PORTABLE SPARSE POLYHEDRAL FRAMEWORK CODE GENERATION USING MULTI LEVEL INTERMEDIATE REPRESENTATION

by

Aaron St George

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

May 2023

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Aaron St George

Thesis Title: Portable Sparse Polyhedral Framework code generation using Multi Level Intermediate Representation

Date of Final Oral Examination: 27 February 2023

The following individuals read and discussed the thesis submitted by student Aaron St George, 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
Hoda Mehrpouyan, Ph.D.	Member, Supervisory Committee
Jim Buffenbarger, 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.

ACKNOWLEDGMENTS

I am sincerely grateful to my advisor Dr. Olschanowsky. She funded, supported, guided, and encouraged me throughout my masters degree. With her help, I've improved my writing ability, technical proficiency, and knowledge base in compilers. She guided me towards a thesis topic that's well aligned with my interests and career goals. I'm grateful to have met experts in the field and made professional contacts on travel opportunities she provided. Thank you Dr. Olschanowsky.

I would additionally like to thank my supervisory committee members Dr. Hoda Mehrpouyan, Dr. Jim Buffenbarger for their feedback on my proposal, thesis, and oral defense. Also, I would like to thank former member of my committee Dr. Jidong Xiao who has since left Boise State.

ABSTRACT

The Sparse Polyhedral Framework (SPF) provides vital support to scientific applications, but is limited in portability. SPF extends the Polyhedral Model to non-affine codes. Scientific applications need the optimizations SPF enables, but current SPF tools don't support GPUs or other heterogeneous hardware targets. As clock speeds continue to stagnate, scientific applications need the performance enhancements enabled by both SPF and newer heterogeneous hardware.

The MLIR (Multi-Level Intermediate Representation) ecosystem offers a large, extensible, and cooperating set of intermediate representations (called dialects). A typical compiler has one main intermediate representation, whereas an MLIR based compiler will have many. Because of this flexibility, the MLIR ecosystem has many dialects designed with heterogeneous hardware platforms in mind.

This work creates an MLIR SPF dialect. The dialect enables SPF optimizations and is capable of generating GPU code as well as CPU code from SPF representations. Previous C based SPF front ends are not capable of generating GPU code. The SPF dialect representations of common sparse scientific kernels generate CPU code competitive with the existing C based front end, and GPU code competitive with standard benchmarks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API – Application Programming Interface

AST – Algebraic Syntax Tree

COO – COOrdinate list

CPD – Canonical Polyadic Decomposition

CPU – Central Processing Unit

CSR – Compressed Sparse Row

CUDA – Compute Unified Device Architecture

FROSTT – Formidable Repository of Open Sparse Tensors and Tools

GCC – GNU Compiler Collection

GNU – GNU's Not Unix

GPU – Graphics Processing Unit

IR – Intermediate Representation

LLVM – Originally stood for "Low Level Virtual Machine", as the LLVM compiler infrastructure grew in scope "LLVM" simply became the name of the project.

MLIR – Mid Level Intermediate Representation

MTTKRP – Matricized Tensor Times Khatri-Rao Product

NUMA – Non-Uniform Memory Access

NVCC – NVidia CUDA Compiler

PASTA – Parallel Sparse Tensor Algorithm benchmark suite

SIMD – Single Instruction Multiple Data

SPF – Sparse Polyhedral Framework

SVD – Singular Value Decomposition

TPU – Tensor Processing Unit

TTM – Tensor Times Matrix

UF – Uninterpreted Function

CHAPTER 1

INTRODUCTION

Scientific applications need to port among a variety of hardware to obtain the desired performance. In the past, the performance of scientific applications increased as hardware improved. Today, clock speed has stagnated and improved performance comes from the increased parallelism found in new architectures such as GPUs and TPUs [10, 12]. Legacy scientific applications need to be ported to new architectures to remain relevant.

Scientific applications need better approaches to memory, as well as faster hardware, to improve performance. New architectures have many of the same bottlenecks as general purpose ones. Regardless of specialization, a computer requires many more cycles to load data than to do arithmetic operations [19]. A workload limited by data access speed is said to be memory bound. A memory bound workload will not see a speed increase on faster hardware. Scientific applications are often memory bound.

Compiler intermediate representations that enable high-level transformations, such as the Sparse Polyhedral Framework (SPF), optimize applications by reducing memory operations. Internally a compiler is free to represent an input program in whatever form is most amenable to optimization. The Polyhedral Model is a mathematical representation that is very useful for memory optimizations. Though powerful, the

polyhedral representation cannot represent sparse codes. The Sparse Polyhedral Framework extends the Polyhedral Model to sparse codes.

Current SPF tools don't support the portability that is increasingly demanded by scientific applications. SPF is a representation of a program that enables optimization. SPF must generate code, also known as lowering, either in the final target language or in another language that can be compiled further. Current SPF tools lower to C, and are thus as portable as C. Because of this approach, SPF tools can target most CPUs, but don't have the portability to reach other hardware targets. By lowering to something with greater reach, SPF tools would improve portability.

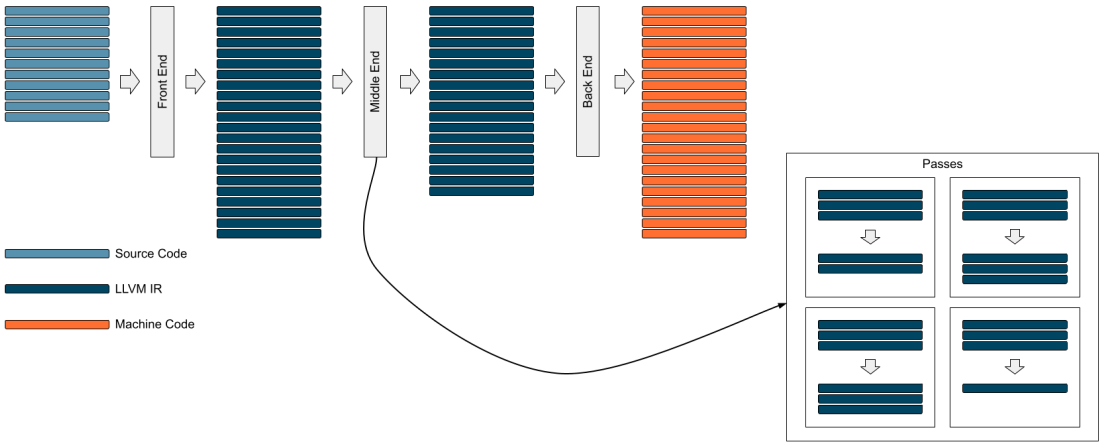

Figure 1.1: An LLVM based compiler

SPF tools based around the MLIR (Multi Level Intermediate Representation) [17] ecosystem could enable SPF based optimizations for a more heterogeneous set of hardware. MLIR is a compiler framework and not a compiler itself. A typical compiler has a single internal representation whereas an MLIR based compiler has many: see Fig: 1.1 vs Fig: 1.2. In an MLIR based compiler, each representation, or dialect

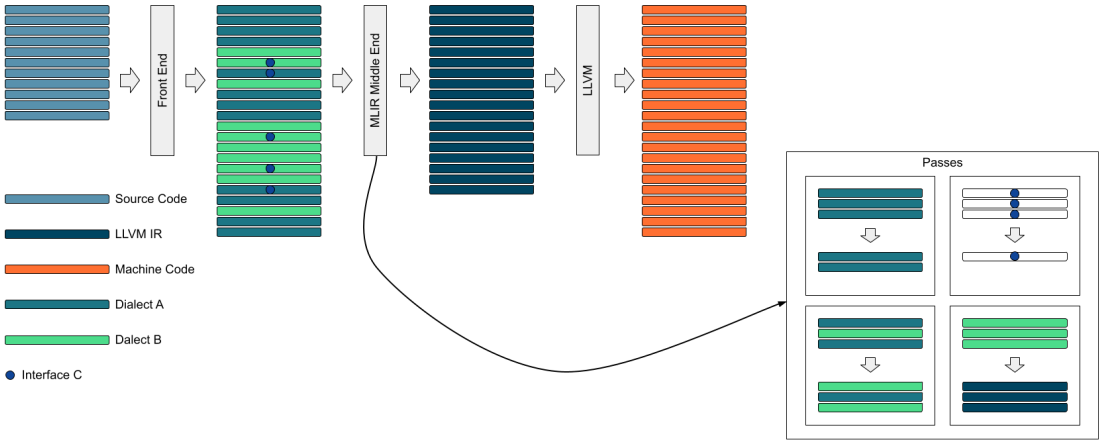

Figure 1.2: An MLIR based compiler

as they are known in MLIR, supports only the abstractions it was designed for. A dialect/representation doesn't have to be a "one size fits all" solution. Because of this flexibility, the MLIR ecosystem has many dialects designed with heterogeneous hardware platforms in mind.

An MLIR based compiler may lower the same code to very different hardware targets based on the path through the compiler. Dialects lower progressively via small steps, rather than all at once. Small steps provide many branch points. When diverging paths are taken, the same high level dialect can lower to GPU, CPU, or even some more exotic accelerator. SPF tools could integrate into this ecosystem rather than relying on C and take advantage of the portability the MLIR approach enables.

1.1 Problem Statement

As clock speeds continue to stagnate, scientific applications will need to be portable to new architectures to achieve the performance scientists need. SPF tools enable optimizations that are needed by scientific applications on new and legacy hardware. Current SPF tools do not have the portability required to provide benefit on newer architectures. This motivates the research question: how do we build more portable SPF tools?

We hypothesize that the heterogeneous targets and progressive lowering approach enabled by MLIR will improve the portability of SPF tools. To evaluate this approach we will generate kernels that run on both CPUs and GPUs. Existing SPF tools are not capable of doing this.

1.2 Contributions

This work creates an MLIR SPF dialect. This dialect enables SPF transformations in a new software ecosystem. MLIR's multiple levels of representation allow independent and efficient optimization across different levels of abstraction. Because of this flexibility, SPF tools utilizing the MLIR dialect can now generate code for both CPU and GPU. The C based SPF front end used in previous work [29, 23] can only generate CPU code.

This work heavily leverages pre-existing infrastructure in MLIR. Specifically, this work creates an MLIR dialect `spf`, and a lowering pass `convert-spf-to-loops`. The `convert-spf-to-loops` pass lowers from `spf` to a host of other MLIR dialects not

created by this work. Many other lowering passes and tools within MLIR (not created by this work) are needed for full CPU and GPU code generation.

A performance evaluation done on common scientific kernels shows competitive performance for code generated from the MLIR front end on both CPU and GPU. Performance evaluation consisted of MTTKRP and TTM implementations from the PASTA benchmark suite, generated from the C front end, and the MLIR front end.

1.3 Organization

This work is organized into several chapters and sections. Chapter 2 provides background on MLIR, the Polyhedral Framework, and the Sparse Polyhedral Framework (SPF). Chapter 3 introduces how the SPF MLIR dialect is represented, transformed, and lowered in MLIR. Chapter 3 also contains the performance evaluation of generated code. Chapter 4 contains a review of related work. Chapter 5 concludes and summarizes this work.

CHAPTER 2

BACKGROUND

In this chapter we discuss concepts that form the basis of our work: the Sparse Polyhedral Framework and MLIR. Additionally as this work extends SPFs portability to GPUs, we give a brief description of GPU architecture.

2.1 Sparse Polyhedral Model

The Polyhedral Model provides a mathematical representation of loop nests that enables transformations and dependency analysis. The Polyhedral Model represents each iteration of a loop nest as lattice points in a polyhedron, and the data dependencies between points as relations. The abstraction provided by the Polyhedral Model allows transformation relations to be applied to the iteration space to change the order of computation. A transformation that preserves the partial order established by the data dependency relations is legal.

```
1     for(int i = 0; i < n; i++) {
2         for(int j = 0; j < i; j++) {
3 S0:         a[i][j]=b[i][j]
4         }
5     }
```

Figure 2.1: Example code

Consider the example in Fig: 2.1. The code contains an `S0` marker. The `S0` marker doesn't have any semantic meaning for the code. The Polyhedral Model abstracts over what a given statement actually does. The Polyhedral Model doesn't require more information about the code represented by `S0` than that it occurs at an iteration tuple $[i, j]$, reads data from `b`, and writes to `a`. Hence, the Polyhedral Model keeps track of this statement by just referring to it as `S0`.

```
1 for(int j = 0; j <= n - 2; j++) {
2     for(int i = j + 1; i <= n-1; i++) {
3         a[i][j]=b[i][j]
4     }
5 }
```

Figure 2.2: Example after transformation

The Polyhedral Model defines the **iteration space** for statement `S0` as a set containing all tuples of induction variables executed by the loop nest.

$$\{[i, j] : 0 \leq i < n \wedge 0 \leq j < i\}$$

Code will be generated that loops over tuples in lexicographic order established by a relation called the **execution schedule**.

$$\{[i, j] \rightarrow [0, i, 0, j, 0]\}$$

Creating a composed relation from the execution schedule and the following transformation relation will result in loop interchange ordering. Underscores are used for unused variables.

$$\{[\_, i, \_\_, j, \_\_\_] \rightarrow [0, j, 0, i, 0]\}$$

Doing code generation after the composed relation is applied to the iteration space yields the code in Fig: 2.2.

```
1     for(int i = 0; i < N ; i++) {
2         for(int k = rowptr[i]; k < rowptr[i + 1] ; k++) {
3             int j = col[k];
4 S0:         printf("i: %d, j:%d\n", i, j ) ;
5         }
6     }
```

Figure 2.3: Iterating over a matrix in CSR

The Polyhedral Model can only express affine iteration spaces. The limited expressiveness of the Polyhedral Model allows fully decidable proofs that a transformation maintains the partial order established by the data dependency relations using decision procedures for Presburger Arithmetic. However, affine iteration spaces cannot express indirect data accesses such as $A[B[i]]$ or non-linear constraints. Sparse codes operate over sparse data structures with highly irregular access patterns and require indirect data accesses to function. An example sparse code iterating over a matrix stored in Compressed Sparse Row (CSR) format is given in Fig: 2.3.

