THESIS

AUTOMATICALLY SIMPLIFYING REDUCTIONS

Submitted by

Ryan Job

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2024

Master's Committee:

Advisor: Sanjay Rajopadhye

Shrideep Pallickara
Christopher Snow

ABSTRACT

AUTOMATICALLY SIMPLIFYING REDUCTIONS

When developing software from a mathematical model, the efficiency of the model and the code which implements it both have significant impacts on the runtime performance of the software. The *reduction simplification* transformation can be used to automatically provide these benefits, improving the runtime performance of programs while simultaneously making it easier to specify a program. This work, which was done in collaboration with Louis Narmour based on a partial implementation by Tomofumi Yuki, tackles the theoretical gaps in this transformation and provides the first complete, automatic implementation of reduction simplification in a compiler. We demonstrate its effectiveness using the real-world problem of RNA secondary structure prediction. Our compiler automatically rediscovers the known optimization for this problem, which required significant human effort to initially discover and implement. In addition, our compiler discovers several previously unknown optimizations for this problem and generates a C implementation of all optimized programs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Many software packages have been developed from mathematical models to help determine or understand the world around us. The models we choose to use and the way we choose to implement them both affect the final performance of our software. Specifying efficient models and implementing them to make good use of modern hardware often requires painstaking effort on behalf of the authors [1–4]. This work required is difficult enough where some authors choose not to rewrite their software to take advantage of better models [5, 6]. To alleviate these issues, software developers often use domain-specific languages or specialized frameworks to abstractly define their computations, then allow compilers to optimize the program that executes them.

The *polyhedral model of computation* is a mathematical framework for analyzing and transforming a domain-specific class of computations. Previous work demonstrates its effectiveness for modeling and optimizing high-level equations. In 2006, Gautam and Rajopadhye [7] described the *reduction simplification* transformation (henceforth referred to as GR06). This transformation rewrites an equational specification of a program such that it can be executed with the minimal asymptotic time complexity (within constant factors). However, several details necessary for an automatic implementation of their transformation were omitted.

1.1 Contributions

This thesis addresses the omissions of the GR06 paper necessary for fully automatic reduction simplification. Most importantly, this includes a way to create equivalence classes of reused values that could be exploited and a heuristic for selecting a specific reuse from each class. Together, these enable the full automation of the Simplifying Reductions transformation.

We have implemented automatic simplification as a standalone tool using the AlphaZ system [8], making it publicly accessible. As part of this, we have developed an open-source implementation for constructing the *face lattice* of an arbitrary convex, parameterized polyhedron (see

Section 5.1) using the integer set library (`isl`) [9]. Additionally, we have improved the memory allocation scheme within AlphaZ to reduce the total memory usage of the automatically generated C code compared to the current bounding box approach.

Finally, we demonstrate the effectiveness of our tool by applying it to RNA secondary structure prediction. In doing so, we have discovered several previously unknown algorithms with the same asymptotic time complexity but differing memory access patterns, constant factors, and lower-order terms.

The contributions of this thesis were developed in collaboration with Louis Narmour and are based on a partial implementation by Tomofumi Yuki.

1.2 Thesis Structure

The remainder of this thesis is structured as follows.

In Chapter 2, we present the RNA secondary structure prediction algorithm as our motivating example. The necessary background information about simplification is covered in Chapter 3. Chapter 4 states the open problems within simplification and how we resolve them. We discuss the implementation of a fully automatic simplification system in Chapter 5, which is empirically evaluated in Chapters 6 and 7. Finally, we discuss related and future work and conclude in Chapters 8, 9, and 10.

Chapter 2

Motivation

Ribonucleic acid (RNA) forms one of the building blocks of life. It is described as a sequence of bases drawn from a 4-letter alphabet. There are several different models for understanding the function or behavior of RNA given its sequence. Of particular importance is the molecule's *secondary structure*, which indicates the bases that are bonded together. Figure 2.1 shows examples of how this is depicted.

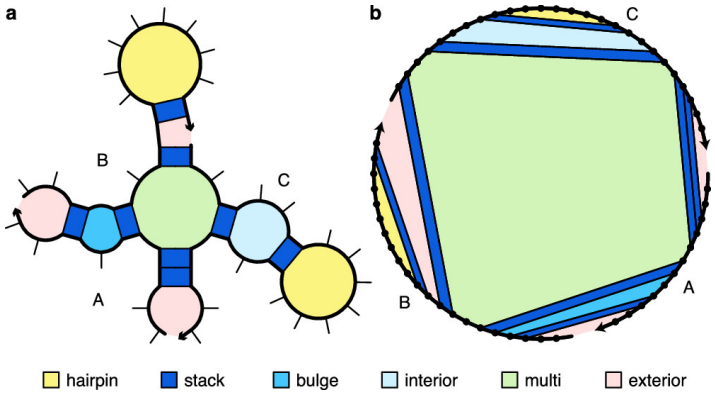

Figure 2.1: Depictions of RNA Secondary Structure [1]

As advancements in next-generation RNA sequencing techniques allows researchers to produce longer reads [10], the need to efficiently compute the secondary structure will continue to grow. One method for predicting this structure uses thermodynamic equations to determine the minimum free energy configuration (i.e., the most likely to occur in nature). Early work in this prediction [11] uses a system of recurrence equations to populate a set of tables through a dynamic programming algorithm. Most of these tables are populated with a time complexity of $O(N^3)$, where N is the number of bases in the RNA strand. However, the one notable exception to this is the table for the energy of an *internal loop* structure, as it is the only one that requires $O(N^4)$ time to populate.

In 1999, Lyngsø et al. [4] demonstrated an invariant property in the internal loop equations. By exploiting this property, they were able to rewrite the equations to eliminate redundant computa-

tions and perform the same computation in only $O(N^3)$ time. These new equations are known as the *fast-i-loops* algorithm. A derivation of this is presented in Chapter 7.

Since its discovery, there have been several implementations of the fast-i-loops algorithm [1–3]. Each implementation required painstaking effort by the authors to first transform the specification into a more usable format and then write the code that implements the algorithm.

These efforts are often very similar, and could be automated using the tools of the polyhedral model. In particular, the GR06 reduction simplification algorithm would allow us to automate the discovery of algorithms like fast-i-loops, and tools such as AlphaZ would enable complex program transformations and code generation.

Chapter 3

Background

We first summarize the Simplifying Reductions transformation as described in GR06 by working through an example. Next, we present an important data structure for implementing automatic simplification known as the *face lattice*. Finally, we discuss several transformations that can be used to enable simplification.

3.1 Simplifying Reductions

Consider (3.1) below, where Y_i refers to the points Y_0 through Y_N inclusive.

$$Y_i = \sum_{j=0}^{i} X_{i-j} \tag{3.1}$$

This equation describes a reduction that combines points in X via addition. A direct implementation of this computation would have a time complexity of $O(N^2)$ (as we will soon see). However, the Simplifying Reductions transformation allows us to rewrite this equation such that its complexity is lowered to $O(N)$.

Per GR06, the asymptotic time complexity for computing a reduction is the number of points in the *reduction body* ($\mathcal{D}$): a polyhedral set of integer points representing the values of program variable indices. We can describe the reduction body of (3.1) via the set (3.2).

$$\mathcal{D} = \{[i, j] \mid 0 \leq j \leq i \leq N\} \tag{3.2}$$

We can visualize this domain per Figure 3.1, where points of X that are read are along the vertical axis, and points written to Y are along the horizontal axis. This set contains $O(N^2)$ integer points, giving us our time complexity.

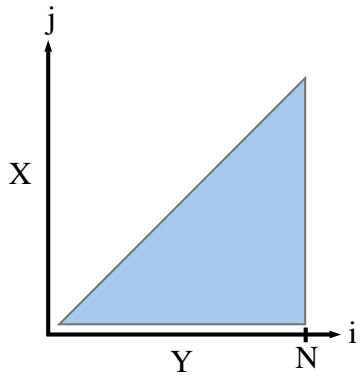

Figure 3.1: Visualization of the domain in (3.2)

The first step of our transformation is to analyze the *reuse space* ($\mathcal{R}$) of the equation, which describes how values (either inputs, results, or intermediate steps) are used (or computed) multiple times throughout the reduction. This is done by computing the kernel of the *dependence function* (f_d), which is an affine map from points in the reduction body to point of the input being read. From (3.1), we have the dependence function (3.3) and its reuse space (3.4)

$$f_d = \{[i, j] \to [i - j]\} \tag{3.3}$$

$$\mathcal{R} = \ker(f_d) = \{[i, j] : j = i\} \tag{3.4}$$

Next, we must select a *reuse vector* (ρ) from the reuse space (i.e., $\rho \in \mathcal{R}$). We will (semi-arbitrarily) select the vector $\rho = (1, 1)$. This indicates that two points (i, j) and $(i + 1, j + 1)$ in the reduction body read the same input value of X. Furthermore, by projecting it onto the answer space (i.e., the horizontal axis), this reuse vector tells us that a previous result Y_i can be used to produce the future result Y_{i+1} (or equivalently, that Y_i is computed from Y_{i-1}). GR06 provides a mathematical definition for the Simplifying Reductions transformation from this information, but we can understand it geometrically as follows (and depicted in Figure 3.2):

1. Translate the reduction body $\mathcal{D}$ (the red triangle) by the reuse vector ρ (the green arrow), producing the domain $\mathcal{D}_s$ (the blue triangle).

2. Remove computations that do not contribute to any answers (the vertical blue line).

3. Rewrite the reduction such that the results in Y depend on previously computed results (the red line) instead of a subset of the values of X.

   (a) This removes the computations at the intersection of $\mathcal{D}$ and $\mathcal{D}_s$ (the intersection of the two triangles).

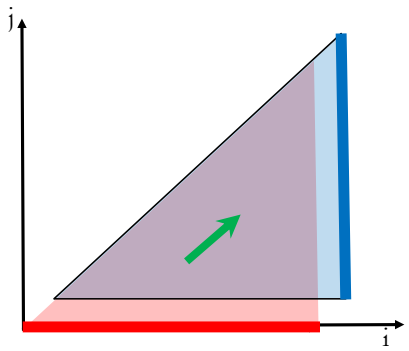

Figure 3.2: Geometric interpretation of the Simplifying Reductions transformation on (3.1).

Applying the Simplifying Reductions transformation to (3.1) with the reuse vector $\rho = (1, 1)$ results in (3.5).

$$Y_i = \begin{cases} X_0 & \text{if } i = 0 \\ Y_{i-1} + X_i & \text{if } i > 0 \end{cases} \tag{3.5}$$

This computation has an $O(N)$ complexity, as can be seen both from its definition and from the red line in Figure 3.2. Furthermore, we may notice that our original equation specifies (in a counter-intuitive way) a prefix scan over the input.

