Spring 2019

Enhancing Portability in High Performance Computing: Designing Fast Scientific Code with Longevity

Jason Orender
Old Dominion University, jason.orender@publicmail.email

ENHANCING PORTABILITY IN HIGH PERFORMANCE COMPUTING: DESIGNING FAST SCIENTIFIC CODE WITH LONGEVITY

by

Jason Orender
B.S. December 1993, University of Texas
MBA, December 2003, George Mason University

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
May 2019

Approved by:

Mohommed Zubair (Director)

Yaohang Li (Member)

Ravi Mukkamala (Member)

ABSTRACT

ENHANCING PORTABILITY IN HIGH PERFORMANCE COMPUTING: DESIGNING FAST SCIENTIFIC CODE WITH LONGEVITY

Jason Orender
Old Dominion University, 2019
Director: Dr. Mohommed Zubair

Portability, an oftentimes sought-after goal in scientific applications, confers a number of possible advantages onto computer code. Portable code will often have greater longevity, enjoy a broader ecosystem, appeal to a wider variety of application developers, and by definition will run on more systems than its pigeonholed counterpart. These advantages come at a cost, however, and a rational approach to balancing costs and benefits requires a systemic evaluation. While the benefits for each application are likely situation-dependent, the costs in terms of resources, including but not limited to time, money, computational power, and memory requirements, are quantifiable. This document will identify strategies for enhancing performance portability on a variety of platforms available to the scientific computing community which will have little or no adverse impact on alternate architectures; this is done by implementing an iterative point solver requiring a high degree of data transfer bandwidth of a type commonly used in high performance applications used for computing a solution to partial differential equations (PDEs). In this thesis, we were able to show significant speed enhancements for architectures as diverse as complex traditional Central Processing Units (CPUs), Graphical Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs). Employing generalized optimizations on a variety of development frameworks we were able to show as much as a 92.5% reduction on a pipelined architecture (FPGA) while having a negligible impact on alternate architectures, and an 88.6% reduction in execution time on a Single Instruction Multiple Data (SIMD) architecture (GPU/CPU) while also having a negligible impact on alternate architectures. By enforcing these design rules in released versions of scientific code, the code has the potential to be optimally positioned for future advancements in computing architecture as well as being performance portable among existing architectures.

Dedicated to my wife Dani, who tolerated my work schedule and inspired my scholarship.

ACKNOWLEDGEMENTS

- Dr. Mohammed Zubair (Old Dominion University)
- Dr. Eric Nielsen (NASA)
- Mike Cardoso (Intel)

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES ix

Chapter

1. INTRODUCTION 1

2. BACKGROUND 3
   - 2.1 PORTABLE STANDARDS 3
   - 2.2 SELECTING A RESPRESENTATIVE PROBLEM 5

3. RELATED WORK 12

4. PROBLEM DEFINITION 14

5. TECHNICAL SOLUTION 16
   - 5.1 CONSOLIDATION OF ARITHMETIC OPERATIONS 16
   - 5.2 IDENTIFY COMMON MEMORY ACCESSES 19
   - 5.3 ACCESS MEMORY IN LARGE CONTIGUOUS BLOCKS 21
   - 5.4 IDENTIFY OPPORTUNITIES FOR VECTORIZING 22
   - 5.5 CONSTRUCT INDEPENDENT LOOPS THAT HAVE A CONSTANT NUMBER OF ITERATIONS 22
   - 5.6 CONSTRUCT INDEPENDENT LOOPS THAT ARE HAVE A CONSTANT NUMBER OF ITERATIONS FOR A SINGLE WORK ITEM 22

6. EVALUATION OF DEVELOPED SOLUTION 29

7. MAJOR CONTRIBUTIONS 32

8. CONCLUSIONS 34

REFERENCES 36

APPENDICES
   - A. BASIC SEQUENTIAL POINT SOLVER 37
   - B. FPGA WRAPPER CODE 42
   - C. FPGA BASIC CODE 45
   - D. FPGA PARTIALLY OPTIMIZED CODE 51
   - E. FPGA OPTIMIZED CODE 57
   - F. GPU BASIC CODE 65

G. GPU PARTIALLY OPTIMIZED CODE 72
H. GPU OPTIMIZED CODE 77
I. DATA TABLES 81
J. HARDWARE SPECIFICATIONS 84

VITA 87

LIST OF TABLES

Table		Page
1.	Source Portability Strategies	3
2.	Time Comparison Summary table (times shown in ms)	30
3.	Optimization Continuum Results	31
4.	Clock Validation for Optimized Code (time in ms)	81
5.	Clock times for Optimized Code without profiler (time in ms)	82
6.	Clock Validation for Non-Optimized (Baseline) Code (time in ms)	82
7.	Clock times for Non-Optimized code without profiler (time in ms)	83
8.	OpenMP Trials for ARM CPU (times in ms)	83
9.	OpenMP Trials for Intel x86 CPU (times in ms)	83
10.	GPU Hardware Data	84
11.	FPGA Hardware Data	85
12.	ARM CPU Hardware Data	85
13.	x86 CPU Hardware Data	86

LIST OF FIGURES

Figure Page

1. A sparse matrix and its BCSR representation with $n_b = 2$. There are 22 non-zero elements in this sparse matrix that has nominally 64 elements. Assuming that the array elements are integers in this case, the storage space required for the uncompressed version is $64 \times 4 = 256$ bytes. The storage space required for the same array in BCSR format would require $(28 \times 4) + (5 \times 4) + (7 \times 4) = 160$ bytes. Many scientific applications employ large matrices that contain mostly zeroes; in these cases the space and time savings gained by iterating over a matrix in BCSR format can be significant. 8

2. A simple example of a four-colored grid with adjacency implying computational dependence. A time step calculation on any single color element can be assumed to be computationally independent of the elements of the same color. A time step calculation on a red element, for example, will be independent any other red element, implying that a single time step could be calculated for each of the colors in parallel (e.g. 16 parallel threads to calculate the red elements, then update the matrix, and then use another 16 parallel threads to calculate the green elements, and so on). 9

CHAPTER 1

INTRODUCTION

Creating portable code for scientific applications faces some unique challenges. Since scientific applications will frequently rely on highly computationally intensive algorithms, the effort to parallelize aspects of the code will often achieve highly asymmetric gains in code functionality [1]. That is, for the amount of effort expended in optimizing the code, the greatest return for this investment is often parallelization. As a result, a tradeoff emerges: the researcher can spend effort to optimize code for a specific platform and get greater computational efficiency, or they can apply their efforts to ensure that their code adheres to a portable standard that that will have wide utility and extended lifetime but at a reduced efficiency. The benefits of the first strategy can be realized very quickly, while the benefits of the latter can take much longer to materialize. In fact, if the reduction in efficiency is too severe, the enhanced utility that is the nominal goal of the second approach might be obviated altogether. This brings into focus several reasons why portability might not be a good choice [2]:

- Extended development time vs. non-portable code
- Reduction in performance vs. non-portable code
- Inability of a specific model to run on alternate hardware
- Model was previously developed for specific hardware, and its function is not well enough understood to create a generalized portable version.

The last two items in this list, while valid considerations for specific code, are not applicable in this analysis. This document will examine the problem as if the code is created from a well understood generalizeable model that does not require specific hardware to achieve a valid result.

This document will introduce the particular PDE solver being studied and explain why it is representative of the type of problem that is commonly approached in

high performance computing, as well as explain the motivation to find more efficient computational frameworks.

Choosing a portability strategy to focus on will be the first step in the analytical process. All strategies are not equivalent with respect to high performance computing and certain frameworks offer distinct advantages to the researcher. After an open standard is chosen, the performance of several platforms, including single threaded CPU, typical multicore CPU, high performance multicore CPU, multicore ARM, FPGA, and GPU will be evaluated, including a baseline version of the code as well optimized versions. A discussion of tradeoffs, as well as advantages and disadvantages of each platform will occur in this portion. An explanation of why certain platforms fail or excel at certain tasks will also be included here. A comparison metric for costs is the final piece of the puzzle that will be discussed, to include a recommendation for the best framework and platform combination for the solution to this particular type of problem.

CHAPTER 2

BACKGROUND

2.1 PORTABLE STANDARDS

It is first necessary to take a step back and discuss what is meant by the term "portability". The most restrtictive form, binary portability, refers to the ability to run a binary executable on multiple platforms without having to change or recompile the binary in any way. This is very difficult to achieve on diverse architectures and is not generally what is meant by the word "portable" in modern discourse; because of this, binary portability will not be discussed in this document. The less restrictive and more common understanding of the term is source portability; that is, the ability to have a program that is adpated at the source level which can be compiled on multiple target environments with little to no modification. There are many ways to achieve this goal, and these generally conform to several well defined categories (see Table 1).

A subcategory of source portability, so-called "performance portability", could be described as code that has similar performance characteristics across multiple architectures. It could also be described as achieving the best realistically achievable performance across multiple architectures; this will be considered an implied paradigm, and while performance portability may not be explicitlty referenced it should be understood as the ultimate goal of well constructed portable code.

TABLE 1: Source Portability Strategies

Strategy	**Example**
Standardized Languages	C/C++, Fortran
Language extensions	CUDA, TBB (Intel)
Open Language Constructs	OpenMP, OpenCL
Virtual Machines	Java
Steering Languages	Python

Standardized languages offer a common strategy for portability. Via precompiler directives, the source can be compiled in myriad ways and invoke many differing types of dependencies. As an example, a C++ program can be written so that it can take advantage of posix threads on a Unix or Linux machine while reverting to a Windows API implementation when required. The negative aspect of this strategy is that much of the program must be re-written for each new platform and provided as an alternate compilation path, possibly by employing a series of precompiler directives to activate or deactivate large blocks of code, since the syntax and optimal arrangement of the code can differ markedly. While this method may work for simple programs that do not need to exploit parrallelism at scale, it may become untenable for code with a high degree of complexity or which requires a great deal of maintenance since every block of code with duplicate functionality will need to be updated seperately.

Language extensions can be thought of as an outgrowth of the standardized languages strategy. The specifics for exploiting specialized hardware are encapsulated in a library that is called when the code is compiled on a machine that can support it. This makes the resulting code simpler than lower level programming methods like individual thread manipulation via Posix threads or the Windows API; by using the Thread Building Blocks (TBB) Intel library, for instance, the code can be tremendously simplified but it will still suffer from the higher level complexity issues that make the standardized language strategy untenable. The changes required to use differing operating systems are resolved because versions of this library exist for all major operating systems that use chips by this particular manufacturer, but differences across hardware are still unresolved. A Graphics Processing Unit (GPU) will still require a version of the code that is syntactically distinct from the CPU code, and an ARM processor would require a still different version of the optimized code.

The next strategy is the utilization of open language constructs like OpenMP and OpenCL. These have the advantage of not only being supported by most major operating systems, but having gleaned support from hardware manufacturers as well. These are examples of consortium standards, and this strategy seemingly incorporates the best aspects of the previous two. There are several disadvantages to using this strategy, however, and these form the basis of defining the tradeoffs that are inherrent in reliable source portability. The first disadvantage is that hardware manufacturers will likely implement new innovations in their own language extensions first before they devote time and resources to updating an open source project;

ways to organize threads into cooperative groups that pass information back and forth on the fly, for example, offer great opportunities for optimization but are also highly hardware dependent at the time of this writing. Standard ways to accomplish optimization tasks are likely to require a consensus that can only be reached after a certain amount of trial and error has already occurred and a favorite method identified by researchers and developers is clear. As a result, there is a necessary time lag between the occurrence of a new innovation and its emergence as a widely supported portable standard. Second, the generalized (and portable) implementation of an optimizing construct will likely be less efficient than the hardware specific implementation; this is one of the metrics that I will examine to determine the price of the tradeoff.

Virtual Machines take the idea of portability to a level that is perhaps unattainable by any of the previous methods, but this comes at the high cost of abstracting the hardware away altogether and incurring significant overhead. There has been significant work as early as 2003 to create distributed virtual machines that can transparently manage multi-threaded applications over several nodes [3], but because of the added overhead required to manage these machines they will by definition never be able to achieve the level of performance available by running code directly on the hardware. For this reason, the Virtual Machine strategy will not be considered.

So called "steering languages", like Python can provide for a rapid development cycle by utilizing many highly optimized libraries [4]. Python, in particular, is highly extensible and boasts a development community that regularly provides updated libraries for general use. These languages can be considered an additional abstraction layer since many of the libraries created must be originally coded using one of the first three methods examined. Since the point of scientific computing is frequently to examine results from novel algorithms, it is this initial development that will be considered in the analysis presented by this document. The value of using preprogrammed libraries in a steering language such as Python cannot be understated in terms of development streamlining, but it is not the focus of this document.

2.2 SELECTING A RESPRESENTATIVE PROBLEM

The second task that needs to be accomplished prior to performing analysis is to pick a representative problem that encapsulates many of the issues that the scientific

community faces when attempting to code a solution. As a general rule, these problems might be divided into two broad categories: 1) computationally intensive and 2) data intensive. In a computationally intensive problem, the time spent converging to a solution and performing calculations will be the limiting factor, while a data intensive problem might rely on simple operations performed on a large amount of data. A third possibility is a problem that incorporates both of these elements and is therefore both computationally intensive and data intensive; a problem of this sort will likely be the most representative benchmark for anlysis. Any metric computed should also be able to differentiate the location of the bottleneck as either in the computational space or the data transfer space.

For this reason, the Partial Differential Equation (PDE) solver used in the Fully Unstructured 3D Grid (Fun3D) modeling software supported by the National Aeronautics and Space Administration (NASA) was selected. It is an iterative PDE solver that computes multiple sparse matrix-vector multiplications per iteration over a multi-dimensional grid. Importantly, it is a widely distributed and well understood piece of code for which large standard data sets are available and valid results are known.

2.2.1 DESCRIPTION OF THE PROBLEM BEING STUDIED

The result of the PDE solver's implicit solution approach is a set of linear equations of the form:

$$Ax = b$$

This equation must be solved frequently during the simulation in which it is used, where:

- A is an $n \times n$ spatial mesh (a matrix).
- x is an input.
- b is the result.

The $n \times n$ matrix is further broken down into sub-matrices of size $n_b \times n_b$ which is the result of linearization of nonlinear equations at each grid point. The matrix A is divided into diagonal D and off-diagonal O matrices:

$$A = D + O \tag{1}$$

The solver initializes the grid points by renumbering them with the reverse Cuthill–McKee algorithm (RCM) [5] to create a band matrix based on a permutation of the sparse matrix.

An array of size $[nnz \times n_b \times n_b]$ is used to store nnz blocks of the diagonal matrix D. For each block D_i, two triangular sub-matrices, the lower L_i and the upper U_i, are generated in-place before running each linear solver for $1 \leq i \leq n$. The L_i and the U_i matrices are then computed using a forward and back substitution algorithm. This is another useful technique used to help improve cache locality.

The off-diagonal matrix O contains nnz non-zero blocks, where each block is stored using a modified block compressed sparse row (BCSR) [6] format. In the modified BCSR format, three arrays are used: ia and ja, to efficiently capture the sparsity pattern of the matrix and a one-dimensional data array of size $[nnz \times n_b \times n_b]$, to store all of the non-zero elements. The integer array ia of size $(n+1)$ is used to keep indexes of all leading non-zero blocks in each row of O (the final entry is for the hypothetical beginning index of the next row beyond the end of the matrix - it is included so that the number of nonzero blocks in the last row of the matrix can be inferred). The ja array of size nnz stores the block-column indexes of all non-zero blocks. Figure 1 shows how a simple matrix can be represented with this block structure.

Studying the use of sparse matrices is significant with respect to scientific computing in that copying large blocks of contiguous memory is, as a general rule, much faster and more efficient than copying individual bytes or small groups of bytes. Every memory access has an overhead associated with it that is relatively independent of the size of the memory being accessed, and in this realization a tradeoff emerges. For the quickest memory access, the matrix cannot be compressed, but at a certain point the added time of accessing large numbers of zeroes outweighs the overhead required to access the non-zeroes independently. In many cases, the size of the matrices in memory is also a limiting factor. For these practical reasons, studying the effects of calculations performed on compressed matrices (for this document the BCSR format is used) will likely yield the most relevant general result.

A "multi-coloring" scheme is used in the point-implicit linear solver which exposes the parallelism in the solver computation. It groups colors and grid points such that no two neighbor points are colored the same. All unknowns associated with a grid point are assigned the color of that point.