The Sparse Polyhedral Framework (SPF) extends the Polyhedral Model by representing non affine data accesses with uninterpreted functions. Uninterpreted functions are treated as a special case of symbolic constants. SPF provides much of the same functionality as the traditional Polyhedral Model. SPF allows transformations to be specified as relations, and code to be generated from SPF representations. An example of the non-affine iteration space of the above CSR code represented using

uninterpreted functions is given below. This example could not be represented in the Polyhedral Model.

$$\{[i,j,k] : 0 \leq i < N \wedge rowptr(i) \leq k < rowptr(i+1) \wedge j = col(k)\}$$

2.2 MLIR

MLIR is a compiler framework that allows for multiple levels of representation. MLIR's original goals include reducing fragmentation in the compiler ecosystem, targeting heterogeneous hardware, and connecting existing compilers together [17]. MLIR has been used extensively in domain specific compilers: for example machine learning systems [6], and climate simulation [8].

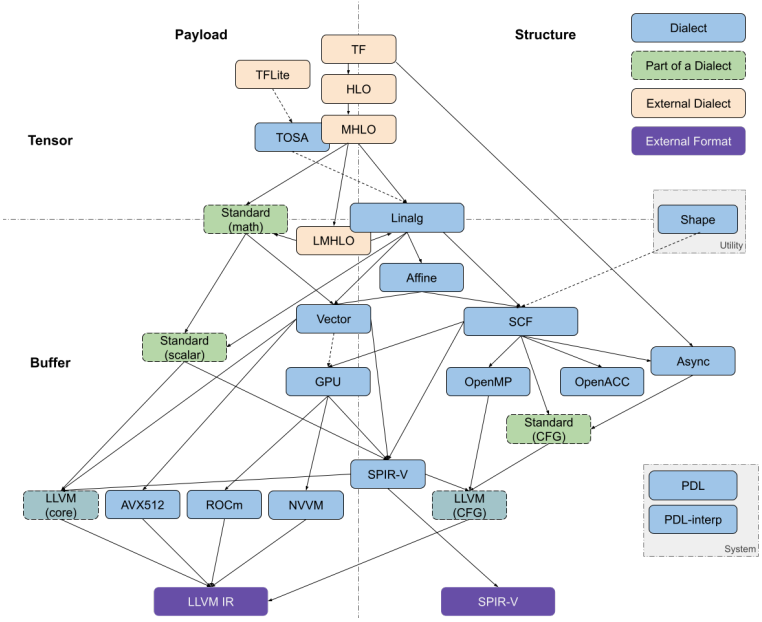

Figure 2.4: MLIR Dialects (image from [32])

An MLIR based compiler allows for many internal representations (called dialects in MLIR). A more typical compiler has one main representation. Moving away from a one-size-fits-all approach opens up a lot of design space. MLIR contains many dialects specialized for particular optimization techniques, or a particular domain. MLIR optimization passes often interact over multiple dialects. Any MLIR code that does anything real is almost certain to contain many dialects.

MLIR lowers from high level dialects to low-level (possibly hardware specific) dialects progressively. A progressive lowering removes structure slowly. As code is lowered through the levels of abstraction, passes remove structure only after it is no longer needed. A progressive approach can prevent the need for expensive and difficult analyses. If structure isn't removed, a compiler has no reason to run an analysis to recover it. Some of the many dialects in MLIR and the many lowering paths between them are shown in Fig: 2.4.

A dialect itself is a container for more fundamental MLIR abstractions. A dialect groups operations, types, and annotations. Below a discussion of each concept and a more complete example are given.

Figure 2.5: Operation figure from [17] available under CC 4.0

Operations: Operations are the basic building block of MLIR. MLIR doesn't fix the set of available operations. Rather, the set of operations is fully extensible. An Operation can return results, take inputs, and have enclosed *regions* (containers for further enclosed operations). See Fig: 2.5. In the human readable format, MLIR prepends the name of the dialect that an operation is part of before the name of the operation. From Fig: 2.5 the name of the dialect for the first operation would be "`d`".

Attributes: Attributes define constant data attached to an operation. Attributes can be of several built-in types, or can be extended to new types.

Types: Every value in MLIR has a type. The type system is fully extensible.

```
scf.for %arg2 = %c0 to %5 step %c1 {
  scf.for %arg3 = %c0 to %6 step %c1 {
    %11 = memref.load %1[%arg2, %arg3] : memref<?x?xf32>
    memref.store %11, %4[%arg2, %arg3] : memref<?x?xf32>
  }
}
```

Figure 2.6: Example MLIR Code

The example in Fig: 2.6 puts these concepts into a more complete context. The example contains operations from two dialects: `scf` and `memref`. `scf` stands for 'structured control flow'. The `scf` dialect contains most of MLIR's looping constructs. The `memref` dialect is used for creating and manipulating references to memory. The first `scf.for` contains a region, enclosed in curly braces, that contains another `scf.for` operation. This second `scf.for`'s enclosed region contains a `memref.load` operation that produces a value of type (types are always on the right after a colon)

of `memref<?x?xf23>`. The `memref` represents a reference to a 2D array of unknown size storing 32 bit floats.

All variables in MLIR are in SSA form [26]. In the human readable form, MLIR prepends a "`%`" before a variable name to indicate that it is in SSA. SSA form ensures that each variable is assigned exactly once. This property makes various analyses and transformations easier for compiler writers.

```
1 ^bb0(%cond: i1):
2   cf.cond_br %cond, ^bb1, ^bb2
3
4 ^bb1:
5   %a = arith.constant 42 : i64
6   cf.br ^bb3(%a: i64)
```

Figure 2.7: Example MLIR Code: Basic Blocks

MLIR code is often structured into basic blocks. A basic block is a series of instructions without branches in except the entry, and without branches out except exit [9]. Fig: 2.7 shows and example of two basic blocks named: `bb0` and `bb1`: Similar to the `%` character for SSA values, MLIR human readable form prepends `^` to the name of basic blocks. Basic blocks are a common abstraction in most compilers, unlike other compilers MLIR also provides block arguments. Block arguments behave in a similar manner to function arguments, but for basic blocks. Block arguments replace the phi-nodes typically used in SSA compilers [26]. Block arguments shift the responsibility of choosing the value an SSA variable takes to the code jumping into a basic block. The *caller* rather than a phi-node at the entry to a block, the *callee*, is responsible for disambiguating SSA values. The MLIR SPF dialect uses block arguments in the presented abstraction.

2.3 GPU Architecture

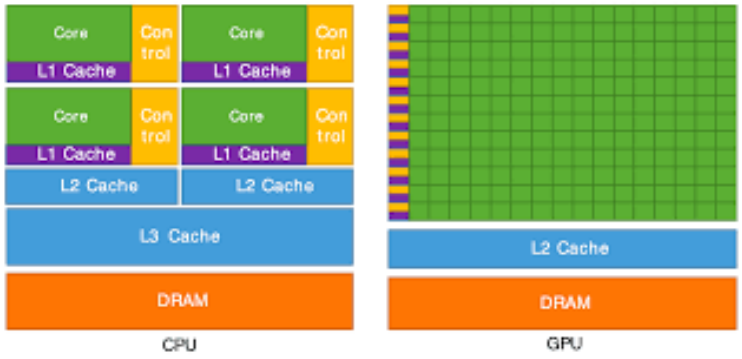

Figure 2.8: GPU Architecture (image from [15])

Graphics workloads demand doing a *lot* of relatively simple, parallel, tasks. While CPUs are designed for low latency, GPUs (Graphics Processing Units) are designed for high throughput. Fig: 2.8 shows an idealized chip layout for a CPU vs GPU. One can see that a CPU has a small number of powerful cores with a decent amount of die area given to caching to improve latency. In contrast, a GPU has a large number of small, relatively weak cores without much cache. The GPUs design ensures high throughput for any workload that has very large number of relatively simple, maybe memory bound, tasks.

While GPUs were originally designed for graphics, CUDA (Compute Unified Device Architecture) brought compute to GPUs. After 2007 when Nvidia released CUDA, programmers could compile code written in the CUDA C/C++ API and run that code on a GPU. GPU cores (typically) have separate memory from CPU cores. Both a compiled GPU program and any data this program will execute on must be first moved to the GPU before the program can execute. On Linux, the CUDA API

moves data and code to the GPU via a syscall to the GPU driver. The GPU driver then runs the code on the GPU.

CHAPTER 3

SPARSE POLYHEDRAL FRAMEWORK IN MLIR

The SPF MLIR dialect increases the portability of SPF tools by embedding SPF in an ecosystem designed for heterogeneous computing. SPF is used in a wide variety of applications, it can be used internally in compilers or in standalone tools, for example recent work used SPF to synthesize and optimize code for translating between different sparse tensor formats [24]. Existing SPF front ends integrate with the C language. While C has excellent portability among CPUs, C cannot naively target GPUs or other accelerators important for today's scientific workloads. MLIR is designed to improve compilation for heterogeneous targets. We present an MLIR dialect for SPF that broadens the reach of SPF tools.

This chapter will detail SPF's representation, transformation, and lowering in MLIR as well as evaluate performance of generated code. Sections proceed in the order of needs of an SPF based tool. For an SPF based tool to use the MLIR front end, it needs to be able to naively represent SPF in MLIR. The first section details the SPF dialect representation. Tools use SPF because it enables transformations that improve performance. The second section walks through how the SPF dialect represents and performs transformations. SPF enables transformations, but to produce executable code SPF representations must generate (also known as *lowering* to) assembly lan-

guage. The third section describes the process of lowering SPF representations. The final section evaluates the performance of lowered code against the existing C front end, and the PASTA benchmark suite [18].

3.1 Representation

To create MLIR based SPF tools, SPF needs an MLIR representation. MLIR is designed to be extensible. As discussed in 2.2, MLIR provides *dialects* as its primary extension mechanism. To embed SPF into MLIR, this work creates an SPF dialect which allows users to represent SPF abstractions and transformations in MLIR. This section details the SPF dialect by first introducing a motivating example, then describing this example's representation in SPF, in the existing SPF tool Computation API (which is leveraged heavily by the MLIR representation), and finally in MLIR.

3.1.1 Example Scientific Kernel In SPF

```
1    for (int t = 1; t <= ub_T / 2; ++t) {
2      for (int x = lb_x; x <= ub_x; ++x)
3  S0:   A[x] = (B[x - 1] + B[x] + B[x + 1]) / 3;
4
5      for (int x = lb_x; x <= ub_x; ++x)
6  S1:   B[x] = (A[x - 1] + A[x] + A[x + 1]) / 3;
7    }
```

Figure 3.1: Jacobi in C

To motivate this work, we present a Jacobi 1D kernel. A C implementation of the inner kernel is presented in Fig: 3.1. This benchmark is intended to be representative of a common pattern seen across many domains doing Jacobi-like computations such

as heat transfer, and discrete wave equations. The specific kernel presented in Fig: 3.1 could be used to simulate heat dissipation through a theoretical 1D rod [3]. To prevent mutating the input to the current time step, the kernel stores the results of even and odd parity timestamps in separate arrays.

A Jacobi kernel written as in the example (which is common in scientific applications) suffers from poor data locality. Data that could be re-used by the second loop will be pushed out of cache by loads required from the first loop before it can be used. The Sparse Polyhedral Framework offers transformations that can improve data locality of this kernel. An SPF representation of the computation must be constructed to make use of the transformations.

3.1.2 SPF Representation

SPF transformations can improve the data locality of the example Jacobi Kernel, but we are required to represent this kernel in SPF first. SPF represents a kernel using a mathematical representation based on Presburger Arithmetic extended with uninterpreted functions (discussed in 2.1). This representation makes transformations easy to do, and easy to check for legality with powerful solvers [25]. We will now walk through the Jacobi kernel expressed in SPF.

