}^{-1}\, neighborData$
9: **end for**
10: **for** $idy \leftarrow 0$ **to** m
11: $neighborData[0:m] \leftarrow u[:, idy, i-1, j]$
12: $neighborData[m+1:2m+1] \leftarrow u[:, idy, i, j]$
13: $smcofs[:, m+1+idy] \leftarrow \widetilde{A_x}^{-1}\, neighborData$
14: **end for**
15: **for** $idx \leftarrow 0$ **to** $2m+1$
16: $scofs[idx, :] \leftarrow \widetilde{A_y}^{-1}(smcofs[idx, :])^T$
17: **end for**
18: $tcofs[:, :, 0] \leftarrow scofs$
19: *Recursion Relation*:
20: $\text{PDE}(tcofs, m, q, h_x, h_y, \Delta t)$

For Algorithm 3.4 a parallel **for** loop is used for each point on the dual-grid in the y direction. With y fixed, then a loop through the dual grid in the x direction is performed. This is one example of many possible domain decompositions for a shared memory implementation. For each grid point, lines 4 through 18 compute the initial coefficients. This is done by first fixing the derivative in y and computing the coefficients from the neighboring grid points in the x direction for the lower y neighbor

in lines 6 through 8. Once the y derivative is fixed, the coefficients are calculated as in the one-dimensional case. Once this has been done for all y derivatives, then the procedure is repeated for the neighbor data in x for the upper y neighbor in lines 11 through 13. Next we fix the x derivative and compute through all y derivatives in line 17. Once we have looped through all the x derivatives in this manner, the recursion relation can be computed in line 20.

Now there are multiple ways to approximate the PDE yielding an order of accuracy of $2m+1$. The first method, where $q = 2m+1$ but must take CFL$= \frac{\Delta t}{h} < 1$, requires fewer floating point operations for the recursion relation. Then the second method which uses $q = 4m+2$, but allows a CFL condition of 1.

Algorithm 3.5 2D PDE: Method 1

1: **function** PDE($tcofs, m, q, h_x, h_y, \Delta t$)
2: **for** $idt \leftarrow 1$ **to** q
3: **for** $idy \leftarrow 0$ **to** $2m+1-idt-1$
4: **for** $idx \leftarrow 0$ **to** $2m+1-idt-1$
5: $tcofs[idx, idy, idt] \leftarrow \frac{\Delta t}{h_x * idt}(idx+1)\, tcofs[idx+1, idy, idt-1]$
6: $+\frac{\Delta t}{h_y * idt}(idy+1)\, tcofs[idx, idy+1, idt-1]$
7: **end for**
8: **end for**
9: **end for**
10: **end function**

Algorithm 3.5, describing the first method, is analogous to the one-dimensional case in that the higher order terms are dropped as the number of derivatives in t are increased; thus, decreasing the number of floating point operations required. Since these terms are so small, an order of accuracy of $2m+1$ is still attained under the correct CFL condition.

Algorithm 3.6 2D PDE: Method 2

1: **function** PDE($tcofs, m, q, h_x, h_y, \Delta t$)
2: **for** $idt \leftarrow 1$ **to** q
3: **for** $idy \leftarrow 0$ **to** $2m+1$
4: **for** $idx \leftarrow 0$ **to** $2m$
5: $tcofs[idx, idy, idt] \leftarrow \frac{\Delta t}{h_x * idt}(idx+1)\,tcofs[idx+1, idy, idt-1]$
6: **end for**
7: **end for**
8: **for** $idy \leftarrow 0$ **to** $2m$
9: **for** $idx \leftarrow 0$ **to** $2m+1$
10: $tcofs[idx, idy, idt] \leftarrow tcofs[idx, idy, idt]$
11: $+\frac{\Delta t}{h_y * idt}(idy+1)\,tcofs[idx, idy+1, idt-1]$
12: **end for**
13: **end for**
14: **end for**
15: **end function**

For Algorithm 3.6, describing the second method, we include these higher order terms at the cost of a higher floating point operation count. If we choose to have the resulting polynomial, computed in lines 22 through 30, to have temporal order $q = 4m+2$, then a CFL condition of 1 can be used.

Algorithm 3.7 2D Time Loop: Part 2

```
21: Taylor Stepping:
22:         uh[:, :, i − 1, j − 1] ← tcofs[0 : m, 0 : m, 0]
23:         for l ← 1 to q
24:             for idx ← 0 to m
25:                 for idy ← 0 to m
26:                     uh[idx, idy, i − 1, j − 1] ← uh[idx, idy, i − 1, j − 1]
27:                                                  +2^(−l) ∗ tcofs[idx, idy, l]
28:                 end for
29:             end for
30:         end for
31:     end for
32: end parallel for
```

Once either process is completed for the entire dual-grid, then uh stores scaled data such that:

$$uh[k, l, i-1, j-1] = \frac{h_x^k h_y^l}{(k!)\,(l!)} \frac{\partial^{k+l} f}{\partial x^k \partial y^l} \left(x_{i-\frac{1}{2}}, y_{j-\frac{1}{2}}\right). \tag{3.7}$$

As in the one-dimensional case, this process is repeated for the interior of the primal-grid, and then periodic conditions are enforced at the boundaries in order to compute the approximation for the boundary points of the primal-grid. All the steps are then repeated until final time as been reached.

In this implementation, each processor works on a set of the spatial domain. In this case, working along the $x-$direction given a fixed y coordinate. This can be reversed, with each processor working with x fixed. Based on the specific domain at hand, this can be chosen to give optimal performance. For the case of distributed

memory, the domain should be broken down so that each node works on a rectangular section of the domain. These rectangles should be structured to reduce the amount of communication, so that latency can be negated. Computing the edges of each sub-domain can be done first, followed by the interior. If this is done, then the edge data can be sent to the neighboring nodes while the interior is computed. Furthermore, $\widetilde{A_x}$, $\widetilde{A_y}$, m, q, and Δt will reside locally on each CPU node, while u and uh should be distributed across the CPU nodes with some overlap. In the case of our single GPU implementation, each thread will work on a single grid point. Unless the number of grid points is large enough to be prohibitive, however, this is unlikely.

Chapter 4

CUDA

The CUDA parallel programming model is designed to be highly scalable because of the large range of GPUs available; of which, have varying number of multiprocessors, memory sizes, and memory partitions. The core of the model exposes three key abstractions to the programmer, thread groups, shared memories, and barrier synchronization. The programmer can then partition the problem into varying subproblems that can be computed independently in parallel by blocks of threads. Each block of threads has its own shared memory, and threads within these a block of threads can have barrier synchronizations. The usage of blocks of threads allows for automatic scalability because each block of threads can be scheduled on any available streaming multiprocessors on the GPU, so only the runtime system needs to know the number of streaming multiprocessors. This is quite the contrast to programming SSE instructions on the CPU by use of intrinsic functions, where the number of parallel instructions is fixed for each intrinsic function.

The CUDA programming model allows the programmer to write the code once and deploy it on a wide range of hardware, ranging from consumer level graphics cards found in laptops all the way up to the Tesla line of compute cards found in

supercomputers.

4.1 Host, Device, and Kernel

CUDA programming has its own terminology to identify hardware and software abstractions. The CPU in control of a GPU is called the **host**, and the GPU is called the **device**. A function which is called from the host, but runs on the device is called a **kernel**.

4.2 Threads, Blocks, and Grids

Execution of a specific kernel is performed by threads. A **thread** on a GPU is basically the same as a thread on a multi-core CPU, but there are generally many more of them. When calling a kernel, it must be specified how many threads will execute that kernel. This is done by specifying the number of threads in a **block**, and the number of blocks to be executed. Thread blocks allow threads within them to synchronize and use a shared memory space.

When a Cartesian topology is suitable for subdividing the computations, built-in functions that specify the number of blocks and threads in up to three dimensions can be used. This partitioning of blocks of threads is called a **grid**; Figure 4.1 illustrates this hierarchy. In this manner, a two-dimensional Cartesian nodal grid can be broken down so that each thread handles tasks on a single grid-point; for example, in our implementation of the Hermite method such a task could be stepping in time the approximate solution of a PDE. As a concrete example, assume a 120×120 node discretisation of a Cartesian spatial domain, where each thread time steps the solution on a single node. First, the number of threads in a block is typically chosen to be a

multiple of 32 (the size of a warp, see below), say 64, then a block of threads could work on, say, $8 \times 8 = 64$ nodal points (again, this type of decomposition is described in Figure 4.1.) Thus to cover the 120×120 nodes, the grid will be a 15×15 set of blocks. Note that if this example instead was using a 119×119 node discretisation, then all block and grid dimensions would remain the same; but more threads than there are nodes would be created. Thus, a conditional statement will need to be placed at the beginning of a kernel telling the threads not associated with a demonetization grid-point, to finish prior to modifying any data. Also note that other choices of the size of the blocks and grid can be made. This may impact the performance, and will be discussed in Chapter 5.

