Using Manycore and Accelerated Architectures to Improve Performance of Robust Structured Multigrid Methods

A thesis

submitted by

Matthew Darsney

In partial fulfillment of the requirements
for the degree of

Master of Science

in

Mathematics

TUFTS UNIVERSITY

August 2012

ADVISOR: Scott MacLachlan

© Copyright 2012 by Matthew Darsney

Abstract

Multigrid methods have long been used as efficient solvers for a large class of discrete differential equations. Recently, parallel multigrid solvers have been primarily implemented within an MPI model, with scalability to thousands of cores. With manycore GPU architectures gaining popularity, there is an increased desire to extend multigrid solvers to these architectures. In these environments, algebraic multigrid methods can be cumbersome due to their reliance on indirect addressing. In contrast, structured-grid methods, such as BoxMG, achieve efficiency for heterogeneous problems with only direct addressing. We will discuss parallelization of BoxMG on a variety of architectures, and outline the implementation of accelerated BoxMG using OpenCL. By minimizing the data transfer between the CPU and GPU, we will show that significant performance improvements can be achieved over serial BoxMG, close to the bounds predicted by Amdahl's Law.

Acknowledgements

I would first like to thank my advisor, Scott MacLachlan. Without his dedication to my understanding and success this thesis would not have been possible. Over the last year he has taught me more than I could have ever hoped, including the importance of finding puns at every possible opportunity, the value of the semicolon, and a bit of math here and there. I'd also like to thank James Adler for always having his door open for a much needed distraction from long hours of programming. Besides being the best pie eater in the Northeast, he also knows his way around the softball diamond, and can break down algorithms like nobody's business. I want to thank Dave Moulton and Ben Bergen for their incredible support over the last several months. Without their patience and instruction throughout the course of hundreds of emails and countless hours of programming, I would have spent the better part of this decade compiling any meaningful results. I would also like to thank Chrisoph Börgers for his helpful advice and commentary during the revision process, and for having the best accent in the department (sorry Boris). My coursework at Tufts has been a pleasure due to the superior instruction and dedication of many professors. I would like to especially thank my undergraduate advisor, Mary Glaser, whose genuine care for students' understanding marks her courses among the best. I would also like to thank the many great teachers I had growing up, especially Marc Corriveau, whose passion for mathematics inspired my interest in the subject. I'd also like to thank all of my office mates from the math attic—you have been among the greatest teachers of all. Finally, I would like to thank my family, especially my parents, for all of their support and encouragement throughout the years. Without your help none of this would have been possible.

For the friends and family who have made this possible

Contents

List of Tables viii

List of Figures ix

1 Introduction 1

2 Solving a Discretized PDE 4

2.1 Discretizing our PDE 4

2.2 Point Relaxation 6

2.3 Line Relaxation 9

3 Multigrid 12

3.1 Gaussian Elimination 12

3.2 Conjugate Gradient 14

3.3 Geometric Multigrid 16

3.4 BoxMG 22

3.5 Algebraic Multigrid 26

3.6 Parallelization 30

4 Parallel Hardware and Programming Models 33

4.1 GPU Architecture 33

4.2 CUDA 36

4.3 OpenCL 36

4.4 MPI 36

4.5 BoxMG in MPI . . . 37
4.6 BoxMG and OpenCL . . . 38
4.7 BoxMG in MPI and OpenCL . . . 39

5 Implementation of BoxMG-OCL 40
5.1 BoxMG Code Structure . . . 40
5.2 OCL-MLA . . . 41
5.3 Solving Tridiagonal Systems . . . 43
5.4 Early Modifications . . . 44
5.5 Minimizing Data Transfer . . . 46
5.6 Residual Calculation . . . 50

6 Numerical Results 53
6.1 BoxMG Profiling . . . 53
6.2 Amdahl's Law . . . 54
6.3 Hardware . . . 55
6.4 Kernel Performance . . . 56
6.4.1 Right-Hand Side Kernel . . . 57
6.4.2 Residual Kernel . . . 59
6.4.3 Tridiagonal Solve Kernel . . . 59
6.4.4 Kernel Speedups . . . 60
6.5 Aggressive Coarsening . . . 61
6.6 Overall Performance Improvements . . . 62

7 Concluding Remarks 64
7.1 Future Work . . . 65

A Kernel Timing Results 67

Bibliography 69

List of Tables

6.1 BoxMG Profiling Results 54
6.2 Device Queued and Invocation Times (μs) 57
6.3 Overall Speedup of GPU Kernel Implementation Over Sequential CPU . . . 61
6.4 Performance of Increased Relaxation Sweeps and Agressive Coarsening . . . 61

List of Figures

1.1 Transistor Counts and Moore's law 2
1.2 Intel Processor Clock Rates 3

2.1 Two-dimensional grid on the unit square 5
2.2 Red-Black pointwise coloring of a two-dimensional grid 8
2.3 Gauss-Seidel relaxation on several error modes 10
2.4 Red-Black line coloring of a two-dimensional grid 11

3.1 Classic Gaussian elimination 12
3.2 The conjugate gradient and PCG algorithms 14
3.3 Smooth error component on fine and coarse grid 16
3.4 Two grids, Ω^h and Ω^{2h} 17
3.5 Full weighting restriction 18
3.6 Linear interpolation 19
3.7 V-Cycle 21
3.8 A 9-point stencil 23
3.9 BoxMG operator-based interpolation 25
3.10 A matrix and its adjacency graph 26
3.11 Domain decomposition of BoxMG, with parallel line smoothing 32

4.1 Diagram of a CPU and GPU die 34
4.2 A simple thread pool topology and GPU memory hierarchy 35

5.1 Relaxing lines in x vs lines in y 42

5.2 Device side data layout for the tridiagonal solve kernel 44
5.3 The first modification to line relaxation for GPU acceleration 45
5.4 The second modification to line relaxation 47
5.5 GPU kernel for right-hand side calculation 48
5.6 Original GPU line solve kernel and modified x line solve kernel 49
5.7 Read/write free host-side line relaxation subroutine 50
5.8 Residual calculation code and GPU kernel 52

6.1 Ware's law 55
6.2 Right-hand Side X Kernel Timings 57
6.3 Right-hand Side Y Kernel Timings 57
6.4 Residual Kernel Timings 59
6.5 Tridiagonal Solve in X Kernel Timings 59
6.6 Tridiagonal Solve in Y Kernel Timings 60
6.7 Total Solve TIme of BoxMG Implementations 62

Chapter 1

Introduction

The solution of discretized partial differential equations (PDEs) is fundamentally important for modelling physical systems in science and engineering. Solving these equations on large domains, at high resolution, and for large timescales requires fast and efficient algorithms. Consider, for example, the problem of modelling contaminant transport in subsurface groundwater flow. Modern algorithms for modelling this physical system can take as long as several days, and many simulations must be performed to gain an accurate understanding of such a system [26]. It is, thus, valuable to investigate accelerated algorithms for such problems in hopes of gaining a better understanding of this phenomenon in significantly less computational time. There is always a desire to solve larger and more complex problems and, as computer architectures change and improve, it is increasingly important to tailor existing algorithms to perform efficiently on new hardware.

For the last several decades, many technological advances have led to improved performance of numerical methods for solving PDEs. Moore's law, which states that the number of transistors that can be placed on a CPU doubles roughly every two years, ensured improved performance of many algorithmically efficient solvers simply by using newer hardware [18, 27]. Recently, despite the predictions of Moore's law, CPU clock rates have stagnated. This is largely because as modern CPU's power and clock rates increase, we are left with fewer practical means of cooling them. Modern computer architectures, thus, favor multi-core architectures rather than highly clocked single-core processors to achieve greater processing power [31].

Figure 1.1: Transistor counts of Intel processors plotted with Moore's law

The scientific computing community has long used computing clusters to improve the perfomance of code by parallelizing algorithms to thousands of cores; the scalability of these algorithms combined with a consistent increase in clock rates has provided reliable performance improvements. Now that clock rates are levelling off, we are left with new choices of how to get the maximum performance out of current architectures. For a variety of reasons, including the cost of building and powering future supercomputers and the practical scalability of current algorithms, it is worth considering alternatives to million CPU supercomputers to reach exascale.

One such alternative is the use of Graphics Processing Units (GPUs) and other accelerated architectures for general purpose scientific computing. Originally developed for rendering high performance graphics, GPUs are also a cheap and efficient way to effect large performance gains by exploiting their affinity for massively parallel operations. GPUs are able to achieve astoundingly high peak FLOPS by simultaneously performing identical operations on hundreds of low clocked (~1GHz) simple processors. Currently, the theoretical peak FLOPS and bandwidth of a modern GPU are roughly 7 times greater than the fastest

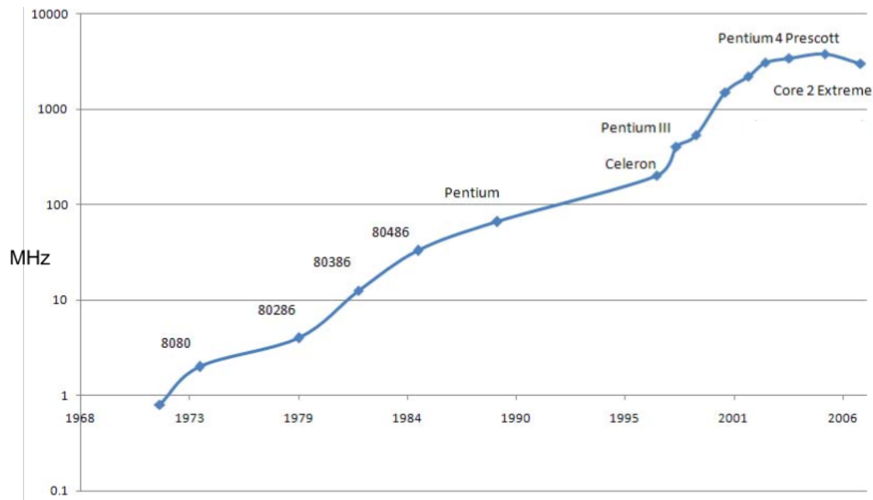


Figure 1.2: Intel Processor Clock Rates over the last few decades

CPU[11]. By using GPUs to solve the large linear systems that arise during the iterative solution of PDEs, we are able to get the maximum number of FLOPS out of a single die.

In this thesis, we will consider the acceleration of a robust structured-grid multigrid method (BoxMG) using GPUs. First, we will show how one formulates a discrete PDE and consider the resulting linear systems that represent the equation. We will then discuss multigrid methods, a class of optimal (or near-optimal) iterative solvers for a large class of discrete partial differential equations. We will also consider the BoxMG algorithm, what makes it robust, and how it compares with Algebraic Multigrid (AMG). In Chapter 4, we will consider classic and modern high performance computing architectures, and their impact on parallelizing multigrid methods. Chapter 5 focuses on the BoxMG implementation, as well as modifications made for GPU acceleration. Finally, we will analyze the numerical results of our implementation. These results show a significant performance increase over sequential BoxMG, and an overall increase comparable to the theoretical boundary predicted by Amdahl's law.

Chapter 2

Solving a Discretized PDE

We will consider solving a simple elliptic PDE in two dimensions with constant coefficients and some suitable boundary conditions,

$$LU(x,y) \equiv a\frac{\partial^2 U}{\partial x^2} + c\frac{\partial^2 U}{\partial y^2} = F(x,y)$$
$$0 < x < 1,\ 0 < y < 1, \tag{2.1}$$

where $U = 0$ on the boundary of the unit square. For practical applications, we will generally consider more complex problems; however, we use this example to motivate the manner in which a continuous PDE can be formulated as a discrete problem. We also note that the methods and notation used here closely resemble those of [9] and [10].

2.1 Discretizing our PDE

Consider approximating the solution to (2.1) on an $N \times N$ element grid discretized by a uniform rectangular mesh with $\Delta x = h = \Delta y,\ h = \frac{1}{N}$, and $h \ll 1$ as in Figure 2.1.

In order to derive a discrete analog of (2.1), we will approximate $\frac{\partial^2 U(x,y)}{\partial x^2}$ by finite differences. First, note that an $O(h^2)$ approximation of $\frac{\partial U(x,y)}{\partial x}$ is given by

$$\frac{\partial U(x,y)}{\partial x} \approx \frac{U(x+h/2,y) - U(x-h/2,y)}{h}, \tag{2.2}$$

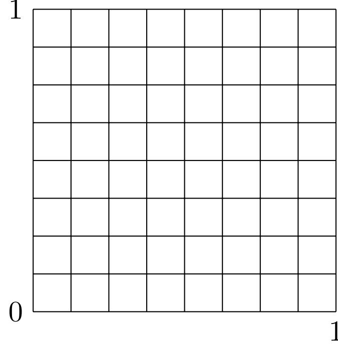


Figure 2.1: A regular two-dimensional grid on the unit square

since, from Taylor's theorem,

$$
\begin{aligned}
U(x+h/2,y) &= U(x,y) + \frac{h}{2}\frac{\partial U(x,y)}{\partial x} + \frac{h^2}{4}\frac{\partial^2 U(x,y)}{\partial x^2} + O(h^3), \\
U(x-h/2,y) &= U(x,y) - \frac{h}{2}\frac{\partial U(x,y)}{\partial x} + \frac{h^2}{4}\frac{\partial^2 U(x,y)}{\partial x^2} + O(h^3).
\end{aligned} \tag{2.3}
$$

Substituting (2.3) into (2.2) gives

$$
\begin{aligned}
\frac{U(x+h/2,y) - U(x-h/2,y)}{h} &= \frac{h\frac{\partial U(x,y)}{\partial x} + O(h^3)}{h} \\
&= \frac{\partial U(x,y)}{\partial x} + O(h^2).
\end{aligned}
$$

Differentiating both sides of (2.2) to approximate $\frac{\partial^2 U(x,y)}{\partial x^2}$ and applying (2.2) to the resulting first derivatives in the numerator yields

$$
\frac{\partial^2 U(x,y)}{\partial x^2} \approx \frac{U(x+h,y) - 2U(x,y) + U(x-h,y)}{h^2}. \tag{2.4}
$$

Similar analysis shows (2.4) is also an $O(h^2)$ approximation. Performing analogous operations for approximating $\frac{\partial^2 U(x,y)}{\partial y^2}$, introducing $u_{\alpha,\beta}$ as the discrete approximation to the continuous solution at the point $(\alpha h, \beta h)$ and $F_{\alpha,\beta}$ as the right-hand-side value at this point, yields the discretized form of our PDE,

$$
\begin{aligned}
(LU)_{\alpha,\beta} \approx a\frac{u_{\alpha+1,\beta} - 2u_{\alpha,\beta} + u_{\alpha-1,\beta}}{h^2} + c\frac{u_{\alpha,\beta+1} - 2u_{\alpha,\beta} + u_{\alpha,\beta-1}}{h^2} &= F_{\alpha,\beta} \\
u_{\alpha,0} = u_{\alpha,N} = u_{0,\beta} = u_{N,\beta} = 0 \quad 1 \le \alpha,\beta \le N-1&.
\end{aligned} \tag{2.5}
$$

We can also formulate this discretized PDE as a linear system of $(N-1)^2$ unknowns. By lexicographically ordering lines in x as $\boldsymbol{u}_i = (u_{i,1}, ..., u_{i,N-1})^T$ and $\boldsymbol{f}_i = (F_{i,1}, ..., F_{i,N-1})^T$ for $1 \leq i \leq N-1$, in the case $a = c = 1$, we can write (2.5) as a block tridiagonal system of the form

$$A\boldsymbol{u} \equiv -\frac{1}{h^2}\begin{bmatrix} B & -I & & & \\ -I & B & -I & & \\ . & . & . & & \\ & . & . & -I & \\ & & -I & B \end{bmatrix}\begin{bmatrix} \boldsymbol{u}_1 \\ . \\ . \\ . \\ \boldsymbol{u}_{N-1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f}_1 \\ . \\ . \\ . \\ \boldsymbol{f}_{N-1} \end{bmatrix} \tag{2.6}$$

where I is the $(N-1) \times (N-1)$ identity matrix, and B is of the form

$$B = \begin{bmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ . & . & . & & \\ & . & . & -1 & \\ & & -1 & 4 \end{bmatrix}.$$

Once the PDE has been discretized, it can also represented as a stencil; stencils are useful for representing the local interactions of the operator on our grid. For example, the two-dimensional stencil for our discretization of this problem is

$$-\frac{1}{h^2}\begin{pmatrix} & -c & \\ -a & 2a+2c & -a \\ & -c & \end{pmatrix},$$

where we notice that this stencil represents the pointwise dependency of unknowns in the grid, based on the discretized form of our PDE in (2.5).

2.2 Point Relaxation

Now that we have a discrete formulation of our PDE, there are many methods we can use to solve it. Two very simple methods for solving (2.5) are the Jacobi and Gauss-Seidel iterations, also called relaxation schemes; both methods update each point of our grid one

by one, but in slightly different manners. Jacobi relaxation replaces $u_{\alpha,\beta}$ with $\bar{u}_{\alpha,\beta}$ such that $\bar{u}_{\alpha,\beta}$ satisfies

$$a\frac{u_{\alpha+1,\beta} - 2\bar{u}_{\alpha,\beta} + u_{\alpha-1,\beta}}{h^2} + c\frac{u_{\alpha,\beta+1} - 2\bar{u}_{\alpha,\beta} + u_{\alpha,\beta-1}}{h^2} = F_{\alpha,\beta}, \tag{2.7}$$

whereas Gauss-Seidel replaces each value $u_{\alpha,\beta}$ with $\bar{u}_{\alpha,\beta}$ such that $\bar{u}_{\alpha,\beta}$ satisfies

$$a\frac{u_{\alpha+1,\beta} - 2\bar{u}_{\alpha,\beta} + \bar{u}_{\alpha-1,\beta}}{h^2} + c\frac{u_{\alpha,\beta+1} - 2\bar{u}_{\alpha,\beta} + \bar{u}_{\alpha,\beta-1}}{h^2} = F_{\alpha,\beta}. \tag{2.8}$$

The schemes are very similar; however, we note that with Jacobi relaxation we only need to use values of $\boldsymbol{u}$ to compute $\bar{\boldsymbol{u}}$ for each α, β, whereas, with Gauss-Seidel, we use already computed values of $\bar{u}_{\alpha-1,\beta}$ and $\bar{u}_{\alpha,\beta-1}$ since, for simplicity, we can sweep the points in lexicographic order and will already have these values when solving for $\bar{u}_{\alpha,\beta}$.

Since Jacobi relaxation does not depend on values from the current update for calculating new values, all new points can be computed independently; hence, Jacobi relaxation is easily parallelized. Though the lexicographic sweep of our points in the Gauss-Seidel scheme presented above is inherently sequential, we can use a red-black ordering of our nodes as in Figure 2.2 if we wish to parallelize our relaxation sweep [10]. This coloring scheme ensures that red points only depend on black points for their update, and vice versa; we can then update an entire color of points in parallel, followed by a parallel update of the other color's points. Other coloring schemes are possible, so long as the node colors are independent with respect to the updating scheme; for larger stencils, more colors may be necessary to expose parallelism. The choice of a scheme is problem dependent, and different coloring schemes could change the effectiveness of Gauss-Seidel as a smoother, although it has been shown for certain coloring schemes, multicolor relaxation will have the same convergence rate as lexicographic relaxation [12].

It is also useful to consider the Jacobi and Gauss-Seidel schemes as matrix operations rather than simply as pointwise updates of our grid. If we consider expressing our discretized PDE in matrix form as in (2.6), and writing $A = D - L - V$, where

$$d_{ij} = \begin{cases} a_{ii} & \text{i=j} \\ 0 & \text{i} \neq \text{j} \end{cases}, \; l_{ij} = \begin{cases} -a_{ij} & \text{i>j} \\ 0 & \text{otherwise} \end{cases}, \; v_{ij} = \begin{cases} -a_{ij} & \text{i<j} \\ 0 & \text{otherwise} \end{cases},$$

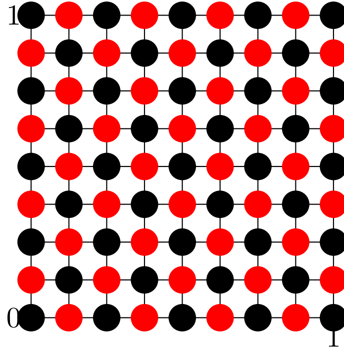


Figure 2.2: A red-black pointwise coloring of a two-dimensional grid on the unit square

then the Jacobi and Gauss-Seidel schemes can be described in matrix form as

$$\begin{aligned} \text{Jacobi} : \bar{\boldsymbol{u}} &= D^{-1}((L+V)u+F) \\ \text{Gauss-Seidel} : \bar{\boldsymbol{u}} &= (D-L)^{-1}(Vu+F). \end{aligned} \tag{2.9}$$

A single update of our points in this manner is called a point relaxation sweep. It is worth noting that one relaxation sweep is not sufficient to solve (2.5); however, it can be shown that both Jacobi and Gauss-Seidel will converge for this problem since the resulting matrix formulation of our discretized PDE is irreducibly diagonally dominant. Let us define iteration matrices, $J = D^{-1}(L+V)$ for Jacobi, and $\mathcal{L} = (D-L)^{-1}V$ for Gauss-Seidel. By analyzing the spectral radius,

$$\rho(A) = \max_i(|\lambda_i|), \text{where } \lambda_1 \ldots \lambda_n \text{ are the eigenvalues of } A,$$

of J and $\mathcal{L}$, we can show that Gauss-Seidel will have an asymptotic rate of convergence, defined as $-\ln\rho$, half that of point Jacobi. We can also use the asymptotic rate of convergence to determine precisely how many iterations it will take for these schemes to converge. In practice, the asymptotic rate of convergence is the simplest measure of rapidity of convergence of an iteration [32].