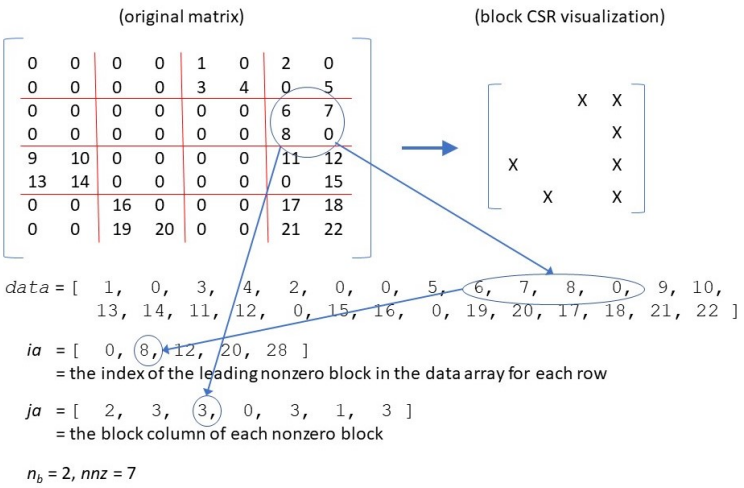

Fig. 1: A sparse matrix and its BCSR representation with $n_b = 2$. There are 22 non-zero elements in this sparse matrix that has nominally 64 elements. Assuming that the array elements are integers in this case, the storage space required for the uncompressed version is $64 \times 4 = 256$ bytes. The storage space required for the same array in BCSR format would require $(28 \times 4) + (5 \times 4) + (7 \times 4) = 160$ bytes. Many scientific applications employ large matrices that contain mostly zeroes; in these cases the space and time savings gained by iterating over a matrix in BCSR format can be significant.

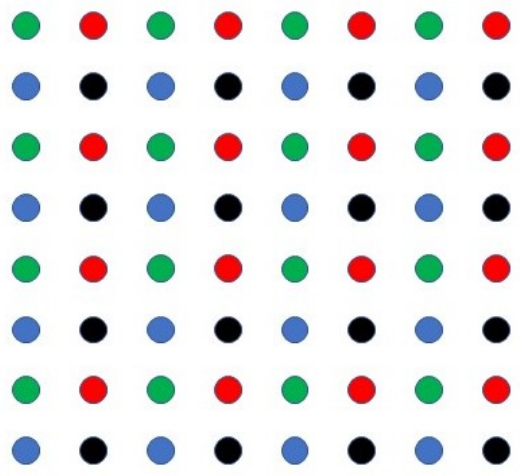

Fig. 2: A simple example of a four-colored grid with adjacency implying computational dependence. A time step calculation on any single color element can be assumed to be computationally independent of the elements of the same color. A time step calculation on a red element, for example, will be independent any other red element, implying that a single time step could be calculated for each of the colors in parallel (e.g. 16 parallel threads to calculate the red elements, then update the matrix, and then use another 16 parallel threads to calculate the green elements, and so on).

In this context a "color" is a grouping of grid points that are not expected to influence the calculations on any other grid points in a cohort if they are assigned identical colors; all grid points that are assigned "red", to extend the analogy, are expected to be computationally independent of each other, while the unknowns associated with "red" points might well have an impact on any given "green" or "black" points. An example is shown in Figure 2. This has been a common strategy to expose parallelism [7, 8] with respect to both scientific and graphics computation for some time.

In the particular case of the PDE solver for Fun3D, an approximate nearest-neighbor flux Jacobian is used to generate A, which results in no data dependencies between the unknowns of the same color; this provides the possibility of updating them in parallel fashion. The process of generating this matrix based on the raw

input data is not part of the calculation studied and will not be described here, but is covered in detail in [9]. The linear solver computation is repeated several times over the entire system, and each time the unknowns are updated with the latest values of x from the other colors. To improve memory access and consequently cache performance, the system of algebraic equations is renumbered so that the unknowns of the same color are grouped together by organizing them consecutively in memory and the arrays of ia and ja are modified to adopt the new matrix structure; this allows for some of the advantages of reduced overhead by copying large blocks of memory to acceleration hardware like a GPU, for example, at once while preserving the practical necessity of employing the sparse matrix format for storage. Once the linear solver computation is done, an inverse map is then used to update the nonlinear solution of the partial differential equations (PDEs) at each grid point.

The full code for all versions of the algorithm are included in the appendices. In the interest of clarity and brevity, a generalized version in pseudocode is presented below in order to give a general idea of where calculations and data transfers are occurring. All versions of the code follow the general format presented below.

Psuedocode of the general algorithm used follows:

```
[transfer data to device from host if required]
for i = 1 to sweeps
  for j = 1 to num_colors
    solve_subroutine(data)
[transfer data from device to host if required]
```

Pseudocode for the solve subroutine follows:

```
def solve_subroutine(data):
  // transfer data from device memory to local memory
  set f1, f2, f3, f4, f5 to residual array elements for this node.
  start = ia[node]
  end   = ia[node+1]-1

  // loop over the nonzero elements
  for i = start to end
    icol = ja[i]
    // set the new values equal to the old values multiplied by a
    // deterministic constant based on nearby nodes
    decrement f1..f5 by the product of the off-diagonal matrix
       values and the previous solution matrix values five times
       (over each column, if the f1..f5 variables are viewed as
```

```
   rows)

// solve forward
decrement f2..f5 by the product of the diagonal matrix values
   and f1
decrement f3..f5 by the product of the diagonal matrix values
   and f2
decrement f4..f5 by the product of the diagonal matrix values
   and f3
decrement f5 by the product of the diagonal matrix value
   and f4

// solve backward
decrement f1..f4 by the product of the diagonal matrix values
   and a factor of f5
decrement f1..f3 by the product of the diagonal matrix values
   and a factor of f4
decrement f1..f2 by the product of the diagonal matrix values
   and a factor of f3
decrement f1 by the product of the diagonal matrix value
   and a factor of f4

set the new solution values to f1..f5
```

CHAPTER 3

RELATED WORK

Much of the work in the area of portability has been with respect to the faithful reproduction of floating point results across various platforms as with [10], the particulars of using specific steering languages and libraries as with [4, 11], and the development of portable frameworks as with [12, 13] for use across multiple platforms.

The most directly comparable work was an investigation of the portability of applications written in OpenCL [14]. This paper studied software engineering techniques that guarantee the maximum level of so-called "performance" portability. That is to say, a program written for a GPU might utilize certain memory structures or code arrangement that would either have no bearing on the performance in multi-core CPU hardware or might actually cause worse performance in that context. That paper investigated the application of standard benchmark code on GPUs and CPUs, though most of the comparison was between differing brands of CPUs.

One principal difference between that paper and this document is the expansion of the evaluation criteria to include the requirement for significant memory bandwidth and an exploration of how that affects portability. The paper also used NASA CFD code as a benchmark; the "LU" benchmark was used, which also employs large-scale Navier-Stokes computations on a three dimensional grid, but their implementation focused solely on compute performance by presuming that the memory accesses can be optimized in one of two ways. They allowed either an array-of-structs (AoS) or a struct-of-arrays (SoA) as the tested memory configuration. In the AoS configuration, the five values associated with each grid point would be adjacent in memory, which creates conditions optimal for the best compute performance in a scalar work item. In the SoA configuration, the values would be split into five separate units, which would allow the best compute performance in a Single Instruction Multiple Data (SIMD) parallel architecture. In a real CFD dataset, the data would generally not be able to be optimized completely for either of these architectures; optimizing some of the data would require random accesses for another portion of the data as a tradeoff because of the high degree of interdependence between grid points. For

this reason, the sparse data access architecture in the Fun3D code is likely a better representation of the memory performance for real scientific applications in general, rather than the simplified uncompressed data used with the LU benchmark in [14]. The analysis of the Fun3D code performance on multiple architectures will show that no matter what compute optimizations are made, memory bandwidth still has an material importance in the performance of the code as a whole.

CHAPTER 4

PROBLEM DEFINITION

The problem can be split into two elements: 1) how can the code be constructed to take advantage of specific hardware characteristics, and 2) the implementation cost differential when comparing a portable semi-optimized version to both the un-optimized version and the fully optimized version.

Defining the specific ways that the code must be altered to take advantage of hardware acceleration leads to three possible basic versions of code that nominally accomplish the same tasks. The first version of the code consists of a simplistic sequential implimentation that is created without regard to memory or loop structures that might be more efficient on alternate architectures; construction of this version of the code is usually the first step, and while it is likely to be portable across every other architecture, it will also probably have severely suboptimal characteristics.

From this first version, a second version of the code could be derived and optimized for a Single Instruction Multiple Data (SIMD) capable platform such as a GPU or multi-core GPU; this would be the portable semi-optimized version. Depending on the specific vendor, this code could be further branched to create specific optimizations that could be made to enhance efficiency at the expense of portability; this would be the fully optimized version.

Again branching from the first version, a third version of the code could be derived and optimized for a pipeline parallel platform like a Field Programmable Gate Array (FPGA); this would also be a portable semi-optimized version. Additional modifications that enhance exposure to pipeline parallelism but cause increased execution time on alternate platforms would constitute the fully optimized version for this case. While there are ways to more fully optimize FPGA platforms that go beyond using the OpenCL standard, for instance by using Register Transfer Language (RTL) or some specific Hardware Definition Language (HDL), they require a specialized level of knowledge that make them an atypical choice as an acceleration technology for use in general scientific computing.

Portability, in this context, could potentially be achieved by creating semi-optimized versions derived from the basic naive version that work equally, or nearly equally, well

on both an SIMD platform or a pipeline parallel platform. The cost of this portable version can then be described in terms of the performance differential in terms of execution time between this version and the fully un-optimized (naive) version as well as the fully optimized version. The performance gap between fully optimized and un-optimized versions is the maximum potential benefit, while the location of the portable semi-optimized version on this continuum can be described in terms of the fraction of maximum potential benefit either gained or lost.

CHAPTER 5

TECHNICAL SOLUTION

The design of code that is portable among several architectures that optimize differently can only be partially effective due to the competing goals of these architectures, but the gains achieved by making subtle changes to the code can be significant. There are several major categories of changes that make the biggest differences:

- Consolidate arithmetic operations that make use of intermediate variables or multiple steps.
- Identify common memory accesses that could be mapped to shared (local) memory when the opportunity presents itself.
- Access global memory in large contiguous blocks instead of randomly selecting smaller sections.
- Identify opportunities for vectorizing data/operations.
- Construct independent loops that have a constant number of iterations that are knowable at compile-time.
- Alternatively, construct independent loops that have a constant number of iterations for a single work item.

Abiding by these general limitations is relatively simple if it is done while while composing the code, but it becomes progressively harder when modifying code that has been previously composed without regard to these guidelines. The degree of difficulty added is highly dependent on the specifics of each individual case. In addition, these modifications can generally be applied to all architectures while accumulating very little additional overhead.

Each of the above listed optimizations can be identified in the partially optimized GPU code (see Appendix G) and FPGA code (see Appendix D). The following sections identify examples of these optimizations and explain why they are necessary.

5.1 CONSOLIDATION OF ARITHMETIC OPERATIONS

This is likely the simplest of the optimization steps but can make significant improvements in pipeline optimized code at no, or virtually no, cost to the run time measured in alternate architectures. In many cases arithmetic statements can be spread out over several operations as the unintended result of the evolution of the code or underlying algorithm over time or simply to make the code more readable. The following code excerpt (full code is located in Appendix C, lines 118-146) is an example of un-optimized arithmetic operations:

```
f1 -= a_off[0+0*NB+(j-1)*NB*NB]*dq[0+icol*NB];
f2 -= a_off[1+0*NB+(j-1)*NB*NB]*dq[0+icol*NB];
f3 -= a_off[2+0*NB+(j-1)*NB*NB]*dq[0+icol*NB];
f4 -= a_off[3+0*NB+(j-1)*NB*NB]*dq[0+icol*NB];
f5 -= a_off[4+0*NB+(j-1)*NB*NB]*dq[0+icol*NB];
// pipeline will stall here

f1 -= a_off[0+1*NB+(j-1)*NB*NB]*dq[1+icol*NB];
f2 -= a_off[1+1*NB+(j-1)*NB*NB]*dq[1+icol*NB];
f3 -= a_off[2+1*NB+(j-1)*NB*NB]*dq[1+icol*NB];
f4 -= a_off[3+1*NB+(j-1)*NB*NB]*dq[1+icol*NB];
f5 -= a_off[4+1*NB+(j-1)*NB*NB]*dq[1+icol*NB];
// pipeline will stall here

f1 -= a_off[0+2*NB+(j-1)*NB*NB]*dq[2+icol*NB];
f2 -= a_off[1+2*NB+(j-1)*NB*NB]*dq[2+icol*NB];
f3 -= a_off[2+2*NB+(j-1)*NB*NB]*dq[2+icol*NB];
f4 -= a_off[3+2*NB+(j-1)*NB*NB]*dq[2+icol*NB];
f5 -= a_off[4+2*NB+(j-1)*NB*NB]*dq[2+icol*NB];
// pipeline will stall here

f1 -= a_off[0+3*NB+(j-1)*NB*NB]*dq[3+icol*NB];
f2 -= a_off[1+3*NB+(j-1)*NB*NB]*dq[3+icol*NB];
f3 -= a_off[2+3*NB+(j-1)*NB*NB]*dq[3+icol*NB];
f4 -= a_off[3+3*NB+(j-1)*NB*NB]*dq[3+icol*NB];
f5 -= a_off[4+3*NB+(j-1)*NB*NB]*dq[3+icol*NB];
// pipeline will stall here

f1 -= a_off[0+4*NB+(j-1)*NB*NB]*dq[4+icol*NB];
```

```
f2 -= a_off[1+4*NB+(j-1)*NB*NB]*dq[4+icol*NB];
f3 -= a_off[2+4*NB+(j-1)*NB*NB]*dq[4+icol*NB];
f4 -= a_off[3+4*NB+(j-1)*NB*NB]*dq[4+icol*NB];
f5 -= a_off[4+4*NB+(j-1)*NB*NB]*dq[4+icol*NB];
// pipeline will stall here
```

Each of the variables f1 through f5 in this case are decremented by an amount calculated from external data. The optimized version of this code excerpt is simply the consolidation of all of these operations into five single line-items (see Appendix D, lines 142-170).

```
f1a[j] = (a_off[0+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
          a_off[0+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
          a_off[0+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
          a_off[0+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
          a_off[0+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);

f2a[j] = (a_off[1+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
          a_off[1+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
          a_off[1+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
          a_off[1+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
          a_off[1+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);

f3a[j] = (a_off[2+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
          a_off[2+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
          a_off[2+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
          a_off[2+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
          a_off[2+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);

f4a[j] = (a_off[3+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
          a_off[3+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
          a_off[3+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
          a_off[3+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
          a_off[3+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);

f5a[j] = (a_off[4+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
          a_off[4+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
          a_off[4+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
          a_off[4+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
          a_off[4+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);
```

```
// pipeline will stall here
```

This simple change resulted in a 33.5% reduction in run time when compiled for an FPGA due to removing the requirement that the intermediate numbers be stored in the destination memory. This minimizes conflicts when scheduling pipelined operations and removes the need to stall after every fifth of the computation. A stall will still occur at the end of the statements, but this consolidated stall will be shorter and in the case of an FPGA specifically, the number of clock-ticks required to sum five products is not simply five times the number required to simply calculate a single product and add it to a register; it is much less. This is because during FPGA code compilation a custom processing unit instruction will be created to accomplish this task by physically configuring the hardware. This custom instruction that is produced will, in effect, take 10 arguments and produce a sum of products in the minimum number of clock-ticks. This modification will have zero cost with respect to run time on alternate architectures.