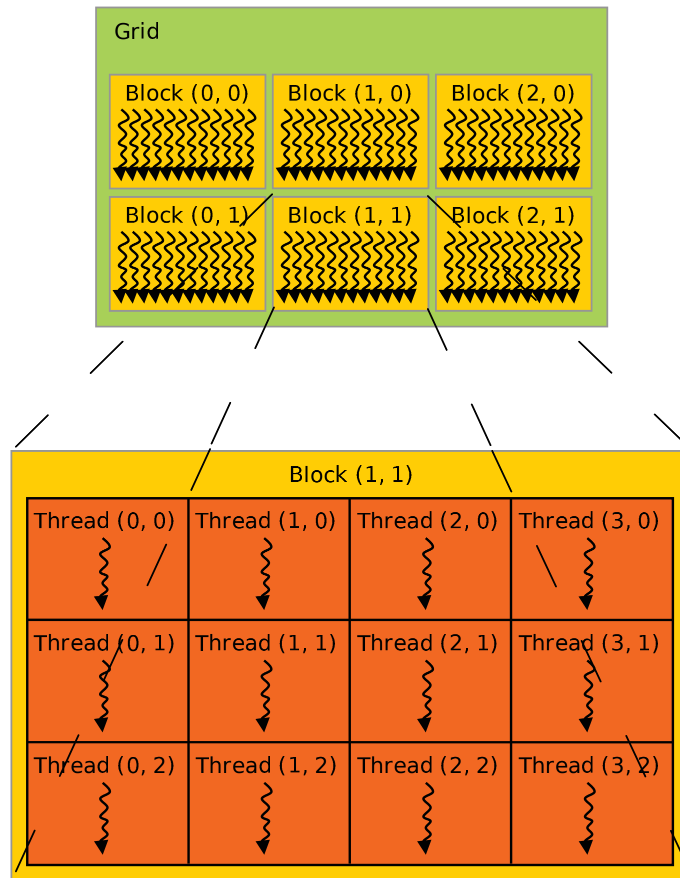

Figure 4.1: The CUDA thread hierarchy, from [3].

4.3 Warps

A block of threads is executed on only a single streaming multiprocessor, which means a sufficient number of blocks must be used in order to utilize all the streaming multiprocessors on a particular GPU (typically there are 1 to 15). At the hardware level, the streaming multiprocessor breaks the block of threads down into smaller chunks. These chunks of threads are called a **warp**. In the example with a block of size 8×8 threads, assuming that the warp size is 32, this would mean that there are two warps each with 32 threads. Each streaming multiprocessor has at least one **warp scheduler**, which is the hardware that decides when and how a warp will be executed. For example, in order to hide memory latency, the warp scheduler may switch execution to a different warp while waiting for a memory fetch.

4.3.1 Divergence

Each warp is executed in the manner of single instruction multiple thread (SIMT.) Because of the nature of SIMT, divergence within a warp, due to conditionals, can occur. Sample Code 4.3.1 illustrates the concept of divergence.

Sample Code 4.1: Example where divergence would likely occur.

```
1 if(var > 0)
2 {
3   var = -1.0*var + PI;
4 }
5 else if(var < 0)
6 {
7   var = var + PI;
8 }
9 else
```

```
10 {
11   var = 2*PI;
12 }
13
14 var = var*var;
```

Sample Code 4.1: Example where divergence would likely occur.

Assuming that Sample Code 4.3.1 has been translated into machine code and represents a segment of a kernel where the variable `var` would be different for each thread, then the following would occur within a warp. The threads within the warp that evaluate to true at line 1 execute line 3 while the remaining threads are idle; then, those that evaluated to true wait at line 13 to proceed. The remaining threads evaluate the conditional statement at line 5, those that return true would execute line 7; and then jump to line 13 to wait; all while the other threads are idle. The remaining threads, having evaluated the first two conditionals to false, execute line 11, and jump to line 13 to wait while the other threads are idle. The ordering of the execution of conditional statements, however, is not guaranteed by the current specification; thus, execution of line 11 could occur prior to execution of line 7. It is, however, guaranteed that once all threads have been rejoined at line 13, the warp continues synchronously; that is, line 14 is executed simultaneously by all threads in the warp.

Note that thread blocks are partitioned into warps in the block's x, then y, and then z direction; thus, the ordering is $(0,0,0), (1,0,0), \ldots, (0,1,0), (1,1,0), \ldots, (0,0,1), (1,0,1), \ldots$. Using this dimensional partitioning, performance can be gained by having warps access contiguous memory, or so that conditionals evaluate in a manner which reduces divergence within warps.

The order of warp execution within a block of threads can only be controlled with a barrier call. Such a barrier call guarantees that all warps within a block of threads have reached the barrier prior to proceeding. If synchronization is needed across multiple blocks, then multiple kernels will be needed. For example, in Hermite methods data dependency between the primal-grid and dual-grid necessitates the use of two kernels, one for the first half step and one for the second. By using multiple kernels, it is guaranteed that the first kernel will finish executing before the second begins. These two types of barriers are currently the only built-in methods for combating race conditions.

4.4 Memory Spaces

CUDA threads can access data from various memory spaces. All threads have access to the same global memory, even if the threads execute different kernels. On current GPUS, global memory is analogous to RAM on the CPU, and is located off chip. Global memory has high latency, so it may take 200 to 800 clock cycles to access data located in this memory space. Global memory accesses can be sped up in the case of read-only memory by using either **constant memory**, **read-only data cache**, or **texture memory**.

Typically, constant memory resides in a 64KB partition of device memory, and is accessed through an 8KB cache on each streaming multiprocessor. Data stored in constant memory is intended to be broadcast to all threads in a warp. Data residing in constant memory has the same scope as global memory.

Read-only data cache is used when the compiler can guarantee that the data is read-only for the lifetime of the kernel, for example when the `const` and `__restrict__` qualifiers are used. The hardware for read-only cache exists only on hardware with

compute capability[1] of 3.5 or newer. Since the data is guaranteed to only be read, the read-only cache does not need to check for coherency; thus, making it faster than a regular data cache. On devices which lack the hardware for read-only cache, using the qualifiers, `const` and `__restrict__`, still allows for compiler optimizations which can significantly improve performance.

Texture memory supports features, such as periodic boundaries, to make programming easier; unfortunately, it does not support double precision and can thus not be used for computations. The texture memory is, however, useful for lower precision real-time visualization of a solution. At this time, texture memory does not have a separate physical memory location, but rather allows the compiler to make specialized optimization. In particular, texture memory optimizes for two-dimensional spatial locality; such as accessing neighboring points on a two-dimensional grid.

In order to help combat latency, CUDA supports shared memory spaces. Shared memory is accessible by threads within a single block, and has the same lifetime as the thread block. The size of this shared memory is dependent on the compute capability of the GPU, but is always stored sequentially in 32 memory banks; one for each thread in a warp. Shared memory resides on chip, and is separate from global memory. Due to this locality, data in shared memory takes as little as 1 clock cycle to access when no memory bank conflicts occur. A bank conflict occurs if multiple threads in a warp access different memory addresses within the same bank; thus, no bank conflict would occur if all threads in a warp accessed the same memory address. If bank conflicts do occur, then the worst case scenario is $Warp\,Size \times Number\,of\,Variables\,Accessed$ clock cycles to access all data for the warp. However, this is extremely unlikely to occur, as this scenario corresponds to when all threads in a warp are accessing different memory addresses within the same bank. Under normal circumstances, it generally only takes 1-4 clock cycles for accesses to shared memory.

[1] Compute capability is the hardware versioning used for CUDA capable devices to identify available feature sets.

The programmer has direct access to manage shared memeory, and may programmatically move data into this space. On GPUs with compute capability 2.0 or newer, the shared memory hardware can also be used as an L1 cache via either the `cudaDeviceSetCacheConfig` or `cudaFuncSetCacheConfig` commands. Using the shared memory as an L1 cache allows for increased performance, without the need for designing and programming complex memory topologies.

Additionally, each thread has its own private local memory space, which lasts the life of the thread. The programmer does not have access directly to how this memory is handled; however, the run-time driver will attempt to keep data in private local memory stored in registers. Thus, local thread level memory is very fast. Figure 4.2 illustrates the memory hierarchy of all the memory spaces discussed.

Due to widely varying attributes of the memory spaces available, efficient use of the different memory spaces are critical for the performance of the numerical solver. This will be discussed in the context of Hermite methods later in this chapter.