For our purposes, however, we will consider the per iteration convergence factor, μ, defined by

$$\mu = \frac{\|\bar{\boldsymbol{v}}\|}{\|\boldsymbol{v}\|}, \text{ where } \quad \boldsymbol{v} = U - \boldsymbol{u}, \quad \bar{\boldsymbol{v}} = U - \bar{\boldsymbol{u}}, \tag{2.10}$$

where U is the exact solution to the discretized PDE, $\boldsymbol{u}$ is an approximate solution, and $\bar{\boldsymbol{u}}$ is the next successive approximation. In this case, μ captures the convergence of our scheme relative to an iteration, rather than asymptotically. We notice that the per iteration convergence factor is only good for the first few iterations of our relaxation scheme and quickly approaches $\mu = 1 - O(h^2)$, which implies $O(h^{-2})$ relaxation sweeps are necessary to reduce the error by an order of magnitude [9].

By locally expanding the error in a Fourier series and studying the convergence factors of individual error modes, we gain valuable insight into the nature of the convergence factor. For example, when a/c is reasonably sized, we notice that the convergence is good for the first few sweeps because we are effectively damping high-frequency modes of the error. After a few sweeps, the convergence slows, because we have significantly smoothed oscillatory modes, and our relaxation scheme is not particularly effective for damping smooth modes [10]. This property is illustrated in Figure 2.3. We note that, in Figure 2.3a, our smooth guess is damped very slowly, in Figure 2.3b, we significantly damp our oscillatory initial guess, and, in Figure 2.3c, we are able to damp the oscillatory component of our error while the smooth component remains.

Unfortunately, in the case that either $a \ll c$ or $c \ll a$, similar analysis of our error modes shows that point Gauss-Seidel relaxation does not sufficiently damp some oscillatory modes [9].

2.3 Line Relaxation

Fortunately, in these degenerate cases, we can smooth along lines of strong connections and again achieve good convergence; this procedure is referred to as line relaxation. For example, let us assume that $a \ll c$, in which case we have strong connections vertically since the magnitude of the coefficient of the y component of our operator is significantly larger than the magnitude of the x component. In this case, we would like to update an entire vertical line of points at once; that is, we want all values $u_{\alpha,\beta}$ with the same α value (i.e. on the same vertical line) to be replaced by the $\bar{u}_{\alpha,\beta}$ which satisfy

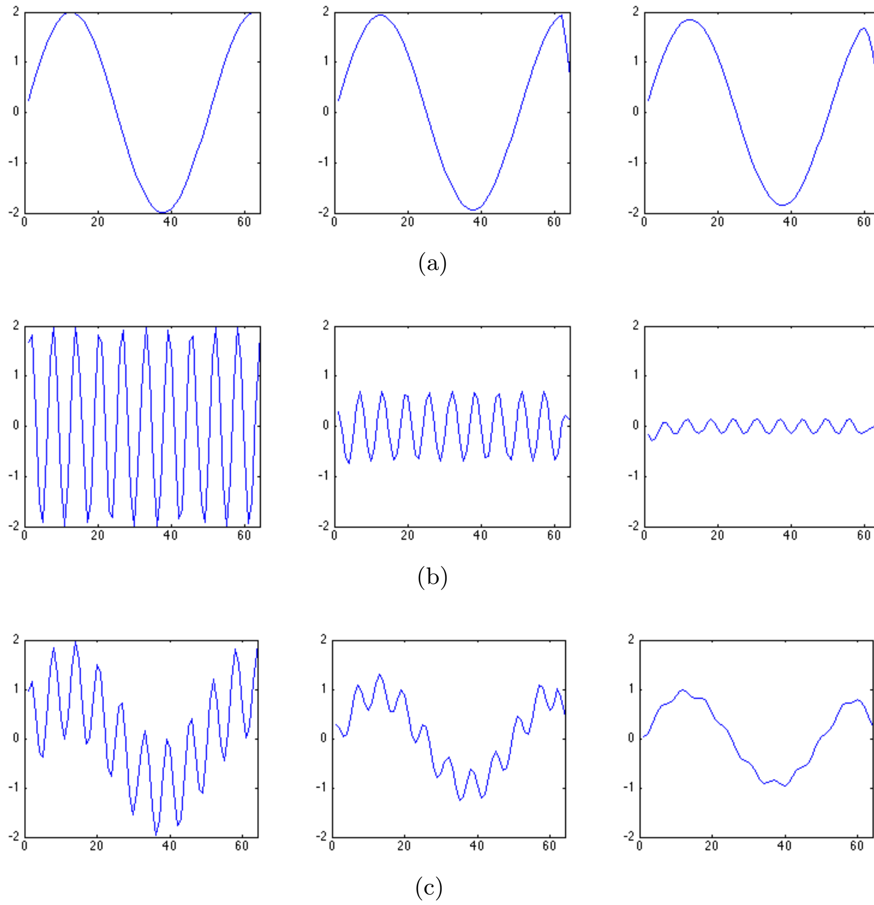


Figure 2.3: Gauss-Seidel relaxation applied to a simple 1D model problem ($A\boldsymbol{u} = \boldsymbol{f}$) with 64 points and (a) a very smooth initial guess, (b) a very oscillatory initial guess, and (c) a linear combination of (a) and (b). These figures show our initial guess (left), our approximation after 2 sweeps (middle), and after 5 sweeps (right)

$$a\frac{u_{\alpha+1,\beta} - 2\bar{u}_{\alpha,\beta} + \bar{u}_{\alpha-1,\beta}}{h^2} + c\frac{\bar{u}_{\alpha,\beta+1} - 2\bar{u}_{\alpha,\beta} + \bar{u}_{\alpha,\beta-1}}{h^2} = F_{\alpha,\beta} \tag{2.11}$$

noting that we have simply replaced $u_{\alpha,\beta+1}$ by $\bar{u}_{\alpha,\beta+1}$ in Equation (2.8) to get (2.11).

We now cannot update our $\bar{u}_{\alpha,\beta}$ values pointwise, we must instead solve a tridiagonal system for each line of the form

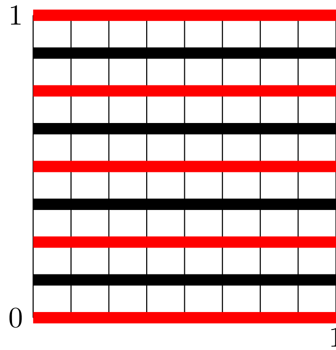


Figure 2.4: A red-black line coloring of a two-dimensional grid on the unit square

$$\begin{bmatrix} -(2a+2c) & c & & \\ c & -(2a+2c) & c & \\ \cdot & \cdot & \cdot & \\ & \cdot & \cdot & \cdot \\ & c & -(2a+2c) & c \\ & & c & -(2a+2c) \end{bmatrix} \begin{bmatrix} \bar{u}_{\alpha,1} \\ \bar{u}_{\alpha,2} \\ \vdots \\ \bar{u}_{\alpha,n-2} \\ \bar{u}_{\alpha,n-1} \end{bmatrix} = \begin{bmatrix} h^2 F_{\alpha,1} - a(u_{\alpha+1,1} + \bar{u}_{\alpha-1,1}) \\ h^2 F_{\alpha,2} - a(u_{\alpha+1,2} + \bar{u}_{\alpha-1,2}) \\ \vdots \\ h^2 F_{\alpha,n-2} - a(u_{\alpha+1,n-2} + \bar{u}_{\alpha-1,n-2}) \\ h^2 F_{\alpha,n-1} - a(u_{\alpha+1,n-1} + \bar{u}_{\alpha-1,n-1}) \end{bmatrix}.$$

In the case that $c << a$, the same principles can be applied for line relaxation in the x-direction.

Much like point relaxation, a lexicographic ordering of line relaxation is inherently sequential. If, however, we wish to parallelize our line relaxation, we note that even numbered lines need only odd lines for their update, and the odd lines only need evens. By coloring our lines red and black as in Figure 2.4, we can update all lines of one color in parallel, followed by all lines of the other color. Line relaxation applied to lines in both x and y is extremely robust, so much so that it is a suitable smoothing procedure for any degenerate constant coefficient problem of the form of (2.5) [9].

Chapter 3

Multigrid

Once we have formulated a continuous PDE as a discrete problem, we must then solve the algebraic system that represents our discretized PDE. We have seen in Chapter 2 that one way of solving our system is by using many iterations of Gauss-Seidel relaxation until sufficient convergence is achieved. Unfortunately, for large systems this method is prohibitively time consuming, as our per iteration convergence factor quickly approches 1-$O(h^2)$. We will instead consider commonly used direct and iterative methods for solving linear systems, including multigrid methods, and discuss why multigrid is the right algorithm for our purposes.

3.1 Gaussian Elimination

Among the first methods many students learn for solving linear systems is Gaussian elimination. When performed by hand, Gaussian elimination generally involves using elementary row operations to reduce a matrix to row echelon form, as in Figure 3.1, with back substitution used to solve the resulting linear system.

Algorithmically, Gaussian elimination in its simplest form is generally referred to as

$$\left[\begin{array}{ccc|c} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \end{array}\right] \xrightarrow[\text{elementary row operations}]{} \left[\begin{array}{ccc|c} b_1 & b_2 & b_3 & b_4 \\ 0 & b_5 & b_6 & b_7 \\ 0 & 0 & b_8 & b_9 \end{array}\right]$$

Figure 3.1: Reducing a general matrix to row-echelon form

LU decomposition, since we effectively factor a matrix A into $A = LV$ where L is lower triangular and V is upper triangular. The process is very similar to row reduction, in that one uses row operations to eliminate the entries of A below the main diagonal, allowing the remaining entries of A to form V, and the scaling factors used for the row operations to form L. Once we have determined L and V, we can solve $A\boldsymbol{u} = f$ as $LV\boldsymbol{u} = f$ by letting $V\boldsymbol{u} = v$, solving $Lv = f$ by forward substitution, and $V\boldsymbol{u} = v$ with back substitution.

The elements on the main diagonal of A in the LU decomposition algorithm are called "pivot elements" when they are used to eliminate the values below them in the matrix. If we swap rows to ensure that the pivot element at each step is the largest element of all entries on or below the main diagonal, the algorithm is called LU decomposition with pivoting. It can be shown that LU decomposition of all invertible square matrices is possible if pivoting is used, and that it is fairly numerically stable, even for ill-conditioned systems[5, 33].

Despite their stability and applicability to very general linear systems, there are many reasons that Gaussian elimination algorithms are not the best choice for solving the linear systems that arise in the numerical solution of PDEs. The linear systems we will be considering are generally sparse, diagonally dominant, and symmetric positive definite (SPD). The primary issue with Gaussian elimination algorithms is that they are for dense systems, they are sequential, and they have suboptimal algorithmic complexity with respect to the structure of the matrices we are considering. LU factorization for a general dense linear system of n^2 unknowns is an $O(n^6)$ algorithm [5].

For the sparse SPD systems arising from discretized PDEs, such as (2.6), we can use more sophisticated algorithms based on Gaussian elimination. The most efficient direct method for solving these linear systems is *nested dissection*. Nested dissection is closely related to *Cholesky decomposition*, which is a Gaussian elimination algorithm for SPD matrices that factorizes an SPD matrix A into $A = LL^T$. Nested dissection is a form of pivoted Cholesky decomposition where the ordering of pivot elements is chosen in such a way that the resulting factorized matrix has a minimal number of nonzero elements. Unfortunately, even using nested dissection, factorization of such SPD systems requires $O(n^3)$ operations. Furthermore, nested dissection is an algorithmically optimal direct method for sparse SPD systems arising from two-dimensional PDEs discretized on regular grids [19].

Conjugate Gradient

1. Compute $\boldsymbol{r}_0 := \boldsymbol{f} - A\boldsymbol{u}_0,\ \boldsymbol{p}_0 := \boldsymbol{r}_0$
2. For $j = 0, 1, ...$ until convergence
3. $\quad \alpha_j := (\boldsymbol{r}_j, \boldsymbol{r}_j)/(A\boldsymbol{p}_j, \boldsymbol{p}_j)$
4. $\quad \boldsymbol{u}_{j+1} := \boldsymbol{u}_j + \alpha_j \boldsymbol{p}_j$
5. $\quad \boldsymbol{r}_{j+1} := \boldsymbol{r}_j - \alpha_j A\boldsymbol{p}_j$
6. $\quad \beta_j := (\boldsymbol{r}_{j+1}, \boldsymbol{r}_{j+1})/(\boldsymbol{r}_j, \boldsymbol{r}_j)$
7. $\quad \boldsymbol{p}_{j+1} := \boldsymbol{r}_{j+1} + \beta_j \boldsymbol{p}_j$

Preconditioned CG

1. Compute $\boldsymbol{r}_0 := \boldsymbol{f} - A\boldsymbol{u}_0,\ \boldsymbol{z}_0 = M^{-1}\boldsymbol{r}_0,\ \boldsymbol{p}_0 := \boldsymbol{r}_0$
2. For $j = 0, 1, ...$ until convergence
3. $\quad \alpha_j := (\boldsymbol{r}_j, \boldsymbol{z}_j)/(A\boldsymbol{p}_j, \boldsymbol{p}_j)$
4. $\quad \boldsymbol{u}_{j+1} := \boldsymbol{u}_j + \alpha_j \boldsymbol{p}_j$
5. $\quad \boldsymbol{r}_{j+1} := \boldsymbol{r}_j - \alpha_j A\boldsymbol{p}_j$
6. $\quad \boldsymbol{z}_{j+1} := M^{-1}\boldsymbol{r}_{j+1}$
7. $\quad \beta_j := (\boldsymbol{z}_{j+1}, \boldsymbol{r}_{j+1})/(\boldsymbol{z}_j, \boldsymbol{r}_j)$
8. $\quad \boldsymbol{p}_{j+1} := \boldsymbol{z}_{j+1} + \beta_j \boldsymbol{p}_j$

Figure 3.2: The conjugate gradient and PCG algorithms

3.2 Conjugate Gradient

Fortunately, we are not forced to use direct methods to solve discretized PDEs; instead, we will use iterative methods. One of the most popular iterative methods for solving SPD systems is the conjugate gradient (CG) method, shown in Figure 3.2. Generally speaking, CG is a steepest descent algorithm for minimizing the quadratic functional

$$J(\boldsymbol{y}) = \frac{1}{2}\boldsymbol{y}^T A\boldsymbol{y} - \boldsymbol{y}^T \boldsymbol{f}, \tag{3.1}$$

which, for SPD systems, is minimized by $\boldsymbol{u}$ when $A\boldsymbol{u} = \boldsymbol{f}$. CG is part of a larger class of algorithms known as Krylov subspace methods, since at each step of CG, for an initial guess $\boldsymbol{u}_0 = 0$, $\boldsymbol{u}_j$ minimizes (3.1) over the Krylov subspace $K_j(A, \boldsymbol{f}) = span\{\boldsymbol{f}, A\boldsymbol{f}, A^2\boldsymbol{f}, ..., A^{j-1}\boldsymbol{f}\}$ [33].

Consider using CG to solve the $n \times n$ SPD system $A\boldsymbol{u} = f$. By construction, the residuals, $\boldsymbol{r}_i$, at the jth step of CG form an orthogonal basis of the Krylov subspace $K_j(A, \boldsymbol{r}_0)$; that is, $span\{\boldsymbol{r}_0, ..., \boldsymbol{r}_{j-1}\} = K_j(A, \boldsymbol{r}_0)$. Then, if after $n-1$ steps of CG we have not found $\boldsymbol{u}$, the residuals $\{\boldsymbol{r}_0, ..., \boldsymbol{r}_{n-1}\}$ will form an orthogonal basis of $\mathbb{R}^n$, and since

$$\boldsymbol{r}_n \perp \boldsymbol{r}_i \quad \text{for } i = 0, ..., n-1,$$

$\boldsymbol{r}_n$ must be 0, hence, CG will terminate in exactly n steps in exact arithmetic [23].

In some cases, CG will converge more quickly; however, for ill-conditioned matrices, CG will typically be very slow to converge. In fact, even for small matrices ($n \leq 100$), CG can fail to converge in n iterations in double precision [13]. For this reason, conjugate gradient

can seriously benefit from preconditioning. If, instead of solving $A\boldsymbol{u} = \boldsymbol{f}$, we use an SPD preconditioner, M, and solve $M^{-1}A\boldsymbol{u} = M^{-1}\boldsymbol{f}$ where $M^{-1}A$ is well-conditioned, CG will converge more quickly. The preconditioned conjugate gradient (PCG) algorithm, as shown in Figure 3.2, reflects this change.

In theory, we would like $M^{-1} = A^{-1}$, in which case the PCG algorithm would converge in 1 iteration. Obviously, if we could efficiently compute A^{-1} we wouldn't be using PCG, so we must settle for another choice of M. A simple choice for a preconditioner is $M = D$, where D is formed from the diagonal elements of A; this is referred to as diagonally-preconditioned conjugate gradient. The diagonal elements of A are a good candidate for a preconditioner, since inverting D is trivial, so it is very easy to calculate $\boldsymbol{z}_j$. Unfortunately, naively choosing M in this way is not suitable in general, since in many cases D^{-1} will not adequately approximate A^{-1}; as a result, diagonally-preconditioned conjugate gradient can still take many iterations to converge.

There are many different ways to choose M. We can also modify the PCG algorithm so no explicit representation of M is necessary. Notice that in lines 1 and 6 of the PCG algorithm, we solve for

$$\boldsymbol{z}_j = M^{-1}\boldsymbol{r}_j, \tag{3.2}$$

assuming we have already determined some M^{-1}. If, instead, we replace the calculation of $\boldsymbol{z}_j$ in (3.2) with the solution to the linear system

$$M\boldsymbol{z}_j = \boldsymbol{r}_j, \tag{3.3}$$

with some choice of $M \approx A$, we can improve the convergence of PCG. By approximately solving $A\boldsymbol{z}_j = \boldsymbol{r}_j$ for $\boldsymbol{z}_j$, we implicitly determine a preconditioner M that will greatly improve the convergence of PCG for general SPD matrices. We must, however, consider the cost of approximating this solution. Multigrid methods are an efficient way to approximate the solutions to such systems, ensuring both improved convergence of the PCG algorithm, and a computationally inexpensive preconditioning step.

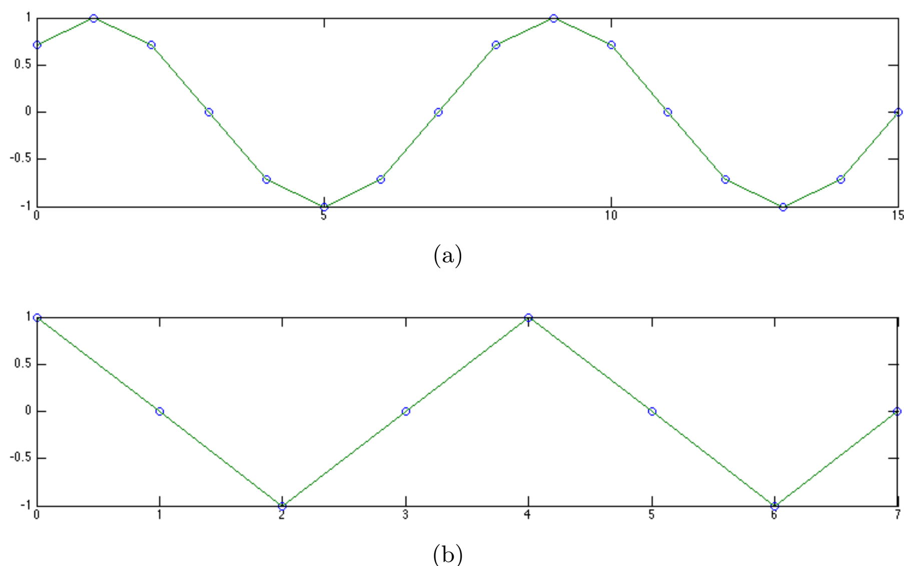


Figure 3.3: A smooth error component. (a) shows the error component represented on a fine grid, (b) shows its restriction to a coarser grid.