5.2 IDENTIFY COMMON MEMORY ACCESSES

Identifying common memory accesses can show improvements across a wide range of architectures since many language extensions, including OpenCL, OpenMP, and CUDA all provide infrastructure to take advantage of on-device memory (if it exists). If on-device memory does not exist for whatever reason, the code behaves as if it were written without taking advantage of the added infrastructure. This optimization category is somewhat harder to implement since it requires a degree of familiarity with the algorithm that is being implemented. The following code example from the optimized GPU code (Appendix H), and the extra time taken to read and store this data into shared memory should be regarded as overhead.

```
__shared__ real8_t
     a_diag_lu_shared[5][5][BLOCK_DIM_Y];

int const k = threadIdx.x % 5;
int const l = threadIdx.x / 5;
int n = start + blockIdx.x * blockDim.y + threadIdx.y - 1;

if (n >= end || l >= 5)
    return;
```

```
// additional unrelated code here
.
.
.
// end - unrelated code

// Collectively load a_diag_lu into shared memory
a_diag_lu_shared[k][l][threadIdx.y] = A_DIAG_LU(k, l, n);

__syncthreads();
```

This speed gain is realized is within the code that follows:

```
if (threadIdx.x < BLOCK_DIM_Y && threadIdx.y == 0 && n < end) {

    // additional unrelated code here

    // Forward...sequential access to a_diag_lu

    f2 = f2 - a_diag_lu_shared[1][0][threadIdx.x] * f1;
    f3 = f3 - a_diag_lu_shared[2][0][threadIdx.x] * f1;
    f4 = f4 - a_diag_lu_shared[3][0][threadIdx.x] * f1;
    f5 = f5 - a_diag_lu_shared[4][0][threadIdx.x] * f1;

    f3 = f3 - a_diag_lu_shared[2][1][threadIdx.x] * f2;
    f4 = f4 - a_diag_lu_shared[3][1][threadIdx.x] * f2;
    f5 = f5 - a_diag_lu_shared[4][1][threadIdx.x] * f2;

    f4 = f4 - a_diag_lu_shared[3][2][threadIdx.x] * f3;
    f5 = f5 - a_diag_lu_shared[4][2][threadIdx.x] * f3;

    f5 = ((f5 - a_diag_lu_shared[4][3][threadIdx.x] * f4)
            * a_diag_lu_shared[4][4][threadIdx.x]);

    // Backward...sequential access to a_diag_lu.

    f1 = f1 - a_diag_lu_shared[0][4][threadIdx.x] * f5;
    f2 = f2 - a_diag_lu_shared[1][4][threadIdx.x] * f5;
    f3 = f3 - a_diag_lu_shared[2][4][threadIdx.x] * f5;
    f4 = ((f4 - a_diag_lu_shared[3][4][threadIdx.x] * f5)
```

```
            * a_diag_lu_shared[3][3][threadIdx.x]);

    f1 = f1 - a_diag_lu_shared[0][3][threadIdx.x] * f4;
    f2 = f2 - a_diag_lu_shared[1][3][threadIdx.x] * f4;
    f3 = ((f3 - a_diag_lu_shared[2][3][threadIdx.x] * f4)
            * a_diag_lu_shared[2][2][threadIdx.x]);

    f1 = f1 - a_diag_lu_shared[0][2][threadIdx.x] * f3;
    f2 = ((f2 - a_diag_lu_shared[1][2][threadIdx.x] * f3)
            * a_diag_lu_shared[1][1][threadIdx.x]);

    f1 = ((f1 - a_diag_lu_shared[0][1][threadIdx.x] * f2)
            * a_diag_lu_shared[0][0][threadIdx.x]);

    // additional unrelated code here

}
```

This additional code at the beginning will take a small amount of extra time, but the speed gains that are realized after repeated accesses to the same set of elements far exceed the little time spent at the beginning reading and storing the data. Elements can be retrieved from shared (on-chip) memory approximately one-hundred times faster than uncached global memory. There are several requirements that limit the gains that can be had using this optimization:

- The absolute size of the common elements must be small (16-64 kB - depending on the platform).
- There must be enough repeated accesses of these elements to make the additional overhead at the beginning of the code advantageous.
- The number of threads that can access common shared memory has a hard limit in the case of GPUs (32 for all major brands - termed a "warp"), and there is a somewhat more flexible limit in the case of FPGAs.

Almost all of the speed improvement from the non-optimized (basic) version of the GPU code (Appendix F) is due to this enhancement. This optimization resulted in an 85% reduction in run time versus the basic version of the GPU code.

5.3 ACCESS MEMORY IN LARGE CONTIGUOUS BLOCKS

This optimization is largely done by organizing the data prior to running the code. Knowing what order the code segments are likely to read the data will allow the data to be organized such that global memory calls start at low addresses and then sequentially progress in a predictable fashion; this may not be possible in every case. The differences in types of memory accesses in GPUs are covered extensively in ref [14]. Enhancing reading the data in this manner can likely be accomplished independent of code organization, so a quantitative treatment of the timing advantages will not be covered here.

5.4 IDENTIFY OPPORTUNITIES FOR VECTORIZING

Explicit vectorization can offer some advantages similar to accessing global memory in larger blocks, as in section 5.3. A vector of four integers that can be read at once, for instance, will take far less time to read than four independent reads of a single integer. In addition, if subsequent operations on each of the integers in that vector are identical, the operations can be conducted on the vector as a whole instead of the individual integer components. This will explicitly invoke SIMD compiler optimizations. While these data constructs offer some additional possibilities for speed-up, most modern compilers will be able to do these optimizations implicitly. For that reason, a quantitative treatment of vectorization will not be covered here.

5.5 CONSTRUCT INDEPENDENT LOOPS THAT HAVE A CONSTANT NUMBER OF ITERATIONS

Making the number of iterations predictable at compile time can have significant implications that can impact both SIMD and pipeline parallel structures. Included in this is the requirement that each loop iteration be independent of previous iterations. For GPU architectures, this allows the ability to independently schedule loop iterations to arbitrary threads and in some cases unroll loops. For pipelined code, this can potentially allow a new loop iteration to start at each new clock tick. These restrictions, however, are very difficult to meet and none of the outer loops in the evaluated code could meet this standard.

5.6 CONSTRUCT INDEPENDENT LOOPS THAT ARE HAVE A CONSTANT NUMBER OF ITERATIONS FOR A SINGLE WORK ITEM

If a constant number of iterations across all work items cannot be accomplished, it is still possible to achieve a significant level of speed-up by making the number of iterations constant across a single work item. As with section 5.5, the greatest speed-up will be observed in pipeline parallel architectures. In the FPGA tested code (Appendix E) this was done by iterating through the outer loops once in advance and noting the largest number of variable loops.

```
for (int sweep=0; sweep < n_sweeps; ++sweep) {
  int cmax=0;
  int nmax = 0;
  for (int i=sweep_start; i!=sweep_end; i+=sweep_stride) {
    if (((color_boundary_end[i]-1) - color_indices[2*i]) > cmax)
      cmax = ((color_boundary_end[i]-1) - color_indices[2*i]);
    if ((color_indices[2*i+1] - color_indices[2*i]) > cmax)
      cmax = (color_indices[2*i+1] - color_indices[2*i]);
    if ((color_indices[2*i+1] - (color_boundary_end[i]+1)) > cmax)
      cmax = (color_indices[2*i+1] - (color_boundary_end[i]+1));
  } //end for (i)

  for (int j=sweep_start; j!=sweep_end; j+=sweep_stride) {
    for (int ipass=1; ipass<=2; ++ipass) {
      int start, end;
      if (j > colored_sweeps) {
        start = 1;
        end   = 0;
      } // end if
      else {
        switch(ipass) {
        case 1:
          if (color_boundary_end[j] == 0) {
            start = 1;
            end   = 0;
          } // end if
          else {
            start = color_indices[2*j];
            end   = color_boundary_end[j] - 1;
          } // end if
          break;
        case 2:
          if (color_boundary_end[j] == 0) {
```

```
        start = color_indices[2*j];
        end   = color_indices[2*j+1];
      } // end if
      else {
        start = color_boundary_end[j] + 1;
        end   = color_indices[2*j+1];
      } // end if
      break;
    } // end switch
  } // end else

  for (int i=0; i<=cmax; i++) {
    if ((i+start) <= end) {
      if (((iam[i+start] - 1) - iam[i+start-1]) > nmax)
        nmax = (iam[i+start] - 1) - iam[i+start-1];
    } // end if (i+start)
  } // end for (i)
 } // end for (ipass)
} // end for (j)
```

Every subsequent loop was then iterated at these maximum values. For loops in which no calculation would have occurred, no operations are executed. This is accomplished by a simple if-statement that encapsulates the interior of the loop and acts as a gatekeeper to ensure that only loops that would result in a valid calculation are performed.

```
#pragma ivdep
for (int i=0; i <= cmax; i++) {
  int   irow, icol; // declaring these up here outside of if
     -blocks
  float f1_temp, f2_temp, f3_temp, f4_temp, f5_temp;

  n = i + start;

  // this if-statement acts as the gatekeeper to ensure that
      no loops
  // are executed on nonsense values

  if (n <= end) {
```

```
if (solve_backwards > 0) {
  f1 = -res[0 + (n-1)*NB];
  f2 = -res[1 + (n-1)*NB];
  f3 = -res[2 + (n-1)*NB];
  f4 = -res[3 + (n-1)*NB];
  f5 = -res[4 + (n-1)*NB];
} // end if (sweep_stride);
else {
  f1 =  res[0 + (n-1)*NB];
  f2 =  res[1 + (n-1)*NB];
  f3 =  res[2 + (n-1)*NB];
  f4 =  res[3 + (n-1)*NB];
  f5 =  res[4 + (n-1)*NB];
} // end else (sweep_stride)

istart = iam[n - 1];
iend   = iam[n] - 1;

#pragma ivdep
for (int j = 0; j <= nmax; j++) {
  irow = j + istart;
  icol = jam[irow-1] - 1;

  f1_temp =(a_off[0+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
            a_off[0+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
            a_off[0+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
            a_off[0+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
            a_off[0+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);

  f2_temp =(a_off[1+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
            a_off[1+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
            a_off[1+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
            a_off[1+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
            a_off[1+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);

  f3_temp =(a_off[2+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
            a_off[2+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
            a_off[2+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
            a_off[2+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
            a_off[2+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);
```

```
    f4_temp =(a_off[3+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
              a_off[3+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
              a_off[3+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
              a_off[3+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
              a_off[3+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);

    f5_temp =(a_off[4+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
              a_off[4+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
              a_off[4+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
              a_off[4+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
              a_off[4+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);

    if ((j+istart) <= iend) {
      f1 -= f1_temp;
      f2 -= f2_temp;
      f3 -= f3_temp;
      f4 -= f4_temp;
      f5 -= f5_temp;
    } // end if (j+istart)
    else {
      f1 -= 0;
      f2 -= 0;
      f3 -= 0;
      f4 -= 0;
      f5 -= 0;
    } // end else (j+istart)

  } // end for loop (j)

  f2 -= a_diag_lu[1 + 0*NB + (n-1)*NB*NB] * f1;
  f3 -= a_diag_lu[2 + 0*NB + (n-1)*NB*NB] * f1;
  f4 -= a_diag_lu[3 + 0*NB + (n-1)*NB*NB] * f1;
  f5 -= a_diag_lu[4 + 0*NB + (n-1)*NB*NB] * f1;

  f3 -= a_diag_lu[2 + 1*NB + (n-1)*NB*NB] * f2;
  f4 -= a_diag_lu[3 + 1*NB + (n-1)*NB*NB] * f2;
  f5 -= a_diag_lu[4 + 1*NB + (n-1)*NB*NB] * f2;

  f4 -= a_diag_lu[3 + 2*NB + (n-1)*NB*NB] * f3;
  f5 -= (a_diag_lu[4 + 2*NB + (n-1)*NB*NB] * f3)
      + (a_diag_lu[4 + 3*NB + (n-1)*NB*NB] * f4);
```

```
            f5 *= a_diag_lu[4 + 4*NB + (n-1)*NB*NB];

            // Backward...sequential access to a_diag_lu.
            f1 -= a_diag_lu[0 + 4*NB + (n-1)*NB*NB] * f5;
            f2 -= a_diag_lu[1 + 4*NB + (n-1)*NB*NB] * f5;
            f3 -= a_diag_lu[2 + 4*NB + (n-1)*NB*NB] * f5;
            f4 -= a_diag_lu[3 + 4*NB + (n-1)*NB*NB] * f5;
            f4 *= a_diag_lu[3 + 3*NB + (n-1)*NB*NB];

            f1 -= a_diag_lu[0 + 3*NB + (n-1)*NB*NB] * f4;
            f2 -= a_diag_lu[1 + 3*NB + (n-1)*NB*NB] * f4;
            f3 -= a_diag_lu[2 + 3*NB + (n-1)*NB*NB] * f4;
            f3 *= a_diag_lu[2 + 2*NB + (n-1)*NB*NB];

            f1 -= a_diag_lu[0 + 2*NB + (n-1)*NB*NB] * f3;
            f2 -= a_diag_lu[1 + 2*NB + (n-1)*NB*NB] * f3;
            f2 *= a_diag_lu[1 + 1*NB + (n-1)*NB*NB];

            f1 -= a_diag_lu[0 + 1*NB + (n-1)*NB*NB] * f2;
            f1 *= a_diag_lu[0 + 0*NB + (n-1)*NB*NB];

            dq[4 + (n-1)*NB] = f5;
            dq[3 + (n-1)*NB] = f4;
            dq[2 + (n-1)*NB] = f3;
            dq[1 + (n-1)*NB] = f2;
            dq[0 + (n-1)*NB] = f1;

          } // end if (n) - the gatekeeper if-statement
        } // end for loop (i)
      } // end for loop (ipass)
    } // end for loop (color)
  } // end for loop (sweep)
```

As a result of this code modification, some overhead calculation time is accumulated and more loop iterations occur, but each iteration can reliably start on a new clock-tick and proceed for a predictable number iterations. If this had not occurred, multiple interior loops of varying length would prevent pipelining the outer loops. This modification resulted in an 88.7% reduction in run time from the version with the simple statement consolidation optimization, and a 92.5% reduction in run time

from the un-optimized version.

CHAPTER 6

EVALUATION OF DEVELOPED SOLUTION

The same data set was used for all runs and consists of one million grid points in order to provide a challenging computation.

Optimizing with CUDA provided a unique way to validate the timing results obtained for the comparison to the OpenCL code executing on the same platform. Nvidia provides a profiling tool ("nvprof") that calculates the cumulative time spent executing each particular kernel, but requires software hooks inserted by the CUDA compilation tools in order to work. As a result, GPU code executed under an OpenCL framework could not be timed using this utility even when running on Nvidia hardware. To validate the times calculated using a monotonic clock executed from the CPU, the results from the Nvidia profiling utility were directly compared with monotonic clock times for the same runs. To summarize, the following general procedure was used:

1. Initiate code execution with the profiler.
2. Start time recorded in CPU code.
3. CUDA version of the code was executed on the GPU.
4. End time recorded in CPU code.
5. Start time recorded in CPU code.
6. OpenCL version of the code was executed on the GPU
   (same physical hardware as #3).
7. End time recorded in CPU code.
8. Profiling results were compared to monotonic clock differential for CUDA.

As would be expected on a shared resource (see data Table 4 in Appendix I), there was some variability in run times, but Nvidia profiler results compared favorably with the results gleaned from tabulating the run times using the CPU clock. The

TABLE 2: Time Comparison Summary table (times shown in ms)

Code Version	**CUDA nvprof**	**CUDA mclock**	**diff**	**OpenCL mclock**
Optimized	121.09	125.65	4.56	126.38
Opt/No Profiler	NA	124.67	NA	126.78
Non-Optimized	806.35	812.49	6.14	817.40
Non-Opt/No profiler	NA	812.41	NA	818.25

difference was constant to within a 0.33 ms maximum variability and represents the aggregate overhead required to actually invoke the kernel from the CPU.

Since there might also be some small amount of overhead involved in profiling, an additional set of runs was conducted in which the profiler was not used (shown as "No Profiler" runs in the summary Table 2). This data gathering procedure was the same with the exception of starting the profiler and reviewing its results:

1. Initiate code execution.
2. Start time recorded in CPU code.
3. CUDA version of the code was executed on the GPU.
4. End time recorded in CPU code.
5. Start time recorded in CPU code.
6. OpenCL version of the code was executed on the GPU.
7. End time recorded in CPU code.

These results do indicate a slight advantage when running native CUDA code over OpenCL code on the same device, though the difference is a minimal 0.6% speedup. This speedup was constant between the optimized and non-optimized versions of the code.

Note that when the FPGA performance number is scaled by the memory bandwidth measured performance (292 GB/s vs. 3.69 GB/s) and the floating point performance (15.7 TFLOPS vs. 1.5 TFLOPS), the 11.83 second result is scaled to 143 ms, which compares favorably with the 125.65 ms number measured on the GPU; this implies that the greater proportion of the performance differential can be traced

TABLE 3: Optimization Continuum Results

Platform	**Not Optimized**	**semi-Optimized**	**Optimized**
CPU (x86) - 16 cores	3768.25 ms	1121.4 ms	986.19 ms
CPU (ARM) - 16 cores	9903.13 ms	1349.7 ms	1340.03 ms
GPU (Nvidia)	812.49 ms	126.38 ms	125.65 ms
FPGA (PAC-10)	2.63 min	1.75 min	11.83 sec

back to these two performance statistics. Sufficient improvements in these performance metrics with respect to FPGAs could give an indication of when might be a prudent time to investigate optimizing a specific piece of code for pipeline parallelism with the ultimate goal of FPGA deployment.