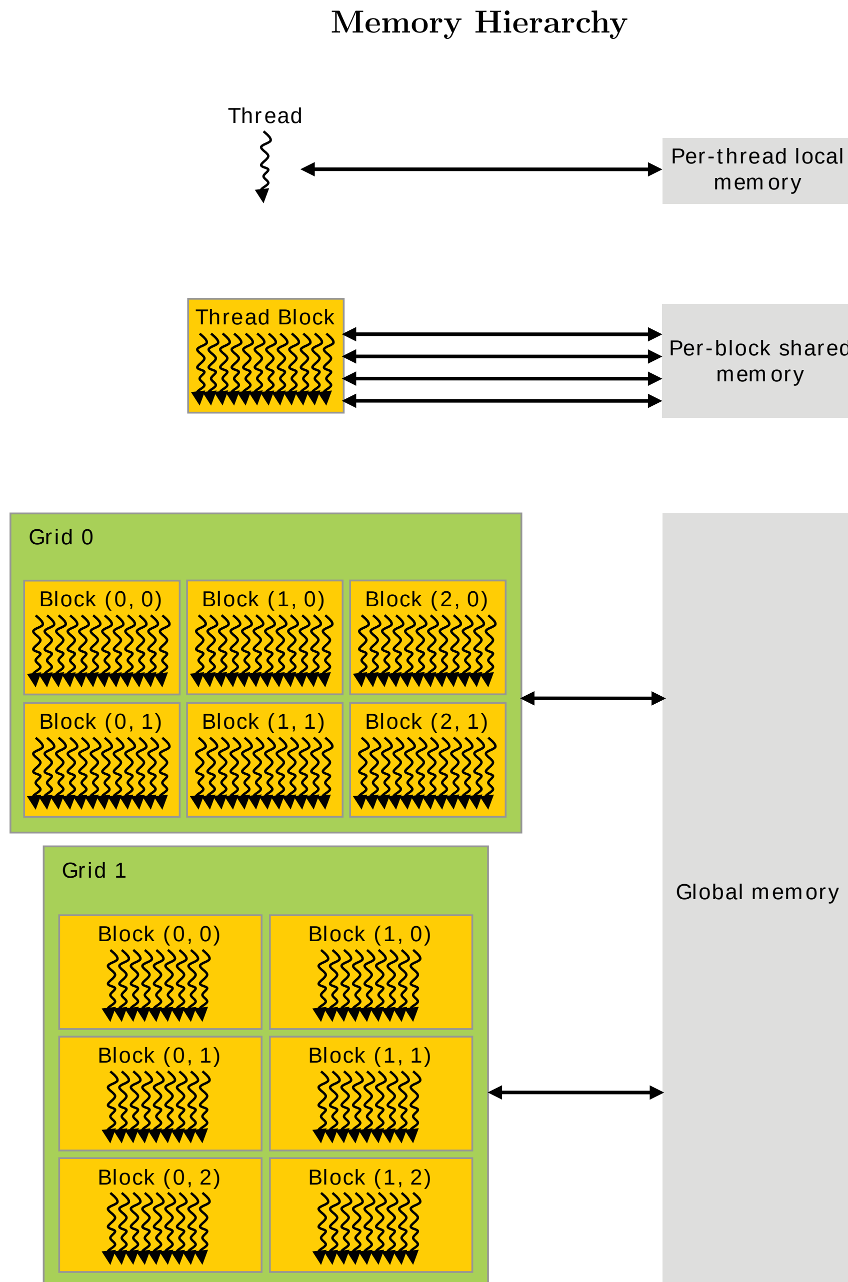

Figure 4.2: The CUDA memeory hierarchy, from [3].

4.5 Special Performance Metrics

Beyond the standard performance metrics of memory bandwidth, compute time, and floating-point operations per second, CUDA also has the metric **occupancy**. Occupancy is the ratio of active warps on a single streaming multiprocessor to the maximum number of warps supported. When occupancy is high, the warp scheduler is able to switch between more warps in order to hide memory access latency. This metric is so important that NVIDIA has an occupancy calculator available on their website, see [3]. However, maximizing theoretical occupancy does not always increase performance.

For memory optimization, the compiler flag `-Xptxas -v` allows the programmer to determine how many registers are used per thread. This means a judgment can be made as to if the number registers per thread is too high. Note that using too many registers can cause some cores to idle. Checking the specification for the compute capability of the hardware may help to determine the number of register to be used. Additionally, running the CUDA profiler can help determine if too many registers are being used. If this is the case, then the number of registers used per thread can be limited with the `-maxrregcount` compiler flag. Reducing the number of threads per block can also lower the total number of registers in use.

In addition to these special metrics, the CUDA profiler allows for detailed performance analysis of all standard metrics. Since performance metrics are important for optimizing kernels, accuracy of these build-in performance metrics can be increased by using special calls within kernels. For more information on the performance metrics available, and how to use them, see [3].

4.6 CUDA Implementation of Hermite Methods

The CUDA implementation, used here, changes little from the two-dimensional parallel implementation discussed in Section 3.2. The primary difference being that instead of using parallel loops over coordinate directions, we parallelize the method in a domain decomposition fashion where each thread updates the approximate solution at a single nodal point. A main concern when designing a CUDA implementation, is what memory space to use for different variables in order to maximize performance. The below choices were made from theoretical considerations based on existing CUDA literature, together with empirical experience gained during the project.

As in Section 3.2, the method is broken down into a first and second half time step. In all kernels, u and uh are stored in global memory. The arrays for the coefficients, $smcofs$, $scofs$, and $tcofs$, are also stored in global memory as they are generally to large to be stored in registers for each thread. In Section 3.2 $smcofs$, $scofs$, and $tcofs$ where stored locally for each CPU core, so $smcofs$ and $scofs$ were of size $(2m+2) \times (2m+2)$ and $tcofs$ was of size $(2m+2) \times (2m+2) \times (q+1)$. For the GPU implementation each thread needs its own space, so $smcofs$ and $scofs$ are of size $(2m+2) \times (2m+2) \times N_x \times N_y$ with $scofs$ of size $(2m+2) \times (2m+2) \times (q+1) \times N_x \times N_y$. No substantial changes are made to the remaining variables.

Both $\widetilde{A_x}^{-1}$ and $\widetilde{A_y}^{-1}$ are stored in global memory, as it is sub-optimal to recreate these matrices at each half time step. And since these matrices do not change, they can make use of the read-only data cache. For the first half time step, u is not altered in any way; thus, u can use the read-only data cache. Similarly for the second half time step, uh can use the read-only data cache.

The constants m, q, h_x, h_y, and Δt are all stored in constant memory, this way each streaming multiprocessor has a copy stored locally; furthermore, accesses to these constants occur at the same time within a warp. Thus, these constants are

broadcast throughout a warp.

Variables for looping are stored locally for each thread in registers. Similarly, the node identifier (i, j) is also stored in registers on a per thread basis.

Algorithm 4.1 uses colored pseudocode to illustrate the above implementation concepts. Note that $neighborData$, lines 5 through 7, is not actually created and stored, as the array u is operated on directly. The color coding used in Algorithm 4.1 is as follows: Global Memory, Read-Only Cache, Constant Memory, and Register.

Algorithm 4.1 CUDA: First Half Time Step

1: **for each** $i \leftarrow 1$ **to** $N_x - 1$ and $j \leftarrow 1$ **to** $N_y - 1$
2: *Compute Coefficients for Hermite Interpolation*:
3: $tcofs[:,:,i,j] \leftarrow 0$
4: **for** $idy \leftarrow 0$ **to** m
5: $\quad neighborData[0:m] \leftarrow u[:,idy,i-1,j-1]$
6: $\quad neighborData[m+1:2m+1] \leftarrow u[:,idy,i,j-1]$
7: $\quad smcofs[:,idy,i,j] \leftarrow \widetilde{A_x}^{-1}\ neighborData$
8: **end for**
9: **for** $idy \leftarrow 0$ **to** m
10: $\quad neighborData[0:m] \leftarrow u[:,idy,i-1,j]$
11: $\quad neighborData[m+1:2m+1] \leftarrow u[:,idy,i,j]$
12: $\quad smcofs[:,m+1+idy,i,j] \leftarrow \widetilde{A_x}^{-1}\ neighborData$
13: **end for**
14: **for** $idx \leftarrow 0$ **to** $2m+1$
15: $\quad scofs[idx,:,i,j] \leftarrow \widetilde{A_y}^{-1}(smcofs[idx,:])^T$
16: **end for**
17: $tcofs[:,:,0,i,j] \leftarrow scofs$
18: *Recursion Relation*:
19: $\text{PDE}(tcofs[:,:,:,i,j], m, q, h_x, h_y, \Delta t)$
20: *Taylor Stepping*:
21: $uh[:,:,i-1,j-1] \leftarrow tcofs[0:m,0:m,0,i,j]$
22: **for** $l \leftarrow 1$ **to** q
23: $\quad$ **for** $idx \leftarrow 0$ **to** m
24: $\quad\quad$ **for** $idy \leftarrow 0$ **to** m
25: $\quad\quad\quad uh[idx,idy,i-1,j-1] \leftarrow uh[idx,idy,i-1,j-1]$
26: $\quad\quad\quad\quad + 2^{-l} * tcofs[idx,idy,l,i,j]$
27: $\quad\quad$ **end for**
28: $\quad$ **end for**
29: **end for**

While some of the variables stored in global memory can be transferred into shared memory to boost performance, the recursion relation, which operates on $tcofs$, does take the vast majority of computational time. Since accesses to $tcofs$ are dependent on the specific PDE being solved, shared memory is used as an L1 cache. Alternatively, although not explored here, performance can be boosted by designing a shared memory topology which takes advantage of the memory access pattern to $tcofs$ for a specific PDE.

