# Programmer-Controlled Readable Polyhedral Optimizations of `syr2k`

Haverford College

Divesh Otwani

April 29th 2019

# Contents

**1 Introduction** **4**
1.1 Overview . . . . . 4
1.2 Problem Statement . . . . . 5
1.3 Motivation . . . . . 5

**2 Background** **7**
2.1 Cache Systems . . . . . 7
2.2 Parallelism . . . . . 12
2.3 The Polyhedral Model . . . . . 13
2.4 Abstractions: Iterators & Records . . . . . 26
2.5 Seminal Papers & Optimization Tools . . . . . 29

**3 Analysis** **44**
3.1 Poor Original Performance . . . . . 44
3.2 Data Flow & Structure . . . . . 45
3.3 Existing Optimizations of `syr2k` . . . . . 47
3.4 Ideas About Optimization . . . . . 51

**4 Results** **52**
4.1 Timing Data . . . . . 52
4.2 Analysis of Optimizations . . . . . 55
4.3 Chapel Abstractions . . . . . 56

**5 Correctness** **59**

**6 Conclusion** **61**

## Abstract

The topic of this thesis is polyhedral optimization of dense matrix codes. Dense matrix codes are functions that update arrays inside nested loops with *affine* assignment statements. These are statements where the expressions that index arrays (the `<exp>` in `A[<exp>]`) are linear combinations of loop bounds and constants. In this thesis, we explore the problem of maintaining program readability while optimizing such programs with polyhedral-based techniques. We focus on one particular code, a benchmark from the PolyBench benchmark suite [PY15], called `syr2k`. Ultimately, we modify a known optimizations to get performance improvements that our unique optimizations could not achieve. Our optimizations can be abstracted for readability, but not without a heavy drop in performance.

## Acknowledgements

First and foremost, I am grateful to the LORD. My life and my work is not my own; it is by your grace. To both my parents: thank you for your sacrifice. Dad: it is because of your labor and concern that I have been able to have the life I have. Mom: it is through your nurturing and cooking that I have understood what it means to have a home. To my advisor Professor David Wonnacott: thank you for your wisdom, gentle kindness, and trust.

# Foreword

There are a few things to say about this thesis that aren't part of the content of this thesis.

- On style: I'm writing for my peers. I would like to keep things simple enough that a fellow Haverford CS major could (with a reasonable effort) fully understand what I'm saying. So, I will only use material or language that a Haverford CS major would know. If something was covered in a core class but is really specific, I might leave a footnote with a reference. I'll try to keep my language simple and jargon-free. Along those lines, I will try to not use Haverford-specific language.
- I might be heavy on some theory (this is a Haverford thesis, after all). If something is critical to what this thesis contributes, I will develop it half-rigorously; I will describe things to the point that making them rigorous is merely time consuming and holds no conceptual content. I do this primarily in defining polyhedral transformations but formalism also appears elsewhere when needed.
- I've set out to keep my discussion/work on my specific code example general enough that I can discover something about the techniques I'm using. That is, I hope to contribute something that reaches for something larger than the specific narrow problem my thesis focuses on.
- I accept full responsibility for the mistakes that are inevitably present in this thesis. Moreover, I am sure I've made quite a few mistakes throughout this thesis despite a solid effort towards being correct.

# Chapter 1

# Introduction

## 1.1 Overview

In a sentence, this thesis is about **optimizing** a specific program to run faster while maintaining **clarity** and **correctness**. More broadly, this thesis analyzes and explores the **optimization techniques** for programs like `syr2k`. Note that an optimization of a program or function does not change the algorithm that is used, but rather makes "surface level" changes.

The specific program we look at is called `syr2k`. This is an example from the Polybench 4.0 benchmark suite [PY15]. PolyBench is a collection of practical C-programs meant to be optimization challenges. At a high level, the program we consider is a subroutine that helps calculate the eigenvalues of a real-valued symmetric matrix. It takes as input matrices $A, B$ and constants $\alpha, \beta$ and outputs a matrix $C$ by doing the following update.

$$C \leftarrow \alpha AB^T + \alpha BA^T + \beta B \tag{1.1.1}$$

In this thesis, we optimize the code for this function while maintaining correctness and keeping the underlying algorithm. This program is an example of a **dense matrix code**. Such codes perform a series of updates to arrays inside a few nested loops.

There are two main ways we performance-optimize dense matrix codes: (1) by taking advantage of the **cache** and (2) by **parallelizing** the computation. We discuss the what a cache is and how it works in Section 2.1. For now, we can just say these are the two key ways of improving speed.

However, to maintain correctness and clarity, we need some other tools. To maintain correctness, we reason about `syr2k` and other dense matrix codes via the **polyhedral model**. This is a way of using precise mathematical reasoning about dense matrix codes and their optimizations. In this model, we can formalize the code under question, and formalize a set of transformations of the code which necessarily preserve correctness. Then, we just have a search problem (which we might do by hand); we just search for a transformation that optimizes the use of cache or enables the most parallelism (or a mix of both).

This still leaves clarity. While we might find a suitable **polyhedral transformation** that improves performance, we might have made the code grotesque. How do we maintain clarity? One approach is to use language features to *abstract optimization code*. The language Chapel is particularly suitable for this since it provides abstractions like iterators and is often as fast as C. The hope is, we can use records or other features of Chapel to abstract away the optimization so that a programmer could look at our optimized code and easily recognize that the algorithm executes the update in equation 1.1.1.

## 1.2 Problem Statement

We can state the problem this thesis tackles in two parts.

First, we want to find at least one optimization in C of `syr2k` satisfying the following conditions.

- **Correctness:** Our version of `syr2k` produces the same results as the PolyBench version of `syr2k`. Our optimization should be based on (though it might not strictly adhere to) the polyhedral model. This should be essential to how we argue that our optimized version is correct.
- **Speed:** Our version of `syr2k`, is as fast or faster with large inputs (i.e., large matrices $A, B, C$) on both single core machines and multicore machines than various well-known optimizations of `syr2k`, including versions from the current best polyhedral optimizers.
- **Abstracting the optimization:** We have a Chapel version of our code that uses language-provided abstractions to hide the abstraction but keep the algorithm clear.

Second, we want to use our optimization(s) of `syr2k` to say something about the optimization techniques we employed. In effect, this example is a case study; the real goal is to better understand optimization techniques. How do our optimization techniques interplay with cache and parallelism? Are the techniques we use easy to abstract with Chapel? Can our techniques be generalized beyond this example? Why or why not?

## 1.3 Motivation

Now, a reader might be wondering, where is the *motivation* for this problem? At a high level, this *particular* area of research is just emerging. Program optimization via the polyhedral model has been around for a long time. What's new is trying to hand-optimize while maintaining clarity and abstracting the optimization. Also, the optimizations we explore are unusual or extend past the traditional polyhedral optimizations.

At a lower level, this work is motivated because `syr2k` is interesting. First, as discussed in Section 3 this code has lots of room for performance improvement and the best polyhedral compilers do not have good single core optimizations for it. This also poses the question as to whether good single core optimizations of `syr2k` could allow even better parallel optimizations of `syr2k`. So, at the first level, we have a basic performance optimization challenge. Second, a thorough consideration of an *abstracting* a optimization of `syr2k` has not been done nor have advanced or tangential optimization techniques been explored (and we touch on both of these areas). Lastly, `syr2k` is part of a benchmark suite. In other words, it is a classic example of a collection of similar programs. More generally, dense matrix codes are incredibly common and perhaps the most executed lines of code.

In brief, this work is important because program clarity is important and because advanced and clarity-preserving optimizations of `syr2k` have yet to be explored in any considerable depth. Having something to say about how polyhedral (and other) optimization techniques can preserve clarity allows us to nicely hand optimize important programs, many of which are dense matrix codes. At an even higher level, how and why optimization techniques achieve these goals gives us insight as to how to write correct, fast and clean programs; it exposes some ideas about effective interplay between hardware, abstractions and reasoning about program correctness.

# Chapter 2

# Background

In this chapter we (1) cover all the terminology and background behind optimizing dense matrix codes and abstracting them and (2) discuss a few seminal papers and tools in the polyhedral community.

We start with the basics on how performance optimization even occurs: with caches and parallelism; that is, we explain what it is we look to optimize in programs that makes them fast. We move on to how the polyhedral model helps us tackle the problem of ensuring correctness. Finally, we look at some ideas on abstracting optimizations in the Chapel programming language. We wrap up this discussion with a lighting run through of the seminal papers & automatic optimization tools (namely PLUTO) in this research area. We focus on how these papers and tools apply to `syr2k`.

The material presented in this chapter draws heavily from [PH98], [FL11], [Col07], [Won10], [BOH+15]. The explanation of cache systems stems from [PH98]. The discussion of the polyhedral model blends and reformulates the ideas from [Won10], [Col07] and [FL11]; we present a formal mathematical model that draws heavily from each of these. The discussion of abstraction via chapel iterators draws out the basic ideas from [BOH+15].

## 2.1 Cache Systems

The **cache** is a piece of super fast memory that the computer automatically handles. Because of the cache, certain `LOAD` or `STORE` instructions execute faster than others. Specifically, we can see loads or stores that use cache are drastically faster than their non-cache counterparts. For millions of loads and stores, this adds up. Hence, a major way to optimize programs involves optimizing the computer's use of cache. So, in this section we (1) define cache and (2) express the ways cache can be optimized.

### 2.1.1 What is Cache?

As mentioned, the **cache** is a super fast piece of memory that the computer automatically handles. What do we mean by "automatically handles"? Unlike other pieces of memory, the programmer cannot choose to write to caches or read from cache. This happens automatically. At a high level, on some `LOAD` instructions we read a larger part of memory (from the disc or RAM) than the instruction asked for. Load instructions give an address for a chunk of main memory that can fit in a register. On executing a `LOAD` instruction, we store the appropriate value in the register and we store a larger chunk of memory in our cache. Then, if we have another `LOAD` instruction addressed to a part of memory that we have already put in the cache (and the cache is up to date with that memory), we read from this super fast memory and *not* from main memory or RAM. As a result, that `LOAD` instruction is very fast. This feature comes from the intuition that if it takes a long time to read from memory, we should read large chunks from memory near to what we actually wanted to read from a `LOAD` instruction. If you only need 1 gallon of water but a stream is a two mile walk away, it makes sense to carry more than 1 gallon back.

Now, with a computer, there has to be a deterministic way in which cache is used. In the remainder of this section, we explain that process. That is, we go through how a computer automatically uses its cache. Basically, we'll add depth to how `LOAD` and `STORE` instructions are executed. For simplicity, we call non-cache memory, simply 'memory'.

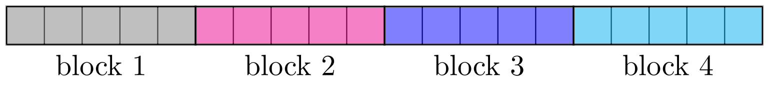

Figure 2.1: The division of cache into *blocks*.

Cache memory is divided into a collection of **blocks**. On certain `LOAD` instructions, a block is written to with a chunk of regular memory. More specifically, these blocks hold (1) a chunk of data from memory, (2) some information about the address of that data and (3) a sign bit (called the valid bit) telling us if we can be sure that the data is up to date with the actual memory (and some other meta-information). The part that holds actual data from memory is broken up into **words**. Recall that for a processor, a word is the number of bytes per register. Now, there's a subtle problem: if we `LOAD` into a register which only holds a word and each block holds a few words of data, how do we avoid data overlaps in different blocks of the cache? It turns out that main memory and RAM is pre-marked into chunks that fit in the data portion of each block of cache. These chunks are called **cache lines**.

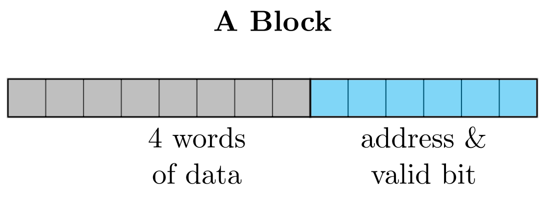

Figure 2.2: A Block

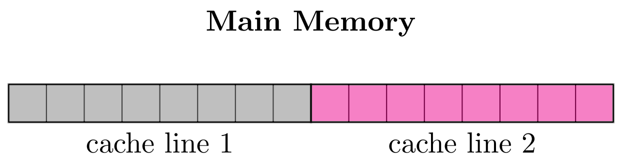

Figure 2.3: Cache Lines

So, when we perform a `LOAD` that will write to cache, we write whichever cache line that word was contained in to cache. With these preliminaries down, we can talk about the deterministic way most computers (and the ones we consider) use cache.

**Reads** Let's say the cache holds a collection of say $2^n$ blocks. Let's also say each block is timestamped with when it was last written to, and at startup all blocks have the same timestamp. Now, what happens when a `LOAD` instruction for a word within cache line $k$ happens?

If the data from cache line $k$ is in one of our $2^n$ blocks and up to date, the `LOAD` is super fast. It simply reads the specific word from all the words stored in that block. If it is not in one of our blocks or not up to date with the actual memory, the `LOAD` is much slower. The clock of the processor stalls[1] until main memory or RAM is read and the bits travel all the way to the register. At this point, we store the data from cache line $k$ in our cache. We do this at a block with the oldest timestamp (and if there are multiple blocks with the smallest timestamp, we break ties by the order of the blocks in cache). If this block's data has yet to be copied to memory, this happens first before we write over it.

Now, this way of choosing a block to write over isn't entirely accurate to how caches actually run. We are giving an abstraction for idealized **set associative** caches. However, for our purposes, this model of `LOAD` instructions is sufficient[2].

When a `LOAD` uses cache as its source of data, that read is called a **cache-hit**. Otherwise, it is considered a **cache-miss**. Cache hits are drastically faster than cache misses. We will focus on only one kind of miss: a **capacity miss**. In this situation a `LOAD` misses cache because the cache line we want has either been overwritten or has yet to be read from startup. Most of the time, we could have avoided such a miss if our cache had more blocks (of the same size), i.e., if it had a larger capacity or total size. Another key kind of cache miss, **associativity misses** occur when the cache is big enough to store what we want, but due to the way we choose the blocks when writing to cache, we end up overwriting things we didn't need to overwrite.

**Writes** Writes are fairly simple. In our model, we will say a `STORE` instruction checks if the word it wants to write to is validly stored in cache. If it is, it

[1]This doesn't necessarily happen, but this suffices for our discussion.

[2]In general, we will keep things as simple as possible.

updates that cache and the CPU asynchronously (without making the clock stall) copies the cache to main memory or RAM. If not, a `STORE` instruction simply takes a bit of time and the clock stalls.

### 2.1.2 Spatial and Temporal Locality

So far, we've developed an understanding of how cache works and a first-level analysis of why certain `LOAD` instructions (and `STORE` instructions) are drastically faster than other `LOAD` (or `STORE` ) instructions.

What is this first level analysis? In summary, we've seen that the speed of execution of these `LOAD` or `STORE` instructions depends on whether or not the execution uses cache; whether it is a cache-hit or cache-miss. We've pointed out that we are only looking at one type of cache-miss: a capacity miss. We do not consider cache-misses that occur because of the way we handle reads or writes – we assume we only "write over only the oldest, least-used item(s) in cache", which is intuitively the most sensible policy. Hence, we think the miss occurred because we didn't have enough cache space, or, capacity.

Going deeper (and in some sense, higher), we want to talk about two ways in which these capacity misses occur at the program level. However, we should use the right terminology; the **locality**[3] of a program (with a notion of its execution) is a high level measure of how well that program structures its accesses and writes to memory to make use of the cache. Namely, the locality of a program is a measure of how much we reuse data in cache. A program with good locality has less cache-misses than a program with bad locality. We want to discuss two ways in which programs can make use of memory to enhance or worsen the use of cache: **spacial locality** and **temporal locality**.

**Spacial Locality** If a program accesses memory such that the addresses of the words accessed are next to each other, it takes advantage of spacial locality. Suppose we represented (in memory) a $n \times n$ matrix $A$ of integers as one long block of $n$ rows of $n$ integers. To access $A[i][j]$ we go to the address $A + in + j$. Suppose we have something like the following code:

```
int i,j;
int sum = 0;
for (i = 0; i < n; ++i)
  for(j = 0; i < n; ++j)
    sum += A[i][j];
```

Figure 2.4: A program with good spacial locality.