The Jacobi kernel has two statements `S0` and `S1`. Each statement executes at tuples formed from the loop induction variables. SPF calls the set of tuples at which a statement executes the iteration space. Statement `S0` and `S1` have the same iteration space:

$$\{[t, x] : 1 \leq t < \texttt{ub\_T}/2 \wedge \texttt{lb\_x} \leq x < \texttt{ub\_x}\}$$

Transformations, as discussed in 2.1, are applied to the execution schedule. SPF applies the execution schedule to the iteration space tuples to create execution schedule tuples. Code generation, though the polyhedral scheduling algorithm that will be discussed in 3.3.1, produces loops iterating over the execution space tuples in lexicographic order. SPF requires the execution schedule to be an invertible relation, ensuring that each execution schedule tuple can be mapped back to an iteration space tuple and hence an execution of the original statement. The example will have the following execution schedule for `S0`:

$$\{[t, x] \rightarrow [t, 0, x, 0]\}$$

When compared to the iteration space tuples, the execution schedule tuples for statement `S0` has an added dimension always having the value 0. The added dimension ensures that the execution schedule tuples of `S0` sort lexicographically before those of `S1` which has execution schedule:

$$\{[t, x] \rightarrow [t, 1, x, 0]\}$$

The added dimension could be thought of as an extra loop as in Fig: 3.2, though code generation won't actually produce this superfluous loop. Code generation will produce code similar to that in Fig 3.1. The statement, iteration space, and execution schedule abstractions are enough to generate code, but not enough to determine the legality of a transformation.

```
for (int t = 1; t <= ub_T / 2; ++t) {
  for (int tmp = 0; tmp <= 1; tmp++)
    if (tmp == 0) {
      for (int x = lb_x; x <= ub_x; ++x)
        A[x] = (B[x - 1] + B[x] + B[x + 1]) / 3;
    } else if (tmp == 1) {
      for (int x = lb_x; x <= ub_x; ++x)
        B[x] = (A[x - 1] + A[x] + A[x + 1]) / 3;
    }
}
```

Figure 3.2: Jacobi C Example With Extra Loop

To establish a partial ordering of execution space tuples, the SPF abstraction requires read and write access relations for each statement. The input to read and write relations is the iteration space, and the output indices at which memory array is accessed. In the Jacobi example statement `S0` reads from `B` at:

$$\{[t, x] \rightarrow [x - 1]\}$$

$$\{[t, x] \rightarrow [x]\}$$

$$\{[t, x] \rightarrow [x + 1]\}$$

and writes to `A` at:

$$\{[t, x] \rightarrow [x]\}$$

statement `S1` reads from `A` at:

$$\{[t, x] \rightarrow [x - 1]\}$$

$$\{[t, x] \rightarrow [x]\}$$

$$\{[t, x] \rightarrow [x + 1]\}$$

and writes to `A` at:

$$\{[t, x] \rightarrow [x]\}$$

The read and write accesses allow SPF to establish the legality of transformations. For example in Jacobi, before transformation, the lexicographic order on execution space tuples tells SPF the execution of `S1` at iteration space tuple $[1, 1]$ happens after the iteration of `S0` at the same tuple. From the write relation for `S0`, SPF knows that the execution of statement `S0` at iteration space tuple $[1, 1]$ writes to `A` at 1. From the read relation for `S1`, SPF knows that the execution of statement `S1` at iteration space tuple $[1, 1]$ reads from `A` at 1, at 0, and at 2. From the read and write relations, SPF knows that the execution of statement `S1` at iteration space tuple $[1, 1]$ reads data written by the execution of statement `S0` at iteration space tuple $[1, 1]$. Given the information in the read and write relations and statement ordering, SPF establishes a data dependence between the execution of statement `S0` at iteration space tuple $[1, 1]$ and the execution of `S1` at the same tuple. From this computed data dependence SPF builds a partial ordering of statements in which `S0` at iteration space tuple $[1, 1]$ comes before `S1`. The partial order SPF computes contains all the happens before relationships that are required to be true for the computation to be correct. The computed partial order can also be thought of as a data-flow graph. SPF can then verify that any transformation keeps this original partial order intact using powerful solvers [25].

3.1.3 Computation API Representation

This work builds upon the Computation API [23]. The Computation API provides a single entry point C++ API to Sparse Polyhedral Framework tools. Such tools include the Inspector/Executor Generation Library (IEGenLib) [28] which provides set and relation manipulation with constraints involving uninterpreted functions, and Omega+ for code generation [5]. Internally, the SPF MLIR dialect constructs a Computation API representation of SPF to take advantage of the capabilities of the existing tool. Though a user of the SPF MLIR dialect would interact with the tool through MLIR, and might not know that the Computation API is used internally, the Computation API forms an integral piece of the MLIR dialect's functionality.

Fig: 3.3 shows the Jacobi example expressed in Computation API. The Computation API provides a C++ object-oriented interface though a C++ Computation class with everything one would need to express an SPF computation. This includes: statements, data spaces, data dependence relations (Reads/Writes), iteration spaces, and execution schedules. A description of each of the concepts is given below.

Statements: Statements are essentially a loop body. A statement performs read and write operations on Data Spaces. A statement will be executed at every tuple represented in the iteration space. The statement in the example is taken from the Jacobi example in Fig: 3.1.

Data Spaces: Data spaces represent n-dimensional non-overlapping memory addresses that are written to or read from by the statement. A 0-dimensional data space would represent a scalar, such as a single variable. A 1-dimensional

```
1  Computation jacobi;
2  jacobi.addDataSpace("A", "double*");
3  jacobi.addDataSpace("B", "double*");
4  jacobi.addStmt(
5      new Stmt(/*stmtSourceCode*/ "A(x)=(B(x-1) + B(x) + B(x+1))/3",
6               /*iterationSpaceStr*/ "{[t,x]: 1<=t<=ub_T and lb_x<=x
    <=ub_x}",
7               /*executionScheduleStr*/ "{[t,x]->[t,0,x,0]}",
8               /*dataReadStrs*/ {
9                                  {"B", "{[t,x]->[c]: c=x-1}"},
10                                 {"B", "{[t,x]->[x]}"},
11                                 {"B", "{[t,x]->[c]: c=x+1}"}
12                               },
13              /*dataWriteStrs*/ {{"A", "{[t,x]->[x]}"}}));
14 jacobi.addStmt(
15     new Stmt(/*stmtSourceCode*/ "B(x)=(A(x-1) + A(x) + A(x+1))/3",
16              /*iterationSpaceStr*/ "{[t,x]: 1<=t<=T and 1<=x<=X}",
17              /*executionScheduleStr*/ "{[t,x]->[t,1,x,0]}",
18              /*dataReadStrs*/ {
19                                 {"A", "{[t,x]->[c]: c=x-1}"},
20                                 {"A", "{[t,x]->[x]}"},
21                                 {"A", "{[t,x]->[c]: c=x+1}"}
22                               },
23              /*dataWriteStrs*/ {{"B", "{[t,x]->[x]}"}}));
```

Figure 3.3: Computation API Jacobi

data space would represent a vector. In the example, the `A` data space represents the 1-dimensional array as does `B`.

Data Reads/Writes: The Computation API uses relations to encode data relationships between statements. The data dependence relations provide a partial ordering which any transformation must respect. In the example, each iteration of the first statement will read from the `B` data space at $x-1$, x, and $x+1$; and write to `A` at x.

Iteration Spaces: The Iteration Space provides an unordered set of all the tuples

at which a statement will execute.

Execution Schedules: Execution schedules are a relation that will determine the order in which statements are executed. SPF will apply the execution schedule to the iteration space and lexicographically order the result to determine the correct execution order of the statements. An execution schedule is required to maintain the partial order established by the read and write relations. The execution schedule in the example will result in a final generated code that looks like the C Jacobi example in Fig: 3.1.

3.1.4 MLIR Representation

The MLIR SPF representation provides the same abstraction presented here mathematically and through Computation API, but in an MLIR specific way. As discussed in the background 2.2, MLIR provides composable abstractions through operations, attributes, and types. We will walk through the Jacobi example in detail below, but in brief the SPF dialect adds two main operations to MLIR:

`spf.computation`: The `spf.computation` operation is analogous to the `Computation` class in the Computation API. All `spf.statement` operations must be nested inside the enclosed region (regions are discussed in the background 2.2).

`spf.statement`: The `spf.statement` operation is analogous to the `Stmt` class in the Computation API. Read and write relations, execution schedules, and iteration spaces are provided as attributes on the `spf.statement` operation. The

`spf.statement` operation's arguments are analogous to Data Spaces in the Computation API.

An example Jacobi kernel represented in SPF MLIR dialect is given in example 3.4. This example doesn't include all the code that would be needed for a running example, it only shows the inner core of the kernel. A running example would require support code to allocate arrays, set up constants, and create an entry point. The example is displayed in MLIR's human readable format. The human readable format is primary used for testing. MLIR based tools and compilers usually interact with the C++ API. Given its use cases, the human readable format is designed for completeness and fidelity rather than readability. To ensure that the meanings are clear, we will walk through the example in detail:

Line 1: This line begins a `spf.computation` operation. It takes no arguments, and has one enclosed region. The `({` lexeme denotes opening the `spf.computation`'s enclosed region. All operations in MLIR prepend the name of the dialect that defines them to their name. In the example, `arith.addf` operation on line 4 isn't surrounded by quotes while `"spf.computation"` is. The presence or absence of quotes is just a syntactic difference: unless a custom parser is defined for an operation, the MLIR parser requires that name of the operation be surrounded by quotes. The `arith.addf` operation defines a custom parser, the `spf.computation` operation does not—hence the quotes around `spf.computation`.

Line 2: This line begins the first of two `spf.statement` operations in the example.

```
"spf.computation"() ({
    "spf.statement"(%ub_T_div_2, %lb_x, %ub_x, %B, %A) ({
    ^stmt(%B_x_plus_one: f64, %B_x: f64, %B_x_minus_one: f64):
    %0 = arith.addf %B_x_plus_one, %B_x : f64
    %1 = arith.addf %0, %B_x_minus_one : f64
    %2 = arith.divf %1, %f3 : f64
    "spf.yield"(%2): (f64) -> ()
    })  {  reads = [
                        [ // data access functions for first input
                            affine_map<(t, x) -> (x+1)>,
                            affine_map<(t, x) -> (x)>,
                            affine_map<(t, x) -> (x-1)>
                        ]
                    ],
           writes = [[affine_map<(t, x) -> (x)>]],
           // symbols, ufInputs, inputs, outputs
           operand_segment_sizes=array<i32: 3,0,1,1>,
           symbolNames= ["ub_T", "lb_x", "ub_x"],
           iteratorTypes = ["reduction", "reduction"],
           executionSchedule = "{[t,x]->[t,0,x]}",
           iterationSpace = "{[t,x]: 1<=t<=ub_T and lb_x<=x<=ub_x}",
           transforms = []
        }:(index,index,index,memref<10xf64>,memref<10xf64>)->()
    "spf.statement"(%ub_T_div_2, %lb_x, %ub_x, %A, %B) ({
    ^stmt(%A_x_plus_one: f64, %A_x: f64, %A_x_minus_one: f64):
    %0 = arith.addf %A_x_plus_one, %A_x : f64
    %1 = arith.addf %0, %A_x_minus_one : f64
    %2 = arith.divf %1, %f3 : f6
    "spf.yield"(%2): (f64) -> ()
    })  {  reads = [
                [ // data access functions for first input
                    affine_map<(t, x) -> (x+1)>,
                    affine_map<(t, x) -> (x)>,
                    affine_map<(t, x) -> (x-1)>
                ]
            ],
           writes = [[affine_map<(t, x) -> (x)>]],
           // symbols, ufInputs, inputs, outputs
           operand_segment_sizes = array<i32: 3,0,1,1>,
           symbolNames = ["ub_T", "lb_x", "ub_x"],
           iteratorTypes = ["reduction", "reduction"],
           executionSchedule = "{[t,x]->[t,1,x]}",
           iterationSpace = "{[t,x]: 1<=t<=ub_T and lb_x<=x<=ub_x}",
           transforms = []
        }:(index,index,index,memref<10xf64>,memref<10xf64>)->()
}): () -> ()
```

Figure 3.4: MLIR SPF Jacobi

In general, `spf.statemetnt` operations may take a variable number of arguments. The operation, starting on line 4, takes 4 arguments whose expected types are spelled out on **Line 23**. All arguments are in SSA form (discussed in the background section 2.2). Arguments to the statement fall into three named groupings specified on **Line 17-18**: symbols, inputs, and outputs.

