THESIS

OPTIMIZING BIOINFORMATICS ALGORITHMS WITH AUTOMATIC INDEX SET SPLITTING

Submitted by

Campbell Bradley

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2026

Master's Committee:

Advisor: Sanjay Rajopadhye
Co-Advisor: Vinayak Prabhu

Louis-Noël Pouchet
Michael Kirby

ABSTRACT

OPTIMIZING BIOINFORMATICS ALGORITHMS WITH AUTOMATIC INDEX SET SPLITTING

A number of important and computationally-expensive bioinformatics algorithms belong to the class of non-serial polyadic dynamic programs (NPDPs). NPDPs tend to admit clever optimizations in the form of index-set splitting: a technique which involves partitioning a nested loop before applying separate transformations to each partition. Unfortunately, these partitions require vast manual effort to find by hand; no existing compiler is able to find index-set splits automatically. We present a linear programming technique which is able to automatically perform index-set splitting, allowing relevant programs to be tiled, which yields improvements in data locality and parallelism. We implement this technique in the AlphaZ system, and demonstrate massive performance gains.

ACKNOWLEDGEMENTS

With much gratitude to Dr. Craig Partridge for funding this research, as well as Drs. Sanjay Rajopadhye and Vinayak Prabhu for their unfaltering support.

TABLE OF CONTENTS

Chapter 1

Introduction

Reduction—the accumulation of an arbitrary number of values under an associative operator—is a fundamental and ever-present operation in many domains of computer science. Reductions underlie many essential algorithms in the domains of linear algebra, dynamic programming, image and signal processing, physical simulations, machine learning, and bioinformatics, just to name a few.

Programs with reductions are generally amenable to high-level optimizations (such as parallelization, data locality optimization, and reuse simplification) which reduce the amount of time and energy they consume. One very common strategy is that of tiling, which blocks the computation into atomic chunks, minimizing the temporal distance between data producers and consumers, and exposing a degree of coarse-grain parallelism. Some, but not all of these optimizations can be found and exploited automatically by current compilers. A large amount of human effort is still needed to find the most efficient implementations of these kinds of programs in many cases [1, 2, 3, 4, 5].

One particularly tricky class of program to reason about is that of the non-serial polyadic dynamic program (NPDP). NPDPs make use of reductions which have multiple self-recurrent, affine dependences, and are the most complax class of dynamic program, as per Li and Wah's taxonomy [6]. Many important bioinformatic algorithms fall into this class [2, 3, 5, 7]. As will be elaborated upon in later sections, NPDPs most often require an index-set splitting—a transformation which splits a loop nest into a semantically equivalent partition of nests, each of which are transformed independently—in order to be fully tiled. Finding these splits by hand is a very non-trivial task, especially when many useful NPDPs feature loop nests of degree six or more.

Enter the polyhedral model[1]: a formalism for representing programs geometrically, optimizing them via affine mappings, and generating code for target hardware. The class of programs representable in the polyhedral model is limited to that of affine static control parts: loop nests with no branching logic whose indices are affine combinations of other indices and parameters. Within this limited scope, linear algebra can be used to make powerful assertions that would not be possible in a broader class of programs. Namely, the Affine Form of Farkas' Lemma allows for linear inequalities defined over polyhedra to be converted into ordinary linear constraints, which can then be evaluated efficiently using linear programs. Schedule-finders such as Feautrier's algorithm and PLuTo leverage this fact [8, 9, 10], allowing for polyhedral compilation as it exists today.

Due to their complex self-recursive structure, NPDPs prove difficult for polyhedral compilers to optimize. No existing compiler is able to fully tile NPDPs (in general), causing generated code to ultimately become compute-bound for large program sizes. This is the problem that we aim to solve in this thesis.

1.1 Original Contributions

Our original contributions (which are more precisely defined and explained in later sections) are as follows:

- Proof that a one-dimensional affine schedule is necessary for a polyhedral program to admit a full linear tiling.
- A novel algorithm that is able to perform index set splittings, and find schedules of one-dimensional span for NPDPs whenever possible, which can then be tiled using the aid of existing algorithms such as PLuTo.
- An implementation of our algorithm in the *AlphaZ* polyhedral suite, and an experiment which demonstrates a speedup of 2.3-8.6$\times$ on benchmark programs.

[1]Referred to by some authors as the polytope model, as the titular 'polyhedra' can take on any integer number of dimensions.

1.2 Thesis Outline

The remainder of this thesis is outlined as follows. Chapter 2 provides some motivating examples of NPDPs, and Chapter 3 establishes the background knowledge necessary for the comprehension of this thesis. Chapter 4 establishes some properties of tiling that are necessary for the proof of our capstone theorem. Chapter 5 outlines the challenges associated with scheduling NPDPs, and Chapter 6 describes a method by which to schedule NPDPs automatically, and 7 illustrates our method in a step-by-step manner on example programs. Chapter 8 describes our implementation of the aforementioned method, and the speedup gained when using it to schedule some benchmark programs. Chapter 9 is our conclusion, and Chapter 10 discusses related work, including the foundational papers of the polyhedral model, recent papers relevant to our work, and other recommended reading. Proofs for all of our theorems are located in the appendix.

Chapter 2

Motivations

We provide some examples of programs that our approach hopes to optimize, in order of increasing complexity.

2.1 Matrix Chain Multiplication

Often, computers are tasked with multiplying a sequence of matrices, each of which may differ in their height and width. In these scenarios, it is vitally important for the matrix multiplications to be associated in such a way that the total number of atomic operations (typically, floating point operations) is minimized. This is known as the Matrix Chain Multiplication (MCM) problem.

Take, for example, the case where one wants to multiply a vector by two matrices. Say, for example, we wish to compute ABC, where A and B are matrices of size $N \times N$, and C is of size $N \times 1$. If the multiplications are performed left-to-right, a total of $N^3 + N^2$ total atomic multiplications are required. If, on the other hand, BC is computed first, and then multiplied by A, that number shrinks to $2N^2$, decreasing the total amount of work by an entire order of magnitude.

The optimal order in which to multiply matrices may be found by the following dynamic program. $\mathrm{Y}[i, j]$ represents the minimum cost of multiplying the ith through the jth matrix; $\mathrm{W}[i]$ represents the number of columns in the ith matrix, and rows in the $i + 1$th matrix:

$$\mathrm{Y}[i, j] = \min_{i \leq k < j} \mathrm{Y}[i, k] + \mathrm{Y}[k + 1, j] + \mathrm{W}[i - 1]\,\mathrm{W}[j]\,\mathrm{W}[k]$$

The optimal matrix multiplication order can then be found by backtracing Y.

The MCM is one of the simplest examples of an NPDP, and an efficient schedule under an index set split has been known since 1979 [1]. However, not all NPDPs are as simple, nor as easy to optimize.

2.2 RNAFold

ViennaRNA is a bioinformatics package which is widely used to predict the secondary structures of single RNA molecules [5]. ViennaRNA itself consists of tens of thousands of lines of code, and is by no means condensible into a single dynamic program. However, one subcomponent of ViennaRNA, RNAFold, is responsible for the majority of its runtime for long sequences. After hoisting some code past function boundaries, RNAFold can be described using the following recurrence equations [11]:

$$\begin{aligned}
\mathrm{FM}[i,j] &= f_{mb}(\mathrm{DML}[i,j], \mathrm{C}[i,j]) \\
\mathrm{C}[i,j] &= f_{dp}(\mathrm{DML}[i+1,j-1], \mathrm{DML}[i+2,j-1], \\
&\qquad \mathrm{DML}[i+1,j-2], \mathrm{DML}[i+2,j-2]) \\
\mathrm{DML}[i,j] &= \min_{i<k<j} \mathrm{FM}[i,k] + \mathrm{FM}[k+1,j]
\end{aligned}$$

Where f_{mb} and f_{dp} are functions which execute in constant time.

Given that the reduction in this program has a near-identical structure to the one in the MCM, it is likely that a similar index set split would work as an optimization.

2.3 BPMax

Predicting the secondary structure of an RNA sequence is a complex and computationally challenging task. Predicting the secondary structure of two interacting RNA sequences, even more so. One program, BPMax, seeks to simplify the problem for the sake of computational efficiency, at the cost of some accuracy. It is defined by the following recurrence equations [3]:

$$\mathrm{S}[i,j] = \begin{cases} 0 & j - i < 4 \\ \max\big(\mathrm{S}[i+1, j-1] + \mathrm{score}(i,j), & \\ \quad \max_{i \le k < j} \mathrm{S}[i,k] + \mathrm{S}[k+1,j]\big) & \text{otherwise} \end{cases}$$

$$\mathrm{H}[i_1,j_1,i_2,j_2] = \max_{i_1 \le k_1 < j_1 \wedge i_2 \le k_2 < j_2} \mathrm{F}[i_1,k_1,i_2,k_2] + \mathrm{F}[k_1+1,j_1,k_2+1,j_2]$$

$$\mathrm{F}[i_1,j_1,i_2,j_2] = \begin{cases} -\infty & i_1 > j_1 \wedge i_2 > j_2 \\ \mathrm{S}[i_1,j_1] & i_1 \le j_1 \wedge i_2 > j_2 \\ \mathrm{S}[i_2,j_2] & i_1 > j_1 \wedge i_2 \le j_2 \\ \mathrm{iscore}(i_1,i_2) & i_1 = j_1 \wedge i_2 = j_2 \\ \max\big(\mathrm{F}[i_1+1, j_1-1, i_2, j_2] + \mathrm{score}(i_1,j_1), & \\ \quad \mathrm{F}[i_1,j_1,i_2+1,j_2-1] + \mathrm{score}(i_2,j_2), & \\ \quad \mathrm{H}[i_1,i_2,j_1,j_2]\big) & \text{otherwise} \end{cases}$$

where iscore and score are both functions that compute pairing scores for the base pairs at the appropriate indices in the given RNA sequences.

With three variables, and multiple reductions of up to degree four each, BPMax is a very complex NPDP, and very difficult to analyze by hand; finding a good index set split would require reasoning about splitting six-dimensional polyhedra of varying sizes and shapes, which is not an easy task for a human. Thus, it is no surprise that such schedule has not yet been found.

Chapter 3

Background and Notation

In this thesis, blackboard bold letters are used to represent numerical sets as per typical convention. $\mathbb{Z}$ represents set of integers, and $\mathbb{Z}_+$ the set of natural numbers (including 0).

A numerical set raised to a power represents the set of vectors or matrices of the appropriate size. For example, $\mathbb{Z}^d$ represents the set of d-dimensional integer vectors, and $\mathbb{Z}^{m\times n}$ the set of real-valued $m \times n$ matrices.

3.1 Linear Algebra

Our thesis makes heavy use of the conventions of linear algebra, including concepts such as vector spaces and matrix multiplication. In addition, we make use of the concepts that we outline below [12]:

- **Affine Mapping**: An affine integer map $f : \mathbb{Z}^n \to \mathbb{Z}^m$ is a combination of a linear mapping and a translation. We define it as follows, for some matrix $A \in \mathbb{Z}^{m\times n}$ and a constant vector $\mathbf{b} \in \mathbb{Z}^m$.

$$f(\mathbf{x}) = A\,\mathbf{x} + \mathbf{b} \tag{3.1}$$

  The composition of two affine maps is also an affine map.

- **Affine Inequality**: an inequality dependent on the vector $\mathbf{x}$, wherein the dot product of $\mathbf{x}$ and a constant vector $\mathbf{a}$ is added to a scalar bias and compared to 0.

$$\mathbf{a}^T\,\mathbf{x} + b \geq 0 \tag{3.2}$$

  A *set* of affine inequalities may be defined as a component-wise inequality of an affine map with the zero vector:

$$A\,\mathbf{x} + \mathbf{b} \geq \mathbf{0} \tag{3.3}$$

- **Linear Program**: A linear program is composed of a set of affine inequalities $A\,\mathbf{x} + \mathbf{b} \geq \mathbf{0}$ and a cost vector $\mathbf{c}$. The solution to the linear program is a point $\mathbf{x}$ that maximizes $\mathbf{c}^T\,\mathbf{x}$, while satisfying the given affine inequalities. **Integer linear programs** additionally require that $\mathbf{x}$ be an integral vector, and are NP-Hard to solve. Despite this, integer linear program solvers are able to efficiently solve most linear programs, in practice.