Fix the constants $i, j$. Suppose we have a cache miss on the read $A[i][j]$, which occurs at memory location $\alpha = A + in + j$. At this point, the processor puts some words in cache, depending on how the cache lines fall, that include $\alpha$. Suppose at the very least, it populates one block with the memory locations

[3]This subsection draws heavily from [FL11].

$[\alpha, \alpha + k]$ for some $k \in \mathbb{N}$. The next line of code that is executed is `sum += A[i][j+1]`. This reads from the address $\alpha + 1$, which is stored in some block of cache. Moreover, the next $k$ reads of $A$ are stored in that block, making all of those $k$ `LOAD` instructions fast.

Because the order of doing these reads happened such that the addresses of the reads were close by, we efficiently used the memory stored in cache. That is spacial locality: how close are the addresses of the reads (or writes) are to each other.

**Temporal Locality** Temporal locality avoids cache misses by re-using pieces of memory before they get overwritten. That is, the number of memory operations between two operations $A, B$ that use the same memory cell are small enough that on operation $B$, the data is still in cache.

Take a simple example. Suppose that in addition to $A$, we have an array $B$ of $k$ integers.

```
int i,j;
int sumA = 0;
int sumB = 0;
for (i = 0; i < n; ++i)
  for(j = 0; i < n; ++j)
    A[i][j] += B[j % k];
```

Figure 2.5: A program with good temporal locality for $B$.

Suppose that $B$ fits nicely inside three cache lines. Suppose further that $k < n$. On the first iteration of the outermost loop for $i = 0$, all of $B$ gets pulled into cache into three different cache lines. Now, because all of $B$ is in cache, the iterations $i = 1 \cdots i = n$ will all have fast reads of $B$ (provided, $n$ isn't so large that the cache lines pulled in for $A$ overwrite the ones for $B$).

### 2.1.3 Cache Levels

There's one detail about caches that we've swept under the rug. There are typically multiple caches in a modern computer. We distinguish between these different caches by assigning each a *cache level*. These caches have different sizes and corresponding speeds. Sometimes there are two levels of cache, called L1 and L2 cache. Sometimes there is a third level called L3 cache. The L1 cache is usually very small but very fast. The L2 cache is usually 4 or 8 times larger and a bit slower. The L3 cache is much (sometimes 24 times) larger than L2 cache and much slower, though still way faster than main memory.

With multiple cache levels, a processor executes a `LOAD` instruction by first checking L1 cache, then L2 and then L3 before going to main memory to retrieve the data. Naturally, this makes cache hits and misses more complicated. We may have a L1 cache miss but a L3 cache hit and so on. In this system, we consider a cache miss to mean a L3 cache miss and a cache hit to mean we hit one of the caches.

For an example computer, take the Intel Core i5-7500 chip [cor19]. It has 4 cores (or processing units). Each core has its own L1 and L2 cache. However, all 4 cores share a L3 cache. The L1, L2 and L3 caches have sizes 32, 256 and 6144 KB respectively.

## 2.2 Parallelism

The second chief way in which we optimize code is through making computations parallel. The way this improve performance is obvious: programs execute faster because multiple CPU instructions are executed at the same time, leading to an overall higher instruction per second rate of execution. Consider the following example.

```
for all (i = 0; i < n; ++i)
  S(i) // statement i
```

Figure 2.6: A generic `forall` loop.

In the example above, we execute statement $S$ on a number of different threads on different cores each with a different value for $i$. Of course, for large $n$ it is foolish to make so many threads. So typically, a range of values will be executed on different cores and threads.

For example, if $n = 64$ and our computer has 4 cores, the language might decide to execute $i = [0, 8]$ on thread 1 of core 1, $i = [9, 16]$ on thread 2 of core 1, $i = [17, 32]$ on thread 1 of core 2 and so on.

Within this framework, our only concern will be to avoid data dependencies between each thread. The threads (on different cores) can use some collection of data that is read-only to all of them and write to their own slice of data that none of the other threads can read from or write to. Of course, this doesn't exclude executing one part of the computation in parallel, waiting for all threads to finish, and then doing another part in parallel. In our example, if we had an outer $k$ loop and we only made the $i$ loop parallel, we would execute the inner loop in parallel but not the outer one. That is, for a given $k_0$ we would first execute $S$ for the index values $\{(k_0, i) : i \in [0, n]\}$ in parallel before moving on to the index values $\{(k_0 + 1, i) : i \in [0, n]\}$.

### 2.2.1 Multi-Core Cache Fights

When we do parallel execution, different cores (or threads) can fight for cache space. Often, 4 core machines have one L1 and L2 cache per core. However, L3 cache is typically shared among cores. Whatever the cache situation, we may enounter a situation with two different threads using the same cache. This can create problems.

For example, say threads $T, T'$ are both using L3 cache. Say that $T$ did a `LOAD` instruction that brought data $D$ into cache that $T$ will later try to read again. Say that as $T$ continues its execution, $T'$ keeps doing `LOAD` instructions that bring more and more cache lines into cache. Eventually, the work done by $T'$ leads to the computer overwriting the cache block that held $D$. Now, when $T$ goes to read some data in $D$, it has to access main memory again.

Of course, this is a super simple example. In the more general case, several threads end up overwriting data from cache before the thread that brought it in could re-use it. For this reason, it is benificial to have one computation per core running in parallel, where each computation's data can fit in L2 cache (or whatever cache level is private to that core).

## 2.3 The Polyhedral Model

We've covered the background on certain properties of programs that make them fast: optimize the use of cache and use multiple cores & threads to parallelize the computation. Of course, to have a correct optimization we need to know the transformed program produces the same results as the original program.

In this section, we cover the best known tool for ensuring a correct optimization: **the polyhedral model**. At a high level, this is a way of modeling and transforming certain dense matrix codes such that we can easily transform the model to a new model while ensuring the code represented by the new model produces the same result as the original code.

These transformations are also "surface level" to the degree that most of the underlying algorithm in the original code is present in the transformed code. (In fact, technically, this is what makes such a transformation an optimization and not a refactoring.) Note that even though the algorithm might be similar, optimized code is often dense and unreadable; this is mainly where our research comes into the picture.

### 2.3.1 Overview

In this section, we strive to (1) precisely describe certain dense matrix codes and (2) state a set of "equivalent" matrix codes given some initial original matrix code. In other words, with the work in this section we should be able to write a program (that is a dense matrix code) according to certain rules and then create a number (perhaps an infinite number) of "equivalent" programs. Applied to `syr2k`, we should be able to express `syr2k` in the language of this section and then describe a set of possible transformations that are all equal to `syr2k`.

Of course, we need a clear idea of what it means when two programs are "equivalent". This is fairly simple for two pure functions: two pure functions are equal if they always return the same output for the same input. Our task is to jump from the idea of equal pure functions to equal programs.

To accomplish the first goal, we need to semi-formally describe a programming language for dense matrix codes and have some idea of its semantics. By "semantics", we mean we need a basic but still formal computer on which we can run any program in our language. The idea is to make the language and semantics powerful enough that we can represent the `syr2k` code in it (but not too powerful because then other things become difficult). With this in place, we need to accomplish the second (and more important) goal: find a (reasonably large) collection of equivalent programs. Of course, when we define our language and how our computer executes programs written in our language, we will define the equivalence of programs.

The second goal is where the polyhedral model comes in; if we want an equivalent matrix code to `syr2k`, we first convert it into a model, transform that model into a different model and then convert that new model back to regular code:

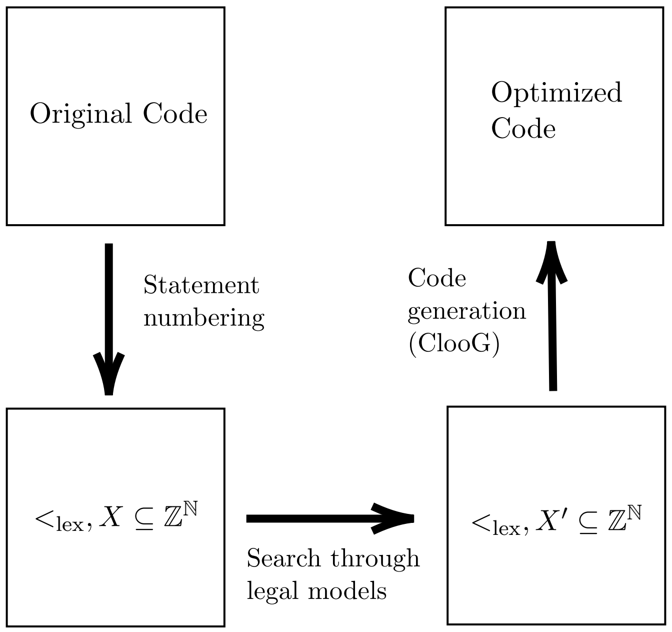
Original Code
Optimized Code
Statement numbering
Code generation (ClooG)
$<_{\text{lex}}, X \subseteq \mathbb{Z}^{\mathbb{N}}$
$<_{\text{lex}}, X' \subseteq \mathbb{Z}^{\mathbb{N}}$
Search through legal models

Figure 2.7: How to use the polyhedral model.

For our second goal, we need three tools. First, we need to be able to go back and forth between our model and code. In Figure 2.7 these are the vertical up and down arrows. Second, we need to have a way of transforming our model to some number of new models. In Figure 2.7 this is the horizontal right arrow. Third, we need a way of checking if a chosen new model is 'legal' – that the new model represents a program equivalent to the program of the first model. Of course, we don't just want to find *any* equivalent program. We want a model that represents an equivalent program that has, say, better spacial locality than the original program. That is, with the tools to browse through equivalent programs, our problem becomes a search problem: find an equivalent program that makes good (or the "best") use of, for example, cache. As an aside, note that there are several details of Figure 2.7 that we won't discuss until the end of this section.

This material breaks down into the following sections. In Section 2.3.2 we accomplish our first goal and the first tool of our second goal. In Section 2.3.3

we will develop our second tool and in Section 2.3.4 we will develop our third tool. Finally, we return to Figure 2.7 and explain all the moving parts (as well as point out the intuition behind why this model is so useful).

### 2.3.2 Polytope Representation of Dense Matrix Codes

Our first goal is to have some formalism for programs like `syr2k` and the machines they run on; this should be just powerful enough to let us reason about correctness.

$\langle \textit{arr-ref} \rangle ::= \langle \textit{array-var} \rangle \; [\langle \textit{int} \rangle]$
  $\langle \textit{array-var} \rangle ::= A, B, C, \cdots$
  $\langle \textit{int} \rangle ::= i \in \mathbb{Z} \mid \min\,(\langle \textit{int} \rangle, \langle \textit{int} \rangle) \mid [-]\; \langle \textit{const-var} \rangle \mid \langle \textit{var} \rangle$
  $\langle \textit{const-var} \rangle ::= a, b, c, \cdots$

$\langle \textit{decls} \rangle ::= \epsilon \mid \langle \textit{type} \rangle \; \langle \textit{arr-ref} \rangle ; \; \langle \textit{decls} \rangle \mid \langle \textit{type} \rangle \; \langle \textit{const-var} \rangle = \langle \textit{const} \rangle ; \; \langle \textit{decls} \rangle$
  $\langle \textit{type} \rangle ::=$ `int` $\mid$ `double`

$\langle \textit{assignment} \rangle ::= \langle \textit{arr-ref} \rangle$ `:=` $\langle \textit{expression} \rangle$
  $\langle \textit{operator} \rangle ::= + \mid - \mid * \mid \div \mid \epsilon$
  $\langle \textit{expression} \rangle ::= \langle \textit{arr-ref} \rangle \mid \langle \textit{constant} \rangle \mid \langle \textit{operator} \rangle (\langle \textit{expression} \rangle)^*$

$\langle \textit{program} \rangle ::= \langle \textit{decls} \rangle \; ; \; \langle \textit{core-program} \rangle$
  $\langle \textit{core-program} \rangle ::= (\langle \textit{assignment} \rangle \mid \langle \textit{core-program} \rangle)^*$
  $\mid$ `for` $\langle \textit{var} \rangle$ `=` $\langle \textit{int} \rangle$ `to` $\langle \textit{int} \rangle$ `do` $(\langle \textit{core-program} \rangle)^*$

Figure 2.8: Our language for dense matrix codes.

**A Dense Matrix Code** In Figure 2.8 we define the language for our dense matrix codes. Essentially, a program is just a collection of declarations followed by assignments to arrays that might be nested inside for loops. The bounds of our for loops can go from, for example, 1 to $n$, where $n$ is a constant variable. For notation, call $P$ the set of programs produced by the grammar $\langle \textit{program} \rangle$. Not every program in $P$ is legal or valid; for instance, a reference might index a 2D array only once (`arr[i]`). It should be fairly easy to formalize all the rules that make a program valid; hence, we are not going to focus on this. In the same vein, we won't formalize constants or what the minimum function does. Let us formalize `syr2k`.

```
void syr2k(int n, int m,  double alpha,  double beta,
  double C[N][N], double A[N][M], double B[N][M]) {
  int i, j, k;
  // Note: N = n and M = m; these are configuration constants.
  for (i = 0; i < n; i++) {
    for (j = 0; j <= i; j++)
      C[i][j] *= beta;
    for (k = 0; k < m; k++)
      for (j = 0; j <= i; j++){
          C[i][j] += A[j][k]*alpha*B[i][k] + \
                     B[j][k]*alpha*A[i][k];
      }
  }
}
```

Figure 2.9: The original `syr2k` code in C.

Above, we have the original `syr2k` code. It is a function that is given particular parameters along with some initialized arrays; on each call, we have constants for `n`, `m`, `alpha` and `beta`. Ultimately, our optimization, then, optimizes a specific call of this function. Hence, we only need to consider programs equivalent to a specific call of this function. Say, on our call, the values of `n` and `m` are $n$ and $m$ respectively (and do the same for `alpha` and `beta`). Now, we can write `syr2k` in our language:

```
double alpha = α;
double beta = β;
double C[n][n];
double A[n][m];
double B[n][m];

for i = 0 to n do {
  for j = 0 to i do {
    C[i][j] := C[i][j] * beta
  }
  for k = 0 to m do {
    for j = 0 to i do {
      C[i][j] := A[j][k]*alpha*B[i][k] +  \
                 B[j][k]*alpha*A[i][k] + C[i][j]
      }
    }
  }
}
```

Figure 2.10: Our formalized `syr2k` code.

We declared the variables up front and then expressed the rest of the code as assignments to array references inside for-loops. Unlike our grammar, we wrote out expressions with operators (like +) in infix form instead of prefix form.

**Our Computer** To give our formal language meaning, we need to define how it executes on a computer; we'll introduce a set of definitions. Since our code is imperative, we start with a definition of memory.

**Definition 2.3.1.** The **memory** of our computer is a function $m : A \to V$ where $A$ is the set of all possible array references and $V$ is the set of all possible values, which for our purposes can be thought of as $\mathbb{Z} \cup \mathbb{R}$. Let $M$ be the set of all possible memory states, i.e., all possible memory functions. (Note that $m$ maps all array references even if two cannot be used at the same time; it maps `B[0]` and `B[1][3]` even though at most one of these will be used in any valid program.)

In imperative programming, computers simply update memory according to a program. We can formalize this as a function, $C : M \times P \to M$ where $P$ is the set of *valid* programs. For simplicity, we will treat $P$ as the set of core programs that are part of some valid program (just drop the declarations from each valid program).

For example, we know $C($`A[0] := 1`$, m_0) = m_1$ where $m_0$ is the function that maps all array references to 0 and $m_1$ maps all array references except $A[0]$ to 0 and maps $A[0]$ to 1. At a higher level, we've defined the *big step semantics* of our computer.

We know, at a lower level, that our computer actually just does a sequence of assignments and the execution of a program is a (finite) sequence of memory functions $e = (m_0, m_1, \cdots, m_n)$. What exactly do we mean by this? Well, we mean any (core) program, is actually just a long list of **simple assignments** which we define to be assignments where all array references that appear in them have literal integer indices. So, for example, `A[2] = A[n] + i` is not a simple assignment while `B[2] = B[4] + 2` is in fact a simple assignment. To make this crystal clear, consider the following program[4].

```
double A[5];
double B[5];
int n = 3;
int m = 1;

for i = 0 to n do {
  A[i] = A[i+1] - i ;     // Statement S0
   for j = 0 to m do {
     B[i] = A[i + j] + B[i + j]   // Statement S1
   }
}
```

Figure 2.11: A simple formal program.