- The symbols grouping contains the first three arguments `%ub_T_div_2`, `%lb_x`, and `%ub_x`. These three arguments will fill in for symbolic constants in the iteration space. For clarity, the arguments are the same as those used in the C example in Fig: 3.1 (`ub_T`, `lb_x`, and `ub_x`) except the first argument, `%ub_T_div_2`, which takes the value that the expression `ub_T / 2` does in line 1 of the C example (`for ( int t = 1; t <= ub_T / 2; ++ t ) {`). All three arguments are of type `index`, which is the MLIR equivalent of C's `uint64_t`.
- The inputs grouping contains only the third argument. The third argument is an input to the statement; it's the equivalent of the `A` array from the C example.
- The outputs grouping contains only the fourth argument. This argument is an output from the statement (an out parameter); it's equivalent to `B` array from the C example.

Both input and output arguments are of type `memref<10xf64>` which denotes a reference to a memory block of ten 64-bit floating point numbers. MLIR's `memref<10xf64>` is equivalent to a `*double[10]` type in C or a `std::array<double,`

`10>` in C++. The final lexeme on line 2 (`({` denotes the opening of a region. This region belongs to this `spf.statement` operation and continues to line 8 where it closes with the `})` lexeme.

Line 3: The MLIR SPF dialect's version of a statement begins on line 3. Line 3 creates a basic block named `stmt`. The `stmt` block forms the MLIR SPF dialect's statement abstraction. This statement will become the core of a loop generated during lowering (discussed in 3.3). `stmt`'s block arguments: `%B_x_plus_one`, `%B_x`, and `%B_x_minus_one` (block arguments are discussed in the background 2.2) are roughly equivalent to variables used in the `stmtSourceCode` argument to the `Stmt` class in the Computation API. Lowering generates loads from the statement inputs `%B` such that the variable `%B_x_plus_one` holds the same value as the result of the expression `B[x+1]` on line 5 (`A[x] = (B[x-1] + B[x] + B[x+1]) / 3;`) from the C example 3.1. The same will be true for `%B_x` and `B[x]`, and `%B_x_minus_one` and `B[x-1]`.

Lines 4-6: These lines make up the core of the statement. Here, operations from the `arith` dialect add all three block arguments together then divide by `%f3`. `%f3` is defined outside of the example and holds the 64 bit floating point number `3.0`. Lines 4-6 correspond to `(B[x-1] + B[x] + B[x+1]) / 3` on line 5 of the C example in Fig: 3.1.

Line 7: The `spf.yield` operation on this line informs lowering of what should be considered the results of a statement. Lowering generates write operations that write the outputs of a computation to the appropriate storage. Through

lowering, this line is part of what becomes analogous to the `+=` from line 5 of the C example.

Lines 8-14: Line 8 ends the region that began on **Line 2**, and begins the attributes section. Attributes are discussed in the background section 2.2. The first attribute, `read`, continues to line 14. The `read` attribute is equivalent to `dataReadStrs` from the Computation API. `read`'s will become read access relations in SPF. The Computation API uses C++'s `std::pair` to associate a read access relation with the data space it reads from. For example, line 10 of the Computation API example in Fig: 3.3 (`{"B", "{[t, x]->[x]}"}`) indicates that the statement reads from data space `"B"` with the given relation. The MLIR front end does this association by mapping the position in `read` list of lists to the position in the input grouping. Each item in the list at the 0^{th} index in `reads` is a read relation associated with the 0^{th} input to the statement. In this case, there are three reads from `%A` for each statement. The SPF dialect expects each individual read to be of the MLIR `affine_map` type. The `affine_map` type is an MLIR primitive for affine relations. While SPF doesn't require data access relations to be affine like `affine_map` does, in practice non-contrived data access relations are always affine.

Line 15: This line sets the `write` attribute. The `write` attribute mirrors the `read` attribute but for write rather than read access relations.

Lines 16-17: These lines and set the `operand_segment_sizes` attribute. Operations across the MLIR ecosystem use the `operand_segment_sizes` attribute. Unlike

other attributes used in the MLIR Jacobi example, `operand_segment_sizes` is not specific to the SPF dialect. The `operand_segment_sizes` attribute stores an array mapping an operation's operands to groupings that the operation defines. `spf.computation` defines four groupings:

symbols holds variables that are expected to fill in for symbolic constants in iteration spaces. Position in the array on **Line 18** determines which operand should fill in for which symbolic constant. We give an expanded description of **Line 18** below.

ufInputs holds arguments that should be passed to any uninterpreted function calls when they are realized during lowering. The Jacobi example doesn't contain any uninterpreted function calls.

inputs holds inputs to the statement. Using Computation API terminology, the inputs grouping holds data spaces the statement reads from and does not write to. Lowering generates load operations reading from inputs based on the write relations provided.

outputs holds outputs from the statement. Using Computation API terminology, the outputs grouping holds data spaces the statement writes to. The statement may read as well as write an output. For example, if a statement contains the equivalent of `+=`. Lowering generates store operations writing to outputs based on the write relations provided.

In our example there are three symbolic constants, no uninterpreted function arguments, one input, and one output.

Line 18: This line sets the `symbolNames` attribute on first `spf.computation` operation. `symbolNames` is an array. The SPF dialect requires it to be the same size as the symbols grouping. Lowering uses the position within the symbols grouping to map operands to strings in the `symbolNames` array. The SPF dialect requires that each symbolic constant used inside the iteration space (which is given as a string) is mapped to an argument through the `symbolNames` `symbols` grouping paring.

Line 19: This line sets the `iteratorTypes` attribute. A programmer or tool marks each loop in the lowered code either as `"parallel"` or `"reduction"`. The presence of a `"parallel"` indicates that a loop is safe to parallelize. For parallelizable loops, lowering generates `scf.parallel` rather than `scf.for` operations. The broader MLIR ecosystem provides lowering paths for `scf.parallel` loops that target Nvidia GPUs, AMD GPUs, OpenMP, and OpenACC. The Computation API cannot naively generate code for any of those targets.

Line 20: This line sets the execution schedule. The provided execution schedule should be a string. Internally, the SPF dialect constructs a Computation API `Stmt` object. The SPF dialect passes the `executionSchedule` attribute as the `executionScheduleStr` argument to the `Stmt` constructor as is shown on lines 7 and 17 in the Computation API example in Fig: 3.3.

Line 21: This line sets the iteration space. Again, the SPF dialect uses this directly in the corresponding Computation API constructor argument.

Line 22: This line would set a list of transformation relations if there were any. The next section will introduce transformations to the Jacobi example.

Line 23: This line ends the attributes and gives the type of the operation. The type signature states that this operation takes 5 arguments, 3 of type `index` and 2 of type `memref<10xf64>`, and has no results.

Lines 24-44: These lines contain the second `spf.statement` operation. This operation largely mirrors the first with two notable differences:

- The C example in Fig: 3.1 has one outer loop over `t` containing two inner loops over `x`. For any `t`, all iterations of `x` for the first loop will execute before any iterations of `x` for the second loop. While C uses control structures, SPF enforces this ordering through execution schedules. The first statement's execution schedule (set on line 20: `"[t,x]->[t,0,x]"`) produces execution space tuples which for any `t` sort all iterations of `x` above execution space tuples produced by the second statement's execution schedule (set on line 20: `"[t,x]->[t,1,x]"`).
- The first inner loop body in the C example in Fig: 3.1 reads from `B` and writes to `A`. The second inner loop body reads from `A` and writes to `B`. The first statement represents executions of the first loop body. The first statement takes `%B` as input and `%A` as output (seen on line 2). The second statement represents executions of the second loop body. the second statement takes `%A` as input and `%B` as output (seen on line 24).

Line 45: This line closes the `spf.computation` operations region opened in **Line 1** and gives the type for the `spf.computation` which takes no arguments and returns no results.

3.2 Transformations

The SPF abstraction exists to make transformations easy to apply and check for legality. Specifically, SPF enables transformations to improve *data access patterns*. Computers require many more cycles to load data from memory than to do arithmetic operations on that data [19]. The memory hierarchy combats this problem by building layers of progressively faster but smaller memory caches. SPF enables transformations that change the order in which steps of a computation are done. That order matters because each step accesses data, and the order that data is accessed (i.e. the computation's data access pattern) determine at what level of the memory hierarchy the data is likely to be found at.

3.2.1 SPF Transformations In The Jacobi Example

SPF transformations can significantly improve data access patterns in the Jacobi example in Fig: 3.1. The example suffers from poor data locality. When the first inner loop writes data, that data will be cached. But for any write, there is a high probability the data will be pushed out of cache by further progress of the first inner loop before the second inner loop can use that data. A transformation called fusion can push the second loop's data read closer to the first's write, increasing the likelihood that data is cached.

```
1 for (int i = 0; i < 100; ++i) {
2     b[i] = f(a[i]);
3 }
4 for (int i = 0; i < 100; ++i) {
5     c[i] = g(b[i]);
6 }
```

Figure 3.5: Fusion Example: Before

```
1 for (int i = 0; i < 100; ++i) {
2     b[i] = f(a[i]);
3     c[i] = g(b[i]);
4 }
```

Figure 3.6: Fusion Example: After

Loop fusion combines two or more loops into one. If two loop bodies access similar data, fusing the loops increases the likelihood repeated accesses are served by cache. Also, after fusion a compiler may be able to remove temporary storage (such as an array) for values accessed by multiple loops. Fusing the two loops in Fig: 3.5 produces the code in Fig: 3.8.

The two inner loops in the Jacobi example cannot be directly fused. The first execution of the second inner loop body reads data written by the first two executions of the first loop body. If the loops were directly fused, the first execution of the second inner loop body would happen after only one execution of the first. But, if we offset the first inner loop relative to the second inner loop, then the loops can be fused. This transformation is known as a shift.

SPF applies the needed shift to the first statement `S0` (which stands in for the first inner loop body) with the following relation (where underscores are used for unused variables):

```
1    for (int t = 1; t <= ub_T / 2; ++t) {
2      for (int x = lb_x-1; x <= ub_x-1; ++x)
3 S0:    A[x] = (B[x] + B[x + 1] + B[x + 2]) / 3;
4
5      for (int x = lb_x; x <= ub_x; ++x)
6 S1:    B[x] = (A[x - 1] + A[x] + A[x + 1]) / 3;
7    }
```

Figure 3.7: Jacobi C Example After Skew Transformation

$$\{[t, \_, x, \_\_] \rightarrow [t, 0, x - 1, 0]\}$$

SPF applies this transformation to the execution schedule tuples for `S0`. Note, the execution schedule tuples for `S0` are produced by applying the original execution schedule to `S0`'s iteration space. After the shift transformation, C code generation produces the code in Fig: 3.7. As the transformation is applied to the first statement `S0`, the transformation shifts first inner loop while leaving the second inner loop unchanged. The shift transformation alone won't improve data access patterns.

The shift transformation does enable a fusion transformation. SPF applies the following relation to statement `S1` to fuse the two inner loops:

$$\{[t, \_, x, \_\_] \rightarrow [t, 0, x, 1]\}$$

The fusion relation removes the 1 at index 1 of the left hand side of the execution schedule (discussed in section: 3.1.2) which forced all inner loop executions of `S0` above `S1`. The transformation still ensures that given the same `x` and `t`, execution tuples of `S1` will sort after execution tuples of `S0`. That ordering is still required. The

shift transformation only bumps executions of S0 up by one. The first execution of S1 requires two executions of S0 before it can read valid data.

```
1    for(t = 1; t <= ub_T; t++) {
2 S0:  A[lb_x] = (B[lb_x - 1] + B[lb_x] + B[lb_x + 1]) / 3;
3      for(x = lb_x; x <= ub_x-1; x++) {
4 S0:    A[x + 1] = (B[x] + B[x + 1] + B[x + 2]) / 3;
5 S1:    B[x] = (A[x - 1] + A[x] + A[x + 1]) / 3;
6      }
7 S1:  B[ub_x] = (A[ub_x - 1] + A[ub_x] + A[ub_x + 1]) / 3;
8    }
```

Figure 3.8: Jacobi C Example After Fusion Transformation

After the fusion transformation, the Jacobi example has improved data locality. After fusion, C code generation produces the code in Fig: 3.8. Executions of statement S1 read elements written to A by S0 almost immediately after they are written. The increased temporal locality virtually ensures that elements written to A by S0 will be in cache when read by S1. A further compiler optimization may even be able to remove the A array entirely.