3.1.1 Alternative Simplifications

When applying reduction simplification, there are often multiple valid directions of reuse. Consider if we had instead selected a reuse vector of $(-1, -1)$ instead. Applying reduction simplification to (3.1) with this vector would produce (3.6).

$$Y_i = \begin{cases} \sum_{j=0}^{N} X_j & \text{if } i = N \\ Y_{i+1} - X_i & \text{if } i < N \end{cases} \tag{3.6}$$

This alternative rewrite still has the improved $O(N)$ time complexity, although with different constant factors. Furthermore, the memory access patterns differ slightly.

3.1.2 Recursive Simplification

Consider (3.7), which reduces the input X over two indices, j and k, resulting in an $O(N^3)$ computation. The dependence function used to read X is given in (3.8), and the reuse space for this function is given in (3.9).

$$Y_i = \sum_{0 \leq (j,k)}^{j \leq i \ \wedge \ k \leq i-j} X_k \tag{3.7}$$

$$f_d = \{[i, j, k] \to [k]\} \tag{3.8}$$

$$\mathcal{R} = \ker(f_d) = \{[i, j, k] : k = 0\} \tag{3.9}$$

The reuse space forms a 2D plane embedded in 3D space. If we pick the reuse vector $\rho = (1, 0, 0)$ and apply the simplification transformation, we produce (3.10) and (3.11). Notice that Y is now described as a prefix scan over Z, which itself matches the prefix sum of (3.1) from before.

$$Y_i = \begin{cases} Z_0 & \text{if } i = 0 \\ Z_i + Y_{i-1} & \text{if } i > 0 \end{cases} \tag{3.10}$$

$$Z_i = \sum_{j=0}^{i} X_{i-j} \tag{3.11}$$

While (3.10) now has an $O(N)$ body, (3.11) has an $O(N^2)$ body. Thus, the computation overall is $O(N^2)$. However, as we've seen before, this can be simplified by the reuse vector $\rho' = (1, 1)$,

rewriting Z as a prefix scan over X.

$$Z_i = \begin{cases} X_0 & \text{if } i = 0 \\ X_i + Z_{i-1} & \text{if } i > 0 \end{cases} \tag{3.12}$$

Each application of simplification lowers the asymptotic time complexity for the computation by exactly one polynomial degree [7]. Thus, whenever there are multiple dimensions of reuse that need to be exploited, we must apply simplification on the equations produced by previous applications of simplification. This technique is referred to as *recursive simplification*.

3.2 Face Lattice

The face lattice of a polyhedron [12] is an important data structure for automatic simplification. In this context, a polyhedron is a defined by a set of affine inequalities of the following form:

$$\mathcal{D} = \{x \in \mathbb{Z}^n : Ax + b \geq 0\} \tag{3.13}$$

Each inequality describes a hyperplane bounding the polyhedron. The intersection of such a hyperplane and the polyhedron itself is defined as a *face* of the polyhedron. Each face is a polyhedron of lower dimensionality than the original, with faces that are exactly one fewer dimension referred to as *facets*. That is, the facets of a d-dimensional polyhedron are its $(d-1)$-dimensional faces.

The *face lattice* is used to describe the faces of a polyhedron. Each node of the graph is either the polyhedron being described or one of its faces. An edge is drawn between a pair of nodes if one is a facet of the other. The nodes are arranged into layers based on their dimensionality, with the original polyhedron at the top and each successive layer below having one fewer dimension. We can then view a node's children as being the facets of the polyhedron, and all of the node's descendants as being its faces.

Consider the reduction body (3.2) from Section 3.1, repeated below as (3.14). This is a two-dimensional set defined by the three inequality constraints, c_0 through c_2, listed in (3.15).

$$\mathcal{D} = \{[i, j] \mid 0 \leq j \leq i \leq N\} \tag{3.14}$$

$$c_0 : 0 \leq j \qquad c_1 : j \leq i \qquad c_2 : i \leq N \tag{3.15}$$

The face lattice of a set can be constructed by incrementally *saturating* its inequalities, changing them into equalities. Each node of the face lattice is uniquely labeled by the set of inequalities that it saturates. The face lattice of (3.14) is shown in Figure 3.3. In this graph, the node $\{\}$ refers to the original domain, the node $\{0\}$ saturates constraint c_0, and so on.

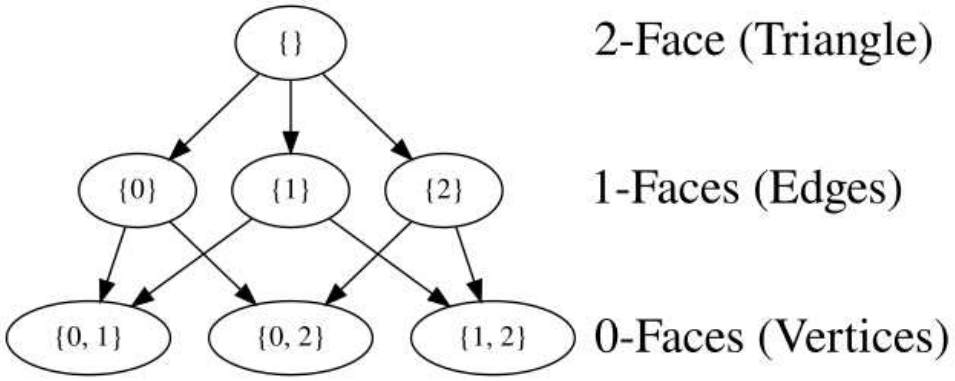

Figure 3.3: Face lattice of (3.14)

When simplification is applied, the remaining computations will all reside along the facets of the reduction body. Thus, it is no coincidence that simplification results in a computation whose time complexity is lowered by one degree. Applying the transformation recursively pushes these computations to facets of facets, resulting in further lowering of the time complexity.

3.3 Simplification Enhancing Transformations

In addition to the simplification transformation, GR06 defines several *simplification enhancing transformations*. These transformations can be applied when expressions within the reduction have reuse, but the reuse cannot be exploited directly.

3.3.1 Idempotent and Higher-Order Operators

Recall from Chapter 3, the accumulation space of a reduction may be multidimensional. Such reductions may contain reuse but are written such that we cannot apply simplification. In this case, we may be able to use *reduction decomposition* to expose the reuse. This is a transformation that rewrites a reduction over multiple variables into a pair of nested reductions. In effect, it introduces a new variable that reduces a subset of points needed for an answer. Each answer is then computed as a reduction over these subsets. By carefully selecting how the nested reductions are written, we can expose the available reuse and allow simplification to take place.

In general, there are infinitely many ways to decompose a reduction. However, our implementation generates only finitely many. These decompositions can serve one of two purposes: exposing partial answers that exhibit reuse among the final answers, and enabling the distributivity property of an expression inside the reduction.

When a reduction operator is applied to the same value multiple times, we can express the entire set of operations with a single value or operation. The approach used changes depending on the properties of the operator.

If our operator is *idempotent*, repeatedly applying the operator to the same value will not change the result. For example, applying the $\min$ operator to the same value repeatedly will not change the value. In this case, we can replace the entire chain of operations with the result. An example of this is shown in (3.16), where we replace repeated minimization of X_i with X_i itself.

$$Y_i = \min_{j=1}^{N} X_i = X_i \tag{3.16}$$

Similarly, if our operator is associated with a *higher-order operator*, we can replace a long chain of our original operator with a single instance of the higher-order operator. For example, in (3.17) below, we can replace the repeated addition of X_i with a single multiplication of X_i times the number of additions.

$$Y_i = \sum_{j=1}^{N} X_i = N \times X_i \tag{3.17}$$

3.3.2 Decomposition for Partial Answers

Consider the example of (3.18), which has a 3D reduction body (over indices i, j, and k) that reduces a 2D input X to produce a 1D answer Y. The reduction body is shown in Figure 3.4, where each triangular region inside the body represents the points reduced to compute a single answer.

$$\begin{gathered} Y_i = \max_{j,k \in \mathcal{D}} X_{j,k} \\ \mathcal{D} = \{[j,k] \mid (i \leq j \leq 2i) \wedge (i \leq k \leq 3i - j)\} \end{gathered} \tag{3.18}$$

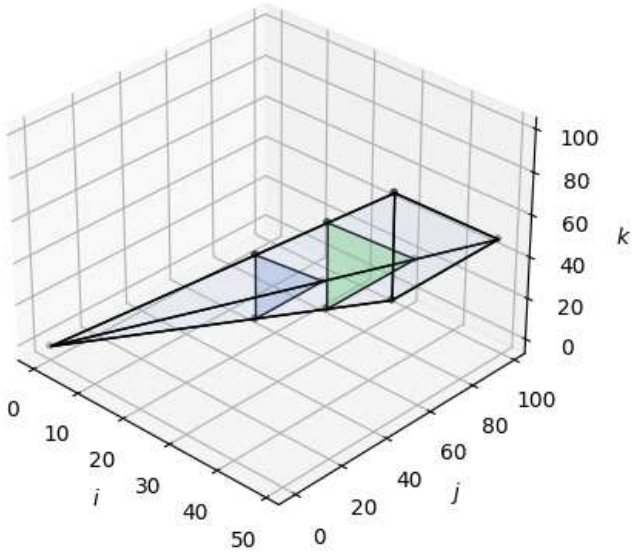

Figure 3.4: Reduction body of (3.18).

While there is reuse along the i dimension (since reading $X_{j,k}$ does not involve i), we cannot exploit it via simplification. Any possible labeling over the faces would require at least one $\ominus$-face,

meaning we would require an inverse to the $\max$ operation. We can see this geometrically: as each successive triangle in the i direction gets larger, it also shifts to the right in the j direction and up in the k direction. To allow simplification, we can use reduction decomposition.

Consider if we divided one of the triangles representing a single answer into line segments. For example, let us make these segments parallel to the back face (i.e., parallel to $j + k = m$ for some constant m). Figure 3.5 shows an example of several line segments where i changes but $j + k$ remains constant. As i decreases, the line segments get larger on each end. This would then allow us to exploit reuse along the i axis (specifically, in the $-i$ direction), meaning we can now apply simplification.

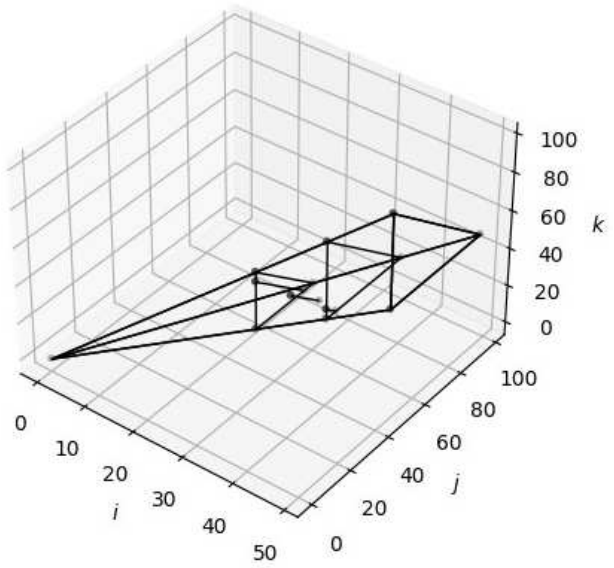

Figure 3.5: Lines parallel to $i + j = m$ within the reduction body of (3.18).