3.3 Geometric Multigrid

Again consider the example problem from Chapter 2. As we have seen, we are attempting to solve the linear system

$$A\boldsymbol{u} = \boldsymbol{f} \tag{3.4}$$

that represents a discretized PDE on a square two-dimensional grid. The relaxation schemes that we have looked at, Jacobi and Gauss-Seidel, have the property that they damp oscillatory components of our error, while smooth components of our error remain. If we could somehow damp the smooth components of our error, then, in theory, we would have a very effective method for solving our linear system.

Consider a smooth error component as in Figure 3.3a. Since we cannot efficiently damp smooth error components using relaxation, it would be helpful if we could somehow represent this error as being more oscillatory. Notice, in Figure 3.3b, we have restricted our error to a grid that is "coarser" than the grid in Figure 3.3a. As a result, this error component appears more oscillatory, since it completes the same number of cycles as on the fine grid, but in half as many grid points. Since this smooth component looks more oscillatory on the coarse grid, we can now use our relaxation schemes to damp it more effectively.

Algorithm 3.1: Coarse-grid correction

```
Relax on Au = f, with initial guess v
Compute r = f − Av
    Relax on Ae = r on a coarse grid
Correct v with the coarse−grid approximation by v := v + e
Relax on Av = f
```

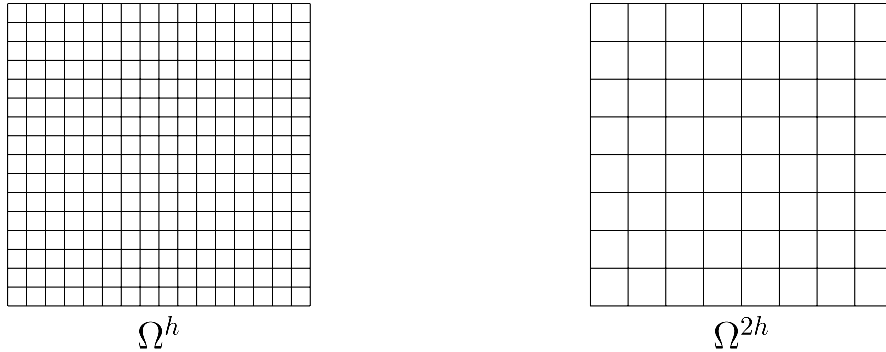


Figure 3.4: Two grids, a fine grid Ω^h, and its corresponding coarse grid Ω^{2h}, which is derived from coarsening Ω^h by a factor of 2.

Now, assume we have some approximation, $\boldsymbol{v}$, to the exact solution, $\boldsymbol{u}$, of (3.4). Then, the error in our approximation, $\boldsymbol{e} = \boldsymbol{u} - \boldsymbol{v}$, satisfies $A\boldsymbol{e} = \boldsymbol{r} = \boldsymbol{f} - A\boldsymbol{v}$. This means that relaxation of (3.4) is equivalent to relaxing on $A\boldsymbol{e} = \boldsymbol{r}$ for an arbitrary initial guess, $\boldsymbol{v}$, which yields $\boldsymbol{r} = \boldsymbol{f} - A\boldsymbol{v}$, and initial guess $\boldsymbol{e} = 0$ for the equation $A\boldsymbol{e} = \boldsymbol{r}$ [10]. Since we can relax on the residual equation $A\boldsymbol{e} = \boldsymbol{r}$ to improve our solution, and since smooth modes look more oscillatory when restricted to coarser grids, this suggests that we should relax on the residual equation on coarser grids and interpolate this solution to finer grids to improve the approximation, $\boldsymbol{v}$. This process is described explicitly in Algorithm 3.1.

Now that we have the intuition of how our scheme should look, we must still define how we restrict our error to the coarse grid and interpolate a correction to the fine grid. First, let us denote our fine grid by Ω^h and our coarse grid by Ω^{2h}, since we will coarsen by a factor of 2, as in Figure 3.4. Similarly, denote A and $\boldsymbol{r}$ on grid h by A^h and $\boldsymbol{r}^h$, with A^{2h} and $\boldsymbol{r}^{2h}$ being their representation on grid $2h$. Here, we took A^{2h} to simply be the discrete formulation of our operator on Ω^{2h}, but we will need to choose how to restrict the value of $\boldsymbol{r}^h$ to Ω^{2h}.

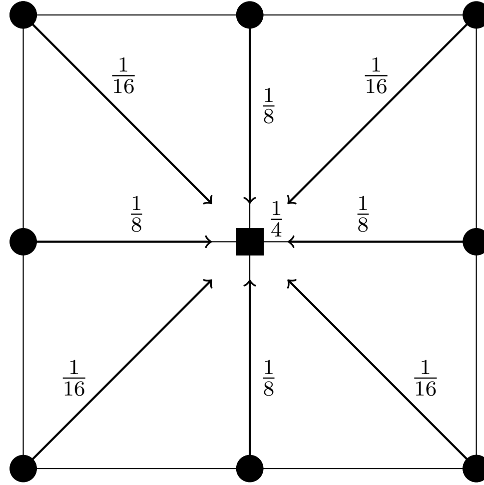


Figure 3.5: Full weighting restriction. Circles represent fine grid points, squares represent coarse grid points.

Let us denote our restriction operator by I_h^{2h}, writing $I_h^{2h}\boldsymbol{r}^h = \boldsymbol{r}^{2h}$. One of the simplest ways to restrict our residual is by simply carrying the fine grid values of $\boldsymbol{r}^h$ to $\boldsymbol{r}^{2h}$; this type of restriction is called *injection*. In terms of our restriction operator, it is represented as

$$I_h^{2h}\boldsymbol{r}^h = \boldsymbol{r}^{2h} \quad \text{where} \quad \boldsymbol{r}^{2h}_{\alpha,\beta} = \boldsymbol{r}^h_{2\alpha,2\beta},\ 1 \leq \alpha, \beta \leq \frac{N}{2} - 1. \tag{3.5}$$

A more sophisticated method of restriction is *full weighting*. This is accomplished by taking an average of fine-grid points surrounding a coarse point as in Figure 3.5. Algebraically this is represented as

$$\begin{aligned} I_h^{2h}\boldsymbol{r}^h &= \boldsymbol{r}^{2h} \quad \text{where} \\ \boldsymbol{r}^{2h}_{\alpha,\beta} = \frac{1}{16}[\boldsymbol{r}^h_{2\alpha-1,2\beta-1} &+ \boldsymbol{r}^h_{2\alpha-1,2\beta+1} + \boldsymbol{r}^h_{2\alpha+1,2\beta-1} + \boldsymbol{r}^h_{2\alpha+1,2\beta+1} \\ &+ 2(\boldsymbol{r}^h_{2\alpha,2\beta-1} + \boldsymbol{r}^h_{2\alpha,2\beta+1} + \boldsymbol{r}^h_{2\alpha-1,2\beta} + \boldsymbol{r}^h_{2\alpha+1,2\beta}) \\ &+ 4\boldsymbol{r}^h_{2\alpha,2\beta}], \quad 1 \leq \alpha, \beta \leq \frac{N}{2} - 1. \end{aligned} \tag{3.6}$$

For the remainder of our discussion of geometric multigrid, we will use full weighting for our restriction operator, even though injection may be a better option in some cases [10]. Now that we have a good candidate for restricting our residual to the coarse grid, we must determine an interpolation operator, I_{2h}^h, such that $I_{2h}^h\boldsymbol{e}^{2h} = \boldsymbol{e}^h$. The simplest interpolation operator is called *bilinear interpolation*. Bilinear interpolation is, in many

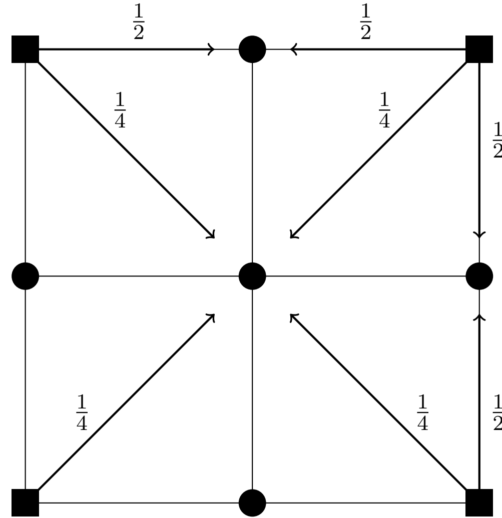


Figure 3.6: Linear interpolation. Circles represent fine-grid points, squares represent coarse-grid points.

ways analogous to full weighting, since, to recover fine-grid values from the coarse-grid approximation, we will simply take an average of the surrounding coarse points, as shown in Figure 3.6. Algebraically, bilinear interpolation is given by

$$\begin{aligned}
I_{2h}^{h}\boldsymbol{e}^{2h} &= \boldsymbol{e}^{h} \quad \text{where} \\
\boldsymbol{e}_{2\alpha,2\beta}^{h} &= \boldsymbol{e}_{\alpha,\beta}^{2h}, \\
\boldsymbol{e}_{2\alpha+1,2\beta}^{h} &= \frac{1}{2}(\boldsymbol{e}_{\alpha,\beta}^{2h} + \boldsymbol{e}_{\alpha+1,\beta}^{2h}) \\
\boldsymbol{e}_{2\alpha,2\beta+1}^{h} &= \frac{1}{2}(\boldsymbol{e}_{\alpha,\beta}^{2h} + \boldsymbol{e}_{\alpha,\beta+1}^{2h}) \\
\boldsymbol{e}_{2\alpha+1,2\beta+1}^{h} &= \frac{1}{4}(\boldsymbol{e}_{\alpha,\beta}^{2h} + \boldsymbol{e}_{\alpha+1,\beta}^{2h} + \boldsymbol{e}_{\alpha,\beta+1}^{2h} + \boldsymbol{e}_{\alpha+1,\beta+1}^{2h}), \quad 0 \le \alpha, \beta \le \frac{N}{2} - 1. \qquad (3.7)
\end{aligned}$$

We now note the relationship between the full weighting restriction operator and the linear interpolation operator is

$$I_{h}^{2h} = c(I_{2h}^{h})^{T}, \quad c \in \mathbb{R}. \qquad (3.8)$$

This is called a *variational property*; this property leads to the *Galerkin condition*, $A^{2h} = I_h^{2h} A^h I_{2h}^{h}$, which defines our coarse-grid operator in terms of our interpolation operator. It has been shown that by using this variational property to define the restriction operator based on the interpolation operator, and the coarse-grid operators based on the Galerkin condition,

for real SPD matrices, we minimize error in the range of interpolation in the energy norm, $\|\boldsymbol{e}\|_A^2 = \boldsymbol{e}^t A \boldsymbol{e}$ [8, 10].

Now that we have defined our interpolation and restriction operators, we can rewrite Algorithm 3.1 in a more explicit way.

Algorithm 3.2: Two-Grid Correction

```
Relax on A^h u^h = f^h , with initial guess v^h
Compute r^h = f^h - A^h v^h
Restrict r^h to Ω^{2h} by I_h^{2h} r^h = r^{2h}
    Relax on A^{2h} e^{2h} = r^{2h} with initial guess 0
    Interpolate e^{2h} to Ω^h by I_{2h}^h e^{2h} = e^h
Correct v^h with the coarse-grid approximation by v^h := v^h + e^h
Relax on A^h v^h = f^h
```

In principle, we would like to exactly solve $A^{2h}\boldsymbol{e}^{2h} = \boldsymbol{r}^{2h}$ on the coarse grid in order to get the best possible correction. Unfortunately, relaxing on $A^{2h}\boldsymbol{e}^{2h} = \boldsymbol{r}^{2h}$ is not an efficient method for exactly solving this linear system. In fact, we will have the same problem relaxing this problem on the coarse grid as relaxing $A^h\boldsymbol{u}^h = \boldsymbol{f}$ on Ω^h, namely, we can only effectively damp oscillatory error modes on Ω^{2h} by relaxing on $A^{2h}\boldsymbol{e}^{2h} = \boldsymbol{r}^{2h}$. To get a better solution, we would like to also damp the smooth error modes on this grid. It is now evident that solving $A^{2h}\boldsymbol{e}^{2h} = \boldsymbol{r}^{2h}$ is the same problem as solving $A^h\boldsymbol{u}^h = \boldsymbol{f}$; hence, we can recursively solve our coarse-grid problem by relaxing on $A^{2h}\boldsymbol{e}^{2h} = \boldsymbol{r}^{2h}$ and restricting our error to Ω^{4h}. We recurse until we are at a sufficiently small grid where our problem can be solved directly, at which point we interpolate back up to successively finer grids. This process is referred to as a *V-Cycle*, due to the popular schematic shown in Figure 3.7, and is defined explicitly in Algorithm 3.3.

Algorithm 3.3: Recursive V-Cycle

```
VCycle(v^h, f^h, h)
    if(Ω^h = coarsest grid)
        Solve A^h v^h = f^h directly
        Return v^h
    else
        Relax on A^h v^h = f^h
        f^{2h} = I_h^{2h}(f^h - A^h v^h)
        v^{2h} = 0
        v^{2h} = VCycle(v^{2h}, f^{2h}, 2h)
        v^h = v^h + I_{2h}^h v^{2h}
        Relax on A^h v^h = f^h
```

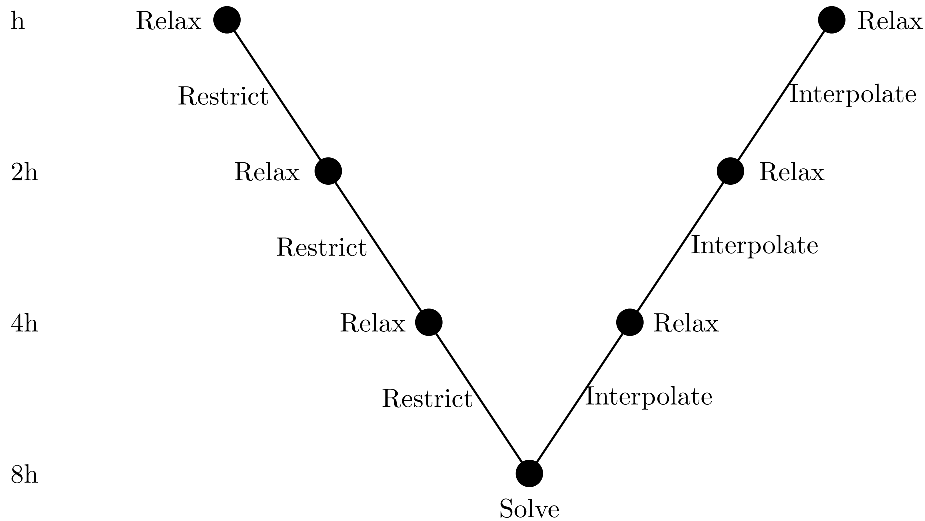


Figure 3.7: Grid schematic for a V-Cycle

The V-Cycle, as we have constructed it, is a simple and very straightforward geometric multigrid scheme. It is an appealing algorithm for solving the linear systems that arise from our discretized PDE, since its algorithmic complexity is near-optimal ($O(n^2 \log n)$) [10]. If, however, we build our initial guess up from the coarsest grid, we perform a *Full-Multigrid Cycle*, a closely related but slightly more complex algorithm than the V-Cycle; the full-multigrid cycle has optimal algorithmic complexity for solving linear systems representing discretized PDEs on structured grids, in 2D ($O(n^2)$), or in 3D ($O(n^3)$) [9].

Unfortunately, geometric multigrid with linear interpolation is not a robust algorithm for solving discretized PDEs. As we have seen in the previous chapter, for strongly anisotropic problems, line relaxation is necessary to achieve good smoothing within a relaxation scheme. This is sufficient for PDEs with constant coefficients, but for problems with strongly discontinuos coefficients, the multigrid scheme we have developed exhibits very poor convergence, even with line relaxation [3]. To achieve robustness for these problems, we must construct an interpolation scheme based on our discrete operator.

3.4 BoxMG

In [3], Alcouffe et. al. considered a problem of the form

$$-\nabla \cdot (K(x,y)\nabla U(x,y)) = f(x,y), \tag{3.9}$$

on a bounded region in $\mathbb{R}^2$, where K and f are allowed to be discontinuous across internal boundaries, Γ, of the domain, and

$$U \text{ and } \mu \cdot (K\nabla U) \text{ are continuous at } (x,y) \text{ for almost every } (x,y) \in \Gamma, \tag{3.10}$$

where, for $(x,y) \in \Gamma$, $\mu(x,y)$ is a fixed normal vector to Γ. Sometimes, geometric multigrid with bilinear interpolation is very effective for such problems; however, in the case where K jumps by several orders of magnitude across Γ, geometric multigrid with bilinear interpolation exhibits poor convergence. This is because bilinear interpolation is based on approximating the continuity of ∇U across Γ. In the case that K jumps by orders of magnitude, however, (3.10) suggests it is more natural to approximate the continuity of $K\nabla U$ across Γ. The solution discussed in [3] uses an operator-dependent interpolation scheme to approximate the continuity of $K\nabla U$.

Suppose that, at a point (α, β) on a grid Ω^h, our discrete operator has a stencil representation like that of Figure 3.8. By constructing our interpolation operator based on this stencil, we can achieve good convergence for problems with strongly discontinuous coefficients such as (3.9).

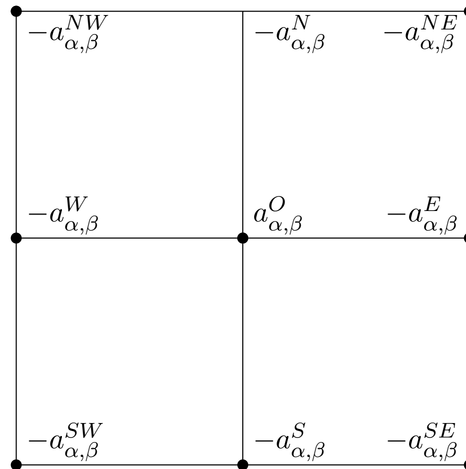


Figure 3.8: A 9-point stencil at the point (α, β) with compass-based notation

In one dimension, the optimal choice for interpolation is determined by scaling coarse-grid points by the diagonal elements of the discrete operator, and using the linear combination of the weighted adjacent coarse-grid points to interpolate fine-grid points. For Poisson's equation in one dimension, this yields linear interpolation. In two dimensions, points embedded along coarse-grid lines are determined by restricting the problem to one dimension, and interpolating the fine grid point along this line by linear interpolation. Points in the center of coarse-grid points are determined by ensuring that the error in interpolating these points with respect to the fine-grid operator is 0. This yields the two-dimensional linear interpolation scheme shown in Figure 3.6.

For more complex problems, we can derive similar interpolation schemes by using the discrete operator. Again, for lines embedded in the coarse grid, we will compress the problem into one dimension, and use the elements of the discrete operator along this line to weight the contributions of coarse-grid points. For points in the center of coarse-grid points, interpolation is again determined such that the error in the range of interpolation with respect to the fine-grid operator is 0. The specific construction of the interpolation

operator is shown in Figure 3.9; algebraically it is represented as

$$
\begin{aligned}
I_{2h}^{h}\boldsymbol{e}^{2h} &= \boldsymbol{e}^{h} \quad \text{where} \\
\boldsymbol{e}^{h}_{2\alpha,2\beta} &= \boldsymbol{e}^{2h}_{\alpha,\beta}, \\
\boldsymbol{e}^{h}_{2\alpha+1,2\beta} &= \frac{a^{SW}_{2\alpha+1,2\beta} + a^{W}_{2\alpha+1,2\beta} + a^{NW}_{2\alpha+1,2\beta}}{a^{O}_{2\alpha+1,2\beta} - a^{S}_{2\alpha+1,2\beta} - a^{N}_{2\alpha+1,2\beta}}\boldsymbol{e}^{2h}_{\alpha,\beta} + \frac{a^{SE}_{2\alpha+1,2\beta} + a^{E}_{2\alpha+1,2\beta} + a^{NE}_{2\alpha+1,2\beta}}{a^{O}_{2\alpha+1,2\beta} - a^{S}_{2\alpha+1,2\beta} - a^{N}_{2\alpha+1,2\beta}}\boldsymbol{e}^{2h}_{\alpha+1,\beta} \\
\boldsymbol{e}^{h}_{2\alpha,2\beta+1} &= \frac{a^{SW}_{2\alpha,2\beta+1} + a^{S}_{2\alpha,2\beta+1} + a^{SE}_{2\alpha,2\beta+1}}{a^{O}_{2\alpha,2\beta+1} - a^{E}_{2\alpha,2\beta+1} - a^{W}_{2\alpha,2\beta+1}}\boldsymbol{e}^{2h}_{\alpha,\beta} + \frac{a^{NW}_{2\alpha,2\beta+1} + a^{N}_{2\alpha,2\beta+1} + a^{NE}_{2\alpha,2\beta+1}}{a^{O}_{2\alpha,2\beta+1} - a^{E}_{2\alpha,2\beta+1} - a^{W}_{2\alpha,2\beta+1}}\boldsymbol{e}^{2h}_{\alpha,\beta+1} \\
\boldsymbol{e}^{h}_{2\alpha+1,2\beta+1} &= (a^{S}_{2\alpha+1,2\beta+1}\boldsymbol{e}^{h}_{2\alpha+1,2\beta} + a^{SW}_{2\alpha+1,2\beta+1}\boldsymbol{e}^{h}_{2\alpha,2\beta} + a^{W}_{2\alpha+1,2\beta+1}\boldsymbol{e}^{h}_{2\alpha,2\beta+1} \\
&\quad a^{NW}_{2\alpha+1,2\beta+1}\boldsymbol{e}^{h}_{2\alpha,2\beta+2} + a^{N}_{2\alpha+1,2\beta+1}\boldsymbol{e}^{h}_{2\alpha+1,2\beta+2} + a^{NE}_{2\alpha+1,2\beta+1}\boldsymbol{e}^{h}_{2\alpha+2,2\beta+2} \\
&\quad a^{E}_{2\alpha+1,2\beta+1}\boldsymbol{e}^{h}_{2\alpha+2,2\beta+1} + a^{SE}_{2\alpha+1,2\beta+1}\boldsymbol{e}^{h}_{2\alpha+2,2\beta})/a^{O}_{2\alpha+1,2\beta+1} 1 \le \alpha, \beta \le \frac{n}{2} - 1.
\end{aligned} \tag{3.11}
$$

Notice the similarities between the operator-dependent interpolation scheme and our linear interpolation scheme; namely, both methods interpolate fine-grid points as a weighted average of the surrounding coarse points; however, with the operator-based scheme, the weights used are from the stencil that represents the discrete operator at the desired point. Again, variational coarsening is used to determine the restriction operator; that is, the restriction operator for this method is the transpose of the interpolation operator. When this interpolation scheme is used in conjunction with a multigrid cycle and line relaxation, the algorithm is known as Black Box Multigrid, or *BoxMG*. BoxMG is a very robust algorithm for PDEs discretized on structured grids [15].

