THESIS

SIMPLIFYING DEPENDENT REDUCTIONS USING DUALITY

Submitted by

Jonathon Yallop

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2025

Master's Committee:

Advisor: Sanjay Rajopadhye

Ravi Mangal
James Wilson

ABSTRACT

SIMPLIFYING DEPENDENT REDUCTIONS USING DUALITY

Programs often contain redundant computations, and frequently the onus is on the developer to detect these repeated computations and remove them. As an alternative to hand optimization, compilers can often be used to automatically optimize program inefficiencies. In this work, we discuss a compiler optimization that can automatically detect and remove certain types of redundant computation. Specifically, this thesis explores the problem of automatically simplifying dependent reductions in the polyhedral model. To simplify dependent reductions, we will use the mathematical concept of duality which allows us to view this problem through the lens of program dependences. Duality greatly simplifies our approach in comparison with prior work, and we will it to augment a preexisting simplifying reductions algorithm.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Rajopadhye, the students in the HPC lab, CSU, and the computer science department for supporting me through this work. Particular thanks also goes to William Scarbro for helping to work through a number of counterexamples and edge cases to our work.

DEDICATION

I would like to dedicate this thesis to my parents, who have support me through all my adventures.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Redundant computations plague software. Often these redundant computations take the form of repeating the same calculation. These repeated computations are sometimes intentional as duplicating a small computation might be more efficient than storing the result somewhere and subsequently retrieving it. However, in most cases, they are simply the result of inefficient code and would lead to performance increases if removed. Furthermore, these computations are often obscured within the structure of how the code has been written, and without close inspection, they can be very challenging for a developer to see. Luckily, well designed compilers can automatically detect and remove unnecessary code via optimizing transformations. In this work we focus on one such optimizing transformation, simplifying dependent reductions.

Reductions are a ubiquitous programming abstraction. Many operations performed by the fold computation from functional languages are reductions. Many polyadic dynamic programming algorithms can be modeled as reductions [27]. They are also a vital part of many important algorithms in computational biology [13, 25], AI [16, 1], and fault-tolerant matrix multiplication [17]. Beyond important scientific algorithms, they are ubiquitous in everyday programming. Since this abstraction is so common, there is a large body of research looking into various aspects of reductions [18, 17, 6, 29, 21, 8, 23].

We focused on reductions specifically in a mathematical model of programming called the polyhedral equational model. This model has prior work examining the optimization of reductions, referred to as simplifying reductions [18, 6, 21, 29, 17]. We will extend this previous work by exploring simplifying dependent reductions, or reductions where steps in the reduction might need to be interleaved with other computations. In this case, we have fewer choices for optimizations that would still produce correct code, optimizations we will call legal or valid. For this extension, we turned to dependences, which tell us which steps in a computation rely on other steps, and polyhedral scheduling, which provides an order for performing the steps of a program.

Much of this work will rely on the relationship between dependences and scheduling, specifically their duality. We will consider everything in terms of program dependences, and we will use dependences to avoid illegal simplification. We will also use dependences to provide a sufficient condition for simplification without any affect on scheduling. In the case where this condition is not met, we also provide a heuristic for reuse vector selection that has led to an optimal version in all our testing. We integrate our work within an implementation of the simplifying reductions algorithm [6, 11, 18].

The layout of this thesis is as follows. Chapter two gives an example of simplifying a reduction to explain what it is and to convey the intuition behind the approach. It then describes the difficulty that arises when extending to a dependent reduction. Chapter three presents the mathematical background on the polyhedral model and introduces a number of key concepts our contributions rely on. Chapter four is where we present our contributions. In chapter five, we demonstrate our approach by simplifying Gibbs Sampling. The final chapters give an overview of future and related work, before we conclude this thesis.