CHAPTER 7

MAJOR CONTRIBUTIONS

The major contributions of this document lie in two areas:

- Enumeration of specific code modifications to employ in scientific code that will allow the streamlined optimization on multiple platforms and architectures.
- Identification of the costs associated with those modifications so that their worth can be evaluated.

In Chapter 5, I introduced several methods that will likely result in code that executes faster on specific architectures:

- Consolidation of arithmetic operations that make use of intermediate variables or multiple steps.
  - Makes code on pipelined architectures faster (33.5% reduction in run-time in this case study).
  - Has a negligible (but probably positive) effect on other architectures.
- Identification of common memory accesses that could be mapped to shared (local) memory when the opportunity presents itself
  - Makes code that uses an acceleration platform faster, since these acceleration platforms typically have a small amount of low latency on-chip memory (85% reduction in run-time in this case study).
  - Requires some code redesign that may not apply to all architectures, and frequently will require some trial and error as well a significant development time investment to implement.
  - Has no effect on architectures that do not have this capability if a single framework (like OpenCL) is used across all architectures. If multiple frameworks are used to achieve this instead, additional complexity would be required in the form of multiple blocks of code that accomplish the same tasks for alternate frameworks activated and deactivated by pragma if/then/else blocks.

- Access global memory in large contiguous blocks instead of randomly selecting smaller sections.
  - Mostly accomplished by preprocessing the input data with knowledge about the order in which the code will access the data.
  - Can be done without affecting the code at all in many cases.
- Identify opportunities for vectorizing data/operations.
  - Offers many of the same advantages discussed when with respect to access of global memory in large contiguous blocks.
  - Can be accomplished by the compiler in large part without explicitly vectorizing the code.
  - Explicit vectorization can provide a small speed-up effect, but is unlikely to make a large difference above the implicit vectorization provided by the compiler.
- Construct independent loops that have a constant number of iterations that are knowable at compile-time.
  - Can make a big difference in pipeline parallel code, as well as a significant (but smaller) difference in SIMD parallel code.
  - A difficult standard to meet in code that requires converging on an answer after a specific criteria is met or in cases where some parts of the calculation have more nonzero data input than other parts.
- Alternatively, construct independent loops that have a constant number of iterations for a single work item.
  - Can make a big difference in pipeline parallel code (92.5% reduction in runtime in this case study), but due to the overhead required to determine the right number of iterations, the difference in SIMD parallel code is unpredictable.
  - Doing this removes some of the advantage of using sparse data sets since this will require iterating over zeroes. These loops will essentially be no-operation (NOOP) iterations.

CHAPTER 8

CONCLUSIONS

In summary, there are three major areas that are worth optimizing and that will have beneficial effects across multiple architectures and increase the portability of scientific code:

- Consolidation of arithmetic operations (33.5% reduction in run-time in this case study).
- Identification of common memory accesses that could be mapped to shared (local) memory for optimizing over a single work item (85% reduction in run-time in this case study).
- Construction of independent loops that have a constant number of iterations over a single work item (92.5% reduction in run-time in this case study).

These code constructs, either used in the initial implementation of new code or introduced as a reworking of existing code can offer scaling and speed benefits over an extended period of time and likely will expose more parallelism across mutliple architectures as acceleration platforms mature and incorporate functional elements from competing architectures.

In the case of these modifications, implementation on architectures in which there is no immediate benefit will cause neglible or no detriment, and will position the code for re-use in differing architectures as opportunity allows.

REFERENCES

[1] "The exascale effect: Benefits of supercomputing investment for u.s. industry." Council on Competetiveness and Intersect360 Research, Sept 2014.

[2] J. D. Mooney, "Developing portable software." International Federation for Information Processing Digital Library; Information Technology, 2004.

[3] W. Zhu, C.-L. Wang, and F. C. M. Lau, "Jessica2: A distributed java virtual machine with transparent thread migration support." IEEE International Conference on Cluster Computing, 2002.

[4] T. E. Oliphant, "Python for scientific computing," *Computing in Science and Engineering*, vol. 9(3), 2007.

[5] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," *Proceedings of the 1969 24th National Conference, ACM '69*, p. 157–172, 1969.

[6] Y. Saad, "Iterative methods for sparse linear systems," *Society for Industrial and Applied Mathematics*, 2003.

[7] L. Chen, I. Fujishiro, and K. Nakajima, "Parallel performance optimization of large scale unstructured data visualization for earth simulator," *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pp. 133–140, 2002.

[8] C. Garcia, M. Prieto, J. Setoain, and F. Tirado, "Enhancing the performance of multigrid smoothers in simultaneous multithreading architectures," *International Conference on High Performance Computing for Computational Science*, pp. 439–451, June 2006.

[9] R. Biedron, J. Carlson, J. Derlaga, P. Gnoffo, D. Hammond, W. Jones, and B. Kleb, "Fun3d manual." NASA, 2018.

[10] Y. Gu, T. Wahl, M. Bayati, and M. Leeser, "Behavioral non-portability in scientific numeric computing," *European conference on Parallel Processing*, pp. 558–569, August 2015.

[11] F. Perez, B. E. Granger, and J. D. Hunter, "Python: an ecosystem for scientific computing," *Computing in Science and Engineering*, vol. 13(2), pp. 13–21, 2011.

[12] S. Z. Guyer and C. Lin. Springer, Boston, MA, 2001.

[13] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas, "Towards efficiency and portability: Programming with the bsp model," *Proceedings of the eighth annual ACM symposium on Parallel alogorithms and architectures*, pp. 1–12, June 1996.

[14] S. J. Pennycook, S. D. Hammond, S. A. Wright, J. A. Herdman, I. Miller, and S. A. Jarvis, "An investigation of the performance portability of opencl," *Journal of Parallel and Distributed Computing*, vol. 73(11), pp. 1439–1450, 2013.

APPENDIX A

BASIC SEQUENTIAL POINT SOLVER

The following is the basic sequential code written in C that was the basis for all of the other code adaptations. Note the structure of the loops; there is an outer iterative sweep, a color sweep (every "color" is verified to designate nodes that can be computed independently of any other node of the same "color", see the "Background" section for a more complete explanation), and then the inner loops that are suitable for fully parallel computation. Additional commenting within the code indicates which portion of the code was used in the GPU benchmarking process as the "Non-Optimized" code.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define N_SWEEPS    1
5  #define BLOCK_DIM_X BLOCK_DIM_X_PS5
6  #define BLOCK_DIM_Y BLOCK_DIM_Y_PS5
7
8  extern "C" {
9    void point_solve_5_cl
10   (intptr_t *ocl_params, intptr_t *ocl_data, int colored_sweeps,
11    int *color_indices, int neq0, int nja, int *iam, int *jam,
12    int solve_backwards, int nb, int n_sweeps, double *res, float *dq
         ,
13    double *a_diag_lu, float *a_off, int *color_boundary_end) {
14
15     int    j,n,sweep,icol,istart,iend,start,end,ipass, color,
16            sweep_start, sweep_end, sweep_stride;
17     float  f1,f2,f3,f4,f5;
18     double f[5];
19     sweep_start  = 1;
20     sweep_end    = colored_sweeps;
21     sweep_stride = 1;
22
```

```
23      if ((solve_backwards > 1) || (solve_backwards < -1)) {
24        sweep_start  = colored_sweeps;
25        sweep_end    = 1;
26        sweep_stride = -1;
27      } // end if (solve_backwards)
28
29      if (neq0 <= 0) {
30        sweep_start  = 1;
31        sweep_end    = 2;
32        sweep_stride = -1;
33      } // end if (neq0)
34
35      for (sweep = 1; sweep <=n_sweeps; sweep++) {
36        for (color = sweep_start; color <= sweep_end;
37             color += sweep_stride) {
38          for (ipass = 1; ipass <= 2; ipass++) {
39
40            if (color > colored_sweeps) {
41              start = 1;
42              end   = 0;
43            } // end if (color)
44            else {
45              switch (ipass) {
46              case 1:
47                if (color_boundary_end[color-1] == 0) {
48                  start = 1;
49                  end   = 0;
50                } // end if (color_boundary_end)
51                else {
52                  start = color_indices[2*(color-1)];
53                  end   = color_boundary_end[color-1];
54                } // end else (color_boundary_end)
55                break;
56              case 2:
57                if (color_boundary_end[color-1] == 0) {
58                  start = color_indices[2*(color-1)];
59                  end   = color_indices[2*(color-1)+1];
60                } // end if (color_boundary_end)
61                else {
62                  start = color_boundary_end[color-1] + 1;
63                  end   = color_indices[2*(color-1)+1];
```

```
64              } // end else (color_boundary_end)
65            } // end switch (ipass)
66          } // end else (color)
67
68          for (n = start; n <= end; n++) {
69          // ******************************************************
70          // * CODE WITHIN THIS LOOP IS USED IN SIMD (GPU) KERNELS *
71          // ******************************************************
72            if (solve_backwards > 0) {
73              f1 = -res[0 + (n-1)*nb];
74              f2 = -res[1 + (n-1)*nb];
75              f3 = -res[2 + (n-1)*nb];
76              f4 = -res[3 + (n-1)*nb];
77              f5 = -res[4 + (n-1)*nb];
78            } // end if (solve_backwards);
79            else {
80              f1 =  res[0 + (n-1)*nb];
81              f2 =  res[1 + (n-1)*nb];
82              f3 =  res[2 + (n-1)*nb];
83              f4 =  res[3 + (n-1)*nb];
84              f5 =  res[4 + (n-1)*nb];
85            } // end else (solve_backwards)
86
87            istart = iam[n-1];
88            iend   = iam[n] - 1;
89
90            for (j = istart; j <= iend; j++) {
91              icol = jam[j-1] - 1;
92
93              f1 -= a_off[0 + 0*nb + (j-1)*nb*nb] * dq[0 + icol*nb];
94              f2 -= a_off[1 + 0*nb + (j-1)*nb*nb] * dq[0 + icol*nb];
95              f3 -= a_off[2 + 0*nb + (j-1)*nb*nb] * dq[0 + icol*nb];
96              f4 -= a_off[3 + 0*nb + (j-1)*nb*nb] * dq[0 + icol*nb];
97              f5 -= a_off[4 + 0*nb + (j-1)*nb*nb] * dq[0 + icol*nb];
98
99              f1 -= a_off[0 + 1*nb + (j-1)*nb*nb] * dq[1 + icol*nb];
100             f2 -= a_off[1 + 1*nb + (j-1)*nb*nb] * dq[1 + icol*nb];
101             f3 -= a_off[2 + 1*nb + (j-1)*nb*nb] * dq[1 + icol*nb];
102             f4 -= a_off[3 + 1*nb + (j-1)*nb*nb] * dq[1 + icol*nb];
103             f5 -= a_off[4 + 1*nb + (j-1)*nb*nb] * dq[1 + icol*nb];
104
```

```
105               f1 -= a_off[0 + 2*nb + (j-1)*nb*nb] * dq[2 + icol*nb];
106               f2 -= a_off[1 + 2*nb + (j-1)*nb*nb] * dq[2 + icol*nb];
107               f3 -= a_off[2 + 2*nb + (j-1)*nb*nb] * dq[2 + icol*nb];
108               f4 -= a_off[3 + 2*nb + (j-1)*nb*nb] * dq[2 + icol*nb];
109               f5 -= a_off[4 + 2*nb + (j-1)*nb*nb] * dq[2 + icol*nb];
110
111               f1 -= a_off[0 + 3*nb + (j-1)*nb*nb] * dq[3 + icol*nb];
112               f2 -= a_off[1 + 3*nb + (j-1)*nb*nb] * dq[3 + icol*nb];
113               f3 -= a_off[2 + 3*nb + (j-1)*nb*nb] * dq[3 + icol*nb];
114               f4 -= a_off[3 + 3*nb + (j-1)*nb*nb] * dq[3 + icol*nb];
115               f5 -= a_off[4 + 3*nb + (j-1)*nb*nb] * dq[3 + icol*nb];
116
117               f1 -= a_off[0 + 4*nb + (j-1)*nb*nb] * dq[4 + icol*nb];
118               f2 -= a_off[1 + 4*nb + (j-1)*nb*nb] * dq[4 + icol*nb];
119               f3 -= a_off[2 + 4*nb + (j-1)*nb*nb] * dq[4 + icol*nb];
120               f4 -= a_off[3 + 4*nb + (j-1)*nb*nb] * dq[4 + icol*nb];
121               f5 -= a_off[4 + 4*nb + (j-1)*nb*nb] * dq[4 + icol*nb];
122
123             } // end for (j)
124
125             f[0] = f1;
126             f[1] = f2;
127             f[2] = f3;
128             f[3] = f4;
129             f[4] = f5;
130
131             // Forward...sequential access to a_diag_lu.
132             f[1] -= a_diag_lu[1 + 0*nb + (n-1)*nb*nb] * f[0];
133             f[2] -= a_diag_lu[2 + 0*nb + (n-1)*nb*nb] * f[0];
134             f[3] -= a_diag_lu[3 + 0*nb + (n-1)*nb*nb] * f[0];
135             f[4] -= a_diag_lu[4 + 0*nb + (n-1)*nb*nb] * f[0];
136
137             f[2] -= a_diag_lu[2 + 1*nb + (n-1)*nb*nb] * f[1];
138             f[3] -= a_diag_lu[3 + 1*nb + (n-1)*nb*nb] * f[1];
139             f[4] -= a_diag_lu[4 + 1*nb + (n-1)*nb*nb] * f[1];
140
141             f[3] -= a_diag_lu[3 + 2*nb + (n-1)*nb*nb] * f[2];
142             f[4] -= a_diag_lu[4 + 2*nb + (n-1)*nb*nb] * f[2];
143
144             f[4] -= a_diag_lu[4 + 3*nb + (n-1)*nb*nb] * f[3];
145             f[4] *= a_diag_lu[4 + 4*nb + (n-1)*nb*nb];
```

```
146
147             // Backward...sequential access to a_diag_lu.
148             f[0] -= a_diag_lu[0 + 4*nb + (n-1)*nb*nb] * f[4];
149             f[1] -= a_diag_lu[1 + 4*nb + (n-1)*nb*nb] * f[4];
150             f[2] -= a_diag_lu[2 + 4*nb + (n-1)*nb*nb] * f[4];
151             f[3] -= a_diag_lu[3 + 4*nb + (n-1)*nb*nb] * f[4];
152             f[3] *= a_diag_lu[3 + 3*nb + (n-1)*nb*nb];
153
154             f[0] -= a_diag_lu[0 + 3*nb + (n-1)*nb*nb] * f[3];
155             f[1] -= a_diag_lu[1 + 3*nb + (n-1)*nb*nb] * f[3];
156             f[2] -= a_diag_lu[2 + 3*nb + (n-1)*nb*nb] * f[3];
157             f[2] *= a_diag_lu[2 + 2*nb + (n-1)*nb*nb];
158
159             f[0] -= a_diag_lu[0 + 2*nb + (n-1)*nb*nb] * f[2];
160             f[1] -= a_diag_lu[1 + 2*nb + (n-1)*nb*nb] * f[2];
161             f[1] *= a_diag_lu[1 + 1*nb + (n-1)*nb*nb];
162
163             f[0] -= a_diag_lu[0 + 1*nb + (n-1)*nb*nb] * f[1];
164             f[0] *= a_diag_lu[0 + 0*nb + (n-1)*nb*nb];
165
166             dq[4 + (n-1)*nb] = f[4];
167             dq[3 + (n-1)*nb] = f[3];
168             dq[2 + (n-1)*nb] = f[2];
169             dq[1 + (n-1)*nb] = f[1];
170             dq[0 + (n-1)*nb] = f[0];
171
172           // ***************************************************
173           // ***************************************************
174           // ***************************************************
175           } // end for loop (n)
176
177         } // end for loop (ipass)
178
179       } // end for loop (color)
180
181     } // end for loop ( sweep)
182
183   } // end point_solve_5()
184
185 } // end extern "C"
```

APPENDIX B

FPGA WRAPPER CODE

The following is an excerpt from the main FORTAN code that invokes the OpenCL code:

```
1   write (*,*) '␣␣Starting␣OpenCL␣point_solve_5...'
2
3   ! calling once just to ensure that it is loaded on the device
4   ! before timing
5   call point_solve_5_cl(c_loc(ocl_params), c_loc(ocl_data), 1)
6
7   call fpga_setvar_f(c_loc(ocl_params), c_loc(ocl_data), D_DQ, &
8                      c_loc(dq_data01))
9   call fpga_setvar_d(c_loc(ocl_params), c_loc(ocl_data), D_RES, &
10                     c_loc(res))
11  call fpga_setvar_d(c_loc(ocl_params), c_loc(ocl_data), D_ADIAG, &
12                     c_loc(a_diag_lu))
13  write(*,*) '␣␣␣␣Setting␣of␣variables␣complete.'
14  call fpga_setvar_f(c_loc(ocl_params), c_loc(ocl_data), D_AOFF, &
15                     c_loc(a_off))
16
17  call fpga_init_events(c_loc(ocl_params))
18  call cpu_time(start_time)
19
20  call point_solve_5_cl(c_loc(ocl_params), c_loc(ocl_data), &
21                        n_meanflow_iters)
22  call fpga_wait_event(c_loc(ocl_params),PARAM_EVENT1)
23  call cpu_time(finish_time)
24  ps5_dt_ocl = (finish_time - start_time)*1E3 - to
25
26  write (*,*) '␣␣␣␣Copying␣data...'
27  call cpu_time(start_time)
28  call fpga_getvar_f(c_loc(ocl_params), c_loc(ocl_data), D_DQ, &
29                     c_loc(dq_ocl))
30  call cpu_time(finish_time)
31  write (*,'(A,F12.3,A)') '␣␣␣␣␣␣Time␣to␣retrieve␣dq:␣', &
32                          ((finish_time-start_time)*1E3-to),'␣ms'
33  write (*,*) '␣␣␣␣Done.'
```

The following is the wrapper for the OpenCL code that is invoked in the main FORTRAN code:

```
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>

#include "ocl_defs.h"
#include "ocl_helpers.h"

#define N_SWEEPS 1
#define BLOCK_DIM_X BLOCK_DIM_X_PS5
#define BLOCK_DIM_Y BLOCK_DIM_Y_PS5
#define DIV         ((int)10)

extern "C" {

  void point_solve_5_cl(intptr_t *ocl_params, intptr_t *ocl_data,
                        int nsweeps) {

    cl_int            ret;
    cl_command_queue *command_queue;
    cl_kernel        *kernel;
    cl_event         *event1, local_events[5];
    cl_mem           *dq_obj, *njac_obj, *nnodes01_obj;
    float*            dq;
    int               njac, nnodes01;

    unsigned long int npass1;

    command_queue = (cl_command_queue *)ocl_params[PARAM_COMQUE];
    kernel        = (cl_kernel *)ocl_params[PARAM_PS5_KNL1];
    event1        = (cl_event *)ocl_params[PARAM_EVENT1];

    local_events[0]  = *event1;

    npass1 = (unsigned long int)nsweeps;
    ret = clSetKernelArg(*kernel, 0, sizeof(unsigned long int),
                         &npass1);

```

```
38      if ((local_events[0] == NULL))
39        ret = clEnqueueTask(*command_queue, *kernel, 0, NULL,
40                            &local_events[1]);
41      else
42        ret = clEnqueueTask(*command_queue, *kernel, 1, local_events,
43                            &local_events[1]);
44
45      if (ret != CL_SUCCESS)
46        fprintf(stderr,"ERROR executing kernel1 (ps5)\n");
47
48      local_events[0] = local_events[1];
49
50      *event1 = local_events[0];
51
52    } // end point_solve_5_cl()
53
54  } // end extern "C"
```

APPENDIX C

FPGA BASIC CODE

The following is the basic code that was adapted to run on the FPGA prior to any optimization, it is little more than the basic sequential C code.

```
1  #include "../include/ocl_defs.h"
2
3  #define NB           5
4  #define N_SWEEPS     1
5  #define DIV          ((int)10)
6
7  __attribute__((reqd_work_group_size(1,1,1)))
8  __kernel void point_solve_5_knl
9  (unsigned long int npass1, unsigned long int npass2,
10  __global int* restrict colored_sweeps_in,
11  __global int* restrict color_indices, __global int* restrict
12                                        neq0_in,
13  __global int* restrict neq_in, __global int* restrict
14                                        solve_backwards_in,
15  __global int* restrict color_boundary_end, __global int* restrict
16                                        iam,
17  __global int* restrict jam, __global double* restrict res,
18  __global float* volatile dq, __global float* restrict a_off,
19  __global double* restrict a_diag_lu) {
20
21   int     colored_sweeps = *colored_sweeps_in;
22   int     neq0 = *neq0_in;
23   int     neq  = *neq_in;
24   int     solve_backwards = *solve_backwards_in;
25
26   int     n, i, j, k, istart, iend, icol, jam0, jam1, gid;
27   int     start, end, solve_sign, n_sweeps;
28   int     bk;
29   double f1=0, f2=0, f3=0, f4=0, f5=0, a=0;
30   double a_diag_lu_local[5][5];
31
```

```
32  // initial color index
33  int sweep_start  = 0;
34  // final color idx +/- sweep_stride
35  int sweep_end    = colored_sweeps;
36  // +/- 1
37  int sweep_stride = 1;
38
39  // parse dynamic arguments
40  n_sweeps = npass1;
41
42  if ( solve_backwards > 1 || solve_backwards < -1 ) {
43    sweep_start  = colored_sweeps - 1;
44    sweep_end    = -1;
45    sweep_stride = -1;
46  } // end if
47
48  if ( neq0 <= 0 ) {
49    sweep_start  = 0;
50    sweep_end    = 1;
51    sweep_stride = -1;
52  } // end if
53
54  for (int sweep=0; sweep < n_sweeps; ++sweep) {
55    for (int color=sweep_start; color!=sweep_end;
56         color+=sweep_stride) {
57      for (int ipass=1; ipass<=2; ++ipass) {
58        int start, end;
59        if (color > colored_sweeps) {
60          start = 1;
61          end   = 0;
62        } // end if
63        else {
64          switch(ipass) {
65          case 1:
66            if (color_boundary_end[color] == 0) {
67              start = 1;
68              end   = 0;
69            } // end if
70            else {
71              start = color_indices[2*color];
72              end   = color_boundary_end[color] - 1;
```

```
73              } // end if
74              break;
75            case 2:
76              if (color_boundary_end[color] == 0) {
77                start = color_indices[2*color];
78                end   = color_indices[2*color+1];
79              } // end if
80              else {
81                start = color_boundary_end[color] + 1;
82                end   = color_indices[2*color+1];
83              } // end if
84              break;
85            } // end switch
86          } // end else
87
88          for (n = start; n <= end; n++) {
89            // read in a_diag_lu
90            for (i=0; i<5; i++) {
91              for (j=0; j<5; j++) {
92                a_diag_lu_local[i][j] =
93                  a_diag_lu[i + j*NB + (n-1)*NB*NB];
94              } // end for (j)
95            } // end for (i)
96
97            if (solve_backwards > 0) {
98              f1 = -res[0 + (n-1)*NB];
99              f2 = -res[1 + (n-1)*NB];
100             f3 = -res[2 + (n-1)*NB];
101             f4 = -res[3 + (n-1)*NB];
102             f5 = -res[4 + (n-1)*NB];
103           } // end if (solve_backwards);
104           else {
105             f1 =  res[0 + (n-1)*NB];
106             f2 =  res[1 + (n-1)*NB];
107             f3 =  res[2 + (n-1)*NB];
108             f4 =  res[3 + (n-1)*NB];
109             f5 =  res[4 + (n-1)*NB];
110           } // end else (sweep_stride)
111
112           istart = iam[n - 1];
113           iend   = iam[n] - 1;
```

```
114
115             for (j = istart; j <= iend; j++) {
116               icol = jam[j-1] - 1;
117
118               f1 -= a_off[0+0*NB+(j-1)*NB*NB]*dq[0+icol*NB];
119               f2 -= a_off[1+0*NB+(j-1)*NB*NB]*dq[0+icol*NB];
120               f3 -= a_off[2+0*NB+(j-1)*NB*NB]*dq[0+icol*NB];
121               f4 -= a_off[3+0*NB+(j-1)*NB*NB]*dq[0+icol*NB];
122               f5 -= a_off[4+0*NB+(j-1)*NB*NB]*dq[0+icol*NB];
123
124               f1 -= a_off[0+1*NB+(j-1)*NB*NB]*dq[1+icol*NB];
125               f2 -= a_off[1+1*NB+(j-1)*NB*NB]*dq[1+icol*NB];
126               f3 -= a_off[2+1*NB+(j-1)*NB*NB]*dq[1+icol*NB];
127               f4 -= a_off[3+1*NB+(j-1)*NB*NB]*dq[1+icol*NB];
128               f5 -= a_off[4+1*NB+(j-1)*NB*NB]*dq[1+icol*NB];
129
130               f1 -= a_off[0+2*NB+(j-1)*NB*NB]*dq[2+icol*NB];
131               f2 -= a_off[1+2*NB+(j-1)*NB*NB]*dq[2+icol*NB];
132               f3 -= a_off[2+2*NB+(j-1)*NB*NB]*dq[2+icol*NB];
133               f4 -= a_off[3+2*NB+(j-1)*NB*NB]*dq[2+icol*NB];
134               f5 -= a_off[4+2*NB+(j-1)*NB*NB]*dq[2+icol*NB];
135
136               f1 -= a_off[0+3*NB+(j-1)*NB*NB]*dq[3+icol*NB];
137               f2 -= a_off[1+3*NB+(j-1)*NB*NB]*dq[3+icol*NB];
138               f3 -= a_off[2+3*NB+(j-1)*NB*NB]*dq[3+icol*NB];
139               f4 -= a_off[3+3*NB+(j-1)*NB*NB]*dq[3+icol*NB];
140               f5 -= a_off[4+3*NB+(j-1)*NB*NB]*dq[3+icol*NB];
141
142               f1 -= a_off[0+4*NB+(j-1)*NB*NB]*dq[4+icol*NB];
143               f2 -= a_off[1+4*NB+(j-1)*NB*NB]*dq[4+icol*NB];
144               f3 -= a_off[2+4*NB+(j-1)*NB*NB]*dq[4+icol*NB];
145               f4 -= a_off[3+4*NB+(j-1)*NB*NB]*dq[4+icol*NB];
146               f5 -= a_off[4+4*NB+(j-1)*NB*NB]*dq[4+icol*NB];
147
148             } // end for (j)
149
150             f2 -= a_diag_lu_local[1][0] * f1;
151             f3 -= a_diag_lu_local[2][0] * f1;
152             f4 -= a_diag_lu_local[3][0] * f1;
153             f5 -= a_diag_lu_local[4][0] * f1;
154
```

```
155             f3 -= a_diag_lu_local[2][1] * f2;
156             f4 -= a_diag_lu_local[3][1] * f2;
157             f5 -= a_diag_lu_local[4][1] * f2;
158
159             f4 -= a_diag_lu_local[3][2] * f3;
160             f5 -= (a_diag_lu_local[4][2] * f3)
161                 + (a_diag_lu_local[4][3] * f4);
162
163             f5 *= a_diag_lu_local[4][4];
164
165             // Backward...sequential access to a_diag_lu.
166             f1 -= a_diag_lu[0 + 4*NB + (n-1)*NB*NB] * f5;
167             f2 -= a_diag_lu[1 + 4*NB + (n-1)*NB*NB] * f5;
168             f3 -= a_diag_lu[2 + 4*NB + (n-1)*NB*NB] * f5;
169             f4 -= a_diag_lu[3 + 4*NB + (n-1)*NB*NB] * f5;
170             f4 *= a_diag_lu[3 + 3*NB + (n-1)*NB*NB];
171
172             f1 -= a_diag_lu[0 + 3*NB + (n-1)*NB*NB] * f4;
173             f2 -= a_diag_lu[1 + 3*NB + (n-1)*NB*NB] * f4;
174             f3 -= a_diag_lu[2 + 3*NB + (n-1)*NB*NB] * f4;
175             f3 *= a_diag_lu[2 + 2*NB + (n-1)*NB*NB];
176
177             f1 -= a_diag_lu[0 + 2*NB + (n-1)*NB*NB] * f3;
178             f2 -= a_diag_lu[1 + 2*NB + (n-1)*NB*NB] * f3;
179             f2 *= a_diag_lu[1 + 1*NB + (n-1)*NB*NB];
180
181             f1 -= a_diag_lu[0 + 1*NB + (n-1)*NB*NB] * f2;
182             f1 *= a_diag_lu[0 + 0*NB + (n-1)*NB*NB];
183
184             dq[4 + (n-1)*NB] = f5;
185             dq[3 + (n-1)*NB] = f4;
186             dq[2 + (n-1)*NB] = f3;
187             dq[1 + (n-1)*NB] = f2;
188             dq[0 + (n-1)*NB] = f1;
189
190           } // end for loop (n)
191
192         } // end for loop (ipass)
193
194       } // end for loop (color)
195
```

```
196    } // end for loop (sweep)
197
198  } // end point_solve_5_knl()
```

APPENDIX D

FPGA PARTIALLY OPTIMIZED CODE

The following is a partially optimized version of the code that was adapted to run on the FPGA. The principal difference between this code and the basic code is the consolidation of several arithmetic operations to make the execution more amenable to pipelining. This simple change resulted in a 33.5% reduction in runtime on the FPGA and is completely transparent to all other code execution architectures.

```
1  #include "../include/ocl_defs.h"
2
3  #define NB       5
4  #define N_SWEEPS 1
5  // must be a power of 2, upper limit of lmax variable
6  #define LMAX     32
7  #define DIV      ((int)10)
8
9  #define IDX1(A,B,R) A+B*NB+((R+istart)-1)*NB*NB
10 #define IDX2(A,R)   A+(jam[(R+istart)-1]-1)*NB
11
12 typedef union varr {
13         double   a[16];
14         double16 v;
15 } varr;
16
17 __constant int POW2[] = { 1, 2, 4, 8, 16, 32, 64, 128, 256, 512,
18                           1024, 2048, 4096, 8192 };
19
20 __attribute__((reqd_work_group_size(1,1,1)))
21 __kernel void point_solve_5_knl
22 (unsigned long int npass1, unsigned long int npass2,
23  __global int* restrict colored_sweeps_in,
24  __global int* restrict color_indices, __global int* restrict
25                neq0_in,
26  __global int* restrict neq_in, __global int* restrict
27                solve_backwards_in,
```

```
28  __global int* restrict color_boundary_end, __global int* restrict
29                iam,
30  __global int* restrict jam, __global double* restrict res,
31  __global float* restrict dq, __global float* restrict a_off,
32  __global double* restrict a_diag_lu) {
33
34   int      colored_sweeps = *colored_sweeps_in;
35   int      neq0 = *neq0_in;
36   int      neq  = *neq_in;
37   int      solve_backwards = *solve_backwards_in;
38
39   int      n, i, j, k, l, istart, iend, jam0, jam1, gid;
40   int      start, end, solve_sign, n_sweeps;
41   int      lmax, lmax_input, lmax_log2, i1, i2, i3;
42   double   f1=0, f2=0, f3=0, f4=0, f5=0, a=0;
43
44   local double a_diag_lu_local[5][5][4];
45
46   int sweep_start  = 0;                // initial color index
47   int sweep_end    = colored_sweeps;   // final color idx +/-
        sweep_stride
48   int sweep_stride = 1;                // +/- 1
49
50   // parse dynamic arguments
51   n_sweeps   = (int)npass1;
52   lmax_input = (int)npass2;
53
54   // find the smallest power of 2 that contains lmax_input (min 16)
55   lmax_log2 = 4;
56   for (lmax = 16; lmax < lmax_input; lmax *=2) { lmax_log2++; }
57
58   if ( solve_backwards > 1 || solve_backwards < -1 ) {
59     sweep_start  = colored_sweeps - 1;
60     sweep_end    = -1;
61     sweep_stride = -1;
62    } // end if
63
64   if ( neq0 <= 0 ) {
65     sweep_start  = 0;
66     sweep_end    = 1;
67     sweep_stride = -1;
```

```
68    } // end if
69
70    for (int sweep=0; sweep < n_sweeps; ++sweep) {
71      for (int color=sweep_start; color!=sweep_end; color+=
            sweep_stride) {
72        for (int ipass=1; ipass<=2; ++ipass) {
73          int start, end;
74          if (color > colored_sweeps) {
75            start = 1;
76            end   = 0;
77          } // end if
78          else {
79            switch(ipass) {
80            case 1:
81              if (color_boundary_end[color] == 0) {
82                start = 1;
83                end   = 0;
84              } // end if
85              else {
86                start = color_indices[2*color];
87                end   = color_boundary_end[color] - 1;
88              } // end if
89              break;
90            case 2:
91              if (color_boundary_end[color] == 0) {
92                start = color_indices[2*color];
93                end   = color_indices[2*color+1];
94              } // end if
95              else {
96                start = color_boundary_end[color] + 1;
97                end   = color_indices[2*color+1];
98              } // end if
99              break;
100           } // end switch
101         } // end else
102
103         for (n = start; n <= end; n++) {
104           // read in a_diag_lu
105           int m = n % 4;
106           for (i=0; i<25; i++) {
107             i1 = i / 5;
```