Thus far, the multigrid methods we have discussed are designed for discretized PDEs on logically rectangular grids. If we wish to minimize the total number of degrees of freedom in the discrete formulation of a PDE, it may make more sense to use a problem-dependent discretization; unfortunately, this discretization could yield a very irregular grid. In this case, there may be no natural choice of a coarse grid. Even if we could construct a coarse grid, the methods we have developed suggest no way of determining interpolation and restriction operators to and from the coarse grid. If no regular structure exists for our grid, we must investigate new multigrid methods for solving our discretized PDE.

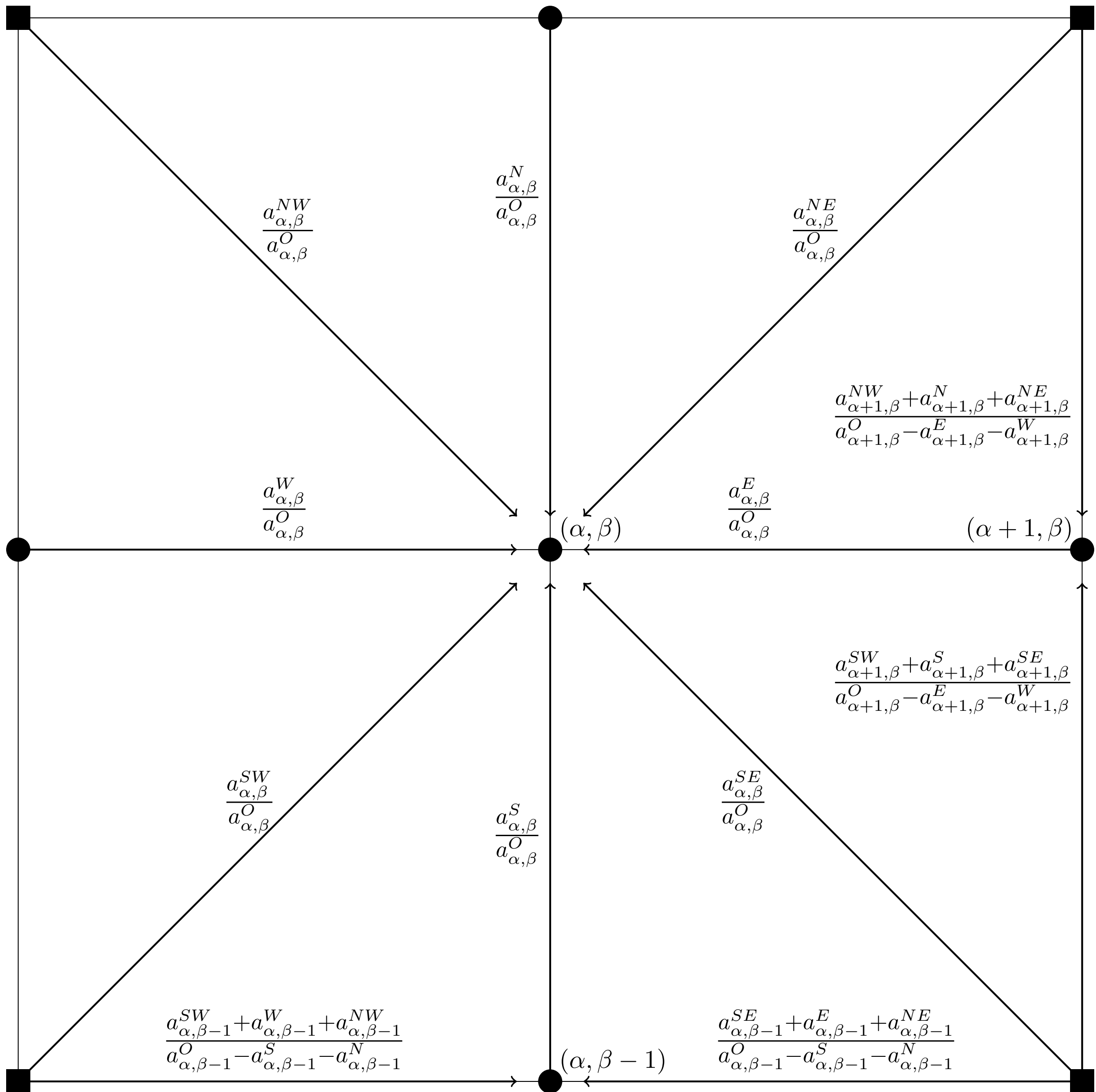


Figure 3.9: Operator-based interpolation used in BoxMG. Circles represent fine-grid points, squares represent coarse-grid points. Notice we use fine-grid points to calculate the center point, so it is assumed fine-grid points along coarse-grid lines have been calculated first. Values along lines are the weights of the corresponding points used to calculate the fine-grid point.

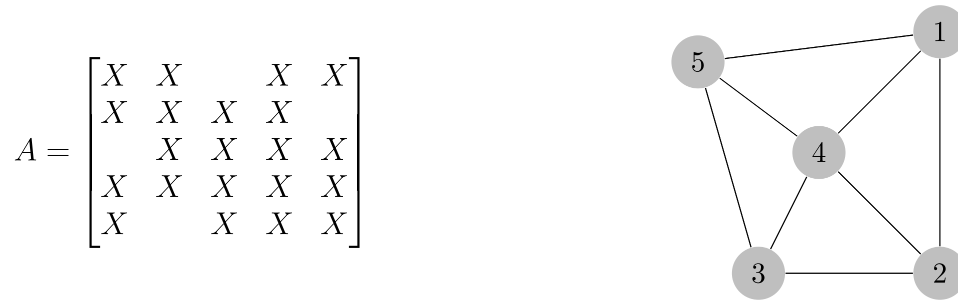


Figure 3.10: A matrix A and its undirected adjacency graph

3.5 Algebraic Multigrid

Consider solving a linear system where the underlying geometric structure that the system represents is not a uniform mesh, or is unknown. Such a system would be difficult to solve with the methods we have developed, since our interpolation and coarsening algorithms are based on having a uniform structured mesh. We can, however, use the algebraic properties of the linear system to design multigrid methods for such problems; this concept is referred to as Algebraic Multigrid (AMG) [8].

We can define a grid for a strictly algebraic problem $A\boldsymbol{u} = f$ by forming an undirected adjacency graph representing the nonzero elements of the matrix as in Figure 3.10. We can use this graph to represent the fine grid of our problem and, by defining the concepts of "smooth" error, restriction/interpolation, and coarsening in a strictly algebraic sense, we can develop a multigrid method for this problem.

For a matrix A with entries a_{ij}, we say two variables u_i and u_j are *strongly coupled* whenever a_{ij} is comparable in magnitude with $\max_{k\neq i} |a_{ik}|$ or $\max_{k\neq i} |a_{jk}|$. Similarly, u_i *strongly depends* on u_j if, for some threshold value $0 < \theta \leq 1$, $-a_{ij} \geq \theta \max_{k\neq i}(-a_{ik})$. In this case, we also say u_j *strongly influences* u_i. As a motivating example, consider the discretized PDE in (2.5). In the strongly anistropic case $a \ll c$, we would say that adjacent points along lines in y, say $u_{\alpha,\beta}$ and $u_{\alpha,\beta+1}$, would be strongly coupled since their coefficients in the resulting matrix formulation will be large; in contrast, adjacent points along lines in x will not be strongly coupled, since they will have comparably small coefficients. Similarly, $u_{\alpha,\beta}$ and $u_{\alpha,\beta+1}$ will strongly influence/depend on each other.

In the construction of geometric multigrid, we noticed that relaxation was only effective

for a few sweeps until the oscillatory modes of the error were effectively eliminated, leaving the smooth error modes to be damped on the coarse grid. Such analysis isn't possible for a strictly algebraic problem; instead, it has been shown that relaxation schemes smooth error along the strongest couplings, or equivalently, smooth error varies slowly along the strongest couplings [10, 8]. AMG relaxation sweeps are performed in the same manner as geometric multigrid, however, instead of performing sweeps until oscillatory error modes have been damped, relaxation is performed until convergence slows. The remaining error after relaxation is then called *algebraically smooth*, although it may be geometrically quite oscillatory.

Once relaxation sweeps have been performed to reduce an initial error to be dominated by algebraically smooth errors, a restriction operator must be chosen to transfer this error to the coarse grid. As with geometric multigrid, we will construct a restriction operator using variational coarsening. Our restriction operator will, thus, be the transpose of our interpolation operator, and we must first construct an interpolation operator.

In the geometric case, we have indirectly seen an emphasis on constructing interpolation weights based on strong connections between points. In the case of bilinear interpolation, strong connections between points were simply determined by their adjacency. In BoxMG, strong connections were determined by both adjacency and the discrete operator. In AMG, we consider a similar process, except instead of using the geometric proximity of points, we will use the algebraic properties of influence and dependence to characterize strong connections. By determining the weights of our interpolation operator from entries of the matrix A, based on influence and dependence between points, we can hope to faithfully interpolate a correction to the fine grid.

We will denote the interpolation operator by I_C^F since h and $2h$ are not necessarily meaningful in this context. Using the notation of [10], let us denote by C_i points on the coarse grid that strongly influence u_i, by D_i^s the neighboring fine-grid points that strongly influence u_i, and by D_i^w the neighboring points (both fine and coarse grid) that do not strongly influence u_i. Then, letting C be the points in the coarse grid and F those in the fine grid, an interpolation operator for AMG is given by

$$I_C^F e_i = \begin{cases} e_i & \text{if } i \in C \\ \displaystyle\sum_{j \in C_i} \omega_{ij} e_j & \text{if } i \in F \end{cases}. \tag{3.12}$$

To determine the interpolation weights ω_{ij}, let us consider that the ith component of the residual can be represented as the sum of components of C_i, D_i^s, and D_i^w, as in

$$a_{ii} e_i \approx -\sum_{k \in C_i} a_{ik} e_k - \sum_{k \in D_i^s} a_{ik} e_k - \sum_{k \in D_i^w} a_{ik} e_k \approx 0. \tag{3.13}$$

Now, let us replace the third term in (3.13) by the result of distributing the terms associated with the error e_j, for $j \in D_i^s$, to the diagonal coefficient; that is, we will substitute

$$\sum_{k \in D_i^w} a_{ik} e_k \approx \left(\sum_{k \in D_i^w} a_{ik} \right) e_i. \tag{3.14}$$

This substitution allows us to represent the contribution of weakly connected neighbors in terms of coarse-grid components, while minimizing error in the case that e_i does not depend strongly on points in D_i^w.

For the second term of (3.13), we will represent the error components of strongly connected fine-grid neighbors as a linear combination of the error components of the coarse-grid points that strongly influence these neighbors. We will then approximate each e_j, for $j \in D_i^s$, as

$$e_j \approx \frac{\sum_{k \in C_i} a_{jk} e_k}{\sum_{k \in C_i} a_{jk}}. \tag{3.15}$$

Here, the choice of the numerator is justified since the coarse-grid components influence the fine-grid points in proportion to the matrix entry a_{ij}, while the denomenator ensures constants are exactly interpolated. We have also assumed that any strongly connected fine-grid points must have at least one point in their common interpolatory sets. Substituting (3.15) and (3.14) into (3.13), with a good deal of algebra, yields the interpolation weights

$$\omega_{ij} = -\frac{a_{ij} + \displaystyle\sum_{m \in D_i^s} \left(\frac{a_{im} a_{mj}}{\displaystyle\sum_{k \in C_i} a_{mk}} \right)}{a_{ii} + \displaystyle\sum_{n \in D_i^w} a_{in}}. \tag{3.16}$$

As with geometric multigrid, interpolation is constructed from a weighted average of coarse points; however, in AMG we emphasize interpolation along strong connections, as determined strictly from the entries of A, rather than the underlying geometric structure of the problem. Letting A^F be the fine-grid operator and A^C the coarse-grid operator, we define A^C using the Galerkin method, $A^C = I_F^C A^F I_C^F$.

Now that we have constructed an interpolation operator, we must only determine how to choose the coarse grid points. This process is accomplished by adhering to two heuristics for choosing a point. Letting S_i be the points that strongly influence a point u_i, these rules are:

1: For each point u_i in F, every point $u_j \in S_i$ should be in C_i or depend strongly on at least one point in C_i.

2: The set of points C should be maximal in the sense that no C point depends strongly on another C point.

Heuristic **1** is chosen since we have required that two strongly connected fine-grid points must have a common point in their coarse-grid interpolatory sets. Heuristic **2** is chosen so that the coarse grid is sufficiently small compared to the fine grid, ideally ensuring that solving the coarse-grid problem takes significantly less time than the fine-grid problem. It may not be possible to satisfy both **1** and **2**, rather, it is important to ensure **1** is satisfied for proper interpolation, while loosely enforcing **2** [10].

The process of choosing a coarse grid occurs in two phases. In the first phase, the point u_i that strongly influences the most points is chosen as the first point in C. Then all points it strongly influences are placed in F. Points that have not been assigned to C or F that strongly influence many points in F are then favored as candidates for placement into C. Another C point is then chosen, its strongly connected neighbors are placed into F, and so on, until all points have been divided into C or F. The second phase places as many F points as necessary into C to ensure heuristic **1** is satisfied. This process is then repeated for the coarse grid to determine the points in the next coarsest grid, and its interpolation operator, and so for all grids until the coarsest grid and interpolation operator have been established.

Once the interpolation operators and coarse grids have been constructed, the AMG

solve phase is defined recursively in exactly the same way as Algorithm 3.3. Algebraic multigrid is an effective and robust algorithm for solving a large class of discretized PDEs on unstructured grids. It is interesting to note that by slightly modifying the interpolation weights of AMG and basing strong connections on geometric distance between nodes, AMG and BoxMG are equivalent on structured-grid problems in terms of interpolation/restriction and choice of coarse grids [24].

3.6 Parallelization

Parallelization of multigrid methods is crucial for solving problems with a large number of degrees of freedom in a reasonable amount of time. Many of the components of multigrid lend themselves nicely to parallelization, and effectively parallelizing the natively serial components of multigrid is an area of active research.

We will consider solving a problem with many degrees of freedom by distributing the domain across several processors, as is the case with a conventional computing cluster. For AMG, pointwise smoothing is usually implemented, and, in the case of Jacobi, involves the independent update of every point. More sophisticated schemes are possible, such as using pointwise Gauss-Seidel on the domain of each processor, and Jacobi along processor boundaries [34]. In either case, smoothing in AMG can be effectively parallelized. Unfortunately, by using unstructured grids to discretize a problem, the unknowns are not necessarily ordered in a logical manner with respect to influence and dependence among nodes; as a result, many components of AMG (e.g. interpolation) tend to achieve a low percentage of peak computing rates due to the indirect addressing of values [20].

The selection of coarse grids in AMG is, as we have described it, an inherently sequential algorithm. Though many parallel AMG implementations exist, a primary difficulty in their development is the choice of a parallel coarse grid selection algorithm. The natural approach is to divide the domain of the fine grid among all processors, perform the coarse-grid selection algorithm discussed earlier on each local domain, then classify the boundary points as coarse or fine-grid points. More sophisticated schemes based on graph partitioning are also possible, as well as hybrid implementations of different coarse-grid selection schemes

[22, 2]. With any parallel coarse-grid selection scheme, great lengths must be taken to ensure that there is proper load balancing of computation; that is, it is crucial that each processor's subdomain is approximately the same size on each grid.

BoxMG, in contrast, has a coarse-grid hierarchy naturally defined by the structured fine grid, and an interpolation operator entirely determined by the discrete operator; as a result, load balancing is not an issue, and interpolation/restriction are readily parallelized. If pointwise Jacobi smoothing is used, BoxMG is extremely parallelizable; to achieve robustness, however, line relaxation is necessary. Fortunately, BoxMG implements a parallel line relaxation scheme that is discussed and analyzed in [6], which we will briefly describe.

Consider distributing the domain of a two-dimensional grid among processors as in Figure 3.11a, where we wish to parallelize the relaxation of the red line across the three processors indicated by the blue dotted lines. As we have seen, the resulting system that must be solved to relax this line is tridiagonal, but the components of the matrix will be stored on separate processors, as shown in Figure 3.11b. The solving process occurs in three phases:

Phase 1: Eliminate lower and upper diagonal elements of the local portion of the tridiagonal system. The resulting structure of the matrix is shown in Figure 3.11c.

Phase 2: Solve the interface tridiagonal system. This tridiagonal system is represented by the ■'s in Figure 3.11c.

Phase 3: Solve the resulting local tridiagonal systems. These systems are represented by the x's in Figure 3.11c.

This algorithm is the key to parallelizing line smoothing in BoxMG; by performing all phases for all lines of a single color independently, we can achieve good parallel efficiency.

With this parallel line smoothing algorithm, all major steps of BoxMG lend themselves nicely to parallelization. Also, since we are discretizing our problem on a structured grid, our unknown values are aligned in memory so that we avoid many of the indirect addressing issues of AMG. This attribute is important in a standard parallel computing environment, but becomes increasingly important when considering accelerated architectures that are more memory bound than traditional supercomputing architectures, such as GPUs; for this reason, robust structured-grid approaches such as BoxMG are a good candidate for GPU

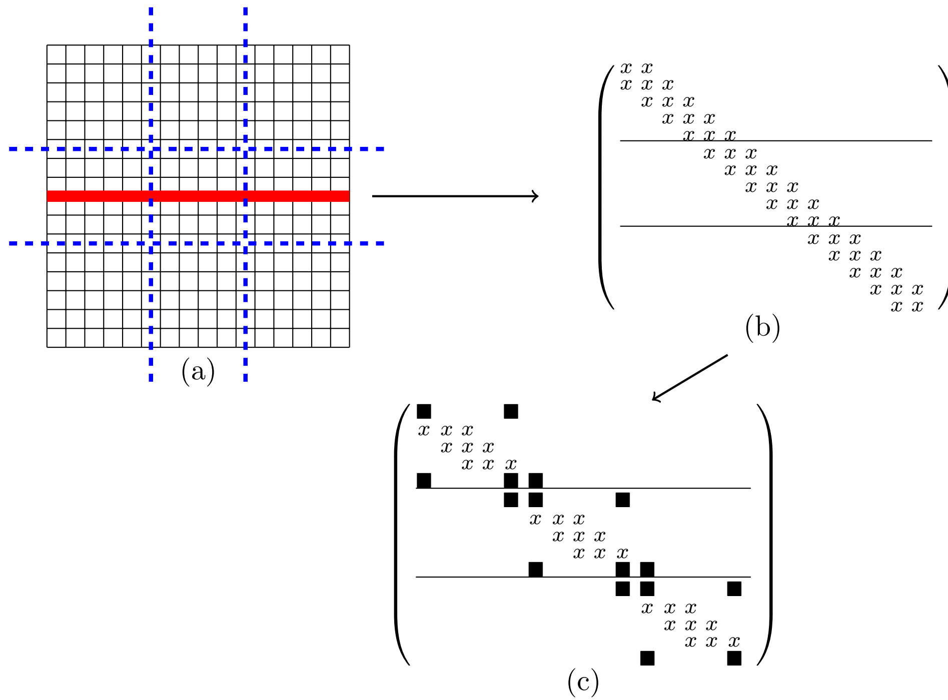


Figure 3.11: Parallel line relaxation in BoxMG. (a) shows an example of a two-dimensional domain distributed among 9 processors, where processor boundaries are represented by the dashed blue lines. (b) shows the tridiagonal matrix that results from smoothing the red line in (a). (c) shows the resulting structure of the matrix after reducing the system to local tridiagonal systems. Horizontal lines represent processor boundaries, x's represent nonzero values of the initial tridiagonal system, ■'s represent nonzero entries resulting from the first phase of the parallel line solve.

acceleration.