- **Lexicographic order**: A comparison between two vectors. A vector $\mathbf{x}$ is said to be **lexicographically greater** than ($\succ$) a vector $\mathbf{y}$ if they differ in at least one dimension, and $\mathbf{x}_i > \mathbf{y}_i$ where i is the first such dimension in which they differ. Formally,

$$\mathbf{x} \succ \mathbf{y} \equiv \exists i \in \mathbb{Z}\big(\,\mathbf{x}_i > \mathbf{y}_i \wedge \forall j \in \mathbb{Z}(j < i \implies \mathbf{x}_i = \mathbf{y}_i)\big) \tag{3.4}$$

  Likewise, a vector $\mathbf{x}$ is **lexicographically greater than or equal to** ($\succcurlyeq$) $\mathbf{y}$ if and only if $\mathbf{x} \succ \mathbf{y}$ or $\mathbf{x} = \mathbf{y}$.

  Whenever we perform a comparison $\mathbf{x} \succ a$ where $\mathbf{x}$ is a vector but a is a scalar, we define the result as the same as $\mathbf{x} \succ \mathbf{y}$, where $\mathbf{y}$ is a vector whose last component is a, and all other components are 0.

Additionally, we use the following terminology in our thesis:

- **Convex Polyhedron**: In the polyhedral model, a convex integer polyhedron is an integral set defined by a set of affine inequalities. A point x is in the polyhedron if and only if it satisfies every given inequality. Note that convex polyhedra in this sense do not need to be three-dimensional, nor do they need to be bounded or finite.

- **Polyhedron**: in the polyhedral model, a polyhedron is a finite union of convex polyhedra.

- **Slice**: a subset of a polyhedron, found by intersecting it with a set of affine equalities. Because an affine equality is the intersection of two affine inequalities, a slice of a polyhedron is itself also a polyhedron.

3.2 Polyhedral Model

The polyhedral model is an analytical system used to transform and optimize programs. In the polyhedral model, nested for-loops are represented as polyhedra of dimension up to the numerical depth of the nest. These polyhedra can be transformed in many ways, improving the program's memory usage and runtime while preserving the original semantics of the program. Many such optimizing transformations can be found automatically, greatly easing the intellectual burden of algorithms researchers.

In much of the existing literature, **static control parts** (SCoPs) are used as the entry point to the polyhedral model. SCoPs are fragments of imperative programs (such as C code) where the only legal control flow statements are conditionals and for loops, all of which must be affine mappings or affine inequalities of previous loop indices. Furthermore, many program fragments which are not apparently SCoPs can be transformed into them, by inlining functions and converting while loops into affine for loops, for example [13].

While SCoPs are useful constructs for the purpose of source-to-source code compilation, we find they are cumbersome when it comes to more general polyhedral analysis. The polyhedral model provides a higher level of abstraction than that of imperative code, and extra effort is required both when writing a program in imperative form, and when performing dependence analysis when extracting the polyhedral information from imperative code.

For this reason, we find it convenient to enter the polyhedral model using **parametric systems of affine recurrence equations** (SAREs), which directly express all the relevant dependence information of a program, while remaining agnostic to that which is irrelevant to polyhedral analysis —such as the order in which statements are executed and the memory addresses which are used to store the data—which are left to the purview of the compiler.

An SARE (without reductions) consists of a finite number of recurrence equations, each corresponding to one variable subdomain $\mathcal{D}$:

$$\forall \mathbf{x} \in \mathcal{D} \subseteq \mathcal{D}_{\mathrm{U}} \ : \ \mathrm{U}[\mathbf{x}] = f(\mathrm{V}_1[r_1(\mathbf{x})], ..., \mathrm{V}_k[r_k(\mathbf{x})]) \tag{3.5}$$

where

- U is a **variable**. Formally, $\mathrm{U} : \mathcal{D}_{\mathrm{U}} \to A$ is a function mapping $\mathcal{D}_{\mathrm{U}}$ to **values** in the set A (which is typically, but not necessarily, the real numbers).
- $\mathcal{D}_{\mathrm{U}}$ is the **domain** of U. Formally, $\mathcal{D}_{\mathrm{U}} \subset \mathbb{Z}^{d_U}$ is a polyhedron of dimension d_U.
- $\mathcal{D}$ is a **subdomain** of $\mathcal{D}_{\mathrm{U}}$. The union of all such subdomains is equal to the domain, with no overlap, ensuring that each point $x \in \mathcal{D}_{\mathrm{U}}$ is mapped to only one value by U.
- Each V_i is also a variable, which may be the same variable as U (which is to say, a variable may depend on itself).
- $\mathbf{x}$ is a **point** in the domain of U. Each point $\mathbf{x} \in \mathcal{D}_{\mathrm{U}}$ is a $\mathrm{d_U}$-dimensional vector.
- f is some atomic function that takes a fixed number of inputs, and takes constant time to evaluate. Functions with an unbounded number of inputs must be represented as reductions, as is explained below.
- Each $r_i \in R_{\mathrm{S}}$ is a **dependence**, where R_{S} is the set of dependences from U. Formally, $r_i : \mathcal{D}_{\mathrm{U}} \to \mathcal{D}_{\mathrm{V}_i}$ is an affine map that maps from the domain of U to the domain of V_i. It is referred to as a dependence because it communicates that the computation of the value $\mathrm{U}[\mathbf{x}]$ depends on the value $\mathrm{V}_i[r_i(\mathbf{x})]$.

We also consider SAREs which contain reductions, which are of the following form:

$$\forall\, \mathbf{x} \in \mathcal{D} \;:\; \mathrm{U}[\mathbf{x}] = \bigoplus_{\{\mathbf{x}' \in \mathcal{D}_{\mathrm{S}} \;\mid\; w(\mathbf{x}') = \mathbf{x}\}} f(\mathrm{V}_1[r_1(\mathbf{x})], ..., \mathrm{V}_k[r_k(\mathbf{x})]) \tag{3.6}$$

where

- $\mathcal{D}_{\mathrm{S}}$ is the **reduction domain**—the set of points over which the reduction expression is evaluated.
- $\mathbf{x}' \in D_{\mathrm{S}}$ is a point in the reduction domain.

- $\bigoplus$ is an operator of unbounded arity that accumulates values according to some associative, commutative[2] operator $\oplus$. For example, the sum reduction operator $\sum$ accumulates values using the addition operator $+$.

- w is a **write function**. Formally, $w : \mathcal{D}_{\mathrm{S}} \to \mathcal{D}$ is an affine map that maps from the reduction domain to the subdomain of the recurrence equation.

- $f(...)$ is known as the **reduction kernel**.

Additionally, we define the **slices** of a reduction to be points in the reduction domain which are written to the same point in the destination variable. Formally, for some $\mathbf{x}$ in $\mathcal{D}_{\mathrm{U}}$,

$$\mathrm{Slice}_{\mathbf{x}} = \{\mathbf{x}' \in \mathcal{D}_{\mathrm{S}} \mid w(\mathbf{x}') = \mathbf{x}\} \tag{3.7}$$

For an SARE to be parametric, a few additional restrictions are necessary. Firstly, we consider the last several dimensions of $\mathbf{x}$ to represent the **parameters** of the program. Namely, $\mathbf{x} = \binom{\mathbf{z}}{\mathbf{p}}$, where $\mathbf{p}$ are the program parameters, and $\mathbf{z}$ are the remaining indices.

Secondly, each **program instance** found by setting $\mathbf{p}$ to a constant value must be bounded in every variable domain. This ensures that there are a finite number of values to compute per program instance.

Lastly, dependence functions and write functions must preserve the parameters of the program instance. That is to say, for each dependence/write function $f\binom{\mathbf{z}}{\mathbf{p}}$, there exists some vector $\mathbf{z}'$ such that $f\binom{\mathbf{z}}{\mathbf{p}} = \binom{\mathbf{z}'}{\mathbf{p}}$. This ensures that program instances are isolated, and do not depend on values from other program instances.

In general, the computability of an SARE—whether or not every parametric instance of the SARE may be converted into an imperative program with finite runtime—is undecidable [14].

[2]For the purposes of this thesis. Non-commutative reduction operators are possible, but do not admit polyhedral transformations as readily as commutative ones.

However, in realistic cases, it is usually sufficient to determine whether or not an SARE can be scheduled using affine maps, as described in the next section[3].

3.3 Affine Scheduling

SAREs do not describe the order in which the program's values should be computed. Therefore, in order to convert the program into an imperative form, it is necessary to find a **schedule**: a set of functions which map the points of a program's variables to a space with an ordering. For the purposes of this thesis, we only consider **multidimensional affine schedules**—which are **affine**, resolving the aforementioned decidability issue, and allowing the set of feasible schedules to be searched using an integer linear program solver—and **multidimensional** (meaning they map points to a lexicographically-ordered domain of multiple dimensions) as some SAREs admit multidimensional schedules, but not linear ones [9]. For the remainder of this thesis, we use 'schedule' to refer to a multidimensional affine schedule unless otherwise stated.

Such a schedule consists of affine **timestamp functions** $t_{\mathrm{U}} : \mathcal{D}_{\mathrm{U}} \to \mathcal{D}_t$, where $\mathcal{D}_t \subseteq \mathbb{Z}_+^{d_t}$, is a polyhedron of dimension d_t. Timestamp functions take the following form, for $\boldsymbol{\lambda}_{\mathrm{U}} \in \mathbb{Z}^{d_t \times d_{\mathrm{U}}}$, a $d_t \times d_{\mathrm{U}}$ matrix, and $\boldsymbol{\alpha}_{\mathrm{U}} \in \mathbb{Z}^{d_t}$, a d_t-dimensional vector:

$$t_{\mathrm{U}}(\mathbf{x}) = \boldsymbol{\lambda}_{\mathrm{U}}\,\mathbf{x} + \boldsymbol{\alpha}_{\mathrm{U}} \tag{3.8}$$

For notational ease, we will occasionally use t without a subscript to refer to whichever timestamp function is appropriate for the respective variable.

Note that only **dependent variables**—variables which appear on the left hand side of an equation in the SARE—are given timestamp functions. **Independent variables** are not defined by recurrence equations, and their values are presumed to be known at the beginning of runtime. Independent variables do not affect the schedule of a program in any way, and may be ignored for the purposes of polyhedral analysis.

[3]The existence of such an affine schedule *is* decidable[8]

```
affine runningSum [N]->{:N>0}
inputs  X: [N]
outputs Y: [N]
let     Y[i] = reduce(+, [j], {:j<=i}: X[j]);
.
```

Figure 3.1: An Alpha program that defines Y as the running sum of X

```
for(c0=0; c0<N; c0++) {
    Y[c0] = 0;
    for(r0=0; r0<=c0; r0++)
        Y[c0] += X[r0];
}
```

Figure 3.2: The program in 3.2 converted to C code, using a schedule of $t_Y(i) = i$

Given a multidimensional iteration domain, a nested loop may be constructed (using a tool such as CLooG [15]) so that the innermost loop iterates over the least significant dimension of the iteration domain, the second innermost loop iterates over the second-least significant dimension, and so on and so forth. Statements are then inserted into the body of the loop such that when a timestamp $\boldsymbol{\tau} \in \mathcal{D}_t$ is visited, all values $\mathrm{U}[\mathbf{x}]$ in the preimage of $\boldsymbol{\tau}$ with respect to t_{U} are computed[4]. Figures 3.1 and 3.2 provide an example input and output to this code-generation process.

Finding a valid schedule for an SARE *without* reductions is a problem that has been addressed by many authors [8, 9, 10, 16, 17, 18, 19]. One of the most generally applicable methods was devised by Feautrier, and involves finding a set of timestamp functions that satisfy the following constraint[5], for every dependence r from a variable U on a variable V [9]:

$$\forall\, \mathbf{x} \in \mathcal{D}_{\mathrm{U}} \;:\; t_{\mathrm{U}}(\mathbf{x}) \succ t_{\mathrm{V}}(r(\mathbf{x})) \tag{3.9}$$

[4]Schedules are usually constructed to be bijective in practice, to save the compiler the trouble of computing said preimage. For analytic purposes, it is often convenient to examine schedules of smaller dimension which are not bijective.