It can be seen from the pseudocode in Section 3.2 that the Hermite method exhibits very little divergence, as there is only one conditional required for the core algorithm. Updates on the dual-grid do not have divergence, and updates on the primal-grid have no divergence on the interior of the grid. The only conditional is for the periodic boundaries, which could be handled by a separate kernel. In general, if the PDE being solved requires conditionals, then it is best to decompose the problem and spatial domain in a manner that reduces or eliminates warp divergence.

Chapter 5

Numerical Examples

In this chapter we will use numerical experiments to illustrate some of the concepts from Chapter 4, and study the performance of our CUDA implementation of the Hermite method on a NVIDIA GPU. For these examples, the code was run on an NVIDIA GTX 560 Ti (compute capability 2.1) with CUDA driver version of 6.0 and run-time version 5.5. The card has 1,024MB of global memory, 65,536 bytes of constant memory, 49,152 bytes of shared memory per block, 524,288 bytes of L2 Cache, and supports 32,768 registers per block. There are 384 CUDA Cores distributed across 8 streaming multiprocessors, for a total of 48 cores per streaming multiprocessor. The GPU is clocked at 1800 Mhz, and the memory is clocked at 2106 Mhz with a bus width of 256 bits. The maximum number of threads per block is 1024, while the maximum number of threads per streaming multiprocessor is 1536. The NVIDIA GTX 560 Ti was parred with an Intel Core i7-2600k clocked at 3.4 GHz, with 8GB of RAM clocked at 1600 Mhz. All utilizing a motherboard with the Intel P67 chipset.

All of the examples in this chapter solve the following problem:

$$
\begin{aligned}
&u_t = u_x + u_y,\ u(x,y,0) = f(x,y)\,,\ x \in [-8,8]\,,\ y \in [-8,8]\,,\\
&u(-8,y,t) = u(8,y,t)\,,\ u(x,-8,t) = u(x,8,t)\,,\ t \geq 0, \qquad (5.1)\\
&f(x,y) = e^{\frac{-\left(x^2+y^2\right)}{2}}.
\end{aligned}
$$

The approximate solution was advanced to time $t = 16$. Since this is one time period, error is calculated by taking the maximum norm of the difference between the initial data, $f(x,y)$, and the computed solution at end time (numerical round off will generally begin to take over once the error is of approximately 10^{-14}.) The nodal discretization of the spatial domain varies from a 5×5 node Cartesian grid to a 160×160 node Cartesian grid. Unless otherwise noted, a CFL condition of 0.1 is used. Compute time is taken as the start of the time stepping process to the end of the time stepping process, as the remaining time is spent on initialization; which will remain approximately constant. Times were taken with CUDA's build-in event timer, which gives results at a resolution of 0.5µs.

5.1 Convergence Study

To ensure that the implementation is correct we performed a convergence study for $m = 3$ and $m = 4$, using $q = 4m+2$ which allows for a CFL of 0.9. In Figures 5.1 and 5.2 the maximum error at time $t = 16$ is plotted as a function of the grid-spacing. It can be seen from these figures that convergence occurs at the expected rate of $2m+1$. Additonally for all the performance experiments below, we also checked that the rates of convergence were as expected for all m. The errors, along with the approximate solutions at end time, were also compared with other serial implementations. From this data, it is reasonable to assume that the code used for these examples has yielded valid results.

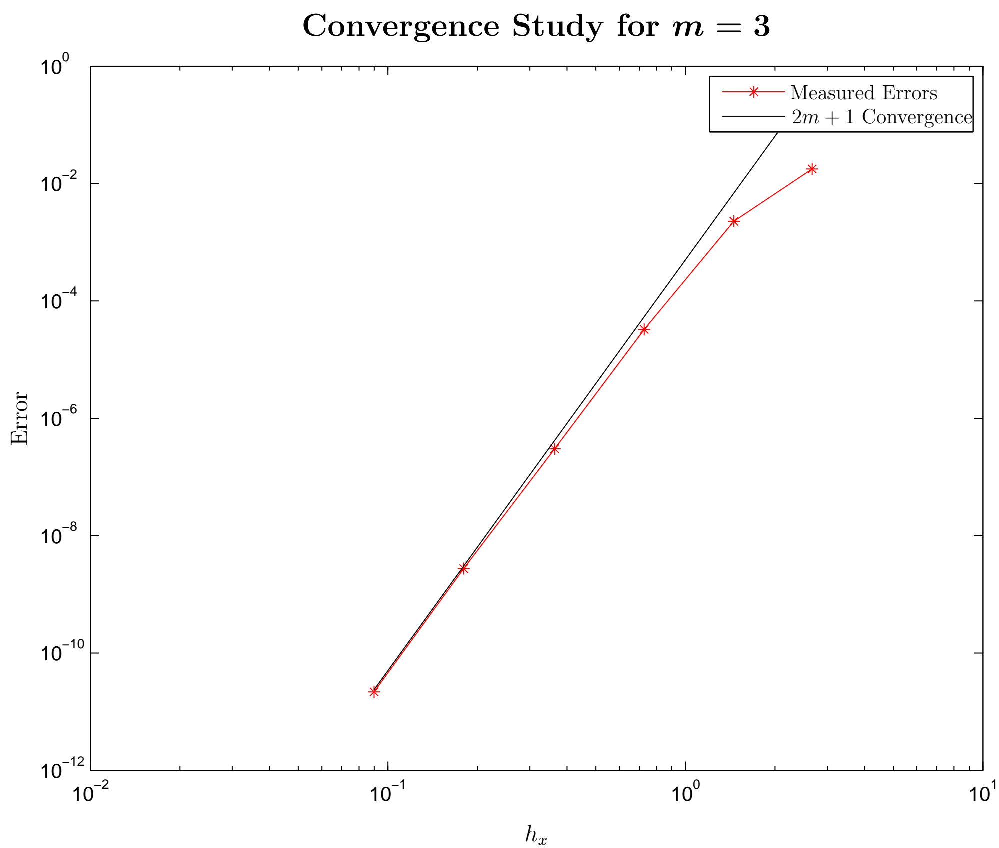

Figure 5.1: Error convergence when $m = 3$, $q = 4m + 2$, and a CFL of 0.9.