Chapter 4

Parallel Hardware and Programming Models

As the performance of supercomputers continues to increase at an exponential rate, it remains to be seen which technology will prevail in the race to produce the first exaflop machine. The latest predictions suggest that the first exascale computer will be in production by 2019 [21], but current architectures would be prohibitively expensive to power at this scale, as investigated in great detail in [17]. GPU computing offers the appeal of achieving high peak throughput while maintaining a flop to watt ratio significantly better than traditional CPUs. For this reason, GPUs could pave a credible path to exascale, and it is worth investigating their use as a high-performance computing architecture.

4.1 GPU Architecture

Originally developed in 1999 as hardware designed specifically for rendering high-perfomance graphics, GPUs have now become a viable means of accelerating a wide range of data-parallel computations. The theoretical peak FLOP/s of a modern GPU are currently ~7X that of a CPU, a performance increase made possible by the specialized architecture of the GPU [29].

A modern GPU is, in some ways, like a multi-core CPU, just with many more cores; the way the device gains efficiency with the cores, however, is slightly different. GPUs are

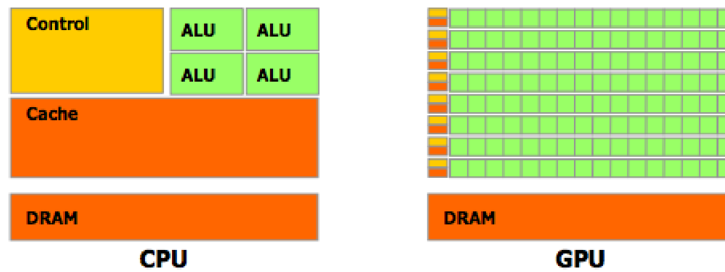


Figure 4.1: Simple schematic of a CPU and GPU die. Notice how much more physical space is dedicated to arithmetic logic units (ALUs) on the GPU compared to the CPU [29].

designed for the massively parallel and computation-intensive tasks common for graphics rendering. As a result, more of the physical hardware of the GPU is dedicated to data processing as compared with a CPU; this is illustrated in Figure 4.1.

Unfortunately, despite the apparent similarities between multi-core CPU and GPU architectures, programming a GPU for efficient parallel computation is significantly different than programming a CPU. The GPU architecture is built around groups of simple processing units; these groups are commonly referred to as *Streaming Multiprocessors*, or SMs. A GPU is generally composed of several, sometimes tens, of SMs.

Because of their unusual architecture, programs designed for GPUs must be designed differently than those for CPUs. A function written to run specifically on the GPU is called a *kernel*. Kernels are different than standard CPU functions in that they are designed to be executed by many threads in parallel; when a kernel is invoked, it spawns several threads to be run on the SMs of the GPU. These threads are organized into some topology by the programmer or compiler; threads are organized into blocks and blocks of threads into a grid, as in Figure 4.2a. The position of a thread in this topology generally determines its execution within the kernel. On the GPU, thread blocks are distributed among SMs, where they are broken into groups, generally of size 32, for execution; these groups are called *warps*. When a warp is dispatched to an SM, the SM executes all threads simultaneously. If there is any data-dependent branching that occurs between the threads within a warp, the SM serially executes each path of the branch; hence, SMs gain maximum efficiency when they operate in a single-instruction multiple data (SIMD) fashion within warps [29].

The topology of the thread pool also corresponds to the memory hierarchy of the GPU.

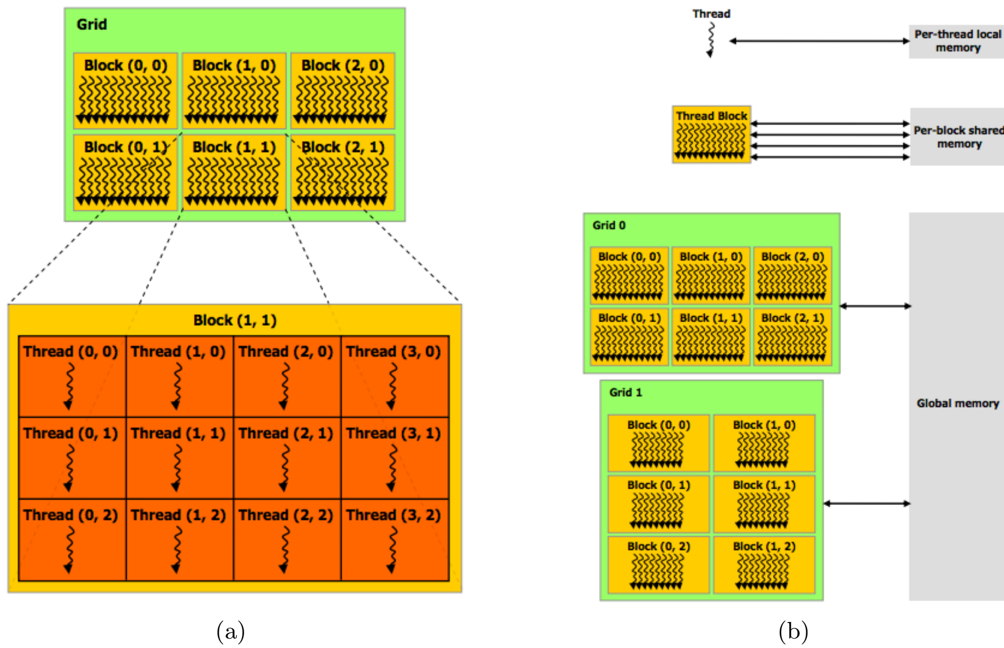


Figure 4.2: (a) A simple thread pool topology. Threads are organized into blocks, and blocks into a grid. (b) Thread pool topology and the GPU memory hierarchy [29].

As one might expect, single threads have a small amount of register space for a single core within an SM. Thread blocks have a comparitively much larger shared memory space, within which they can read and write values for interprocessor use. At the highest level there is a large amount of DRAM, usually several GB, that all thread blocks and grids have global access to. This hierarchy is illustrated in Figure 4.2b. When a warp must read or write data from global memory, there is a latency period during which no instructions can be executed within that warp until the data transfer is completed. In order to fully utilize the GPU, it is important to have a large enough thread pool that when one warp is latent another warp can execute on the SM, effectively hiding the latency of the data transfer [29]. In addition to the concern of memory access latency, GPUs achieve their greatest efficiency when memory is properly aligned with respect to kernel execution [28]. Not all algorithms lend themselves to GPU acceleration—it is essential to exploit fine-grained data parallelism to see significant performance gains from using the GPU.

4.2 CUDA

Originally released in 2006, the Compute Unified Device Architecture (CUDA) was developed by Nvidia as one of the first solutions for implementing GPU kernels in high-level languages. CUDA provides extensions for commonly used programming languages such as C and FORTRAN to handle interactions between the CPU and GPU, and allows users to write GPU kernels with C-like syntax. Despite its compact syntax and relative ease of implementation, CUDA is currently somewhat limited by its ability to be implemented only on Nvidia hardware.

4.3 OpenCL

Writing and executing kernels for the GPU requires additional constructs beyond normal programming language features. The Open Computing Language (OpenCL) provides standardized general-purpose language extensions for programming a variety of parallel architectures, including GPUs. A complete description of the OpenCL standard can be found in [1].

In the context of this thesis, OpenCL was used for all interactions between the CPU (host) and the GPU (device), including reading/writing of data between host and device, and programming and execution of the kernels. For GPU computing, OpenCL is in many ways like CUDA (see [29]); however, without modification to the source code, OpenCL code can run on a variety of hardware, including multi-core CPUs.

4.4 MPI

The Message Passing Interface (MPI) is a library of functions designed for programming parallel processors, such as those found in a computer cluster. It is a widely used standard, and is one of the first of its kind. MPI works by assigning an ID to each processor (rank) of a group of processors and executing parallel processes on each rank, with message passing used to coordinate the transfer of data between ranks. A good introduction to MPI can be found in [30]. Many parallel algorithms, including BoxMG and AMG, have been implemented

using MPI. As supercomputing architectures evolve, heterogeneous computing nodes will likely become more popular, including nodes with both CPUs and GPUs; as a result, the combination of parallel programming standards, such as MPI and OpenCL, will become necessary to achieve peak performance.

4.5 BoxMG in MPI

As we have seen in Chapter 3, serial BoxMG assumes exactly the same form as the V-Cycle algorithm for classic geometric multigrid, just with operator-based interpolation and Galerkin coarse-grid operators.

On a computing cluster comprised of several CPUs using MPI, the parallel BoxMG algorithm is a lot like what was discussed at the end of Chapter 3. The domain of the problem is distributed among MPI ranks as in Figure 3.11a, where each local domain is approximately the same size to ensure good load balancing. The restriction and interpolation operations are performed locally, thus all subdomains can perform restriction and interpolation in parallel. Residual calculation and coarse-grid correction are also performed pointwise, so in a similar fashion, these calculations can be performed on each subdomain in parallel. At each step, some communication is necessary to exchange values at processor boundaries, but in general these calculations are easily parallelized.

As we have also discussed, pointwise Jacobi smoothing can be done in parallel for all points; unfortunately this is only sufficient for very easy problems. In Chapter 3, we discussed the parallel line smoothing algorithm that allows us to gain parallelism and robustness for anisotropic problems. This algorithm naturally requires some communication between processors; specifically, processors along a line must collect the interface system on a single rank for **Phase 2**, solve the system, then broadcast the solution of the interface system to all ranks along the line. After this broadcast, all lines can independently solve the local tridiagonal systems in **Phase 3**. Despite the communication necessary for line smoothing, in general, BoxMG is well suited to parallel computation in an MPI environment.

4.6 BoxMG and OpenCL

In serial, BoxMG contains many fine-grained data parallel operations that are naturally suited to GPU acceleration with OpenCL. These are realized in all major steps of the algorithm: restriction, interpolation/coarse-grid correction, residual calculation, and red-black line relaxation.

Restriction in serial BoxMG is implemented as a lexicographic pointwise update of the coarse-grid points based on the restriction operator and the current fine-grid residual. Restriction of each point requires approximately 20 arithmetic operations on a very small amount of data. Such computations are well suited to implementation as a GPU kernel; by structuring a restriction kernel based on a single thread updating one point on the coarse grid, we achieve a very high degree of fine-grained data parallelism and, in turn, can get good performance out of the GPU.

Residual calculation is implemented in a similar manner to restriction, that is, it is a lexicographic pointwise calculation, and is, thus, implemented on the GPU much in the same way with similar performance benefits. Interpolation and coarse-grid correction are implemented together in serial BoxMG, but this calculation is well suited to the GPU in much the same manner as restriction and residual calculation.

Pointwise Jacobi relaxation in serial BoxMG is easily implemented on the GPU as a parallel pointwise update in the same manner as residual calculation. As we have seen, this is not sufficient for many problems, thus, we must use line relaxation. Consider using a red-black coloring scheme for line relaxation as in Figure 2.4. In serial, line relaxation is implemented by relaxing one color followed by another. We have seen in Chapter 2 that relaxing a single line is accomplished by solving a tridiagonal system. When relaxing lines of a single color, the partial residual representing the right-hand sides of the tridiagonal systems is first calculated, followed by solving each tridiagonal system for that color. The calculation of the partial residual is a lexicographic pointwise calculation in a similar manner to residual calculation; hence, it is well suited to computation on the GPU. Once we have determined the right-hand sides of the tridiagonal systems, instead of solving the systems sequentially, we can use the GPU to solve all of the systems simultaneously, ideally achieving good performance improvements for line relaxation. We will discuss the specific algorithm

for solving the tridiagonal systems on the GPU in Chapter 5.

4.7 BoxMG in MPI and OpenCL

BoxMG in MPI lends itself to GPU acceleration much in the same way it does in serial. Consider BoxMG implemented in an MPI environment as above, except with a GPU per rank. The restriction, interpolation/coarse-grid correction, and residual calculation in MPI are all realized as a lexicographic pointwise update of the local domain of each MPI rank. Then, by implementing a kernel that spawns a thread per point for each update, we can achieve a high level of parallelism on each MPI rank for all of these operations.

Pointwise Jacobi smoothing is similarly parallelizable, but again for robustness we will need to use the parallel line smoothing algorithm. In MPI, calculation of the partial residual, which is the right-hand side of the tridiagonal systems we solve to smooth a color, is a pointwise lexicographic calculation on the local domain of a rank. This process then lends itself to GPU acceleration in the same manner we have seen with the right-hand side calculation in serial. For the parallel line solve algorithm, we will consider doing all lines of a single color at once. First, we will complete **Phase 1** for all lines of a single color simultaneously, on the GPU. For **Phase 2**, we will gather all values for the interface systems on all MPI ranks that span each line, and solve the interface systems in parallel on the GPU on each rank. **Phase 3** can then be accomplished by solving all local tridiagonal systems of all lines of one color simultaneously on an MPI rank.

It is now apparent that many operations in BoxMG exhibit fine-grained data parallel operations that would benefit from being implemented on the GPU. Unfortunately, exposing these operations is not the only key to achieving good performance. Transferring data between host and device is incredibly time consuming and should be avoided if at all possible. As a result, it is imperative that a large portion of the necessary data structures, including the interpolation and discrete operators for all accelerated grids, reside on the GPU. By preventing unnecessary data transfers between host and device, we achieve good performance improvements by accelerating BoxMG with GPUs.

Chapter 5

Implementation of BoxMG-OCL

After identifying portions of the BoxMG code that were good candidates for implementation on the GPU, several modifications were made to to allow for GPU acceleration of the most time consuming data parallel computations. After making many changes to the data movement structure within the accelerated code, good performance improvements were achieved. For the remainder of this chapter, we will discuss specifics of the serial implementation of BoxMG and the necessary refactoring steps that were taken to accelerate the code with a GPU.

5.1 BoxMG Code Structure

In the discussion that follows, we adopt the same variable naming conventions that are present in the BoxMG code. We also only consider 9-point discretizations—five point discretizations simply omit the values not in the cardinal directions of the stencil. Also, it is not expected that performance improvements are possible for all grids; for this reason, in our discussion we consider *accelerated grids* as those grids on which some computation is done on the GPU. All coarser grids are computed solely on the CPU, since the level of parallelism on these grids is not sufficient to justify the necessary overhead of using the GPU.

The name *Black Box Multigrid* is appropriate for BoxMG, since the user only needs to provide the pointwise stencil representing the difference equations of the discretized PDE on

the finest grid; BoxMG calculates the resulting coarse-grid formulations and interpolation operators based on this stencil. Other user-defined parameters include the type of relaxation scheme (point Gauss-Seidel, lines in x/y, etc.), the number of relaxation sweeps, the shape of the multigrid cycle used, and a stopping tolerance. A thorough discussion of the user-defined parameters of BoxMG is described in [15].

The pointwise stencil for all grids is stored in a three-dimensional array called SO. The first two dimensions of SO are used to index into a point on the specified two-dimensional grid, while the third dimension is used to specify the direction of the weight in the stencil, as in the compass representation of Figure 3.8. The entries of the tridiagonal systems used for line smoothing can be extracted from this stencil by choosing the central and negative western (southern) components for the main and off diagonals for relaxation in lines in x (y). These diagonal elements are explicitly stored in the BoxMG code in a three-dimensional array called SOR; the resulting matrices stored in SOR are factorized before line relaxation on the CPU occurs. It is worth noting that since Fortran stores multidimensional arrays in column-major order, we index into SO differently when calculating the right-hand sides of the tridiagonal systems for x and y. Also, for lines in x, we store the solution to the tridiagonal systems directly in Q, whereas, for lines in y, we must store the solution in an auxiliary array B, then copy the solution into Q. These differences are illustrated in Figure 5.1.

The current right-hand side of the equation we are solving (f^h in $A^h\boldsymbol{v}^h = f^h$ from Algorithm 3.3) is stored in a two-dimensional array, QF, where the dimensions are used to index into a point on the specified grid. The current approximate solution ($\boldsymbol{v}^h$ in $A^h\boldsymbol{v}^h = f^h$) is stored in a two-dimensional array, Q, that is structured the same way as QF. The values of SO, QF, and Q are then the only values needed for line relaxation and residual calculation.

5.2 OCL-MLA

In order to call OpenCL kernels from BoxMG, we used a mid-level abstraction layer to the OpenCL runtime called OCL-MLA [7]. OCL-MLA provides Fortran bindings to a C library that greatly simplifies the implementation of data transfer and kernel invocation operations

```
        !Loop over red lines, then black lines
        DO JBEG=JBEG_START, JBEG_END, JBEG_STRIDE
           JEND=2*((J1-JBEG)/2)+JBEG
           DO J=JBEG,JEND,2
              DO I=2,I1
                 Q(I,J)=QF(I,J)+SO(I,J,KS)*Q(I,J-1)+SO(I,J+1,KS)
&                     *Q(I,J+1)+SO(I,J,KSW)*Q(I-1,J-1)+SO(I+1,J,KNW)
&                     *Q(I+1,J-1)+SO(I,J+1,KNW)*Q(I-1,J+1)
&                     +SO(I+1,J+1,KSW)*Q(I+1,J+1)
              ENDDO
           ENDDO
           !Solve tridiagonal systems with LAPACK
           DO J=JBEG,JEND,2
              CALL DPTTRS (I1-1, 1, SOR(2,J,1), SOR(3,J,2),
&                  Q(2,J), I1-1, INFO)
           ENDDO
```

(a)

```
        !Loop over red lines, then black lines
        DO IBEG=IBEG_START,IBEG_END,IBEG_STRIDE
           IEND=2*((I1-IBEG)/2)+IBEG
           DO  I=IBEG,IEND,2
              DO  J=2,J1
                 B(J)= QF(I,J)+SO(I,J,KW)*Q(I-1,J)+SO(I+1,J,KW)
&                     *Q(I+1,J)+SO(I,J,KSW)*Q(I-1,J-1)+SO(I+1,J,KNW)
&                     *Q(I+1,J-1)+SO(I,J+1,KNW)*Q(I-1,J+1)
&                     +SO(I+1,J+1,KSW)*Q(I+1,J+1)
              ENDDO
              !Solve tridiagonal systems with LAPACK
              CALL DPTTRS (J1-1, 1, SOR(2,I,1), SOR(3,I,2),
&                  B(2), J1-1, INFO)
              !Copy solution to Q
              DO j=2,J1
                 Q(I,J) = B(J)
```

(b)

Figure 5.1: (a) Shows the code for relaxing lines in x on the CPU. (b) Shows line relaxation for lines in y on the CPU. Notice that we must copy the solution to an auxiliary array in (b) since Fortran uses column-major ordering. In (a), Q is the RHS of the tridiagonal systems, and is overwritten with the solution to the systems.

necessary for accelerating code with OpenCL. OCL-MLA defines a set of logical devices that run OpenCL C kernels, and provides a user interface for moving data, setting arguments, and timing events on these devices. By specifying the architecture of the logical devices used for acceleration with OpenCL, OCL-MLA makes it possible to run the OpenCL kernels within BoxMG on a variety of different hardware without any modification to the source code.

5.3 Solving Tridiagonal Systems

The most time consuming routine within BoxMG is line relaxation, thus this portion of the code is a good candidate for GPU acceleration. We have seen that line relaxation is accomplished by red-black coloring the lines of the domain and smoothing one color at a time. By simultaneously smoothing all lines of a single color, we achieve a high degree of parallelism, and could potentially see good performance improvements.

Smoothing all lines simultaneously requires solving the many tridiagonal systems that arise, in parallel. For this task, we used the parallel cyclic reduction kernel described in [35]. Generally speaking, parallel cyclic reduction is a divide and conquer algorithm for solving tridiagonal systems that has suboptimal complexity ($O(n \log n)$), but is better suited to parallel computation than inherently sequential algorithms with optimal complexity, such as the Thomas algorithm.