Chapter 2

Simplifying Reductions

For those unfamiliar with dependent reduction simplification, in this chapter we give an overview of the problem. We start with an example reduction simplification, followed by an intuition for how to achieve this simplification automatically [6]. The final section of this chapter discusses the additional complications that arise when considering dependent reductions. This should demonstrate how reduction simplification identifies and removes repeated computation. Throughout the chapter we will introduce a number of important ideas in the context they will be used. We will only give a brief overview, and full definitions may be found in Chapter 3. Similar to other works on reductions, we explore simplification via the prefix sum computation.

2.1 An Example Simplification

A prefix sum operation is the cumulative sum of an array up to each corresponding index and is written with the following equation:

$$X[i] = \sum_{j=0}^{j \leq i} Y[j] \tag{2.1}$$

We can write this equation as the following code in a C-like language:

```
for(int i = 0; i < N; i++) {
  for(int j = 0; j <= i; j++) {
    X[i] += Y[j];
  }
}
```

The above implementation however repeats a number of computations. To explore these repeated computations and explain their removal, we will use polyhedral domains. Domains are necessary in the full formulation of the problem and central to the polyhedral model. From the

prefix sum program, we can construct a number of polyhedral domains. First is the reduction body, which is all combinations of loop indices in the imperative version. In the above code, it would be the set $\{[i, j] : 0 \leq j \leq i \text{ and } 0 \leq i \leq N\}$. Each of our variables also have their own respective domains. The data domains for X and Y are both $\{[i] : 0 \leq i \leq N\}$. Figure 2.1 visualizes the reduction body of the above for loops as a collection of integer points over a polyhedral domain where each point within the highlighted domain represents an individual loop iteration.

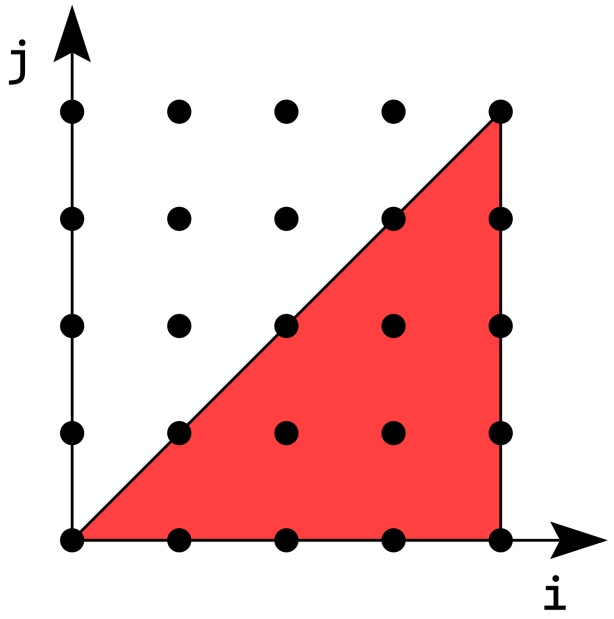

Figure 2.1: A visualization of the set of points described by $\{[i, j] : 0 \leq j \leq i \text{ and } 0 \leq i \leq N\}$, in this case $N = 4$

In Figure 2.2, we demonstrate the computations occurring over this reduction body. Specifically for our example, each row of points reads the same value in Y and the accumulation of each column defines the values in X. Our repeated computation arises because each row of points reads the same value, and therefore the columns where we accumulate the outputs are combining together the same values as well.

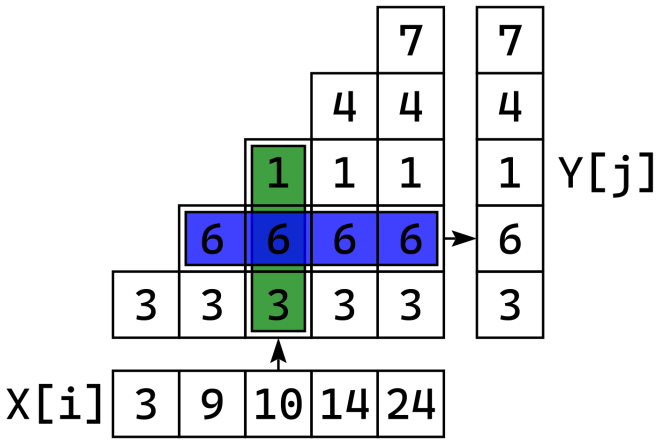

Figure 2.2: All the points in the green box are contribute to the same cell in X. All the points in the blue box will read from the same box in Y that the arrow points to.

