A CASE STUDY IN REPRESENTING SCIENTIFIC APPLICATIONS (*GeoAc*) USING THE SPARSE POLYHEDRAL FRAMEWORK

by

Ravi Shankar

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

August 2021

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Ravi Shankar

Thesis Title: A Case Study in Representing Scientific Applications ($GeoAc$) using the Sparse Polyhedral Framework

Date of Final Oral Examination: 24th May 2021

The following individuals read and discussed the thesis submitted by student Ravi Shankar, and they evaluated the presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Catherine Olschanowsky, Ph.D.	Chair, Supervisory Committee
Dylan Mikesell, Ph.D.	Member, Supervisory Committee
Steven Cutchin, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Catherine Olschanowsky, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

Dedicated to my family.

ACKNOWLEDGMENTS

The author would like to express gratitude to his advisor Dr. Catherine Olschanowsky for her unwavering support, encouragement and belief in him. Her patience, advice and motivation over the years have been invaluable.

The author would like to thank his supervisory committee members Dr. Dylan Mikesell and Dr. Steven Cutchin for their feedback and support.

The author would also like to acknowledge the help and collaboration of current and former members of the ADaPT Data Flow Optimizations lab especially Aaron Orenstein, Anna Rift, Tobi Popoola and Dr. Eddie Davis.

This material is based upon work supported partially by the National Science Foundation under Grant Numbers 1849463 and 1563818. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. This research utilized the high-performance computing support of the R2 compute cluster (DOI: 10.18122/B2S41H) provided by Boise State University's Research Computing Department.

ABSTRACT

Tsunami detection and forecasting is a difficult problem that scientists are trying to tackle. Early path estimation and accurate prediction of the arrival time and size of a tsunami can save lives and help with impact assessment. Tsunami inducing earthquakes cause ground and sea-surface displacements that push up on the atmosphere. This atmospheric disturbance propagates upwards as an acoustic wave and eventually hits the ionosphere. *IonoSeis* is a software simulation package that leverages satellite-based ionospheric remote-sensing techniques to determine the epicenter of these earthquakes.

The execution time of the ray-tracing component of *IonoSeis* prevents its use as a real-time modeling tool. A proposed solution is to replace this component with a newer ray-tracing package developed by Los Alamos National Laboratory called *GeoAc* and parallelize it. This research is a case study that uses the sparse polyhedral framework (SPF) to represent the operational *GeoAc* code and thereby drive the requirements for a SPF optimization framework that is being actively developed.

TABLE OF CONTENTS

ABSTRACT vi

LIST OF TABLES ix

LIST OF FIGURES x

LIST OF ABBREVIATIONS xiii

LIST OF SYMBOLS xiv

1 Introduction 1

1.1 Problem Statement 3

1.2 Contributions 3

1.3 Organization 4

2 Background 5

2.1 *IonoSeis* 5

2.2 *GeoAc* 7

2.3 Memory Optimization 9

2.4 Optimization Frameworks 10

2.4.1 The Polyhedral Model 10

2.4.2 The Sparse Polyhedral Framework (SPF) 11

2.4.3 The Computation API 13

3 Engineering update of *GeoAc* 21
3.1 Parallelizing *GeoAc* 21
3.1.1 Performance Profiling 22
3.1.2 Identify Candidate Loops 24
3.1.3 Refactor to prepare for OpenMP 24
3.1.4 Implementation and Testing 26
3.1.5 Performance 27
3.2 Upgrading data formats to industry standards 28
3.2.1 The Network Standard Data Format (NetCDF) 29

4 Representing *GeoAc* using the SPF 31
4.1 Iterative Development 32
4.2 Function Inlining 33
4.3 Data-dependent Early Loop Exits 35
4.4 Rewrite code using the SPF 37

5 Related Work 43
5.1 Applying polyhedral optimizations to scientific applications 43

6 Conclusion 45

REFERENCES 46

A Experimental Setup 49

LIST OF TABLES

3.1 Performance results of parallelized *GeoAc* 27

LIST OF FIGURES

1.1 Satellite-based remote sensing techniques detect tsunami induced ionospheric disturbances. 2

2.1 Flow diagram for the *IonoSeis* modeling framework. Red ovals indicate model inputs, and blue boxes indicate individual steps that each have their own BASH script with input parameters for that step [16]. 6

2.2 An simplified loop nest from the *GeoAc* codebase. 10

2.3 The iteration space: statements of the loop-nest in Figure 2.2 can be viewed as lattice points on a polyhedron. 12

2.4 Dense Matrix Vector Multiply. 14

2.5 Sparse Matrix Vector Multiply. 14

2.6 Computation data space setup 15

2.7 Partial specification of a statement using the Computation API. 16

2.8 Data dependences between statements. 17

2.9 A statement's execution schedule. 17

2.10 Computation API specification for dense and sparse matrix vector multiply 19

2.11 Sparse matrix vector multiply C code generated by calling `codegen` on the Computation 20

3.1 *GeoAc's* dot graph. Functions where most time is spent are marked as saturated red, and functions where little time is spent are marked as dark blue. Functions where negligible or no time is spent do not appear in the graph by default. 23
3.2 Zoomed in sections of *GeoAc's* dot graph identifying potential performance bottlenecks 24
3.3 Candidate Loops. 25
3.4 Combining the global variables into a `struct`. 26
3.5 Test parameters for `GeoAc's` performance study. 27
3.6 `GeoAc's` average runtime measured over four runs for varying number of threads. Lower is better. 28
3.7 `GeoAc's` speedup measured over four runs for varying number of threads. Higher is better. 29
3.8 The structure of the resulting NetCDF file from *GeoAc* represented using CDL. 30

4.1 The `appendComputation` function. 34
4.2 A nested function call using `appendComputation`. 35
4.3 The original `Find_Segment` function in the `G2S_GlobalSpline1D.cpp` compilation unit 36
4.4 The modified `Find_Segment` function to take care of early exits and `return` statements 37
4.5 A statement in the `Find_Segment` Computation. 38
4.6 The data reads and writes of the `Find_Segment`. 39
4.7 The execution schedule of the `Find_Segment` Computation. 39

4.8 The complete `Find_Segment` Computation function 40
4.9 Code generated for a single call to the `c` function from the `GeoAc_UpdateSources` function results in the Computation expressed as macros. The macro `s12` on line 14 represents the `Find_Segment` function. 41
4.10 Calls to the macros defined in Figure 4.9. Line 13 represents the call to the `Find_Segment` function. 42

A.1 The `lscpu` command displays the CPU architecture information of R2. 50

LIST OF ABBREVIATIONS

CDL – Common Data Language

GNSS – Global Navigation Satellite Systems

IEGenLib – Inspector/Executor Generator Library

IR – Intermediate Representation

LANL – Los Alamos National Laboratory

NetCDF – Network Common Data Form

NUMA – Non-Uniform Memory Access

ODE – Ordinary Differential Equation

PDFG – Polyhedral Dataflow Graph

SAC – Seismic Analysis Code

SPF – Sparse Polyhedral Framework

WASP3D – Windy Atmospheric Sonic Propagation

LIST OF SYMBOLS

$\wedge$	Logical and
$\mid$	Set theory notation - such that
$\{\}$	Set theory notation - A set

CHAPTER 1

INTRODUCTION

Tsunami detection and forecasting is a difficult problem that scientists are trying to tackle. The unpredictability of tsunamis make it difficult for early warning and evacuation of at-risk coastlines. Early estimation and accurate prediction of the arrival time and size of a tsunami can save lives and help with impact assessment to buildings and infrastructure.

Analysis of satellite-based sensing data may help predict tsunami behaviour. Tsunami-inducing earthquakes cause ground and sea-surface displacements that push on the atmosphere, and propagate through the atmosphere into the ionosphere. Global Navigation Satellite Systems (GNSS) monitor ionospheric disturbances induced by such phenomena. Such satellite-based remote sensing techniques can be used to estimate the earth's surface deformation and predict the arrival time of a tsunami. *IonoSeis* [16] is a software simulation package that leverages these techniques to determine the epicenter of earthquakes that could cause these tsunamis.

There are a few key areas where *IonoSeis* can be improved. One of its key components is a three-dimensional ray-tracing package called the Windy Atmospheric Sonic Propagation (*WASP3D*). *WASP3D* models waves arriving in the ionosphere from the epicenter of an earthquake. This ray-tracing package is currently not fast enough to meet the needs of the workflow and inhibits real-time monitoring application. There

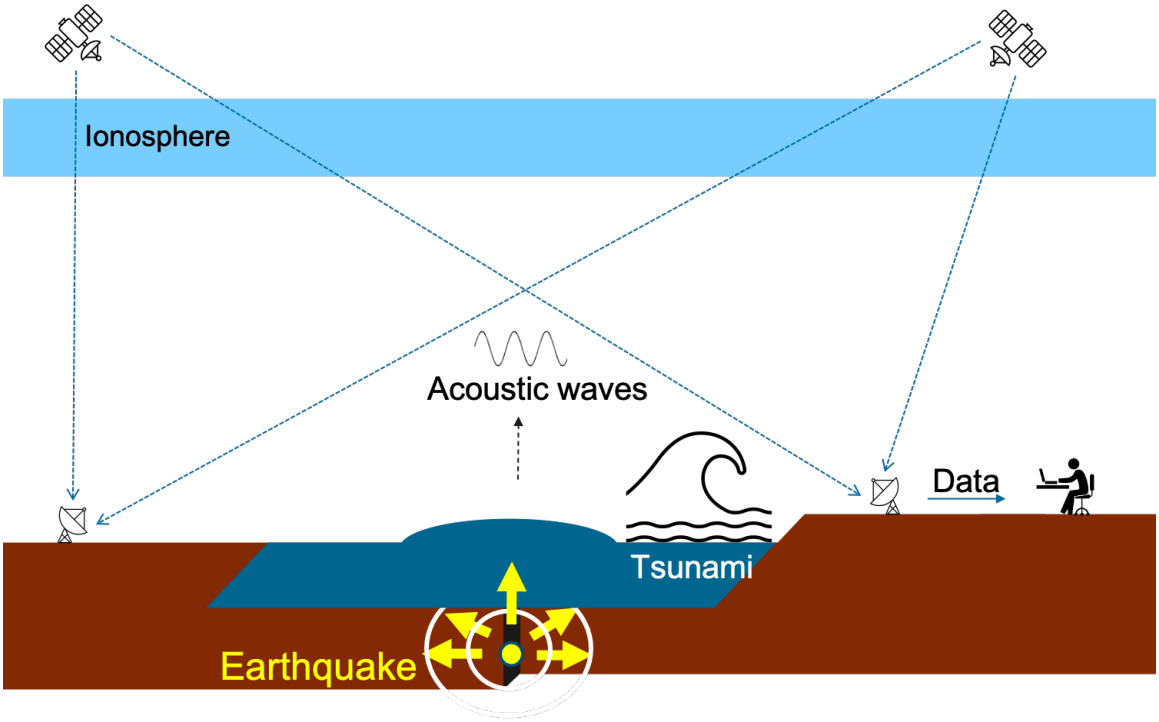

Figure 1.1: Satellite-based remote sensing techniques detect tsunami induced ionospheric disturbances.

are also no provisions to simulate many potential epicenters simultaneously. One of the objectives of this thesis replaces this component of *IonoSeis* with the *GeoAc* [2] modelling tool.