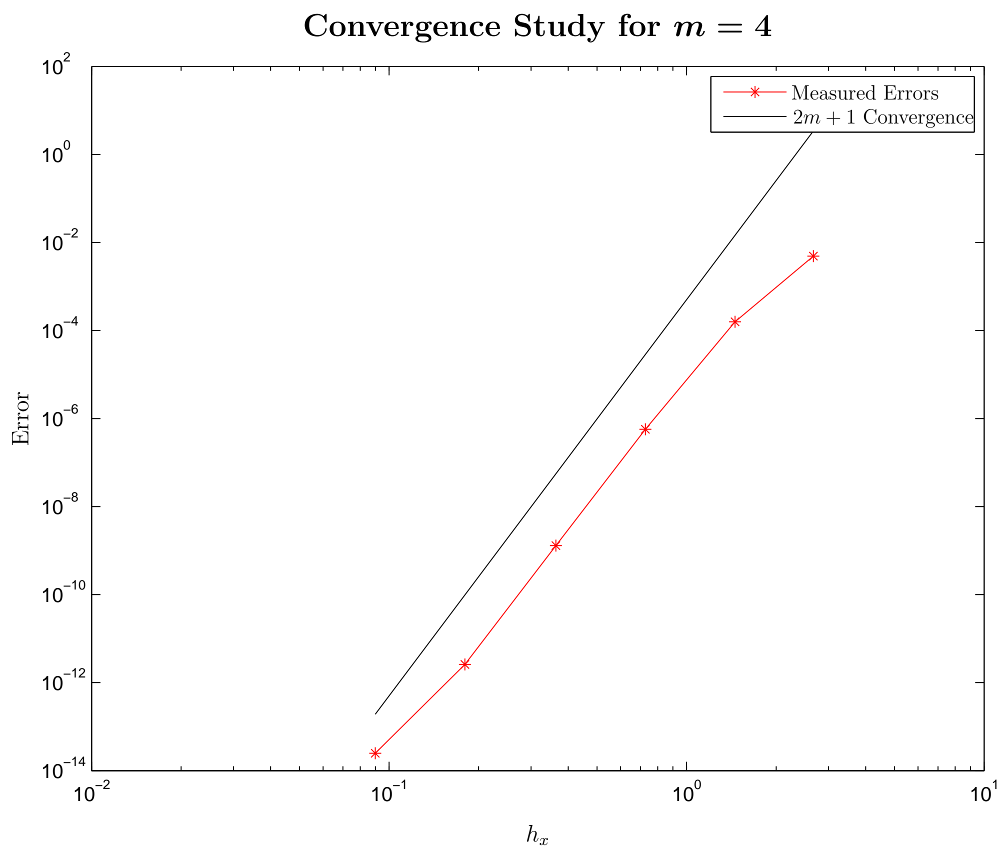

Figure 5.2: Error convergence when $m = 4$, $q = 4m + 2$, and a CFL of 0.9.

5.2 Performance Studies

In the below examples, a comparison of error vs compute time is presented. Error decreases due to a decrease in the spatial step size, that is, with increasing number of nodes on which the approximation is computed.

5.2.1 Example 1: Performance at Maximal Number of Threads Per Block

For this first example, a fixed block size of 1024 is used. This is the maximum block size allowed by the hardware, and it is fairly common that new CUDA programmers will select the maximal size. Such a choice is due to the fact that the more active threads there are, the more memory access latency can be hidden. However for smaller grids, this choice of maximal block size does not utilize all of the streaming multiprocessors. Figure 5.3 shows how this causes little difference in the ratio of error to compute time when m is changed. The use of only one streaming multiprocessor also exacerbates the problem of latency, because as m becomes larger so does the amount of memory used; thus, a smaller fraction of the data fits into cache while the stride of memory accesses also becomes larger. Figure 5.4 depicts results for a much higher sampling rate, in the number of nodes, in order to give a better understanding of what is occurring. At an error of order 10^{-4} and compute time on the order of 10^0, the number of streaming multiprocessors, for $m = 3$, increases from one to four. Thus, compute time decreases substantially while the error also decreases. This phenomena can also be seen for $m = 4$ at an error of order 10^{-6}. These scaling properties continue as the number of nodes increases, but becomes less prevalent as more streaming multiprocessors are being used. Prior to utilizing a sufficient number streaming multiprocessors, the scaling is analogous to weak scaling on the CPU. Once all streaming multiprocessors are being used, however, the scaling becomes closer to

that of strong scaling.

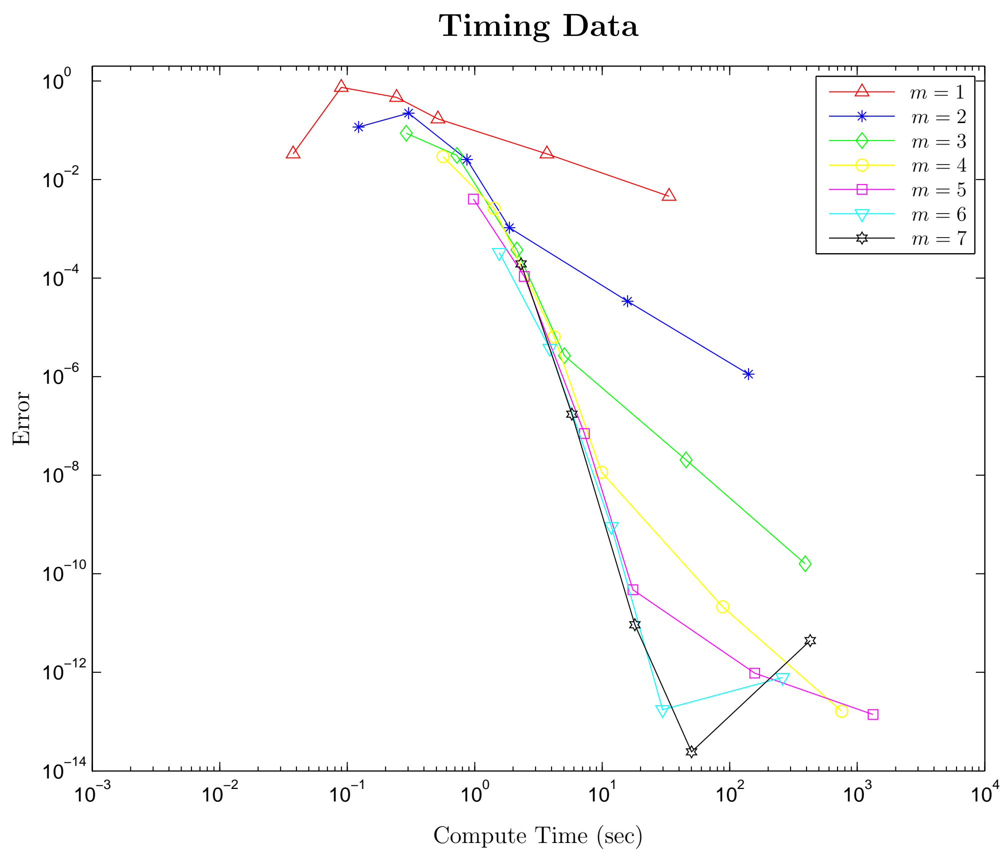

Figure 5.3: Performance with a fixed block size of 1024, $q = 2m + 1$, and a CFL of 0.1.

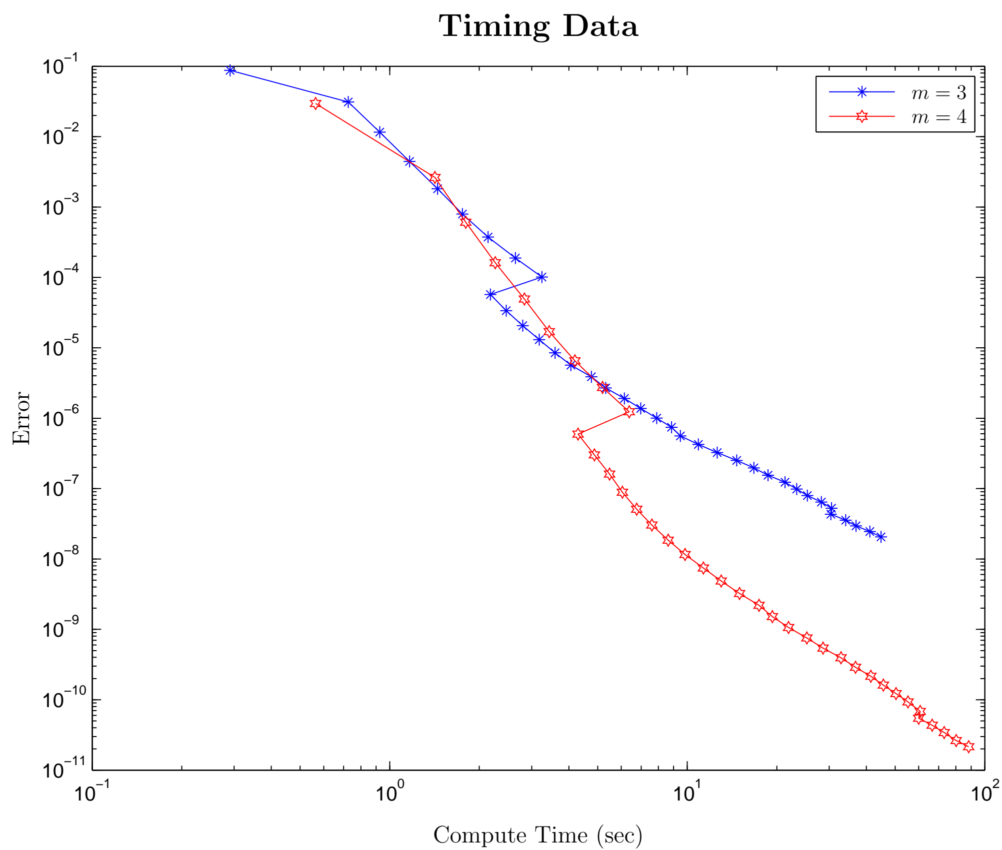

Figure 5.4: Performance with a fixed block size of 1024, $q = 2m + 1$, and a CFL of 0.1.