[5]The positivity of a schedule is also typically used as a constraint, but is unnecessary for the purpose of developing existential conditions, as a non-positive schedule can trivially be transformed into a positive one by adding a sufficiently large constant factor to all timestamp functions.

This is the causality constraint, which ensures that no value is computed until the values it depends on have themselves been computed.

Notably, once a schedule satisfies the causality constraint, additional dimensions may be freely added to the schedule, and the constraint will still hold. Conversely, the timestamp functions of a schedule may have more dimensions than are strictly required to satisfy the causality constraints. In this thesis, we refer to the minimal band of dimensions[6] needed to satisfy the causality constraint as the **time dimensions** of the schedule, and the remainder as the **space dimensions**. All points whose timestamps match in the time dimensions correspond to values which may be computed in parallel, meaning the **span** of the program—the time it would take to execute given an unlimited number of parallel processors—is determined solely by the number of time dimensions.

The affine form of Farkas lemma [8] allows for equations such as Equation 3.9—affine, lexicographic inequalities over points in a polyhedron, precicely speaking—to be converted into equivalent constraints in an integer linear program. Thus, schedules may be found for SAREs (without reductions) automatically using a linear program solver.

3.3.1 Tiling

Schedules such as the ones found by the affine form of Farkas' lemma suffer one major drawback: they make no account for data locality. Values which are adjacent in memory are, more often than not, required at distant times. This requires data to be loaded from memory more times than is optimal, forcing the program to become communication-bound.

Tiling is a technique which attempts to solve this problem by partitioning the iteration domain into **tiles**: atomic chunks, which can be made small enough that all of the corresponding data can fit in cache memory [20]. We consider only **linear tilings** [21] in this thesis (and likewise, 'tiling' will henceforth refer to 'linear tiling' unless otherwise stated), in which tiles take the shape of

[6]Discounting **serial dimensions**: dimensions of the schedule whose value is bounded by a constant.

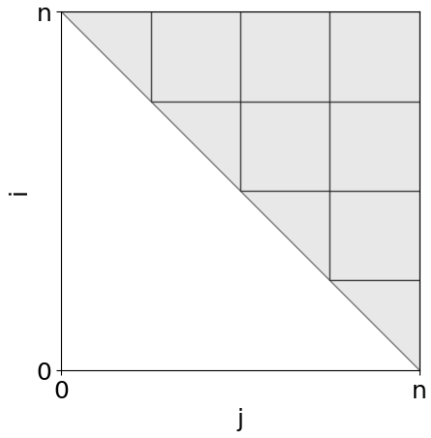

Figure 3.3: A triangular domain, tiled with $\phi_1 = \binom{1}{0}$ and $\phi_2 = \binom{0}{1}$. The tiles have a maximum size of $s_i \times s_j$, regardless of the value of n.

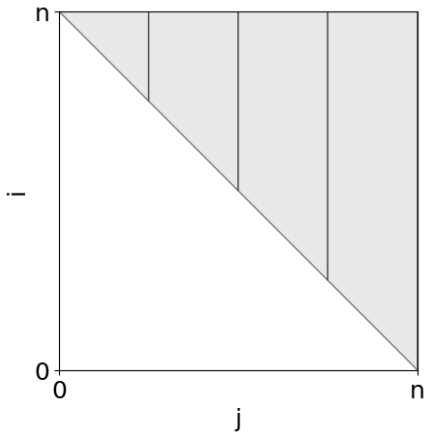

Figure 3.4: The same domain, but tiled only with $\phi_1 = \binom{1}{0}$. The size of the tiles is bounded by $s_1 \times n$, which grows with the size of n.

parallelepipeds. We define an linear tiling as a set of linearly-independent[7] **tiling vectors** $\phi_i \in \mathbb{Z}^{d_t}$ (where d_t is the dimension of the iteration domain $\mathcal{D}_t$), and a set of **tile sizes** $s_i \in \mathbb{Z}$.

For each tiling vector ϕ_i, the iteration space is paritioned by an infinite family of hyperplanes (linear spaces of dimension $d_t - 1$)—each perpendicular to ϕ_i, and a distance of s_i apart from one another. If there are d_t tiling vectors total, then the result is that the iteration space is tiled by parallelapipeds whose faces are each perpendicular to a tile vector, and whose edge lengths are given by the tile sizes.

If there are less than d_t linearly-independent tiling vectors, tiles will be unbounded in at least one direction (as shown in Figure 3.4), and the number of points within will increase by at least a linear factor with respect to the parameters. As a result, the memory footprint of the values corresponding to each tile will quickly exceed the capacity of cache memory, and the generated code will once again become communication-bound. To avoid this problem, we search for only **full tilings** (as in Figure 3.3)—which involve tiles of bounded size, necessitating d_t tiling vectors.

[7] A set of vectors is linearly independent if no vector in the set is equal to a linear combination of the other vectors. This ensures the vectors may be used as a basis for a linear space.

```
affine SquareMatMult [N]->{:N>0}
inputs  A, B: [N,N]
outputs C: [N,N]
let C[i,j]) = reduce(+,[k], {:}: A[i,k] * B[k,j]);
.
```

Figure 3.5: An Alpha program that defines C as product of two square matrices, A and B.

```
for(t0=0; t0 < N/8; t0++)
    for(t1=0; t1 < N/8; t1++)
        for(c0=t0; c0 < t0+8; c0++)
            for(c1=t1; c1 < t1+8; c1++) {
                C[c0, c1] = 0;
                for(r0=0; r0<N; r++)
                    C[c0, c1] = A[c0, r0] * B[r0, c1];
            }
```

Figure 3.6: The program in 3.5 converted to C code, using a schedule of $t_C(i,j) = \binom{i}{j}$, canonical tiling vectors $\phi_1 = \binom{1}{0}$ and $\phi_2 = \binom{0}{1}$, and tile sizes of 8.

Tiling vectors are subject to the following atomicity constraint [8], for every dependence from a variable U to a variable V:

$$\forall \mathbf{x} \in \mathcal{D}_{\mathrm{U}}, \forall i \in [0, d_t) : \phi_i \cdot t(\mathbf{x}) \geq \phi_i \cdot t(r(\mathbf{x})) \tag{3.10}$$

The atomicity constraints induce a partial ordering on each tile, such that the values of each tile only depend on the values of that tile, and those of lower-ordered tiles. As a result, there are no cyclic dependencies between tiles, and thus generated code may visit all the points in one tile before moving to the next.

Given tiling vectors and tile sizes, a **tiled schedule** $\hat{t} : \mathcal{D}_{\mathrm{U}} \rightarrow \mathbb{Z}_{+}^{2d_t}$, can be defined dimension-wise as follows:

[8] This constraint is bilinear, and cannot directly be used in an integer linear program. See Chapter 4 for our workaround to this.

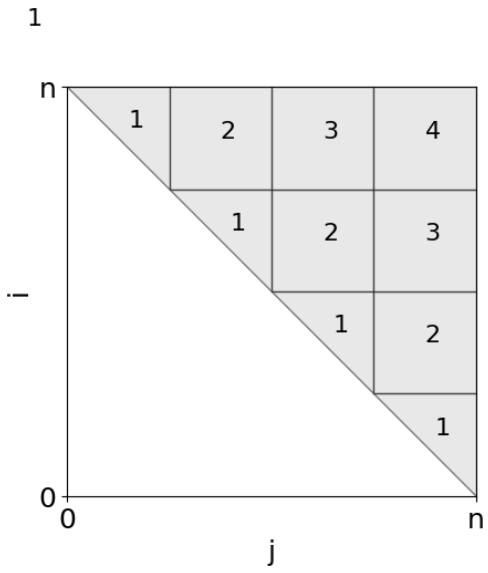

Figure 3.7: The tiled domain from Figure 3.3, labeled with the order in which a wavefront schedule visits the tiles.

$$\hat{t}_i(\mathbf{x}) = \begin{cases} \lfloor \frac{\phi_i \cdot t_i(\mathbf{x})}{s_i} \rfloor & i < d_t \\ t_i(\mathbf{x}) \mod s_i & i \geq d_t \end{cases} \tag{3.11}$$

Code generated using this augmented schedule will visit each tile using the d_t outermost loops, while the d_t innermost loops iterate over all the points in the given tile. An example of a tiled code snippet is given in Figure 3.6.

An additional transformation may be applied to a fully tiled schedule (as shown in Figure 3.7, producing a **wavefront tiled schedule** $\tilde{t} : \mathcal{D}_{\mathrm{U}} \to \mathbb{Z}_{+}^{2d_t}$. In the wavefront schedule, the first dimension of time corresponds to a diagonal strip of tiles, each of which may be executed in parallel [22]:

$$\tilde{t}_i(\mathbf{x}) = \begin{cases} \sum_{0 \leq j < d_t} \hat{t}_j(\mathbf{x}) & i = 0 \\ \hat{t}_i(\mathbf{x}) & i > 0 \end{cases} \tag{3.12}$$

In addition to the performance benefits of a full tiling, a wavefront tiled schedule offers a high degree of coarse-grain parallelism[9].

[9] As well as fine-grain parallelism within each tile; the exploitation of which is beyond the scope of this thesis.

Chapter 4

Properties of Tiling

We now present two powerful theorems regarding the properties of tilings. The first helps to motivate our work, and the second allows us to simplify future theorems.

Theorem 1. *If an SARE can be fully tiled, it permits a schedule with a one-dimensional span.*

Proof. See Appendix A □

This theorem is easiest to understand through its informal proof: If you can tile a schedule, then you can set all of the tile sizes to 1 and take the tile wavefront as your new dimension of time. The tile wavefront satisfies all causality constraints, and thus works as a schedule with one dimension of span.

Recent work on the polyhedral model has produced scheduling algorithms (such as PLuTo [10] and TRACO [23]) which disregard the span of schedules in favor of finding a favorable tiling. However, our Theorem 1 provides a strong motivation for Feautrier-like schedulers that attempt to minimize the span of schedules: The existence of a schedule with one-dimensional span is a strict requirement for a full linear tiling.

Most programs we have analyzed that admit a linear-span schedule are also fully tileable, but this pattern does not always hold. Take, for example, the following contrived SARE that repeatedly finds the perfect shuffle of an array (where $0 \leq i < N$ for some even N, and W is an input variable):

$$\mathrm{Y}[i,j] = \begin{cases} \mathrm{W}[i] & j = 0 \\ \mathrm{Y}[2i, j-1] & j > 0 \wedge 2i \leq N \\ \mathrm{Y}[2i-N-1, j-1] & j > 0 \wedge 2i > N \end{cases} \tag{4.1}$$

The schedule $t_{\mathrm{Y}}(i,j) = (j,i)$ has a one-dimensional span, but no tiling vectors other than $\phi = \hat{j}$ are possible, which makes a full tiling impossible.

For the next theorem, we define a tiling as **rectangular** if every tiling vector is a canonical unit vector. This produces a tiling where every tile is a rectangle whose edges are aligned to the canonical axes.

Theorem 2. *If an SARE can be fully tiled, it permits a full rectangular tiling.*

Proof. See Appendix B □

As a result of Theorem 2, we can fix each tiling vector ϕ_i to a canonical unit vector without making the constraint any more restrictive. This allows us to convert the bilinear Equation 3.10 into the following linear form, which is amenable for use in an integer linear program:

$$\forall\, \mathbf{x} \in \mathcal{D}_{\mathrm{U}}, \forall i \in [0, d_t) : t_{\mathrm{U}}(\mathbf{x}) \geq_i t_{\mathrm{V}}(r(\mathbf{x})) \tag{4.2}$$

Chapter 5

Scheduling SAREs with reductions

Once reductions are introduced to an SARE, the problem of finding a schedule becomes much more difficult. We examine two different ways of approaching this problem.

The first is to explicitly serialize all reductions in the SARE, replacing each with a variable. Each point in a new variable domain corresponds to one point in the original reduction domain, and through a series of self-dependences, a 'result' point in the variable is defined such that it equals the result of the reduction kernel. This approach requires a great deal of pre-processing, and the new variables of the SARE will inevitably contain a large number of subdomains, which, in our experience, translate to an unwieldy number of loop bounds in generated code. Additionally, the added self-dependences will increase the size of the linear program used to find a schedule for the program, leading to a longer compilation time.