To carry out this decomposition, we first introduce the variable $m = j + k$ and replace all instances of j accordingly. Then, we rewrite the single reduction over both m and k into two nested reductions, with the outer being over m and the inner being over k. The inner reduction is

then moved to a new temporary variable, Z. This results in (3.19).

$$\begin{aligned} Y_i &= \max_{m=2i}^{3i} Z_{i,m} \\ Z_{i,m} &= \max_{k=i}^{m-i} X_{m-k,k} \end{aligned} \tag{3.19}$$

With this rewrite, we can now exploit reuse in our new equation for Z. Each $Z_{i,m}$ can be computed from $Z_{i+1,m}$ and two additional points from our input X. Thus, we can rewrite Z as shown in (3.20).

$$Z_{i,m} = \max\big(Z_{i+1,m},\ X_{m-i,i},\ X_{i,m-i}\big) \tag{3.20}$$

Our original reduction body contained $O(N^3)$ points, meaning a program that implements it would require $O(N^3)$ time to compute. However, our final equation for Y has a reduction body with $O(N^2)$ points, and each of the $O(N^2)$ points in Z can be computed in constant time. Thus, applying reduction decomposition allows us to reduce the overall time complexity from $O(N^3)$ down to $O(N^2)$.

3.3.3 Decomposition for Distributivity

Consider the case where a reduction's body contains a binary operation that distributes over the reduction operator (e.g., multiplication inside a summation). If one of the terms is invariant at all points within the reduction body, it can be factored out. This in turn may expand the reuse space of the reduction, enabling simplification. Additionally, if only a subspace of points is invariant, our tool uses reduction decomposition to isolate this subspace, enabling distributivity.

Equation (3.21) below and its equivalent Alpha specification (Figure 6.3), which perform a summation over the product of two terms, demonstrates such a situation:

$$\begin{aligned} Y_i &= \sum_{\{j,k\}\in\mathcal{D}} (A_{i,j+k} \times B_{k,j}) \\ \mathcal{D} &= \{[j,k] \mid (0 \le j \le i) \wedge (0 \le k \le i)\} \end{aligned} \tag{3.21}$$

When computing a particular Y_i, all points in the reduction where $j + k$ are the same will read the same value of A. With the summation as written, we cannot exploit this. However, we can apply the same reduction decomposition transformation as in Section 6.2 to help us expose the reuse. Performing the change of basis to introduce $m = j + k$ and replace j such that the outer sum is over m results in (3.22):

$$Y_i = \sum_{m=0}^{2i} \left(\sum_{k=\max(0,m-i)}^{\min(i,m)} (A_{i,m} \times B_{m-k,k}) \right) \tag{3.22}$$

Now, notice that $A_{i,m}$ is invariant at all points of the inner summation over k. We can exploit the fact that multiplication distributes over addition to pull this term out, then isolate the inner reduction to its own variable, producing (3.23):

$$Y_i = \sum_{m=0}^{2i} A_{i,m} \times Z_{i,m} \qquad\qquad Z_{i,m} = \sum_{k=\max(0,m-i)}^{\min(i,m)} B_{m-k,k} \tag{3.23}$$

Simplification can be applied to $Z_{i,m}$, allowing each value to be computed in constant time. There are two choices for how to reuse answers: either $Z_{i+1,m}$ or $Z_{i-1,m}$.

Chapter 4

Reuse Vector Selection

For applying the simplification transformation, we provide both the equations to transform and a reuse vector to exploit. This eliminates the redundant computations, leaving only a set of residual computations. GR06 indicates that the set of reuse vectors, which is infinite for reduction bodies with two or more dimensions, may be partitioned into equivalence classes. However, they do not provide a method for constructing these classes or for selecting a reuse vector from the infinite set of vectors in each class. In this chapter, we address how these can be accomplished. The theory for the approaches in this chapter were developed in collaboration with Louis Narmour.

4.1 Constructing the Equivalence Classes

When single-step simplification is applied, the residual computations reside exclusively along the faces of the reduction body. Furthermore, there are only three ways in which these computations may contribute to answers. The type of contribution exhibited is based on the dot product of the face's normal vector (v_i) and the reuse vector (ρ):

1. If $v_i \cdot \rho > 0$, points on the facet contribute with the reduction operator.
2. If $v_i \cdot \rho < 0$, points on the facet contribute with the inverse of the reduction operator.
3. If $v_i \cdot \rho = 0$, points on the facet do not contribute.

If we compute the face lattice of a reduction body, we can label each face according to the applicable case. We call these $\oplus$-faces, $\ominus$-faces, and $\oslash$-faces respectively. Furthermore, the space of all reuse vectors can be partitioned into equivalence classes according to the set of labels induced. For a specific labeling $\mathcal{L}$ of the faces, we can define the set of reuse vectors that induce that labeling, $\mathcal{R}_{\mathcal{L}}$, using the reuse space, $\mathcal{R}$, and the normal vectors of the $\oplus$, $\ominus$, and $\oslash$ faces ($\mathcal{N}_{\oplus}$, $\mathcal{N}_{\ominus}$,

and $\mathcal{N}_\oslash$ respectively).

$$\mathcal{R}_\mathcal{L} = \mathcal{R} \cap \left\{\rho \mid (\rho \cdot \mathcal{N}_\oplus > 0) \wedge (\rho \cdot \mathcal{N}_\ominus < 0) \wedge (\rho \cdot \mathcal{N}_\oslash = 0)\right\} \tag{4.1}$$

4.1.1 Pruning the Search Space

A bounded convex polyhedron with n facets will have a maximum of 3^n labelings. However, we can prune this search space by exploiting the properties of such a polyhedron. Note that these approaches only reduce the number of labelings by constant factors or lower-order terms, so the number of labelings remains exponential in the number of facets.

Any possible labeling must have at least one $\oplus$ face and at least one $\ominus$ face. We can understand this geometrically: if we translate the polyhedron by a non-zero vector, there must be at least one face in the original polyhedron that is not in the shifted polyhedron. Any such face would necessarily be an $\oplus$ face. Likewise, there must be at least one face that is in the shifted polyhedron but not the original, which would be labeled as an $\ominus$ face. Applying this optimization reduces the search space to $n(n-1)3^{n-2}$ labelings.

We can also constrain parallel faces to further reduce the search space by constant factors. Being parallel, the normal vectors will either point in the same direction or opposite directions. If they point in opposite directions, their dot products with the reuse vector will be negations of each other. This means either that both faces are $\oslash$ faces (if the reuse vector is perpendicular to them), or one face is labeled $\oplus$ while the other is labeled $\ominus$. Applying this optimization to a single such pair reduces the search space by a constant factor of 3, and we can repeatedly apply this to all such pairs. Since the simplification is only applied to convex polyhedra, we do not need to consider the case where two parallel faces have normal vectors in the same direction, as it is impossible to construct such a polyhedron without violating convexity, assuming redundant constraints have been removed.

4.2 Selecting a Reuse Vector

Given a labeling $\mathcal{L}$ and the set of reuse vectors $\mathcal{R}_{\mathcal{L}}$ that induce it, we must select a specific vector. Per GR06, when the reduction body has at least two dimensions, each set of reuse vectors will either be empty (indicating there is no reuse available for that labeling) or unbounded. While any vector will lower the asymptotic time complexity by one degree, the constant factors and lower-order terms may not be identical.

Consider again the example (3.1) from Section 3.1, repeated below.

$$Y_i = \sum_{j=0}^{i} X_{i-j} \tag{4.2}$$

We simplified this prefix sum with the reuse vector $(1, 1)$, resulting in (3.5) (copied below).

$$Y_i = \begin{cases} X_0 & \text{if } i = 0 \\ Y_{i-1} + X_i & \text{if } i > 0 \end{cases} \tag{4.3}$$

However, if we instead selected the vector $(2, 2)$, which is in the same equivalence class, we get the following result, which performs two additions for $i > 0$ (compared to one addition previously).

$$Y_i = \begin{cases} X_0 & \text{if } i = 0 \\ X_0 + X_1 & \text{if } i = 1 \\ Y_{i-2} + X_{i-1} + X_i & \text{if } i > 1 \end{cases} \tag{4.4}$$

Determining a reuse vector that minimizes the constant factors and lower order terms remains an open problem. However, we employ a simple heuristic of selecting the smallest integer point closest to the origin. If there are multiple such points, one is picked arbitrarily.

4.3 Example Reuse Vector Selection

Consider the equation (3.1) and its reduction reduction body (3.2) from the example in Section 3.1. These are repeated below as (4.5) and (4.6) respectively.

$$Y_i = \sum_{j=0}^{i} X_{i-j} \tag{4.5}$$

$$\mathcal{D} = \{[i, j] : 0 \leq j \leq i \leq N\} \tag{4.6}$$

The reduction body forms a triangular domain with three 1-dimensional facets (i.e., the sides of the triangle). There are twelve possible labelings, illustrated in Figure 4.1 with the $\oplus$, $\ominus$, and $\oslash$ faces colored blue, red, and black respectively. Note that these do not yet take the reuse space $\mathcal{R}$ into account.