5.2.2 Example 2: Performance at 64 Threads Per Block

In this example a block size of 64 is used. This block size yields a theoretical occupancy of 33.3% for the NVIDIA GTX 560 Ti, but exhibits better scaling characteristics than the previous example. Figure 5.5 illustrates how the scaling is much smoother than before, this is due to the more even distribution of threads across the GPU's streaming multiprocessors. The switch from a weak to strong scaling now occurs at an error of order 10^{-5} for $m = 3$, for $m = 4$ this occurs at an error on the order of 10^{-7}. Figure 5.6 illustrates the effect of a smaller block size for m from 1 through 7. While increasing m by 1 does not really increase performance when using very few nodes, a performance difference can be seen for fewer nodes than in example 1.

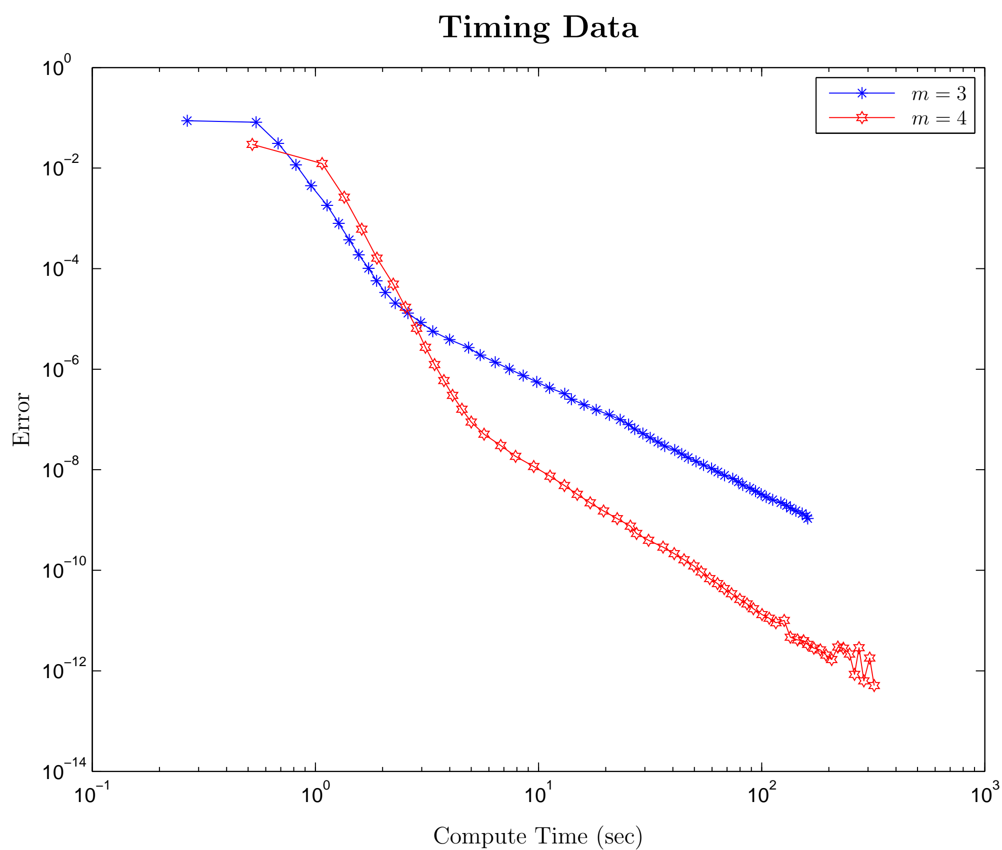

Figure 5.5: Performance with a fixed block size of 64, $q = 2m + 1$, and a CFL of 0.1.

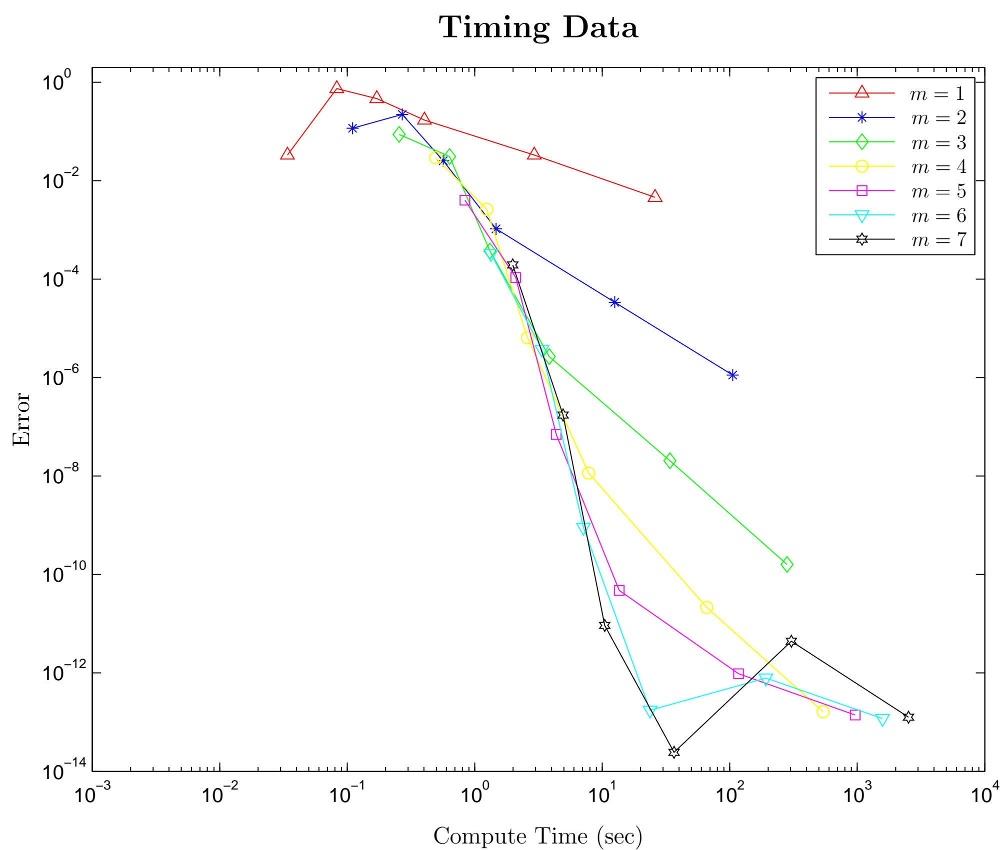

Figure 5.6: Performance with a fixed block size of 64, $q = 2m + 1$, and a CFL of 0.1.

5.2.3 Example 3: Performance with Maximal q

In this example we examine the difference in performance when using the PDE method from Algorithm 3.6 along with $q = 4m+2$. This allows for the CFL condition to be set to 0.9, which reduces the number of time stepping calls needed. In turn reducing the overhead of the host calling kernels on the device. These changes make for a fairly significant speedup from the PDE method from Algorithm 3.5, $q = 2m+1$, and a CFL condition of 0.1 used in the previous examples. In fact, the change is large enough that $m = 6$ gives better performance than $m = 7$; since for $m = 7$ round-off takes over so quickly. These results are in contrast to the CPU, where it actually takes longer to execute using this method. These results, in particular, show that the architectural differences between the CPU and GPU must be accounted for when implementing numerical methods on the GPU.