```
108             i2 = i % 5;
109             i3 = i1 + i2*NB + (n-1)*NB*NB;
110             a_diag_lu_local[i1][i2][m] = a_diag_lu[i3];
111           } // end for (i)
112
113           if (solve_backwards > 0) {
114             f1 = -res[0 + (n-1)*NB];
115             f2 = -res[1 + (n-1)*NB];
116             f3 = -res[2 + (n-1)*NB];
117             f4 = -res[3 + (n-1)*NB];
118             f5 = -res[4 + (n-1)*NB];
119           }  // end if (sweep_stride);
120           else {
121             f1 =  res[0 + (n-1)*NB];
122             f2 =  res[1 + (n-1)*NB];
123             f3 =  res[2 + (n-1)*NB];
124             f4 =  res[3 + (n-1)*NB];
125             f5 =  res[4 + (n-1)*NB];
126           } // end else (sweep_stride)
127
128           istart = iam[n - 1];
129           iend   = iam[n] - 1;
130
131           double f1a[LMAX] = { 0 };
132           double f2a[LMAX] = { 0 };
133           double f3a[LMAX] = { 0 };
134           double f4a[LMAX] = { 0 };
135           double f5a[LMAX] = { 0 };
136
137           for (j = 0; j < lmax; j++) {
138             int irow = j+istart;
139             if (irow <= iend) {
140               int icol = jam[irow-1] - 1;
141
142               f1a[j] = (a_off[0+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
143                         a_off[0+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
144                         a_off[0+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
145                         a_off[0+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
146                         a_off[0+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);
147
148               f2a[j] = (a_off[1+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
```

```
149                         a_off[1+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
150                         a_off[1+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
151                         a_off[1+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
152                         a_off[1+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);
153
154               f3a[j] = (a_off[2+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
155                         a_off[2+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
156                         a_off[2+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
157                         a_off[2+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
158                         a_off[2+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);
159
160               f4a[j] = (a_off[3+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
161                         a_off[3+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
162                         a_off[3+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
163                         a_off[3+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
164                         a_off[3+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);
165
166               f5a[j] = (a_off[4+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
167                         a_off[4+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
168                         a_off[4+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
169                         a_off[4+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
170                         a_off[4+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);
171
172             } // end if (irow)
173
174           } // end for (j)
175
176           for (j = 0; j < LMAX; j++) {
177             f1 -= f1a[j];
178             f2 -= f2a[j];
179             f3 -= f3a[j];
180             f4 -= f4a[j];
181             f5 -= f5a[j];
182           } // end for (j)
183
184           f2 -= a_diag_lu_local[1][0][m] * f1;
185           f3 -= a_diag_lu_local[2][0][m] * f1;
186           f4 -= a_diag_lu_local[3][0][m] * f1;
187           f5 -= a_diag_lu_local[4][0][m] * f1;
188
189           f3 -= a_diag_lu_local[2][1][m] * f2;
```

```
190             f4 -= a_diag_lu_local[3][1][m] * f2;
191             f5 -= a_diag_lu_local[4][1][m] * f2;
192
193             f4 -= a_diag_lu_local[3][2][m] * f3;
194             f5 -= (a_diag_lu_local[4][2][m] * f3)
195                 + (a_diag_lu_local[4][3][m] * f4);
196
197             f5 *= a_diag_lu_local[4][4][m];
198
199             // Backward...sequential access to a_diag_lu.
200             f1 -= a_diag_lu_local[0][4][m] * f5;
201             f2 -= a_diag_lu_local[1][4][m] * f5;
202             f3 -= a_diag_lu_local[2][4][m] * f5;
203             f4 -= a_diag_lu_local[3][4][m] * f5;
204             f4 *= a_diag_lu_local[3][3][m];
205
206             f1 -= a_diag_lu_local[0][3][m] * f4;
207             f2 -= a_diag_lu_local[1][3][m] * f4;
208             f3 -= a_diag_lu_local[2][3][m] * f4;
209             f3 *= a_diag_lu_local[2][2][m];
210
211             f1 -= a_diag_lu_local[0][2][m] * f3;
212             f2 -= a_diag_lu_local[1][2][m] * f3;
213             f2 *= a_diag_lu_local[1][1][m];
214
215             f1 -= a_diag_lu_local[0][1][m] * f2;
216             f1 *= a_diag_lu_local[0][0][m];
217
218             dq[4 + (n-1)*NB] = f5;
219             dq[3 + (n-1)*NB] = f4;
220             dq[2 + (n-1)*NB] = f3;
221             dq[1 + (n-1)*NB] = f2;
222             dq[0 + (n-1)*NB] = f1;
223
224           } // end for loop (n)
225         } // end for loop (ipass)
226       } // end for loop (color)
227     } // end for loop (sweep)
228 } // end point_solve_5_knl()
```

APPENDIX E

FPGA OPTIMIZED CODE

The following is a fully optimized version of the code that was adapted to run on the FPGA. The difference between this code and the partially optimized code is is that the loop structure was altered to ensure that the number of iterations was predictable prior to loop execution. While the number of loops is still unknown at compile time, there is a pre-calculation done before main loop execution which ensures that the number of iterations is exactly the same for every kernel execution. This is a more complex change, but resulted in an 88.7% reduction in runtime over the partially optimized code and a 92.5% reduction in runtime when compared to the original unoptimized (basic) code. The extra iterations consume very little in the way of compute resources, and so add a very small amount to the runtime when compared to other architectures, but provide a level of predictability that is necessary in order to more efficiently pipeline the code execution. The loops in which no calculations are actually done could be considered manually inserted "stalls" in the pipeline.

Note the use of the "ivdep" #pragma statements. These statements inform the compiler to ignore variable dependencies. This pre-compiler directive must be used very carefully since the onus of preventing race conditions and out of order calculations is now placed upon the programmer. In this particular case, the structure of the data array was generated to preclude these complications, but this may not be the case with every application.

```
1  #include "../include/ocl_defs.h"
2
3  #define NB          5
4  #define N_SWEEPS    1
5  // equivalent to cycle lag for fp operations
6  #define LMAX        12
7  #define DIV         ((int)10)
8
9  #define NMAX        32
10
```

```
11 #define IDX1(A,B,R) A+B*NB+((R+istart)-1)*NB*NB
12 #define IDX2(A,R)   A+(jam[(R+istart)-1]-1)*NB
13
14 typedef union varr {
15         double   a[16];
16         double16 v;
17 } varr;
18
19 __constant int POW2[] = { 1, 2, 4, 8, 16, 32, 64, 128, 256, 512,
20                           1024, 2048, 4096, 8192 };
21
22 __attribute__((reqd_work_group_size(1,1,1)))
23 __kernel void point_solve_5_knl
24 (unsigned long int npass1, unsigned long int npass2,
25  __global int* restrict colored_sweeps_in,
26  __global int* restrict color_indices, __global int* restrict
27                neq0_in,
28  __global int* restrict neq_in, __global int* restrict
29                solve_backwards_in,
30  __global int* restrict color_boundary_end, __global int* restrict
31                iam,
32  __global int* restrict jam, __global double* restrict res,
33  __global float* restrict dq, __global float* restrict a_off,
34  __global double* restrict a_diag_lu) {
35
36   int      colored_sweeps = *colored_sweeps_in;
37   int      neq0 = *neq0_in;
38   int      neq  = *neq_in;
39   int      solve_backwards = *solve_backwards_in;
40
41   int      n, i, j, k, l, istart, iend, jam0, jam1, nmax;
42   int      start, end, solve_sign, n_sweeps;
43   int      lmax, lmax_input, lmax_log2, i1, i2, i3;
44   double   f1=0, f2=0, f3=0, f4=0, f5=0, a=0;
45   local double a_diag_lu_local[5][5][4];
46
47   int sweep_start  = 0;               // initial color index
48   int sweep_end    = colored_sweeps;  // final color idx +/-
        sweep_stride
49   int sweep_stride = 1;               // +/- 1
50
```

```
51    // parse dynamic arguments
52    n_sweeps   = (int)npass1;
53    lmax_input = (int)npass2;
54
55    // find the smallest power of 2 that contains lmax_input (min 16)
56    lmax_log2 = 4;
57    for (lmax = 16; lmax < lmax_input; lmax *=2) { lmax_log2++; }
58
59    if ( solve_backwards > 1 || solve_backwards < -1 ) {
60      sweep_start  = colored_sweeps - 1;
61      sweep_end    = -1;
62      sweep_stride = -1;
63    } // end if
64
65    if ( neq0 <= 0 ) {
66      sweep_start  = 0;
67      sweep_end    = 1;
68      sweep_stride = -1;
69    } // end if
70
71    for (int sweep=0; sweep < n_sweeps; ++sweep) {
72      int cmax=0;
73      int nmax = 0;
74      for (int i=sweep_start; i!=sweep_end; i+=sweep_stride) {
75        if (((color_boundary_end[i]-1) - color_indices[2*i]) > cmax)
76          cmax = ((color_boundary_end[i]-1) - color_indices[2*i]);
77        if ((color_indices[2*i+1] - color_indices[2*i]) > cmax)
78          cmax = (color_indices[2*i+1] - color_indices[2*i]);
79        if ((color_indices[2*i+1] - (color_boundary_end[i]+1)) > cmax)
80          cmax = (color_indices[2*i+1] - (color_boundary_end[i]+1));
81      } //end for (i)
82
83      for (int j=sweep_start; j!=sweep_end; j+=sweep_stride) {
84        for (int ipass=1; ipass<=2; ++ipass) {
85          int start, end;
86          if (j > colored_sweeps) {
87            start = 1;
88            end   = 0;
89          } // end if
90          else {
91            switch(ipass) {
```

```
92            case 1:
93              if (color_boundary_end[j] == 0) {
94                start = 1;
95                end   = 0;
96              } // end if
97              else {
98                start = color_indices[2*j];
99                end   = color_boundary_end[j] - 1;
100             } // end if
101             break;
102           case 2:
103             if (color_boundary_end[j] == 0) {
104               start = color_indices[2*j];
105               end   = color_indices[2*j+1];
106             } // end if
107             else {
108               start = color_boundary_end[j] + 1;
109               end   = color_indices[2*j+1];
110             } // end if
111             break;
112         } // end switch
113       } // end else
114
115       for (int i=0; i<=cmax; i++) {
116         if ((i+start) <= end) {
117           if (((iam[i+start] - 1) - iam[i+start-1]) > nmax)
118             nmax = (iam[i+start] - 1) - iam[i+start-1];
119         } // end if (i+start)
120       } // end for (i)
121     } // end for (ipass)
122   } // end for (j)
123
124   #pragma ivdep
125   for (int color=sweep_start; color!=sweep_end; color+=
        sweep_stride) {
126
127     #pragma ivdep
128     for (int ipass=1; ipass<=2; ++ipass) {
129       int start, end;
130       if (color > colored_sweeps) {
131         start = 1;
```

```
132             end   = 0;
133           } // end if (color)
134           else {
135             switch(ipass) {
136             case 1:
137               if (color_boundary_end[color] == 0) {
138                 start = 1;
139                 end   = 0;
140               } // end if
141               else {
142                 start = color_indices[2*color];
143                 end   = color_boundary_end[color] - 1;
144               } // end if
145               break;
146             case 2:
147               if (color_boundary_end[color] == 0) {
148                 start = color_indices[2*color];
149                 end   = color_indices[2*color+1];
150               } // end if
151               else {
152                 start = color_boundary_end[color] + 1;
153                 end   = color_indices[2*color+1];
154               } // end if
155               break;
156             } // end switch
157           } // end else (color)
158
159           #pragma ivdep
160           for (int i=0; i <= cmax; i++) {
161             int   irow, icol; // declaring these up here outside of if
                    -blocks
162             float f1_temp, f2_temp, f3_temp, f4_temp, f5_temp;
163
164             n = i + start;
165
166             if (n <= end) {
167               if (solve_backwards > 0) {
168                 f1 = -res[0 + (n-1)*NB];
169                 f2 = -res[1 + (n-1)*NB];
170                 f3 = -res[2 + (n-1)*NB];
171                 f4 = -res[3 + (n-1)*NB];
```

```
172                 f5 = -res[4 + (n-1)*NB];
173               } // end if (sweep_stride);
174               else {
175                 f1 =  res[0 + (n-1)*NB];
176                 f2 =  res[1 + (n-1)*NB];
177                 f3 =  res[2 + (n-1)*NB];
178                 f4 =  res[3 + (n-1)*NB];
179                 f5 =  res[4 + (n-1)*NB];
180               } // end else (sweep_stride)
181
182               istart = iam[n - 1];
183               iend   = iam[n] - 1;
184
185               #pragma ivdep
186               for (int j = 0; j <= nmax; j++) {
187                 irow = j + istart;
188                 icol = jam[irow-1] - 1;
189
190                 f1_temp =(a_off[0+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
191                           a_off[0+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
192                           a_off[0+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
193                           a_off[0+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
194                           a_off[0+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);
195
196                 f2_temp =(a_off[1+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
197                           a_off[1+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
198                           a_off[1+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
199                           a_off[1+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
200                           a_off[1+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);
201
202                 f3_temp =(a_off[2+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
203                           a_off[2+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
204                           a_off[2+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
205                           a_off[2+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
206                           a_off[2+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);
207
208                 f4_temp =(a_off[3+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
209                           a_off[3+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
210                           a_off[3+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
211                           a_off[3+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
212                           a_off[3+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);
```

```
213
214                 f5_temp =(a_off[4+0*NB+(irow-1)*NB*NB]*dq[0+icol*NB] +
215                           a_off[4+1*NB+(irow-1)*NB*NB]*dq[1+icol*NB] +
216                           a_off[4+2*NB+(irow-1)*NB*NB]*dq[2+icol*NB] +
217                           a_off[4+3*NB+(irow-1)*NB*NB]*dq[3+icol*NB] +
218                           a_off[4+4*NB+(irow-1)*NB*NB]*dq[4+icol*NB]);
219
220                 if ((j+istart) <= iend) {
221                   f1 -= f1_temp;
222                   f2 -= f2_temp;
223                   f3 -= f3_temp;
224                   f4 -= f4_temp;
225                   f5 -= f5_temp;
226                 } // end if (j+istart)
227                 else {
228                   f1 -= 0;
229                   f2 -= 0;
230                   f3 -= 0;
231                   f4 -= 0;
232                   f5 -= 0;
233                 } // end else (j+istart)
234
235               } // end for loop (j)
236
237               f2 -= a_diag_lu[1 + 0*NB + (n-1)*NB*NB] * f1;
238               f3 -= a_diag_lu[2 + 0*NB + (n-1)*NB*NB] * f1;
239               f4 -= a_diag_lu[3 + 0*NB + (n-1)*NB*NB] * f1;
240               f5 -= a_diag_lu[4 + 0*NB + (n-1)*NB*NB] * f1;
241
242               f3 -= a_diag_lu[2 + 1*NB + (n-1)*NB*NB] * f2;
243               f4 -= a_diag_lu[3 + 1*NB + (n-1)*NB*NB] * f2;
244               f5 -= a_diag_lu[4 + 1*NB + (n-1)*NB*NB] * f2;
245
246               f4 -= a_diag_lu[3 + 2*NB + (n-1)*NB*NB] * f3;
247               f5 -= (a_diag_lu[4 + 2*NB + (n-1)*NB*NB] * f3)
248                   + (a_diag_lu[4 + 3*NB + (n-1)*NB*NB] * f4);
249
250               f5 *= a_diag_lu[4 + 4*NB + (n-1)*NB*NB];
251
252               // Backward...sequential access to a_diag_lu.
253               f1 -= a_diag_lu[0 + 4*NB + (n-1)*NB*NB] * f5;
```

```
254             f2 -= a_diag_lu[1 + 4*NB + (n-1)*NB*NB] * f5;
255             f3 -= a_diag_lu[2 + 4*NB + (n-1)*NB*NB] * f5;
256             f4 -= a_diag_lu[3 + 4*NB + (n-1)*NB*NB] * f5;
257             f4 *= a_diag_lu[3 + 3*NB + (n-1)*NB*NB];
258
259             f1 -= a_diag_lu[0 + 3*NB + (n-1)*NB*NB] * f4;
260             f2 -= a_diag_lu[1 + 3*NB + (n-1)*NB*NB] * f4;
261             f3 -= a_diag_lu[2 + 3*NB + (n-1)*NB*NB] * f4;
262             f3 *= a_diag_lu[2 + 2*NB + (n-1)*NB*NB];
263
264             f1 -= a_diag_lu[0 + 2*NB + (n-1)*NB*NB] * f3;
265             f2 -= a_diag_lu[1 + 2*NB + (n-1)*NB*NB] * f3;
266             f2 *= a_diag_lu[1 + 1*NB + (n-1)*NB*NB];
267
268             f1 -= a_diag_lu[0 + 1*NB + (n-1)*NB*NB] * f2;
269             f1 *= a_diag_lu[0 + 0*NB + (n-1)*NB*NB];
270
271             dq[4 + (n-1)*NB] = f5;
272             dq[3 + (n-1)*NB] = f4;
273             dq[2 + (n-1)*NB] = f3;
274             dq[1 + (n-1)*NB] = f2;
275             dq[0 + (n-1)*NB] = f1;
276
277           } // end if (n)
278         } // end for loop (i)
279       } // end for loop (ipass)
280     } // end for loop (color)
281   } // end for loop (sweep)
282 } // end point_solve_5_knl()
```

APPENDIX F

GPU BASIC CODE

The following is non-optimized code that has been altered to run in a naive fashion on a Graphics Processing Unit (GPU). Although it was written specifically to run on a GPU, a similar version should run on any Single Instruction Multiple Data (SIMD) archiecture. The outer loops are executed on the CPU, while the inner loops are distributed over numerous GPU processors; everything inside the color sweeps (see the basic code in Appendix A for an implementation of all of the loops in the same code segment) is included here. The outer loops in this case are implemented in FORTRAN. An excerpt of the FORTRAN used to implement these outer loops is shown at the end of this appendix.