The second, and vastly more elegant approach, is to simply assign timestamp functions to reductions as if they were variables. Then, during code generation, the statement in the loop nest corresponding to the schedule of the reduction should take the following form:

$$\texttt{Y(w(x))} \;=\; \texttt{Y(w(x))} \;\oplus\; \texttt{f(...);}$$

Where `Y` is an indexed data structure (such as a multidimensional array) that stores the result of the reduction expression, and `f(...)` is the kernel of the reduction.

Using this second approach, an SARE with reductions may be converted into code without any new variables or subdomains; or any transformations in the polyhedral representations at all, for that matter. All that is needed is a schedule, with timestamp functions for each reduction domain in addition to those for the variables.

5.1 The CRCW Schedule

A naive approach to scheduling reductions is to employ a similar constraint to that of Equation 3.9—one that ensures the points read by each reduction expression are computed before the result is written to the destination variable[10]:

$$\forall x' \in \mathcal{D}_{\mathrm{S}}, \forall i \in [1, k] \ : \ t_{\mathrm{U}}(w(\mathbf{x}')) \succ t_{\mathrm{V}_i}(r_i(\mathbf{x}')) \tag{5.1}$$

There is a fundamental problem with this approach, however. Equation 5.1—adapted from the scalar constraint equation given by Equation 7 in Redon and Feautrier's paper [24, 25]—is constructed under the assumption that an arbitrary number of values may be accumulated under the reduction operator in a single unit of time. For this assumption to hold, the machine used to evaluate the program must make use of **CRCW memory**: it must allow a single memory value to be read and written to concurrently by different processors.

Unfortunately, CRCW memory access is a purely theoretical notion, and cannot be accomplished in reality. Actual machines require a span of at least $O(\log(n))$ to accumulate a reduction of n values [25]. Furthermore, reductions in our target class of programs tend to be too fine-grained for the task to be efficiently distributed on multiple processors, so it is generally inefficient to try to parallelize reductions in this manner. Perhaps more importantly, scheduling a reduction in logarithmic time would require a non-affine schedule, which cannot be discovered or analyzed using the techniques we present here.

5.2 Slack

Furthermore, consider the notion of 'availability' that is induced on a reduction by the timestamp functions of the variables it depends on. A value corresponding to a point in the reduction body is only *available* for computation after all of the value it depends on have been computed. Thus, given a CRCW schedule, it may be the case points in a reduction slice become available

[10] We defer the problem of assigning a timestamp function to S until Chapter 5.3.

gradually, such that there is sufficient time to accumulate each of them—or, it may be the case that all points in a reduction slice become available at once, a constant number of time-steps before the accumulation is scheduled to terminate. In the latter case, the CRCW schedule must be slowed down by a linear factor in order to be realistically viable.

To formally distinguish 'good' schedules from 'bad' ones, we must consider the notion of **slack**: The span of time allotted to accumulate a set of points under a given schedule.

Firstly, we define the slack of a point x' in a reduction domain $\mathcal{D}_{\mathrm{S}}$ as the difference between the timestamp of the point it writes to, and the maximum timestamp of the points it depends on:

$$\mathrm{Slack}_{\mathrm{S}}(\mathbf{x}') = t_{\mathrm{U}}(w(\mathbf{x}')) - \max_{r \in R_S}(t_{\mathrm{V}}(r(\mathbf{x}'))) \tag{5.2}$$

Secondly, we define the slack of a *set of points* as the maximum slack of each of the points within:

$$\mathrm{Slack}_{\mathrm{S}}(\mathcal{D}) = \max_{\mathbf{x}' \in \mathcal{D}}(\mathrm{Slack}_{\mathrm{S}}(\mathbf{x}')) \tag{5.3}$$

So long as the number of points in a reduction slice is bounded by their slack multiplied by some constant factor, the schedule may be slowed down by said factor in order for there to be sufficient time to perform the accumulation [11]. An issue only arises if, as one or more program parameters increase, the slices of the reduction domain have cardinality which increases faster than their slack. When this occurs, the CRCW schedule must be slowed down by a linear factor in order for the processor to have enough time to accumulate each slice.

As an example, consider the Smith-Waterman counting algorithm[12], which computes the number of possible secondary structures of an RNA subsequence (where $\mathrm{Y}[i, j]$ gives the answer for the subsequence from i to j, and l is the minimum loop size):

[11] This was proven by Gupta et. al for monadic reductions [25], and it follows from our soon-to-be-shown Theorem 3 that it applies to polyadic reductions as well.

[12] Introduced in the same paper as, but distinct from the Smith-Waterman algorithm [26]

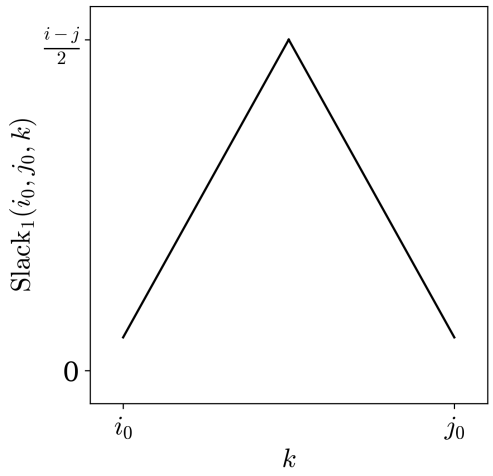

Figure 5.1: The slack of a slice of $\mathcal{D}_{\mathrm{S}}$ for $t_{\mathrm{Y}} = (j - i)$.

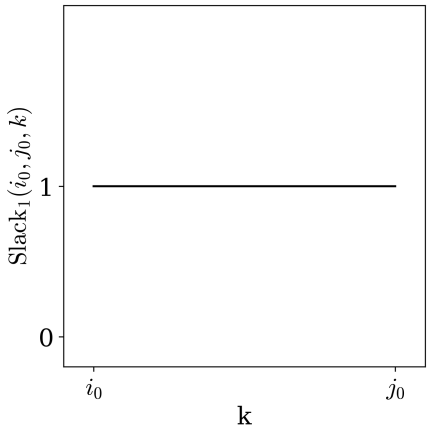

Figure 5.2: The slack of a slice of $\mathcal{D}_{\mathrm{S}}$ for $t_{\mathrm{Y}} = (j)$.

$$\mathrm{Y}[i, j] = \sum_{i \leq k < j-l} \mathrm{Y}[i, k-1] \times \mathrm{Y}[k+1, j-1] \times \text{can\_pair}(k, j)$$

Two possible CRCW schedules for this SARE are $t_{\mathrm{Y}} = (j - i)$, and $t_{\mathrm{Y}} = (j)$. The former schedule produces slices whose points have a first dimension of slack of up to $\frac{j-i}{2}$, as shown in Figure 5.1. These points can be accumulated as soon as their dependent points have been computed, starting with the middlemost point, and working outwards. This slates no more than two points in the reduction body to be computed with the same first dimension of time. The latter schedule, however, gives each slice a first dimension of slack of 1 to work with—as shown in Figure 5.2—forcing every single point in each slice of the reduction to be accumulated in a constant span.

Because of this, an additional dimension of time must be added to the latter schedule in order to have enough slack to accumulate the points in each reduction slice. As a result of Theorem 1, this precludes the possibility of fully tiling the program.

To avoid cases such as this, we wish to find schedules that allow for reduction domains to be accumulated without slowdown. More precisely, we wish to find CRCW timestamp functions for the variables of a program, such that we can subsequently generate timestamp functions for each reduction without increasing the dimension of the schedule's span. We say a CRCW schedule is

lenient if its span is of the lowest possible order (or, equivalently, the schedule has the minimal number of time dimensions), and it allows all reductions to be accumulated without slowdown.

5.3 Piecewise Schedules

We committed a small but deliberate error in the previous section by claiming that $t_Y = (j - i)$ could be used as a lenient CRCW schedule for the counting algorithm. This is not actually possible using our current definition of a schedule. As per the current definition, a reduction timestamp function must be a single affine map over its entire domain. The two vertices of the reduction slice are only computable beginning at the timestamp $(j - i - 1)$, and the extrema of an affine map on a convex polyhedron are always found on its vertices; thus, the minimum and maximum of t_{S} are both equal to $(j - i - 1)$[13]. Each slice of the reduction is once again slated to be accumulated in constant span, which, as previously discussed, is intractable on a realistic machine.

We may resolve this issue by expanding our definition of 'schedule' to include **piecewise schedules**: sets of timestamp functions in which more than one timestamp function may be defined per variable or reduction. The domains of these timestamp functions must form a finite partition of the variable or reduction domain in question.

We can construct a piecewise schedule with appropriate timestamp functions with appropriate domains by determining where each dependence of the reduction **dominates**. We say a dependence r dominates a reduction S in a region $\mathcal{D} \subseteq \mathcal{D}_{\mathrm{S}}$ if and only if for all its points $\mathbf{x}'$, the timestamp of $r(\mathbf{x}')$ is greater than that of any other dependences of $\mathbf{x}'$:

$$\forall\, \mathbf{x}' \in \mathcal{D}, \forall i \in (0, k] \left(t(r(\mathbf{x}')) \succcurlyeq t(r_i(\mathbf{x}'))\right) \tag{5.4}$$

Observe the following fact:

[13] A serial dimension must additionally be inserted so that points in D_{S} and D_{Y} are not scheduled at the same timestamp. See Chapter 7 for details.

Theorem 3. *Given a schedule, a reduction domain may be partitioned into a finite number of polyhedral subdomains such that each individual subdomain is dominated by a single dependence.*

Proof. See Appendix C □

The key term in this theorem is 'polyhedral'. Because each dependence r dominates a polyhedral region of the reduction body, we can assign a timestamp function to that polyhedral region such that each point $\mathbf{x}'$ is computed immediately after $t(r(\mathbf{x}'))$. We refer to this as an 'ASAP accumulation', as every point in the reduction body is scheduled for computation the timestamp immediately after all of its dependences have been computed.

As demonstrated in Section 5.1, there are some SAREs for which a valid CRCW schedule may or may not be lenient, and existing scheduling algorithms are not guaranteed to find schedules that are lenient. We solve this issue in Chapter 6, by deriving linear constraints that can be used to filter for lenient CRCW schedules.

Chapter 6

Automatically Finding Lenient CRCW Schedules

We use the following conventions for our next theorem:

- The **Chebyshev distance** of two points $\|\mathbf{x}_1 - \mathbf{x}_2\|_\infty$ is the maximum absolute value of the components of their difference.
- The **Manhattan distance** of two points $\|\mathbf{x}_1 - \mathbf{x}_2\|_1$ is the sum of the absolute values of the components of their difference.
- The **peak** of a reduction slice is the set of points in the slice with the least slack. Formally,

$$\mathrm{Peak}_\mathrm{S}(\mathbf{x}) = \{\mathbf{x}' \in \mathrm{Slice}_\mathrm{S}(x') \mid \forall\, \mathbf{y}' \in \mathrm{Slice}_\mathrm{S}(x')\, (\mathrm{Slack}_\mathrm{S}(\mathbf{x}') \preccurlyeq \mathrm{Slack}_\mathrm{S}(\mathbf{y}'))\} \tag{6.1}$$

- The **size** $\mathrm{Size}(\mathcal{P})$ of a polyhedron $\mathcal{P}$ is the minimum time needed to compute a reduction over a polyhedron using a linear accumulation scheme. It has an upper bound of the Manhattan distance of its two most distant vertices [25].

Whenever there is no ambiguity, we will use $\mathrm{Slack}(\mathcal{P})$, $\mathrm{Peak}_{\mathbf{x}'}$, and $\mathrm{Slice}_{\mathbf{x}'}$ to refer to whichever reduction (sub)domain is presently under discussion, for the sake of notational ease.

Lemma 1. *A reduction domain with a single dependence may be scheduled without slowdown under a CRCW schedule if and only if the size of its peak is less than its slack*[14]*:*

$$\forall\, \mathbf{x} \in \mathcal{D}_\mathrm{S} \;:\; \mathrm{Slack}(\mathrm{Peak}_{w(\mathbf{x})}) \succcurlyeq \mathrm{Size}(\mathrm{Peak}_{w(\mathbf{x})}) \tag{6.2}$$

This is simply a rephrasing of Gupta et al.'s Theorem 4 using our thesis' terminology [25].