3.2.2 Transformations In The SPF Dialect

In the MLIR front end, a user would apply the transformations to the MLIR Jacobi example in Fig: 3.4 using the transforms attribute. The attribute takes a string representation of relations. Fig: 3.9 shows an abbreviated version of the MLIR Jacobi example in Fig: 3.4 with the shift and fuse transformations set. The string representation of shift transformation on line 6 creates a new variable c to hold the result of x-1. A quirk in the Computation API parser requires an extra variable to be created; the variable doesn't have any special meaning.

```
"spf.computation"() ({
    "spf.statement"(%ub_T_div_2, %lb_x, %ub_x, %B, %A) ({
        ...
    })  {  reads = [
                ...
           transforms = ["{[t,a,x,b]->[t,0,c,0]:c=x-1}"]
        }:(index,index,index,memref<10xf64>,memref<10xf64>)->()
    "spf.statement"(%ub_T_div_2, %lb_x, %ub_x, %A, %B) ({
        ...
    })  {  reads = [
                ...
           transforms = ["{[t,a,x,b]->[t,0,x,1]}"]
        }:(index,index,index,memref<10xf64>,memref<10xf64>)->()
}): () -> ()
```

Figure 3.9: MLIR SPF Jacobi

Internally, the SPF dialect applies the transformations to its Computation API representation using the `addTransform` method on the `Computation` class. In turn, the Computation API applies the transformation to its internal representation. The next section details how the Computation API's representations are used to generate executable code.

3.3 Lowering

This section describes *lowering*, the process of generating executable code from the SPF dialect. The MLIR SPF dialect we've shown so far enables transformations but it must be lowered to assembly language before execution. Existing SPF front ends lower to C, then use existing C compilers to produce assembly. To produce executable code, the SPF dialect generates lower level MLIR dialects. In turn, the dialects targeted by SPF lowering have lowering passes of their own. Lowering to

executable code requires a *pipeline* of lowing passes, starting with the SPF dialect.

SPF dialect lowering can be broken up into a series of separate tasks: generating loops, generating statement execution, and uninterpreted function call lookup. Each piece is contained within the `convert-spf-to-loops` pass created in this work, all other passes discussed in this section were not created as part of this work. After those tasks, lowering has removed all SPF concepts. Lowering passes from the wider MLIR ecosystem then further lower the generated MLIR to executable code. Below, we discuss each step of SPF dialect lowering in detail, and touch on the pipeline that lowers from the generated code to executable code.

3.3.1 Generating Loops

The SPF dialect builds upon the Computation API, and leverages it heavily for loop generation. From the MLIR SPF interface discussed in Section: 3.1.4, SPF dialect internally creates a Computation API representation. The SPF dialect primarily uses an exiting loop generation infrastructure within the Computation API representation. The Computation API is a single entry point for a variety of sparse polyhedral tools. Internally, it relies on another tool CodeGen+ [5, 13] to provide loop generation through an algorithm called polyhedral scheduling.

The Computation API builds on the code generation system CodeGen+. We first discuss, and give an example of, CodeGen+'s polyhedral scheduling algorithm before describing its integration with SPF MLIR dialect. Before giving an example of the underlying algorithm, some preliminaries must first be discussed. CodeGen+ uses the Omega [25] system to provide Presburger Arithmetic manipulation capabilities. The

constraint satisfaction algorithm relies on a few key set and relation manipulation operations: Project, Gist, and Hull.

Project: This operation eliminates a variable from all equations and inequalities using an approach based on Fourier-Motzkin. The idea is to eliminate a variable by projecting its constraints onto the rest of the system. During this process, `Project` may generate additional constraints.

$$\texttt{Project}(\{x \leq y + 10 \wedge y \leq 15 \wedge y \geq -x + 20\}, y) = \{5 \leq x \leq 25\}$$

Gist: Gist takes two relations A and B and extracts the constraints in A not already represented in B.

$$\texttt{Gist}(\{i > 10 \wedge j > 10\}, \{j > 10\}) = \{[i] : i > 10\}$$

Hull: Hull takes a large set of constraints and returns a (potentially) smaller set that must include all the lattice points in the polyhedron formed by the original.

$$\texttt{Hull}(\{0 \leq i \leq 10\} \cup \{0 \leq i \leq 100\}) = \{i \leq 100\}$$

The CodeGen+ polyhedral scanning algorithm has two phases. The first phase takes the constraints provided in Presburger Arithmetic by SPF or some other tool, and uses `Project` to construct an initial AST with the constraints that need to be fulfilled at each node. The second uses `Hull` and `Gist` to compute bounds for

```
1     for(int i = 0; i < I; i++) {
2         for(int j = 0; j < J; j++) {
3 S0:         a[i][j] = b[i][j]
4         }
5     }
```

Figure 3.10: Simple Example

generated loops and ensures that all constraints are met. The second phase may modify the AST to fulfill the constraints. Amongst other things, it may add guard nodes which will yield `if` blocks in generated code.

For an example, take the code in Fig: 3.10. The code is represented using the polyhedral model and will have the following constraints for statement `S0`

$$\{[i,j] : 0 \leq i < I \wedge 0 \leq j < J\}.$$

Generated code will loop over the points in this set in lexicographic order. A compiler or user of a polyhedral system might apply a loop interchange transformation relation such as

$$\{[i,j] \rightarrow [i',j'] : i' = j \wedge j' = i\}.$$

The execution schedule tuples after applying this transformation will be

$$\{[j,i] : 0 \leq i < I \wedge 0 \leq j < J\}.$$

The first phase of CodeGen+'s polyhedral scanning algorithm will create two AST nodes with initial constraints by projecting out each dimension in turn.

Table: 3.1 shows the initial generated constraints. The initial constrains can also

Table 3.1: Initial Constraints

level	constraints
0	$\{0 \leq j < J\}$
1	$\{0 \leq i < I \wedge 0 \leq j < J\}$

be read as an AST with two levels. The first level of the AST is required to fulfill the constraints $\{0 <= j < J\}$, and the second is required to fulfill $\{0 <= i < I \wedge 0 <= j < J\}$.

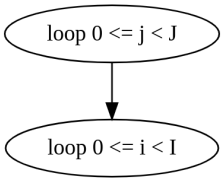

Figure 3.11: AST

```
1 for(int j = 0; j < J; j++) {
2     for(int i = 0; I < n; i++) {
3         a[i][j] = b[i][j]
4     }
5 }
```

Figure 3.12: AST

The final step, CodeGen+ computes bounds for the AST. As the first level establishes the constraint $\{0 <= j < J\}$, the second phase will not generate code that loops over the full polyhedron represented by $\{0 <= i < I \wedge 0 <= j < J\}$ during the second phase. `Gist` will determine that the condition on i has already been met at a different level, and the second phase will only generate the bounds that are required. CodeGen+ will then generate C code in Fig: 3.12 from the AST in Fig: 3.11.

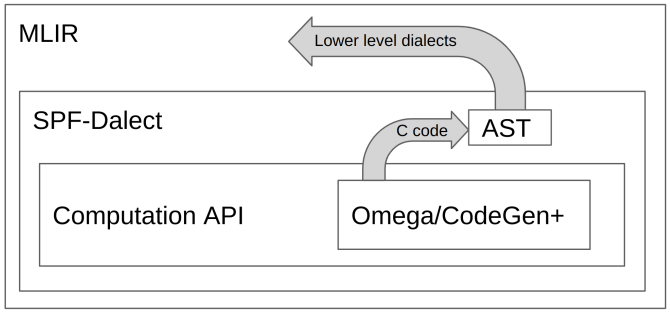

Figure 3.13: Generating MLIR

Initially, the SPF MLIR front end generated MLIR code from the CodeGen+ AST. But with anything but simple examples, several implementation decisions in CodeGen+ make the underlying AST very difficult to detach from the C generation. Now, the MLIR front end parses CodeGen+'s generated C code into an AST more amenable to MLIR code generation, and uses that AST to generate MLIR code. Fig: 3.13 shows the flow.

The parser produces the AST in Fig: 3.14 (AST displayed in string output format) from the transformed SPF dialect Jacobi example presented in Fig: 3.9. As discussed in Section: 3.1.4, lowering translates the AST's `loop` nodes straightforwardly into `scf.for` or `scf.parallel` operations (`scf` dialect discussed in Section: 2.2). After this step lowering has produced loops and knows where statements need to executed. To remove all SPF concepts in the Jacobi example, lowering needs to produce code to execute those statements.

```
loop{inductionVar:t1,
     start:int{val:1},
     stop:symbol{symbol:ub_T, increment:1},
     step:1,
     body:[call{statementNumber:0,
                args:[symbol{symbol:t1},
                      int{val:0},
                      symbol{symbol:lb_x, increment:-1},
                      int{val:0}]
               },
           loop{inductionVar:t3,
                start:symbol{symbol:lb_x},
                stop:symbol{symbol:ub_x},
                step:1,
                body:[
                      call{statementNumber:0,
                           args:[symbol{symbol:t1},
                                 int{val:0},
                                 symbol{symbol:t3},
                                 int{val:0}]
                          },
                       call{statementNumber:1,
                            args:[symbol{symbol:t1},
                                  int{val:0},
                                  symbol{symbol:t3},
                                  int{val:1}]
                          }]
               },
           call{statementNumber:1,
                args:[symbol{symbol:t1},
                      int{val:0},
                      symbol{symbol:ub_x},
                      int{val:1}]
               }]
    }
```

Figure 3.14: Jacobi Example: AST Parsed From Generated C

3.3.2 Generating Statement Execution

Lowering generates code to execute a statement in four steps.

1. Lowering generates code to recover the iteration space from the execution space.
2. Lowering generates load operations from the inputs to an operation.
3. Lowering in-lines the statement.
4. Lowering generates store operations from the outputs.

```
1 for(int i=0; i<I; i++)
2   for(int j=0; j<J; j++)
3     A[i,j] = B[i,j]
```

Figure 3.15: Double For Loop

```
1 for(int j=0; j<J; j++)
2   for(int i=0; i<I; i++)
3     A[i,j] = B[i,j]
```

Figure 3.16: Double For Loop After Loop Permutation

Table 3.2: Before Transformation

Execution Space Tuple	Iteration Space Tuple
$[0, 0]$	$[0, 0]$
$[0, 1]$	$[0, 1]$
$[0, 2]$	$[0, 2]$
$[1, 0]$	$[1, 0]$
$[1, 1]$	$[1, 1]$
$[1, 2]$	$[1, 2]$

To motivate why the first step is necessary, take a simple double for loop in Fig: 3.15. In SPF this would have iteration space

$$\{[i, j] : 0 \leq i < I \wedge 0 \leq j < J\}$$

and execution schedule

$$\{[i, j] \rightarrow [i, j]\}.$$

Table: 3.2 shows the tuples produced and sorted theoretical execution with $I = 1$ and $J = 2$. If a user applies a loop permutation transformation

$$\{[i, j] \rightarrow [j, i]\}$$

to produce code such as that in Fig: 3.16 the execution schedule tuples will no longer correspond to the iteration space tuples. Table: 3.2 shows the tuples produced and sorted after this transformation.

Table 3.3: After Transformation

Execution Space Tuple	Iteration Space Tuple
$[0, 0]$	$[0, 0]$
$[0, 1]$	$[1, 0]$
$[1, 0]$	$[0, 1]$
$[1, 1]$	$[1, 1]$
$[2, 0]$	$[0, 2]$
$[2, 1]$	$[1, 2]$

CodeGen+, in Section: 3.3.1, generates loops and statement calls on the execution space tuples. The transformed code in Fig: 3.15 doesn't index into `A` at $[2, 1]$. An

optimization causing the code to index into `A` at $[2, 1]$, as Table: 3.3 shows the execution schedule tuples do, would be incorrect. In order for the permutation transformation to be correct, something has to map the execution space back to the iteration space.

The loop induction variables keep their original names when permuted between Fig: 3.15 and Fig: 3.16, allowing variable binding to ensure that the statement is called correctly. code generated by CodeGen+ won't have the variable binding the C example does'. But by inverting the execution schedule, the SPF dialect can use the Computation API to create a function from the execution space to the iteration space.