This program is equivalent to this finite sequence of simple assignments:

[4]This example is not completely within our language, but still illustrates the point.

```
A[0] = A[1] - 0 ;          // (0,0)
B[0] = A[0]  + B[0] ;      // (0,1,0)
B[0] = A[1]  + B[1] ;      // (0,1,1)

A[1] = A[2] - 1 ;          // (1,0)
B[1] = A[1]  + B[1] ;      // (1,1,0)
B[1] = A[2]  + B[2] ;      // (1,1,1)

A[2] = A[3] - 2 ;          // (2,0)
B[2] = A[2]  + B[2] ;      // (2,1,0)
B[2] = A[3]  + B[3] ;      // (2,1,1)

A[3] = A[4] - 3 ;          // (3,0)
B[3] = A[3]  + B[3] ;      // (3,1,0)
B[3] = A[4]  + B[4] ;      // (3,1,1)
```

Figure 2.12: The sequence of simple statements.

Of course, we don't write code like this, but we can see, intuitively, that each core program $p$ is equivalent to a finite sequence of simple assignments $s(p)$. A computer[5], in this step by step view, is a function $c$ that takes a memory function $m$ and a simple assignment $a$ and produces a new memory function $m'$. For our purposes, we can think each computer starts with everything mapped to zero, that is with memory $m_0$ such that $\forall x, m_0(x) = 0$. If a program $p$ has sequence $s(p) = a_0, \cdots, a_n$, we can write $c[s(p)]$ for the sequence $(c(a_0, m_0), c(a_1, c(a_0, m_0)), \cdots, c(a_m, \cdots))$. In this notation, $C(p) = c(a_m, \cdots)$. Without getting bogged down in the details of formalizing these things, we can provide the following definitions.

**Definition 2.3.2.** A **computer** is the function $c : M \times A \to M$ where $A$ is the set of all simple assignments that behaves as expected according to our intuition. The execution of a computer can be defined as follows. Let $s : P \to A^{\mathbb{N}}$ be the injection that sends a core program $p$ to a finite sequence of simple assignments that represent $p$. Write $s(p) = (a_1, \cdots, a_n)$. Then, the **execution** of a program $p$ on a computer $c$ is the sequence $c[s(p)] = (c(a_1, m_0), c(a_2, c(a_1, m_0)), \cdots, c(a_m, \cdots))$. The **result** of an execution is the last element of the sequence $c[s(p)]$ denoted $C(p)$. The **length** of the execution is $n$.

With these definitions, it should be apparent when two programs are equal:

**Definition 2.3.3.** Two legal core programs, $p_1, p_2$ are **equivalent** if their executions have the same result: $C(p_1) = C(p_2)$.

**The Polytope Representation** Finally, we can talk about how our dense matrix codes are modeled. Essentially, for each core program $p$, we assign a tuple of integers to each simple statement in $s(p)$ and then order these tuples

[5]Technically, we are talking about the execution of a computer but we can ignore this detail.

to indicate the order in which the computer $c$ should progress through memory states.

Instead of formalizing the way in which statements are numbered, we provide a clear and unambiguous description of the numbering of statements:

Recall that a grammar is essentially a tree. A core program is a tree whose leaves are one or more assignments. The interior nodes are for-loops that hold multiple other nodes in a linear order. The (tree) depth of each statement indicates the number of loop indices it has access to. We can number each statement using the path from the root of the core program to the assignment. Start with an empty tuple. If we hit an interior node, it represents some for loop. Append the loop index/variable of that loop to the tuple. If that interior node that has more than one core program child, append to our tuple the position of that child in the list of children. Otherwise don't append anything at this point. When we arrive at a leaf, append the position of the assignment in the list of assignments of that leaf, if there is more than one assignment.

This is not as complicated as it sounds. Take the example from Figure 2.11. We have two statements, commented as $S0$ and $S1$. Let's trace the path to each statement so we can number each statement. Let's start with statement 0. The root of the core-program tree has index $i$. So, the first value in the tuples for both statements is $i$. Now, since that root has two children, we add append the order of the children to the tuple for each statement. Hence, the tuple for statement 0 is $(i, 0)$. This is how we identify all the instance of statement 0. Let's do the same for statement 1. Since it is in the second node under the root, its tuple so far is $(i, 1)$. Now we hit another interior node and append its index $j$. However, since this node has only one statement as a child, we do not need to append orders of statements. Hence, each instance of statement 1 is numbered by the tuple $(i, 1, j)$. These numbers are depicted in comments in figure 2.12. With this in hand, we can state the following definition.

**Definition 2.3.4.** Let $I : P \to T$ be the function described so far from (valid) programs to $T = \mathcal{P}(\mathbb{Z}^{\mathbb{N}})$. Note that $T$ is simply the set of all sets of integer tuples. We know, $\{(0,0), (1,0,0), \cdots\}$ from the last example is in $T$. Call this the **tupling function**.

Finally, we can state our model.

**Definition 2.3.5.** Let $p \in P$ be a valid (core) program. The **polytope model of** $p$ is the triple $(I(p), <_{\text{lex}}, \pi)$ where $<_{\text{lex}}$ is the standard short-biased linear order on $I(p)$ and $\pi : I(p) \to A$ is the intuitive map from each integer tuple to some simple statement corresponding to the original matrix code.

In our example from figure 2.11, our set $I(p)$ is something like

$$I(p) = \{(0,1), (0,2), (0,3), (1,0,0), (1,0,1), (1,1,0), (1,1,1), \cdots\}.$$

The order is $<_{\text{lex}}$, which is the lexicographic order of integer tuples that is short biased, so $(1,2,3) < (1,2,3,4)$. The function $\pi$ mapped, for example,

$(1, 2, 1)$ to this instance of statement 2: `B[2] = A[3] + B[3]`. So, with this example, we can take the model of our program and execute it without looking at the code. We linearly order the elements of $I(p)$, find the simple statement each integer tuple corresponds to with $\pi$ and keep applying our computer $c$ with initial memory $m_0$ walking along the sequence.

We remark that we could adapt this model to use some strict partial order instead of $<_{\text{lex}}$ to represent the idea of executing simple statements in parallel. Namely, we would execute each part of the linear pre-order of $\prec$ in parallel.

**Definition 2.3.6.** Let $\sigma = (S, <_{\text{lex}}, \pi)$ be the polytope model of some program. Let $S = \{x_1, \cdots, x_n\}$ where $x_i <_{\text{lex}} x_{i+1}$. Let $x = (x_1, \cdots, x_n)$. The (not unique) **execution** of $\sigma$ is defined as $c[x]$ and the **result** of that execution is the last element of $c[x]$. As before, two models are **equivalent** if all of their executions have the same result.

Observe that for any valid program $p$, the result of the execution of $p$ and the result of the execution of $\sigma = (I(p), <_{\text{lex}}, \pi)$ are the same. Going back to figure 2.7, when we go from our changed model back to code, we ensure that we go to some program whose result matches the result of our model. In fact, that upward vertical arrow in the figure is fairly easy with the right model. Hence, to ensure a correct transformation, all we really need is to ensure the model we transform to from $\sigma$ is equivalent to $\sigma$.

### 2.3.3 Polyhedral Transformations

How do we go from one model to another model? That is, we need a way of calculating other models from a given model such that a reasonable portion of those other models are equivalent to the given one. Formally, we want some function $t : \mathcal{M} \to \mathcal{P}(\mathcal{M})$ where $\mathcal{M}$ is the set of all models.

We discuss an easy way of going about doing this. We could consider all the transformations that take a model $(X, <_{\text{lex}}, \pi)$ and output $(f(X), <_{\text{lex}}, \pi \circ f^{-1})$ where $f$ is some bijection on $X$ whose range is some subset of $\mathcal{P}(\mathbb{Z}^{\mathbb{N}})$. This is the set of all sets of integer tuples – each set $Q \in \mathcal{P}(\mathbb{Z}^{\mathbb{N}})$ could be obtained by applying the tupling function $I$ to some program. So, our choice of $f$ must be such that $f[X] = I(p)$ for some $p \in P$.

What exactly are we doing? Consider a basic program like this:

```
for i= 0 to 1 do { A[i] = i }.
```

It is modeled by $(X = \{(0), (1)\}, <_{\text{lex}}, \pi)$ where $\pi((0)) =$ `(A[0] = 0)` and $\pi((1)) =$ `(A[1] = 1)`. The function $f$ is a bijection on $X$. It could send $X$ to $Y = \{(1), (2)\}$ with $f((x)) = (x + 1)$. Our new model would be $(f[X] = \{(1), (2)\}, <_{\text{lex}}, \pi \circ f^{-1})$. Consider $g = \pi \circ f^{-1}$; for the tuple $(1)$, $g((1))$ is $\pi$ applied after reversing $f$, which is $\pi$ applied to the tuple $(0)$. This is `(A[0] = [0])`. In other words, since $f$ is a bijection, we ensure that whatever statement instance $\pi$ associated to $x \in X$, the same instance remains for $f(x)$.

**Definition 2.3.7.** Let $\sigma = (X, <_{\text{lex}}, \pi)$. Define $t(\sigma)$ as follows:

$$t(\sigma) := \{(f[X], <_{\text{lex}}, \pi \circ f^{-1}) : f \in F(X)\}$$

where $F(X)$ is the set of bijections $f$ whose domain is $X$ and whose range, $f[X]$, is some set of integer tuples such that there is a $p \in P$ satisfying $I(p) = f[X]$. We call $t$ the **transformation function**.[6]

It's important to take a step back and notice how $t$ operates. We are representing changes in models with changes to the set of integer tuples or integer points. That is, using $f$ alone, we are representing a change from $\sigma$ to $\sigma'$. We call the set of integer tuples in the model of a program, the **iteration space** of that program. We call our transformations **iteration space transformations** because they change models by simply applying bijections on our iteration space.

Of course, this isn't the whole story. When are our transformations *polyhedral*? Basically, we don't scan all the models in $t(\sigma)$ since that is still too difficult. Instead, we put further restrictions on what $F(X)$ looks like; that is, we put more requirements on what the bijection $f$ does.

Our transformations become polyhedral when we construct $f$ in a piecewise and affine way. Partition $X$ by the different statements. Going back to the example from figure 2.11, we would partition into all the tuples that come from statement 1 and all the tuples that come from statement 2. Then, we would construct piecewise bijections on each part of the partition. This is clearer from the reverse direction: if we take several bijections whose domains are pairwise disjoint and whose ranges are pairwise disjoint and union them, we still get a bijection.

How do we construct these piecewise bijections? Let's continue with the example from Figure 2.11. Let $X$ be the iteration space. Consider the subset $Y$ that is all tuples that come from statement 1. Namely,

$$Y = \{(0, 0), (0, 1), (0, 2), (0, 3)\}.$$

These, by construction of our tupling function $I$ are all of the same size. We represent a bijection on $Y$ by a $2 \times 2$ matrix $A$ where the bijection is $f'(x) = Ax + b$ (for some fixed two tuple $b$). Drawing directly from [Col07], we can define $f'$ to have the properties we want. To explain this, a definition would be useful:

**Definition 2.3.8.** An **affine function** is a function $f : \mathbb{Z}^n \to \mathbb{Z}^m$ defined by $f(x) = Ax + \vec{b}$ where $A$ is some $m \times n$ integer matrix and $\vec{b} \in \mathbb{Z}^m$ .

In our example (and more generally), we know that (1) the output of affine functions on $Y$ gives us a set that can be represented by a program, i.e., there is some $p$ such that $f'[Y] = I(p)$ and (2) affine functions with square matrices are bijections if and only if the determinant of $A$ is 1 or -1. We can use this to build a bijection $f$ on all of $X$ with one piece as $f'$. It is reasonably easy

[6]Exercise: Does $t$ ever return the empty set?

to search through all $2 \times 2$ matrices whose determinant is 1 or -1 and have integer values. Hence, it is fairly easy to find all polyhedral transformations that we care about. Now, it might be the case that the range of the bijection $f'$ overlaps with the range of the bijection we construct on $X - Y$. If that is the case, we can select our constant vector $b$ in such a way to avoid this overlap (and maybe to ensure that we preserve data flow – see the next section).

Finally, we can use our bijection $f'$ on the subset of the iteration space $Y \subsetneqq X$ to construct a bijection on the whole iteration space $X$. Namely, we could fix $X - Y$ and that way, the identity function on $X - Y$ and $f'$ are both bijections with non-overlapping domains and ranges. Hence, their union is a bijection, which we can call $f$. This function $f$ is an example of a polyhedral transformation. After a long wait, we can provide a definition.

**Definition 2.3.9.** Let $\sigma = (X, <_{\text{lex}}, \pi)$ be a model of some program with assignment statements $S_1, \cdots, S_n$. Let $Q = \{Q_1, \cdots, Q_n\}$ be a $n$ partition of $X$ such that $Q_i$ contains the image under the tupling function $I$ of all the instances of statement $S_i$. The function $f$ is a **polyhedral transformation** on $\sigma$ if $\sigma_f = (f[X], <_{\text{lex}}, \pi \circ f^{-1}) \in t(\sigma)$ and $f$ is piecewise affine on $Q$. The set of all polyhedral transformations of $\sigma$ is denoted $T(\sigma)$.

Note that $f$ is not the transformed model. The polyhedral transformations are functions on the iteration space that are used to go from our initial model to a new model.

Before we conclude, we should say something about why this was important to begin with. The current approach to polyhedral optimization is not primarily hand-based.[7] Hence, a major part of automatic, program-run polyhedral optimization involves algorithmic implementation issues. We could have stopped with our definition of a transformation if we wanted a mathematically sufficient tool for reasoning about correctness. However, since we need to reason about transformations that we can use to create optimized code, we need affine piecewise bijections.

**Aside: What's Polyhedral Here?** Polyhedra are basically $n$ dimensional polygons, often called polytopes. Formally, a polytope $P$ is a set of points in $\mathbb{R}^n$ such that $P$ is the intersection of a finite number of half-spaces. That is, $P$ is a finite set of points satisfying a finite number of equations of the form $c \cdot x + k \leq 0$ where $c$ is a vector of constants and $k \in \mathbb{R}$.

From the way we construct our core programs, when we partition our iteration spaces by statement, each part of the partition is the set of integer points that lie inside some polytope. It turns out that our polyhedral transformations (because they are affine) send this part of the partition of the iteration space to another set of points which is also the set of integer points inside a polytope. So, in some sense, we have the following picture. Our iteration space $X$ is a collection of different dimensional polytopes (and $X$ is the union of the set of integer points inside these polytopes). Our polyhedral transformation

[7] See for example the polyhedral compiler PluTo [Bon].

sends each polytope of dimension $k$ to another polytope, possibly of different dimension $k'$.

### 2.3.4 Data Flow & Legal Transformations

How can we be sure our transformed model is equivalent to the first? Let's ask this question formally.

**Definition 2.3.10.** Let $\sigma = (X, <_{\text{lex}}, \pi)$ be a model. Let $f \in T(\sigma)$. Let $\sigma_f$ denote $(f[X], <_{\text{lex}}, \pi \circ f^{-1})$.

For any model $\sigma$ and $f \in T(\sigma)$, how do we know the result of $\sigma_f$ is equal to the result of $\sigma$? The answer is through a concept called *data flow.*

**The Intuition** Go back to the example in Figure 2.12. It should be apparent that our polyhedral transformations merely change the order of these simple statements.

Well, without knowing anything about the values of $A$ or $B$, can we swap the order of the first two statements? No. The second statement reads $A[0]$ and the first writes to $A[0]$. If, on some execution, the first statement wrote the value 5 to $A[0]$ and $A[0]$ was initially 7, the two orders do not write the same value to $B[0]$. In the first original order we write `B[0] = 5 + B[0]` and in the second order we write `B[0] = 7 + B[0]`. It is possible that our final result will be different.