[14] Previously, we have used $\mathbf{x}$ to denote a point in a proper variable domain, and $\mathbf{x}'$ to denote a point in a reduction domain. From this point onward, we use $\mathbf{x}$ to denote points in a reduction domain, for the sake of readability.

Theorem 4. *A reduction domain $\mathcal{D}_{\mathrm{S}}$ admits an accumulation without slowdown under a CRCW schedule if and only if every outgoing dependence r satisfies the constraint in Equation 6.3*

Proof. See Appendix D □

$$\forall\, \mathbf{x}_1, \mathbf{x}_2 \in \mathcal{D}_{\mathrm{S}} \;:\; w(\mathbf{x}_1) = w(\mathbf{x}_2) \implies \mathrm{Slack}(\mathbf{x}_1) + \mathrm{Slack}(\mathbf{x}_2) \succcurlyeq ||\, \mathbf{x}_1 - \mathbf{x}_2 \,||_\infty \tag{6.3}$$

As an intuitive interpretation of Theorem 4, one can imagine drawing a line segment between two arbitrary points in a reduction slice. If the length of any such line segment[15] grows faster than the average slack of its points, then the only way to allow the points to be accumulated in time is by slowing the schedule down by a linear factor—by adding another time dimension to the schedule.

By the definition of the Chebyshev distance, we can rewrite Equation 6.3 as d_{S} distinct constraints, for $0 \leq i < d_{\mathrm{S}}$:

$$\forall\, \mathbf{x}_1, \mathbf{x}_2 \in \mathcal{D}_{\mathrm{S}} \;:\; w(\mathbf{x}_1) = w(\mathbf{x}_2) \implies \mathrm{Slack}(\mathbf{x}_1) + \mathrm{Slack}(\mathbf{x}_2) \succcurlyeq (\mathbf{x}_1 - \mathbf{x}_2)_i \tag{6.4}$$

Each such constraint restricts an affine form to be nonnegative in a convex polyhedron, so the set of constraints may be converted into an integer linear program, alongside the CRCW schedule constraints, using the affine form of Farkas' Lemma.

In Chapter 7, we outline the process of finding and applying piecewise schedules for two simple NPDPs.

[15] Manhattan, Euclidean, or Chebychev length. They are all of the same order.

Chapter 7

Examples

7.1 Illustrative 2D Example

Of the useful NPDPs that we are aware of, all have reduction bodies that are three-dimensional or greater. Given the inherent difficulty of parsing three-dimensional figures, however, we have elected to first present a contrived two-dimensional example for the sake of reader comprehension.

Take the following SARE, which defines each element in Y as the dot product of all the previous elements of the array with the reverse, plus some initial value in W[16]:

$$\mathrm{Y}[i] = \mathrm{W}[i] + \sum_{0<j<i} \mathrm{Y}[j] \cdot \mathrm{Y}[i-j] \tag{7.1}$$

Figure 7.1 illustrates how points in this SARE depend on one another.

The only possible timestamp functions for Y are of the form $t_{\mathrm{Y}}(i) = (ai+b, ...)$, with the most natural choice being $t_{\mathrm{Y}}(i) = (i)$ (as illustrated in Figure 7.2). As a result, there is no need to search for a schedule using the constraints from Theorem 4. Either the given schedule allows for ASAP reduction accumulation, or it does not.

We first split the reduction domain, $\mathcal{D}_{\mathrm{S}}$, according to the dominant dependence. Let $\mathcal{D}_{\mathrm{S},1}$ be defined as the subdomain where the dependence on $\mathrm{Y}[j]$ dominates, and $\mathcal{D}_{\mathrm{S},2}$ be the subdomain where the dependence on $\mathrm{Y}[i-j]$ dominates. The timestamp of the first dependence is $t(j) = (j)$, and the timestamp of the second is $t(i-j) = (i-j)$. Therefore, we obtain the following subdomains:

$$\begin{aligned} \mathcal{D}_{\mathrm{S},1} &= \{(i,j) \in \mathcal{D}_{\mathrm{S}} : j \geq i-j\} \\ \mathcal{D}_{\mathrm{S},2} &= \{(i,j) \in \mathcal{D}_{\mathrm{S}} : j < i-j\} \end{aligned} \tag{7.2}$$

[16] Without the introduction of W, this SARE would represent a scan, which could be simplified to a linear-time computation. [27]

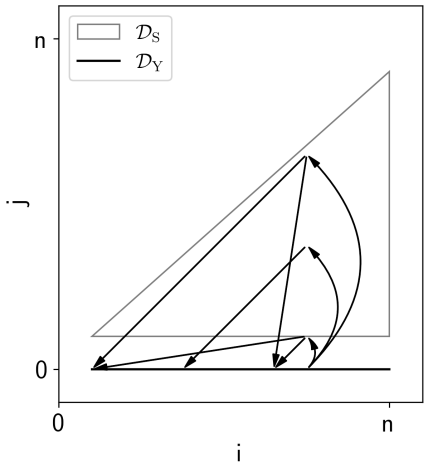

Figure 7.1: The incoming and outgoing dependences of three points in $\mathcal{D}_S$. Arrows point from consumer to producer.

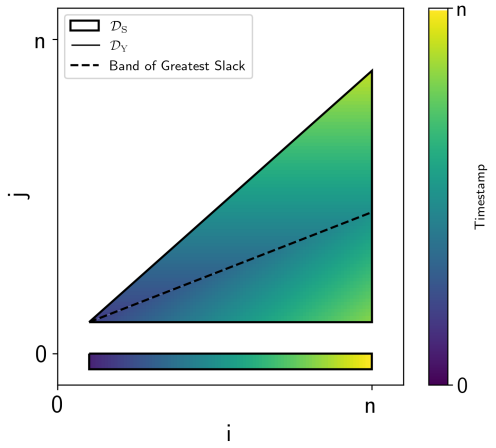

Figure 7.2: $\mathcal{D}_Y$ and $\mathcal{D}_S$, with a schedule of $t_Y(i) = (i)$. The gradient on $\mathcal{D}_Y$ indicates the timestamp of each point, and the gradient on $\mathcal{D}_S$ indicates the timestamp of each point's dominant dependence.

Note that the split we made is along the dashed line in Figure 7.2: the band of points which have the earliest timestamp, and therefore the greatest slack in their respective slices.

Next, we insert a serial dimension so that the values in the reduction are not scheduled at the same time as their dependences or dependees, and assign each subdomain of the reduction domain an ASAP schedule according to the dominant dependence:

$$\begin{aligned} t_Y(i) &= (i, 0) \\ t_{S,1}(i, j) &= (j, 1) \\ t_{S,2}(i, j) &= (i - j, 2) \end{aligned} \tag{7.3}$$

As can be seen in Figure 7.3, each subdomain has bounded peaks, so this is a valid ASAP accumulation scheme.

Finally, we use PLuTo-like constraints to search for the third dimension of each schedule. This ensures that schedule admits a canonical tiling.

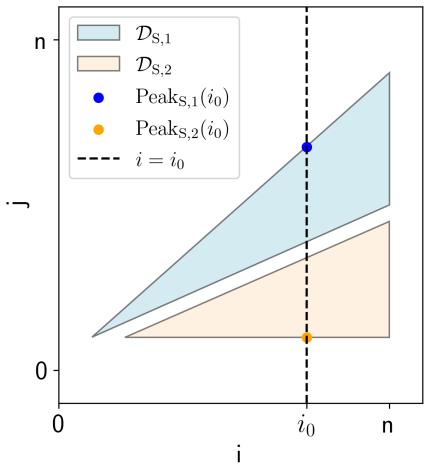

Figure 7.3: $\mathcal{D}_S$ split into two subdomains, as described in Equation 7.3. As the peaks of $\mathcal{D}_{S,1}$ lie along its top edge, the peaks along $\mathcal{D}_{S,2}$ lie along its bottom edge, and each slice is parallel to the j axis, it is clear by inspection that the slices of each subdomains have a singular peak.

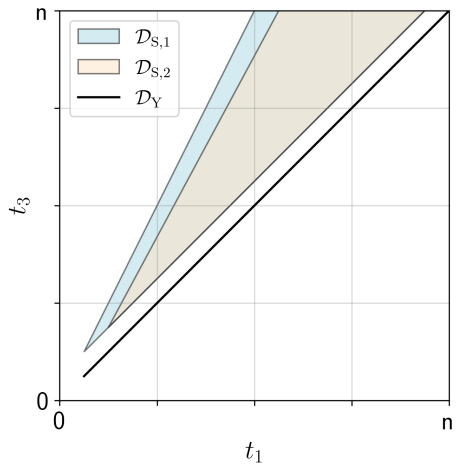

Figure 7.4: The images of each domain with respect to their timestamp functions (with the sequencing dimension omitted). The grid marks show a possible canonical tiling.

$$
\begin{aligned}
t_Y(i) &= (i, 0, i) \\
t_{S,1}(i, j) &= (j, 1, i) \\
t_{S,2}(i, j) &= (i - j, 2, i)
\end{aligned} \tag{7.4}
$$

Figure 7.4 shows the iteration domain that results from this schedule.

7.2 Smith-Waterman Counting Algorithm

In the previous example, finding a lenient CRCW schedule was a trivial matter, as there was only one reasonable choice. We now present a program for which scheduling is a much less trivial matter.

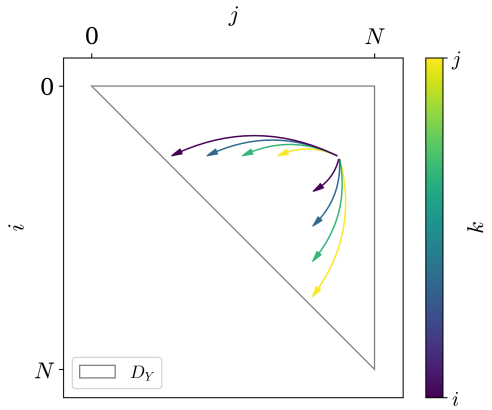

Figure 7.5: Dependences from D_S to D_Y. The k component of the consumers in D_S (at the tail of each arrow) is indicated by the color of the arrow.

Recall the SARE for the Smith-Waterman counting algorithm:

$$\mathrm{Y}[i,j] = \sum_{i \le k < j-l} \mathrm{Y}[i,k-1] \times \mathrm{Y}[k+1,j-1] \times \mathrm{can\_pair}(k,j) \tag{7.5}$$

Figure 7.5 illustrates how points in this SARE depend on one another.

One obvious schedule for Y is $t_Y(i,j) = (j, ...)$, as all points $\binom{i}{j}$ depend on some $\binom{i'}{j'}$, where $j' < j$. As a CRCW schedule, the first dimension satisfies Equations 3.9 and 5.1, and the schedule is therefore of linear span. In practice, however, this schedule demands each slice of the reduction be accumulated in constant time, which is impossible, as the number of points in each slice is unbounded.

Let us instead use the constraints in Theorem 4 to narrow our schedule space. This is not something that can readily be done by hand, but running a Feautrier-like scheduler with the additional constraints (or by guessing and checking), we find that the schedule $t_Y(i,j) = (j-i)$ admits an ASAP accumulation. Using this to split the reduction domain, we obtain the following subdomains (as illustrated in Figure 7.6):

$$\begin{aligned}\mathcal{D}_{S,1} &= \{(i,j,k) \in \mathcal{D}_S : (k-1) - (i) \ge (j-1) - (k+1)\} \\ \mathcal{D}_{S,2} &= \{(i,j,k) \in \mathcal{D}_S : (k-1) - (i) < (j-1) - (k+1)\}\end{aligned} \tag{7.6}$$

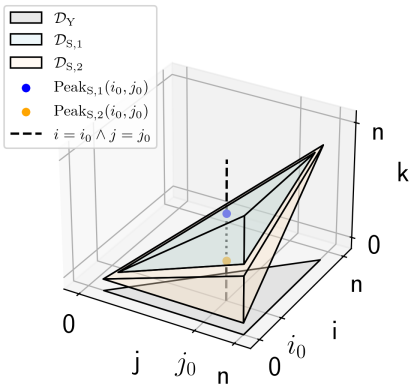

Figure 7.6: $\mathcal{D}_{\mathrm{S}}$ split into two subdomains, as described in Equation 7.6. Although harder to see than in a 2D example, the intersection of each slice with the top face of $\mathcal{D}_{\mathrm{S},1}$ and bottom face of $\mathcal{D}_{\mathrm{S},2}$ are all single points.