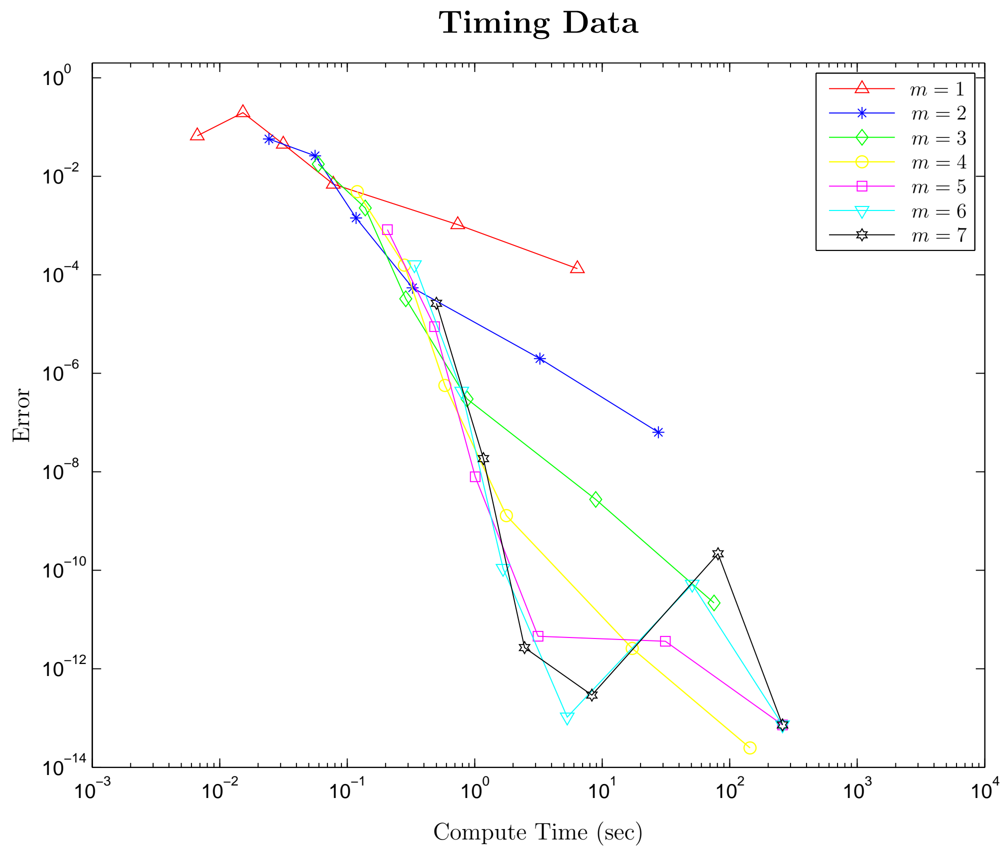

Figure 5.7: Performance with a fixed block size of 64, $q = 4m + 2$, and CFL of 0.9.

Chapter 6

Summary and Outlook

To summarize, we have considered how scalar and systems of hyperbolic equations can be approximated using Hermite methods in one- and two-dimensions. In particular this thesis has investigated different implementations on traditional CPU based architectures, as well as more modern GPU based accelerators.

It is very important to consider the architectural differences between CPUs and GPUs, and how these differences affect performance. For instance, a GPU may see a decrease in compute time in spite of an increase in floating point operations, which is rare on the CPU. Furthermore, the GPU has many more memory spaces to consider than the CPU, and in fact, perhaps the largest performance factor for computations performed on a GPU is memory usage. Thus, it is important to make sure each variable, or constant, is using the appropriate memory space on the GPU.

Choosing the correct number of threads in a block is important for utilizing all of the streaming multiprocessors available on a GPU. A poor choice of block size may limit the number active of streaming multiprocessors, which greatly reduces parallelism; for instance, the NVIDIA GTX Titan Black has 192 cores on each of its 15 streaming multiprocessors. In addition to reducing the number of active streaming

multiprocessors, a poorly chosen block size can limit the size of a warp; which also reduces the parallelism achieved.

There are many other attributes to explore, since this is the first implementation of the Hermite method on GPUs, as far as we know. In the future we would like to explore more memory topologies that are independent of the PDE in order to find whether or not performance can be increased in this manner for the general case; particularly, on GPUs with large memory buses. Additionally, exploration of other domain decomposition may yield increased performance. I particular where the spatial domain of each block of threads overlaps with its neighbors and that sub-domain of the array u is stored in shared memory, so that each block can advance several time steps before synchronization is needed; thus, reducing global memory accesses.

In conclusion, Hermite methods do in fact work very well on GPUs, particularly when the performance considerations discussed in thesis are implemented. But since little is yet known, there is still much more to explore.

Appendices

A Creation of $\widetilde{A}^{-1}$ **55**

B Notes on Scaling **61**

Appendix A

Creation of $\widetilde{A}^{-1}$

In order to create the matrix $\widetilde{A}^{-1}$, we need to know x_i, x_{i+1}, and $x_{i+\frac{1}{2}}$ for some i. This allows the case when $x_{i+\frac{1}{2}}$ is not exactly halfway between x_i and x_{i+1}. Here we denote x_i by `xl`, x_{i+1} by `xr`, and $x_{i+\frac{1}{2}}$ by `xc`. The variable `bcofs` is an array containing the binomial coefficients up to m, while $\widetilde{A}^{-1}$ is stored in `tmat`.

Sample Code A.1: Creation of the matrix $\widetilde{A}^{-1}$.

```
1  void hermiteMatrix(double *tmat, const int &m, const double
      &xl, const double &xr, const double &xc, const double &
      icase)
2  {
3    /*
4    icase < 0 => xc=xl (left boundary case)
5    icase = 0 => xl < xc < xr
6    icase > 0 => xc=xr (right boundary case)
7    */
8    int twoMplusTwo = 2*m+2;
9    int mPlusOne = m + 1;
10   double sign = 1.0;
```

```
11  double *bcofs = (double *)malloc(sizeof(double)*(m+2)*(m
       +2));
12  int bcofsOffset = m+2;
13  double h = xr-xl;
14  double z = (xc-xl)/h;
15  double zc = z-1.0;
16
17  if (icase > 0)
18  {
19    z=1.0;
20    zc=0.0;
21  }
22  else if (icase < 0)
23  {
24    z=0.0;
25    zc=-1.0;
26  }
27
28  bcofs[0]=1.0;
29
30  for(int i = 1; i < bcofsOffset; i++)
31  {
32    bcofs[bcofsOffset*i]=1.0;
33    for(int j = 1; j < (i+1); j++)
34    {
35      bcofs[bcofsOffset*i + j]=bcofs[bcofsOffset*i + (j-1)
          ]*(i-j+1)/(j);
36    }
37  }
38
```

```
39  double adl, adr, c1l, c1r, c2l, c2r;
40  int jStart, jBound;
41  for(int i=0; i < twoMplusTwo; i++)
42  {
43    adl=0.0;
44    adr=0.0;
45    jStart = std::max(0,i-m);
46    jBound = std::min(i+1, m+2);
47    for(int j = jStart; j < jBound; j++)
48    {
49      if((m-i+j) == 0 )
50      {
51        c2l = 1.0;
52        c2r = 1.0;
53      }
54      else
55      {
56        c2l = pow(z, (m-i+j));
57        c2r = pow(zc, (m-i+j));
58      }
59
60      if((m+1-j) == 0)
61      {
62        c1l = 1.0;
63        c1r = 1.0;
64      }
65      else
66      {
67        c1l = pow(zc, (mPlusOne-j));
68        c1r = pow(z, (mPlusOne-j));
```

```
69          }
70
71          adr=adr+bcofs[bcofsOffset*(m+1) + j]*bcofs[bcofsOffset
                *m + (i-j)]*c1r*c2r;
72          adl=adl+bcofs[bcofsOffset*(m+1) + j]*bcofs[bcofsOffset
                *m + (i-j)]*c1l*c2l;
73        }
74        tmat[i + (2*m+1)*twoMplusTwo]=adr;
75        tmat[i + m*twoMplusTwo]=(pow((-1.0), (m+1)))*adl;
76      }
77
78      // Now loop over the other columns backwards
79
80      for(int k = m-1; k > -1; k--)
81      {
82        for(int i = 0; i < twoMplusTwo; i++)
83        {
84          adl=0.0;
85          adr=0.0;
86          jStart = std::max(0,i-k);
87          jBound = std::min(i+1, m+2);
88          for(int j = jStart; j < jBound; j++)
89          {
90            if((k-i+j) == 0 )
91            {
92              c2l = 1.0;
93              c2r = 1.0;
94            }
95            else
96            {
```

```
97            c2l = pow(z, (k-i+j));
98            c2r = pow(zc, (k-i+j));
99          }
100
101         if((m+1-j) == 0)
102         {
103           c1l = 1.0;
104           c1r = 1.0;
105         }
106         else
107         {
108           c1l = pow(zc, (mPlusOne-j));
109           c1r = pow(z, (mPlusOne-j));
110         }
111
112         adr=adr+bcofs[bcofsOffset*(m+1) + j]*bcofs[
              bcofsOffset*k + (i-j)]*c1r*c2r;
113         adl=adl+bcofs[bcofsOffset*(m+1) + j]*bcofs[
              bcofsOffset*k + (i-j)]*c1l*c2l;
114       }
115
116       tmat[i + (k+m+1)*twoMplusTwo]=adr;
117       tmat[i + k*twoMplusTwo]=(pow((-1.0), (m+1)))*adl;
118       sign=1.0;
119       for(int j = k+1; j < mPlusOne; j++)
120       {
121         sign=-sign;
122         tmat[i + k*twoMplusTwo]=tmat[i + k*twoMplusTwo]-sign
              *bcofs[bcofsOffset*(m+1) + (j-k)]*tmat[i + j*
              twoMplusTwo];
```

```
123          tmat[i + (k+m+1)*twoMplusTwo]=tmat[i + (k+m+1)*
                 twoMplusTwo]-bcofs[bcofsOffset*(m+1) + (j-k)]*
                 tmat[i + (j+m+1)*twoMplusTwo];
124        }
125      }
126    }
127
128    free(bcofs);
129  }
```

Sample Code A.1: Creation of the matrix $\widetilde{A}^{-1}$.