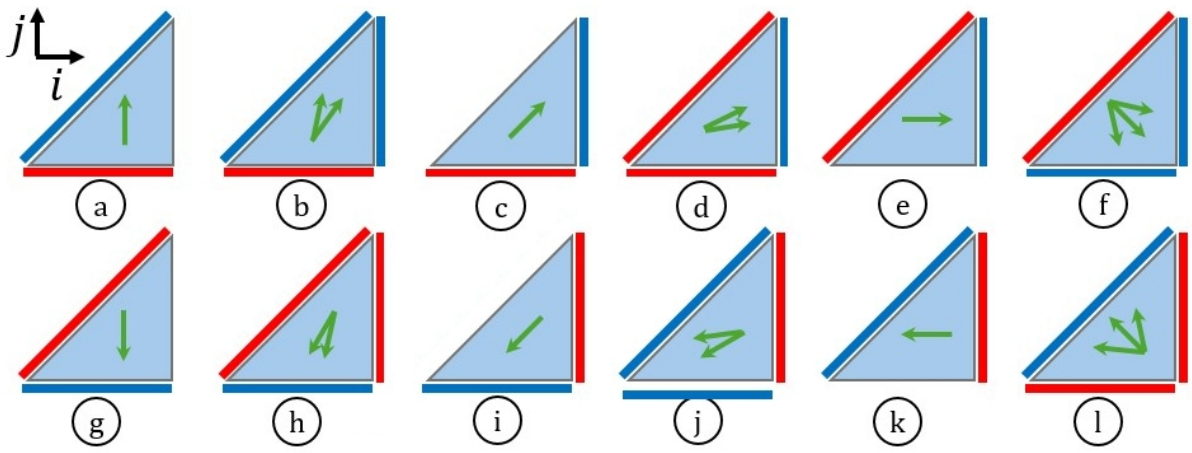

Figure 4.1: Visualization of the equivalence classes of all possible labelings for the domain (4.6)

Each labeling a through l is induced by a cone of possible reuse vectors. We can describe these by their extremal rays, as shown in (4.7). Again, these do not yet take the reuse space into account.

$$
\begin{array}{ll}
\mathcal{L}_a : (0,1) & \mathcal{L}_b : \text{rays between } (1,0) \text{ and } (1,1) \text{ exclusive} \\
\mathcal{L}_c : (1,1) & \mathcal{L}_d : \text{rays between } (1,1) \text{ and } (1,0) \text{ exclusive} \\
\mathcal{L}_e : (1,0) & \mathcal{L}_f : \text{rays between } (1,0) \text{ and } (0,-1) \text{ exclusive} \\
\mathcal{L}_g : (0,-1) & \mathcal{L}_h : \text{rays between } (0,-1) \text{ and } (-1,-1) \text{ exclusive} \\
\mathcal{L}_i : (-1,-1) & \mathcal{L}_j : \text{rays between } (-1,-1) \text{ and } (-1,0) \text{ exclusive} \\
\mathcal{L}_k : (-1,0) & \mathcal{L}_l : \text{rays between } (-1,0) \text{ and } (0,1) \text{ exclusive}
\end{array}
\tag{4.7}
$$

To construct the equivalence classes for the reuse space $\mathcal{R}$ in (3.4), we simply compute $\mathcal{R}_a = \mathcal{R} \cap \mathcal{L}_a$, $\mathcal{R}_b = \mathcal{R} \cap \mathcal{L}_b$, and so on. Nearly all of these are sets are empty, with the exception of $\mathcal{R}_c$ and $\mathcal{R}_i$ remaining unchanged from $\mathcal{L}_c$ and $\mathcal{L}_i$. Thus, we have partitioned our reuse space into two equivalence classes. Picking the smallest integer point in $\mathcal{R}_c$ and $\mathcal{R}_i$ per our heuristic gives us the reuse vectors $(1,1)$ and $(-1,-1)$ respectively.

Chapter 5

Implementation

AlphaZ is a tool for exploring program transformations and optimizations within the polyhedral model [13]. This tool, and the associated Alpha language [14–16], model reductions as polyhedral collections of values combined with an associative and commutative operator to produce collections of values, which is compatible with the simplification transformation. Furthermore, an implementation for a single step of the transformation already exists that we can extend to full simplification. For these reasons, we chose to extend AlphaZ for our implementation. The work here was done in collaboration with Louis Narmour and builds upon previous work by Tomofumi Yuki.

5.1 The Face Lattice

The AlphaZ system utilizes the integer set library (`isl`) [9], which handles the description and manipulation of polyhedral sets and maps. This library is used for implementing the face lattice described in Section 3.2. Our implementation is primarily contained in two classes: `Face` and `FaceLattice`. A UML diagram for these classes is given in Figure 5.1.

The `Face` class records a single node in the face lattice. Each face is constructed from an `isl BasicSet`, which contains the constraints that define a polyhedron as in (3.13).

The constraints of a polyhedron are considered to be either *saturated* or *unsaturated*. Equality constraints are always *saturated*. Any inequality constraint of the form $a^T z + \alpha \geq 0$ is also saturated if there is another constraint of the form $a^T z + \alpha \leq \tau$ for some non-negative integer τ. In other words, pairs of parallel constraints with a constant number of points between them are saturated, such as the constraints $i + j \geq 0$ and $i + j \leq 5$. Any constraint which is not saturated is *unsaturated*.

The saturated and unsaturated constraints are stored separately, with unsaturated constraints being mapped with an ID number as in (3.15). Once a face has been constructed, its dimensionality

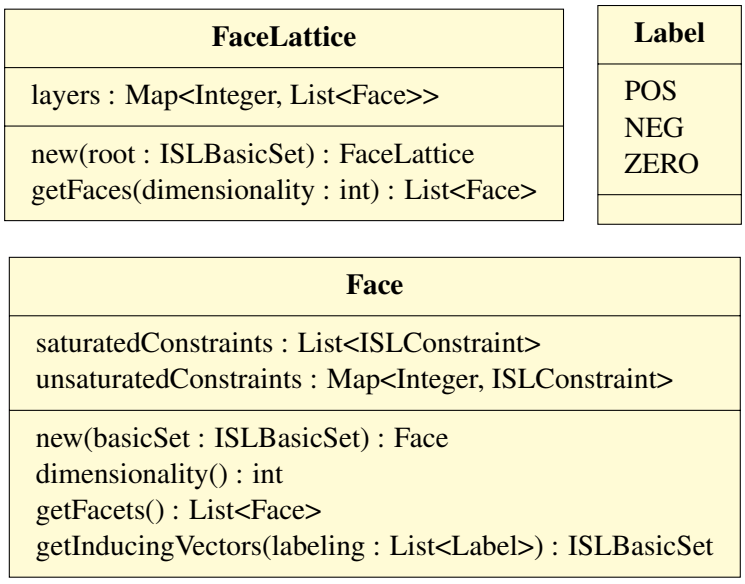

Figure 5.1: UML class diagrams for the face lattice implementation.

is computed as the number of indices the set is defined over minus the number of pairs of saturated constraints, with equality constraints being considered a pair of inequality constraints. This means that the set $\{[i, j] : (0 \leq i < N) \wedge (i \leq j < i + 5)\}$ is considered to be 1-dimensional. We can better understand this by acknowledging that this set contains $5N$ integer points, meaning a set's dimensionality is tied to the number of points it contains instead of how many indices were used.

The facets of a face are computed by saturating one of the unsaturated constraints (i.e., changing it from an inequality to an equality) and checking that the dimensionality has decreased by exactly one. If the dimensionality has decreased by two or more, then at least one additional previously unsaturated constraint has been incidentally saturated. While this is still a face of our original face, it is not a facet, so is ignored.

A full face lattice for a polyhedron is computed first by finding all facets of the polyhedron. As they will have the same dimensionality ($(d-1)$-dimensional if the original polyhedron was d-dimensional), they are grouped together. Then, the facets of these facets are found and grouped together, with identical facets being merged into a single node of the lattice. This continues layer by layer until the 0-dimensional faces (i.e., the vertices of the polyhedron) have been found.

5.2 Optimal Simplifying Reductions

Our implementation of optimal simplification built upon existing transformations in the AlphaZ system. We can think of it as a depth-first search over a space of programs produced by applying one of several *transformation steps*, each of which yield a new program. The kinds of steps that can be applied are:

- Applying single-step simplification with a reuse vector selected per Section 4.2.
- Exploiting an *idempotent* or a *higher-order* operator per GR06.
- *Decomposing* a reduction over two or more introduced indices into a set of two nested reductions.

Our initial program, and all programs generated from the above steps, are always rewritten into a normalized form that ensures all reuse is exposed.

5.2.1 Program Normalization

The AlphaZ system defines a normalized form for Alpha programs [8,15,16]. Our implementation of optimal simplification ensures that the original program and all generated programs follow this normal form. This ensures that reduction bodies are all defined as a single polyhedron, which is necessary for simplification.

Once the program is in a normal form, we apply a transformation called *same operator simplification* [7]. This rewrites reductions containing the reduction operator as a top-level operation, separating into multiple reductions whose results are combined. That is, given a reduction of the form:

$$\text{reduce}(\oplus, f_p, E_1 \oplus E_2)$$

it is rewritten as follows:

$$\text{reduce}(\oplus, f_p, E_1) \oplus \text{reduce}(\oplus, f_p, E_2)$$

The resulting reductions are then rewritten into the normal form. If E_1 and E_2 exhibit different kinds of reuse which are incompatible, separating them into different reductions allows both forms of reuse to be exploited.

After applying the same operator simplification, a final transformation can be applied, called *distributivity*. This can be applied if the expression inside the reduction is an operation that distributes over the reduction operator (e.g., a multiplication inside a summation). If one of the terms in this operation is invariant at all points of the reduction, it can be distributed out. An example of this was presented in Section 3.3.3.

5.2.2 Applying Single-Step Simplification

When considering single-step simplification, our optimal simplification algorithm constructs the equivalence classes for the reuse vectors per Section 4.1. All possible labelings of the facets are enumerated. Labelings which are impossible, or which require an inverse operator when none exists (e.g., reductions over $\min$ or $\max$) are dropped immediately. Then, we construct the set of reuse vectors which induce the labeling. If that set is empty or contains only the zero vector, the labeling is discarded. For the remaining labelings, we select the integer point closest to the origin (per our heuristic from Section 4.2). This forms the set of candidate reuse vectors which we can use to apply single-step simplification.

Once we have our set of candidate reuse vectors, we try to use each one in a depth-first manner. That is, we apply single-step simplification to our program with the first reuse vector, then recursively apply the optimal simplification algorithm to the resulting program. Once the recursive call returns, the process repeats again with the next reuse vector, and so on until all reuse vectors have been tried.

There is a special case of simplification where there is a single subspace of unique answers that all others retrieve. In this case, the optimal simplification will compute the unique answers exactly once, then refer to these answers as needed. To determine if this case occurs for a specific reuse vector, we look at the labeling of the face lattice that it induces. For each $\oplus$ and $\ominus$ face, we find the

linear space of the face as the intersection of the linear parts of each saturated constraint. Then, we compute the intersection of all the linear spaces with the reduction's *accumulation space*, defined as the kernel of the projection function. If this intersection is non-empty, then the reuse vector that induces this labeling causes the special case to occur. Only this simplification is used, and all other candidate reuse vectors are discarded.

5.2.3 Idempotent and Higher-Order Operators

In addition to simplification, the recursive algorithm can try exploiting idempotent and higher-order operators. Detecting if either of these can be applied is done in the same manner. First, we determine the space of identical values that are reduced together. This is the intersection of the reuse space with the reduction's *projection function*, which is a user-supplied mapping from points in the reduction body to the answer they contribute to. We call this intersection the *collapse space*. If this space is empty, we cannot exploit an idempotent or a higher-order operator.