GeoAc is a newer ray-tracing package developed by Los Alamos National Laboratory (LANL) that more accurately models wave propagation phenomena compared to the *WASP3D*, the current ray-tracing model in *IonoSeis*. However, the execution time of *GeoAc* is also too long. Scientific applications like *GeoAc* are often memory-bound, requiring large amounts of memory, and spend a significant amount of execution time reading and writing data. This can significantly affect performance. Shared memory parallelization was used within *GeoAc* to meet the needs of *IonoSeis* workflow and potentially permit real-time monitoring of tsunami hazards.

Several transformation frameworks have been developed to optimize memory bound applications. However, they suffer from usability and expressivity limitations. One such framework is the sparse polyhedral framework. Tools like the SPF are used to express sparse computations and apply transformations automatically. This case study drives the development of a SPF optimization framework to enable the use of compiler transformations for performance optimization to be more expressive and therefore more usable.

1.1 Problem Statement

The execution time of the ray-tracing component of *IonoSeis* is not fast enough to meet the needs of the workflow and prevents its use as a real-time modelling tool. The proposed replacement to this ray-tracing component is memory-bound and spends a significant amount of time reading and writing data thereby affecting performance.

SPF tools suffer from a lack of expressiveness that captures the true behavior of scientific applications.

1.2 Contributions

This thesis presents a case-study that drives the development of a SPF optimization framework. The optimization framework enables the use of compiler transformations for performance optimization in other scientific applications.

WASP3D, the ray-tracing component of *IonoSeis* is replaced with an OpenMP implementation of LANL's *GeoAc* acoustic ray-tracing tool. The tool was customized to meet the requirements of the workflow and generates NetCDF data files required by subsequent steps in *IonoSeis*.

A performance study was conducted with varying threads on the OpenMP implementation of *GeoAc* comparing run-times and speed-ups with different thread counts.

1.3 Organization

This thesis is organized as six chapters with a varying number of sections. Chapter 2 provides an overview of the domain science software tools and background information on high performance computing concepts. Chapter 3 details the software engineering effort to parallelize *GeoAc* using OpenMP and customize it to meet the needs of the workflow. Chapter 4 covers the efforts to represent *GeoAc* using the SPF and thereby drive the optimization framework. Chapter 5 contains a review of related work. Chapter 6 concludes this thesis and provides a summary of this work.

CHAPTER 2

BACKGROUND

This chapter provides an overview of the domain science software tools that this research is built on, background information on high performance computing concepts, and the optimization framework that this case study drives. Section 2.1 provides an overview of the *IonoSeis* software simulation package and details how the ray-tracing component *WASP3D* does not meet the needs of the workflow and inhibits real-time monitoring application. Section 2.2 introduces the *GeoAc* software package that is used to replace *WASP3D* and is then optimized. Section 2.3 talks about the memory wall and how memory performance is a constraint in scientific computing. Section 2.4 explores the polyhedral and sparse polyhedral compiler optimization frameworks and introduces the Computation API [21].

2.1 *IonoSeis*

Many earthquakes cause sudden mass displacements at the earth's surface. When this type of earthquake occurs under the ocean, is of strong enough magnitude, and meets certain other criteria, a tsunami is generated. Ground or sea-surface displacements push on the atmosphere, which in turn generates an atmospheric disturbance. This disturbance propagates upward as an acoustic wave eventually inducing a local change in the electron density of the ionosphere. Global Navigation

Satellite Systems (GNSS) monitor ionospheric disturbances induced by such phenomena. Such satellite-based remote sensing methods are used to estimate the earth's surface deformation and predict the arrival time of a tsunami.

IonoSeis is a software package that combines multiple existing codebases into a single package to model GNSS-derived electron perturbations in the ionosphere due to the interaction of the neutral atmosphere and charged particles in the ionosphere. Running *IonoSeis* is a 5 step process as indicated in Figure 2.1. The Seismic Analysis

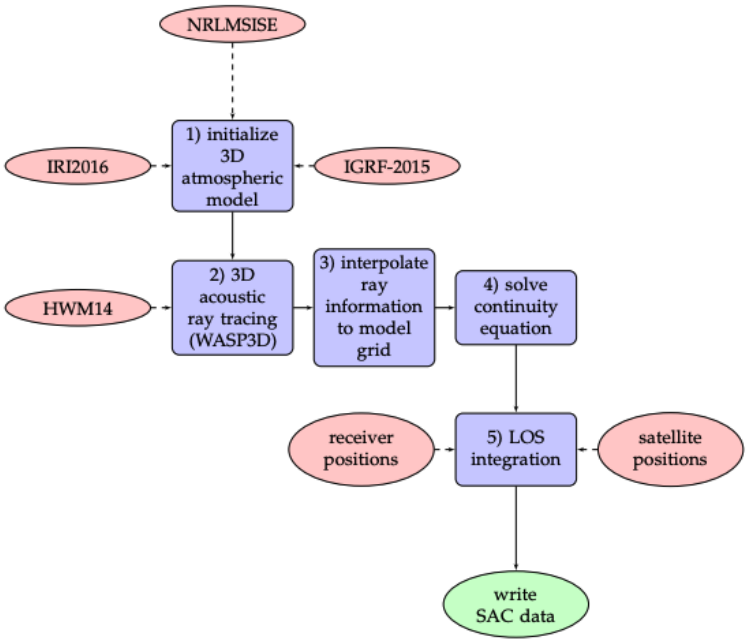

Figure 2.1: Flow diagram for the *IonoSeis* modeling framework. Red ovals indicate model inputs, and blue boxes indicate individual steps that each have their own BASH script with input parameters for that step [16].

Code (SAC) data format indicated by the green oval under step 5 Figure 2.1 is used to store the electron perturbation time series. Network Common Data Form (NetCDF) files are used to store grid information between each step.

WASP3D is one of the key components of *IonoSeis* and is a three-dimensional

ray-tracing software in spherical coordinates. It is, however, an older codebase and has certain limitations:

- The execution time is not fast enough to to meet the needs of *IonoSeis* and inhibits real-time monitoring application.
- There are no provisions to simulate many potential epicenters simultaneously. Currently only a single epicenter can be simulated at a time.
- *WASP3D* does not accurately represent the physics of wave propagation.
  - It does not account for viscous energy dissipation or wave amplification due to the rarefied atmosphere at high altitudes.
  - It cannot model ionospheric perturbations greater than 500-600 km until the sampling of the atmosphere by shooting rays at nonlinear intervals can be improved.
  - It does not model nonlinear waveform behavior.
  - It only models in spherical coordinates. While this is useful for *IonoSeis*, scientists would like to be able to extend the package so that national labs that study explosives and other local acoustic sources can use a Cartesian version of the code.

A preliminary contribution of this thesis replaces *WASP3D* with *GeoAc*

2.2 *GeoAc*

GeoAc [2] is a ray-tracing software tool developed at Los Alamos National Laboratory. The software is written in C++ and is used to model the propagation of

acoustic waves through the atmosphere using a *fourth-order Runge–Kutta (RK4)* method. *GeoAc* models propagation in an azimuthal plane and include methods to use a Cartesian coordinate system as well as a spherical coordinate system [2]. One of the contributions of this thesis is to explore the feasibility of using an optimized version of *GeoAc* to perform ionospheric ray-tracing and potentially run in real-time.

A performance analysis described in Section 3.1.1 indicates that the RK4 function is the most expensive operation in *GeoAc* and is therefore chosen for further analysis and optimization. Figure 3.2 shows the potential performance bottlenecks. Runge–Kutta methods are a family of iterative solutions commonly used to approximate solutions to initial value problems of ordinary differential equations (ODEs) [30, 1]. The fourth-order Runge-Kutta method is the most widely used method of solving ODEs. It is commonly used as a solver in many frameworks and libraries, providing a balance between computational cost and accuracy.

An ODE is a differential equation specifying the relationship between a dependent variable, y, and an independent variable, x as show in in Equation 2.1. The term 'ordinary' indicates that it has an ordinary derivative rather than a partial derivative.

$$\frac{dy}{dx} + xy = 0 \tag{2.1}$$

ODEs are used to model problems in engineering and various sciences that involve the change of some variable with respect to another. Most of these problems require the solution of an initial value problem. An initial value problem is an ODE together with an initial condition that specifies the value of the unknown function at some given point in the domain. *GeoAc* solves the equations governing acoustic propagation through the atmosphere in the geometric limit using a RK4 algorithm.

2.3 Memory Optimization

Memory is a key bottleneck in scientific computing. Scientific applications are memory-bound, often requiring large amounts of data from memory. As a result, most of its execution time is spent reading and writing data. Even though microprocessor and memory speed are improving exponentially, the rate of improvement of microprocessor speed far exceeds that of memory, creating what is referred to as the *memory wall* [29]. For a a few decades now, processor speed has increased by about 80% per year [14], as opposed to memory speed that has only improved at about 7% per year [11]. This increasing processor-memory performance gap results in the processor spending a significant amount of time, and as a result energy, waiting for data from the memory. As a result, the performance of scientific applications that are memory-bound are significantly impacted.