However, if we swap statements so that the *data flow* is never interrupted, then we're fine. Look at it this way. Number the simple statements in order as $s_0 \cdots s_n$. Each statement $s_i$ depends on some number of other statements. Call this set of dependencies $D(s_i)$. Suppose we re-ordered these statements into the order $s_{x_1}, \cdots, s_{x_n}$. If for each statement $s_{x_j}$, all the dependencies $s_{x_q} \in D(s_{x_j})$ are such that $q < j$, then the result must be the same. All we need to do now, is formalize this intuition.

However, we should address an important question. Why are we avoiding an analysis that uses the values we know are in $A$ and $B$? Briefly, moving around statements with that level of analysis is really hard to (a) reason about properly and (b) get right in a general way that can be programmed.

#### Legal Transformations

**Definition 2.3.11.** Let $A$ be the set of all simple statements and $s \in A$. That is, $s$ is an assignment (from figure 2.8) where all elements of $\langle int \rangle$ are literal integers. The **read function** is $r : A \to \mathcal{P}(R)$ where $R$ is the set of all possible array references; $r$ is defined so that $r($`<arr-ref> := <expression>`$)$ traverses the tree `<expression>` for leaves which are references. The **write function** $w : A \to R$ is thus defined: $w($`a := e`$) =$ `a`.

The write and read functions will help us express dependencies formally. Say simple statement $s$ is executed before simple statement $t$ in some dense matrix code. When must we maintain the order of these statements?[8] There are three cases. Either (1) $s$ writes to a reference read by $t$, (2) $t$ writes to a reference read by $s$ or, (3) $s$ and $t$ both write to the same reference. In the first two cases, we need to make sure that the read of the reference has the same value which won't let us switch the order of the statements. In the third case, if we swap the order of the statements, a later read of that reference could have a different value. Hence, in all three of these cases, the order of the statements cannot safely change.[9]

**Definition 2.3.12.** Let $\sigma = (X, <_{\text{lex}}, \pi)$ be a model. Let $a, b \in X$ and say $a <_{\text{lex}} b$. We say $\pi(a)$ **depends on** $\pi(b)$, denoted $\pi(a) \rightsquigarrow \pi(b)$, if at least one of the following holds.

- $w(\pi(a)) \in r(\pi(b))$
- $w(\pi(b)) \in r(\pi(a))$
- $w(\pi(a)) = w(\pi(b))$

Now, we can define data flow.

**Definition 2.3.13.** Let $\sigma = (X, <_{\text{lex}}, \pi)$ be a model. The **data flow** of $\sigma$ is the following relation.

$$D(\sigma) := \{(a, b) : (a, b \in X) \wedge (a <_{\text{lex}} b) \wedge (\pi(a) \rightsquigarrow \pi(b))\}. \tag{2.3.1}$$

For ease of notation, we will write $a \rightsquigarrow b$ to mean $(a, b) \in D(\sigma)$ and $\pi(a) \rightsquigarrow \pi(b)$. Now, the transformations we care about are ones that preserve data flow.

**Definition 2.3.14.** If $\sigma_f \in t(\sigma)$, we say $\sigma_f$ is a **legal $\sigma$-transform** if both have the same data flow:

$$(a \rightsquigarrow b) \Longleftrightarrow (f(a) \rightsquigarrow f(b)).$$

Finally, we can state[10] our central theorem.

**Theorem 2.3.15.** *Let $\sigma = (X, <_{lex}, \pi)$ be a model and let $\sigma' \in t(\sigma)$ be a legal $\sigma$-transform. Then, the results of $\sigma$ and $\sigma'$ are the same. That is, the models encode equivalent programs.*

So, what we really want are legal $\sigma$-transforms that are also polyhedral transformations. This the the space of transformations that we search through for finding a suitable optimization.

[8]This is a bit more general than data flow, but easier to think about.
[9]Exercise: Why are these three cases sufficient?
[10]A proof, while valuable, is not necessary for this thesis.

**The Moral of The Story** At last, we can return to Figure 2.7 and explain the whole picture. We get from our original program $p$ to a model via our tupling function $I$ and an intuitive creation of $\pi$; we go from $p$ to the model

$$\sigma = (X = I(p), <_{\text{lex}}, \pi).$$

Let's move onward to the right horizontal arrow in the figure. We generate a bunch of possible polyhedral transformations (and thus, other models) by looking at all statement-piecewise affine invertible functions $f$ on our iteration space $X$ that are also bijections. We filter out the ones that do not preserve data flow. The remaining ones create models that are, by theorem, equivalent to $\sigma$. We search through the legal models formed by these remaining polyhedral transformations for a model that optimizes cache or parallelism. We pick one transformation and the corresponding model with iteration space $X'$.

Finally, we use a program called ClooG as a black box to go from our new model back to code. Since we used polyhedral transformations, ClooG is able to do this work for us.

### 2.3.5 Representations of Transformations and Data Flow

Both of the important sets we deal with are relations: transformation functions are relations and so are data flow relations. Because of the way our for-loop bounds are constructed, we can represent these relations symbolically.

Consider one of the affine pieces, $b : Y \to Y'$, of a polyhedral transformation $f$ for model $\sigma = (X, <_{\text{lex}}, \pi)$ and $Y \subsetneq X$. As we know, $b$ can be represented by a square matrix $A$ (and a constant vector). Suppose further that $Y$ is two dimensional and each vector in $Y$ is a statement executed inside of two indices $i, j$. Then,

$$b(\begin{bmatrix} i \\ j \end{bmatrix}) = A(\begin{bmatrix} i \\ j \end{bmatrix}) = \begin{bmatrix} \alpha_1 i + \beta_1 j \\ \alpha_2 i + \beta_2 j \end{bmatrix} + \begin{bmatrix} \gamma_1 \\ \gamma_2 \end{bmatrix}$$

for some integers $\alpha_l, \beta_l, \gamma_1, \gamma_2$. That is, we can represent the transformation symbolically as $[i, j] \to [\alpha_1 i + \beta_1 j + \gamma_1, \alpha_2 i + \beta_2 j + \gamma_2]$ with bounds that might look like $0 \leq i \leq n$ and $0 \leq j \leq m$. Of course, this is equivalent to the set $b$, but is a compact representation that can be reasoned about combinatorially.

The same occurs for dependence relations. Going back to figure 2.12 and looking only at statement 2, we can write the part of the dependence relation only dealing with integer tuples corresponding to statement 2 like so: $\{(1, i, j) \prec (1, i', j') : j < j', 0 \leq j < 1, 0 \leq i < n\}$. In fact, citing [Won10], the program analysis tool Omega can automatically look at code like that in figure 2.11 and generate the symbolic relation

$$D = \{(i, j) \prec (i', j') : i = i' + j', 0 \leq i, i' < n, 0 \leq j, j' < 1, (i, j) < (i', j')\}$$

by looking at the reads and writes of statement 2. This just asserts that the write of the first point $(i, j)$ is the value of the read of the second, later point $(i', j')$. Omega can then simplify this relation keeping it symbolic to something that very clearly expresses the data flow of the program and often enables easy optimization. Looking at $D$ alone will not help someone trying to manually optimize code.

### 2.3.6 Non-Polyhedral Transformations

Non-affine transformations come in many forms. Some of them are simply transformations $f$ that still preserve data flow but aren't affine or bijective.

Of course, maintaining data flow is sufficient but not necessary to ensure two programs are equivalent.[11] It is possible to swap two consecutive assignment statements that change data flow but make no difference. Take for example, two increments, `A[i] += B[j] + 2` and `A[i] += B[j+1] + 2` which occur sequentially. We can swap their order and the result is the same.[12]

Finally, there are changes to the source code which don't re-order the simple statements but instead change how memory is stored and therefore accessed. These are called **data space transformations**. As we walk down the linear order of simple statements, we also walk through memory with the reads of certain array references. If we had a sequence of statements that had the reads $A[i][j], A[i+1][j], \cdots, A[i+n][j]$ we might think that we're jumping around memory a lot. Usually, $A$ is stored row by row, so the reads of memory have terrible spacial locality. If, we restored $A$ column by column (i.e., took its transpose), then the memory cell next to $A[i][j]$ is $A[i+1][j]$. So, with the same order of execution, we improve locality. However, we could do both: we could perform some data space transformations and some iteration space transformations.[13]

## 2.4 Abstractions: Iterators & Records

So far we've discussed how to optimize programs for speed and maintain correctness. How do we achieve clarity? The code produced by ClooG is often unreadable and even manually produced code from the polyhedral model is confusing. We have two important answers and/or directions. These have been to a large degree, unexplored. There is plenty to do here (and hence, plenty of motivation for this work).

**Records** Records are a fundamental tool for abstracting data space optimizations.[14] Data space optimizations re-store arrays that are used in a dense matrix code to improve locality.

For example, consider the code in Figure 2.13. The inner-most loop sweeps through rows of $A$. In memory, $A[j][i]$ and $A[j-1][i]$ are $n$ cells apart. Hence,

[11] If it was, the halting problem would be solved.

[12] This situation actually applies to our example `syr2k`.

[13] If we did data space transformations that weren't injective (e.g., putting a $n \times n$ array inside an $n$-vector) we would need a more sophisticated way to describe legal transformations.

[14] Of course, there might be more ways that records help, but we have only seen this one in practice.

```
for (int i=1; i<n; ++i){
  for  (int j=1; j<n; ++j){
    A[j][i] += A[j-1][i] + 3;
  }
}
```

Figure 2.13: A code suitable for a data-space transformation.

for large $n$, the `LOAD` of $A[j][i]$ will not bring $A[j-1][i]$ into cache. This code has terrible locality. If we transposed $A$ into $At$ we could have much better locality. The statement would become equation 2.4.1 and the code would read through values of $At$ that are next to each other in memory.

$$At[i][j] + = At[i][j-1] + 3; \tag{2.4.1}$$

Now, this optimization could be abstracted with a record to hide the newly transposed array $At$. Namely, we could make a record named, say, `MatrixTranspose` that, given a matrix and its dimensions, constructs a transpose that can be indexed as though it was the original matrix. Internally, that overloaded index function would simply swap the indices. That is, the read of `Transposed[i][j]` would turn into, say `privateMemberArrVar[j][i]`. When an instance of this record is constructed, we could execute some fast transposing code to re-store the given matrix in `privateMemberArrVar`.

Let's zoom out and notice that data space transformations often lend themselves to easy abstractions by records. These records (1) transform the data upon construction and (2) provide adapters (which are usually computationally negligible) that maintain the original interfaces from accessing the data. In our example, our record performs a transpose on construction and swaps indices upon access.

**Chapel Iterators** Programming languages like Chapel provide certain special functions, called **iterators** which return (typically finite) sequences of values. If these values are integer tuples, these iterators can be used to provide the values of a loop index. To illustrate this, we will consider how we can use iterators (along with other language features) to abstract `syr2k` (without any optimizations).

When we convert `syr2k` into Chapel directly, we get the code in Figure 2.14. There are a few minor changes we make in the translation. The syntax {`1..n`} stands for what Chapel calls a range (or more broadly, a domain). This is a sequence of integers that can be used in the `for` constructs shown in the code. The second major change is that we start our arrays with index 1 instead of 0. This means that our indices go over different ranges. Instead of $i$ going from 0 to $n-1$, it goes from 1 to $n$. Knowing this, we can see that this code is directly the same as the original `syr2k` code.

We abstract our iteration order in the code in Figure 2.15. Here, we call a function, `org`, which yields, so to speak, a domain. This domain is a collection

```
for i in {1..n} do {
  for j in {1..i} do
    C[i,j] *= beta;
  for k in {1..m} do
    for j in {1..i} do
      C[i,j] += A[j,k]*alpha*B[i,k] + B[j,k]*alpha*A[i,k];

}
```

Figure 2.14: The Chapel version of `syr2k`.

```
for (i,k,j,stmt) in org(n,m) do {
  select stmt {
    when Stmt.betamult do {
      C[i,k] *= beta;
    }
    when Stmt.updateC do {
      C[i,j] += A[j,k]*alpha*B[i,k] + B[j,k]*alpha*A[i,k];
    }
  }
}
```

Figure 2.15: The iterator abstraction on `syr2k`.

of four tuples. The first three values in each tuple are integers. The last value is a enum that tells us which of the two statements we are executing. We use a select statement on `stmt` and based on that determine which statement to execute. In this abstracted code, it is possible for the iterator to re-order the execution of statements without showing the complexities of how that happens. (Of course, the iterator could also destroy the order of these statements. However, the fact that an abstraction technique could be used wrongly does not mean the technique itself is terrible.)

The iterator for this code is shown in Figure 2.16. It basically repeats the for loop in the original code but yields a four tuple that specifies which statement ought to be executed and with what indices.

This iterator abstraction technique comes from [BOH$^+$15] and has been shown to work on much more complex examples. There is one catch with iterators, however. The code in Figure 2.15 does not give us some default order or domain for each statement. That code alone does not specify the algorithm used in the original `syr2k` code.[15] Hence, by convention, the specification of the algorithm is the combination of this code and the `org` iterator. That is, the original translation does not hide details so much as organize them. Once a programmer has looked at both the code in Figure 2.15 and the iterator, she understands the specification. After this point, if there was a version of the abstracted code with some new iterator `optimalIter`, she would not need to look inside this iterator to understand that it abstracts an optimization that

[15]Whether there is a better abstraction is an open question.

```
enum Stmt {betamult, updateC};

iter org(n:int, m:int): (int, int, int, Stmt) {
  for i in {1..n} do {
    for j in {1..i} do {
      yield (i,j,0,Stmt.betamult);
    }
    for k in {1..m} do
      for j in {1..i} do {
        yield (i,k,j,Stmt.updateC);
      }
  }
}
```

Figure 2.16: The basic iterator for `syr2k`.

```
for (i=0; i<n; i++)
  A[i] = A[n-1-i]*B[i];
```

Figure 2.17: An example dense matrix code from [Won10].

maintains the data flow that the original iterator had specified.

## 2.5 Seminal Papers & Optimization Tools

In this section, we look at several key papers or reports in this research area and connect them to our present work on `syr2k` [Won10] [Won02] [BHRS08] [LCL99]. Some papers present optimization techniques. For those papers, we apply those techniques to `syr2k`. One paper [BHRS08] presents an automatic optimizer called PLuTo. We cover the basic ideas underlying the algorithm of PLuTo. One report – the Omega Retrospective [Won10] – simply covers several key tools or ideas used in this realm of optimization.

### 2.5.1 The Omega Retrospective

The central thing we use from the omega retrospective [Won10] by Dave Wonnacott is a symbolic representation of dependence relations and transformations. In Section 2.3.5 we saw an example of a symbolic representation for a data flow relation and an overview of how this is useful to us. Here we discuss this topic in a bit more depth. Specifically, we (1) look at an intuitive example of making a symbolic data flow relation, (2) discuss the general way to use the Omega library (which manipulates symbolic representations of sets of relations) to determine data flow and (3) see how programmers can use these symbolic relations for optimizations.

**Example: A Symbolic Data Flow Relation**

Taking an example straight from [Won10] consider the dense matrix code in figure 2.17. Since there is only one statement, we number all instances of this statement by the one-tuple of the index; $(i)$ corresponds to the simple assingment executed when the index is $i$. The dependence relation is then a relation $D$ between indicies such that $(i, j) \in D$ if and only if the following conditions are satisfied.

- The indicies $i, j$ are in range. That is, $0 \leq i, j < n$.
- The indicies correspond to statments executed in the order $i$ then $j$. Because the indicies are ordered lexicographically, this is just the same as saying $i < j$.
- The statement corresponding to index $i$ reads some array reference (e.g., $A[3]$) that is written to by the statement corresponding to index $j$ (or the statement of index $i$ writes some array reference that is read by the statement of index $j$; or, of course, both statements write to that index).
  - For the read-write situation, this means $n - 1 - i = j$.
  - For the write-read situation, this means $i = n - 1 - j$.
  - For the write-write situation, this means $i = j$.

We can represent all of this symbolically:

$$\begin{aligned} D := \{(i, j) : (0 \leq i, j < n) \wedge (i < j) \\ \wedge ((n - 1 - i = j) \vee (i = n - 1 - j) \vee (i = j))\} \end{aligned}$$

We should be able to see from the in-order constraint that we never have $i = j$. The other two constraints are algebraically the same. Hence, we can simplify:

$$D := \{(i, j) : (0 \leq i, j < n) \wedge (i < j) \wedge (n - 1 - i = j)\}.$$

Note that this process still works if we had multiple statements that our tupling function $I$ from Section 2.3 would give integer tuples of different lengths.