If the collapse space is non-empty, we then generate any function whose kernel is the collapse space. We call this the *collapse function*. This is a many-to-one mapping such that all points in the reduction body are mapped to the same point in the collapse space if and only if they read the same value and contribute to the same answer. A new reduction is then created such that its body is the application of the collapse function to the original reduction body.

If the reduction operator is idempotent, the expression inside the new reduction is the same as in the original reduction. No further work is needed, as the transformed reduction eliminates the unnecessary operations.

If there is a higher-order operator for the reduction operator, we must use it to express the repeated operation. The Barvinok library within `isl` [9, 17] is used to compute the cardinality of the collapse function. This returns a polynomial expression for the number of points in the original reduction body that are mapped to some point in the image of the collapse function (i.e., the number of times the reduction operator was applied to the same value). The higher-order operator is then applied to the original reduction's expression and this polynomial.

5.2.4 Reduction Decomposition

As discussed in Sections 3.3.2 and 3.3.3, optimal simplification may require reduction decomposition. In general, there are infinitely many ways to decompose a reduction. However, our implementation selects finitely many as candidates. All decomposition candidates are tried in a depth-first pattern, as with reuse vectors.

Candidate decompositions that expose reuse among partial answers, per Section 3.3.2, will always be parallel to one of the facets of the reduction body. Any face parallel to the reduction's accumulation space is rejected as a candidate, as this would result in a trivial decomposition where each final answer exactly equals one partial answer.

Decompositions that enable distributivity, per Section 3.3.3, will be parallel to the reuse exhibited by an expression within the reduction. As before, this must not be parallel to the reduction's accumulation space.

5.2.5 Approximate Number of Programs Generated

After applying recursive simplification, the resulting program can be described as computations over a subset of the original reduction body's faces. To estimate the bounds on the total number of programs that could be generated, we can consider the face lattice for the reduction body.

Consider a d-dimensional reduction body defined by n constraints. Since the reduction body must be bounded, we have $n > d$. Any $(d - k)$-dimensional face is constructed by saturating exactly k inequality constraints. In the worst case, there are at most $\binom{n}{k}$ such faces. However, since k is the number of dimensions of reuse, it will often be much smaller than the number of constraints. In this case, there are $O(n^k)$ faces to consider.

When single-step simplification is performed, the residual computations lie exclusively on a strict subset of the reduction body's facets. Recursive simplification further pushes the computation down to facets of facets. Our final program is thus described as a subgraph of the face lattice. In the worst case, recursive simplification must enumerate all subgraphs of the face lattice. For a given value of n, the face lattice will be largest when $d = n - 1$. The total number of nodes in such a

lattice is $\sum_{k=0}^{n-1} \binom{n}{k} = 2^n - 1$, with a total of $O(2^{2^n})$ possible subgraphs. Our simplification engine avoids this potential issue by allowing the user to limit the total number of programs generated.

5.3 Code Generation

The code generator was developed as a two-pass system that first transforms an Alpha AST into an AST for a simplified model of the C language, then prints the C code itself. The grammar for this simplified C AST is shown in Figure 5.2, with minor details omitted for brevity. This model is a proper subset of the C language, containing only the aspects of the language necessary or useful for our generated code.

The two-pass system allows building the C AST out of order and easily combining the sub-trees into a single program. This is unlike previous versions of the AlphaZ system, which would build a string representing the C code file while walking the Alpha AST. Allowing the AST to be assembled in any order simplifies the code generation process, making it easier to add new features or create different code generation schemes.

The generated code is single threaded and uses a naive *demand-driven* execution scheme [8,18]. In this scheme, the programs outputs are computed in lexicographic order. Values are computed as needed and memoized to avoid re-computation. Auxiliary variables are used to detect cyclic data dependencies, terminating the program upon detection.

We have extended the code generator to include a new memory allocation scheme to minimize the total memory used. Using `isl` [9,17], we can generate the C code for a polynomial expression of the number of integer points in a set (i.e., its *cardinality*). This is used both for determining the size of the array and the index of any point within it. Each point within its set is addressed according to its *rank*, or the number of integer points which are lexicographically smaller than that point.

⟨program⟩ ⊨ ⟨comment⟩ ⟨include⟩ ∗ ⟨macro⟩ ∗ ⟨variable decl⟩ ∗ ⟨function⟩ ∗
⟨comment⟩ ⊨ *a (possibly empty) set of strings to print as a comment*
⟨include⟩ ⊨ *an identifier to populate as an* `#include` *directive*
⟨macro⟩ ⊨ *The name, arguments, and replacement for a C-style* `#define` *macro*
⟨variable decl⟩ ⊨ ⟨data type⟩ ⟨ident⟩
⟨function⟩ ⊨ ⟨data type⟩ ⟨ident⟩ ⟨variable decl⟩ ∗ ⟨statement⟩ ∗
⟨data type⟩ ⊨ *a C data type and a series of zero or more* ∗ *characters*

⟨statement⟩ ⊨ ⟨macro⟩ | ⟨variable decl⟩ | ⟨assign stmt⟩ | ⟨return stmt⟩ |
⟨if stmt⟩ | ⟨for loop stmt⟩ | ⟨expression⟩

⟨assign stmt⟩ ⊨ ⟨ident⟩ (= | += | ∗=) ⟨expression⟩
⟨return stmt⟩ ⊨ *a* `return` *statement with optional returned expression*
⟨if stmt⟩ ⊨ ⟨condition branch⟩ + [⟨else branch⟩]
⟨for loop stmt⟩ ⊨ ⟨ident⟩ ⟨expression⟩ ⟨expression⟩ ⟨expression⟩ ⟨statement⟩ ∗
The loop variable, initial value, loop condition, increment, and body

⟨condition branch⟩ ⊨ ⟨expression⟩ ⟨statement⟩ ∗
⟨else branch⟩ ⊨ ⟨statement⟩ ∗

⟨expression⟩ ⊨ ⟨paren expr⟩ | ⟨cast expr⟩ | ⟨array access expr⟩ | ⟨call expr⟩ |
⟨unary expr⟩ | ⟨binary expr⟩ | ⟨ternary expr⟩

⟨paren expr⟩ ⊨ *parentheses surrounding another expression*
⟨cast expr⟩ ⊨ *casts an expression to a data type*
⟨array access expr⟩ ⊨ *an identifier accessed via one or more indexing expressions*
⟨call expr⟩ ⊨ *calls a function with zero or more argument expressions*
⟨unary expr⟩ ⊨ ⟨unary op⟩ ⟨expression⟩
⟨binary expr⟩ ⊨ ⟨expression⟩ ⟨binary op⟩ ⟨expression⟩
⟨ternary expr⟩ ⊨ ⟨expression⟩ `?` ⟨expression⟩ **:** ⟨expression⟩

⟨unary op⟩ ⊨ *one of the C unary operators*
⟨binary op⟩ ⊨ *one of the C binary operators*

Figure 5.2: Representative grammar for the simplified C AST.

Suppose we have some domain $\mathcal{D}$ with indices $i_0, i_1, \ldots, i_n$. We construct the set of points lexicographically smaller than some point $(I_0, I_1, \ldots, I_n)$ as follows:

$$\begin{aligned}\mathcal{D} \cap \{[i_0, i_1, \ldots, i_n] :&(i_0 < I_0) \vee (i_0 = I_0 \wedge i_1 < I_1) \vee (i_0 = I_0 \wedge i_1 = I_1 \wedge i_2 < I_2) \vee \\ &\cdots \vee (i_0 = I_0 \wedge i_1 = I_1 \wedge \cdots \wedge i_{n-1} = I_{n-1} \wedge i_n < I_n)\}\end{aligned} \tag{5.1}$$

By constructing this set and computing its cardinality, we can determine the rank of a point within our domain. A C expression for this polynomial is then generated and used to index the array.

As an example, consider a triangular domain $\mathcal{D}$, defined as follows:

$$\mathcal{D} = \{[i, j] : 0 \leq j \leq i < N\} \tag{5.2}$$

The set of points lexicographically smaller than some point (I, J) is defined as follows:

$$\mathcal{D} \cap \{[i, j] : (i \leq I) \vee (i = I \wedge j < J)\} \tag{5.3}$$

When this domain is given to `isl`, it produces the C expression shown in Figure 5.3. Even though this is a ternary expression and the polynomials are not written according to Horner's method, GCC is able to optimize their evaluation when optimization is turned on. For example, when evaluating points in lexicographic order, GCC identifies that the index increments by one at each call.

```
1 (-1 + I >= 0 && -1 + N - I >= 0 && -1 + J >= 0 && I - J >= 0)
2    ? ((((I + I*I) + 2 * J))/2)
3 : (J == 0 && -1 + I >= 0 && -1 + N - I >= 0)
4    ? (((I + I*I))/2)
5 : 0
```

Figure 5.3: C expression generated by `isl` for the cardinality of (5.3).

Chapter 6

Evaluation

In this chapter, we evaluate the efficacy of our implementation based on the design choices described in Chapters 4 and 5. We confirm that we can find simplified versions of all examples and generate code that produces the correct answers and exhibits the expected reduced asymptotic complexity.

Table 6.1 shows the initial runtime complexity and the expected complexity after simplification for each tested program. We have created a specification for each example and confirmed that our compiler produces the expected simplifications. We provide the original and simplified specifications with their corresponding generated C code in an accompanying artifact.

Table 6.1: Program simplification starting and expected asymptotic runtime complexities

Program	Starting Complexity	Final Complexity
Recursive Simplification	$O(N^3)$	$O(N)$
Reduction Decomposition	$O(N^3)$	$O(N^2)$
Distributivity	$O(N^3)$	$O(N^2)$

6.1 Recursive Simplification

Programs with multiple dimensions of reuse require recursive simplification to produce an optimal specification. To test our implementation of this, we constructed the specification (Figure 6.1) for the example from Section 3.1.2.

The reduction body of this equation contains $O(N^3)$ points, meaning a program implementing it requires $O(N^3)$ time to compute. However, as discussed before, this equation specifies a prefix scan of a prefix scan. This can be performed in linear time by performing a prefix scan over our inputs, saving these results in a temporary variable, then performing a prefix scan over the

```
affine RecursiveSimplification [N]->{:N>2}
inputs
  X: {[k]: 0<=k<N}

outputs
  Y: {[i]: 0<=i<N}

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

Figure 6.1: Alpha code for (3.7)

temporary variable. This can be done as shown previously, and repeated here as (6.1) and (6.2):

$$Y_i = \begin{cases} Z_0 & \text{if } i = 0 \\ Z_i + Y_{i-1} & \text{if } i > 0 \end{cases} \tag{6.1}$$

$$Z_i = \begin{cases} X_0 & \text{if } i = 0 \\ X_i + Z_{i-1} & \text{if } i > 0 \end{cases} \tag{6.2}$$

A single application of simplification is only able to lower the time complexity by one degree, from $O(N^3)$ to $O(N^2)$. It is only through recursive simplification where we can produce a set of equations with an $O(N)$ time complexity. Our compiler being able to produce any correct linear-time programs verifies a correct implementation of this feature.

When given the specification for Equation 3.7, our compiler generates 16 programs with a linear runtime, including the version described by (3.10) and (3.12), confirming that our compiler correctly applies recursive simplification.

6.2 Reduction Decomposition

Decomposing a reduction into partial answers, as shown in Section 3.3.2, may be necessary for exploiting reuse, especially in the absence of an inverse operation. We developed an Alpha

specification for (3.18) and used our compiler to optimize it. This input specification is given in Figure 6.2.