Memory optimizations include a range of techniques used to optimize memory usage that can then be used to improve the performance of scientific applications. Loops are a prime candidate for optimization since they may provide a great source for data parallelism. Loop fusion is a compiler optimization and transformation technique that joins two or more statements from separate loops into a single loop. Read-reduce fusion involves fusing loops that read from the same memory location and provides opportunities to reduce the number of times the same data is read. This results in better *data locality*. Data locality is the property where references to the same memory location or adjacent locations are reused within a short period of time [15]. Better data locality ensures that the most frequently accessed data can be retrieved quickly. Producer-consumer fusion involves merging loops where one loop writes a variable that is then read by the second loop thereby reducing the temporary storage

requirement [6].

2.4 Optimization Frameworks

This section provides an overview of the polyhedral and sparse polyhedral compiler optimization frameworks, and presents the Computation API, a SPF optimization framework that this case study drives.

2.4.1 The Polyhedral Model

The polyhedral model is a mathematical framework used to represent and manipulate loop nests of programs that perform large and complex operations in a compact form. The iterations of the loop-nest in Figure 2.2 can be represented as lattice points on a polyhedron as shown in Figure 2.3. The associated iteration space is represented graphically as a two-dimensional space `(i,j)` where each node in the graph represents an iteration. An iteration space that describes a loop nest is considered an affine space when the lower and upper bounds of each loop can be expressed as a linear function.

```
1 int phi_bounds = 3;
2 int theta_bounds = 3;
3 for(int i = 1; i <= phi_bounds; i++){
4     for(int j = 1; j <= theta_bounds; j++){
5         S1(i,j)
6     }
7 }
```

Figure 2.2: An simplified loop nest from the *GeoAc* codebase.

A loop nest can be represented with the following components:

- Iteration Domain: The set of executed instances of each statement of the loop. These sets are represented by affine inequalities.

- Relations: The set of reads, writes, and may-writes that relate statement instances in the iteration space to data locations.
- Dependences: The set of data dependences that impose restrictions on the execution order.
- Schedule: The overall execution order of each statement instance represented by a lexicographically ordered set of tuples in a multidimensional space [9].

A transformation in the polyhedral model is represented by a set of affine functions, one for each statement, called scheduling functions. An affine transformation is a linear mapping that preserves points, lines, and planes. Affine transformations can be applied to the polyhedron which is then converted into equivalent but optimized code. This conceptualization enables the compiler to reason about loop transformations as manipulating iteration spaces of loop-nests. A limitation to the polyhedral model is that the constraints within iteration spaces and transformations must be affine.

2.4.2 The Sparse Polyhedral Framework (SPF)

The Sparse Polyhedral Framework extends the polyhedral model by supporting non-affine iteration spaces and transforming irregular computations using *uninterpreted functions* [13]. Uninterpreted functions are *symbolic constants* that represent data structures such as the index arrays in sparse data formats. Symbolic constants are constant values that do not change during the course of a computation. The SPF represents computations with indirect memory accesses and *run-time reordering transformations* with integer tuple sets and relations with affine constraints and constraints involving uninterpreted function symbols [26, 27]. Run-time data reordering

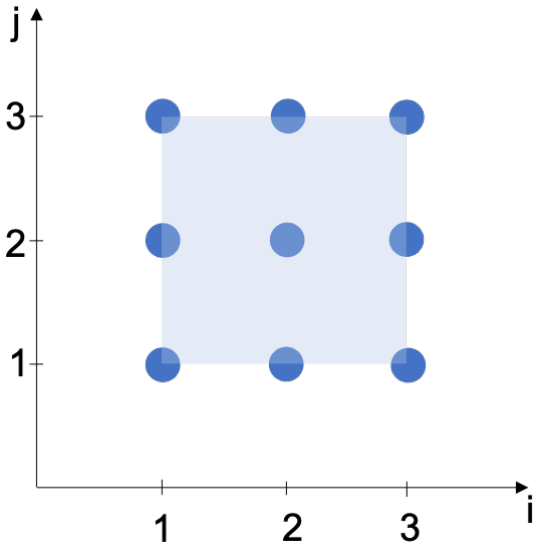

Figure 2.3: The iteration space: statements of the loop-nest in Figure 2.2 can be viewed as lattice points on a polyhedron.

techniques attempt to improve the spatial and temporal data locality in a loop by reordering the data based on the order in which it was referenced in the loop [25].

Sets and Relations in the SPF

At the core of the SPF are sets and relations. Data and iteration spaces are represented with sets, while access functions, data dependences, and transformations are represented with relations [26].

Sets are specified as shown in Equation 2.2, where each x is an integer tuple and each c is a constraint. The arity of the set is m, the dimensions of the tuples.

$$s = \{[x_1, ..., x_m] \mid c_1 \wedge ... \wedge c_n\} \tag{2.2}$$

The constraints in a set can be either equalities or inequalities which are expressions

containing symbolic constants or uninterpreted functions [26].

A relation is a mapping of sets of input integer tuples to output integer tuples. Relations are specified as shown in Equation 2.3.

$$s = \{[x_1, ..., x_m] \rightarrow [y_1, ..., y_n] \mid c_1 \wedge ... \wedge c_z\} \tag{2.3}$$

Each x is an input tuple variable, each y is an output tuple variable and c is a constraint.

Equation 2.4 represents the iteration domain of the dense matrix vector multiply code show in in Figure 2.4. i and j are the iterators in the 2 dimensions of the set and $0 <= i < N$ and $0 <= j < M$ are the affine inequalities or constraints that bind the iteration space.

$$I = \{[i, j] \mid 0 <= i < N \wedge 0 <= j < M\} \tag{2.4}$$

2.4.3 The Computation API

This section describes the Computation API from a workshop paper that has been accepted [21] for publication. This case study contributed to the design and testing of the Computation API. The Computation API is an object-oriented API that provides a precise specification of how to combine the individual components of the SPF to create an intermediate representation (IR). An IR is a data structure used internally by the compiler to represent source code. This IR can directly produce polyhedral dataflow graphs (PDFGs) [6] and translates the graph operations defined for PDFGs into relations used by the *Inspector/Executor Generator Library (IEGenLib)* [26] to perform transformations.

IEGenLib is a C++ library with data structures and routines that can represent, parse, and visit integer tuple sets and relations with affine constraints and uninterpreted function symbol equality constraints [26]. The Computation API is implemented as a C++ class in *IEGenLib* [26] and contains all of the components required to express a Computation or a series of Computations. Dense and sparse matrix vector multiplication, shown in Figure 2.4 and Figure 2.5 respectively, are used as examples to represent the computations in the SPF. The components of a Computation are: data spaces, statements, data dependences, and execution schedules. The proceeding subsections describe the design and interface for each of these components.

Dense vector multiply

```
1 for (i = 0; i < N; i++) {
2     for (j=0; j<M; j++) {
3         y[i] += A[i][j] * x[j];
4     }
5 }
```

Figure 2.4: Dense Matrix Vector Multiply.

CSR Sparse matrix vector multiply

```
1 for (i = 0; i < N; i++) {
2     for (k=rowptr[i]; k<rowptr[i+1]; k++) {
3         j = col[k];
4         y[i] += A[k] * x[j];
5     }
6 }
```

Figure 2.5: Sparse Matrix Vector Multiply.

Data spaces

A data space represents a collection of conceptually unique memory addresses. Each combination of data space name and input tuple is guaranteed to map to a unique space in memory for the lifetime of the data space. The data spaces represented in the matrix vector multiply examples shown in Figures 2.4 and 2.5 are `y`, `A`, and `x`. In the sparse version the index arrays `rowptr` and `col` are also considered data spaces.

```
1 // Dense matrix vector multiply
2 Computation* denseComp = new Computation();
3 denseComp->addDataSpace("y");
4 denseComp->addDataSpace("A");
5 denseComp->addDataSpace("x");
6
7 // Sparse matrix vector multiply
8 Computation* sparseComp = new Computation();
9 sparseComp->addDataSpace("y");
10 sparseComp->addDataSpace("A");
11 sparseComp->addDataSpace("x");
```

Figure 2.6: Computation data space setup

Statements

Statements perform read and write operations on data spaces. Each statement has an iteration space associated with it. This iteration space is a set containing every instance of the statement and has no particular order. A statement is expressed as the set of iterators that it runs over, subject to the constraints of its iteration (loop bounds).

A statement is written as a string and the names of the data spaces are delimited with a $ symbol (see lines 2 and 5 in Figure 2.7). The iteration space is specified as a

set using the *IEGenLib* syntax, with the exception of delimiting all data spaces with a $ symbol (see lines 3 and 6 in Figure 2.7).

```
1 Stmt* denseS0 = new Stmt(
2   "$y$(i) += $A$(i,j) * $x$(j);",
3   "{[i,j]: 0 <= i < N && 0 <= j < M}", ...
4
5 Stmt* sparseS0 = new Stmt(
6   "$y$(i) += $A$(k) * $x$(j)",
7   "{[i,k,j]: 0 <= i < N && rowptr(i) <= k < rowptr(i+1) && j = $col(k)}", ...
```

Figure 2.7: Partial specification of a statement using the Computation API.

Data dependences

Data dependences exist between statements. They are encoded using relations between iteration vectors and data space vectors. In the matrix vector multiply examples, the data reads and writes are specified as shown in Figure 2.8.