Appendix B

Notes on Scaling

Let us examine the computation of coefficients from the initial data, which breaks down into the following set of equations:

$$\frac{\partial^k p_{i+\frac{1}{2}}\left(x_i, 0\right)}{\partial x^k} = \frac{\partial^k f\left(x_i\right)}{\partial x^k}, \tag{B.1}$$

$$\frac{\partial^k p_{i+\frac{1}{2}}\left(x_{i+1}, 0\right)}{\partial x^k} = \frac{\partial^k f\left(x_{i+1}\right)}{\partial x^k}, \tag{B.2}$$

$$k = 0, 1, \ldots, m,$$

where

$$\frac{\partial^k p_{i+\frac{1}{2}}\left(x_i, 0\right)}{\partial x^k} = \frac{\partial^k}{\partial x^k} \sum_{j=0}^{2m+1} c_{j0} \left(x_i - x_{i+\frac{1}{2}}\right)^j, \tag{B.3}$$

$$\frac{\partial^k p_{i+\frac{1}{2}}\left(x_{i+1}, 0\right)}{\partial x^k} = \frac{\partial^k}{\partial x^k} \sum_{j=0}^{2m+1} c_{j0} \left(x_{i+1} - x_{i+\frac{1}{2}}\right)^j. \tag{B.4}$$

To further understand this, let us first make the following definitions where $A \in \mathbb{R}^{(2m+2)\times(2m+2)}$, $\vec{c} \in \mathbb{R}^{2m+2}$, and $\vec{f} \in \mathbb{R}^{2m+2}$:

$$A = \begin{bmatrix} 0! & \frac{-h}{2} & \frac{h^2}{2^2} & \frac{-h^3}{2^3} & \frac{h^4}{2^4} & \cdots & \frac{-h^{2m+1}}{2^{2m+1}} \\ 0 & 1! & \frac{-2h}{2} & \frac{3h^2}{2^2} & \frac{-4h^3}{2^3} & \cdots & \frac{(2m+1)h^{2m}}{2^{2m}} \\ \vdots & \ddots & \ddots & \ddots & \cdots & \cdots & \vdots \\ 0 & \cdots & 0 & m! & \frac{-(m+1)!\,h}{2} & \cdots & \frac{-(2m+1)!\,h^{m+1}}{(m+1)!\,2^{m+1}} \\ 0! & \frac{h}{2} & \frac{h^2}{2^2} & \frac{h^3}{2^3} & \frac{h^4}{2^4} & \cdots & \frac{h^{2m+1}}{2^{2m+1}} \\ 0 & 1! & \frac{2h}{2} & \frac{3h^2}{2^2} & \frac{4h^3}{2^3} & \cdots & \frac{(2m+1)h^{2m}}{2^{2m}} \\ \vdots & \ddots & \ddots & \ddots & \cdots & \cdots & \vdots \\ 0 & \cdots & 0 & m! & \frac{(m+1)!\,h}{2} & \cdots & \frac{(2m+1)!\,h^{m+1}}{(m+1)!\,2^{m+1}} \end{bmatrix}, \tag{B.5}$$

$$\vec{c} = \begin{bmatrix} c_{00} & c_{10} & c_{20} & \cdots & c_{2m+1,0} \end{bmatrix}^T, \tag{B.6}$$

$$\vec{f} = \begin{bmatrix} f(x_i) \\ \frac{\partial f(x_i)}{\partial x} \\ \frac{\partial^2 f(x_i)}{\partial x^2} \\ \vdots \\ \frac{\partial^m f(x_i)}{\partial x^m} \\ f(x_{i+1}) \\ \frac{\partial f(x_{i+1})}{\partial x} \\ \frac{\partial^2 f(x_{i+1})}{\partial x^2} \\ \vdots \\ \frac{\partial^m f(x_{i+1})}{\partial x^m} \end{bmatrix}. \tag{B.7}$$

With these definitions, the problem amounts to solving $A\vec{c} = \vec{f}$. However, even for relatively simple cases, A is not well conditioned. Now let us apply the following scaling to $\vec{c}$ and $\vec{f}$:

$$\tilde{\vec{c}} = \begin{bmatrix} c_{00} & hc_{10} & h^2c_{20} & \cdots & h^{2m+1}c_{2m+1,0} \end{bmatrix}^T, \tag{B.8}$$

$$\tilde{\vec{f}} = \begin{bmatrix} f(x_i) \\ \frac{h}{1!}\frac{\partial f(x_i)}{\partial x} \\ \frac{h^2}{2!}\frac{\partial^2 f(x_i)}{\partial x^2} \\ \vdots \\ \frac{h^m}{(m)!}\frac{\partial^m f(x_i)}{\partial x^m} \\ f(x_{i+1}) \\ \frac{h}{1!}\frac{\partial f(x_{i+1})}{\partial x} \\ \frac{h^2}{2!}\frac{\partial^2 f(x_{i+1})}{\partial x^2} \\ \vdots \\ \frac{h^m}{(m)!}\frac{\partial^m f(x_{i+1})}{\partial x^m} \end{bmatrix}. \tag{B.9}$$

This implies that A must be scaled in the following manner:

$$\widetilde{A} = \begin{bmatrix} 1 & \frac{-1}{2} & \frac{1}{2^2} & \frac{-1}{2^3} & \frac{1}{2^4} & \cdots & \frac{-1}{2^{2m+1}} \\ 0 & 1 & \frac{-2}{2} & \frac{3}{2^2} & \frac{-4}{2^3} & \cdots & \frac{(2m+1)}{2^{2m}} \\ \vdots & \ddots & \ddots & \ddots & \cdots & \cdots & \vdots \\ 0 & \cdots & 0 & 1 & \frac{-(m+1)}{2} & \cdots & \frac{(-1)^{m+1}(2m+1)!}{m!\,(m+1)!\,2^{m+1}} \\ 1 & \frac{1}{2} & \frac{1}{2^2} & \frac{1}{2^3} & \frac{1}{2^4} & \cdots & \frac{1}{2^{2m+1}} \\ 0 & 1 & \frac{2}{2} & \frac{3}{2^2} & \frac{4}{2^3} & \cdots & \frac{(2m+1)}{2^{2m}} \\ \vdots & \ddots & \ddots & \ddots & \cdots & \cdots & \vdots \\ 0 & \cdots & 0 & 1 & \frac{(m+1)}{2} & \cdots & \frac{(2m+1)!}{m!\,(m+1)!\,2^{m+1}} \end{bmatrix}. \tag{B.10}$$

The scaling yields a well-conditioned matrix for reasonable values of m. Note that in practice, the inverse, $\widetilde{A}^{-1}$, is directly computed and stored in memory for

use at each interpolation step. This scaling does, however, cause the recursion relation to change some. By using $\vec{\tilde{c}}$ directly, c_{jk}, when $k = 1, 2, \ldots, 2m+1$ and $j = 0, 1, \ldots, 2m+1-k$, in (2.7) becomes:

$$h^j c_{jk} = a\frac{(j+1)}{kh} h^{j+1} c_{j+1,k-1} = a\frac{(j+1)}{k} h^j c_{j+1,k-1}. \tag{B.11}$$

Now since we are working with everything scaled in this manner, equation (2.8) changes in the following way:

$$\frac{h^j}{j!}\frac{\partial^j u}{\partial x^j}\left(x_{i+\frac{1}{2}}, t_{\frac{1}{2}}\right) \approx \sum_{k=0}^{2m+1-j} h^j c_{jk} \left(\frac{\Delta t}{2}\right)^k, \quad j = 0, \ldots, m. \tag{B.12}$$

This is the correctly scaled data for the next time step. Thus the scaling only needs to be done initially, and the scaling propagates correctly throughout the remaining time steps.

Bibliography

[1] X. Chen, D. Appelö, and T. Hagstrom. A hybrid Hermite–discontinuous Galerkin method for hyperbolic systems with application to Maxwell's equations. *Journal of Computational Physics*, 257, Part A:501 – 520, 2014.

[2] J. Goodrich, T. Hagstrom, and J. Lorenz. Hermite methods for hyperbolic initial-boundary value problems. *Math. Comp.*, 75(254):595–630, 2005.

[3] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, February 2014.