```
affine ReductionDecomposition [N]->{:N>2}
inputs
  X: {[j,k]: 0<=j<=N and 0<=k<=3N}

outputs
  Y: {[i]: 0<=i<=N}

let
  Y[i] = reduce(max, (i,j,k->i),
    {:0<=i<=N and i<=j<=2i and i<=k<=3i-j}: X[j,k]);
.
```

Figure 6.2: Alpha specification for (3.18).

This reduction has $O(N^3)$ points in the body and produces $O(N)$ answers. While there is reuse present (along the i direction), any attempt to exploit it directly requires an inverse to the $\max$ operator, which does not exist. However, as shown previously, we can use reduction decomposition first to subdivide each answer into a set of partial answers, then simplify the reduction that produces the partial answers. As shown previously, this can be done as shown in (6.3) and (6.4).

$$Y_i = \max_{m=2i}^{3i} Z_{i,m} \tag{6.3}$$

$$Z_{i,m} = \max\left(Z_{i+1,m},\ X_{m-i,i},\ X_{i,m-i}\right) \tag{6.4}$$

Our compiler was able to produce the improved $O(N^2)$ program automatically, confirming its ability to automatically apply reduction decomposition and recursively apply simplification to the result.

6.3 Distributivity

Decomposing a reduction such that a term can be distributed outside the inner reduction, as shown in Section 3.3.3, may also be needed. An Alpha specification for (3.21) was developed and given to our compiler to optimize. This specification is given as Figure 6.3.

```
1  affine Distributivity [N]->{:N>2}
2  inputs
3    A: {[i,j]: 0<=i<N and 0<=j<2N}
4    B: {[k,j]: 0<=k,j<N}
5
6  outputs
7    Y: {[i]: 0<=i<N}
8
9  let
10   Y[i] = reduce(+, (i,j,k->i),
11     {:0<=i<N and 0<=j,k<=i }: A[i,j+k] * B[k,j]);
12 .
```

Figure 6.3: Alpha specification for (3.21).

We can create partial answers for Y where each partial answer groups all points where $j + k$ is a constant (referred to as m). The A term would then be invariant for any partial answer, so we can distribute it outside the reduction, which can then be simplified. This results in a computation per (6.5) and (6.6).

$$Y_i = \sum_{m=0}^{2i} (A_{i,m} \times Z_{i,m}) \tag{6.5}$$

$$Z_{i,m} = \sum_{\max(0,m-i)}^{\min(i,m)} (B_{m-k,k}) \tag{6.6}$$

Our compiler is able to produce this $O(N^2)$ program automatically. Thus, we can confirm that our compiler can correctly use decomposition to expose and exploit distributivity, then simplify the result.

Chapter 7

Simplifying the RNA Energy Equations

In this chapter, we will evaluate our compiler's ability to automatically reproduce the same *fast-i-loops* algorithm that Lyngsø et al. discovered. To do this, we will first show how the improved algorithm can be derived by hand, following the notation of Jacob et al. [3]. Then, we will briefly discuss how our compiler systematically produces this algorithm, plus three alternative algorithms produced from the same initial specification. Finally, we will empirically confirm that the simplified programs produce the correct results and that they exhibit the expected asymptotic runtime characteristics.

7.1 Reproducing the Fast-i-Loops Algorithm

In the original RNA secondary structure prediction algorithm [11], computing the energy of an interior loop structure causes the algorithm to have an $O(N^4)$ runtime. Equation (7.1) below is a representative example for the form of this equation.

$$Y_{i,j} = \min_{i<p<q<j} \left(A_{p,q} + B_{p-i+j-q} + C_{|p-i-j+q|} \right) \tag{7.1}$$

First, we can decompose this reduction by introducing $k = p-i+j-q$ (the indexing expression for B), replacing p accordingly. We let the outer minimization be over k and the inner be over q.

$$Y_{i,j} = \min_{2\leq k<j-i} \left(\min_{j-k<q<j} \left(A_{i-j+k+q,q} + B_k + C_{|k-2j+2q|} \right) \right) \tag{7.2}$$

Notice that the B_k term is now invariant within the inner minimization and thus can be factored out. We then isolate the inner minimization to its own variable, Z.

$$Y_{i,j} = \min_{2 \le k < j-i} \left(B_k + Z_{i,j,k} \right) \tag{7.3}$$

$$Z_{i,j,k} = \min_{j-k < q < j} \left(A_{i-j+k+q,q} + C_{|k-2j+2q|} \right) \tag{7.4}$$

Finally, we can notice that $Z_{i+1,j-1,k-2}$ minimizes a subset of the terms that $Z_{i,j,k}$ does. We can then apply simplification to rewrite $Z_{i,j,k}$ as the minimum of $Z_{i+1,j-1,k-2}$ and a constant number of additional points in the reduction body. This gives us our desired $O(N^3)$ algorithm, repeated below.

$$Z_{i,j,k} = \min \begin{pmatrix} A_{i+1,j-k+1} + C_{|-k+2|} \\ A_{i+k-1,j-1} + C_{|k-2|} \\ Z_{i+1,j-1,k-2} \end{pmatrix} \tag{7.5}$$

7.2 Newly Discovered Algorithms

We developed a specification for the full $O(N^4)$ fast-i-loops algorithm, matching the equation format and variable names presented by Jacob et al. [3]. This is presented as Figure 7.1, with the relationship between the specification and (7.1) presented in Table 7.1. When given to our compiler, it was able to automatically discover this same algorithm that Lyngsø et al. described, plus three alternative algorithms that were previously unknown.

Table 7.1: Correlation between representative and actual RNA secondary structure terms.

Representative Equation Term	Alpha Specification Term
$Y_{i,j}$	`VBI[i,j]`
$A_{p,q}$	`V[p,q] + stack[p,q]`
$B_{p-i+j-q}$	`size[p-i+j-q-2]`
$C_{\|p-i-j+q\|}$	`asym_abs[p-i-j+q]`

One of the newly discovered algorithms performs a different decomposition using the expression used to index C instead of B. That is, it introduced $l = p - i + q - j$. Since this value may

```
affine fast_i_loops [N]->{:N>=10}
inputs
  asymmetry, size: {[i]: 0<=i<=N}
  stack: {[i,j]: 1<=i,j<=N}

outputs
  VBI: {[i,j]: 1<=i and j<=N and j-i>=3}

locals
  asym_abs: {[i]: -N<=i<=N}
  ebi: {[i,j,i1,j1]: 1<=i<i1<j1<j<=N}

let
  VBI[i,j] = reduce(min, (i,j,p,q->i,j), V[p,q] + ebi[i,j,p,q]);

  ebi[i,j,p,q] = stack[i,j] + stack[p,q] +
    size[p-i+j-q-2] + asym_abs[p-i-j+q];