Execution schedules

Execution schedules are determined using scheduling functions that are required to respect data dependence relations. Scheduling functions take as input the iterators that apply to the current statement, if any, and output the schedule as an integer tuple that may be lexicographically ordered with others to determine correct execution order of a group of statements. Iterators, if any, are used as part of the output tuple, indicating that the value of iterators affects the ordering of the statement. For example, in the scheduling function $\{[i,j] \rightarrow [0,i,0,j,0]\}$, the position of i before j signifies that the corresponding statement is within a loop over j, which in turn is within a loop over i.

```
/* 4th and 5th parameters to Stmt constructor */
// Dense matrix vector multiply
...
{ // reads
  {"y", "{[i,j]->[i]}"},
  {"A", "{[i,j]->[i,j]}"},
  {"x", "{[i,j]->[j]}"}
},
{ // writes
  {"y", "{[i,j]->[i]}"}
}

// Sparse matrix vector multiply
...
{ // reads
  {"y", "{[i,k,j]->[i]}"},
  {"A", "{[i,k,j]->[k]}"},
  {"x", "{[i,k,j]->[j]}"}
},
{ // writes
  {"y", "{[i,k,j]->[i]}"}
}
```

Figure 2.8: Data dependences between statements.

Figure 2.10 shows the complete specification of the dense matrix vector multiply followed by the sparse matrix vector multiply.

Code Generation

Polyhedra scanning is used to generate optimized code. The Computation class interfaces with CodeGen+ [4] for code generation. CodeGen+ uses Omega sets and

```
/* 3rd parameter to the Stmt constructor */
// Dense matrix vector multiply
 "{[i,j] ->[0,i,0,j,0]}"

// Sparse matrix vector multiply
"{[i,k,j]->[0,i,0,k,0,j,0]}"
```

Figure 2.9: A statement's execution schedule.

relations for polyhedra scanning. Omega [12] is a set of C++ classes for manipulating integer tuple relations and sets. Omega sets and relations have limitations in the presence of uninterpreted functions. Uninterpreted functions are limited by the prefix rule whereby they must be a prefix of the tuple declaration. Uninterpreted functions cannot have expressions as parameters. Code generation overcomes this limitation by modifying uninterpreted functions in IEGenLib to be Omega compliant.

Figure 2.11 shows the results of code generation for the sparse matrix vector multiplication Computation defined in Figure 2.10. Line 2 of Figure 2.11 defines a macro for the statement `s0`, lines 7 - 9 remap the Omega compliant uninterpreted function back to its original, and lines 11 - 16 are a direct result of polyhedra scanning from CodeGen+. The Computation implementation provides all of the supporting definitions for fully functional code.

Full Computation Specification

```
// Dense matrix vector multiply
Computation* denseComp = new Computation();
denseComp->addDataSpace("y");
denseComp->addDataSpace("A");
denseComp->addDataSpace("x");
Stmt* denseS0 = new Stmt(
  // source code
  "$y$(i) += $A$(i,j) * $x$(j);",
  // iter domain
  "{[i,j]: 0 <= i < $N$ && 0 <= j < $M$}",
  // scheduling function
  "{[i,j] ->[0,i,0,j,0]}",
  { // data reads
    {"y", "{[i,j]->[i]}"},
    {"A", "{[i,j]->[i,j]}"},
    {"x", "{[i,j]->[j]}"}
  },
  { // data writes
    {"y", "{[i,j]->[i]}"}
  }
);
denseComp->addStmt(denseS0);

// Sparse matrix vector multiply
Computation* sparseComp = new Computation();
sparseComp->addDataSpace("y");
sparseComp->addDataSpace("A");
sparseComp->addDataSpace("x");
Stmt* sparseS0 = new Stmt(
  "$y$(i) += $A$(k) * $x$(j)",
  "{[i,k,j]: 0 <= i < N && rowptr(i) <= k < rowptr(i+1) && j = col(k)}",
  "{[i,k,j]->[0,i,0,k,0,j,0]}",
  {
    {"y", "{[i,k,j]->[i]}"},
    {"A", "{[i,k,j]->[k]}"},
    {"x", "{[i,k,j]->[j]}"}
  },
  {
    {"y", "{[i,k,j]->[i]}"}
  }
);
sparseComp->addStmt(sparseS0);
```

Figure 2.10: Computation API specification for dense and sparse matrix vector multiply

SPMV codegen

```
#undef s0
#define s0(__x0, i, __x2, k, __x4, j, __x6) y(i) += A(k) * x(j)

#undef col(t0)
#undef col_0(__tv0, __tv1, __tv2, __tv3)
#undef rowptr(t0)
#undef rowptr_1(__tv0, __tv1)
#undef rowptr_2(__tv0, __tv1)
#define col(t0) col[t0]
#define col_0(__tv0, __tv1, __tv2, __tv3) col(__tv3)
#define rowptr(t0) rowptr[t0]
#define rowptr_1(__tv0, __tv1) rowptr(__tv1)
#define rowptr_2(__tv0, __tv1) rowptr(__tv1 + 1)

for(t2 = 0; t2 <= N-1; t2++) {
  for(t4 = rowptr_1(t1,t2); t4 <= rowptr_2(t1,t2)-1; t4++) {
    t6=col_0(t1,t2,t3,t4);
    s0(0,t2,0,t4,0,t6,0);
  }
}

#undef s0
#undef col(t0)
#undef col_0(__tv0, __tv1, __tv2, __tv3)
#undef rowptr(t0)
#undef rowptr_1(__tv0, __tv1)
#undef rowptr_2(__tv0, __tv1)
```

Figure 2.11: Sparse matrix vector multiply C code generated by calling `codegen` on the Computation

CHAPTER 3

ENGINEERING UPDATE OF *GEOAC*

This chapter provides details on the software engineering updates performed on *GeoAc*. The contributions described in this chapter include section 3.1 which describes the various steps culminating in the shared memory parallelization *GeoAc*, performance results as a result of the parallelization in section 3.1.5, and section 3.2, which details upgrading data formats from a generic data file to industry standard NetCDF format.

3.1 Parallelizing *GeoAc*

Scientific applications typically run on large clusters/supercomputers. Power is an important design constraint of these machines. The longer an application runs the more power it consumes. A direct result is a requirement for expensive cooling solutions to keep the hardware running optimally. Scientific applications that are designed to take advantage of parallelism run faster and as a result save time, energy and money.

While *GeoAc* improved the physics of the generated models, it still wasn't fast enough to be used as a real-time monitoring tool. A serial run on *GeoAc* with the input parameters as specified in Figure 3.5 took approximately 19 hours on Boise

State's R2 compute cluster. The following sections describe the effort taken to parallelize *GeoAc* using OpenMP [5].

3.1.1 Performance Profiling

Profiling *GeoAc* helped identify which parts of the application were expensive, and could be good candidates for a code rewrite to potentially make the application run in parallel. Profiling tools record what fraction of a program's run-time is spent in various portions of the code. The GNU *gprof* [8] utility is one such tool that reports this data at function-level granularity. *GeoAc* was profiled using *gprof* to generate the call graph seen in Figure 3.1. Each node in the graph represents a function. The first number is the percent of cumulative time spent in the function or its subroutines against the total runtime of the application. The second number, listed in parentheses, is the percentage of time spent directly in the function. The third number is the total number of calls to the function.

An open source python script *gprof2dot* [7] was used to convert the call graph into a dot graph. Graphviz's dot renderer was then used to generate an image of the dot graph. Examination of the dot graph helped us gain an overview of the relationships between the different functions in *GeoAc*.

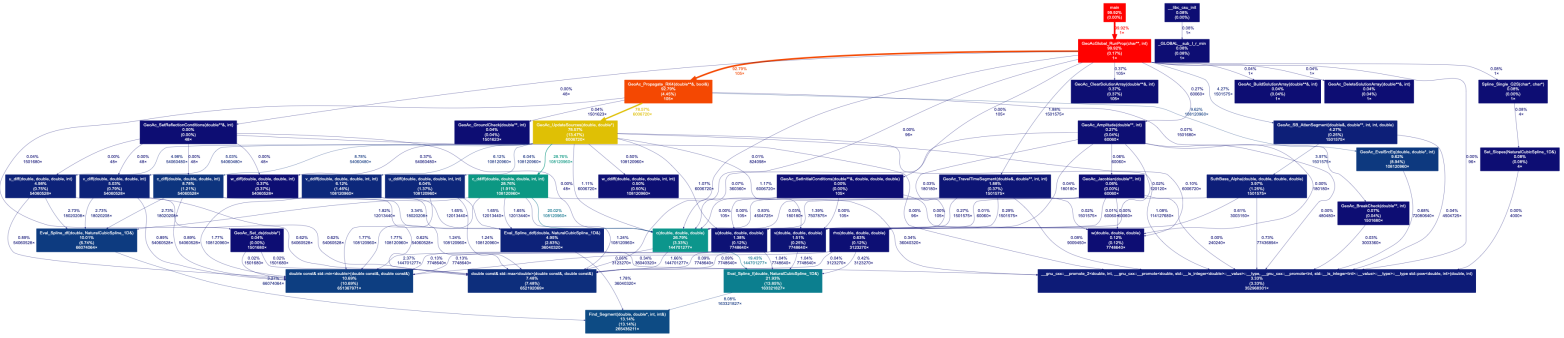

Figure 3.1: *GeoAc's* dot graph. Functions where most time is spent are marked as saturated red, and functions where little time is spent are marked as dark blue. Functions where negligible or no time is spent do not appear in the graph by default.

From the dot graph, we were able to identify that a majority of the time spent by the application was in the `GeoAcGlobal_RunProp` function. This function in turn called the `GeoAc_Propagate_RK4` function.