**Using Omega To Manipulate Symbolic Relations**

We can do this symbolic calculation of data flow with the help of algorithms in the Omega library.[16] These algorithms are implemented as functions and can be used by a command line tool called the Omega calculator. We only talk about these operations abstractly in the following discussion.

Here is how we can generically calculate data flow using Omega's algorithms.

[16]See `http://www.cs.umd.edu/projects/omega/` to learn more.

1. Label all array accesses (e.g., bits of code like `A[n-1-i]`) with unique letters.
    - For our example, call `A[i]` $a$, call `A[n-1-i]` $b$ and call `B[i]` $c$.
2. Find each pair of *distinct* accesses $(x, y)$ where at least one of $x, y$ is a write and both $x, y$ access the same array.
    - For our example, the only pairs are $(a, b)$ and $(b, a)$.
3. For each pair $(x, y)$ find the data flow relation for that pair, $D_{x,y}$ by following the following steps.
    (a) Find the index relations for the accesses, $M_x$ and $M_y$. The *index relation* for array access $q$, $M_q$ is the relation of pairs $(i, z)$ where $i$ is an integer tuple that corresponds to some instance of the statement that has the access $q$ and $z$ is an integer that is the value of the index for the access $q$ at that instance. These relations are symbolic.
    - For example, for $b$ at $i = 3$, the index of `A` is $n - 1 - 3$. So, $M_b = \{(i, n - 1 - i) : 0 \leq i < n\}$. Similarly, $M_a = \{(i, i) : 0 \leq i < n\}$.

    (b) Find the forward-in-time relation between integer tuples that identify the statements of $x, y$ respectively: $T_{x,y}$.
    - For example, $T_{a,b}$ and $T_{b,a}$ are $\{(i, j) : (i < j) \wedge (0 \leq i, j < n)\}$.

    (c) Compute $J = M_x \cdot (M_y)^{-1}$, the pairs of integer tuples of the statement for access $x$ and $y$ where the array accesses are the same. Note that the inverse of a relation $R^{-1}$ simply reverses all pairs in that relation. Recall that the join $S \cdot T$ of two relations $S, T$ where $S \subseteq A \times B$ and $T \subseteq B \times C$ is $\{(a, c) \in A \times C : \exists b \in B, ((a, b) \in S \wedge (b, c) \in T)\}$. The inverse and join operations can be done on our symbolic sets via algorithms provided by the Omega library. The library also has sophisticated algorithms to simplify our constraints (provided they are affine) to make the symbolic representation of the set have less variables and less constraints.
    - For example, $M_a \cdot (M_b)^{-1} = \{(i, j) : (i = n-1-j) \wedge (0 \leq i, j < n)\}$. Also, $M_b \cdot (M_a)^{-1} = \{(i, j) : (n-1-i = j) \wedge (0 \leq i, j < n)\}$, which is the same set.

    (d) Compute $J \cap T_{x,y}$ and this is our desired dependence for this pair of accesses $D_{x,y}$. It is all pairs of indices where the corresponding statement instances refer to the same memory location, one of which is a write and are in order of execution.
    - For both pairs in our example, this set is $D = \{(i, j) : (i < j) \wedge (i = n - 1 - j) \wedge (0 \leq i, j < n)\}$.
4. Compute $D$ as the union of all the dependence relations $D_{x,y}$ for each pair of array accesses we care about.

- Since both $D_{a,b} = D_{b,a} = D$, our dependence relation is just $D$. We note that the Omega library simplifies $D$ to be

$$D = \{(i, n-1-i) : (0 \le i) \wedge (2i \le n-2)\}.$$

Intuitively, the second constraint just means $i < \frac{n}{2}$.

**Manual Optimization Using Symbolic Relations**

There are two key uses of these symbolic ways of representing relations. First, we could represent polyhedral transformations as symbolic relations. Recall that formally, a polyhedral transformation is a specific kind of function between sets of integer tuples (representing statement instances). The lexicographic order is the default order in which we execute those instances. So, we could represent such functions (which are relations) symbolically. For example, in the code in figure 2.18, we could interchange the $i$ and $j$ loops (because there is no data flow). We could represent this symbolically as

$$\{((i,j,k),(j,i,k)) : (0 \le i,j,k < X)\}.$$

```
for (int i=0; i<X; i++)
  for (int j=0; j<X; j++)
    for (int k=0; j<X; z++)
      A[j][k] += 1
```

Figure 2.18: A dense matrix code for loop interchange.

Second, we can use symbolic representations of the dependence relation of a dense matrix code to identify *conditional dependencies*. That is, we can identify affine constraints using loop bounds like $X$ in the example above and $n$ in figure 2.17 that gives us a criterion for when we have a dependency and when we do not. This enables us to give a polyhedral transformation that is proved correct symbolically!

For example, look at the simple example in figure 2.17 again. Recall that the data flow is (basically)

$$D = \{(i, n-1-i) : (0 \le i) \wedge (i < \frac{n}{2})\}.$$

**Proposition 2.5.1.** *There are no pairs $(i,j) \in D$ such that $j < \frac{n}{2}$.*

*Proof.* Consider the following inequalities following from the constraint $i < \frac{n}{2}$

from the symbolic representation of $D$.

$$\begin{aligned} i &< \frac{n}{2} \\ -i &> -\frac{n}{2} \\ (n-1)-i &> (n-1)-\frac{n}{2} \\ (n-1-i) &> \frac{n}{2}-1 \\ (n-1-i) &\geq \frac{n}{2} \end{aligned}$$

Hence, looking at the symbolic form of $D$, all possible $j$ must be of the form $(n-1-i)$ which are greater than or equal to $\frac{n}{2}$. $\square$

Also, the following proposition is obvious.

**Proposition 2.5.2.** *There are no pairs $(i, j) \in D$ such that $i \geq \frac{n}{2}$.*

The first proposition allows us to execute all points $i < \frac{n}{2}$ in parallel (before doing anything else) because we cannot have a dependence between any two points satisfying that constraint. The second allows us to execute all points satisfying $j \geq \frac{n}{2}$ in parallel (after all other points are executed) since we can never have a dependency between any two of these points. Since all points satisfy exactly one of these two constraints, we can optimize this code by executing the first half in parallel and then executing the second half in parallel. Symbolically, we have the transformation

$$\{(i, 2i \text{ div } n) : 0 \leq i < n\}.$$

### 2.5.2 PLuTo's Automatic Optimizations

In this subsection, we look at a key paper by Bondhugula et al. presenting the automatic polyhedral-based optimizer named pluto [BHRS08].[17] Pluto a C-to-C compiler that indicated dense matrix codes and automatically optimizes them within the polyhedral framework. Our goal is to communicate the basic idea of the algorithm of this tool.

We begin with understanding how transformations are represented in pluto's framework and how correctness is discussed. Then, we present the basic idea behind the algorithm.

**Representing Transformations** In pluto's framework, polyhedral transformations are represented by a collection of one-dimensional affine transformations.

[17]The name is an abbreviation and should technically be written "PLuTo" instead of "pluto" but this is pedantic.

**Definition 2.5.3.** Let $P$ be a dense matrix code with statement $S_i$. Suppose that the instances of $S_i$ are represented by $k$ tuples (when we apply our tupling function $I$). A **one-dimensional affine transform** $\phi_i : \mathbb{Z}^k \to \mathbb{Z}$ is a function defined by

$$\phi_i(\vec{i}) = \vec{c} \cdot \vec{i} + c_0$$

for some $\vec{c} \in \mathbb{Z}^k$ and $c_0 \in \mathbb{Z}$.

These affine transforms represent the re-ordering of statement instances. If $\phi((0,0)) = 3$ and $\phi((0,1)) = 4$ then we execute the instance of $S_i$ with index vector $(0,0)$ before the instance with index vector $(0,1)$. If the transform is *not* injective, then we execute some instances at the same time.

**Definition 2.5.4.** Let $P$ be a dense matrix code with statements $S_1, \cdots S_k$. Then we say $\phi = \{\phi_1, \cdots, \phi_k\}$ is a **affine transform for** $P$.

**The Data Dependence Graph** Recall that the correctness of polyhedral transforms is based on data flow. In this framework, this is discussed through something called the data dependence graph. Note that this is actually a *multi-graph*: a graph which can have multiple edges between two vertices.

**Definition 2.5.5.** Let $P$ be a dense matrix code with statements $S_1, \cdots S_k$. The **data dependence graph** (denoted DDG) of $P$ is the directed multi-graph whose vertices are statements and whose edges are dependencies between index vectors that represent instances of statements.

For example, say $S_1$ had an instance represented by the point $(0,1)$ and $S_2$ had an instance represented by the point $(0,1,3)$. By lexicographical ordering, the first instance is executed before the second. Now, suppose further that the second point depended on the first.[18] Then, there would be an edge $e$ in the DDG of $P$ going from $S_1$ to $S_2$ with the tag of the two instance vectors $(0,1)$ and $(0,1,3)$.

In this formulation, the paper states a lemma that defines correctness.

**Lemma 2.5.6.** *Let $P$ be a dense matrix code and $\phi = \{\phi_1, \cdots, \phi_k\}$ be an affine transform for $P$. Then, $\phi$ is **legal** if for every edge $e$ between $S_i, S_j$ for iteration vectors $s, t$ in the DDG of $P$, the following holds.*

$$0 \leq \phi_i(t) - \phi_j(s)$$

Even if $\phi$ is not injective (and executes some points in parallel), this condition suffices to say the entire transformation preserves data flow.

**The Idea of The Algorithm** The basic idea of the algorithm is to construct each $\phi_i$ as to minimize the timing distance between dependencies. What does that mean?

[18]This means one of our three cases for dependencies between simple statements applies.

**Definition 2.5.7.** The **cost function** for a dense matrix code $P$, transformed by some legal $\phi$, with some edge $e$ between vector $s$ of $S_i$ and $t$ of $S_j$ in its DDG is the following.

$$\delta_e(s,t) = \phi_j(t) - \phi_i(s) \tag{2.5.1}$$

If $\phi_i, \phi_j$ represent the order in which points are executed, then $\delta_e$ represents the re-use distance. That is, it represents the number of other statements – which could over-write the values brought in cache from executing point $s$ – that occur before the dependent reference in $s$ is used in $t$. If $\delta_e$ is small enough[19] then we have good locality.

It could be the case that $\phi$ groups the points into tiles. If the range of $\phi$ is six, for example, we could group all instance of statements into six tiles. All the points in each tile could be executed in parallel. (Remember, $\phi$ is a legal transform.) However, we would need to wait until the previous tile is finished to execute the next one. In this case, $\delta_e$ would represent the **communication volume** between two tiles. That is, it would serve as a metric for how many tiles worth of final values we need to keep track of as we pass between tiles. If this and the tile sizes were small enough, we could keep all the needed results of each tile in cache allowing us to avoid accessing main memory to get data from the execution of a previous tile.

The key idea of the algorithm is to construct a transform $\phi$ that both tiles and then orders points within tiles to optimize locality. The algorithm looks at the edges between two statements $\{S_i, S_j\}$, and uses properties of $\delta_e$ to construct the relevant parts of $\phi_i, \phi_j$ that minimize $\delta_e$. It then walks through all the other pairs of vertices, iteratively building a $\phi$ that minimizes (with caveats) all the $\delta_e$ cost functions.

### 2.5.3 Time Skewing

In this subsection we briefly present a technique called *time skewing* from [Won02] by David Wonnacott. This technique applies to `syr2k`.

Time skewing takes dense matrix codes of a specific form and optimizes them to have *scalable locality.* A dense matrix code has scalable locality if we have the property that as we increase the problem size (e.g., input matrix sizes), we increase locality without ever making memory bandwidth slow down the computation. Well, what does that mean? We can define locality, the amount of reuse of values in cache, as

$$\frac{(\text{\# Floating point Ops})}{(\text{\# Main memory accesses})}.$$

Say we have some dense matrix code parameterized by the size of one matrix $n$ and call it $P(n)$. Say $P$ has a locality that is always less than 3 independent of the size of $n$. Say a processor does 10 floating point operations per second and 2 memory accesses per second. So, on average, when $P$ is being executed and 3 floating point operations are performed, (at least) one main

[19]This depends on the specific code.

memory access is performed. The floating point operations took 0.3 seconds and the main memory access took 0.5 seconds. So, in general, no matter how much we increase $n$, we will always spend more time on memory accesses than floating point operations. Hence, even if we increase the rate processors do floating point operations, we are bogged down by the rate memory operations occur. It is this situation this technique avoids for certain kinds of dense matrix codes.

We will (1) define the class of problems $P$ time skewing works on and note how it might apply to `syr2k`; (2) introduce some definitions to more precisely define scalable locality for our class of problems and (3) give a very brief overview of the algorithm.

### Time Step Stencils

**Definition 2.5.8.** A **time step stencil** is a dense matrix code in which the following hold.

- Each assignment replaces an array reference by a function of its neighbors.
- There is a loop called the time loop $t$ such that for any array read in loop iteration $t+1$ that memory location was last written to (if at all) in loop iteration $t$.

```
// define cur[1:T-1][0] to be equal to cur[0][0] and
// cur[1:T-1][N-1] to be cur [0][N-1]
for (int t=0; t<T; t++)
  for (int i=1; i<=N-1; i++)
    cur[t+1][i] = 0.25 * \
                    (cur[t][i-1] + 2*cur[t][i] + cur[t][i+1])
```

Figure 2.19: A time stencil taken from Figure 10 in [Won02].

For example, consider the code in Figure 2.19. Here, the matrix `cur` has dimension $T \times N$. On each iteration of $t$, the whole $(t+1)$th row of `cur` is written to except for the first and last element. The first and last column of `cur` is never touched. On iteration $t$, the assignment reads array elements from row $t$ of `cur` which was written to on iteration $(t-1)$ (if at all). Hence the outermost loop is a time loop where all array reads were last written to on the last iteration. Second, the write to `cur[t+1][i]` is a function of the neighbors of that array reference.

We can actually see a time step stencil inside of `syr2k`. As commented in Figure 2.20, the $k$ loop can be a time loop. If the current iteration is $(i, k, j)$, the last write to `C[i][j]` happened on iteration $(i, k-1, j)$ if at all. Further, `C[i][j]` is its own neighbor. Hence, it might be useful to apply time skewing to this part of `syr2k`.

```
for i = 0 to n do {
  for j = 0 to i do {
    C[i][j] := C[i][j] * beta
  }
  for k = 0 to m do { // TIME LOOP
    for j = 0 to i do {
      // Let (i,k,j) be the current iteration.
      // C[i][j] was last written to on (i,k-1,j).
      C[i][j] := A[j][k]*alpha*B[i][k] + \
                 B[j][k]*alpha*A[i][k] + C[i][j]
    }
  }
  }
}
```

Figure 2.20: The time stencil portion of `syr2k`.

## Definitions

To have a clear picture of scalable locality is, we need to state a few definitions and understand them in the context of time stencil programs. These definitions are not fully rigorous but have enough detail for our purposes.

**Definition 2.5.9.** The **machine balance** of a processor is

$$\frac{\text{(processor rate of FP operations)}}{\text{(processor rate of memory operations)}}.$$

The machine balance represents the locality at which the amount of time a processor spends on floating point operations equals the amount of time spent on main memory accesses. Another useful definition is the compute balance:

**Definition 2.5.10.** The **compute balance** of a dense matrix code $P$ is

$$\frac{\text{(\# FP operations in } P\text{)}}{\text{(\# live values at the start and end of } P\text{)}}.$$

Generally, the "live" values – values in arrays that are inputs and outputs to the dense matrix code – must involve main memory reads of the inputs and writes of the outputs. So, the compute balance is like an upper bound on the locality of a dense matrix code.

If the compute balance of $P$ is (for all sizes of inputs to $P$) less than the machine balance of the processor, we can never reach a locality where the execution of $P$ spends more time on the floating point operations than the memory accesses. Hence, we need the compute balance to be more than the machine balance. This is one of the goals achieved by time skewing.