  asym_abs[i] = case {
    {:i>=0}: asymmetry[i];
    {:i<0}: asymmetry[-i];
  };
.
```

Figure 7.1: Alpha specification for the $O(N^4)$ RNA secondary structure prediction algorithm.

be negative (as $q < j$), and since its absolute value is used to access C, different answers must be reused if l is positive or negative. Equations 7.6 and 7.7 below are representative of the algorithm produced by our compiler. We refer to this algorithm as the "New Algorithm".

$$Y_{i,j} = \min_{i-j+3 \le l \le j-i-3} \left(C_{|l|} + Z'_{i,j,l} \right) \tag{7.6}$$

$$Z'_{i,j,l} = \begin{cases} \min \begin{pmatrix} Z'_{i-1,j-1,l+2} \\ A_{i+l+1,j-1} + B_{l+2} \end{pmatrix} & \text{if } l \ge 0 \\ \min \begin{pmatrix} Z'_{i+1,j+1,l-2} \\ A_{i+1,j+l-1} + B_{-l+2} \end{pmatrix} & \text{if } l < 0 \end{cases} \tag{7.7}$$

The final two algorithms are a hybrid approach. In short, they split the minimization of $Y_{i,j}$ into two cases early on, based on the positive and negative values of $l = p - i + q - j$. One of the splits use the Lyngsø simplification described in Section 7.1, while the other split uses the alternative simplification of Equations 7.6 and 7.7. These two algorithms are referred to as "Hybrid 1" and "Hybrid 2".

Our compiler produces polynomials for the total number of loop iterations performed by each program as a metric for estimating the performance of each algorithm. These are listed in Table 7.2.

Table 7.2: Polynomials for the number of loop iterations in each fast-i-loop algorithm.

Program	Loop Iterations
Original Algorithm	$\frac{1}{24}N^4 + \frac{1}{12}N^3 - \frac{1}{24}N^2 + \frac{35}{12}N - 1$
Lyngsø Algorithm	$N^3 - 5N^2 + 17N - 30$
New Algorithm	$\frac{4}{3}N^3 - 8N^2 + \frac{62}{3}N - 14$
Hybrid 1	$\frac{13}{12}N^3 - \frac{43}{8}N^2 + \frac{271}{24}N + \frac{1}{4}\lfloor\frac{N}{2}\rfloor - 3$
Hybrid 2	$\frac{5}{4}N^3 - \frac{61}{8}N^2 + \frac{211}{8}N - \frac{1}{4}\lfloor\frac{N}{2}\rfloor - 41$

7.3 Empirical Verification of Expected Complexities

The original $O(N^4)$ algorithm and all four $O(N^3)$ algorithms were used to generate single threaded, naive, demand-driven C code [18]. All of the programs were compiled by GCC using the Makefile provided by the code generator. We tested the produced executables to empirically verify:

- The simplified versions produce the same results as the $O(N^4)$ original program.
- The asymptotic time complexity of each program is as expected.

Correctness of the results was determined by using the program for the original specification as the ground truth. Random inputs were generated and given to both the original and simplified programs. The outputs between the programs were then checked for equality (within floating point

precision). All of the simplified programs were found to produce the correct results for a variety of problem sizes over multiple executions with different random inputs.

All programs were executed on a Linux system equipped with an Intel Core i7-12700K CPU. The amount of memory used during each execution was confirmed to be less than the amount of memory available via the Linux `time` utility[1]. This ensured that memory paging did not affect performance since all program data was small enough to fit in DRAM.

Figure 7.2 shows a plot for the algorithm runtimes across problem sizes from $N = 100$ to $N = 3000$ in increments of 100. This plot uses logarithmic scaling for both axes to show polynomials as straight lines with a slope relating to the polynomial degree. Functions for $O(N^3)$, and $O(N^4)$ are plotted alongside the results to visually confirm that the measured complexities match what we theoretically expected. That is, the original specification runs in $O(N^4)$ time, while the simplified versions run in $O(N^3)$ time.

Based on the polynomials for the number of loop iterations per program (see Table 7.2), we expect the Lyngsø-equivalent program to have the best performance, followed closely by the "Hybrid 1" program. The "Hybrid 2" and "New Algorithm" programs are expected to be about 25% and 33% slower respectively. To evaluate the accuracy of this, we computed the relative runtime of each program as the ratio of the measured runtime and the runtime of the fastest program for the same problem size. These relative runtimes were then averaged and are presented in Figure 7.3 alongside the ratios of the leading coefficients from Table 7.2.

The greatest error in this estimation was for the "Hybrid 2" algorithm, where our measured performance differed from the expected performance by 7.72%. This shows that, for the tested programs, the execution time is an acceptable proxy for the total number of operations performed.

[1]https://www.man7.org/linux/man-pages/man1/time.1.html

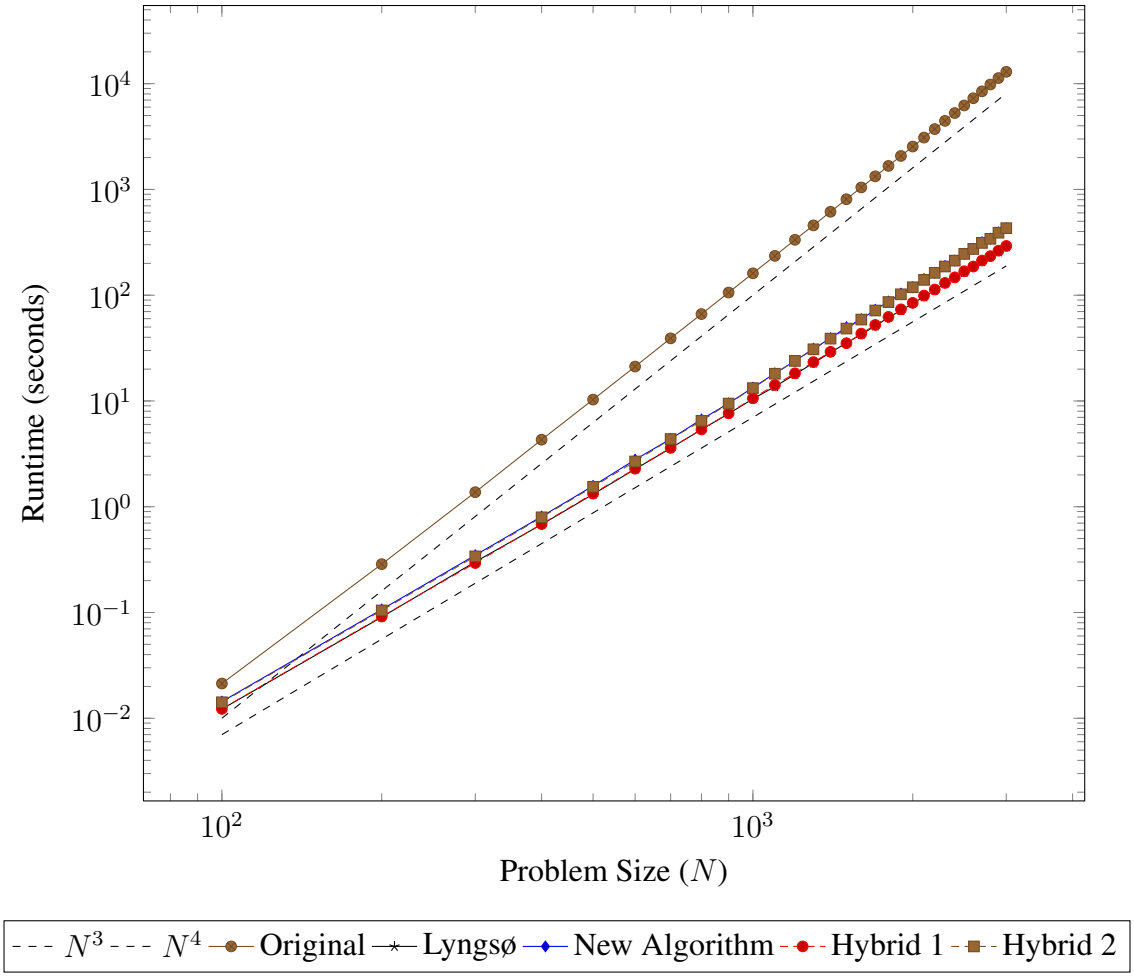

Figure 7.2: Average runtimes of all generated fast-i-loops programs.

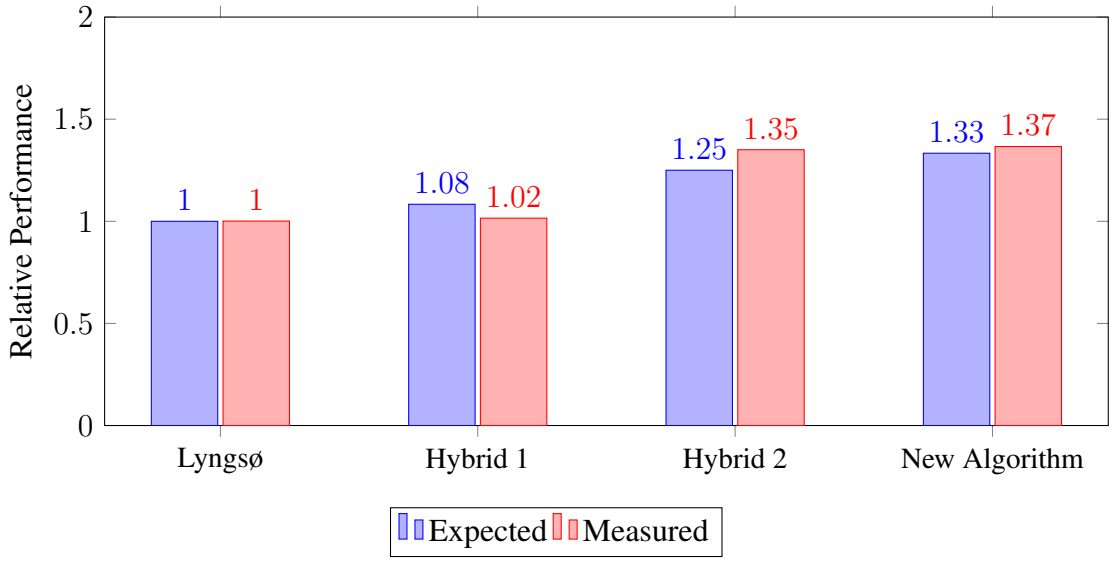

Figure 7.3: Relative performance of the simplified fast-i-loops programs.

Chapter 8

Related Work

Simplification has garnered renewed interest recently. Asymptotic inefficiencies are present, even in deployed codes. Ding and Shen [19] noted that nine of the 30 benchmarks in Polybench 3.0, and two deployed PDE solvers have such inefficiencies, although subsequent releases of Polylib have addressed this. Separately, Yang, et al. [20] showed that simplification is useful for many algorithms in statistical learning like Gibbs Sampling (GS), Metropolis Hasting (MH) and Likelihood Weighting (LW). Their bencharks include Gaussian Mixture Models (GMM), Latent Dirichlet Allocation (LDA) and Dirichlet Multinomial Mixtures (DMM). See their paper for details of benchmarks, algorithms, size parameters, machine specs, etc.

There is ongoing research into the RNA secondary structure algorithms. The optimizations developed by Lyngsø et al. have been implemented by a variety of others [1–3], but not all applications incorporate this algorithmic improvement [5, 6]. Separately, work is being done to schedule and optimize these and similar equations within the polyhedral model [21, 22]. However, there still exist other equations in this domain that do not have known optimized versions but could be optimized by our compiler, such as the Maximum Expected Accuracy (MEA) equations [23, 24].

Simplification is closely related to tensor contractions, a problem that arises in many domains. They can be computed optimally using Pearl's "summary passing" or "message passing" algorithm [25] for Bayesian inference, or the generalized distributive law (GDL) due to Aji and McEliece [26]. GDL has been discovered in many different contexts and is equivalent to the sum-product algorithm proposed by Kschischang et al. [27] and its general framework described by Shafer and Shenoy [28]. We first explain the technique through a simple example drawn from Aji and McEliece [26] and then describe how it complements reduction simplification.

$$X_{i,l} = \sum_{j,k=1}^{N} A_{i,j,l} \times B_{i,k} \qquad\qquad Y_j = \sum_{i,k,j=1}^{N} A_{i,j,l} \times B_{i,k} \tag{8.1}$$

Naively, each equation would have $O(N^4)$ complexity since each result is the accumulation of a triple summation, and there are a $O(N)$ answers. However, if we define two new variables, $T_{i,l} = \sum_{j=1}^{N} A_{i,j,l}$ and $T'_i = \sum_{k=1}^{N} B_{i,k}$, the following is an equivalent simplified system of equations, and the complexity is only $O(N^3)$.

$$X_{i,l} = T_{i,l} \times T'_i \qquad\qquad Y_j = \sum_{i,l=1}^{N} A_{i,j,l} \times T'_i \tag{8.2}$$

The GDL algorithm efficiently evaluates such systems of equations, in particular, those that formulate a problem called the *marginalization of product functions* (the MPF problem), which is nothing but a set finite number of tensor contractions, using a set of common input tensors. A large number of problems in diverse domains can be formulated as MPFs and therefore the GDL serves a unifying, generic algorithm leading to cross-disciplinary insight. GDL consists of building what is called a junction tree or forest from the sets of labels representing the input and output variables. The construction of this forest is guaranteed to succeed, possibly at the expense of additional auxiliary variables, and the computation of the functions is essentially a bottom-up traversal of the forest (possibly followed by a top-down one for multiple outputs).

The GDL and related algorithms have had a big impact in the design and re-emergence of low-density parity checking codes, and have provided deep insight into the links between belief propagation in artificial intelligence and coding theory. In certain instances, new algorithms have been discovered using this framework [29, 30]. Nevertheless, GDL has usually been applied by hand: we are not aware of tools that implement it automatically. A typical scientist seeking to use these methods tends to (i) write the equations; (ii) study their algebraic and structural properties; (iii) build the underlying graph structure (in the case of GDL); (iv) develop insight into the intermediate equations to define; (v) write down the optimized equations; and (vi) eventually produce an executable program that embodies the new algorithm.

MPF problems, or tensor contractions, are a subset of the class of the equations we handle via simplification. Typically the algorithms that optimize them handle multiple contractions, all

involving the same set of input tensors. The complementarity of the polyhedral model and GDL is highlighted by the following observations:

In tensor contractions, the reuse space is only along a subset of the canonic vectors: no "oblique" reuse is allowed. This is because the dependences on the right-hand side are only a *permutation of a subset* of the indices. Note that in some instances (e.g., belief propagation) it may not even make sense to formulate an equation with oblique reuse. Furthermore, the problem where some oblique reuse is present has recently received some attention [31], but the optimizations only exploit distributivity and not reduction simplification.

Similarly, the accumulation space is along a subset of the canonic vectors, and no "oblique" accumulations are allowed. Again, this may not make sense in some domains. Moreover, most *initial* specifications for the reduction simplification method also do not have oblique accumulations. However, some equations require an oblique decomposition of the accumulation space to optimally expose/exploit the reuse, as we saw in the Lyngsø algorithm. Because of this, the reuse and accumulation spaces can be represented as a subset of index names, rather than arbitrary subspaces of the index space (note that the power set of index names is finite, albeit combinatorially large, but there are infinitely many vector subspaces). There is a distinct subset for each subexpression in the product expression.