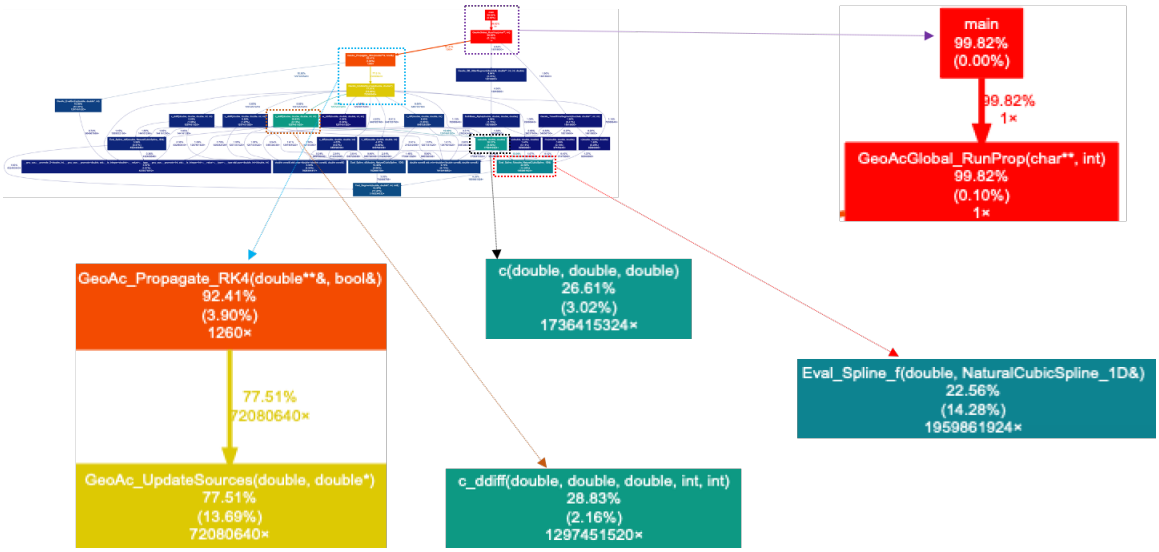

Figure 3.2: Zoomed in sections of *GeoAc's* dot graph identifying potential performance bottlenecks

3.1.2 Identify Candidate Loops

Profiling *GeoAc* also helped us identify important loops that could be candidates for optimization. A manual analysis of the code revealed the legality of parallelizing expensive sections of the code. Figure 3.3 shows a loop nest within the `GeoAcGlobal_RunProp` function. This seemed to be an ideal target for optimization.

3.1.3 Refactor to prepare for OpenMP

A manual dataflow analysis was conducted to determine reads and writes within parallel sections. Any variable written to in the parallel section must be thread local. All global variables that were not constant were refactored and made local so

```
1 ...
2 for(double phi = phi_min; phi <= phi_max; phi+=phi_step){
3   for(double theta = theta_min; theta <= theta_max; theta+=theta_step){
4     ...
5     ...
6     for(int bnc_cnt = 0; bnc_cnt <= bounces; bnc_cnt++){
7       k = GeoAc_Propagate_RK4(solution, BreakCheck);
8       if(WriteRays || WriteCaustics){
9         if(WriteCaustics){
10          ...
11        }
12        // write profiles to data files and vector arrays
13        for(int m = 1; m < k ; m++){
14          ...
15        }
16        ...
17      }
18      ...
19    }
20    ...
21  }
22  ...
23 }
```

Figure 3.3: Candidate Loops.

that every thread has a local copy of the variable. After the dataflow analysis, all non-constant global variables were re-declared locally and added as parameters to functions where the globals were being used. In some functions, global `structs` were being used. Since each thread requires a local copy of the `struct`, it was redefined and moved to a header file so that local instances could be called by the driver. In one particular function, a new struct called `SplineStruct` was created as shown in Figure 3.4, that held all the global variables the function used in order to conveniently pass the required variable around.

```
1 // Struct that contain variables that were previously global
2 // Each thread gets its own copy
3 struct SplineStruct{
4   //----------------------------------------//
5   //------Parameters for Interpolation------//
6   //----------------------------------------//
7   int r_cnt; // Number of vertical points
8   int accel = 0; // Acceleration index
9   double* r_vals; // r_k elements (r_cnt length)
10  double* T_vals; // Temperature at r_k
11  double* u_vals; // E-W winds at r_k
12  double* v_vals; // N-S winds at r_k
13  double* rho_vals; // Density at r_k
14  //-----------------------------------------------------//
15  ////-------------Combined Function to Input-------------//
16  ////------G2S Files and Generate the Interpolation------//
17  ////-----------------------------------------------------//
18  struct NaturalCubicSpline_1D Temp_Spline;
19  struct NaturalCubicSpline_1D Windu_Spline;
20  struct NaturalCubicSpline_1D Windv_Spline;
21  struct NaturalCubicSpline_1D Density_Spline;
22 };
```

Figure 3.4: Combining the global variables into a `struct`.

3.1.4 Implementation and Testing

After refactoring the code, OpenMP was used to achieve shared memory parallelization. OpenMP consists of a set of compiler directives (`#pragmas`) and library routines that provides support for parallel programming in shared-memory environments. These directives are designed such that even if the compiler does not support them, the program will still execute, but without any parallelism.

The `'#pragma omp for'` directive was applied to the nested loop identified in section 3.1.2. The loop construct `'for'` specifies that the iterations of loops will be distributed among and executed by the encountering team of threads. By default when OpenMP encounters the `'#pragma omp for'` directive, it creates as many threads as there are cores in the system.

3.1.5 Performance

Correctness was ensured by running the application successfully using the following test parameters in Figure 3.5 without any errors or warnings.

```
0 <= theta <= 90 ; theta_step=0.1
0 <= phi <= 90 ; phi_step=1
bounces=0
rng_max=5000
```

Figure 3.5: Test parameters for `GeoAc's` performance study.

Multiple runs of the application were conducted with varying number of threads and a preliminary performance analysis was conducted.

Table 3.1: Performance results of parallelized *GeoAc*

GeoAc performance results						
Threads	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Avg. (s)	Speedup
1	68552	69043	69401	69424	69105	
4	18712	18924	18642	19060	18835	3
8	9917	9909	9951	9961	9934	6
16	5078	5076	5087	5081	5080	13
28	3032	3016	3026	3023	3024	22

Figure 3.6 shows the average runtime measured over four runs with 1, 4, 8, 16 and 28 threads. For this particular set of input parameters, 28 threads was the clear winner with an average runtime of about 50 minutes as opposed to 19 hours with the serial version. The 28 thread run resulted in a 22x speedup over the serial version of the code as shown in Figure 3.7.

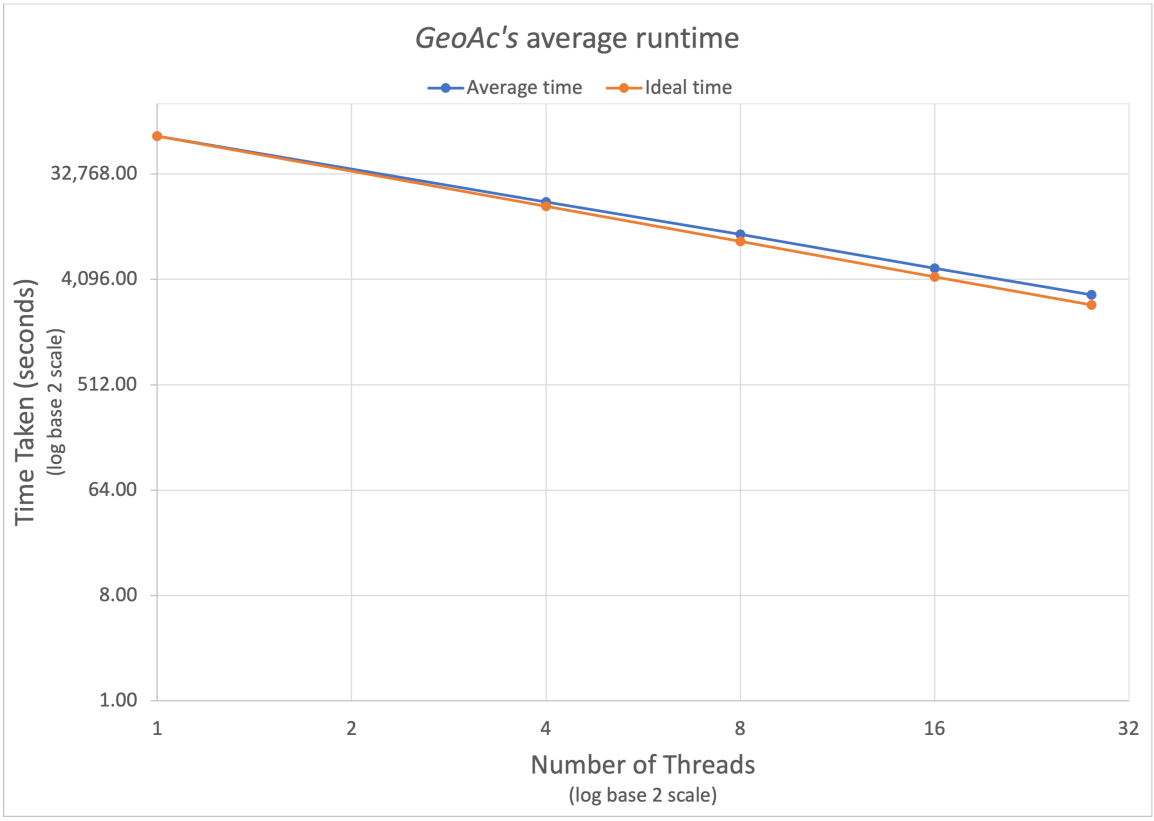

Figure 3.6: `GeoAc's` average runtime measured over four runs for varying number of threads. Lower is better.

3.2 Upgrading data formats to industry standards

GeoAc outputs data in generic DAT files. One of the problems with this data type is that when *GeoAc* is run with larger parameter sets, the output files can grow significantly large. Another problem that was discussed earlier in section 2.1, is that the NetCDF format is used to store grid information between each step of *IonoSeis*. At the request of the domain scientist and in the interest of conforming to *IonoSeis* and industry standards, the code was modified to output NetCDF files rather than generic DAT files.