The SPF dialect stores the relation from execution space to iteration space as an `affine_map` (discussed in Section: 3.1.4). Given an `affine_map`, existing MLIR infrastructure can generate code to apply that map. To accomplish step 1, lowering generates code with the `affine_map` infrastructure recover the iteration space from the loop induction variables running over the execution space generated in the last Section: 3.3.1.

```
scf.for %arg0 = %c1 to %1 step %c1 {
    scf.for %arg1 = %c1 to %c9 step %c1 {
      %2 = arith.addi %arg1, %c1 : index
      %3 = memref.load %B[%2] : memref<10xf64>
      %4 = memref.load %B[%arg1] : memref<10xf64>
      %c-1 = arith.constant -1 : index
      %5 = arith.addi %arg1, %c-1 : index
      %6 = memref.load %B[%5] : memref<10xf64>
```

Figure 3.17: Jacobi Lowering Step 2

Given variables in the iteration space from step 1, lowering can straight forwardly

accomplish step 2. A user of the SPF dialect provides read access relations (discussed in Section: 3.1.4). Read access relations take iteration space variables as input and output indices at which reads should be done. Lowering again leverages the existing `affine_map` infrastructure to compute indices from a read map, then generates a load-/store at those indices. For example, the read maps for `%B` from the Jacobi example in Fig: 3.4 are: `affine_map<(t, x) -> (x+1)>`, `affine_map<(t, x) -> (x)>`, and `affine_map<(t, x) -> (x-1)>`. For step 2 lowering generates lines 3-8 in Fig: 3.17; lines 1-2 are generated during loop generation in Section: 3.3.1. Lines 3, 6, and 7 compute indices, and lines 4, 5, and 8 do the required loads (the `memref` and `arith` dialects were discussed in Section: 2.2).

```
1 ^stmt(%B_x_plus_one: f64, %B_x: f64, %B_x_minus_one: f64):
2       %0 = arith.addf %B_x_plus_one, %B_x : f64
3       %1 = arith.addf %0, %B_x_minus_one : f64
4       %2 = arith.divf %1, %f3 : f64
5       "spf.yield"(%2): (f64) -> ()
```

Figure 3.18: Jacobi Statement

```
1  scf.for %arg0 = %c1 to %1 step %c1 {
2      scf.for %arg1 = %c1 to %c9 step %c1 {
3        %2 = arith.addi %arg1, %c1 : index
4        %3 = memref.load %B[%2] : memref<10xf64>
5        %4 = memref.load %B[%arg1] : memref<10xf64>
6        %c-1 = arith.constant -1 : index
7        %5 = arith.addi %arg1, %c-1 : index
8        %6 = memref.load %B[%5] : memref<10xf64>
9        %0 = arith.addf %3, %4 : f64
10       %1 = arith.addf %0, %6 : f64
11       %2 = arith.divf %1, %f3 : f64
12       "spf.yield"(%2): (f64) -> ()
```

Figure 3.19: Jacobi Lowering: Step 3

To accomplish step 3, lowering only needs to in-line the statement kernel into the generated code. The MLIR SPF interface expects the statement arguments to be filled in by generated reads, and now those reads have been generated. For example, step 3 will in line the first statement Jacobi example without transforms in Fig: 3.18 into the code produced so far in Fig: 3.17 producing the result in Fig: 3.19.

```
1  scf.for %arg0 = %c1 to %1 step %c1 {
2      scf.for %arg1 = %c1 to %c9 step %c1 {
3        %2 = arith.addi %arg1, %c1 : index
4        %3 = memref.load %B[%2] : memref<10xf64>
5        %4 = memref.load %B[%arg1] : memref<10xf64>
6        %c-1 = arith.constant -1 : index
7        %5 = arith.addi %arg1, %c-1 : index
8        %6 = memref.load %B[%5] : memref<10xf64>
9        %7 = arith.addf %3, %4 : f64
10       %8 = arith.addf %0, %6 : f64
11       %9 = arith.divf %1, %f3 : f64
12       memref.store %9, %A[%arg1] : memref<10xf64>
13     }
```

Figure 3.20: Jacobi Lowering: Step 4

Step 4 proceeds in a similar fashion to step 2. At this point lowering has all the requirements for building a write: the user provides a write `affine_map` which can calculate indexes into storage, the statement provides what should be written with the `scf.yield` operation, step 1 provides the iteration space input to the write map. Lowering generates indexing variables from the write map again using the `affine_map` infrastructure. For the variables a statement provides via `scf.yield`, lowering generates stores into the associated storage location at the generated indices. After step 4, lowering produces the code in Fig: 3.20. Line 12 contains the code step 4 generated with write map `affine_map<(t, x) -> (x)>` on `spf.statement` output

%A.

For reference, the full output of lowering for the transformed Jacobi kernel in Fig: 3.9 is included in Fig: 3.21.

```
scf.for %arg0 = %c1 to %1 step %c1 {
  %2 = arith.addi %c0, %c2 : index
  %3 = memref.load %alloc_2[%2] : memref<10xf64>
  %4 = arith.addi %c0, %c1 : index
  %5 = memref.load %alloc_2[%4] : memref<10xf64>
  %6 = memref.load %alloc_2[%c0] : memref<10xf64>
  %7 = arith.addf %3, %5 : f64
  %8 = arith.addf %7, %6 : f64
  %9 = arith.divf %8, %cst_0 : f64
  memref.store %9, %alloc[%4] : memref<10xf64>
  scf.for %arg1 = %c1 to %c8 step %c1 {
    %16 = arith.addi %arg1, %c2 : index
    %17 = memref.load %alloc_2[%16] : memref<10xf64>
    %18 = arith.addi %arg1, %c1 : index
    %19 = memref.load %alloc_2[%18] : memref<10xf64>
    %20 = memref.load %alloc_2[%arg1] : memref<10xf64>
    %21 = arith.addf %17, %19 : f64
    %22 = arith.addf %21, %20 : f64
    %23 = arith.divf %22, %cst_0 : f64
    memref.store %23, %alloc[%18] : memref<10xf64>
    %24 = memref.load %alloc[%18] : memref<10xf64>
    %25 = memref.load %alloc[%arg1] : memref<10xf64>
    %c-1 = arith.constant -1 : index
    %26 = arith.addi %arg1, %c-1 : index
    %27 = memref.load %alloc[%26] : memref<10xf64>
    %28 = arith.addf %24, %25 : f64
    %29 = arith.addf %28, %27 : f64
    %30 = arith.divf %29, %cst_0 : f64
    memref.store %30, %alloc_2[%arg1] : memref<10xf64>
  }
  %10 = memref.load %alloc[%c9] : memref<10xf64>
  %11 = memref.load %alloc[%c8] : memref<10xf64>
  %c7 = arith.constant 7 : index
  %12 = memref.load %alloc[%c7] : memref<10xf64>
  %13 = arith.addf %10, %11 : f64
  %14 = arith.addf %13, %12 : f64
  %15 = arith.divf %14, %cst_0 : f64
  memref.store %15, %alloc_2[%c8] : memref<10xf64>
}
```

Figure 3.21: Lowered Transformed Jacobi Example

3.3.3 Uninterpreted Function Call Lookup

The final lowering step concerns uninterpreted functions. Uninterpreted functions (discussed in Section: 2.1) provide an abstraction allowing SPF to model sparse codes. The Jacobi example used thus far does not contain any uninterpreted functions. To motivate this lowering step, we provide an example that does: sparse MTTKRP (Matricized Tensor Times Khatri-Rao Product).

```
for (uint64_t i = 0; i < I; i++) {
    for (uint64_t k = 0; k < K; k++) {
        for (uint64_t l = 0; l < L; l++) {

            // loop over j dimension of A matrix
            for (uint64_t j = 0; j < J; j++) {
                A[i, j] += B[i, k, l] * D[l, j] * C[k, j];
            }
        }
    }
}
```

Figure 3.22: Dense MTTKRP Kernel In C

```
// loop over number of non zero
for (uint64_t i_nnz = 0; i_nnz < nnz; i_nnz++) {
    // Read coordinates out of b matrix stored in COO format
    uint64_t i = BCoords0[i_nnz];
    uint64_t k = BCoords1[i_nnz];
    uint64_t l = BCoords2[i_nnz];
    // Read value out of b matrix stored in COO format
    double BIKL = BVals[i_nnz];

    // loop over j dimension of A matrix
    for (uint64_t j = 0; j < J; j++) {
        A[i, j] += BIKL * D[l, j] * C[k, j];
    }
}
```

Figure 3.23: Sparse COO MTTKRP Kernel In C

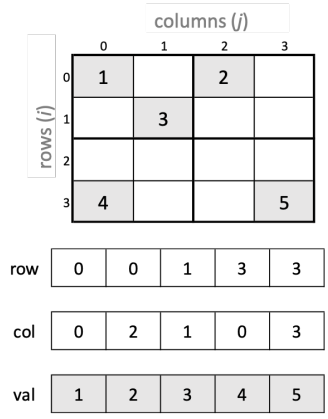

Figure 3.24: COO sparse format

MTTKRP is the bottleneck in various algorithms such as the Canonical Polyadic Decomposition (CPD) [14]. CPD is tensor generalization of the Singular Value Decomposition (SVD) for matrices, it approximates a tensor with a sum of rank-1 matrices (vectors). A decomposition factors a tensor into the product of several smaller constituent parts. The SVD and CPD arrange factored parts in order of magnitude of effect (as determined by singular value). Principal Component Analysis and other important techniques rely on CPD decompositions. MTTKRP in dense C code is shown in Fig: 3.22, and sparse in Fig: 3.23. The sparse code uses the COO sparse tensor format detailed in Fig: 3.24. MTTKRP can be expressed in index notation for a dense tensor as

$$A_{ij} = B_{ikl} \cdot D_{lj} \cdot C_{kj}.$$

An SPF representation of sparse MTTKRP has the following iteration space

$$\{[j, i\_k, i, l] : 0 \leq i\_k < nnz \wedge i = UFi(i\_k) \wedge k = UFk(i\_k) \wedge l = UFl(i\_k) \wedge 0 \leq j < J\}.$$

Note the use of uninterpreted functions to represent the irregular memory access into the COO coordinate arrays. An SPF representation of sparse MTTKRP has the following execution schedule:

$$\{[j, i\_k, i, k, l] \rightarrow [j, i\_k, i, k, l]\}$$

The data reads would be the following:

$$\{A\{[j, i\_k, i, k, l] \rightarrow [j, i\_k, i, k, l]\}\},$$

$$\{B\{[j, i\_k, i, k, l] \rightarrow [j, i\_k, i, k, l]\}\},$$

$$\{C\{[j, i\_k, i, k, l] \rightarrow [j, i\_k, i, k, l]\}\},$$

$$\{D\{[j, \_k, i, k, l] \rightarrow [j, i\_k, i, k, l]\}\}.$$

Data writes would be the following:

$$\{A\{[j, i\_k, i, k, l] \rightarrow [j, i\_k, i, k, l]\}\}.$$

Loop generation in Section: 3.3.1 produces calls to uninterpreted functions (UFs) as well as statements. Inside CodeGen+ UF calls behave similarly to loops. As discussed in Section: 3.3.1, CodeGen+ fulfills most constrains, such as $0 \leq j < J$ from the sparse MTTKRP example, with a loop. But if while building the AST, CodeGen+

```
"spf.computation"() ({
    "spf.statement"(%NNZ, %J, %argb_coord_0, %argb_coord_1,
                    %argb_coord_2, %argb_values, %argc,
                    %argd, %arga) ({
    ^bb0(%b_i_k_l : f64, %d_l_j : f64,
         %c_k_j : f64, %a_i_j : f64):
    %0 = arith.mulf %b_i_k_l, %d_l_j : f64
    %1 = arith.mulf %0, %c_k_j : f64
    %2 = arith.addf %1, %a_i_j : f64
    "spf.yield"(%2) : (f64) -> ()
    }) {
        reads = [
            [affine_map<(z, i, k, l, j) -> (z)>],
            [affine_map<(z, i, k, l, j) -> (k, j)>],
            [affine_map<(z, i, k, l, j) -> (l, j)>]
        ],
        writes = [
            [affine_map<(z, i, k, l, j) -> (i, j)>]
        ],
        // symbols, ufInputs, inputs, outputs
        operand_segment_sizes = array<i32: 2,3,3,1>,
        symbolNames = ["NNZ", "J"],
        iteratorTypes = ["reduction", "reduction",
                         "reduction", "reduction", "parallel"],
        executionSchedule = "{[z,i,k,l,j]->[z,i,k,l,j]}",
        iterationSpace = "{[z,i,k,l,j]: 0<=z<NNZ and
                                        i=UFi(z) and
                                        k=UFk(z) and
                                        l=UFl(z) and
                                        0<=j<J}",
        transforms = []
    } : (index, index,
         memref<?xindex>, memref<?xindex>,
         memref<?xindex>, memref<?xf64>,
         memref<?x?xf64>, memref<?x?xf64>,
         memref<?x?xf64>) -> ()
}) : () -> ()
```

Figure 3.25: Sparse COO MTTKRP Kernel In MLIR

finds a constraint that must be fulfilled with a UF call, such as $l = UFl(i\_k)$ from sparse MTTKRP, it generates a UF call rather than a loop.

```
1  loop{inductionVar:t1,
2       start:int{val:0},
3       stop:symbol{symbol:NNZ},
4       step:1,
5       body:[ufAssignment{inductionVar:t2,
6                          ufName: UFi,
7                          args:[symbol{symbol:t1}]},
8             ufAssignment{inductionVar:t3,
9                          ufName: UFk,
10                         args:[symbol{symbol:t1}]},
11            ufAssignment{inductionVar:t4,
12                         ufName: UFl,
13                         args:[symbol{symbol:t1}]},
14            loop{inductionVar:t5,
15                 start:int{val:0},
16                 stop:symbol{symbol:J},
17                 step:1,
18                 body:[call{statementNumber:0,
19                            args:[symbol{symbol:t1},
20                                  symbol{symbol:t2},
21                                  symbol{symbol:t3},
22                                  symbol{symbol:t4},
23                                  symbol{symbol:t5}]},
24                       ]},
25            ]}
```

Figure 3.26: Sparse COO MTTKRP AST

From the sparse MTTKRP example in Fig: 3.25 CodeGen+ and the C parser produce the AST in Fig: 3.26. We have already discussed how the `loop` and `call` nodes are lowered, this section is concerned with the `ufAssignment` nodes. `ufAssignment` nodes represent a call to and assignment from a UF. Lowering will eventually produce something similar to line 4 in the C example (`uint64_t i = BCoords0[i_nnz];`) in Fig: 3.23 from the `ufAssignment` on lines 5-7 in Fig: 3.26.