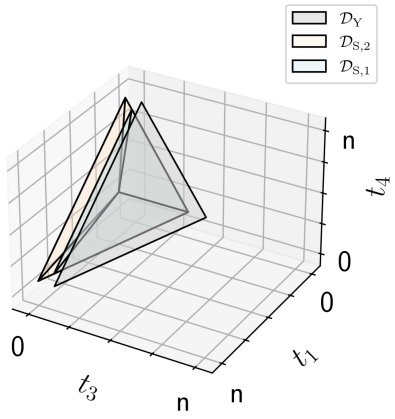

Figure 7.7: The images of each domain with respect to their timestamp functions (with the sequencing dimension omitted). The grid marks show a possible canonical tiling.

Once again, we assign the reduction subdomains ASAP schedules with an added serial dimension, and use a PLuTo-like scheduler to find a tileable third and fourth dimension.

$$t_{\mathrm{Y}}(i, j) = (j - i, 0, n - i, j)$$

$$t_{\mathrm{S},1}(i, j, k) = (k - i, 1, n - i, j)$$

$$t_{\mathrm{S},2}(i, j, k) = (j - k - 1, 2, n - i, j)$$

Figure 7.7 shows the iteration domain that results from this schedule.

Chapter 8

Implementation

We used the AlphaZ system [28] to provide a proof-of-concept implementation of our findings. AlphaZ makes use of ISL [29] to interpret and apply a diverse array of transformations to Alpha programs [30], which are essentially rich representations of an SARE.

Our implementation takes Alpha files as input, and extracts their parametric reduced dependence graph (the graph representing the dependence of each variable on another; the PRDG, for short) for analysis. For each reduction in the PRDG, we derive linear constraints as per Equation 6.4. Two linear programs are then run: one with Feautrier's constraints, one with PLuTo's, with the aforementioned constraints appended to each. The time dimensions of the Feautrier schedule are then prepended to the PLuTo schedule, which yields a schedule of minimum possible span, and maximum number of tileable dimensions. The bodies of each reduction are then manually given piecewise schedules such that each point in the domain is manually scheduled to be executed immediately after its dependences are available. Finally, a tiling is found using Renganarayan et al.'s algorithm for fixed tile sizes [31], and applied along with a wavefront schedule. This schedule may then be used for code generation, if desired.

8.1 Evaluation

To evaluate our approach, we used our implementation to optimize several of the programs in Palkowski and Bielecki's NPDP benchmark suite [7]. We implemented the programs in Alpha, and used AlphaZ to analyze them and apply our optimizations.

Of the 10 kernels featured in the benchmark suite, 3 (Smith–Waterman, Smith–Waterman 3D, and Needleman–Wunsch) are not *recursively* polyadic, as their polyadic reductions feature one recursive dependence, and one dependence on an input variable. As a result, it is possible to find a fully-tileable schedule for the programs using existing techniques, and our approach offers no significant improvement. An additional 2 programs (Zuker and mea) yielded a fully-tileable

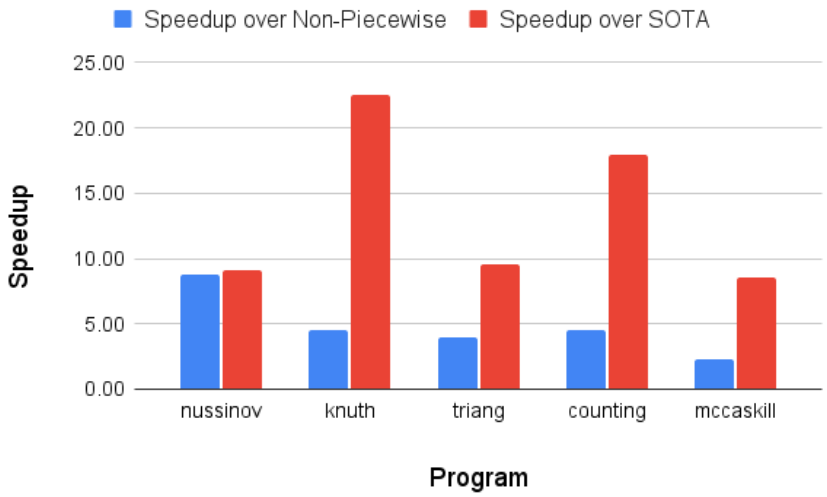

Figure 8.1: Speedup of piecewise AlphaZ code, compared to non-piecewise AlphaZ code and State-of-the-Art code (the faster of PLuTo and TRACO for each program)

piecwise schedule, but due to the number of variable subdomains created in the process, the code generated by AlphaZ was too bloated to compile. Possible solutions for this will be discussed in Chapter 10.

To maintain parity with the benchmark code, we used a tile size of 16 for all programs, and generated OpenMP parallel directives before the outermost loops that permitted it. Our code was compiled using clang (version 19.1.7) with the -O3 flag.

We generated two versions of code for each of the programs. One version utilized our technique to find a piecewise schedule and fully tile the code, and one did not. These were then tested against the PLuTo and TRACO code from the benchmark on a 13th Gen Intel® Core^{TM} i9-13900K (5.8 GHz, 24 cores, 32 threads, 36M/32M/896k cache). An input size of N=10,000 was used for all programs, and their mean runtime was computed over 10 trials.

8.2 Experimental Results

Our numerical results are summarized in Figure 8.1, and shown in full in Figure E.1. The 'Speedup over Non-Piecewise' row displays how much faster the piecewise AlphaZ code was

compared to the non-piecewise version. The 'Speedup over State-of-the-Art' displays how much faster our piecewise code was compared to the better of the PLuTo- and TRACO-generated code.

Our piecewise code was 2.3-8.6$\times$ faster than our non-piecewise code on the given benchmarks. Additionally, and quite unexpectedly, our non-piecewise code was faster than the PLuTo- and TRACO-generated code for all programs. A cursory examination of the generated code revealed a noteworthy pattern: The tile wavefronts in our AlphaZ-generated code consisted mostly of tiles with near-identical size, whereas the tile wavefronts in the PLuTo- and TRACO-generated code consisted of tiles of uneven size. This results in load imbalance between different threads—some processors will finish computing the points in their assigned tile(s), and have to wait for the other processors to finish with their workloads in order to synchronize—which we suspect is the primary reason for the discrepancy in execution time.

This was not something we anticipated, as we paid little mind to load balance while designing our scheduler; rather, it appears to be a result of the fact that our tile wavefront happened to be oriented such that each tile in the wavefront had reduction slices of equal size, for all tested programs. This is not a necessary consequence of using a wavefront schedule; if we were to skew just one of the tiling vectors, then this would misalign our tiling wavefront, and the load imbalance would be re-introduced.

Further research into tiling schemes which minimize load imbalance is warranted.

Chapter 9

Conclusion

Many important programs—especially in the field of bioinformatics—fall into the class of nonserial polyadic dynamic programs, and are particularly difficult to optimize. The polyhedral model is a framework used to optimize programs with static control parts, but little research has been done in the field into the automatic optimization of NPDPs.

linear tiling is a transformation that grants a high degree of coarse-grain parallelism and data locality to programs that admit it. We have shown that an affine schedule with linear span is necessary for a full linear tiling. NPDPs often admit a linear-span schedule, but require a particular index set splitting as a prerequisite.

The linear programming constraints we presented are able to find such an index set splitting along with a schedule for a program. This typically also allows for a full linear tiling. Our experiments have shown massive speedup using this technique to optimize a set of benchmark programs.

9.1 Threats to Validity

Firstly, it bears repeating that all of our experiments involved Alpha-to-C transpilation, rather than the C-to-C transpilation that most polyhedral compilers perform. Because of this, our results are *not* a demonstration of superiority over existing polyhedral compilers. This would not be a fair comparison, as, by using Alpha programs as input, AlphaZ is able to skip the complex task of raising C code to a polyhedral representation. Rather, our results demonstrate the speedup gained specifically by using our approach to find piecewise schedules.

Secondly, the scope of our testing was minimal, with a single set of parameters tested on a single machine. It is possible that the speedup we found does not generalize to the extent that we would hope. For that that reason, further testing with a broader set of tile sizes and target machines is warranted.

Thirdly, we were only able to compile and test programs with reduction bodies of up to 3 dimensions. We believe our approach will work well on reduction bodies of higher dimension (such as those in Zuker or mea), but we cannot generalize our findings to higher-dimensional programs as it is. Improving AlphaZ codegen to allow for the compilation of more complex programs is our highest priority, going forward.

9.2 Future Work

Due to the limit of this thesis' scope, there are some possible avenues of research that remain unaddressed. We list several below, some or all of which we hope to address in follow-up papers:

- **Optimization of Memory Allocation**: Current polyhedral compilers (including AlphaZ) tend to overallocate memory for each variable in generated code. Typically, memory is allocated using a multidimensional array, which represents the bounding box of the corresponding variable domain. This results in an amount of wasted memory which grows exponentially with the dimension of the variable.

  One strategy we tried was to allocate single-dimensional arrays, and then index into them using Barvinok's rank function, which assigns each point in a polyhedron a sequential integer 'rank'. The resulting indexing expression, however, also grows exponentially with the dimension of the variable, causing unacceptable slowdowns during code generation, compilation, and runtime.

  The use of arrays-of-arrays to represent variable domains, rather than indexing into a single-dimensional array, might be a good trade-off between excess memory allocation and slowdown due to complex indexing expressions.

- **Simplified Tile Bounds**: Presently, AlphaZ codegen constructs the intra-tile loops such that only points which lie within the iteration domain proper are visited. As a result, tiled code necessitates loop bounds that are drastically more complex than those of untiled code; these

overly-complex loop bounds are the primary reason our generated code for Zuker and mea was unable to compile.

One possible solution to this issue might be to do away with these intra-tile loop bounds entirely, and simply visit every point inside each visited tile. Because the number of partial tiles[17] grows an order of magnitude lower than the total number of tiles, the proportion of unnecessary computations incurred by this strategy would decrease as the problem size increases.

- **Processor Load Balance Optimization**: When a partial tiling is applied a program, it is often the case that the size of the tiles grows in a certain direction of the iteration space. If the tile wavefront is oriented such that different processors operate on tiles of different sizes, then there will be a period of time prior to synchronization in which one or more processors will be idle.

  Optimally, the tiling wavefront vector will be oriented such that all tiles computed in parallel are of similar sizes. Rather than obtaining the wavefront vector by simply summing the tiling vectors, it might be prudent to use a positive combination of tiling vectors that points in the direction of increasing tile size. Alternatively, it might be necessary to introduce additional constraints when searching for tile vectors.

- **Interaction with Simplification**: Simplification is a technique for eliminating redundant computation in programs with reductions. In some cases, this can reduce the amount of work done by a polynomial degree. An algorithm for automatically simplifying SAREs has recently been discovered, and the class of targeted programs overlaps with ours [27].

  There are often multiple ways to simplify a program, and it might be the case that some simplifications admit faster piecewise schedules than others. If so, can additional constraints be formulated for the simplification step to ensure the resulting program admits a good schedule?

[17] tiles which are not fully contained within the iteration domain

Chapter 10

Related Work

Karp et al. described a polyhedral method for finding single-dimensional schedules for systems of uniform recurrence equations in [32], and Lamport applied this method to Fortran code. Karp's results were generalized to a conservative class of SAREs by Delosme and Ipsen [16]. Rajopadhye et al. later developed constructive method for determining whether affine recurrence equations could be pipelined into equivalent uniform recurrence equations [17]. Quinton and Van Dongen formalized SAREs with more rigor, introducing the concept of system normalization, and solving the problem of scheduling SAREs with single-dimensional schedules [18]. The class of schedulable SAREs was then broadened by Rajopadhye et al., whose algorithm used a vertex-based computation to find piecewise linear schedules [19]. Feautrier provided a more efficient algorithm based on Farkas' lemma, which he extended to make use of multidimensional affine schedules [8][9]. Modern polyhedral scheduling algorithms utilize Farkas' lemma in the same manner as Feautrier's scheduling algorithm.