**Definition 2.5.11.** The **average locality** of a dense matrix code $P$ is

$$\frac{\text{(\# FP operations in } P\text{)}}{\text{(\# main memory accesses of } P\text{)}}.$$

The denominator of average locality includes (typically) all "live" values. But it also includes temporary values:

**Definition 2.5.12.** A **temporary value** of a dense matrix code $P$ is an array reference that is not an input or output of $P$.

This means the denominator of average locality is at least the number of live values at the start and end (because those memory locations are read and written to) and all the times we write and then read temporary values from main memory. For a given code $P$, to achieve a locality at or close to the compute balance we need to keep (all or most) temporary values exclusively in cache. This is the second goal achieved by time skewing.

**Definition 2.5.13.** Let $P(n)$ be a dense matrix code where the problem size is $n$. Suppose the average locality $L(n)$ of $P$ depends upon the problem size. Suppose the same holds for the amount of cache assumed to be available $C(n)$. We say $P$ has **scalable locality** if $O(n) \leq L(n) \leq O(n)$ and $C(n) < O(n)$.

Note that if the locality increases with the problem size, the compute balance has to be larger than the machine balance for sufficiently large inputs. In other words, a code with scalable locality entails a large enough compute balance. In summary, we achieve scalable locality (which entails a large enough compute balance) for some time stencil codes through time skewing.

### Rough Overview of The Algorithm

Time skewing takes a time stencil code and tries to optimize it to a code that has scalable locality. The basic idea of the algorithm is presented in Figure 2.21. To explain this algorithm, we take a cursory glance at an example of optimizing the code in figure 2.19. This example is taken directly from [Won02].

Suppose we have a time stencil $P(\vec{x})$ with a vector of inputs $\vec{x}$.

1. Block the time loop by $s$ so that the compute balance of each tile can be scaled up by increasing the tile size or an input parameter in $\vec{x}$.
   (a) If the compute balance is only scaled up by $s$, pick $s$ so that the compute balance of each tile is larger than the machine balance of the processor.
2. Re-order the statement instances in each tile so that the number of *simultaneously live temporary values* grows less than linearly with the problem size (i.e., less than $O(\vec{x})$).

Figure 2.21: Rough description of time skewing.

Instead of presenting the optimized version of Figure 2.19, we present an iteration space graph (from [Won02]) in figure 2.22. In this figure, the points are instances of the lone statement (`cur[t+1][i] = ..`). The tiles are rectangles that are executed from left to right. In general, dividing the iteration space into polygons that are executed in some linear order is called **tiling**. When we tile by rectangles (or $n$ dimensional "rectangles") this is called **blocking**. In each tile, we execute the points that form a right triangle in the bottom left and then execute diagonals that go from top left to bottom right. For example, in the second tile we first execute the triangle $(2,1),(2,2),(3,1)$. Then we execute the diagonal $(2,3),(3,2)$ followed by the diagonal $(2,4),(3,3)$ and so on.

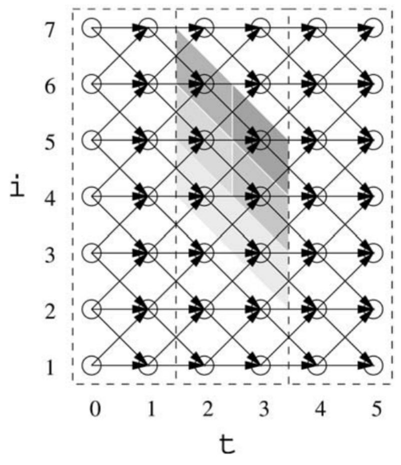

Figure 2.22: A time skewing example taken from [Won02].

This gives us scalable locality. Consider the compute balance of each tile. Each tile does $s(N-1)4$ floating point operations, or about $4sN$ operations. The number of floating point operations in each tile is the number of instances of the statement (the number of circles in the tile) times the number of operations per statement. The statement does four operations: two multiplications and two additions. The number of circles in the tile is $s(N-1)$ or about $sN$. The $i$ loop goes from 1 to $N-1$ so the height of each tile is about $N$. The width is $s = 2$.

The tile has about $2N$ live values at the end and at the start. The values that are live at the start are visually circles outside the tile that have dependencies that go into the tile. This is $N-1$ circles. For example, in the second tile, the entire column of circles with $t = 1$ have dependent arrows that enter the tile. Similarly, the values that are live at the end of the tile are all the circles in the tile that have arrows that exit the tile. These are also $N-1$ circles. For the second tile, these are all the circles with $t = 3$.

Hence, each tile has about $\frac{4sN}{2N} = 2s$ compute balance. As per our algorithm, our blocking can be scaled by the tile size. Let the machine balance of the processor we plan to run this on be $m$. Then, we will pick an $s$ such that $2s > m$.

Without proving this, we claim that because of the way we execute points within each tile, we only use $2s+1$ temporary values simultaneously. Typically, the machine balance $m$ is small enough that for our choice of $s$, $2s+1$ can fit in cache. Hence, for each tile we have an average locality that is about the compute balance, $2s$. So the overall average locality is $2sT$ which is $O(T)$. Since our needed cache size is some fixed constant, we have scalable locality as we've defined it above: the average locality grows linearly with the problem size while the needed cache size grows less than linearly with the problem size.

### 2.5.4 Minimizing Communication

Amy Lim and Gerald Cheong et. al. presented an algorithm for parallelizing dense matrix codes across multiple processors in [LCL99]. The goal was to maximize parallelism while minimizing *communication*, the amount of data that needs to flow between processors. Since their framework and algorithm are quite complex and sophisticated, we only cover the basic ideas behind the work. Then, we walk through the application of their algorithm on `syr2k`.

**Representing Space and Time Partitions** Transformations are represented via affine partition mappings.

**Definition 2.5.14.** Let $s$ be a statement in dense matrix code $P$ with iteration vectors in $\mathbb{Z}^n$. An **affine mapping of** $s$ (of dimension $m$) is a function $\phi_s(\vec{i}) = C_s\vec{i} + \vec{c_s}$ for some $m \times n$ integer matrix $C_s$ and some $c_s \in \mathbb{Z}^m$. An **affine partition mapping of** $P$ (of dimension $m$) is a collection of affine mappings for each of the $k$ statements in $P$: $\phi = [\phi_1, \cdots, \phi_k]$.

We use $\phi$ in two ways. First, we can represent iteration space transformations via $\phi$; it can represent a re-ordering of the points. Second, we can use $\phi$ to represent assigning different iterations to different processors (or cores). If we assign each processor or core to some vector in the range of $\phi$, then $\phi$ represents a way to divide the iteration instances among processors. We call such partitions, **space partitions**.

**Synchronization-Free Parallelism** One goal with assigning different instances to different processors is having no synchronization.

**Definition 2.5.15.** A space partition $\phi$ of $P$ is **synchronization free** if for any two iteration vectors $\vec{i}, \vec{j}$ where the simple statement of the second depends on the simple statement of the first $(\pi(\vec{i}) \rightsquigarrow \pi(\vec{j}))$,

$$\phi(\vec{i}) - \phi(\vec{j}) = 0.$$

This definition simply states the condition that a space partition sends all instances that depend upon each other to the same processor. Hence, with such a space partition, we could run all processors in parallel without any issues involving communication between processors.

Of course, the trivial partition ($\phi(x) = 1$) always satisfies this condition. We might guess that what we want is a space partition that sends iteration vectors to as many processors as possible while maintaining this condition. However, this often proves to be too small. The goal of [LCL99] is to relax this condition to have a one-dimensional space partition and have the property $\phi(\vec{i}) - \phi(\vec{j}) = \lambda$ for some fixed constant $\lambda$ for a specific category of pairs of statements. Then, through a rather complex framework, we maximize the number of processors we use while minimizing the amount of data that needs to flow between processors. Often, this gives way to pipelined parallelism.

**Applying The Algorithm to `syr2k`** As stated, we do not go through the algorithm in detail, but instead walk through the application to `syr2k`. We will enumerate the steps of the algorithm that apply to `syr2k` and perform those steps while explaining our work. For reference, we have repeated the original `syr2k` code here.

```
 1  void syr2k(int n, int m,  double alpha,  double beta,
 2    double C[N][N], double A[N][M], double B[N][M]) {
 3    int i, j, k;
 4    // Note: N = n and M = m; these are configuration constants.
 5    for (i = 0; i < n; i++) {
 6      for (j = 0; j <= i; j++)
 7        C[i][j] *= beta;
 8      for (k = 0; k < m; k++)
 9        for (j = 0; j <= i; j++){
10            C[i][j] += A[j][k]*alpha*B[i][k] + \
11                       B[j][k]*alpha*A[i][k];
12        }
13    }
14  }
```

Figure 2.23: Referencing the original `syr2k` code.

1. Let $G$ be the data dependence graph[20]. Let $G'$ be the DDG without allowing multiple edges between two vertices. Break $G'$ into strongly connected components (i.e., subgraphs where any two vertices have an edge between them).

   - Let's call the statement in line 7 $S$ and the statement in line 10 $T$. The graph $G'$ has two edges: one from $S$ to $T$ and one from $T$ to itself. The strongly connected components are just $\{S\}, \{T\}$.

2. For each strongly connected component, try to find a synchronization free space partition. Try to make the difference between dependent instances across components a fixed constant vector.

[20]See Section 2.5.2.

- We can identify all instances of $S$ with the tuple of indices $(i, j)$. Each of these can go to its own processor. So, symbolically, our affine space mapping for the first strongly connected component is just
$$\{[i, j] \to [i, j] : (0 \le i < n), (0 \le j < i)\}.$$
- We can identify all instance of $T$ with $(i, k, j)$ tuples. Looking at Section 3.2, we can see that the data flow in this strongly connected component is just $\{[i, k, j] \to [i, k', j] : k < k'\}$. Hence, we can map each iteration to a processor based on its $i, j$ indices. Roughly, we have the mapping
$$\{[i, k, j] \to [i, j]\}.$$
- From Section 3.2, we know there is only one dependence between these two components. Namely, statements of $T$ identified by $(i, k, j)$ depend on statements of $S$ identified by $(i, j)$. Under the space mappings we found, we see that both map to the vector $(i, j)$. Hence, their difference is $\vec{0}$.

3. See if the fixed constant can be made $\vec{0}$.

   - We have already achieved this.

4. Sequentially order the instances in each space partition by following data dependencies.

   - For a given space partition for some fixed $i$ and $j$ we only have one possible order. We execute $S$ with indices $(i, j)$ Then, we execute $T$ with indices $[(i, 0, j), (i, 1, j), \cdots, (i, m, j)]$ in that order.

Of course, we don't want to make a different thread for each possible $(i, j)$ pair (because there would be way too many). Hence, we distribute collections of these pairs over different cores. Practically, this algorithm gives us several possible optimized versions of `syr2k`. We could make either the $i$ or $j$ or both loops parallel in both versions of `syr2k` depicted in Figure 2.24. All those 6 codes represent the optimized execution order we found from applying the algorithm.

```
// i loop before j loop
for (i = 0; i < n; i++) {
  for (j = 0; j <= i; j++) {
      C[i][j] *= beta;
      for (k = 0; k < m; k++){
          C[i][j] += A[j][k]*alpha*B[i][k] + \
                     B[j][k]*alpha*A[i][k];
      }
  }
}

// j loop before i loop
for (j = 0; j < n; j++) {
  for (i = j; i < n; i++) {
      C[i][j] *= beta;
      for (k = 0; k < m; k++){
          C[i][j] += A[j][k]*alpha*B[i][k] + \
                     B[j][k]*alpha*A[i][k];
      }
  }
}
```

Figure 2.24: Possible optimizations of `syr2k`.

# Chapter 3

# Analysis

In this chapter we use the background we've developed to analyze `sy2k`. We inspect the original code and explain why the performance is poor. We determine the data flow of the program. We look at existing optimizations of `syr2k` including ones produced by pluto. We look at what these existing optimizations consider and what they lack. Finally, we list several decent directions for performance optimization. As we we will see in Chapter 4, some of these ideas are fruitful, though they could use improvement.

## 3.1 Poor Original Performance

In this section we give a reasonable answer to why the original `syr2k` code is so slow. It's useful to take another look at our formalized version of the code.

```
for i = 0 to n do {
  for j = 0 to i do {
    C[i][j] := C[i][j] * beta ; // Statement 1
    // represented by (i,0,j)
  }
  for k = 0 to m do {
    for j = 0 to i do
      C[i][j] := A[j][k]*alpha*B[i][k] + // Statement 2
                 B[j][k]*alpha*A[i][k] + C[i][j] ;
                 // represented by (i,1,k,j)
    }
  }
}
```

Figure 3.1: Our formalized syr2k code.

Informally, we remark that as we increase the size of $n$ and $m$ so that matrices $A$ and $B$ cannot fit in cache, our performance drops by a factor of four. Even without considering parallel versions of this code, there is room for improvement in the single core case: it seems the use of cache is quite poor. Let's see why.

**Locality Problems With Statement 2** Putting the first statement aside for a moment, we see some performance issues with the second statement. Note that the reads of $A$ and $B$ are mirrored. Since those matrices have the same dimensions, we will only look at how the memory of $A$ is traversed as simple statements are executed. We denote these simple statements $S(i, k, j)$ for fixed literals $i, k, j$. For example, $S(1, 2, 3)$ is the following simple statement.

```
C[1][3] := A[3][2]*alpha*B[1][2] + B[3][2]*alpha*A[1][2] + C[1][3] ;        (3.1.1)
```

In our memory system, $A, B$ are stored row by row in memory. If we imagine memory is a long tape the order of values is $\langle A[0][0], A[0][1], \cdots, A[0][m-1], A[1][0], A[1][1], \cdots, A[1][m-1], \cdots A[n-1][m-1]\rangle$. As we go from $S(i, k, j)$ to $S(i, k, j+1)$ we read down columns of $A$ via the read $A[j][k]$. The value of $A[j][k]$ is $m$ values before $A[j+1][k]$. Thus, for large $m$, we won't pull enough of $A$ into cache on the read of $A[j][k]$ to have $A[j+1][k]$ in cache. We will have cache misses on these jumps down the columns of $A$. As $j$ becomes large (which happens for large $n$), the number of column jumps we do increases. In other words, with a large $n$ and large enough $m$ to not fit a value in a column above in cache, we have terrible spacial locality.

We also have bad temporal locality. As we move from $S(i, k, 0)$ to $S(i, k+1, 0)$, for large enough $i$, the index $j$ is larger than the block count. All the parts of rows of $A$ pulled into cache from the early cache-misses $A[j][k], A[j+1][k], \cdots, A[j+l][k]$ (for some small $l \in \mathbb{N}$) are overwritten. Hence, on $S(i, k+1, 0)$ when we do the reads $A[0][k+1]$ and $A[1][k+1]$, we have cache-misses.

This poor temporal locality applies to the other read as well. As $i$ gets large, $j$ gets large which might lead to "wasted" cache misses for the read $A[i][k]$. Say we read $A[i][k]$ at $S(i, k, 0)$. Before the loop index $k$ increments, if $j$ is too large, the reads $A[j][k]$ and $B[j][k]$ for $j = [0, i]$ will fill up all cache blocks so that $A[i][k+1]$ is no longer in cache when we execute $S(i, k+1, 0)$.

## 3.2 Data Flow & Structure

**The First Statement Alone** Take a look at Figure 2.10. The first statement writes to $C[i][j]$ and that is also its only read. Denote each instance by $T(i, j)$. Clearly, $T(i, j)$ only depends on itself. Hence there is no data flow from any two distinct instances $T(i, j)$ and $T(i', j')$.