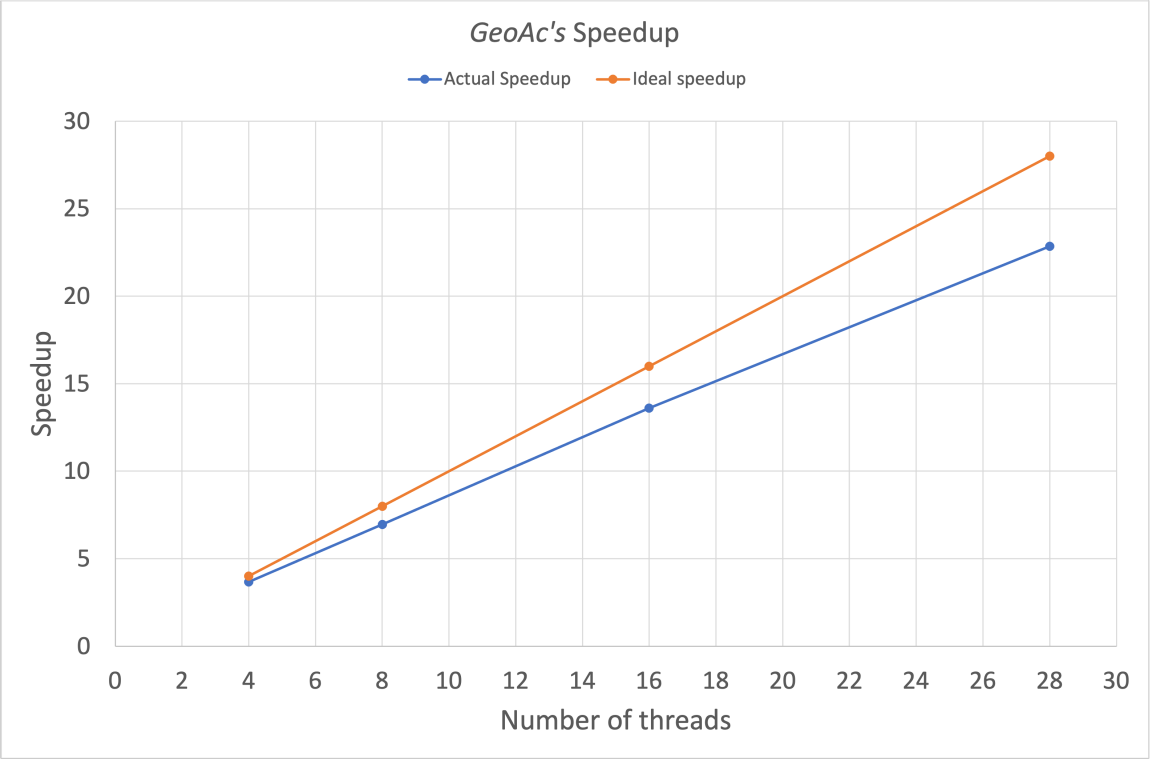

Figure 3.7: `GeoAc's` speedup measured over four runs for varying number of threads. Higher is better.

3.2.1 The Network Standard Data Format (NetCDF)

NetCDF is a machine-independent data abstraction commonly used for storing and retrieving multi-dimensional scientific data [24]. A NetCDF file contains dimensions, variables, and attributes. These components are used together to describe data fields in a multi-dimensional dataset. A dimension has a name and size and can be used to represent a physical dimension like the number of rays and the maximum points along a ray. Variables have a datatype, name, and dimension and store the bulk of the data. Attributes are used to provide more information about a particular variable or the NetCDF file in general. Figure 3.8 shows the structure of a NetCDF file generated with *GeoAc* using the CDL (Common Data Language) notation. CDL

is a human-readable notation for NetCDF data.

```
netcdf GeoAcResultsAndRaypaths {
    dimensions:
      nRays = 756 ;
      maxPointsAlongRay = 600 ;
    variables:
      double theta(nRays) ;
      double wasp_takeoff_angle(nRays) ;
      int n_b(nRays) ;
      double lat_0_deg(nRays) ;
      double lon_0_deg(nRays) ;
      double TravelTime_s(nRays) ;
      double Celerity_kmps(nRays) ;
      double TurningHeight_km(nRays) ;
      double Inclination_deg(nRays) ;
      double BackAzimuth_deg(nRays) ;
      double GeoAtten_dB(nRays) ;
      double AtmoAtten_dB(nRays) ;
      double wasp_altitude_km(nRays, maxPointsAlongRay) ;
      double wasp_colatitude_deg(nRays, maxPointsAlongRay) ;
      double wasp_longitude_deg(nRays, maxPointsAlongRay) ;
      double wasp_kr_deg(nRays, maxPointsAlongRay) ;
      double wasp_kt_deg(nRays, maxPointsAlongRay) ;
      double wasp_kf_deg(nRays, maxPointsAlongRay) ;
      double wasp_amplitude_dB(nRays, maxPointsAlongRay) ;
      double wasp_Veff_dB(nRays, maxPointsAlongRay) ;
      double wasp_arrival_time_s(nRays, maxPointsAlongRay) ;
}
```

Figure 3.8: The structure of the resulting NetCDF file from *GeoAc* represented using CDL.

NetCDF is not thread-safe. As a result, it was not possible to parallelize the file i/o without using locks or some sort of mutual exclusion which would have significantly increased the complexity of the implementation. The parallel implementation wrote data files per thread and combined them at the end of the parallel section. Instead of combining each of the thread's data files, a better option was to gather the data from each thread and write it directly to a NetCDF file.

CHAPTER 4

REPRESENTING *GEOAC* USING THE SPF

Based on the profile and a manual dataflow analysis of the code, portions of *GeoAc* that could potentially be expressed using the sparse polyhedral framework were identified. The `GeoAc_Propagate_RK4_2` function was deemed the most suitable. Data and iteration spaces are represented using sets while access functions, data dependences, and transformations are represented using relations. This chapter discusses the development of the optimization framework (Computation API) by incrementally expressing `GeoAc` code using the SPF and tackling limitations as they arise.

From `GeoAc's` dot graph in Figure 3.2, it is evident that the `GeoAc_RunProp`, `GeoAc_Propogate_RK4` and `GeoAc_UpdateSources` functions have the greatest percent of cumulative time spent in them. The core of the acoustic ray-tracing of GeoAc is performed by an RK4 algorithm, so the `GeoAc_Propogate_RK4` was deemed most appropriate to represent using the SPF. The strategy of representing the operational code with the SPF involved identifying the leaf function called from the `GeoAc_Propogate_RK4` function and representing it using the Computation API. We work our way upwards from there.

To determine the leaf function, a manual dataflow analysis was conducted to identify all the function calls. Each function was traversed iteratively until the leaf func-

tion is reached. The first call in `GeoAc_Propogate_RK4` is to the `GeoAc_UpdateSources` function. The `GeoAc_UpdateSources` function calls the `c` function which in turn calls the `Eval_Spline_f` function. The `Eval_Spline_f` function then calls the `Find_Segment` function which was determined to be the leaf function from a call to the `GeoAc_Propogate_RK4`. It should be noted that the `GeoAc_Propogate_RK4` function, the `GeoAc_UpdateSources` and the remaining functions were all in different compilation units.

After the dataflow analysis, the `Find_Segment` function is expressed using the sparse polyhedral framework. This is done using function inlining with the Computation API described in Section 2.4.3.

4.1 Iterative Development

Iterative transformations are a key goal of the Computation API. Contributions of this thesis evaluate how well the iterative transformations work. We breakdown the SPF representation process into small iterative pieces that are tested with the operational `GeoAc` code. We start by representing the leaf function `Find_Segment` as a reusable function that returns a Computation which is outlined in Section 4.2. Code generation is called on the Computation with a driver developed in IEGenLib. This results in a C file with compiler generated statement macros. This file is dropped into the operational `GeoAc` code in place of the `Find_Segment` function as a preprocessor directive (`# include`). `GeoAc` is then run on a test case with known results and the new results are compared for correctness.

4.2 Function Inlining

Functions are useful in breaking down programming solutions to smaller reusable modules and help with code structure. They, however, present challenges with interprocedural analysis. Function inlining is an optimization that copies code from the function definition directly into the calling function's code. This eliminates the need for a separate set of instructions in memory and exposes optimization opportunities. One of the goals of the Computation API is to eliminate the interprocedural analysis challenge.

The `c`, `v` and `u` functions within `GeoAc_UpdateSources` each called two other functions - `Eval_Spline_f` and `Find_Segment`. Furthermore, since the parent function `GeoAc_UpdateSources` is within a for loop, each of these functions end up being called multiple times. To enable code reuse, functions that return a `Computation*` were created in the optimization driver wherein each function in the driver corresponds to the SPF representation of the function we were optimizing in `GeoAc`. The `Computation*` returning functions are only ever called once. Subsequent calls to a function involved appending the same Computation multiple times. Figure 4.8 shows the leaf function `Find_Segment_Computation`.

Nested function calls resulted in the development of the `appendComputation` function shown in Figure 4.1. This function take as its parameters the Computation to be appended (appendee), the iteration domain of the appendee context, the execution schedule of the appendee context and a vector of strings containing arguments to Computation to be appended. The return valued of this function is a `AppendComputationResult` struct that consists of a vector of strings that holds the return value (name of a dataspace or a literal) of the function that is being appended

and an integer that is the depth of the last statement in the appendee.

```
1 AppendComputationResult appendComputation(
2         const Computation* other,
3         std::string surroundingIterDomainStr,
4         std::string surroundingExecScheduleStr,
5         const std::vector<std::string>& arguments = {})
```

Figure 4.1: The `appendComputation` function.

One of the problems with inlining functions across multiple compilation units was name collisions between the caller and callee functions. The `addParameter` function takes as its parameters a pair of strings - the name of the argument and its datatype. Data space renaming prevents name conflicts between the caller and callee as well as multiple invocations of the callee within the same context. This is taken care of by the `appendComputation` function under the hood. The argument is checked for whether it is a data space in the caller, and an exception is thrown if it is not. Literals were initially not supported. The caller is required to assign literals to temporary variables and then pass those variables in as arguments. If a check on the argument indicates that it is not a data space, it is considered a literal and evaluated. Expressions in function calls are not supported. The caller will need to assign the expression to a temporary data space.

Since inlining copies code from the function definition directly into the calling function's code, return statements are invalid. The `appendComputation` function supports return values by returning a `struct` which consists of the depth of the last appended statement and a vector of strings that hold the return value (name of a dataspace or a literal). It is the caller's responsibility to keep track of whether the return type is a literal or data space.