Redon and Feautrier provided the first method for scheduling reductions under the CRCW model [24]. Gautam et al. devised additional constraints for the above method that would find a lenient CRCW schedule for non-polyadic reductions [25], and recognized that their method could work on polyadic reductions, if a method to properly split their domains could be found. Separately, Griebl et al. developed a method for performing index set splits on SAREs, but did not consider how this could be applied to reductions [33]. Our thesis is the first to provide a general-purpose method for finding index set splits for reductions.

Quilleré et al. provided the first working algorithm for merging scheduled variable domains into a single loop nest [34]. Their algorithm was extended and implemented by Bastoul, allowing for polyhedral source-to-source transpilation [35]. Many other authors provided their own polyhedral compilers, including PoCC [36], PLuTo [10], and TRACO [23]. All of these compilers have

their strengths and weaknesses—and most are more mature than AlphaZ overall—but none are capable of finding optimal piecewise schedules for programs with reductions.

A body of recent polyhedral research has focused on finding ways to optimize bioinformatics algorithms with polyadic reductions [2, 3, 4, 11, 37, 38]. These papers mainly explore manual optimizations to specific programs, rather than more general-purpose compiling techniques. A method for automatically simplifying programs with reductions—reducing their work complexity by a linear factor or more—has also been developed [27], and there is some overlap between their target class of programs and ours. Investigation into how their method interacts with ours is warranted.

Bibliography

[1] L. J. Guibas, H. T. Kung, and C. D. Thompson, "Direct VLSI implementation of combinatorial algorithms," 1979. [Online]. Available: https://api.semanticscholar.org/CorpusID:6109561

[2] S. Varadarajan, "Polyhedral optimizations of rna-rna interaction computations," Master's thesis, Colorado State University, 2017.

[3] A. Ebrahimpour-Boroojeny, S. Rajopadhye, and H. Chitsaz, "BPPart and BPMax: RNA-RNA interaction partition function and structure prediction for the base pair counting model," 2020. [Online]. Available: https://arxiv.org/abs/1904.01235

[4] V. V. Save, "ViennaRNA– optimizing a real-world RNA folding program," 2023.

[5] R. Lorenz, S. H. Bernhart, C. Höner zu Siederdissen, H. Tafer, C. Flamm, P. F. Stadler, and I. L. Hofacker, "Viennarna package 2.0," *Algorithms for molecular biology*, vol. 6, no. 1, p. 26, 2011.

[6] G.-J. Li and B. W. Wah, "Parallel processing of serial dynamic programming problems," in *Proceedings of COMPSAC*, vol. 85, 1985, pp. 81–89.

[7] M. Palkowski and W. Bielecki, "NPDP benchmark suite for the evaluation of the effectiveness of automatic optimizing compilers," *Parallel Computing*, vol. 116, p. 103016, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819123000224

[8] P. Feautrier, "Some efficient solutions to the affine scheduling problem: I. one-dimensional time," *Int. J. Parallel Program.*, vol. 21, no. 5, p. 313–348, Oct. 1992. [Online]. Available: https://doi.org/10.1007/BF01407835

[9] ——, "Some efficient solutions to the affine scheduling problem part ii multidimensional time," *International Journal of Parallel Programming*, vol. 21, 01 1997.

[10] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Affine transformations for communication minimal parallelization and locality optimization of arbitrarily-nested loop sequences," The Ohio State University, Tech. Rep. OSU-CISRC-5/07-TR43, May 2007.

[11] V. Save, C. Mondal, and S. Rajopadhye, "Program optimization vs. software engineering in RNA computations," 2025, unpublished manuscript.

[12] M. Berger, *Geometry i*. Springer Science & Business Media, 2009.

[13] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *International Conference on Compiler Construction*. Springer, 2010, pp. 283–303.

[14] Y. Saouter and P. Quinton, "Computability of recurrence equations," *Theoretical Computer Science*, vol. 116, no. 2, pp. 317–337, 1993.

[15] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* IEEE, 2004, pp. 7–16.

[16] J. M. Delosme and I. C. F. Ipsen, "Systolic array synthesis: computability and time cones," in *Proceedings of the International Workshop on Parallel Algorithms & Architectures*. NLD: North-Holland Publishing Co., 1986, p. 295–312.

[17] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto, "On synthesizing systolic arrays from recurrence equations with linear dependencies," in *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 1986, pp. 488–503.

[18] P. Quinton and V. Van Dongen, "The mapping of linear recurrence equations on regular arrays," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 1, no. 2, pp. 95–113, 1989.

[19] S. Rajopadhye, L. Mui, and S. Kiaei, "Piecewise linear schedules for recurrence equations," in *6th IEEE Workshop on VLSI Signal Processing*. Institute of Electrical and Electronics Engineers Inc., 1992, pp. 375–384.

[20] J. Ramanujam and P. Sadayappan, "Tiling multidimensional iteration spaces for multicomputers," *Journal of Parallel and Distributed Computing*, vol. 16, no. 2, pp. 108–120, 1992. [Online]. Available: https://www.sciencedirect.com/science/article/pii/074373159290027K

[21] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1988, pp. 319–329.

[22] M. E. Wolf, *Improving locality and parallelism in nested loops*. stanford university, 1992.

[23] M. Palkowski, T. Klimek, and W. Bielecki, "TRACO: An automatic loop nest parallelizer for numerical applications," in *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 2015, pp. 681–686.

[24] X. Redon and P. Feautrier, "Scheduling reductions," in *Proceedings of the 8th international conference on Supercomputing*, 1994, pp. 117–125.

[25] G. Gupta, S. Rajopadhye, and P. Quinton, "Scheduling reductions on realistic machines," 08 2002, pp. 117–126.

[26] M. S. Waterman and T. F. Smith, "RNA secondary structure: A complete mathematical analysis," *Mathematical Biosciences*, vol. 42, no. 3-4, pp. 257–266, 1978.

[27] L. Narmour, T. Yuki, and S. Rajopadhye, "Maximal simplification of polyhedral reductions," *Proceedings of the ACM on Programming Languages*, vol. 9, no. POPL, p. 67–94, Jan. 2025. [Online]. Available: http://dx.doi.org/10.1145/3704839

[28] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye, "AlphaZ: A system for design space exploration in the polyhedral model," in *Languages and Compilers for Parallel Computing*,

H. Kasahara and K. Kimura, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 17–31.

[29] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *International Congress on Mathematical Software*. Springer, 2010, pp. 299–302.

[30] C. Mauras, "Alpha : un langage equationnel pour la conception et la programmation d'architectures paralleles synchrones," Ph.D. dissertation, 1989, thèse de doctorat dirigée par Quinton, Patrice Informatique Rennes 1 1989. [Online]. Available: http://www.theses.fr/1989REN10116

[31] L. Renganarayanan, D. Kim, M. M. Strout, and S. Rajopadhye, "Parameterized loop tiling," *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 1, May 2012. [Online]. Available: https://doi.org/10.1145/2160910.2160912

[32] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *Journal of the ACM (JACM)*, vol. 14, no. 3, pp. 563–590, 1967.

[33] M. Griebl, P. Feautrier, and C. Lengauer, "Index set splitting," *International Journal of Parallel Programming*, vol. 28, no. 6, pp. 607–631, 2000.

[34] F. Quilleré, S. Rajopadhye, and D. Wilde, "On code-generation in the polyhedral model," 2001.

[35] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* IEEE, 2004, pp. 7–16.

[36] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *International Conference on Compiler Construction*. Springer, 2010, pp. 283–303.

[37] G. Rizk, D. Lavenier, and S. Rajopadhye, "Gpu accelerated rna folding algorithm," in *GPU Computing Gems Emerald Edition*. Elsevier, 2011, pp. 199–210.

[38] J. Li, S. Ranka, and S. Sahni, "Multicore and gpu algorithms for nussinov rna folding," in *2013 IEEE 3rd International Conference on Computational Advances in Bio and medical Sciences (ICCABS)*. IEEE, 2013, pp. 1–2.

Appendix A

Proof of Theorem 1

If an SARE can be fully tiled, it permits a schedule with a one-dimensional span.

Proof. We will prove this constructively. Let there exist an SARE with a schedule and a full tiling. That is to say that Equations 3.9 and 4.2 are satisfied for all dependences r from a variable U to a variable V. We define a new schedule, $t'(\mathbf{x})$, such that its first dimension is calculated as follows:

$$t'_1(\mathbf{x}) = \sum_{0 \leq i < d_t} \phi_i \cdot t(\mathbf{x}) \tag{A.1}$$

Consider the tiling matrix Φ whose ith row is ϕ_i. Because each ϕ_i is defined as linearly independent, this matrix is invertible, and thus bijective when used as a transformation.

For each $x \in D_U$, there therefore must exist at least one i such that

$$\phi_i \cdot t(\mathbf{x}) > \phi_i \cdot t(r(\mathbf{x})) \tag{A.2}$$

Because otherwise, $\Phi t(\mathbf{x})$ and $\Phi t(r(\mathbf{x}))$ would be equal for some $t(\mathbf{x}) \succ t(r(\mathbf{x}))$, and that would contradict the fact that Φ is bijective.

Thus, by inequality properties,

$$\sum_{0 \leq i < d_t} \phi_i \cdot t(\mathbf{x}) > \sum_{0 \leq i < d_t} \phi_i \cdot t(r(\mathbf{x})) \tag{A.3}$$

$$t'_1(\mathbf{x}) > t'_1(r(\mathbf{x})) \tag{A.4}$$

The other dimensions of t' do not matter, as the first fully satisfies the causality constraints. Therefore, t' has one-dimensional span.

□

Appendix B

Proof of Theorem 2

If an SARE can be fully tiled, it permits a full rectangular tiling.

Proof. We construct $t'(\mathbf{x})$ in the same manner as in the proof of Theorem 1. Additionally, we define each dimension of the schedule after the first as follows:

$$t'_i(\mathbf{x}) = \phi_i \cdot t(\mathbf{x}) \tag{B.1}$$

Then, let each new tiling vector ϕ'_i be the unit vector along the ith dimension of the iteration domain. That is to say, $\phi'_i \cdot t'(\mathbf{x}) = t'_i(\mathbf{x})$.

We already know t' satisfies causality constraints from Equation A.4.

From the same equation, it follows that the ϕ' satisfies the tiling constraints in its first dimension:

$$\phi'_1 \cdot t(\mathbf{x}) > \phi'_1 \cdot t(r(\mathbf{x})) \tag{B.2}$$

And, because we defined our new schedule and tiling vectors such that, for the remaining dimensions,

$$\phi'_i \cdot t'(\mathbf{x}) = \phi_i \cdot t(\mathbf{x}) \tag{B.3}$$

It follows from the validity of the original tiling that ϕ' must satisfy Equation 4.2 for $i > 1$ as well.

Therefore, t' and ϕ' satisfy Equation 4.2, meaning ϕ' is a valid full rectangular tiling.

□

Appendix C

Proof of Theorem 3

Proof. Given a schedule, a reduction domain may be partitioned into a finite number of polyhedral subdomains such that each individual subdomain is dominated by a single dependence.

Let S be a reduction, let t refer to the timestamp function appropriate to its arguments, and let $[r_0, r_1 ... r_{m-1}]$ be the list of outgoing dependences from S. We partition $\mathcal{D}_\mathrm{S}$, the domain of S, as follows:

$$D_{\mathrm{S},i} = D_\mathrm{S} \cap \bigcap_{0 \leq j < m} \left\{ \mathbf{x} \in \mathbb{Z}^{d_\mathrm{S}} \;\middle|\; \begin{array}{ll} t(r_i(\mathbf{x})) \succ t(r_j(\mathbf{x})) & \text{if } i > j \\ t(r_i(\mathbf{x})) \succcurlyeq t(r_j(\mathbf{x})) & \text{if } i \leq j \end{array} \right\}$$

Firstly, each of these subdomains is a polyhedron, as they are the intersections of the polyhedron $\mathcal{D}_\mathrm{S}$ with a finite number of affine constraints.