**The Second Statement Alone** As before, we number and label the simple assignments as $S(i, k, j)$. Consider for some symbolic constants $i, k, j$ the simple assignment $S(i, k, j)$. The only array reference that it reads that is written to is $C[i][j]$. Intuitively, we know that $C[i][j]$ is written to in $S(i, k', j)$ for all $k'$ such that $0 \leq k' < k$. Let $S$ be the set of all the valid tuples of iteration indices $(i, k, j)$. Let $\sigma = (S, <_{\text{lex}}, \pi)$ be our model of the code (with $\pi$ being the typical conversion function). Now, symbolically, the data flow relation is

$$D = \{(i, k, j) \rightsquigarrow (i, k', j) : k' < k, (i, k, j), (i, k', j) \in S\}.$$

```
dotwani:~$ /home/davew/bin/iscc
S := [N,M] -> { [i,k,j] : 0 <= i < N && 0 <= k < M && 0 <= j <= i };
AllPairs := unwrap (S cross S);
FwdInTime := { [i,k,j] -> [i',k',j'] : (i < i')
  || (i = i' && k < k') || (i = i' && k = k' && j < j') };
AtLeastOneWriteDependency :=  { [i,k,j] -> [i',k',j'] : i = i' && j = j' };
PairsInTimeOrder := FwdInTime * AllPairs;
DataFlow := AtLeastOneWriteDependency * PairsInTimeOrder ;
DataFlow ;
$0 := [N, M] -> { [i, k, j] -> [i' = i, k', j' = j] :
  i < N and k >= 0 and 0 <= j <= i and k < k' < M }
dotwani:~$
```

Figure 3.2: The ISCC calculation of `syr2k`'s data flow.

Using the tupling function $I$, this is

$$D = \{(i, 1, k, j) \rightsquigarrow (i, 1, k', j) : k' < k\}.$$

However, even this doesn't tell the full story. Since $S$ is an increment, maintaining data flow doesn't matter. If we increment a value by 3 and then 2 or by 2 and then 3, overall, we still increment it by 5 (commutativity of addition). The only concern we do have is with parallelizing the code. Increments need to be atomic. We should not have two increments occur such that both read before either writes.

**Data Flow of Statement Two Via ISCC** We can confirm our intuitive analysis with the ISCC tool [isc19]. Our calculation is shown in Figure 3.2. Here is the breakdown of what we do:

- First, we construct the symbolic set of all possible iterations of our central statement which we label `S`.
- Then, we construct the lexicographic order on these pairs with `FwdInTime`. This is the order in which the instances execute.
- Then, we construct all pairs of indices of statement `S` which both have an array access to the same element of `C` and at least one of those accesses is a write. This relation is named "At least one write dependency". Since `S(i,k,j)` both reads and writes to `C[i][j]`, this relation must be all pairs where the $i, j$ indices match. In section 2.5.1 we did this by creating an index relation for each access in `S`. All these relations would be $\{(i, k, j) \to (i, j) : \text{(the indices are in range)}\}$. Finding the pair of indices for these accesses that write to the same place would end up with the same relation that we wrote down.
- Finally, we simply took the "intersection" of these three sets to find all pairs of indices in the correct time order that share a memory access

(one of which is a write) and are in range. This is the `DataFlow`. The intersection operation is `*`.

**The Data Flow Between Statements** Call the first statement $T$ and the second $S$. Denote instances by $T(i,j)$ and $S(i,k,j)$. For any $k$, both $T(i,j)$ and $S(i,k,j)$ write to $C[i][j]$. For any $k$, $T(i,j)$ writes to $C[i][j]$ before $S(i,k,j)$ reads $C[i][j]$. Conversely, the write of $S(i,k,j)$ to $C[i][j]$ is not read by $T(i,j)$ because that simple statement is executed before $S(i,k,j)$ for any fixed $i,j$. Basically, we can see that if $S$ and $T$ are executed with different values of $i$ or $j$ they do not touch the same array references. Hence there can't be any data flow in that case. If they are run with the same values of $i,j$ then the write of $T(i,j)$ is read by $S(i,k,j)$ for all $k$. Hence, the data flow (with the right index bounds) is

$$D' = \{(i,j) \rightsquigarrow (i,k,j)\}.$$

If we stick to the tupling function's conventions, we have

$$D' = \{(i,0,j) \rightsquigarrow (i,1,k,j)\}.$$

We could verify this calculation with ISCC but that would be pedantic and overkill for such a simple code.

**Summary: The Overall Data Flow** So, in summary, for our dense matrix code $P$, we have our model $\sigma = (I(P), <_{\text{lex}}, \pi)$ where $I(P)$ is defined as below.

$$\begin{gathered} I(p) = X \cup Y \\ X = \{(i,0,j) : (0 \le i < n) \wedge (0 \le j \le i)\} \\ Y = \{(i,1,k,j) : (0 \le i < n) \wedge (0 \le k < m) \wedge (0 \le j \le i)\} \end{gathered}$$

Let $\sigma = (I(P), <_{\text{lex}}, \pi)$ be the standard model of $P$. Our data flow $D(\sigma) \subseteq (I(P) \times I(P))$ is

$$D(\sigma) = D \cup D'.$$

## 3.3 Existing Optimizations of `syr2k`

**Sequential Optimizations** The best known[1] sequential optimization we found is the one we achieve by applying the pluto optimizer without any command line arguments.

[1]To the best of our ability, we believe we've explored all the major optimizations and sufficiently played around with tile sizes of tiling optimizations.

```
for (i=0;i<=n-1;i++) {
  for (j=0;j<=i;j++) {
    C[i][j] *= beta;
    }
  }
for (i=0;i<=n-1;i++) {
  for (j=0;j<=i;j++) {
    for (k=0;k<=m-1;k++) {
      C[i][j] += A[j][k]*alpha*B[i][k] + \
                 B[j][k]*alpha*A[i][k];
      }
    }
  }
```

Figure 3.3: Pluto's best sequential optimization.

We remark that in this and other such codes we take the liberty of renaming loop bounds for readability.

The statements are split into two different loops. The first statement is basically untouched. For the second statement, the optimization swapped the order of the $j$ and $k$ loops. As a result, all the reads of $A$ and $B$ have indices that occur in the order of the loops. That is, for some read $R[x][y]$, we have the $x$ loop being outside the $y$ loop. This, as we've mentioned, improves locality.

**Parallel Optimizations** The best known parallel optimizations we found were pluto's tiled and parallel optimization and the one from following the algorithm in [LCL99].

Pluto's best parallel optimization (Figure 3.4) effectively tiles the sequential version and executes the tiles in parallel. The top loop blocks the reads to $C$ in squares with side length 32. The bottom loop blocks the iteration space by cubes along $i, j, k$ dimensions of size 32. In both loops we execute about $\frac{n}{32}$ tiles in parallel.

These (`pragma omp parallel`) pragmas indicate that the language should divide the iteration range of the loop below it by the number of cores and distribute the iterations of the below loop to those cores. After the for loop below that pragma is completed, the execution joins up again and continues sequentially.

Now, there are a few things this optimization does not consider. First, the locality could be improved by putting the first statement inside the $j$ loop of the second statement. We happen to read and write to $C[i][j]$ twice. Instead, we could (with the right tile sizes) read $C[i][j]$ once into cache, multiply it by $\beta$, do a series of increments and then write that back into main memory.

Second, the parallelization does not load balance. That is, as we divide the range of the $ii$ loop, which is $[0, \frac{n}{32}]$, among $k$ cores, the cores get a different amount of work. For example, the first core might get the values of $[0, 10]$ while some other core might get the values $[50, 60]$. The first core has to go through $\sum_{a=0}^{10} a$ iterations of the $jj$ loop. The last core has to go through $\sum_{a=50}^{60} a$ iterations, which is much larger.

```
ubp=floord(n-1,32);
#pragma omp parallel for private(lbv,ubv,ii,jj,i,j)
for (ii=0;ii<=ubp;ii++) {
  for (jj=0;jj<=ii;jj++) {
    for (i=32*ii;i<=min(n-1,32*ii+31);i++) {
      ubv=min(i,32*jj+31);
      for (j=32*jj;j<=ubv;j++) {
        C[i][j] *= beta;
        }
      }
    }
  }

#pragma omp parallel for private(lbv,ubv,ii,jj,kk,i,j,k)
for (ii=0;ii<=ubp;ii++) {
  for (jj=0;jj<=ii;jj++) {
    for (kk=0;kk<=floord(m-1,32);kk++) {
      for (i=32*ii;i<=min(n-1,32*ii+31);i++) {
        for (j=32*jj;j<=min(i,32*jj+31);j++) {
          for (k=32*kk;k<=min(m-1,32*kk+31);k++) {
            C[i][j] += A[j][k]*alpha*B[i][k] + \
                       B[j][k]*alpha*A[i][k];
            }
          }
        }
      }
    }
  }
```

Figure 3.4: Pluto's best parallel optimization.

```
#pragma omp parallel for private(i,j,k)
for (i = 0; i < n; i++) {
#pragma omp parallel for private(j,k)
  for (j = 0; j <= i; j++) {
      C[i][j] *= beta;
      for (k = 0; k < m; k++){
          C[i][j] += A[j][k]*alpha*B[i][k] + \
                     B[j][k]*alpha*A[i][k];
      }
  }
}
```

Figure 3.5: The parallel optimization from hand-applying [LCL99].

The second optimization, by [LCL99], is depicted in Figure 3.5. Now, as we've mentioned before, there are multiple codes which represent the correct optimization that paper describes. Experimentally, we found this one to have the best results.[2]

Effectively, this optimization is like pluto's optimization but without tiling and with the loop merge we've discussed. This optimization flips the $k$ and $j$ loops from the original code and then, it merges the $j$ loop around the first statement with the $j$ loop around the second. Then, the $i$ and $j$ loops are made parallel.

This optimization lacks tiling and load balancing. Because it is not tiled, the code will have issues with temporal locality as the reads of $A[j][k]$ for large $m$ will kick things out of cache that we might eventually return to. For example, we can expect to read $B[i][0]$ from main memory multiple times. Consider executing the code given a fixed $i$. Let $j_0$ be the value of $j$. By the time the innermost $k$ loop finishes and we increment the $j$ loop, we have pulled in so many cache lines from the reads of $A[j][k], B[j][k]$ that we will overwrite the cache line pulled in from the initial read of $B[i][0]$.

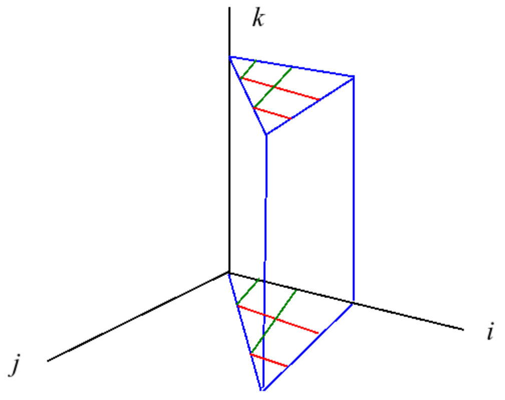

Figure 3.6: Visualizing the parallelization of the second statement.

The load balancing problem is the same as the one from before. Both the $i$ and $j$ loops are run in parallel. The iteration vectors of the second statement form a wedge in three space, as depicted in Figure 3.6. Imagine the red and green lines form vertical planes that slice the wedge into long pieces. If we just slice by the green lines, we have pieces which are relatively similar in size (as in pluto's version). If we slice by both the red and green lines, the disparity among the sizes of the pieces becomes much larger. The pieces that touch the hypotenuse of the triangle are much smaller than all the other pieces. The work is not evenly distributed among cores or threads.

[2]To see this and other failed attempts, see the link at the start of Chapter 4.

1. Swap the $j, k$ loops and merge the $j$ loops for locality.
2. Transpose $A$ and $B$ and when convenient use the transposed matrices for reads.
3. Tile the $i$ or $j$ loops.
4. Time tile the $k$ loop, leaving the $j$ loop inside.
5. Re-store the upper triangular matrix $C$.
6. Make the $i$ or $j$ loops parallel.

Figure 3.7: Our central optimization ideas.

## 3.4 Ideas About Optimization

Our ideas about optimization are summarized in Figure 3.7. All except the last are about optimizing locality in the $A$ and $B$ reads. Of course, we could combine some of these ideas and still maintain correctness. For instance, we could do (2) and then (6). Pluto basically does (1), (3) and part of (6).

The first idea is already done in the optimization we found from [LCL99]. It makes the reads of $A, B$ have indices that go in order of the loops.

The second idea is more original. We will see that this actually enables us to have a better sequential optimization than pluto. This would allow, for example the $A[j][k]$ read in the original code to be rewritten as $At[k][j]$ where $At$ is the transpose of $A$. This would make the indices of the read go in the order of the loops.

The third idea is quite straightforward. By tiling, we improve the temporal locality problems we had pointed out earlier. The reads of $A[j][k]$ and $A[i][k]$ no longer fight with each other for small enough tiles.

The fourth idea is directly applying the algorithm from [Won02]. The fifth idea is meant to avoid cache misses along the diagonal of $C$. The last idea is just a direct way to make the computation parallel without obstructing the data flow. Of course, this doesn't consider load balancing – making sure that the amount of iteration points we give to each core is about the same.

# Chapter 4

# Results

In this chapter, we present a narrow slice of our optimization results with a focus on the optimizations that were the strongest. These results along with several bad optimizations can be found at the github repository below.

`https://github.com/Divesh-Otwani/syr2k-results-thesis-2019`

## 4.1 Timing Data

### 4.1.1 Sequential Results

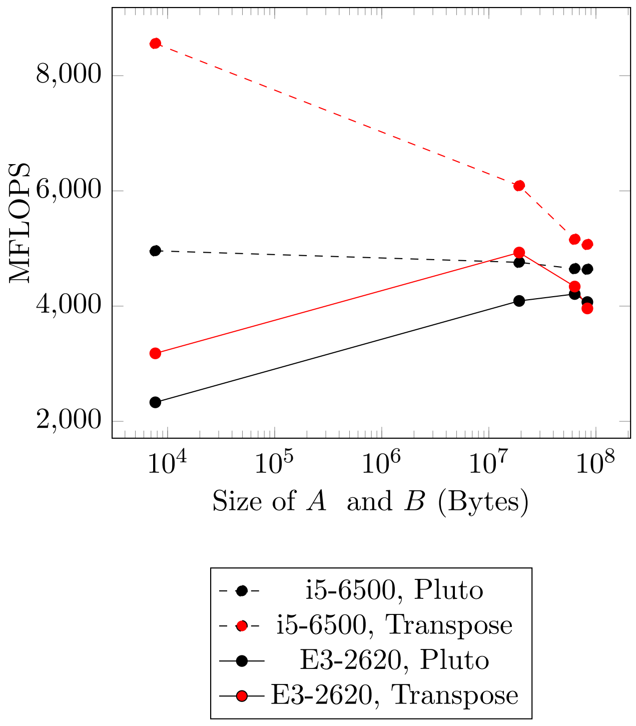

Figure 4.1: Sequential Timing Results

In our graphs, the $x$-axis is the amount of bytes used by the matrices $A$ and $B$. This is how we measure the problem size. The $y$-axis is MFLOPS: millions of floating point operations per second. This is the rate at which the computer executes our code. We run our tests on two machines. The first is a core i5-6500 chip. It has 4 L1 and L2 caches, one per core. The sizes of the L1,L2,L3 caches are 32,256 and 6144 kilobytes respectively. The second machine is a high performance machine. It has two Xeon E3-2620 chips, each with 8 cores. Each core has a L1 and L2 cache but L3 caches are shared among all 8 cores. The sizes of these caches are 32, 256 and 20480 kilobytes respectively. Our results on the core i5 machine are drawn with dotted lines while our results on our E3 machine are drawn with full lines. We select our sequential times by running our optimization three times and taking the *median.* We select our parallel results by taking the median of five runs of the given optimization. Of course, we tested all of our optimizations on some input sizes to see if they give the same result as the original `syr2k` code.

We've selected two optimizations to examine in our sequential runs.

- **Pluto:** Pluto's best sequential version from Figure 3.3
- **Transpose:** A version in which we create transposes $A$ and $B$ and use those matrices for reads where the order of indices was originally bad for locality. The time in our figure includes the time taken to make the transposed matrices.

The second optimization is depicted below.

```
// At and Bt are transposed
for (i = 0; i < n; i++) {
  for (j = 0; j <= i; j++) {
    C[i][j] *= beta;
  }
  for (k = 0; k < m; k++){
    for (j = 0; j <= i; j++){
      C[i][j] += At[k][j]*alpha*B[i][k] + \
                 Bt[k][j]*alpha*A[i][k];
    }
  }
}
```

Figure 4.2: Our best sequential optimization.

Recall that replacing the $A[j][k]$ and $B[j][k]$ reads by $At[k][j]$ and $Bt[k][j]$ improves locality. We can see that on both machines, for small problem sizes our transposed optimization surpasses pluto's best. However, for large problem sizes, the performance gap becomes negligible.