Our use case necessitated function calls nested within loops. Functions being called within loops of the calling function is a fairly common pattern in scientific applications. In this situation the inlining procedure must adjust the iteration spaces and scheduling functions of the callee to appear within the loop structures. By copying data from previous computations up to the given depth of the surrounding execution schedule, the inlining procedure was able to support function calls within loops. Figure 4.2 shows a nested function call. The third parameter of the `appendComputation` function on line 13 indicates that a call to `aComputation` is the first statement in an `i` loop (nested function call).

```
1 //The current computation (appender)
2 Computation* temp = new Computation();
3
4 //The computation to be appended (appendee)
5 Computation* comp = aComputation();
6
7 //The return datatype of the appendComputation function is a struct
8 AppendComputationResult aCompRes;
9 aCompRes = temp->appendComputation(comp, "{[0]}", "{[0]->[1, i, 1]}", args);
```

Figure 4.2: A nested function call using `appendComputation`.

4.3 Data-dependent Early Loop Exits

The `Find_Segment` function shown in Figure 4.3 is used to find the segment index as well as evaluate the 1D vertical spline. From the code, and by examining results of debug print statements, it was observed that the function was originally written with a loop optimization. The start, middle and end of the `x_vals` array were checked at the start of the function to quickly calculate the segment index without having to iterate through the entire array. There were multiple data-dependent early loop exits in the code.

The original `Find_Segment` function

```
//-----------------------------------------------//
//------Functions to Find the Segment Index------//
//-----and Evaluate the 1D Vertical Splines------//
//-----------------------------------------------//
int Find_Segment(double x, double* x_vals, int length, int & prev){
    if(x > x_vals[length-1] || x < x_vals[0]){
        cout << "Cannot interpolate outside of given bounds. x = " << x << "
            is invalid." << '\n';
    } else {

        // Check previous index and bounding segments
        if(x >= x_vals[prev] && x <= x_vals[prev+1]){
            done = true;
        }

        if(!done && prev+2 <= length-1){
            if(x >= x_vals[prev+1] && x <= x_vals[prev+2]){
                done = true;
                prev = prev + 1;
            }
        }
        if(!done && prev-1 >= 0){
            if(x >= x_vals[prev-1] && x <= x_vals[prev]){
                done = true;
                prev = prev - 1;
            }
        }

        if(!done){
            for(int i = 0; i < length; i++){
                if(x >= x_vals[i] && x <= x_vals[i+1]){
                    index = i;
                    break;
                }
                if (x >= x_vals[length - 2 - i] && x < x_vals[length - 1 - i])
                    {
                    index = (length - 2) - i;
                    break;
                }
            }
            prev = index;
        }
        return prev;
    }
}
```

Figure 4.3: The original `Find_Segment` function in the `G2S_GlobalSpline1D.cpp` compilation unit

Data dependent loops have data dependencies, making parallelization difficult without extensive refactoring. With the function now inlined, data-dependent early exits would cause an early exit in the inlined code resulting in unforeseen errors. As a result of this finding, research is ongoing to add data-dependent early exits and data-dependent control flow in general to the SPF. The Computation API does not at present support data-dependent early exits and as a result, the function was refactored as shown in Figure 4.4.

The modified `Find_Segment` function

```
1 int Find_Segment(double x, double* x_vals, int length, int & prev){
2     for(int i = 0; i < length; i++){
3         if(x >= x_vals[i] && x <= x_vals[i+1]){
4             prev = i;
5         }
6     }
7 }
```

Figure 4.4: The modified `Find_Segment` function to take care of early exits and `return` statements

Once the `Find_Segment` function was represented in the SPF, we move up to the next one which is the `Eval_Spline_f` function. We repeat this process until the code in the function we were originally trying to represent in the SPF is replaced with code generated using the Computation API.

4.4 Rewrite code using the SPF

This section presents operational *GeoAc* code represented in the SPF using the Computation API. The Computation API is used to represent the modified `Find_Segment` function in Figure 4.5. The first parameter is the source code within the `i` loop and

is written as a string on lines 1 and 2 of Figure 4.5. The names of the dataspaces in the source code are delimited with $ symbols.

```
1 Stmt* s0 = new Stmt("if($x$ >= $x_vals$[i] &&
2                       $x$ <= $x_vals$[i+1]) $prev$ = i;", //Statement
3     "{[i]: i>=0 && i<length}", //Iteration schedule
4   ...
```

Figure 4.5: A statement in the `Find_Segment` Computation.

Statements have an iteration space associated with them. This iteration domain is a set containing every instance of the statement and is typically expressed as the set of iterators that the statement runs over, subject to the constraints of their loop bounds. The iteration space is represented on line 3 of Figure 4.5 using the IEGenLib syntax with the exception of wrapping data spaces with the $ sign. Equation 4.1 represents the iteration space of the `Find_Segment` function.

$$I = \{[i] \mid i >= 0 \wedge i < length\} \tag{4.1}$$

As discussed in Section 2.4.2, constraints to an iteration space should be provided as symbolic constants. An interesting case that we came across is when constraints were passed in as arguments to the function. By default, adding parameters to a Computation also added them as data spaces to the Computation. It was necessary to identify variables that are never written to and make sure that they aren't renamed.

Data dependences are encoded using relations between iteration vectors and data space vectors. The data reads and writes can be specified as shown in Figure 4.6:

Execution schedules are determined using scheduling functions. They take iterators that apply to the current statement as the input, if any, and output the schedule as an integer tuple that may be lexicographically ordered with others to determine

```
1 {{"$x$","{[0]->[0]}"}, {"$x_vals$","{[i]->[i]}"}, {"$x_vals$","{[i]->[ip1]:
      ip1 = i+1}"}}, //Reads
2 {{"$prev$","{[0]->[0]}"}} //writes
```

Figure 4.6: The data reads and writes of the `Find_Segment`.

correct execution order of a group of statements. Figure 4.7 shows the execution schedule of the `Find_Segment` function.

```
1 //3rd parameter to the Stmt constructor
2 ...
3 "{[i]->[0, i, 0]"},
4 ...
```

Figure 4.7: The execution schedule of the `Find_Segment` Computation.

Once we have all the different components of a Computation, we can specify the full Computation as Figure 4.8. The Computation class interfaces with *CodeGen+* for code generation. Calling `codegen` on a Computation generates C code in the form of statement macros and calls to them. A macro is a fragment of code which has been given a name. Whenever the macro name is used, it is replaced by the contents of the macro. Figures 4.9 and 4.10 shows the result of code generation.

The complete `Find_Segment` Computation

```
Computation* Find_Segment_Computation(){

  Computation* FindSegmentComputation = new Computation();
  FindSegmentComputation->addParameter("$x$", "double");
  FindSegmentComputation->addParameter("$x_vals$", "double*");
  FindSegmentComputation->addParameter("$length$", "int");
  FindSegmentComputation->addParameter("$prev$", "int&");

  //Creating s0
  //if(x >= x_vals[i] && x <= x_vals[i+1]) prev = i;
  Stmt* s0 = new Stmt("if($x$ >= $x_vals$[i] &&
      $x$ <= $x_vals$[i+1]) $prev$ = i;",
      "{[i]: i>=0 && i<length}", //Iteration schedule
      "{[i]->[0, i, 0]}", //Execution schedule
      { //Data Reads
        {"$x$","{[0]->[0]}"},
        {"$x_vals$","{[i]->[i]}"},
        {"$x_vals$","{[i]->[ip1]: ip1 = i+1}"}
      },
      { //Data writes
        {"$prev$","{[0]->[0]}"}}
      );

  //Adding s0 to the computation
  FindSegmentComputation->addStmt(s0);

  //Adding the return value
  FindSegmentComputation->addReturnValue("$prev$", true);

  return FindSegmentComputation;
}
```

Figure 4.8: The complete `Find_Segment` Computation function

Compiler generated macros

```
1  ...
2  #define s0(__x0) double _iegen_2r = r;
3  #define s1(__x0) double _iegen_2theta = theta;
4  #define s2(__x0) double _iegen_2phi = phi;
5  #define s3(__x0) NaturalCubicSpline_1D & _iegen_2Temp_Spline = spl.
       Temp_Spline;
6  #define s4(__x0) double _iegen_2r_eval = min(_iegen_2r, r_max);
7  #define s5(__x0) _iegen_2r_eval = max(_iegen_2r_eval, r_min);
8  #define s6(__x0) double _iegen_2_iegen_1x = _iegen_2r_eval;
9  #define s7(__x0) struct NaturalCubicSpline_1D & _iegen_2_iegen_1Spline =
       _iegen_2Temp_Spline;
10 #define s8(__x0) double _iegen_0_iegen_2_iegen_1x = _iegen_2_iegen_1x;
11 #define s9(__x0) double* _iegen_2_iegen_1_iegen_0x_vals =
       _iegen_2_iegen_1Spline.x_vals;
12 #define s10(__x0) int _iegen_2_iegen_1_iegen_0length =
       _iegen_2_iegen_1Spline.length;
13 #define s11(__x0) int& _iegen_2_iegen_1_iegen_0prev = _iegen_2_iegen_1Spline
       .accel;
14 #define s12(__x0, i, __x2) if(_iegen_0_iegen_2_iegen_1x >=
       _iegen_2_iegen_1_iegen_0x_vals[i] && _iegen_0_iegen_2_iegen_1x <=
       _iegen_2_iegen_1_iegen_0x_vals[i+1]) _iegen_2_iegen_1_iegen_0prev = i;
15 #define s13(__x0) int _iegen_2_iegen_1k = _iegen_2_iegen_1_iegen_0prev;
16 #define s14(__x0) double _iegen_2_iegen_1result = 0.0;
17 #define s15(__x0) if(_iegen_2_iegen_1k < _iegen_2_iegen_1Spline.length) {
       double _iegen_2_iegen_1X = (_iegen_2_iegen_1x - _iegen_2_iegen_1Spline.
       x_vals[_iegen_2_iegen_1k])/(_iegen_2_iegen_1Spline.x_vals[
       _iegen_2_iegen_1k+1] - _iegen_2_iegen_1Spline.x_vals[_iegen_2_iegen_1k]);
        double _iegen_2_iegen_1A = _iegen_2_iegen_1Spline.slopes[
       _iegen_2_iegen_1k] * (_iegen_2_iegen_1Spline.x_vals[_iegen_2_iegen_1k+1]
       - _iegen_2_iegen_1Spline.x_vals[_iegen_2_iegen_1k]) - (
       _iegen_2_iegen_1Spline.f_vals[_iegen_2_iegen_1k+1] -
       _iegen_2_iegen_1Spline.f_vals[_iegen_2_iegen_1k]); double
       _iegen_2_iegen_1B = -_iegen_2_iegen_1Spline.slopes[_iegen_2_iegen_1k+1] *
        (_iegen_2_iegen_1Spline.x_vals[_iegen_2_iegen_1k+1] -
       _iegen_2_iegen_1Spline.x_vals[_iegen_2_iegen_1k]) + (
       _iegen_2_iegen_1Spline.f_vals[_iegen_2_iegen_1k+1] -
       _iegen_2_iegen_1Spline.f_vals[_iegen_2_iegen_1k]); _iegen_2_iegen_1result
        = (1.0 - _iegen_2_iegen_1X) * _iegen_2_iegen_1Spline.f_vals[
       _iegen_2_iegen_1k] + _iegen_2_iegen_1X * _iegen_2_iegen_1Spline.f_vals[
       _iegen_2_iegen_1k+1] + _iegen_2_iegen_1X * (1.0 - _iegen_2_iegen_1X) * (
       _iegen_2_iegen_1A * (1.0 - _iegen_2_iegen_1X ) + _iegen_2_iegen_1B *
       _iegen_2_iegen_1X);}
18 #define s16(__x0) double _iegen_2c_result = sqrt(gamR *
       _iegen_2_iegen_1result);
19 #define s17(__x0, __x1, __x2) sources.c = _iegen_2c_result;
20
21 unsigned int t2 = 0;
22 ...
```