The kernel is structured so that each row of each tridiagonal system corresponds to a single thread. For example, assume we wish to solve M tridiagonal systems of N unknowns each—we would then invoke the kernel with M blocks of N threads each. Data is aligned in a similar manner; the subdiagonals of each matrix are stored in a device side one-dimensional array, a, of size M$\times$N, with a leading 0 at the beginning of each subdiagonal. Main diagonals and the right-hand sides are stored in the same manner, in b and d respectively, with no leading 0. Super diagonals are stored in c in the same manner as sub diagonals, instead with a trailing 0. This data layout is illustrated in Figure 5.2. When the kernel is invoked, each thread block loads all necessary data for its tridiagonal system into shared memory, performs parallel cyclic reduction to solve the system, and stores the result in the device-side

0	$a_{1,1}$	$a_{1,2}$	$\cdots$	$a_{1,N-2}$	$a_{1,N-1}$	0	$a_{2,1}$	$a_{2,2}$	$\cdots$	$a_{2,N-2}$	$a_{2,N-1}$	$\cdots$	$a_{M,N-2}$	$a_{M,N-1}$

$b_{1,1}$	$b_{1,2}$	$b_{1,3}$	$\cdots$	$b_{1,N-1}$	$b_{1,N}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$	$\cdots$	$b_{2,N-1}$	$b_{2,N}$	$\cdots$	$b_{M,N-1}$	$b_{M,N}$

$c_{1,1}$	$c_{1,2}$	$\cdots$	$c_{1,N-2}$	$c_{1,N-1}$	0	$c_{2,1}$	$c_{2,2}$	$\cdots$	$c_{2,N-2}$	$c_{2,N-1}$	0	$\cdots$	$c_{M,N-1}$	0

$d_{1,1}$	$d_{1,2}$	$d_{1,3}$	$\cdots$	$d_{1,N-1}$	$d_{1,N}$	$d_{2,1}$	$d_{2,2}$	$d_{2,3}$	$\cdots$	$d_{2,N-1}$	$d_{2,N}$	$\cdots$	$d_{M,N-1}$	$d_{M,N}$

Figure 5.2: Device side data layout for the tridiagonal solve kernel. a are the subdiagonals, b main diagonals, c super diagonals, and d right-hand sides. A value $k_{i,j}$ is the jth element of the ith linear system.

solution vector x.

5.4 Early Modifications

Since line relaxation is so time consuming and, in theory, quite parallelizable, early modifications to the BoxMG code focused on formatting data to use the kernel for solving tridiagonal systems. This meant using OCL-MLA to create device-side buffers for a, b, c, d, and a solution vector x. Once these buffers were created, the logical next step was to write the necessary values to these buffers from the host. Early implementations of this write procedure were incredibly time consuming, due to the fact that 4 write calls were used for each tridiagonal system. For large systems, this quickly totalled hundreds of write calls before invoking the kernel; as a result, performance was crippled, on the order of $100\times$ slower than sequential LAPACK calls. In addition to these write calls, after the solution was read from the device, a memory copy to host-side data structures was necessary. Fortunately, solve times for the tridiagonal systems were significantly faster than sequential LAPACK calls; performance was strictly degraded by the movement of data to and from the GPU. The changes these modifications made to the line relaxation code are illustrated in Figure 5.3.

It quickly became apparent that writing each diagonal element for each system was not a viable option for achieving real performance gains. Instead of writing one diagonal of a system at a time, we could instead store the diagonals for the accelerated grids in a new

```
        !Loop over red lines, then black lines
        DO JBEG=JBEG_START, JBEG_END, JBEG_STRIDE
           JEND=2*((J1-JBEG)/2)+JBEG
           DO J=JBEG,JEND,2
              DO I=2,I1
                 Q(I,J)=QF(I,J)+SO(I,J,KS)*Q(I,J-1)+SO(I,J+1,KS)
     &                *Q(I,J+1)+SO(I,J,KSW)*Q(I-1,J-1)+SO(I+1,J,KNW)
     &                *Q(I+1,J-1)+SO(I,J+1,KNW)*Q(I-1,J+1)
     &                +SO(I+1,J+1,KSW)*Q(I+1,J+1)
              ENDDO
           ENDDO
           !Call LAPACK to solve each tridiagonal system
           DO J=JBEG,JEND,2
              CALL DPTTRS (I1-1, 1, SOR(2,J,1), SOR(3,J,2),
     &             Q(2,J), I1-1, INFO)
           ENDDO
```

(a)

```
        !Loop over red lines, then black lines
        DO JBEG=JBEG_START, JBEG_END, JBEG_STRIDE
           JEND=2*((J1-JBEG)/2)+JBEG
           DO J=JBEG,JEND,2
              DO I=2,I1
                 Q(I,J)=QF(I,J)+SO(I,J,KS)*Q(I,J-1)+SO(I,J+1,KS) &
                      *Q(I,J+1)+SO(I,J,KSW)*Q(I-1,J-1)+SO(I+1,J,KNW) &
                      *Q(I+1,J-1)+SO(I,J+1,KNW)*Q(I-1,J+1) &
                      +SO(I+1,J+1,KSW)*Q(I+1,J+1)
              ENDDO
           ENDDO
           OFFSET=0
           CALL ocl_clear_event_wait_list(wait_list, ierr)
              !Write diagonals and RHS for all lines for this color
              DO J=JBEG,JEND,2
                 !fill arrays on device side
                 !get lower diagonal ptr
                 diagptr = C_LOC(SOR(2,J,2))

                 !fill lower diag buffer
                 CALL ocl_enqueue_write_buffer(OCL_PERFORMANCE_DEVICE, &
                      d_a, 1, OFFSET*bytes, bytes, diagptr, event, ierr)
                 CALL ocl_add_event_to_wait_list(wait_list, event, ierr)

                 !get main diagonal ptr
                 diagptr = C_LOC(SOR(2,J,1))

                 CALL ocl_enqueue_write_buffer(OCL_PERFORMANCE_DEVICE, &
                      d_b, 1, OFFSET*bytes, bytes, diagptr, event, ierr)
                 CALL ocl_add_event_to_wait_list(wait_list, event, ierr)

                 !get upper diagonal ptr
                 diagptr = C_LOC(SOR(3,J,2))

                 CALL ocl_enqueue_write_buffer(OCL_PERFORMANCE_DEVICE, &
                      d_c, 1, OFFSET*bytes, bytes, diagptr, event, ierr)
                 CALL ocl_add_event_to_wait_list(wait_list, event, ierr)

                 !get RHS ptr
                 diagptr = C_LOC(Q(2,J))

                 CALL ocl_enqueue_write_buffer(OCL_PERFORMANCE_DEVICE, &
                      d_d, 1, OFFSET*bytes, bytes, diagptr, event, ierr)
                 CALL ocl_add_event_to_wait_list(wait_list, event, ierr)

                 OFFSET=OFFSET+1
              ENDDO

              CALL ocl_wait_for_events(wait_list, ierr)

              !Call OpenCL tridiagonal solve kernel
              CALL ocl_enqueue_kernel_ndrange(OCL_PERFORMANCE_DEVICE, &
                 'program' // C_NULL_CHAR, 'pcr' // C_NULL_CHAR, &
                 1, global_offset, global_size, local_size, event, ierr)

              !Read solution from solution buffer x
              CALL ocl_enqueue_read_buffer(OCL_PERFORMANCE_DEVICE, &
                   d_x, 1, global_offset, readbytes,C_LOC(GPUQ), event,ierr)
```

(b)

Figure 5.3: (a) Shows the original BoxMG line relaxation (in x) code. (b) shows the first attempt at using the GPU for line relaxation. Notice that each line requires 4 write calls to the GPU before the kernel is called.

data structure that was formatted specifically for the GPU tridiagonal solve kernel, and write all diagonal elements in a single call. These structures are LOWERX, MAINX, and UPPERX; they also exist for lines in y with Y replacing X in the variable name. The storage of these values is done in the setup phase, eliminating unnecessary memory copying and, in the relaxation phase, we are able to store the right-hand sides in a GPU specific vector called GPUQ1. Again, after invoking the kernel, the solution to the systems had to be read back and copied to host-side data structures in the proper format. These modifications are shown in Figure 5.4. Unfortunately, even by minimizing the number of write calls necessary for a single color, using the GPU for line relaxation was still several times slower than performing sequential line relaxation on the CPU.

5.5 Minimizing Data Transfer

Despite the fact that the GPU kernel is capable of relaxing several lines at once substantially faster than relaxing them sequentially on the CPU, after seeing no performance improvement from writing the necessary matrix data for each color, it was obvious that the cost of reading and writing data to and from the device would dominate the time required for line smoothing on the GPU. In an attempt to eliminate all write and read calls from the relaxation step, after assessing the data structures necessary for line relaxation, the clear solution was to force more data to reside on the GPU. As previously stated, the only values necessary for line relaxation are the stencil of the particular grid, SO, the current right-hand side of the equation we are attempting to solve, QF, and the current approximation, Q. If, instead of formatting the matrices on the host and writing them to the device, we store these values on the GPU and construct the matrices on the device, we would eliminate all unnecessary data transfer in the relaxation step.

Refactoring the code to minimize data movement required a few modifications. The first change implemented was writing the stencil for all accelerated grids to the GPU in the setup phase. A new data structure on the host was created to keep track of the starting position of each grid on the device-side buffers, since the multi-dimensional host data structures are stored as one dimensional vectors on the device. This data structure was used to tell the

```
!Loop over red lines, then black lines
DO JBEG=JBEG_START, JBEG_END, JBEG_STRIDE
    JEND=2*((J1-JBEG)/2)+JBEG
    OFFSET=0
    DO J=JBEG,JEND,2
       DO I=2,I1
          GPUQ1(OFFSET*(II-2)+(I-1))=QF(I,J) &
               +SO(I,J,KS)*Q(I,J-1)+SO(I,J+1,KS) &
               *Q(I,J+1)+SO(I,J,KSW)*Q(I-1,J-1)+SO(I+1,J,KNW) &
               *Q(I+1,J-1)+SO(I,J+1,KNW)*Q(I-1,J+1) &
               +SO(I+1,J+1,KSW)*Q(I+1,J+1)
       ENDDO
       OFFSET=OFFSET+1
    ENDDO
    CALL ocl_clear_event_wait_list(wait_list, ierr)

    !determine if red or black lines
    IF ((JBEG_STRIDE==-1.AND.JBEG==3).OR. &
          (JBEG_STRIDE==1.AND.JBEG==3))THEN
       DIM=1
    ELSE
       DIM=2
    ENDIF

    diagptr=C_LOC(LOWERX(GOFFSET+1,DIM))
    CALL ocl_enqueue_write_buffer(OCL_PERFORMANCE_DEVICE, &
       d_a, 1, foo, bytes, diagptr, event, ierr)
    CALL ocl_add_event_to_wait_list(wait_list, event, ierr)

    diagptr=C_LOC(MAINX(GOFFSET+1,DIM))
    CALL ocl_enqueue_write_buffer(OCL_PERFORMANCE_DEVICE, &
       d_b, 1, foo, bytes, diagptr, event, ierr)
    CALL ocl_add_event_to_wait_list(wait_list, event, ierr)

    diagptr=C_LOC(UPPERX(GOFFSET+1,DIM))
    CALL ocl_enqueue_write_buffer(OCL_PERFORMANCE_DEVICE, &
       d_c, 1, foo, bytes, diagptr, event, ierr)
    CALL ocl_add_event_to_wait_list(wait_list, event, ierr)

    diagptr=C_LOC(GPUQ1)
    CALL ocl_enqueue_write_buffer(OCL_PERFORMANCE_DEVICE, &
       d_d, 1, foo, bytes, diagptr, event, ierr)
    CALL ocl_add_event_to_wait_list(wait_list, event, ierr)

    CALL ocl_wait_for_events(wait_list, ierr)

    CALL ocl_enqueue_kernel_ndrange(OCL_PERFORMANCE_DEVICE, &
      'program' // C_NULL_CHAR, 'pcr' // C_NULL_CHAR, &
     1, global_offset, global_size, local_size, event, ierr)

    call ocl_finish(OCL_PERFORMANCE_DEVICE)

    diagptr = C_LOC(GPUQ1)
    CALL ocl_enqueue_read_buffer(OCL_PERFORMANCE_DEVICE, &
         d_x, 1, foo, bytes, diagptr, event, ierr)
```

Figure 5.4: The second modification to line relaxation. Diagonals for the tridiagonal kernel are stored in the proper format in the setup phase, and the right-hand sides are calculated and stored in the proper format in GPUQ1. Only 4 write calls total are necessary for a single color.

```
__kernel void rhs_x(__global real_t *d_qf, __global real_t *d_q,
    __global real_t *d_d, __global real_t *d_sos, __global real_t *d_sosw,
    __global real_t *d_sonw, int system_size, int num_systems, int sooffset,
    int jbeg, int stencilnum) {

  size_t thid = get_local_id(0);
  size_t blid = get_group_id(0);
  int ssp2= system_size+2;
  int aoff = thid+1+(2*blid+jbeg-1)*ssp2;

  d_d[thid+blid*system_size]=d_qf[sooffset+aoff]+
    d_sos[aoff+sooffset]*d_q[sooffset+aoff-ssp2]+
    d_sos[aoff+sooffset+ssp2]*d_q[sooffset+aoff+ssp2]+
    d_sosw[sooffset+aoff]*d_q[sooffset+aoff-ssp2-1]+
    d_sonw[sooffset+1+aoff]*d_q[sooffset+1+aoff-ssp2]+
    d_sonw[aoff+ssp2+sooffset]*d_q[sooffset+aoff+ssp2-1]+
    d_sosw[1+aoff+ssp2+sooffset]*d_q[sooffset+1+aoff+ssp2];
}
```

Figure 5.5: GPU kernel for calculating the right-hand sides of the tridiagonal systems for line relaxation in x. Notice the similarities to the CPU code; namely, the calculation is the same, the loop is simply unrolled to calculate each value with a single thread.

kernel the number of indices needed to offset by in the stencil vectors to access the proper data for a given accelerated grid. Two new tridiagonal solve kernels were written for lines in x and y due to the different indexing necessary in each direction. Minor modifications were also made to write the solution to the device-side Q vector instead of x.

New kernels were also written for calculating the right-hand sides of the tridiagonal systems in both x and y on the device before invoking the solve kernel. As we have seen, right-hand side calculation on the CPU is implemented as a simple pointwise calculation. On the GPU kernel, the same calculation was implemented, except instead of calculating points sequentially, we spawn a thread to calculate the value of each point on the right-hand side, thus exposing a large amount of fine-grained parallelism.

The changes made to the tridiagonal solve kernel are shown in Figure 5.6, while the new kernel for calculating right-hand sides is shown in Figure 5.5. Both figures show the kernel for calculations in x—the y kernels are identical except for the indexing into device-side data structures.

On the host side, to account for the new kernels, all read and write calls were removed from the relaxation code. Instead, new functions were added that use OCL-MLA to set grid-dependent arguments for the modified kernels, such as size of the tridiagonal systems to be solved and the number of indices to offset in the stencil buffers. The changes to the host-side line relaxation code are shown in Figure 5.7.

By allowing the stencil of our discrete operator to reside on the GPU, and by keeping

```
__kernel void pcr_branch_free_kernel(__global real_t *a_d,
   __global real_t *b_d, __global real_t *c_d, __global real_t *d_d,
   __global real_t *x_d, __local real_t *shared, int system_size,
   int num_systems, int iterations) {

  size_t thid = get_local_id(0);
  size_t blid = get_group_id(0);
  int delta = 1;

  __local real_t * a = shared;
  __local real_t * b = &a[system_size+1];
  __local real_t * c = &b[system_size+1];
  __local real_t * d = &c[system_size+1];
  __local real_t * x = &d[system_size+1];

  //load the relevant matrix data into shared memory
  a[thid] = a_d[thid + blid * system_size];
  b[thid] = b_d[thid + blid * system_size];
  c[thid] = c_d[thid + blid * system_size];
  d[thid] = d_d[thid + blid * system_size];

  //Perform parallel cyclic reduction to solve systems...
  ...
  //Store solution in device side buffer x_d
  x_d[thid + blid * system_size] = x[thid];
}
```

(a)

```
__kernel void pcr_branch_free_kernel_x(__global real_t *a_d,
   __global real_t *b_d, __global real_t *d_d,
   __global real_t *d_q, __local real_t *shared, int system_size,
   int num_systems, int iterations, int sooffset, int jbeg) {
  //a_d is western stencil values, b_d is central, d_d is from
  //the RHS kernel, d_q is device side Q vector


  size_t thid = get_local_id(0);
  size_t blid = get_group_id(0);

  int delta = 1;

  __local real_t * a = shared;
  __local real_t * b = &a[system_size+1];
  __local real_t * c = &b[system_size+1];
  __local real_t * d = &c[system_size+1];
  __local real_t * x = &d[system_size+1];

  //Calculate the offset into stencil for matrix diagonals
  int aoff = sooffset + thid + 1 + (2*blid+jbeg-1) * (system_size+2);

  //Load the relevant matrix data into shared memory
  a[thid] = -a_d[aoff]; //subdiagonal
  b[thid] = b_d[aoff];  //main
  c[thid] = -a_d[aoff+1]; //super
  d[thid] = d_d[thid+blid * system_size]; //RHS

  //Perform parallel cyclic reduction to solve systems...
  ...
  //Store solution in device side current solution d_q
  d_q[aoff] = x[thid];
}
```

(b)

Figure 5.6: (a) Shows the original kernel for solving tridiagonal systems. (b) Shows the modified kernel for smoothing lines in x based on using the stencil to construct the tridiagonal systems on the device.

```
!Loop over red lines, then black lines
DO JBEG=JBEG_START, JBEG_END, JBEG_STRIDE

   !Set params for RHS kernel
   CALL BMG2_SymStd_WRITE_RHS(II, JJ, I1-1, (J1-1)/4, &
        JBEG, OFFSETS(KF-K+1), .TRUE., 9)

   global_offset=0
   !build RHS of our tridiagonal systems
   CALL ocl_enqueue_kernel_ndrange(OCL_PERFORMANCE_DEVICE, &
      'program' // C_NULL_CHAR, 'rhsx' // C_NULL_CHAR, &
      1, global_offset, global_size, local_size, event, ierr)

   !Set params for tridiagonal solve kernel
   call BMG2_SymStd_MidSetup_OCL(I1-1,(J1-1)/4, &
        JBEG, OFFSETS(KF-K+1), .TRUE.)

   !Solve tridiagonal systems
   CALL ocl_enqueue_kernel_ndrange(OCL_PERFORMANCE_DEVICE, &
      'program' // C_NULL_CHAR, 'pcrx' // C_NULL_CHAR, &
      1, global_offset, global_size, local_size, event, ierr)
   CALL ocl_finish(OCL_PERFORMANCE_DEVICE)

ENDDO
```

Figure 5.7: Read and write free host-side line relaxation code.

Q on the GPU for the entirety of line relaxation, we greatly reduce the amount of time spent on transferring data to the GPU; specifically, as we travel down the V-cycle, we must only write QF and Q for the current grid, relax on the GPU, then read Q back from the GPU before residual calculation and restriction. On the way up, we must only write Q for a given accelerated grid, then read Q back after relaxation. By minimizing data transfer to the GPU in this way, we achieve good performance improvements by accelerating line relaxation with the GPU.

5.6 Residual Calculation

In the efficient implementation of line relaxation above, on each accelerated grid we must read Q from the device in order to calculate the residual $f - A^h \boldsymbol{v}^h$. As we have already discussed, however, the only values necessary for calculating the residual are QF, Q, and SO, which already reside on the GPU. Also, residual calculation is performed pointwise much like right-hand side calculation for the tridiagonal systems; hence, by using a single thread for calculating the residual at each point we can gain substantial performance improvements by implementing residual calculation on the GPU. Figure 5.8 shows the original CPU implementation of residual calculation, the new CPU code that invokes the kernel, and the GPU kernel.