We remark that the optimization that time-tiles the $k$ loop from [Won02] was not useful.[1]

[1]It is possible that this is due to our errors alone.

### 4.1.2 Parallel Results

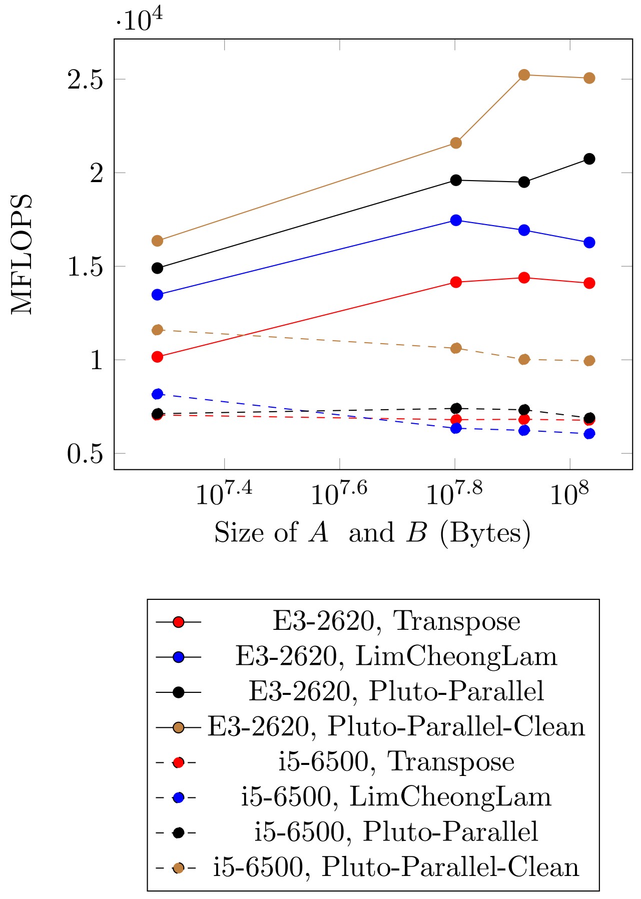

Figure 4.3: Parallel Timing Results

We consider four parallel optimizations:

- **Transpose:** This is our sequential transposed code except with the $i$ and $j$ loop made parallel. Note that the time taken to perform the transpose is included in our results. (A modification of Figure 4.2.)
- **LimCheongLam:** The optimization from [LCL99] with both the $i$ and $j$ loop made parallel. (Figure 3.5.)
- **Pluto-Parallel:** Pluto's best parallel version. This is found by running pluto with the `--tile` and `--parallel` command line arguments. (Figure 3.4.)
- **Pluto-Parallel-Clean:** Our hand-modified optimization of pluto's best optimization. We fuse the loops and un-tile the $k$ loop. (Figure 4.4.)

We've seen all these optimizations except for the last. The cleaned up version of pluto's optimization is shown in Figure 4.4. The first statement

```
lbp=0;
ubp=floord(n-1,32);
#pragma omp parallel for private(lbv,ubv,ii,jj,i,j,k)
for (ii=lbp;ii<=ubp;ii++) {
   for (jj=0;jj<=ii;jj++) {
      for (i=32*ii;i<=min(n-1,32*ii+31);i++) {
         for (j=32*jj;j<=min(i,32*jj+31);j++) {
            C[i][j] *= beta;
            for (k=0;k < m;k++) {
               C[i][j] += A[j][k]*alpha*B[i][k] + \
                          B[j][k]*alpha*A[i][k];
            }
         }
      }
   }
}
```

Figure 4.4: Our optimization of pluto's best parallel optimization.

is pushed inside the $j$ loop, which improves locality. We un-tile the $k$ loop because it appears as though that doesn't add anything. From a visual inspection, we see that traversing all the $k$ values in the innermost loop has great spacial locality and the tiling of the $i$ and $j$ loops gives us temporal locality for the $A[i][k]$ and $B[i][k]$ reads.

We see in our graph in Figure 4.3 shows that our cleaned up version of pluto's best wins in both the i5 and E3 machines. Our transposed code for large problem sizes is last on our E3 machine. It fares about as well as pluto's optimization on our i5 machine, however.

## 4.2 Analysis of Optimizations

Here, we take a closer look at our parallel optimizations (and look past the sequential results which are not as exciting).

| **Optimization** | **L2 Misses** | **L3 Misses** |
|---|---|---|
| Transpose | 2,616,343,209 | 2,320,371,075 |
| LimCheongLam | 2,391,525,165 | 2,041,839,135 |
| PlutoParallel | 149,416,070 | 70,995,708 |
| PlutoParallelClean | 2,352,714,539 | 67,191,951 |

Table 4.1: Cache misses of our parallel codes on the E3 machine for size $(n = 2700, m = 2500)$.

In Table 4.1, we see the L2 and L3 cache misses we have when we run our parallel optimizations on our E3 machine with $n = 2700$ and $m = 2500$ as our matrix sizes. Recall that L2 cache is private to each core while L3 cache is shared among 8 cores. We found these measurements with the PAPI profiling tool that is set up with the PolyBench codes [PY15].

First, we can see that the L3 cache misses decrease as we move down the optimizations, which matches our performance graph. The difference between the transposed code and the "LimCheongLam" is fairly small. We can see that for large problem sizes, the these two codes have similar numbers of cache misses. Whereas, the gap between these codes and the pluto-based codes is drastic and this also matches our performance graph. In our last set of data points, these codes are much faster than the first two. In other words, locality appears to be the key factor between those two sets of codes.

Now, one interesting matter is that Pluto's L2 cache misses are really low and our "PlutoParallelClean" version has a lot of L2 cache misses (and it is faster). However, we see that our cleaned up version has about 4 million less L3 cache misses. So, our cleaned up code has better communication among the 8 cores than pluto's parallel version. This is one possible explanation for the better performance. We also found (via the PAPI tool) that our cleaned up code used about 83 trillion clock cycles while pluto's used about 81. This suggests a bit more parallelism is at play in our cleaned up version.

## 4.3 Chapel Abstractions

It turns out that we are able to express our codes using Chapel iterators and records cleanly. However, performance drops drastically for all codes.

In Figure 4.5 we have the modified chapel kernel that uses iterators and records to abstract optimization. To change which optimization we are using all we need to do is compile with different compile time parameters for `iterChoice` and `recChoice`. This will determine the type of the record we use. The choice of `iterChoice` will be input to the `chooseIter` function which chooses an iterator. See Figure 4.6 for the general setup. We define these configuration parameters and provide default values. The `iterChoice` is an enum representing the possible iterators we have. The `recChoice` is an actual data type that is defined after we define two records.

The choices of `recChoice` are two records, one of which transposes the arrays $A, B$ while the other does not. Ultimately, we see that Chapel is powerful enough to neatly abstract away our optimizations to our heart's content.

In Figure 4.7 we have our iterator for pluto's best parallel code. With the `ptilepar` given as a configuration parameter, we are able to get our code to use this iterator. Note that the `forall` loop is essentially the same as the `(pragma omp ...)` stuff we've seen before. As we can see, we can make any iterator we want (for all of our optimizations) and use them just as we used this one. We just need to add a value to the `iterChoice` enum and update the `chooseIter` function.

```
proc kernel(
      n : int, m : int
    , beta : real
    , alpha : real
    , C : [1..N,1..N] real
    , A : [1..N,1..M] real
    , B : [1..N,1..M] real) : real {
  var timer2: Timer;
  timer2.start();

  var Adelta: recChoice = new recChoice(n, m, A);
  var Bdelta: recChoice = new recChoice(n, m, B);

  for (i,k,j,stmt) in chooseIter(n,m,iterChoice) do {
    select stmt {
      when Stmt.betamult do {
        C[i,k] *= beta;
      }
      when Stmt.updateC do {
        C[i,j] += Adelta[j,k]*alpha*Bdelta[i,k] + \
                  Bdelta[j,k]*alpha*Adelta[i,k];
      }
    }
  }
  timer2.stop();
  return (timer2.elapsed():real);
}
```

Figure 4.5: The abstracted `syr2k` kernel.

```
enum IterChoice { original, ptile, ptilepar, pluto };
config param iterChoice = IterChoice.original;

// Definition of OriginalArr Record ...
// Definition of TransposedArr Record ...

config type recChoice = OriginalArr;
```

Figure 4.6: Compile time parameters in Chapel.

```
iter plutoTilePar(param tag: iterKind, n: int, m:int):
(int, int, int, Stmt)
where tag == iterKind.standalone {
  const t2bound: int = divfloor(n-1,32);
  const t4bound2: int = divfloor(m-1,32): int;
  forall t2 in {0..t2bound} do {
    for t3 in {0..t2} do {
      var t4bound: int = min(n-1, 32*t2 + 31);
      for t4 in {(32*t2)..t4bound} do {
        var lbv: int = 32*t3;
        var ubv: int=min(t4, 32*t3 + 31);
        for t5 in {lbv..ubv} do {
          yield (t4+1, t5+1, 0, Stmt.betamult);
        }
      }
    }

    var t5bound: int = min(n-1, 32*t2+31);
    for t3 in {0..t2} do {
      for t4 in {0..t4bound2} do {
        var t7bound: int = min(m-1,32*t4+31);
        for t5 in {(32*t2..t5bound)} do {
          var t6bound: int = min(t5, 32*t3 + 31);
          for t6 in {(32*t3)..t6bound} do {
            for t7 in {(32*t4)..t7bound} do {
              yield (t5+1,t7+1,t6+1, Stmt.updateC);
            }
          }
        }
      }
    }
  }
}
```

Figure 4.7: Parallel iterator for pluto's best parallel optimization.

# Chapter 5

# Correctness

**Formally Checking Correctness** We reason about the correctness of our codes by checking the data flow is preserved. Since `syr2k` is so simple, we can do this by hand. Recall that our data flow is as follows.

- $I(p) = X \cup Y$
- $X = \{(i, 0, j) : (0 \leq i < n) \wedge (0 \leq j \leq i)\}$
- $Y = \{(i, 1, k, j) : (0 \leq i < n) \wedge (0 \leq k < m) \wedge (0 \leq j \leq i)\}$
- $D = \{(i, 1, k, j) \rightsquigarrow (i, 1, k', j) : k' < k\}$
- $D' = \{(i, 0, j) \rightsquigarrow (i, 1, k, j)\}$
- $D(\sigma) = D \cup D'$

We want to check that after performing the transformation $f$ on our tuples, we still have the same data flow. That is, if $\sigma = (I(P), <_{\text{lex}}, \pi)$ is our original model and $\sigma_f = (f[I(P)], <_{\text{lex}}, \pi \circ f^{-1})$ is our optimized model, then we preserve data flow if the following holds for all $\vec{i}, \vec{j} \in I(P)$.

$$\vec{i} \rightsquigarrow \vec{j} \in D(\sigma) \iff f(\vec{i}) \rightsquigarrow f(\vec{j}) \in D(\sigma_f) \tag{5.0.1}$$

We need to prove two things: the forward ($\Rightarrow$) and converse ($\Leftarrow$) directions of the equivalence 5.0.1. First, we look at an arbitrary pair of iteration vectors in the original code with data flow between them: $\vec{i} \rightsquigarrow \vec{j}$. We will need to take cases on whether $(\vec{i}, \vec{j})$ is in $D$ or $D'$. We then show that such a pair must have the same data flow after the transformation $f$. In the converse direction, we look at an arbitrary pair of iteration vectors in $\sigma_f$ that have data flow from one to the other. We apply the inverse of $f$ to find the original iteration vectors and check that the data flow is preserved again. That is, we will need to show the converted pair is in $D$ or $D'$.

**Testing** Writing formal proofs each time we try out an optimization is quite tedious. Of course, we cannot rely on our intuition to think that our optimizations are correct. So, instead we test our optimizations against the original code. Using a standard matrix-printing function, we print out the matrix C for various test sizes from the original code. Then, with some optimization, we print out the matrix it produces and check if it matches the results of our original code. We do this on both medium and large sizes. It might be the case that some parallel optimization looks correct only because it worked on some small sizes where the number of threads was so low that no data-races messed up the computation.

# Chapter 6

# Conclusion

We have a few conclusions that we can draw from our results and our work. We return to the questions of (1) can we get fast, correct and clean versions of `syr2k` (which are as fast or faster than known optimizations) and (2) what do we learn about the optimization techniques we've tried?

**Chapel Loses Performance** Our experiments with abstracting our codes in Chapel dropped the performance of all codes, while maintaining the relative order of performance. This suggests that something about our translation in Chapel drastically loses performance. Here are a few possible explanations and non-explanations:

- We are using Chapel version 1.19.0. This version has changed some of the ways iterators and arrays work.
- Chapel's iterators are not the problem. When we compare versions of the same optimization that use Chapel iterators and do not use Chapel iterators, we get the same results.
- Chapel's records are not the problem. Again, codes without this abstraction run just as fast.
- Our compilation setup uses Docker and the `--static` flag. These shouldn't really impact the performance of our code, but there might be a strange way in which they do.
- Chapel's `forall` loops seem suspicious. It might be the case that these do not work as well as the openMP parallelization that we do with our `(pragma omp parallel)` stuff.

All in all, we were able to get one significant performance improvement by a few modifications to pluto's best parallel results. However, we are unable to get these optimizations to be as readable as we'd like without losing performance.

**Parallel Optimizations: Tiling over Transposing** We learn one major point about optimization techniques: *tiling and loop-swaping beats transposing for large inputs.* Our central optimization involved taking the transpose of $A$ and $B$ (using blocking to maintain speed). It did not stand a chance against swapping the $k, j$ loops and merging the two $j$ loops into one. That achieved the basic ideas of spacial locality that the transpose was after in the first place. Then, the tiling of the $i, j$ loops helped achieve the temporal locality and split the work for parallel execution. All of this is far more effective than transposing.

**Conclusions about PLuTo and [LCL99]** We did see a few more things, however, about the key optimization tools and algorithms we worked with.

Pluto's best parallel optimization split the two statements into different loops. We are not sure why this is the case, but it hurt locality a bit. However, the more important observation is that the tiling of the $k$ loop significantly hurt performance. It creates tiles that when executed (within one sequential thread) walk down the columns (in chunks) of $A$ or $B$. This effectively introduces the same locality problems as before. We surmise, that it also creates more cache fights for L3 cache.

Lastly, the algorithm from [LCL99] simply does not consider tiling and as with PluTo does not load balance. Overall, both tools have their benefits and shortcomings when it comes to `syr2k` and have large areas for improvement: not tiling or over-tiling and load balancing.

# Bibliography

[BHRS08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6):101–113, June 2008.

[BOH$^+$15] 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. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 197–206, New York, NY, USA, 2015. ACM.

[Bon] Uday Kumar Reddy Bondhugula. Pluto - an automatic parallelizer and locality optimizer for affine loop nests. From `http://pluto-compiler.sourceforge.net/#performance` on 09.30.2018.

[Col07] Jean-Francois Collard. *Reasoning about program transformations: imperative programming and flow of data*. Springer Science & Business Media, 2007.

[cor19] Intel core i5-7500 specifications, 2019 (accessed April 29th 2019). `http://www.cpu-world.com/CPUs/Core_i5/Intel-Corei5i5-7500.html`.

[FL11] Paul Feautrier and Christian Lengauer. *Polyhedron Model*, pages 1581–1592. Springer US, Boston, MA, 2011.

[isc19] Polyhedral.info: Iscc online demonstrator, 2019 (accessed April 29th, 2019). http://polyhedral.info/2014/01/21/ISCC-demo-online.html.

[LCL99] Amy W Lim, Gerald I Cheong, and Monica S Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proc. 13th ACM SIGARCH International Conference on Supercomputing*. Citeseer, 1999.

[PH98] D Patterson and J Hennessey. Computer organization and design. pp501-02, 1998.

[PY15] LN Pouchet and T Yuki. Polybench/c 4.1, 2015.

[Won02] David Wonnacott. Achieving scalable locality with time skewing. *International Journal of Parallel Programming*, 30(3):181–221, 2002.

[Won10] David G Wonnacott. A retrospective of the omega project. *rap. tech. HC-CS-TR-2010-01, Haverford College Computer Science*, 2010.