```
1  #include "../include/ocl_defs.h"
2
3  #define nb          5
4  #define n_sweeps    1
5  #define DIV         ((int)10)
6
7  #define BLOCK_DIM_X BLOCK_DIM_X_PS5
8  #define BLOCK_DIM_Y BLOCK_DIM_Y_PS5
9
10 __kernel void point_solve_5_knl
11 (unsigned long int npass1, unsigned long int npass2,
12  __global int* restrict iam, __global int* restrict jam,
13  __global double* restrict res, __global float* restrict dq,
14  __global float* restrict a_off,
15  __global double* restrict a_diag_lu) {
16
17   int     n, j, k, l, istart, iend, icol, jam0, jam1, gid, lid,
18           tx, ty;
19   int     start, end, solve_sign;
20   int     bk;
21   double  f1=0, f2=0, f3=0, f4=0, f5=0;
22
```

```
23    // parse dynamic arguments
24    start      = npass1;
25    end        = npass2/DIV;
26    solve_sign = npass2 - DIV*end - 2;
27
28    // calculate the index variables
29    gid = get_global_id(0);
30    lid = get_local_id(0);
31    bk  = gid/(BLOCK_DIM_X*BLOCK_DIM_Y);
32    ty  = lid/BLOCK_DIM_X;
33    tx  = lid - BLOCK_DIM_X*ty;
34    l   = tx / 5;
35    k   = tx - 5*l;
36    n   = start + gid/BLOCK_DIM_X; // constant over a warp
37
38    if (n > end || tx > 0) return;
39
40    if (solve_sign > 0) {
41      f1 = -(res[0 + (n-1)*nb]);
42      f2 = -(res[1 + (n-1)*nb]);
43      f3 = -(res[2 + (n-1)*nb]);
44      f4 = -(res[3 + (n-1)*nb]);
45      f5 = -(res[4 + (n-1)*nb]);
46    }  // end if (solve_backwards);
47    else {
48      f1 =  (res[0 + (n-1)*nb]);
49      f2 =  (res[1 + (n-1)*nb]);
50      f3 =  (res[2 + (n-1)*nb]);
51      f4 =  (res[3 + (n-1)*nb]);
52      f5 =  (res[4 + (n-1)*nb]);
53    }  // end else (solve_backwards)
54
55    istart = iam[n - 1];
56    iend   = iam[n] - 1;
57
58    for (j = istart; j <= iend; j++) {
59      icol = jam[j-1] - 1;
60
61      f1 -= a_off[0 + 0*nb + (j-1)*nb*nb] * dq[0 + icol*nb];
62      f2 -= a_off[1 + 0*nb + (j-1)*nb*nb] * dq[0 + icol*nb];
63      f3 -= a_off[2 + 0*nb + (j-1)*nb*nb] * dq[0 + icol*nb];
```

```
64      f4 -= a_off[3 + 0*nb + (j-1)*nb*nb] * dq[0 + icol*nb];
65      f5 -= a_off[4 + 0*nb + (j-1)*nb*nb] * dq[0 + icol*nb];
66
67      f1 -= a_off[0 + 1*nb + (j-1)*nb*nb] * dq[1 + icol*nb];
68      f2 -= a_off[1 + 1*nb + (j-1)*nb*nb] * dq[1 + icol*nb];
69      f3 -= a_off[2 + 1*nb + (j-1)*nb*nb] * dq[1 + icol*nb];
70      f4 -= a_off[3 + 1*nb + (j-1)*nb*nb] * dq[1 + icol*nb];
71      f5 -= a_off[4 + 1*nb + (j-1)*nb*nb] * dq[1 + icol*nb];
72
73      f1 -= a_off[0 + 2*nb + (j-1)*nb*nb] * dq[2 + icol*nb];
74      f2 -= a_off[1 + 2*nb + (j-1)*nb*nb] * dq[2 + icol*nb];
75      f3 -= a_off[2 + 2*nb + (j-1)*nb*nb] * dq[2 + icol*nb];
76      f4 -= a_off[3 + 2*nb + (j-1)*nb*nb] * dq[2 + icol*nb];
77      f5 -= a_off[4 + 2*nb + (j-1)*nb*nb] * dq[2 + icol*nb];
78
79      f1 -= a_off[0 + 3*nb + (j-1)*nb*nb] * dq[3 + icol*nb];
80      f2 -= a_off[1 + 3*nb + (j-1)*nb*nb] * dq[3 + icol*nb];
81      f3 -= a_off[2 + 3*nb + (j-1)*nb*nb] * dq[3 + icol*nb];
82      f4 -= a_off[3 + 3*nb + (j-1)*nb*nb] * dq[3 + icol*nb];
83      f5 -= a_off[4 + 3*nb + (j-1)*nb*nb] * dq[3 + icol*nb];
84
85      f1 -= a_off[0 + 4*nb + (j-1)*nb*nb] * dq[4 + icol*nb];
86      f2 -= a_off[1 + 4*nb + (j-1)*nb*nb] * dq[4 + icol*nb];
87      f3 -= a_off[2 + 4*nb + (j-1)*nb*nb] * dq[4 + icol*nb];
88      f4 -= a_off[3 + 4*nb + (j-1)*nb*nb] * dq[4 + icol*nb];
89      f5 -= a_off[4 + 4*nb + (j-1)*nb*nb] * dq[4 + icol*nb];
90
91    } // end for (j)
92
93    dq[4 + (n-1)*nb] = f5;
94    dq[3 + (n-1)*nb] = f4;
95    dq[2 + (n-1)*nb] = f3;
96    dq[1 + (n-1)*nb] = f2;
97    dq[0 + (n-1)*nb] = f1;
98
99  } // end point_solve_5_knl()
```

The following is an excerpt of the FORTRAN code used to implement the outer loops of the computation. This code segment calls the wrapper that was written in C and is linked with the compiled FORTRAN code to handle the invokation of the OpenCL code.

```
1   write (*,*) '  Starting OpenCL point_solve_5...'
2   call ocl_setvar(c_loc(ocl_params), c_loc(ocl_data), D_RES,      &
3                   c_loc(res_seq))
4   call ocl_setvar(c_loc(ocl_params), c_loc(ocl_data), D_ADIAG,    &
5                   c_loc(a_diag_seq_temp))
6   call ocl_setvar(c_loc(ocl_params), c_loc(ocl_data), D_AOFF,     &
7                   c_loc(a_off_seq))
8   call ocl_setvar(c_loc(ocl_params), c_loc(ocl_data), D_DQ,       &
9                   c_loc(dq_seq))
10  call cpu_time(start_time)
11  do i = 1, (n_meanflow_iters+0)   ! outer sweeps
12     call point_solve_5_cl(c_loc(ocl_params), c_loc(ocl_data),    &
13                           colored_sweeps, c_loc(color_indices), &
14                           nnodes0, nnz0,                        &
15                           relaxation_schedule_direction_1,      &
16                           c_loc(color_boundary_end))
17  end do
18  call cpu_time(finish_time)
19  ps5_dt_ocl = (finish_time - start_time)*1E3 - to
```

The following is a listing of the C wrapper that is called by the FORTRAN code and is used to invoke the OpenCL code.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <CL/cl.h>
4
5  #include "ocl_defs.h"
6  #include "ocl_helpers.h"
7
8  #define N_SWEEPS 1
9  #define BLOCK_DIM_X BLOCK_DIM_X_PS5
10 #define BLOCK_DIM_Y BLOCK_DIM_Y_PS5
11 #define DIV         ((int)10)
12
13 extern "C" {
14   void point_solve_5_cl
15   (intptr_t *ocl_params, intptr_t *ocl_data, int colored_sweeps,
16    int *color_indices, int neq0, int neq, int solve_backwards,
17    int *color_boundary_end) {
18
19     cl_int           ret;
20     cl_command_queue *command_queue;
21     cl_kernel        *kernel;
22     size_t           local_item_size,global_item_size;
23
24     // initial color index
25     int sweep_start  = 0;
26     // final color index +/- sweep_stride
27     int sweep_end    = colored_sweeps;
28     // +/- 1
29     int sweep_stride = 1;
30     unsigned long int npass1,npass2;
31
32     command_queue    = (cl_command_queue *)ocl_params[PARAM_COMQUE];
33     kernel           = (cl_kernel *)ocl_params[PARAM_PS5_KNL];
34     local_item_size  = BLOCK_DIM_X*BLOCK_DIM_Y;
35
36     if ( solve_backwards > 1 || solve_backwards < -1 ) {
37       sweep_start  = colored_sweeps - 1;
```

```
38        sweep_end    = -1;
39        sweep_stride = -1;
40      } // end if (solve_backwards)
41
42      if ( neq0 <= 0 ) {
43        sweep_start  = 0;
44        sweep_end    = 1;
45        sweep_stride = -1;
46      } // end if (neq0)
47
48      for (int sweep=0; sweep < N_SWEEPS; ++sweep) {
49        for (int color=sweep_start; color!=sweep_end;
50             color+=sweep_stride) {
51          for (int ipass=1; ipass<=2; ++ipass) {
52            int start, end;
53            if (color > colored_sweeps) {
54              start = 1;
55              end   = 0;
56            } // end if (color)
57            else {
58              switch(ipass) {
59              case 1:
60                if (color_boundary_end[color] == 0) {
61                  start = 1;
62                  end   = 0;
63                } // end if (color_boundary_end)
64                else {
65                  start = color_indices[2*color];
66                  end   = color_boundary_end[color] - 1;
67                } // end else (color_boundary_end)
68                break;
69              case 2:
70                if (color_boundary_end[color] == 0) {
71                  start = color_indices[2*color];
72                  end   = color_indices[2*color+1];
73                } // end if (color_boundary_end)
74                else {
75                  start = color_boundary_end[color] + 1;
76                  end   = color_indices[2*color+1];
77                } // end else (color_boundary_end)
78                break;
```

```
79              } // end switch (ipass)
80            } // end else (color)
81
82            if(start < (end + 1)) {
83              npass1 = start;
84              npass2 = DIV*end + (sweep_stride+2);
85              ret = clSetKernelArg(*kernel, 0,
86                                   sizeof(unsigned long int),
87                                   &npass1);
88              ret = clSetKernelArg(*kernel, 1,
89                                   sizeof(unsigned long int),
90                                   &npass2);
91
92              // Execute the OpenCL kernel
93              global_item_size = EVENSIZE((end - start + 1)*
                    BLOCK_DIM_X,
94                                          local_item_size);
95              ret = clEnqueueNDRangeKernel(*command_queue, *kernel, 1,
96                                           NULL, &global_item_size,
97                                           &local_item_size, 0, NULL,
98                                           NULL);
99              if (ret != CL_SUCCESS)
100               fprintf(stderr,"ERROR executing kernel (ps5)\n");
101           } // end if (start)
102         } // end for loop (ipass)
103       } // end for loop (color)
104     } // end forloop (sweep)
105   } // end point_solve_5_kernel()
106 } // end extern "C"
```

APPENDIX G

GPU PARTIALLY OPTIMIZED CODE

The following is code that has been optimized for the GPU in OpenCL; it is considered partially optimized in this context because it is not the native CUDA code that shows a marginal, but quantifiable, 0.6% edge over its OpenCL counterpart. This version makes use of shared memory (this is the same between the CUDA and OpenCL versions) as well as a reduction vector that executes a 5-thread summation consolidated to every 5th thread in the warp using three steps. This is less efficient than the CUDA "__shfl" native command (which takes but a single step), but is more efficient than adding them up sequentially every 5th thread (which would take 5 steps). This is emblematic of the types of tradeoffs that must occur to create portability. As with the non-optimized GPU version, this code only represents the code inside the color sweep loop. The FORTRAN code and the C wrapper code are exactly the same as with the non-optimized version. See Appendix H for the optimized CUDA code that accomplishes the same task.

```
1  #include "../../include/ocl_defs.h"
2
3  #define nb           5
4  #define n_sweeps     1
5  #define DIV          ((int)10)
6
7  #define BLOCK_DIM_X  BLOCK_DIM_X_PS5
8  #define BLOCK_DIM_Y  BLOCK_DIM_Y_PS5
9
10 #define A_OFF(i,j,k)      a_off[(i)+((j)*nb)+ \
11                                 ((unsigned long long)(k)*nb*nb)]
12 #define A_DIAG_LU(i,j,k) a_diag_lu[(i)+((j)*nb)+((k)*nb*nb)]
13 #define DQ_IN(i,j)        dq[(i)+((j)*nb)]
14 #define DQ_OUT(i,j)       dq[(i)+((j)*nb)]
15 #define RES(i, j)         res[(i)+((j)*nb)]
16
17 // Parallel reduction vector
```

```
18 typedef struct prvec {
19   union {
20     double  a[8];
21     double8 v;
22   } data;
23 } prvec;
24
25 __kernel void point_solve_5_kernel
26 (unsigned long int npass1, unsigned long int npass2,
27  __global int* restrict iam, __global int* restrict jam,
28  __global double* restrict res, __global float* restrict dq,
29  __global double* restrict a_diag_lu,
30  __global float* restrict a_off) {
31
32   int     n, j, k, l, istart, iend, jam0, jam1, gid, lid, tx, ty;
33   int     start, end, solve_sign;
34   int     bk;
35   double  fk;
36   double  f1=0, f2=0, f3=0, f4=0, f5=0;
37   __local prvec  fc[5][BLOCK_DIM_Y];
38   __local double fs[5][BLOCK_DIM_Y];
39   __local double a_diag_lu_shared [5][5][BLOCK_DIM_Y];
40
41   // parse dynamic arguments
42   start      = npass1;
43   end        = npass2/DIV;
44   solve_sign = npass2 - DIV*end - 2;
45
46   // initialize parallel reduction vectors
47   #pragma unroll
48   for (int i=0; i < BLOCK_DIM_Y; i++)
49     fc[0][i].data.v = fc[1][i].data.v = fc[2][i].data.v = \
50                       fc[3][i].data.v = fc[4][i].data.v = 0.0;
51
52   // calculate the index variables
53   gid = get_global_id(0);
54   lid = get_local_id(0);
55   bk  = gid/(BLOCK_DIM_X*BLOCK_DIM_Y);
56   ty  = lid/BLOCK_DIM_X;
57   tx  = lid - BLOCK_DIM_X*ty;
58   l   = tx / 5;
```

```
59    k    = tx - 5*l;
60    n    = start + gid/BLOCK_DIM_X - 1;
61
62    // the last 7 threads in the warp are unused
63    if (n >= end || l >= 5) return;
64
65    istart = iam[n];
66    iend   = iam[n + 1] - 1;
67
68    // Loop over Non Zeros
69    fk = 0;
70    for(j = istart-1; j < iend; j++) {
71      jam0 = jam[j];
72      f1   = A_OFF(k,l,j);
73      f2   = DQ_IN(l,jam0-1);
74      fk  += f1 * f2;
75    } // end for (j)
76
77    // Reduction along the subcolumns, threads with v.s0 holding
78    // the complete sum
79    fc[k][ty].data.a[l] = fk;
80
81    // Collectively load a_diag_lu into shared memory
82    a_diag_lu_shared[k][l][ty] = A_DIAG_LU(k, l, n);
83
84    // Save results of off-diagonal multiplication in shared memory
85    if (l != 0) return;
86
87    fc[k][ty].data.v.s0123     += fc[k][ty].data.v.s4567;
88    fc[k][ty].data.v.s01       += fc[k][ty].data.v.s23;
89    fc[k][ty].data.v.s0        += fc[k][ty].data.v.s1;
90
91    fs[k][ty] = -solve_sign*RES(k, n) - fc[k][ty].data.v.s0;
92
93    // this must be a barrier and not a simple fence
94    barrier(CLK_LOCAL_MEM_FENCE);
95
96    // Redistribute work from all warps to first four threads
97    // in the first warp
98    n += tx;
99    if ((tx >= BLOCK_DIM_Y) || (ty != 0) || (n >= end)) return;
```

```
100
101     // Retrieve data from shared memory
102     f1 = fs[0][tx];
103     f2 = fs[1][tx];
104     f3 = fs[2][tx];
105     f4 = fs[3][tx];
106     f5 = fs[4][tx];
107
108     // Forward...sequential access to a_diag_lu
109
110     f2 = f2 - a_diag_lu_shared[1][0][tx] * f1;
111     f3 = f3 - a_diag_lu_shared[2][0][tx] * f1;
112     f4 = f4 - a_diag_lu_shared[3][0][tx] * f1;
113     f5 = f5 - a_diag_lu_shared[4][0][tx] * f1;
114
115     f3 = f3 - a_diag_lu_shared[2][1][tx] * f2;
116     f4 = f4 - a_diag_lu_shared[3][1][tx] * f2;
117     f5 = f5 - a_diag_lu_shared[4][1][tx] * f2;
118
119     f4 = f4 - a_diag_lu_shared[3][2][tx] * f3;
120     f5 = f5 - a_diag_lu_shared[4][2][tx] * f3;
121
122     f5 = ((f5 - a_diag_lu_shared[4][3][tx] * f4)
123           * a_diag_lu_shared[4][4][tx]);
124
125     // Backward...sequential access to a_diag_lu.
126
127     f1 = f1 - a_diag_lu_shared[0][4][tx] * f5;
128     f2 = f2 - a_diag_lu_shared[1][4][tx] * f5;
129     f3 = f3 - a_diag_lu_shared[2][4][tx] * f5;
130     f4 = ((f4 - a_diag_lu_shared[3][4][tx] * f5)
131           * a_diag_lu_shared[3][3][tx]);
132
133     f1 = f1 - a_diag_lu_shared[0][3][tx] * f4;
134     f2 = f2 - a_diag_lu_shared[1][3][tx] * f4;
135     f3 = ((f3 - a_diag_lu_shared[2][3][tx] * f4)
136           * a_diag_lu_shared[2][2][tx]);
137
138     f1 = f1 - a_diag_lu_shared[0][2][tx] * f3;
139     f2 = ((f2 - a_diag_lu_shared[1][2][tx] * f3)
140           * a_diag_lu_shared[1][1][tx]);
```

```
141
142    f1 = ((f1 - a_diag_lu_shared[0][1][tx] * f2)
143          * a_diag_lu_shared[0][0][tx]);
144
145    DQ_OUT(0,n)= f1;
146    DQ_OUT(1,n)= f2;
147    DQ_OUT(2,n)= f3;
148    DQ_OUT(3,n)= f4;
149    DQ_OUT(4,n)= f5;
150
151  } // end point_solve_5_kernel()
```