Secondly, these subdomains are disjoint, as for any two distinct subdomains $\mathcal{D}_{\mathrm{S},i}$ and $\mathcal{D}_{\mathrm{S},j}$ (where, without loss of generality, $i > j$) the former has the inequality $t(r_i(\mathbf{x})) \succ t(r_j(\mathbf{x}))$ and the latter has the negated inequality $t(r_i(\mathbf{x})) \preccurlyeq t(r_j(\mathbf{x}))$.

Thirdly, each subdomain $\mathcal{D}_{\mathrm{S},i}$ is dominated by dependence r_i, as the subdomain is, by definition, constrained by the inequality $t(r_i(\mathbf{x})) \succcurlyeq t(r_j(\mathbf{x}))$ for all j.

Fourthly, the union of these subdomains is $\mathcal{D}_\mathrm{S}$, which we prove as follows. Let $\mathbf{x}$ be a point in $\mathcal{D}_\mathrm{S}$, and i be the lowest integer such that r_i dominates $\mathbf{x}$. $\mathbf{x}$ must belong to $\mathcal{D}_{r,i}$, as by the definition of dominance, $t(r_i(\mathbf{x})) \succcurlyeq t(r_j(\mathbf{x}))$ must hold for all j. Furthermore, it must be the case that $t(r_i(\mathbf{x})) \succ t(r_j(\mathbf{x}))$ for all $i > j$, because $t(r_i(\mathbf{x})) = t(r_j(\mathbf{x}))$ would contradict the fact that i is the lowest integer for which the corresponding dependence dominates $\mathbf{x}$. Thus, every point in $\mathcal{D}_\mathrm{S}$ must belong to at least one $\mathcal{D}_{\mathrm{S},i}$.

In conclusion, each $\mathcal{D}_{\mathrm{S},i}$ is dominated by a single dependence, and together, they form a partition of $\mathcal{D}_\mathrm{S}$.

□

Appendix D

Proof of Theorem 4

Proof. A reduction domain $\mathcal{D}_{\mathrm{S}}$ may be scheduled without slowdown under a CRCW schedule if and only if every outgoing dependence r satisfies the constraint in Equation 6.3

Via Theorem 3, $\mathcal{D}_{\mathrm{S}}$ may be partitioned into polyhedra that are each dominated by a single dependence, each of which may be scheduled separately. If all of these polyhedra may be scheduled without slowdown, then so may the combined domain, via a piecewise schedule.

Let $\mathcal{D}$ be the subset of $\mathcal{D}_{\mathrm{S}}$ dominated by r. The non-dominant dependences of $\mathcal{D}$ do not impact its schedule, so it may be treated as a reduction with a single dependence for the purpose of scheduling. Thus, Lemma 1 applies to $\mathcal{D}$, and we can make use of Equation 6.2.

Next, note that if either $\mathrm{Sl}(\mathbf{x}_1)$ or $\mathrm{Sl}(\mathbf{x}_2)$ is truly multidimensional (any dimension but the last is nonzero), Equation 6.4 holds, and the reduction can be accumulated without slowdown, as accumulation can be performed with one dimension of time. Thus, the only nontrivial case is when both slacks are only nonzero in their final dimension. For the remainder of this proof, we will refer to the scalar versions of Equations 6.2 and 6.4, with Slack now referring to the final dimension of the slack:

$$\forall x \in \mathcal{D} \ : \ \mathrm{Slack}(\mathrm{Peak}_{w(\mathbf{x})}) \geq \mathrm{Size}(\mathrm{Peak}_{w(\mathbf{x})}) \tag{D.1}$$

$$\forall\, \mathbf{x}_1, \mathbf{x}_2 \in \mathcal{D} \mid w(\mathbf{x}_1) = w(\mathbf{x}_2) \implies \mathrm{Sl}(\mathbf{x}_1) + \mathrm{Sl}(\mathbf{x}_2) > (\mathbf{x}_1 - \mathbf{x}_2)_i \tag{D.2}$$

We will show both implicational directions of this theorem, by first deriving Equation D.2 from Equation D.1, and then vice versa.

$\implies$

As per Gupta et al's Theorem 2, the size of a polyhedron is of the same order as the size of its canonical bounding box [25], i.e., the Manhattan length of its principle diagonal. The Manhattan

distance of a vector in a dimension d is between 1 and d times its Chebyshev distance; and thus the same order. Letting b be the principle diagonal of the peak's bounding box, this gives us the following:

$$\forall \mathbf{x} \in \mathcal{D} \ : \ \mathrm{Size}(\mathrm{Peak}_{w(\mathbf{x})}) \geq M\|b\|_\infty \tag{D.3}$$

for some positive constant M. Because the bounding box is canonical, its bounds are located at the maximum and minimum values of the points of the peak in each dimension. The Chebyshev distance of b is the greatest difference between the bounds in each dimension, and thus is equal to the maximum Chebyshev distance of any two points in the peak. Therefore,

$$\forall \mathbf{x} \in \mathcal{D}_{\mathrm{U}}, \forall \mathbf{x}_1, \mathbf{x}_2 \in \mathrm{Peak}_{w(\mathbf{x})} \ : \ \mathrm{Size}(\mathrm{Peak}_{w(\mathbf{x})}) \geq M\|\mathbf{x}_1 - \mathbf{x}_2\|_\infty \tag{D.4}$$

And, per our definition of the slack of a polyhedron, and the fact that all points in the peak have the same slack, the following equality holds:

$$\forall \mathbf{x} \in \mathcal{D}, \forall \mathbf{x}_1 \in \mathrm{Peak}_{w(\mathbf{x})} \ : \ \mathrm{Slack}(\mathrm{Peak}_{w(\mathbf{x})}) = \mathrm{Slack}(\mathbf{x}_1) \tag{D.5}$$

Combining the previous two equations as well as Equation D.1, we obtain

$$\forall \mathbf{x} \in \mathcal{D}, \forall \mathbf{x}_1, \mathbf{x}_2 \in \mathrm{Peak}_{w(\mathbf{x})} \ : \ \mathrm{Slack}(\mathbf{x}_1) \geq M\|\mathbf{x}_1 - \mathbf{x}_2\|_\infty \tag{D.6}$$

Because the points in the peak of a slice definitionally have the least amount of slack,

$$\forall \mathbf{x} \in \mathcal{D}, \forall \mathbf{x}_1, \mathbf{x}_2 \in \mathrm{Slice}_{w(\mathbf{x})} \ : \ \mathrm{Slack}(\mathbf{x}_1) \geq M\|\mathbf{x}_1 - \mathbf{x}_2\|_\infty \tag{D.7}$$

Then, obtaining the symmetric equation for $\mathbf{x}_2$ and adding it to the above,

$$\forall \mathbf{x} \in \mathcal{D}, \forall \mathbf{x}_1, \mathbf{x}_2 \in \mathrm{Slice}_{w(\mathbf{x})} \ : \ \mathrm{Slack}(\mathbf{x}_1) + \mathrm{Slack}(\mathbf{x}_2) \geq 2M\|\mathbf{x}_1 - \mathbf{x}_2\|_\infty \tag{D.8}$$

And by the definition of the Chebyshev distance,

$$\forall \mathbf{x} \in \mathcal{D}_{\mathrm{S}}, \forall \mathbf{x}_1, \mathbf{x}_2 \in \mathrm{Slice}_{w(\mathbf{x})} \ : \ \mathrm{Slack}(\mathbf{x}_1) + \mathrm{Slack}(\mathbf{x}_2) \geq 2M \max_{0 \leq i < d_{\mathrm{S}}} (\mathbf{x}_1 - \mathbf{x}_2)_i \tag{D.9}$$

Then, we may obtain a new schedule by multiplying each timestamp function by some integer $c \geq 2M$ to eliminate the constant on the right-hand side.

$$\forall \mathbf{x} \in \mathcal{D}_{\mathrm{S}}, \forall \mathbf{x}_1, \mathbf{x}_2 \in \mathrm{Slice}_{w(\mathbf{x})} \ : \ \mathrm{Slack}(\mathbf{x}_1) + \mathrm{Slack}(\mathbf{x}_2) \geq \max_{0 \leq i < d_{\mathrm{S}}} (\mathbf{x}_1 - \mathbf{x}_2)_i \tag{D.10}$$

We may then separate this out into an inequality for each dimension i:

$$\forall \mathbf{x} \in \mathcal{D}_{\mathrm{S}}, \forall \mathbf{x}_1, \mathbf{x}_2 \in \mathrm{Slice}_{w(\mathbf{x})} \ : \ \mathrm{Slack}(\mathbf{x}_1) + \mathrm{Slack}(\mathbf{x}_2) \geq (\mathbf{x}_1 - \mathbf{x}_2)_i \tag{D.11}$$

The quantifier may then be transformed as follows,

$$\forall \mathbf{x}_1, \mathbf{x}_2 \in \mathcal{D}_{\mathrm{S}} \ : \ w(\mathbf{x}_1) = w(\mathbf{x}_2) \implies \mathrm{Slack}(\mathbf{x}_1) + \mathrm{Slack}(\mathbf{x}_2) \geq (\mathbf{x}_1 - \mathbf{x}_2)_i \tag{D.12}$$

And finally, subtracting the right-hand side of the equation from both sides yields Equation D.2.

Thus, the constraints in Equation 6.4 are implied by the constraint in Equation 6.2, which is in turn implied by the proposition that the given schedule allows the reduction domain to be accumulated without slowdown.

$\Longleftarrow$

From Gupta et al's theorem, we can derive the following, symmetrically of Equation D.3.

$$\forall \mathbf{x} \in \mathcal{D} \ : \ \|b\|_\infty \geq M \cdot \mathrm{Size}(\mathrm{Peak}_{w(\mathbf{x})}) \tag{D.13}$$

And because $||b||_\infty = (\mathbf{x}_1 - \mathbf{x}_2)_i$ for the $\mathbf{x}_1$ and $\mathbf{x}_2$ in the peak with the greatest distance along dimension i,

$$\forall\, \mathbf{x} \in \mathcal{D} \;:\; (\mathbf{x}_1 - \mathbf{x}_2)_i \geq M \cdot \mathrm{Size}(\mathrm{Peak}_{w(x)}) \tag{D.14}$$

Combining this with Equation D.2,

$$\forall\, \mathbf{x} \in \mathcal{D} \;:\; \mathrm{Slack}(\mathbf{x}_1) + \mathrm{Slack}(\mathbf{x}_2) \geq M \cdot \mathrm{Size}(\mathrm{Peak}_{w(\mathbf{x})}) \tag{D.15}$$

Because $\mathbf{x}_1$ and $\mathbf{x}_2$ are in the peak of their slice, we can rewrite this as follows:

$$\forall\, \mathbf{x} \in \mathcal{D} \;:\; 2\,\mathrm{Slack}(\mathrm{Peak}_{w(\mathbf{x})}) \geq M \cdot \mathrm{Size}(\mathrm{Peak}_{w(\mathbf{x})}) \tag{D.16}$$

Finally, we can slow the schedule down by multiplying all the timestamp functions by some integer $c > \frac{M}{2}$ to cancel out the constants on either side, and obtain Equation D.1.

Thus, the constraints in Equation 6.4 implies the constraint in Equation 6.2, which in turn implies that the given schedule allows the reduction domain to be accumulated without slowdown.

As we have also proved the other direction of implication, we can conclude that the constraints in Equation 6.4 are necessary and sufficient for the given schedule to allow accumulation without slowdown.

□

Appendix E

Full Experiment Results

	nussinov	knuth	triang	counting	mccaskill
PLuTo	50.408	53.201	61.757	52.573	354.266
TRACO	22.22	303.936	56.133	50.183	87.292
Non-piecewise AlphaZ	21.299	10.664	23.611	12.557	23.031
Piecewise AlphaZ	2.431	2.363	5.882	2.801	10.169
Speedup over Non-Piecewise	8.8×	4.5×	4.0×	4.5×	2.3×
Speedup over State-of-the-Art	9.1×	22.5×	9.5×	17.9×	8.6×

Figure E.1: Mean run time of code in seconds, on 32 threads, over 10 trials each, for N=10,000.