Lowering generates actual function calls in place of uninterpreted function calls. The SPF dialect representation of MTTKRP in Fig: 3.25 has 3 `ufInputs` argu-

```
1 func.func private @UFi(%uf_argb_coord_0 : memref<?xindex>,
2                        %uf_argb_coord_1 : memref<?xindex>,
3                        %uf_argb_coord_2 : memref<?xindex>,
4                        %z: index)-> index {
5     %i = memref.load %uf_argb_coord_0[%z] : memref<?xindex>
6     return %i : index
7 }
```

Figure 3.27: Example Uninterpreted Function

ments (shown on line 18). As discussed in Section: 3.1.4, the `ufInput` argument grouping stores extra arguments to uninterpreted function calls. Upon finding a `ufAssignment` node, lowering looks for an MLIR function taking the `ufInput` arguments as well as any arguments to the UF the AST identifies. For example when encountering the `ufAssignment` on lines 5-7 in Fig: 3.26, lowering will search the current symbol table for a function such as that in Fig: 3.27. If found, lowering generates a call operation `%0 = func.call @UFi(%arg2, %arg3, %arg4, %arg9) : (memref<?xindex>, memref<?xindex>, memref<?xindex>, index) -> index`. If not found, lowering returns an error.

For reference the full output of lowering for the sparse MTTKRP kernel in Fig: 3.25 is included in Fig: 3.21.

Lowering has now replaced all SPF dialect operations with operations from other MLIR dialects. Lowering is complete. But, the user doesn't yet have executable code. Lowering delegates the remaining compilation flow to a further *existing* lowering pipeline.

```
scf.for %arg9 = %c0 to %arg0 step %c1 {
  %0 = func.call @UFi(%arg2, %arg3,
                      %arg4, %arg9) : (memref<?xindex>,
                                       memref<?xindex>,
                                       memref<?xindex>,
                                       index) -> index
  %1 = func.call @UFk(%arg2, %arg3,
                      %arg4, %arg9) : (memref<?xindex>,
                                       memref<?xindex>,
                                       memref<?xindex>,
                                       index) -> index
  %2 = func.call @UFl(%arg2, %arg3,
                      %arg4, %arg9) : (memref<?xindex>,
                                       memref<?xindex>,
                                       memref<?xindex>,
                                       index) -> index
  scf.parallel (%arg10) = (%c0) to (%arg1) step (%c1) {
    %3 = memref.load %arg5[%arg9] : memref<?xf64>
    %4 = memref.load %arg6[%1, %arg10] : memref<?x?xf64>
    %5 = memref.load %arg7[%2, %arg10] : memref<?x?xf64>
    %6 = memref.load %arg8[%0, %arg10] : memref<?x?xf64>
    %7 = arith.mulf %3, %5 : f64
    %8 = arith.mulf %7, %4 : f64
    %9 = arith.addf %8, %6 : f64
    memref.store %9, %arg8[%0, %arg10] : memref<?x?xf64>
  }
}
```

Figure 3.28: Lowered Sparse MTTKRP Example

3.3.4 Pipelines

```
spf-opt
  -convert-spf-to-loops
  -lower-affine
  -gpu-map-parallel-loops
  -convert-parallel-loops-to-gpu
  -lower-affine
  -convert-vector-to-scf
  -convert-scf-to-cf
  -func-bufferize
  -arith-bufferize
  -finalizing-bufferize
  -gpu-kernel-outlining
  | spf-opt -pass-pipeline='builtin.module(gpu.module(strip-debuginfo,
                                                     convert-gpu-to-nvvm,gpu-to-cubin))'
  | spf-opt -gpu-to-llvm
    -convert-vector-to-llvm
    -convert-memref-to-llvm
    -convert-complex-to-standard
    -convert-math-to-llvm
    -convert-complex-to-llvm
    -convert-math-to-libm
    -convert-func-to-llvm
    -reconcile-unrealized-casts
```

Figure 3.29: Compilation Pipeline

Lowering is done with the `spf-opt` tool which is a lightly wrapped version of the MLIR tool `mlir-opt`. The `spf-opt` tool is a version of `mlir-opt` built during this work. The `mlir-opt` tool provides passes for optimization and transformation of MLIR code. Lowering proceeds in a series of passes, each of which may transform the code with an optimization, or lower a dialect to another. A user of `mlir-opt` provides passes and parameters to those passes on the command line. The wrapping

adds the `-convert-spf-to-loops` pass which implements the lowering discussed in Sections: 3.3.1, 3.3.2, and 3.3.3. All passes besides `-convert-spf-to-loops` already existed in MLIR.

While compiling to final machine code, the `spf-opt` tool generates LLVM IR. Technically `spf-opt` produces the MLIR LLVM dialect which is then translated to LLVM IR via the tool `mlir-translate`. LLVM is a lower level compiler that can target CPUs and GPUs [16]. LLVM compiles a restricted set of its common IR with required GPU-vendor specific intrinsics to GPU code [2, 1]. LLVM cannot target CPUs and GPUs from the same code as can be done with MLIR. Fig: 3.29 shows a pipeline that produces LLVM specialized for Nvidia GPUs.

As discussed previously, different pipelines can generate code for a wide variety of targets from the same SPF dialect representation of a kernel. The pipeline in Fig: 3.29 compiles the MTTKRP example from Section: 3.25 to LLVM code specialized for Nvidia GPUs, but with minor modifications to the pipeline it will produce CPU code. Though untested by this work, different pipelines should be able to lower the `scf.parallel` operations produced by `-lower-affine` not only to Nvidia GPU and CPU but also to AMD GPU, OpenMP, and OpenACC.

This work used the `llc` tool to compile LLVM IR to final executable machine code. There are many potential options, for example the `clang` compiler can also compile LLVM IR. The `llc` tool compiles LLVM IR to an object file. The resulting object file can be linked into an existing program. This work linked object files produced from SPF MLIR kernels into a bench-marking harness implemented in C++.

3.4 Performance Evaluation

This section details the performance evaluation of code generated from the MLIR SPF font end. The benchmark suite consists of SPF MLIR implementations of two common sparse scientific kernels: Sparse Matricized Tensor Times Khatri-Rao Product (MTTKRP), and Sparse Tensor Times Matrix (TTM). On CPU, the evaluation benchmarks MLIR implementations of the kernels against implementations produced using the C SPF front end, and implementations from the PASTA benchmark suite [18]. On GPU, as the C SPF front end cannot produce GPU code, the evaluation benchmarks MLIR implementations of the kernels only against implementations from the PASTA benchmark suite. The results show the MLIR implementations have generally competitive performance on CPU and GPU.

3.4.1 Benchmark Suite

```
1  // loop over number of fibers
2  for (uint64_t f = 0; f < Mf; f++) {
3      // loop over items in each fiber
4      for (uint64_t m = fptr[f]; m < fptr[f + 1]; m++) {
5          // read value out of constant dimension of COO storage
6          uint64_t k = xCoordConstant[m];
7          // loop over dimension of U
8          for (uint64_t r = 0; r <= R - 1; r++) {
9              y[f * R + r] += xValues[m] * u[k * R + r];
10         }
11     }
12 }
```

Figure 3.30: Sparse COO TTM Kernel In C

The benchmark suite consists of implementations of two common sparse scientific kernels MTTKRP and TTM.

MTTKRP was discussed in Section: 3.3.3.

TTM is the bottleneck in various algorithms such as the Tucker Decomposition [14]. Similarly to CPD, the Tucker decomposition is a generalization of the Singular Value Decomposition (SVD) for matrices. The Tucker decomposition breaks a tensor into a "core" tensor and several matrices ordered by magnitude of effect (as determined by singular value). Principal Component Analysis and other important techniques rely on Tucker decompositions.

TTM, also known as the n-mode product, is the multiplication of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \ldots \times I_n \times \ldots \times I_N}$ with a matrix $U \in \mathbb{R}^{I_n \times R}$. The n-mode product multiplies each mode-n fiber, obtained by holding all modes of $\mathcal{X}$ constant except n, by the matrix U. Calculating TTM for COO (COO also discussed in Section: 3.3.3) sparse $\mathcal{X}$ requires first pre-processing to calculate M_F, the number of n-mode fibers in $\mathcal{X}$, and f_{ptr} array size M_F storing the beginnings of each n-mode. Fig: 3.30 shows sparse TTM in C code. TTM can be expressed in index notation as

$$\mathcal{Y}_{i_1 \ldots i_{n-1} i_{n+1} \ldots i_N} = \sum_{i_n=1}^{I_n} \mathcal{X}_{i_1 \ldots i_{n-1} i_n i_{n+1} \ldots i_N} \mathcal{U}_{i_n} r.$$

3.4.2 Experimental Setup

The performance evaluation used a GPU node in the R2 cluster at Boise State University. R2 GPU nodes are configured with two Intel Xeon E5-2680 v4 14-core

CPUs running at a base clock of 2.4ghz, and a Max turbo clock of 3.30ghz. The Xeon E5-2680 CPUs have a 35MB L3 cache, 256K L2 cache, and 32K L1 data and instruction caches. GPU nodes are configured with with 192GB of memory split into two NUMA nodes. Each GPU node has two Nvidia Tesla P100 GPUs (the benchmarks used only one GPU) with 12GB of graphics memory.

Pasta and IEGenLib implementations of the benchmark suite were compiled with GCC 10.2.0 and NVCC 11.2 using the O3 flag. SPF dialect implementations of the benchmarks were compiled, as discussed in Section: 3.3.4, with: `spf-opt`, `mlir-translate`, and `llc` with -O3 flag. All LLVM and MLIR tools were built from source on LLVM 16 at sha:`570117b` linked against NVCC 11.2.

Results compare PASTA, IEGenLib, and SPF MLIR implementations of TTM and MTTKRP. Each implementation is benchmarked on a set of 3 dimensional tensors from the FROSTT tensor collection [27]. Results compare average time over five runs.

3.4.3 Results

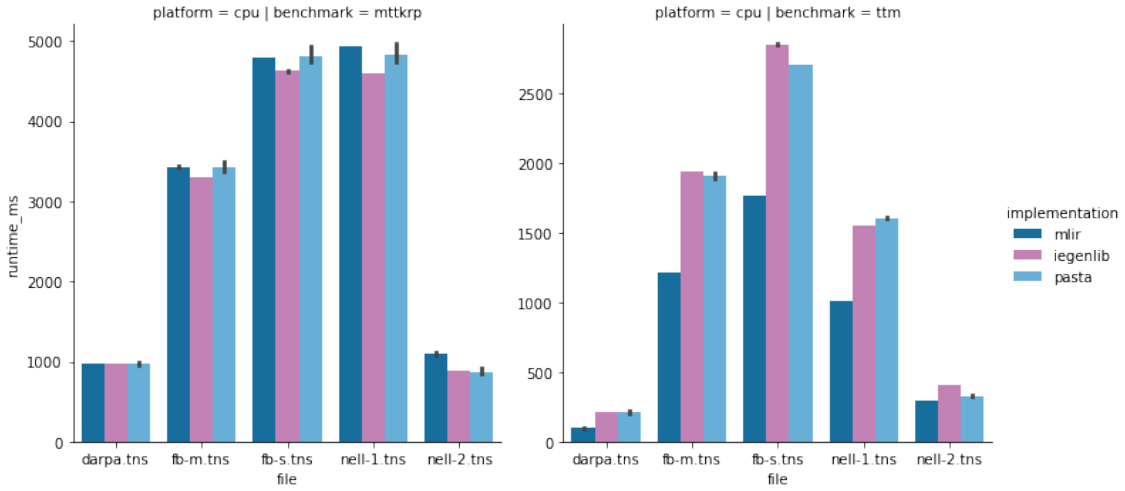

Figure 3.31: CPU Benchmarks

This section presents the results of the performance evaluation. Fig: 3.31 shows the run time in milliseconds of CPU implementations of the benchmarks. The results show generally competitive performance between all implementations.