The domains of the GDL accumulation expressions are *hyper-rectangular parallelepipeds*. Again, no "oblique" boundaries are allowed. Hence, scans cannot be expressed and/or detected in the MPF formulation.

Finally, in many tensor contractions, the complexity is measured in terms of the *number* of subexpressions, and the bounds of the summations are assumed to be (small) constants, leading to "exponential" complexity. Reduction simplification on the other hand, comes from the domain of loops and compiler optimization, so the accumulation bounds are program size parameters, and the number of subexpressions are small constants, and the program complexity is polynomial in the parameters.

Chapter 9

Future Work

Our compiler works only on *independent* reductions, i.e., those where the expressions/values over which the reduction operator is applied do not use *any instance* of the result variable. However, Yang et al. [20] show that dealing with *dependent* reductions, where some outputs may depend recursively on computed results, is an important problem in practice. Our compiler currently may produce equations that do not admit a legal schedule. This can, of course, be detected by running a scheduler on the simplified program. We are working on integrating scheduling with simplification in the standard recursion down the face lattice.

Consistent with GR06, we only handle programs with a single size parameter. Extending the theory to multiple size parameters is important for many algorithms, e.g., in RNA computations, we may have two sequences of respective lengths, N and M. As noted by Loechner and Wilde [12], such domains may be decomposed into *chambers*, each with a distinct face lattice, hence polyhedral tools are capable of handling multiple parameters. However, the function describing complexity is still a multivariate polynomial involving two non-comparable parameters, N and M. This introduces a partial order among the complexity functions and requires extending the core simplification algorithm.

Currently, our compiler only generates single-threaded C code with our demand-driven execution scheme. While this is sufficient for confirming the validity of the generated programs, it is in no way optimal. Directly supporting scheduled and parallelized code would be beneficial to the AlphaZ system [32–35].

Chapter 10

Conclusions

Nearly two decades after the theory was first proposed by Gautam and Rajopadhye [7], we implemented the reduction simplification transformation, a powerful program transformation capable of automatically exploiting reuse in programs involving reductions. The original theory omitted several key components required for employing simplification in practice, which we discussed and addressed, thus providing the first complete push-button implementation of simplification in a compiler.

We evaluated its effectiveness in automatically rediscovering several key results in algorithmic improvement previously only attainable through manual human analysis. In particular, we illustrated how simplification discovered three new cubic algorithms for RNA secondary structure prediction. Our work takes a step toward raising the level of abstraction for the user and demonstrates how what used to require clever, painstaking analysis can be systematically employed as a sequence of program transformations in a compiler.

Bibliography

[1] Mark E. Fornace, Nicholas J. Porubsky, and Niles A. Pierce. A Unified Dynamic Programming Framework for the Analysis of Interacting Nucleic Acid Strands: Enhanced Models, Scalability, and Speed. *ACS Synthetic Biology*, 9(10):2665–2678, October 2020. Publisher: American Chemical Society.

[2] Robert M. Dirks and Niles A. Pierce. A partition function algorithm for nucleic acid secondary structure including pseudoknots. *Journal of Computational Chemistry*, 24(13):1664–1677, 2003. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/jcc.10296.

[3] Arpith C. Jacob, Jeremy D. Buhler, and Roger D. Chamberlain. Rapid RNA Folding: Analysis and Acceleration of the Zuker Recurrence. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 87–94, Charlotte, NC, USA, 2010. IEEE.

[4] R B Lyngsø, M Zuker, and C N Pedersen. Fast evaluation of internal loops in RNA secondary structure prediction. *Bioinformatics*, 15(6):440–445, June 1999.

[5] Ronny Lorenz, Stephan H. Bernhart, Christian Höner zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F. Stadler, and Ivo L. Hofacker. ViennaRNA Package 2.0. *Algorithms for Molecular Biology*, 6(1):26, November 2011.

[6] David H. Mathews. Using an RNA secondary structure partition function to determine confidence in base pairs predicted by free energy minimization. *RNA*, 10(8):1178–1190, August 2004. Company: Cold Spring Harbor Laboratory Press Distributor: Cold Spring Harbor Laboratory Press Institution: Cold Spring Harbor Laboratory Press Label: Cold Spring Harbor Laboratory Press Publisher: Cold Spring Harbor Lab.

[7] Gautam and S. Rajopadhye. Simplifying reductions. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 30–41, New York, NY, USA, January 2006. Association for Computing Machinery.

[8] Tomofumi Yuki, Vamshi Basupalli, Gautam Gupta, Guillaume Iooss, DaeGon Kim, Tanveer Pathan, Pradeep Srinivasa, Yun Zou, and Sanjay Rajopadhye. Alphaz: A system for analysis, transformation, and code generation in the polyhedral equational model. Technical Report CS12-101, Colorado State University, May 2012.

[9] Sven Verdoolaege. isl: An Integer Set Library for the Polyhedral Model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software – ICMS 2010*, Lecture Notes in Computer Science, pages 299–302, Berlin, Heidelberg, 2010. Springer.

[10] Kimberly R. Kukurba and Stephen B. Montgomery. RNA Sequencing and Analysis. *Cold Spring Harbor protocols*, 2015(11):951–969, April 2015.

[11] M Zuker and P Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Research*, 9(1):133–148, January 1981.

[12] Vincent Loechner and Doran K. Wilde. Parameterized Polyhedra and Their Vertices. *International Journal of Parallel Programming*, 25(6):525–549, December 1997.

[13] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. AlphaZ: A System for Design Space Exploration in the Polyhedral Model. In Hironori Kasahara and Keiji Kimura, editors, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 17–31, Berlin, Heidelberg, 2013. Springer.

[14] F. de Dinechin, P. Quinton, and T. Risset. Structuration of the ALPHA language. In *Programming Models for Massively Parallel Computers*, pages 18–24, October 1995.

[15] H. Le Verge. Reduction operators in Alpha. In Daniel Etiemble and Jean-Claude Syre, editors, *PARLE '92 Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science, pages 397–411, Berlin, Heidelberg, 1992. Springer.

[16] Christophe Mauras. *Alpha : un langage equationnel pour la conception et la programmation d'architectures paralleles synchrones*. These de doctorat, Rennes 1, January 1989.

[17] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions. *Algorithmica*, 48(1):37–66, May 2007.

[18] D. Wilde and S. Rajopadhye. The naive execution of affine recurrence equations. In *Proceedings The International Conference on Application Specific Array Processors*, pages 1–12, Strasbourg, France, July 1995. IEEE. ISSN: 1063-6862.

[19] Yufei Ding and Xipeng Shen. GLORE: generalized loop redundancy elimination upon LER-notation. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):74:1–74:28, October 2017.

[20] Cambridge Yang, Eric Atkinson, and Michael Carbin. Simplifying dependent reductions in the polyhedral model. *Proceedings of the ACM on Programming Languages*, 5(POPL):20:1–20:33, January 2021.

[21] Marek Palkowski and Wlodzimierz Bielecki. Parallel tiled Nussinov RNA folding loop nest generated using both dependence graph transitive closure and loop skewing. *BMC Bioinformatics*, 18(1):290, June 2017.

[22] David Wonnacott, Tian Jin, and Allison Lake. Automatic tiling of "mostly-tileable" loop nests. In *5th International Workshop on Polyhedral Compilation Techniques*, Amsterdam, 2015.

[23] Zhi John Lu, Jason W. Gloor, and David H. Mathews. Improved RNA secondary structure prediction by maximizing expected pair accuracy. *RNA*, 15(10):1805–1813, October 2009.

Company: Cold Spring Harbor Laboratory Press Distributor: Cold Spring Harbor Laboratory Press Institution: Cold Spring Harbor Laboratory Press Label: Cold Spring Harbor Laboratory Press Publisher: Cold Spring Harbor Lab.

[24] Marek Palkowski and Wlodzimierz Bielecki. Parallel tiled cache and energy efficient codes for O(n4) RNA folding algorithms. *Journal of Parallel and Distributed Computing*, 137:252–258, 2020.

[25] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, San Mateo, CA, 1988.

[26] S.M. Aji and R.J. McEliece. The generalized distributive law. *IEEE Transactions on Information Theory*, 46(2):325–343, March 2000. Conference Name: IEEE Transactions on Information Theory.

[27] F.R. Kschischang, B.J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, February 2001.

[28] Glenn R. Shafer and Prakash P. Shenoy. Probability propagation. *Annals of Mathematics and Artificial Intelligence*, 2(1):327–351, March 1990.

[29] R.J. Drost and A.C. Singer. Image segmentation using factor graphs. In *IEEE Workshop on Statistical Signal Processing, 2003*, pages 150–153, September 2003.

[30] Robert J. Drost and Andrew C. Singer. Factor-Graph Algorithms for Equalization. *IEEE Transactions on Signal Processing*, 55(5):2052–2065, May 2007.

[31] P. Pakzad and V. Anantharam. A new look at the generalized distributive law. *IEEE Transactions on Information Theory*, 50(6):1132–1155, June 2004.

[32] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI '08, pages 101–113, New York, NY, USA, June 2008. Association for Computing Machinery.

[33] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[34] DaeGon Kim and Sanjay Rajopadhye. Efficient Tiled Loop Generation: D-Tiling. In Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li, editors, *Languages and Compilers for Parallel Computing*, pages 293–307, Berlin, Heidelberg, 2010. Springer.

[35] Yun Zou and Sanjay Rajopadhye. Scan detection and parallelization in "inherently sequential" nested loop programs. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 74–83, New York, NY, USA, March 2012. Association for Computing Machinery.