By implementing line relaxation and residual calculation on the GPU, we accelerate the majority of the computation of the BoxMG solve step, essentially using the CPU code for setup, restriction, interpolation, control of data movement and kernel invocation. By structuring the distribution of data so that only one read and write are necessary at each grid in our V-cycle, we gain good efficiency and performance improvements over the sequential BoxMG code.

```
      DO J=2,J1
         DO I=2,I1
            RES(I,J) = QF(I,J)
     &               + SO(I   ,J   ,KW )*Q(I-1,J)
     &               + SO(I+1,J   ,KW )*Q(I+1,J)
     &               + SO(I   ,J   ,KS )*Q(I   ,J-1)
     &               + SO(I   ,J+1,KS )*Q(I   ,J+1)
     &               + SO(I   ,J   ,KSW)*Q(I-1,J-1)
     &               + SO(I+1,J   ,KNW)*Q(I+1,J-1)
     &               + SO(I   ,J+1,KNW)*Q(I-1,J+1)
     &               + SO(I+1,J+1,KSW)*Q(I+1,J+1)
     &               - SO(I   ,J   ,KO )*Q(I   ,J)
         ENDDO
      ENDDO
```

(a)

```
         !use the GPU for the residual...setup
         ssize=J1-1
         !determine offset for GPU kernel indexing
         sooffset=OFFSETS(GRIDNUM+1)

         !set GPU kernel arguments
         call ocl_set_kernel_arg_int('program' // C_NULL_CHAR, &
              'res' // C_NULL_CHAR, 8, sooffset, ierr)
         call ocl_set_kernel_arg_int('program' // C_NULL_CHAR, &
              'res' // C_NULL_CHAR, 9, 9, ierr)
         call ocl_set_kernel_arg_int('program' // C_NULL_CHAR, &
              'res' // C_NULL_CHAR, 10, ssize, ierr)

         !enqueue residual kernel
         global_offset=0
         global_size=(J1-1)*(I1-1)
         local_size=I1-1
         off = OFFSETS(GRIDNUM+1)*sizeof(RES(1,1))

         CALL ocl_enqueue_kernel_ndrange(OCL_PERFORMANCE_DEVICE, &
              'program' // C_NULL_CHAR, 'res' // C_NULL_CHAR, &
              1, global_offset, global_size, local_size, event, ierr)
         call ocl_finish(OCL_PERFORMANCE_DEVICE)

         !read result from GPU
         diagptr = C_LOC(RES(1,1))
         bytes=II*JJ*sizeof(RES(1,1))
         CALL ocl_enqueue_read_buffer(OCL_PERFORMANCE_DEVICE, &
              d_res, 1, foo, bytes, diagptr, event, ierr)
         call ocl_finish(OCL_PERFORMANCE_DEVICE)
```

(b)

```
__kernel void residual(__global real_t *d_qf, __global real_t *d_q,
   __global real_t *d_sow, __global real_t *d_sos, __global real_t *d_sosw,
   __global real_t *d_sonw, __global real_t *d_soo, __global real_t *d_res,
   int sooffset, int stencilnum, int systemsize)  {

  //we have I1-1 threads, systemsize blocks.
  size_t thid = get_local_id(0);
  size_t blid = get_group_id(0);
  int ssp2 = systemsize+2;
  int aoff = thid+1+sooffset+(blid+1)*(ssp2);

  d_res[aoff] = d_qf[aoff] +
    d_sow[aoff]*d_q[aoff-1]+d_sow[aoff+1]*d_q[aoff+1] +
    d_sos[aoff]*d_q[aoff-ssp2]+d_sos[aoff+ssp2]*d_q[aoff+ssp2] +
    d_sosw[aoff]*d_q[aoff-1-ssp2]+d_sonw[aoff+1]*d_q[aoff+1-ssp2] +
    d_sonw[aoff+ssp2]*d_q[aoff-1+ssp2] +
    d_sosw[aoff+1+ssp2]*d_q[aoff+1+ssp2] - d_soo[aoff]*d_q[aoff];
}
```

(c)

Figure 5.8: (a) Shows the original CPU code for residual calculation (b) Shows the modified CPU code for calling the GPU kernel for residual calculation (c) Shows the residual GPU kernel

Chapter 6

Numerical Results

All of the tests performed in this section are for a 9-point finite volume discretization of Laplace's equation, discretized on a uniform square grid. We have seen that BoxMG is suitable for much more difficult problems, but, as our concern is with performance testing, this problem is sufficient to analyze the timings of BoxMG components.

6.1 BoxMG Profiling

The percentage of time taken in the BoxMG solve step for the major parts of the V-Cycle is shown in Table 6.1. These percentages disregard the pre-solve setup phase necessary for BoxMG, which includes specifying the fine-grid discrete operator, while the "Other" calculations include the cost of constructing the coarse-grid and interpolation operators, as well as the cost of interpolation and restriction during the V-Cycle. It is worth noting that the additional pre-solve setup time necessary for GPU accelerated BoxMG is approximately .003 seconds.

These results suggest a significant portion of the computation during the solve step is spent on line relaxation and residual calculation. We also notice that calculating the right-hand sides of the tridiagonal systems for lines in y takes significantly longer than lines in x. This is because this percentage includes the time necessary for copying the solution from the LAPACK tridiagonal solve routine into the current approximation, Q.

Table 6.1: BoxMG Profiling Results

Problem Size	RHS Y	RHS X	Tridiagonal Solve	Residual	Other
512×512	37.1%	13.7%	16.1%	19.0%	14.1%
1024×1024	45.8%	13.7%	14.2%	16.9%	9.4%

6.2 Amdahl's Law

We will first establish bounds on the maximum possible performance gains from accelerating BoxMG. Consider parallelizing a program where a fraction of total runtime, k, of the program is parallelizable, while the remainder, $j = 1 - k$, is inherently sequential.

Imagine that we implement the parallelizable portion on 2 processors and it performs exactly twice as fast. This will generally not be the case, but if it were possible, we would expect that our code would run $\frac{1}{j+\frac{k}{2}}$ times faster than the original code. Similarly, if we could implement the parallelizable portion of our code on hundreds or thousands of processors, with perfect scaling, we would expect that our code would run about $\frac{1}{j}$ times faster than the original code, since the running time of the parallel portion would be negligible. For any number of processors between, we would expect some improvement proportional to the number of parallel processors we use.

We can formalize the maximum speedup we can expect from parallelizing our code with the following formula:

$$S_N = \frac{N}{k + (1-k)N}, \tag{6.1}$$

where N is the number of parallel processors used, and k is defined as above. The relationship between performance and CPU time is generally referred to as *Amdahl's law*, since he recognized the connection in [4], whereas (6.1) is often referred to as *Ware's Law*, since he was the first to express this relationship as an explicit formula [14, 16]. Figure 6.1 shows (6.1) plotted for several k values. Using (6.1) and our BoxMG profiling results, we see that the maximum performance gains we could achieve are a 7.1× speedup for a 512×512 problem, and a 10.6× speedup for a 1024×1024 problem.

Now that we have a good idea of the maximum possible performance improvement we could see from parallelizing a portion of BoxMG, we have a good method for evaluating our accelerated implementation. In practice, however, it would be nearly impossible to achieve

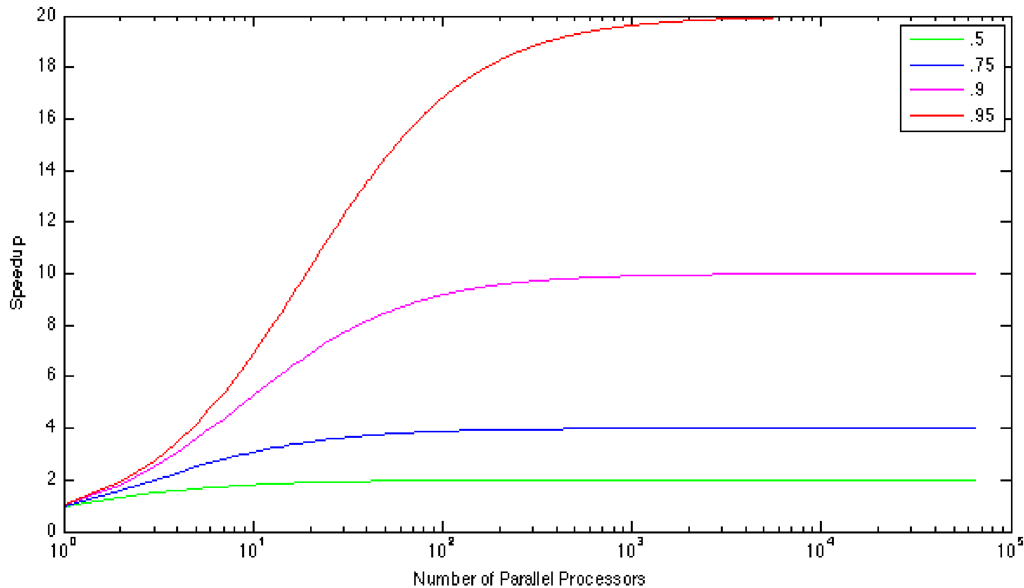


Figure 6.1: Ware's law plotted for several k values.

these maximum gains due to the necessary overhead of parallelizing a significant portion of the code, and the communication costs associated with GPU hardware.

6.3 Hardware

The accelerated BoxMG code was tested on a variety of different hardware, including discrete GPUs, integrated CPU-GPU architectures, and multi-core CPUs. In the following numerical results, we will refer to these devices as follows:

- **CPU** Serial implementation of BoxMG on an AMD Opteron 6168 CPU
- **AMD** Accelerated BoxMG on a 4×12-core AMD Opteron 6168 CPU
- **ATI** Accelerated BoxMG on an ATI Radeon HD 5800 Series GPU, with an AMD Opteron 6168 CPU
- **NVIDIA** Accelerated BoxMG on an NVIDIA Tesla M2090 GPU, with an AMD Opteron 6168 CPU
- **FUSION** Accelerated BoxMG on an AMD A8 Series APU

As we have already discussed, by specifying the architecture of the accelerated device, OCL-MLA allows us to run the accelerated BoxMG code on a variety of different hardware without any modification to the source code. All code was tested in both single and double precision to investigate potential advantages of mixed precision on manycore architectures, using a stopping tolerance of an absolute residual in the L_2 norm of 10^{-18}. For nontrivial right-hand sides, single precision BoxMG will not converge to this tolerance due to numerical roundoff, hence a zero right-hand side was used to ensure a significant run time in single precision.

6.4 Kernel Performance

In general, the GPU kernel implementation of the right-hand side calculation, tridiagonal solves, and residual calculation significantly outperformed their serial CPU implementation. In these tests, it is worth noting that the AMD Fusion does not support double precision, so only single precision was tested on this architecture. Also, problem size is limited by the maximum number of threads per block allowed to be launched by the GPU kernel, or, in the case of the Nvidia Tesla in double precision, the maximum amount of shared memory per block. Efforts to accelerate BoxMG on larger problems is the subject of future work.

All timings reflect the total duration of the actual computation performed during kernel execution. "Queued" time (time that the kernel waits on the host to be dispatched to the device), and "invocation" time (time the kernel waits on the device before being executed) were disregarded. In general, queued and invocation times vary by device and are independent of the kernel being executed and the size of the thread pool. Average queued and invocation times for the devices tested are shown in Figure 6.2. For comparison, consider that right-hand side calculation for a 256X256 problem in double precision takes ~250μs on the Nvidia Tesla. We will see that for the ATI Radeon (ATI) and the AMD Fusion (FUSION), queued and invocation time for the GPU kernel is on the order of the duration of the computation time of the largest problems. The Nvidia Tesla (NVIDIA) does not suffer from this problem; queued and invocation times are negligible when compared to computation time even on relatively small problems. This suggests that the ATI and FUSION devices

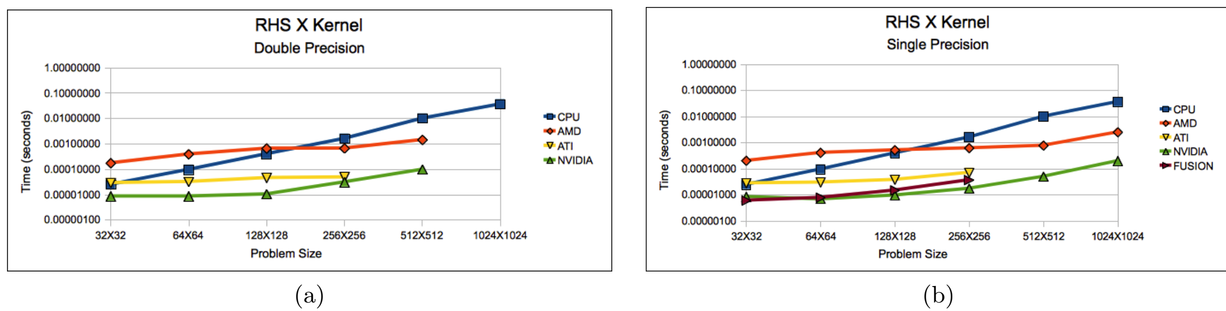
Table 6.2: Device Queued and Invocation Times (μs)

Step	AMD	ATI	NVIDIA	FUSION
Queued	54.78	44.52	3.27	27.71
Invocation	52.03	156.87	6.00	182.58

would benefit from either reducing the time it takes to dispatch kernels from the host, or calling kernels directly from the device without interaction with the host.

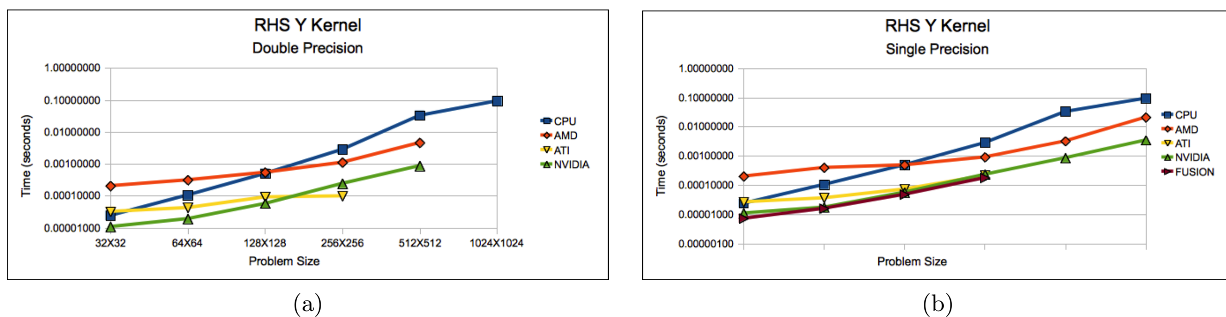
In the following sections, the specific values used to generate the timing plots are included in Appendix A. These values are worth considering when comparing single and double precision implementations on the same architecture, since the logarithmic scaling in the plots makes these subtle differences less apparent.

6.4.1 Right-Hand Side Kernel



(a) (b)

Figure 6.2: Right-hand Side in X Kernel Timings



(a) (b)

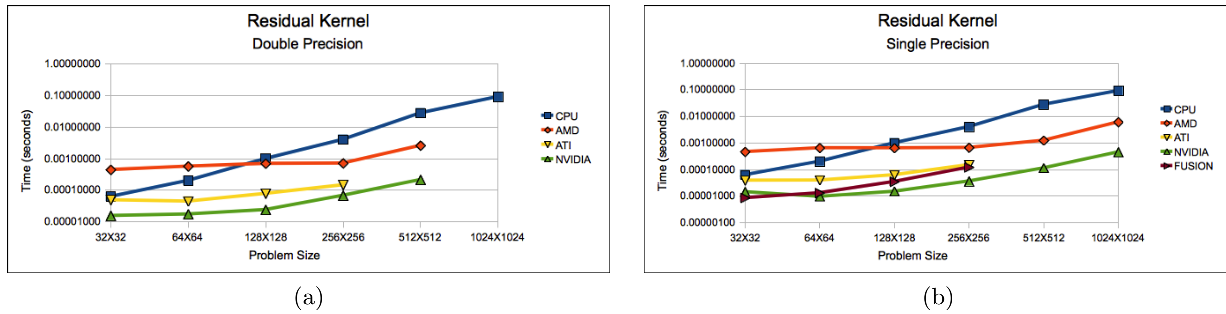
Figure 6.3: Right-hand Side in Y Kernel Timings

Figure 6.2 and 6.3 show plots of the timing results for the execution of the kernel for calculating the right-hand sides of the tridiagonal systems, on all hardware, and in serial

on the CPU. The GPU implementations outperformed the serial CPU implementation on nearly all grids, while the multi-core CPU performed poorly for small problem sizes, likely due to the high overhead of launching hundreds of threads on the CPU as compared with the relatively small computation each thread performed. As a result, for right-hand side calculations in X, the multi-core CPU time was dominated by overhead until relatively large grids where it gained efficiency over the serial code. Also, the thread topology implemented by all kernels in the accelerated code was designed for GPUs, and it is likely that a different topology designed specifically for multi-core CPUs would exhibit significantly better scaling.

The GPU implementations of right-hand side calculation for lines in x exhibit relatively flat scaling for very small problem sizes. We notice this phenomenon in other kernels, and it is due to the fact that we gain maximum efficiency from the GPU when several thousand threads are launched. This is because we need a sufficiently large thread pool to saturate the SMs of the GPU and to hide memory access latency; in the case of right-hand side calculation in x, we don't realize a thread pool of this magnitude until we approach a problem size of 128×128.

Right-hand sides for lines in y exhibit significantly worse performance than those in x due to the alignment of data on the device. The stencil written to the device is in column-major order; as a result, more values accessed by the kernel for right-hand sides in x are adjacent in device memory than those for lines in y. This means that the cost of right-hand side calculation in x is dominated by computation, whereas the cost in y is dominated by memory accesses. For this reason, the computation time of right-hand sides in y generally varies with the problem size, whereas in x, computation time is more dependent on the size of the thread pool. Similarly, calculation of right-hand sides in x on the Nvidia GPU are faster in single-precision, whereas there is no benefit in using single precision in y since the computation is memory bound. In this case, there also seems to be little advantage to using single precision on the ATI GPU for right-hand sides in x. In all kernel implementations, there is no benefit from using single precision on the multi-core CPU since it has no single-precision registers.



(a) (b)

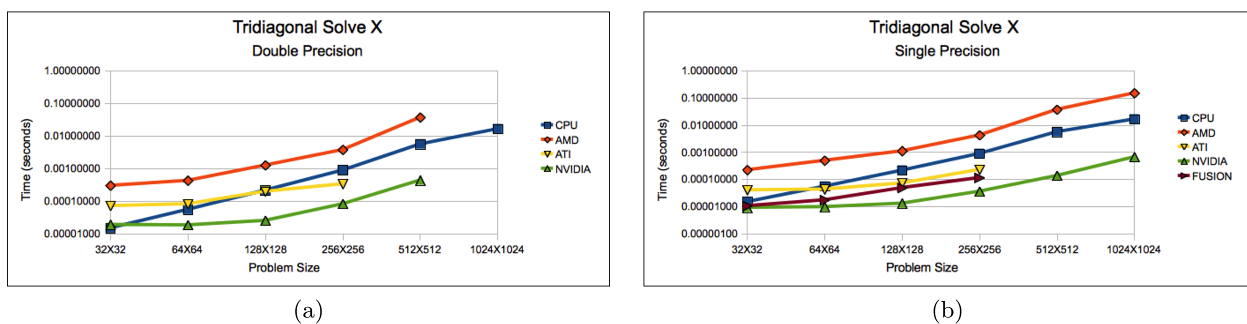
Figure 6.4: Residual Kernel Timings

6.4.2 Residual Kernel

Timing results for the residual calculation kernel, on all devices, and in serial on the CPU, are shown in Figure 6.4. These results suggest similar trends and efficiency as right-hand side calculation in x, which we might expect since the kernels are very similar.

Again, we notice the multi-core CPU implementation is dominated by the overhead of spawning many threads on the CPU for small problem sizes. Also, the GPU implementation takes a comparable amount of time for small problems until we sufficiently saturate the device with computation. As we have seen with right-hand side calculation, the GPU kernel is approximately twice as fast in single precision on the Nvidia GPU, while there is no performance benefit from single-precision on the ATI and multi-core CPU.

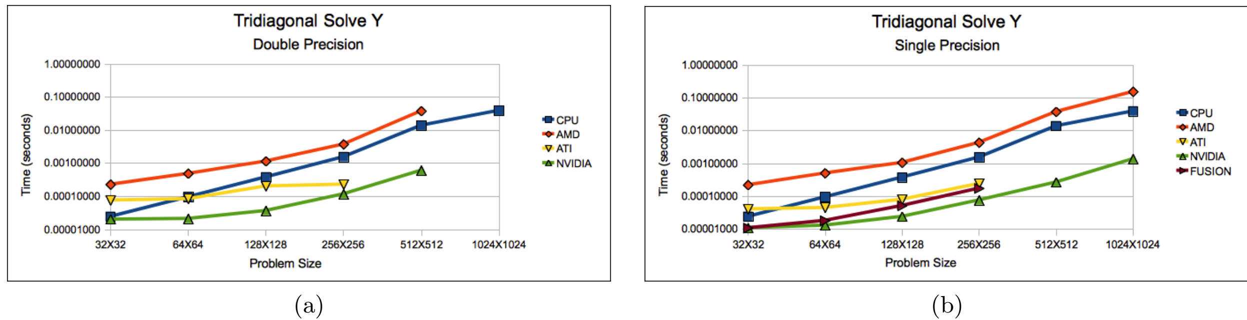
6.4.3 Tridiagonal Solve Kernel



(a) (b)

Figure 6.5: Tridiagonal Solve in X Kernel Timings

Figure 6.5 and 6.6 show the timing results for the tridiagonal solve kernel, on all devices, and in serial on the CPU. In this case, the GPU implementation outperformed the serial



(a) (b)

Figure 6.6: Tridiagonal Solve in Y Kernel Timings

CPU implementation on nearly all grids, while the multi-core CPU implementation failed to outperform the serial CPU computation on any grid. This is because the topology of the thread pool in this case makes very little sense on a multi-core CPU architecture. If, instead of spawning a single thread for each line of the tridiagonal systems, we spawn only 48 threads, each of which solve a fraction of the tridiagonal systems serially, we would potentially see significant performance improvements. It is thus worth investigating kernel implementations that depend on the accelerated hardware used to achieve efficiency on a variety of architectures.