APPENDIX H

GPU OPTIMIZED CODE

The following is code that has been fully optimized for the GPU in CUDA. It makes use of shared memory and intra-thread communication using the "__shfl" native CUDA command (which takes a single step to consolidate a sum using information from 5 separate threads), and is more efficient than either adding them up sequentially every 5th thread (which would take 5 steps) or the vector reduction strategy used in the OpenCL version (which takes 3 steps). As stated in the OpenCL code lead-in (see Appendix G), this is an example of the type of tradeoff that must occur to create portability, but results in increased performance when using language extensions optimized for specific hardware. As with the non-optimized GPU version, this code only represents the code inside the color sweep loop. The FORTRAN code and the C wrapper code are exactly the same as with the non-optimized version.

```
1  #include <cstddef>
2  #include <cuda.h>
3  #include "gpu_util.h"
4  #include <assert.h>
5
6  #include "block_dim.h"
7  #define nb 5
8  #define n_sweeps 1
9  #define BLOCK_DIM_X BLOCK_DIM_X_PS5
10 #define BLOCK_DIM_Y BLOCK_DIM_Y_PS5
11
12 #define A_OFF(i,j,k) a_off[(i)+((j)*nb)+ \
13                            ((unsigned long long)(k)*nb*nb)]
14 #define A_DIAG_LU(i,j,k) a_diag_lu[(i)+((j)*nb)+((k)*nb*nb)]
15 #define DQ_IN(i,j) dq_in[(i)+((j)*nb)]
16 #define DQ_OUT(i,j) dq[(i)+((j)*nb)]
17 #define RES(i, j) res[(i)+((j)*nb)]
18
19
20 __global__ void cuda_point5_kernel
```

```
21 (int solve_sign,idx_t start, idx_t end,
22         int_const_ptr_t const iam, int_const_ptr_t const jam,
23         real_const_ptr_t const a_off, real8_const_ptr_t const
               a_diag_lu,
24         real_ptr_t dq, real8_const_ptr_t const res)
25 {
26
27   real_const_ptr_t const dq_in = dq;
28
29     __shared__ real_t
30     fs[5][BLOCK_DIM_Y];
31     __shared__ real8_t
32          a_diag_lu_shared[5][5][BLOCK_DIM_Y];
33
34     int const k = threadIdx.x % 5;
35     int const l = threadIdx.x / 5;
36     int n = start + blockIdx.x * blockDim.y + threadIdx.y - 1;
37
38     if (n >= end || l >= 5)
39         return;
40
41     idx_t istart = iam[n];
42     idx_t iend = iam[n + 1] - 1;
43
44     real_t  fk;
45     real8_t f1, f2, f3, f4, f5;
46
47     // Loop over Non Zeros, 2x unrolled
48     fk = 0;
49     int jam0, jam1;
50     //double dq0, dq1;
51
52     int j = istart-1;
53
54     jam1 = jam[j];
55
56     for( ; j<iend; j++) {
57       jam0 = jam1;
58       jam1 = jam[j+1];
59       fk += A_OFF(k,l,j)*  DQ_IN(l,jam0-1);
60     }
```

```
61
62      // Reduction along the subcolumns,
63      // threads with l=0 hold the complete sum
64      f1 = fk;
65      f1 = f1 + __shfl(fk, k + 1 * 5);
66      f1 = f1 + __shfl(fk, k + 2 * 5);
67      f1 = f1 + __shfl(fk, k + 3 * 5);
68      f1 = f1 + __shfl(fk, k + 4 * 5);
69
70      f1 = -solve_sign*RES(k, n) - f1;
71
72      // Save results of off-diagonal multiplication in shared memory
73      if (l == 0) {
74          fs[k][threadIdx.y] = f1;
75      }
76
77      // Collectively load a_diag_lu into shared memory
78      a_diag_lu_shared[k][l][threadIdx.y] = A_DIAG_LU(k, l, n);
79
80      __syncthreads();
81
82      // Redistribute work from all warps to first four threads
83      // in the first warp
84      n += threadIdx.x;
85
86      if (threadIdx.x < BLOCK_DIM_Y && threadIdx.y == 0 && n < end) {
87
88          // Retrieve data from shared memory
89          f1 = fs[0][threadIdx.x];
90          f2 = fs[1][threadIdx.x];
91          f3 = fs[2][threadIdx.x];
92          f4 = fs[3][threadIdx.x];
93          f5 = fs[4][threadIdx.x];
94
95          // Forward...sequential access to a_diag_lu
96
97          f2 = f2 - a_diag_lu_shared[1][0][threadIdx.x] * f1;
98          f3 = f3 - a_diag_lu_shared[2][0][threadIdx.x] * f1;
99          f4 = f4 - a_diag_lu_shared[3][0][threadIdx.x] * f1;
100         f5 = f5 - a_diag_lu_shared[4][0][threadIdx.x] * f1;
101
```

```
102         f3 = f3 - a_diag_lu_shared[2][1][threadIdx.x] * f2;
103         f4 = f4 - a_diag_lu_shared[3][1][threadIdx.x] * f2;
104         f5 = f5 - a_diag_lu_shared[4][1][threadIdx.x] * f2;
105
106         f4 = f4 - a_diag_lu_shared[3][2][threadIdx.x] * f3;
107         f5 = f5 - a_diag_lu_shared[4][2][threadIdx.x] * f3;
108
109         f5 = ((f5 - a_diag_lu_shared[4][3][threadIdx.x] * f4)
110                 * a_diag_lu_shared[4][4][threadIdx.x]);
111
112         // Backward...sequential access to a_diag_lu.
113
114         f1 = f1 - a_diag_lu_shared[0][4][threadIdx.x] * f5;
115         f2 = f2 - a_diag_lu_shared[1][4][threadIdx.x] * f5;
116         f3 = f3 - a_diag_lu_shared[2][4][threadIdx.x] * f5;
117         f4 = ((f4 - a_diag_lu_shared[3][4][threadIdx.x] * f5)
118                 * a_diag_lu_shared[3][3][threadIdx.x]);
119
120         f1 = f1 - a_diag_lu_shared[0][3][threadIdx.x] * f4;
121         f2 = f2 - a_diag_lu_shared[1][3][threadIdx.x] * f4;
122         f3 = ((f3 - a_diag_lu_shared[2][3][threadIdx.x] * f4)
123                 * a_diag_lu_shared[2][2][threadIdx.x]);
124
125         f1 = f1 - a_diag_lu_shared[0][2][threadIdx.x] * f3;
126         f2 = ((f2 - a_diag_lu_shared[1][2][threadIdx.x] * f3)
127                 * a_diag_lu_shared[1][1][threadIdx.x]);
128
129         f1 = ((f1 - a_diag_lu_shared[0][1][threadIdx.x] * f2)
130                 * a_diag_lu_shared[0][0][threadIdx.x]);
131
132         DQ_OUT(0,n)= f1;
133         DQ_OUT(1,n)= f2;
134         DQ_OUT(2,n)= f3;
135         DQ_OUT(3,n)= f4;
136         DQ_OUT(4,n)= f5;
137     }
138 }
```

APPENDIX I

DATA TABLES

Tables of data from trial runs using optimized and non-optimized code.

TABLE 4: Clock Validation for Optimized Code (time in ms)

Run #	**CUDA nvprof**	**CUDA mclock**	**diff**	**OpenCL mclock**
1	120.60	124.91	4.31	126.15
2	120.40	125.29	4.89	126.13
3	120.55	125.37	4.82	126.24
4	120.50	125.09	4.59	127.43
5	123.11	127.44	4.33	127.03
6	123.51	128.05	4.54	126.42
7	120.61	125.22	4.61	125.87
8	120.66	125.22	4.56	126.15
9	120.59	125.22	4.63	126.11
10	120.38	124.65	4.27	126.29
AVG	121.09	125.65	4.56	126.38

Note: Ten runs were performed and their results averaged to produce the numbers displayed in Tables 8 and 9.

TABLE 5: Clock times for Optimized Code without profiler (time in ms)

Run #	**CUDA mclock**	**OpenCL mclock**
1	126.22	126.52
2	126.29	126.60
3	123.06	126.38
4	123.38	127.78
5	123.40	126.53
6	123.31	126.28
7	123.61	126.51
8	126.35	126.68
9	127.68	126.73
10	123.38	127.76
AVG	124.67	126.78

TABLE 6: Clock Validation for Non-Optimized (Baseline) Code (time in ms)

Run #	**CUDA nvprof**	**CUDA mclock**	**diff**	**OpenCL mclock**
1	805.67	811.74	6.07	821.92
2	805.15	811.43	6.28	817.08
3	805.12	811.43	6.31	816.68
4	805.19	811.69	6.50	817.05
5	805.18	811.05	5.87	816.92
6	809.78	815.87	6.09	816.63
7	811.25	817.31	6.06	816.78
8	805.51	811.72	6.21	816.83
9	805.25	811.11	5.86	817.31
10	805.39	811.51	6.12	816.73
AVG	806.35	812.49	6.14	817.40

TABLE 7: Clock times for Non-Optimized code without profiler (time in ms)

Run #	**CUDA mclock**	**OpenCL mclock**
1	816.12	818.22
2	812.95	818.22
3	810.67	817.99
4	814.10	818.06
5	811.61	818.48
6	810.67	818.11
7	815.19	818.81
8	811.02	818.36
9	810.98	818.24
10	810.78	818.04
AVG	812.41	818.25

TABLE 8: OpenMP Trials for ARM CPU (times in ms)

Run Type	**2-Cores**	**4-Cores**	**8-Cores**	**16-Cores**
Generic OpenMP	73948.36	38184.80	19856.00	9903.13
Native Compiler	7778.57	4245.74	2507.89	1349.70
Optimized	7173.11	4073.80	2448.82	1340.03

TABLE 9: OpenMP Trials for Intel x86 CPU (times in ms)

Run Type	**2-Cores**	**4-Cores**	**8-Cores**	**16-Cores**
Generic OpenMP	23189.99	12067.51	7222.09	3768.25
Native Compiler	7725.20	3731.14	2024.83	1121.40
Optimized	6235.33	3028.18	1725.86	986.19

APPENDIX J

HARDWARE SPECIFICATIONS

The following data describe the hardware on which the code was ultimately executed. This data is meant to provide context for the statistical timing data used to compare algorithm implimentations. While the platforms themselves are not being compared as such, the data is still useful as a touchstone to provide an indication of how well the code could be expected to run additional hardware platforms.

TABLE 10: GPU Hardware Data

Description	**Value**
Vendor	Nvidia
Identity String	Pascal P100
Nominal Bandwidth	732 GB/s max
Measured Bandwidth	292 GB/s
# Cores (FP32)	3584
FP32 TFLOPS	9.3
Memory	16 GB
L2 Cache	4096 KB
Shared (local) Memory	up to 96 KB

Since the Nvidia profiling utility does not explicitly calculate real memory bandwidth used, a special kernel was prepared that performed all of the same floating point reads that are actually performed within the point solver kernel, but none of the floating point operations. The time difference between the fully functional kernel and this specially prepared kernel (with the overhead subtracted out) is indicative of the amount of time actually spent reading the data from the on-device memory.

Cost data for AWS Graviton indicates that identical instances run at about 40% less per core than an equivalent Intel x86 instance (Skylake architecture optimized for computational efficiency) as of November 2018. Whether this is more efficient clearly depends on the specific characteristics of the computational load being studied. In this case, noting that the point solver problem runs in 1340 ms (see Table 8) using

TABLE 11: FPGA Hardware Data

Description	**Value**
Vendor	Intel
Identity String	PAC10 (Aria)
Nominal Bandwidth	36 GB/s max
Measured Bandwidth	3.69 GB/s
# Cores (FP32)	NA
FP32 TFLOPS	1.5
Memory	8 GB
L2 Cache	512 KB
Shared (local) Memory	256 KB

TABLE 12: ARM CPU Hardware Data

Description	**Value**
Vendor	Arm
Identity String	AWS Graviton (Cortex-A72)
Nominal Bandwidth	51.2 GB/s max
Estimated Bandwidth	25.6 GB/s max
Clock Speed	1.3 GHz
# Cores	16
FP32 TFLOPS	0.166 (8 FLOPS/core/cycle)
Memory	32 GB
L2 Cache	2048 KB
Shared (local) Memory	NA

optimized command line options on the ARM (Graviton) processor and at 986 ms (see Table 9) on the x86 Skylake architecture, the cost to run on ARM is (1340*(1-0.4))/986 = 81.5% of the x86.

TABLE 13: x86 CPU Hardware Data

Description	**Value**
Vendor	Intel
Identity String	Haswell (Family: 6, Model: 63)
Nominal Bandwidth	128 GB/s
Estimated Bandwidth	61.5 GB/s max
Clock Speed	2.9 GHz
# Cores	16
FP32 TFLOPS	1.484 (32 FLOPS/core/cycle)
Memory	32 GB
L2 Cache	1024 KB
Shared (local) Memory	NA

VITA

Jason Orender
Department of Computer Science
Old Dominion University
Norfolk, VA 23529

Jason Orender spent a 20 year career as an officer in the US Navy and retired in the Summer of 2015, whereupon he promptly initiated a degree plan in Computer Science at Old Dominion University. He has participated in multiple NASA hack-a-thons with the intent to contribute to the Fun3D fully unstructured 3D computational fluid dynamics code base and has been cited as a contributor in several of NASA's more recent papers and presentations on the subject. He developed an interest in high performance computation, and this work is one result of that effort. He is also currently enrolled in the PhD program at the time of submission of this thesis and intends to continue his scholarship in this area.

Typeset using LaTeX.