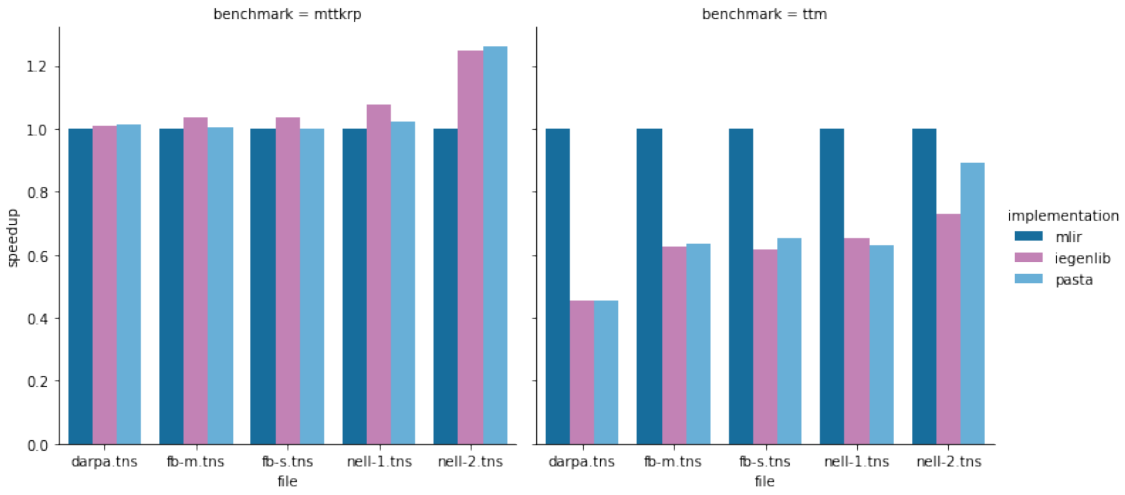

Figure 3.32: CPU Speedup

Fig: 3.32 shows the speedup of each implementations relative to the SPF dialect implementation. In this speedup graph the SPF implementation will always have a value of 1.0. A value higher than 1.0 indicates the competing implementation ran faster than the MLIR implementation, a value below 1.0 indicates the MLIR implementation ran faster than the competing implementations. The results show generally competitive performance between all implementations of MTTKRP. For TTM, the results show that the SPF dialect implementation has an advantage. All implementation's final generated assembly code has very similar characteristics. It's not currently understood what causes the increase for the MLIR implementation.

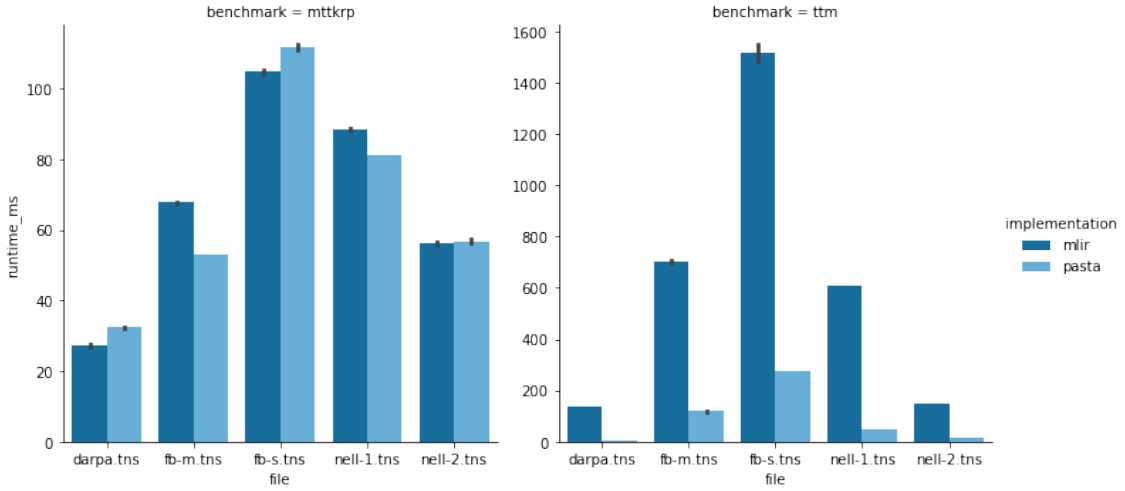

Figure 3.33: GPU Benchmarks

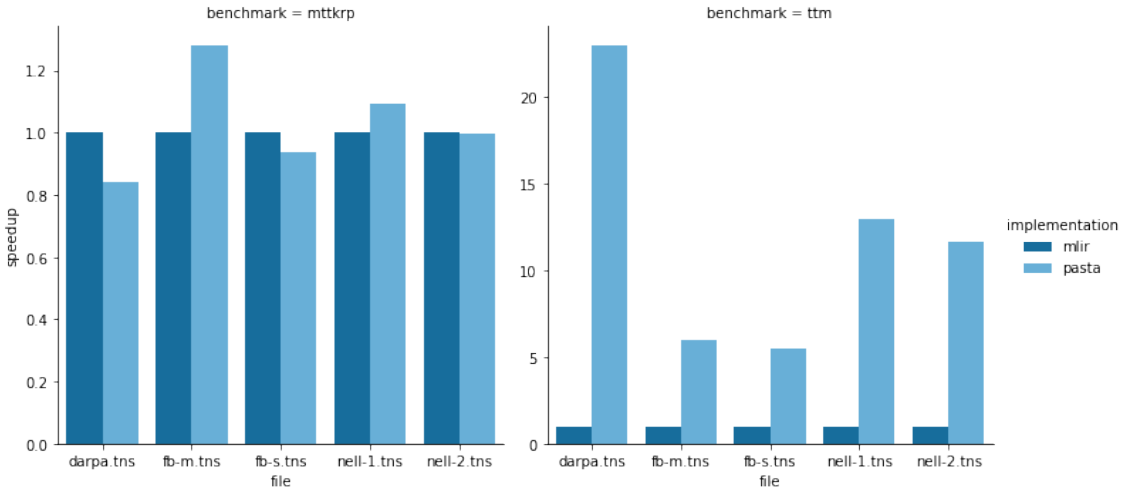

Figure 3.34: GPU Speedup

Fig: 3.33 shows the run time in milliseconds of GPU implementations of the benchmarks. Fig: 3.34 shows the same data as a speedup relative to MLIR implementation. The graph does not rank IEGenLib, as the C SPF front end cannot

produce GPU code. The results show generally competitive performance between MLIR and PASTA for MTTKRP, and generally worse performance MLIR relative to PASTA for TTM. SPF could represent the transformations necessary to turn the CPU version of MTTKRP into the GPU version, but the current implementation lacks the required features. The MLIR SPF dialect GPU implementation for MTTKRP is specialized for GPU, whereas the TTM version is the same as the CPU version just compiled with a different pipeline. Without an implementation that's designed for GPU hardware, the SPF MLIR dialect implementation of TTM can't compete with the PASTA implementation.

CHAPTER 4

RELATED WORK

This work builds on research done on Polyhedral Model tools, the Sparse Polyhedral Framework, and MLIR.

Polyhedral Model tools are used to reason about and optimize codes with affine memory accesses. Examples of such tools include: Polly [7], Pluto [4], Loopy [21], PolyMage [20], and fpl [22]. Pluto built on existing research in polyhedral scheduling, providing a way to optimize generated code for parallelization and locality. Polly provided a way to do polyhedral optimizations on low level program representations, specifically LLVM IR. This ensures that optimizations can be leveraged by any language targeting LLVM. The fpl library used transprecision to provide a speed increase relative to existing Presburger Arithmetic libraries, such as isl [31]. Their approach ensures that constraints are stored using the smallest possible primitives. The storage reduction increases performance through decreased memory contention. PolyMage leverages polyhedral techniques in a domain specific language for image processing. Loopy allows programmers to specify loop transformations which are then formally verified using the Polyhedral Model.

This work heavily leverages the object-oriented interface to the Sparse Polyhedral Framework known as the Computation API [23]. The Computation API provides

a single entry point to Sparse Polyhedral Framework tools. Such tools include the Inspector/Executor Generation Library (IEGenLib) [28] which provides set and relation manipulation with constraints involving uninterpreted functions, and Omega+ for code generation [5].

This work builds on approaches to GPU code generation in MLIR [11]. This work attempts uphold the MLIR design goal of building composable high level abstractions that keep structure needed for optimization [30] as long as it is relevant. Without the extensible design of the core MLIR infrastructure [17], this work wouldn't be possible.

CHAPTER 5

CONCLUSION

This work creates an SPF MLIR dialect to extend the portability of SPF code to GPU as well as CPU. Clock speed has stagnated, increased parallelization and specialization now drives increases in performance. Scientific applications, in which SPF is used, need memory optimizations *and* need to be able to run on heterogeneous hardware. Previous SPF tools could be used to reason about and optimize data flow for memory optimizations but could only produce CPU code. By integrating SPF into an ecosystem designed for heterogeneous hardware, this work extends SPF to GPUs and positions SPF well to integrate with many other hardware platforms.

Performance evaluation showed competitive performance of SPF generated CPU and GPU code. The benchmark suite contains two common scientific kernels (MTTKRP, and TTM) which form the basis many important applications. The evaluation benchmarked code generated from the SPF MLIR dialect against code generated using the C SPF front end, and the PASTA benchmark suite. CPU benchmarks showed competitive to slightly better performance for MLIR generated kernels. GPU benchmarks showed competitive performance if GPU hardware was considered when writing the kernel, and worse results for MLIR generated kernels targeted naively to GPU.

Future work could integrate better heuristics and transformations for targeting GPUs. The foundation laid by this work could be extended to new platforms such as TPUs or other accelerators. MLIR provides an abstraction known as interfaces allowing an arbitrary number of dialects to use optimizations written in terms of the interface rather than a dialect. Future work would could look at providing SPF transformations as an interface that could be lifted out of the dialect built in this work.

REFERENCES

[1] LLVM Authors. User guide for amdgpu backend, 2022.

[2] LLVM Authors. User guide for nvptx back-end, 2022.

[3] Ian J. Bertolacci, Catherine Olschanowsky, Ben Harshbarger, Bradford L. Chamberlain, David G. Wonnacott, and Michelle Mills Strout. Parameterized diamond tiling for stencil computations with chapel parallel iterators. ICS '15, page 197–206, New York, NY, USA, 2015. Association for Computing Machinery.

[4] 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.

[5] Chun Chen. Polyhedra scanning revisited. *SIGPLAN Not.*, 47(6):499–508, jun 2012.

[6] IREE Developers. Iree (intermediate representation execution environment). https://google.github.io/iree. Accessed: 2022-05-29.

[7] Tobias Grosser, Groesslinger Armin, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04), December 2012.

[8] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. Domain-specific multi-level ir rewriting for gpu: The open earth compiler for gpu-accelerated climate simulation. *ACM Trans. Archit. Code Optim.*, 18(4), sep 2021.

[9] Chris Hathhorn. Engineering a compiler, second edition by keith d. cooper and linda torczon. *SIGSOFT Softw. Eng. Notes*, 37(1):36–37, jan 2012.

[10] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, jan 2019.

[11] Navdeep Katel, Vivek Khandelwal, and Uday Bondhugula. High performance gpu code generation for matrix-matrix multiplication using mlir: Some early results, 2021.

[12] Paul Kelly. Advanced computer architecture - the von neumann bottleneck and the turing tax, 2020.

[13] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Proceedings Frontiers '95. The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, 1995.

[14] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

[15] Steve Lant. Understanding gpu architecture virtual workshop, 2021.

[16] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[17] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law, 2020.

[18] Jiajia Li and Kevin Barker. PASTA: A parallel sparse tensor algorithm benchmark suite, Dec 2019.

[19] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, page 162, New York, NY, USA, 2004. Association for Computing Machinery.

[20] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. *SIGARCH Comput. Archit. News*, 43(1):429–443, mar 2015.

[21] Kedar S. Namjoshi and Nimit Singhania. Loopy: Programmable and formally verified loop transformations. In Xavier Rival, editor, *Static Analysis*, pages 383–402, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[22] Arjun Pitchanathan, Christian Ulmann, Michel Weber, Torsten Hoefler, and Tobias Grosser. Fpl: Fast presburger arithmetic through transprecision. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[23] Tobi Popoola, Ravi Shankar, Anna Rift, Shivani Singh, Eddie C. Davis, Michelle Mills Strout, and Catherine Olschanowsky. An object-oriented interface to the sparse polyhedral library. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1825–1831, 2021.

[24] Tobi Popoola, Tuowen Zhao, Aaron St. George, Kalyan Bhetwal, Michelle Strout, Mary Hall, and Catherine Olschanowsky. Code synthesis for sparse tensor format conversion and optimization. *International Symposium on Code Generation and Optimization.*

[25] William Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, aug 1992.

[26] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 12–27, New York, NY, USA, 1988. Association for Computing Machinery.

[27] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The formidable repository of open sparse tensors and tools, 2017.

[28] Michelle Mills Strout, Geri Georg, and Catherine Olschanowsky. Set and relation manipulation for the sparse polyhedral framework. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 61–75. Springer Berlin Heidelberg, 2012.

[29] 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.

[30] Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. Composable and modular code generation in mlir: A structured and retargetable approach to tensor compiler construction, 2022.

[31] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Lecture Notes in Computer Science,*, pages 299–302. Springer, September 2010.

[32] Alex Zinenko. Codegen dialect overview, 2021.