In these results, we again notice relatively flat scaling of the GPU performance until the device has a sufficiently large thread pool. Despite the fact that the serial CPU implementation is roughly twice as slow for lines in y versus lines in x, the GPU is slightly less affected by line direction, owing only minor performance degradation in y to memory indirection due to the column-major ordering of device-side data. Also, in general, there is about a $2\times$ speedup on both the ATI and Nvidia GPUs using single precision.

6.4.4 Kernel Speedups

In nearly all tests, there were significant performance gains from using the GPU for major components of the BoxMG V-Cycle, especially for larger grids. Of all the hardware tested, we achieved the best performance from the Nvidia GPU, both in total computational time, and in queued and invocation time. On this device, speedups of upwards of $100\times$ over the serial CPU implementation of the kernels were possible, even when considering the necessary overhead of interaction between the host and device. Table 6.3 shows the overall speedup

Table 6.3: Overall Speedup of GPU Kernel Implementation Over Sequential CPU

Precision	RHS Y	RHS X	Tridiagonal Solve X	Tridiagonal Solve Y	Residual
Double	94×	37×	12×	22×	124×
Single	173×	26×	24×	28×	199×

of the GPU kernel implementation of right-hand side calculation, tridiagonal solves, and residual calculation, over the serial CPU implementation, for the largest grids, in both single and double precision, including queued and invocation time.

6.5 Aggressive Coarsening

In all of the tests performed in this chapter thus far, we used one relaxation sweep of lines in x and y, on all grids, on both the way up and down the V-Cycle, with standard coarsening by a factor of two. We have seen, in Chapter 2, the smoothing property of relaxation; that is, relaxation damps oscillatory components of the error, while smooth components remain. Given the remarkable efficiency of relaxation on the GPU, it seemed that if we could smooth a larger portion of the error spectrum by performing more relaxation sweeps, and coarsen by a factor of four, we may be able to improve the performance of BoxMG over coarsening by two with one relaxation sweep.

Unfortunately, despite improving the convergence rate of BoxMG, no combination of coarsening by a factor of four and increasing the number of relaxation sweeps improved the overall solve time versus coarsening by a factor of two with one relaxation sweep. The results of coarsening by a factor of 4 with more relaxation sweeps are shown in Table 6.4, where, for example, A,B denotes coarsening by a factor of A, with B relaxation sweeps.

Table 6.4: Performance of Increased Relaxation Sweeps and Agressive Coarsening

	2,1	2,2	4,1	4,2	2,5	4,5	4,10	2,10
V-Cycles Necessary	18	16	26	20	13	16	14	12
Time to Convergence (s)	1.33	1.44	1.54	1.57	1.75	1.76	2.35	2.74

It is also interesting to note that the run time of the 2,1 and 4,1 cycles exactly match their theoretical running times. That is, if we equate the amount of work done on a grid

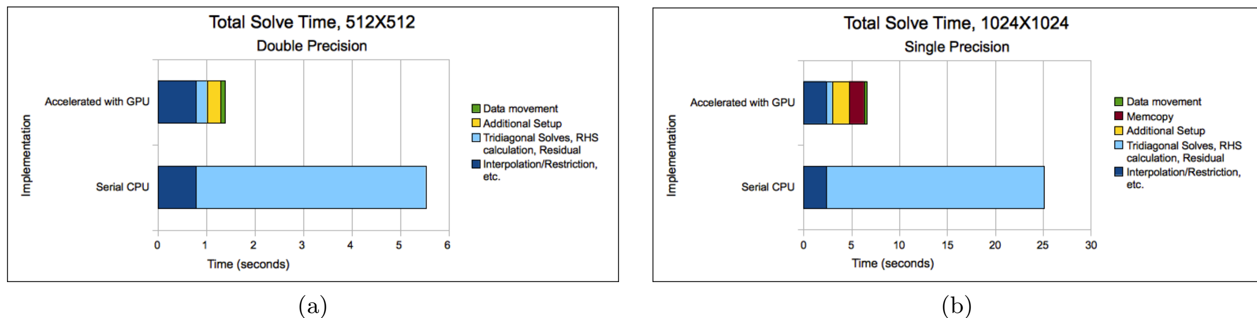
with the number of degrees of freedom it has, then coarsening by a factor of two yields a V-Cycle with complexity $1 + \frac{1}{4} + \frac{1}{16} + \ldots = \frac{4}{3}$, whereas coarsening by a factor of four yields a complexity of $1 + \frac{1}{16} + \frac{1}{256} + \ldots = \frac{16}{15}$. We would then expect the ratio of the running time of the schemes to equal the ratio of the complexity of their V-Cycles, times the number of cycles each completed, or

$$\frac{T_{2,1}}{T_{4,1}} = \frac{18(\frac{4}{3})}{26(\frac{16}{15})}. \tag{6.2}$$

In fact, substituting, in turn, the calculated values for the time to convergence of 2,1 and 4,1 in (6.2) yields the predicted times $T_{4,1} = 1.54$ for measured $T_{2,1} = 1.33$, and $T_{2,1} = 1.33$ for measured $T_{4,1} = 1.54$, exactly matching their experimental values.

6.6 Overall Performance Improvements

The significant performance improvement of the GPU kernel computations over the serial CPU implementation allowed us to achieve remarkable gains within the scope of these computations. Unfortunately, the only architecture on which good performance improvements over the entire BoxMG solve step were achieved, was the Nvidia GPU. This is because, despite the improvement in solve times made by the GPU kernels, the problems we were able to test on the other devices were too small for the kernel improvements to outweigh the overhead of using the GPU, especially considering the extremely long queued and invocation times on these devices. Despite the disappointing performance of the other hardware, we were able to achieve good performance improvements over the entire solve step using the Nvidia GPU.



(a) (b)

Figure 6.7: Total Solve Time of BoxMG

Total time to convergence for both the serial and GPU accelerated implementation (on the Nvidia GPU) of BoxMG are shown in Figure 6.7. The double precision implementation achieved a speedup of 4× over the serial implementation, while the single precision implementation achieved a 3.78× speedup.

A significant amount of time in both implementations is spent on transferring data to and from the device. In the future, it will likely become advantageous for manufacturers to move towards architectures with a CPU and GPU on the same die; such is already the case with the AMD Fusion and Intel Ivy Bridge architectures. Currently, there is no mechanism for using the same address space for both the CPU and GPU; however, the ability to share address space between devices will be a feature in later OpenCL iterations. This feature will ultimately remove the unneccessary transfer of data between host and device, allowing for even better performance of the GPU accelerated code. Also, the additional setup necessary for accelerating BoxMG includes formatting the host side data properly for the device, and writing the stencil for all accelerated grids—similarly, this additional setup step would benefit from not having to write a significant amount of data to the device.

In addition to transferring data between host and device, much of the overall computation time in single precision was dedicated to copying the single precision data received from the device into the double precision data structures on the host. This time consuming procedure could be eliminated if we had used a single precision implementation of BoxMG. Unfortunately, a single precision implementation in general cannot converge below a residual of 10^{-8}. If, however, single precision BoxMG is used as a preconditioner within the conjugate gradient algorithm, it would be possible to achieve significant performance improvements over the double precision code in this context.

Chapter 7

Concluding Remarks

We have seen that the BoxMG algorithm contains many fine-grained data parallel operations that are conducive to efficient implementation on the GPU. Also, the logically rectangular mesh construction circumvents many of the indirect addressing issues of algebraic multigrid methods, allowing arithmetic operations to dominate the cost of the accelerated implementation, rather than memory indirection. As a result, the GPU kernel implementations of line relaxation and residual calculation achieve speedups of more than an order of magnitude over their serial CPU implementations. This improvement yielded an overall speedup of $4\times$ in the BoxMG solve step, despite the significant overhead necessary for CPU-GPU interaction.

In the future, architectural trends could favor designs that would mitigate the overhead necessary for GPU acceleration, allowing even more significant performance improvements of the accelerated BoxMG code, with minimal modifications. By minimizing this overhead, the accelerated implementation could quickly achieve speedups close to the theoretical bounds predicted by Amdahl's law. It is also worth investigating the use of single-precision accelerated BoxMG as a preconditioner to exploit the efficiency of single-precision arithmetic on GPUs.

7.1 Future Work

Despite the significant performance improvements of BoxMG with GPU acceleration, the thread structure of the tridiagonal solve kernel prevented us from solving systems whose order was larger than the maximum number of threads per block allowed by the device. This prevented us from achieving significant performance improvements of the overall solve step on AMD hardware, and confined us to solving problems of 1 million degrees of freedom or smaller. Though the other kernels were also confined by this restriction, they could easily be refactored to solve larger systems by reducing the size of each thread block and indexing properly into device memory.

Unfortunately, allowing larger thread block sizes is not a viable solution to the issue of solving larger tridiagonal systems, since, for larger systems, it is likely that we will be confined by the maximum allowable shared memory for a block; such is already the case on the Nvidia GPU in double precision. It is, thus, necessary to construct a new kernel that is independent of the maximum number of threads allowed per block. A kernel based on the parallel line solve algorithm described in Chapter 3 is currently being implemented, and will be incorporated into the accelerated BoxMG code soon. This kernel effectively reduces very large tridiagonal systems to many small systems, in the same way as shown in Figure 3.11, and uses the current tridiagonal solver to solve the many small "local" systems, in parallel, on the GPU. With the addition of this kernel, and slight modifications to the right-hand side and residual calculation kernels, it is expected that BoxMG will achieve similar significant performance improvements for large problems.

In addition to solving larger problems in a single CPU-GPU environment, future improvements to the BoxMG code will include accelerating BoxMG in an MPI/OpenCL environment, ideally using GPUs to accelerate each MPI rank, as described in Chapter 4. This will include the acceleration of the parallel line solving algorithm, and potentially a kernel implementation of this algorithm for the large local systems, as described above, though the stability of a nested reduction in this manner could be problematic. Local residual and right-hand side calculation, however, could be implemented much in the same way as the serially accelerated code.

It is also desirable to accelerate a 3D implementation of BoxMG. In 3D, BoxMG implements plane relaxation, which effectively reduces to line relaxation of a series of planes in the xy, yz, and xz directions. Accelerating this implementation is seemingly quite feasible, since the 2D BoxMG code is used for many of the 3D subroutines. To achieve efficiency and minimize data transfer, however, it is necessary to store the pointwise stencil of the discrete problem on the GPU. Even for moderately sized 3D problems, this stencil could quickly approach the maximum amount of memory on the device. Also, if small enough problems are solved that the GPU can store the entire stencil, it is not likely that using the GPU will significantly improve performance, since plane relaxation will be performed on relatively small domains.

Finally, it would be worthwhile to compare the performance of GPU-accelerated BoxMG, with patch-based refinement, against a GPU-accelerated implementation of AMG. It has already been shown in [25] that BoxMG is significantly faster than AMG on problems with structured grids, mainly due to the costly setup phase and indirect addressing issues of AMG. Because of the inherently sequential nature of coarse-grid selection, and the cost of memory indirection on the GPU, it does not appear that AMG would naturally lend itself to GPU acceleration. For a general problem, AMG would have minimal degress of freedom, whereas patch-based refinement would be necessary for BoxMG to achieve comparable performance, thus requiring many more degrees of freedom. It would then be interesting to compare the performance of a GPU-accelerated AMG implementation, with minimal degrees of freedom, against GPU-accelerated BoxMG, with significantly more degrees of freedom, but a potentially more efficient implementaton on the GPU architecture.

Appendix A

Kernel Timing Results

The values on the following page were used for generating the line plots in Chapter 6. All values are in seconds.

RESIDUAL DOUBLE	32X32	64X64	128X128	256X256	512X512	1024X1024
CPU	0.00006200	0.00019978	0.00098620	0.00400420	0.02764960	0.09096240
AMD	0.00043980	0.00056654	0.00069320	0.00070850	0.00257850	
ATI	0.00004850	0.00004400	0.00007725	0.00014525		
NVIDIA	0.00001540	0.00001720	0.00002375	0.00006725	0.00021260	
RESIDUAL SINGLE	32X32	64X64	128X128	256X256	512X512	1024X1024
CPU	0.00006200	0.00019978	0.00098620	0.00400420	0.02764960	0.09096240
AMD	0.00045720	0.00064100	0.00063720	0.00065750	0.00122375	0.00604125
ATI	0.00003850	0.00003900	0.00006200	0.00014775		
NVIDIA	0.00001420	0.00000950	0.00001475	0.00003600	0.00011200	0.00044780
FUSION	0.00000850	0.00001300	0.00003450	0.00012000		
RHS Y DOUBLE	32X32	64X64	128X128	256X256	512X512	1024X1024
CPU	0.00002420	0.00010600	0.00049860	0.00284980	0.03290000	0.09386440
AMD	0.00020580	0.00031566	0.00054418	0.00111640	0.00462400	
ATI	0.00003220	0.00004260	0.00009200	0.00009920		
NVIDIA	0.00001080	0.00001920	0.00005720	0.00024400	0.00087420	
RHS Y SINGLE						
CPU	0.00002420	0.00010600	0.00049860	0.00284980	0.03290000	0.09386440
AMD	0.00020360	0.00040280	0.00049438	0.00092300	0.00324320	0.02113640
ATI	0.00002700	0.00003700	0.00007380	0.00021600		
NVIDIA	0.00001120	0.00001780	0.00005780	0.00023580	0.00086680	0.00348560
FUSION	0.00000740	0.00001620	0.00004900	0.00018120		
RHS X DOUBLE	32X32	64X64	128X128	256X256	512X512	1024X1024
CPU	0.00002500	0.00009680	0.00040000	0.00161380	0.01008060	0.03667940
AMD	0.00017436	0.00039180	0.00066280	0.00066760	0.00144860	
ATI	0.00002920	0.00003220	0.00004600	0.00004880		
NVIDIA	0.00000860	0.00000860	0.00001040	0.00003120	0.00009760	
RHS X SINGLE	32X32	64X64	128X128	256X256	512X512	1024X1024
CPU	0.00002500	0.00009680	0.00040000	0.00161380	0.01008060	0.03667940
AMD	0.00020480	0.00041620	0.00052274	0.00062500	0.00078120	0.00252420
ATI	0.00002860	0.00003060	0.00003880	0.00007220		
NVIDIA	0.00000860	0.00000700	0.00000980	0.00001780	0.00005080	0.00020180
FUSION	0.00000620	0.00000800	0.00001520	0.00003760		
TRIDIAGONAL SOLVE X DOUBLE	32X32	64X64	128X128	256X256	512X512	1024X1024
CPU	0.00001520	0.00005620	0.00021740	0.00089100	0.00560880	0.01635780
AMD	0.00030340	0.00043178	0.00126460	0.00375960	0.03666140	
ATI	0.00007320	0.00008320	0.00020040	0.00034000		
NVIDIA	0.00001940	0.00001880	0.00002580	0.00008300	0.00044060	
TRIDIAGONAL SOLVE X SINGLE	32X32	64X64	128X128	256X256	512X512	1024X1024
CPU	0.00001520	0.00005620	0.00021740	0.00089100	0.00560880	0.01635780
AMD	0.00022440	0.00049880	0.00111260	0.00423320	0.03670480	0.14848060
ATI	0.00004120	0.00004400	0.00007400	0.00023120		
NVIDIA	0.00000940	0.00001000	0.00001340	0.00003660	0.00013840	0.00067180
FUSION	0.00001080	0.00001780	0.00004980	0.00011640		
TRIDIAGONAL SOLVE Y DOUBLE	32X32	64X64	128X128	256X256	512X512	1024X1024
CPU	0.00002440	0.00009580	0.00038160	0.00155960	0.01391740	0.03940560
AMD	0.00022860	0.00048720	0.00115320	0.00375300	0.03800620	
ATI	0.00007640	0.00008420	0.00020340	0.00023060		
NVIDIA	0.00002020	0.00002120	0.00003680	0.00011760	0.00060880	
TRIDIAGONAL SOLVE Y SINGLE	32X32	64X64	128X128	256X256	512X512	1024X1024
CPU	0.00002440	0.00009580	0.00038160	0.00155960	0.01391740	0.03940560
AMD	0.00022320	0.00051020	0.00107020	0.00436140	0.03777640	0.15450940
ATI	0.00004120	0.00004540	0.00008060	0.00024500		
NVIDIA	0.00001100	0.00001320	0.00002440	0.00007720	0.00027040	0.00138640
FUSION	0.00001100	0.00001840	0.00005240	0.00017680		

Bibliography

[1] *OpenCL 1.2 Specification*, 2011.

[2] David M. Alber and Luke N. Olson. Parallel coarse-grid selection. *Numerical Linear Algebra with Applications*, 14(8):611–643, 2007.

[3] R. E. Alcouffe, Achi Brandt, J. E. Dendy, Jr., and J. W. Painter. The multigrid method for the diffusion equation with strongly discontinuous coefficients. *SIAM J. Sci. Statist. Comput.*, 2(4):430–454, 1981.

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[5] Kendall E. Atkinson. *An introduction to numerical analysis*. John Wiley & Sons Inc., New York, second edition, 1989.

[6] Travis Austin, Markus Berndt, and David Moulton. A memory efficient parallel tridiagonal solver. Preprint LA-UR-03-4149, 2004.

[7] Ben Bergen. Ocl-mla, http://sourceforge.net/projects/ocl-mla/, March 2012.

[8] A. Brandt, S. McCormick, and J. Ruge. Algebraic multigrid (AMG) for sparse matrix equations. In *Sparsity and its applications (Loughborough, 1983)*, pages 257–284. Cambridge Univ. Press, Cambridge, 1985.

[9] Achi Brandt. Multi-level adaptive solutions to boundary-value problems. *Math. Comp.*, 31(138):333–390, 1977.

[10] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A multigrid tutorial (2nd ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[11] Andri R. Brodtkorb, Trond R. Hagen, and Martin L. Saetra. Graphics processing unit (gpu) programming strategies and trends in gpu computing. *Journal of Parallel and Distributed Computing*, pages –, 2012.

[12] Edmond Chow, Robert D. Falgout, Jonathan J. Hu, Raymond S. Tuminaro, and Ulrike M. Yang. A Survey of Parallelization Techniques for Multigrid Solvers. Technical report.

[13] Paul Concus, Gene H. Golub, and Dianne P. O'Leary. A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations. In *Studies in numerical analysis*, volume 24 of *MAA Stud. Math.*, pages 178–198. Math. Assoc. America, Washington, DC, 1984.

[14] Rand Corporation and W.H. Ware. *Ultimate Computer*. Rand Corporation Paper P-. 1969.

[15] J. E. Dendy, Jr. Black box multigrid. *J. Comput. Phys.*, 48(3):366–386, 1982.

[16] Jack J. Dongarra, Lain S. Duff, Danny C. Sorensen, and Henk A. Vander Vorst. *Numerical Linear Algebra for High Performance Computers*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[17] Peter Kogge et. al. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

[18] Petri Fast and Michael J. Shelley. Moore's law and the saffman-taylor instability. *Journal of Computational Physics*, 212(1):1 – 5, 2006.

[19] Alan George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973. Collection of articles dedicated to the memory of George E. Forsythe.

[20] Dominik Göddeke, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, Sven H.M. Buijssen, Matthias Grajewski, and Stefan Turek. Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. *Parallel Computing, Special issue: High-performance computing using accelerators*, 33(10–11):685–699, November 2007.

[21] Bill Goodwin. Supercomputers will reach 'exascale' speeds within decade, April 2012.

[22] Van Emden Henson and Ulrike Meier Yang. BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Appl. Numer. Math.*, 41(1):155–177, 2002. Developments and trends in iterative methods for large systems of equations—in memoriam Rüdiger Weiss (Lausanne, 2000).

[23] Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *J. Research Nat. Bur. Standards*, 49:409–436 (1953), 1952.

[24] S. P. MacLachlan, J. D. Moulton, and T. P. Chartier. Robust and adaptive multigrid methods: comparing structured and algebraic approaches. 19(2):389–413, March 2012.

[25] S. P. MacLachlan, J. M. Tang, and C. Vuik. Fast and robust solvers for pressure-correction in bubbly flow problems. *J. Comput. Phys.*, 227(23):9742–9761, 2008.

[26] Victor Minden. Improved iterative methods for napl transport through porous media, May 2012.

[27] Ethan Mollick. Establishing Moore's law. *IEEE Ann. Hist. Comput.*, 28(3):62–75, 2006.

[28] V. Osipov N. Leischner and P. Sanders. Fermi architecture white paper. Technical report, NVIDIA, 2009.

[29] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, April 2012.

[30] Peter S. Pacheco. A user's guide to mpi, 1998.

[31] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann*

Series in Computer Architecture and Design). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[32] Richard S. Varga. *Matrix iterative analysis*, volume 27 of *Springer Series in Computational Mathematics.* Springer-Verlag, Berlin, expanded edition, 2000.

[33] David S. Watkins. *Fundamentals of matrix computations.* John Wiley & Sons, Inc., New York, NY, USA, 1991.

[34] Ulrike Meier Yang. On the use of relaxation parameters in hybrid smoothers. *Numer. Linear Algebra Appl.*, 11(2-3):155–172, 2004.

[35] Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the gpu. In *PPOPP*, pages 127–136, 2010.