Figure 4.9: Code generated for a single call to the `c` function from the `GeoAc_UpdateSources` function results in the Computation expressed as macros. The macro `s12` on line 14 represents the `Find_Segment` function.

Macros in use

```
1  s0(0);
2  s1(1);
3  s2(2);
4  s3(3);
5  s4(4);
6  s5(5);
7  s6(6);
8  s7(7);
9  s8(8);
10 s9(9);
11 s10(10);
12 s11(11);
13 for(t2 = 0; t2 <= _iegen_2_iegen_1_iegen_0length-1; t2++) {
       s12(12,t2,0);
       }
14 s13(13);
15 s14(14);
16 s15(15);
17 s16(16);
18 s17(17,0,0);
19 ...
```

Figure 4.10: Calls to the macros defined in Figure 4.9. Line 13 represents the call to the `Find_Segment` function.

CHAPTER 5

RELATED WORK

5.1 Applying polyhedral optimizations to scientific applications

Polyhedral Dataflow Graphs (PDFG) are a compiler internal representation that exposes optimization opportunities such as loop transformation and temporary storage reductions. To implement PDFG, the framework introduces a specification language called Polyhedral Dataflow Language (PDFL). This specification language can be written directly, derived from existing codes, or lifted from another intermediate representation.

Existing work demonstrates the benefit of polyhedral data flow optimizations. Olschanowsky et al. demonstrated this benefit on a computational fluid dynamic (CFD) benchmark [20]. Davis et al. automated the experiments from the previous work using modified macro dataflow graphs [6]. Strout et al. extended the polyhedral model for sparse computations allowing indirect array accesses [26]. This research distinguishes itself by being applied to a full application in a different domain.

Tools such as *Polly* [10], *Pluto* [3], *Loopy* [19], *PolyMage* [18] and *Halide* [22, 23, 17] use the polyhedral model to transform regular codes. *PolyMage* and *Halide* are two domain specific languages and compilers for optimizing parallelism, locality, and

recomputation in image processing pipelines. *Halide* separates the definition of the algorithm from the concerns of optimization making the algorithms simpler, more modular and more portable. This lets us play around with different optimizations with a guarantee not to change the result. *PolyMage's* optimization strategy relies primarily on the transformation and code generation capabilities of the polyhedral compiler framework and performs complex fusion, tiling, and storage optimization automatically.

The *isl* (integer set library) is a thread-safe C library for manipulating sets and relations of integer tuples bounded by affine constraints [28]. This library is used in the polyhedral model for program analysis and transformations and forms the basis for affine transformations used in all the tools discussed previously in this section.

CHAPTER 6

CONCLUSION

This thesis presents a case study to represent the *GeoAc* acoustic ray-tracing tool using the sparse polyhedral framework. *GeoAc* more accurately models wave propagation phenomena as compared to the current ray-tracing tool used in the *IonoSeis* modelling package. *GeoAc* was customized to meet the needs of *IonoSeis's* workflow. OpenMP was used to parallelize *GeoAc* which resulted in a 22x speedup over the serial version of the code. The run-time went down from about 19 hours to just under 51 minutes for a large set of input parameters.

The parallelized *GeoAc* code drives the development of a SPF optimization framework called the Computation API. The leaf functions of the *GeoAc* codebase were incrementally expressed in the SPF using the Computation API and code was generated as statement macros. These macros were used to replace the leaf functions and the application was run on a set of test parameters to determine the correctness of the results. Each iterative expression of the code in the SPF presented new challenges which helped drive the development of the optimization framework.

REFERENCES

[1] Martha L. Abell and James P. Braselton. 2 - first-order ordinary differential equations. In Martha L. Abell and James P. Braselton, editors, *Differential Equations with Mathematica (Fourth Edition)*, pages 45–131. Academic Press, Oxford, fourth edition edition, 2016.

[2] P Blom. Geoac: Numerical tools to model acoustic propagation in the geometric limit. `https://github.com/LANL-Seismoacoustics/GeoAc`, 2014.

[3] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System. *PLDI 2008 - 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–15, 2008.

[4] Chun Chen. Polyhedra scanning revisited. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 499–508, 2012.

[5] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[6] Eddie C Davis, Michelle Mills Strout, and Catherine Olschanowsky. Transforming loop chains via macro dataflow graphs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 265–277. ACM, 2018.

[7] José Fonseca. Gprof2dot: Convert profiling output to a dot graph,.

[8] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: A call graph execution profiler. 39(4):49–57, April 2004.

[9] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral ast generation is more than scanning polyhedra. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(4):12, 2015.

[10] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly - Polyhedral optimization in LLVM. *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT '11)*, page None, 2011.

[11] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[12] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The omega library interface guide. 1995.

[13] Alan LaMielle and Michelle Mills Strout. Enabling code generation within the sparse polyhedral framework. *Technical report, Technical Report CS-10-102*, 2010.

[14] John D McCalpin. A survey of memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, 19:25, 1995.

[15] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, July 1996.

[16] Thomas Mikesell, Lucie Rolland, Rebekah Lee, Florian Zedek, Pierdavide Coïsson, and Jean-Xavier Dessa. Ionoseis: A package to model coseismic ionospheric disturbances. *Atmosphere*, 10(8):443, Aug 2019.

[17] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):83, 2016.

[18] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 429–443. ACM, 2015.

[19] Kedar S. Namjoshi and Nimit Singhania. Loopy: Programmable and formally verified loop transformations. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9837 LNCS(July):383–402, 2016.

[20] Catherine Olschanowsky, Michelle Mills Strout, Stephen Guzik, John Loffeld, and Jeffrey Hittinger. A study on balancing parallelism, data locality, and recomputation in existing pde solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 793–804, 3 Park Ave, New York, NY, USA, 2014. IEEE Press, IEEE Press.

[21] Tobi Popoola, Ravi Shankar, Anna Rift, Shivani Singh, Eddie Davis, Michelle Strout, and Catherine Olschanowsky. An object-oriented interface to the sparse polyhedral library. In *2021 Ninth Workshop on Data-Flow Execution Models for Extreme Scale Computing*, Accepted.

[22] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. 2012.

[23] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[24] R. Rew and G. Davis. Netcdf: an interface for scientific data access. *IEEE Computer Graphics and Applications*, 10(4):76–82, 1990.

[25] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Compile-time composition of run-time data and iteration reorderings. *SIGPLAN Not.*, 38(5):91–102, May 2003.

[26] Michelle Mills Strout, Geri Georg, and Catherine Olschanowsky. Set and relation manipulation for the sparse polyhedral framework. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 61–75. Springer, 2012.

[27] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. An approach for code generation in the sparse polyhedral framework. *Parallel Computing*, 53:32–57, 2016.

[28] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*, pages 299–302. Springer, 2010.

[29] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[30] L. Zheng and X. Zhang. Chapter 8 - numerical methods. In Liancun Zheng and Xinxin Zhang, editors, *Modeling and Analysis of Modern Fluid Problems*, Mathematics in Science and Engineering, pages 361–455. Academic Press, 2017.

APPENDIX A

EXPERIMENTAL SETUP

The work in this thesis has been developed and tested on the R2 high performance cluster at Boise State University. Each node on the cluster is a dual socket, Intel Xeon E5-2680 v4 CPU at 2.40 GHz clock frequency. Each node consists of two Non-Uniform Memory Access (NUMA) domains each containing 14 cores. Each core has its own local memory and each node is connected with one another by means of high speed interconnect links. The cores include a 32KB L1, 256KB L2 cache. In addition, each core has access to a shared 35MB L3 data cache in each NUMA domain. Figure A.1 shows the CPU architecture information as a result of the `lscpu` command.

```
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 28
On-line CPU(s) list: 0-27
Thread(s) per core: 1
Core(s) per socket: 14
Socket(s): 2
NUMA node(s): 2
Vendor ID: GenuineIntel
CPU family: 6
Model: 79
Model name: Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz
Stepping: 1
CPU MHz: 1546.875
CPU max MHz: 3300.0000
CPU min MHz: 1200.0000
BogoMIPS: 4800.30
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 35840K
NUMA node0 CPU(s): 0,2,4,6,8,10,12,14,16,18,20,22,24,26
NUMA node1 CPU(s): 1,3,5,7,9,11,13,15,17,19,21,23,25,27
```

Figure A.1: The `lscpu` command displays the CPU architecture information of R2.