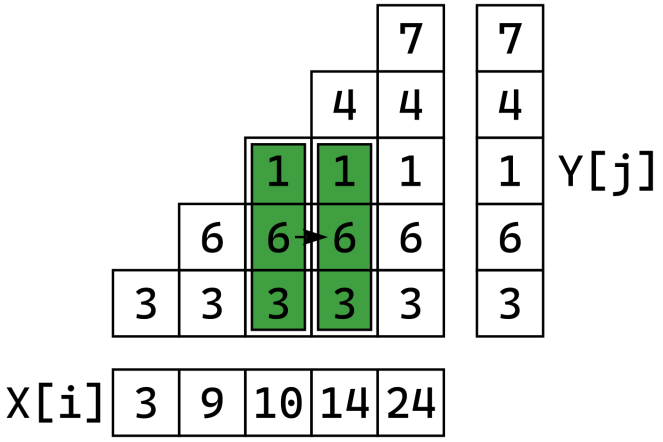

Figure 2.3: Both green boxes perform the same summation, that is Y[0] + Y[1] + Y[2] and this helps us visualize the repeated computation.

For an optimal computation, instead of computing the whole column, we simply combine the points in the diagonal with the previously computed values stored in X. Computation along this diagonal is exactly what our simplified C code will do and a simplified version would look like:

```
X[0] = Y[0];
for(int i = 1; i < N; i++) {
  X[i] = X[i - 1] + Y[i];
}
```

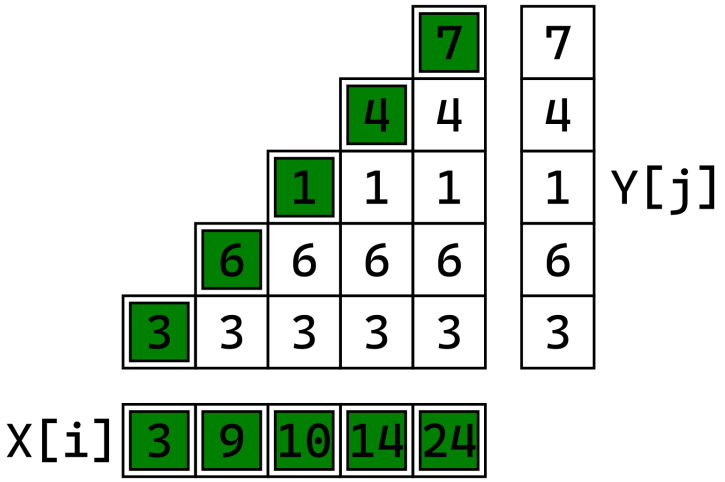

Figure 2.4: The green boxes now represent the only squares we need to combine to compute our final answer. To compute each value in X, it's corresponding square in the diagonal would be added to the previous result in X.

The goal of the simplifying reductions framework is to have the compiler automatically generate simplified versions of reductions like was just exemplified through the prefix sum. While the example presented is relatively straightforward, there are many examples of reductions where simplification is not so obvious. Previous work provides examples of everyday code that is non-trivial to simplify [6, 17]. The following is an example of the difficulty of hand optimization, albeit one this is somewhat contrived. It has computational complexity of $O(N^4)$, but admits an efficient version with complexity $O(N^2)$.

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=i; k<=2i; k++)
      for (l=i+j; l<=2j; l++)
        Y[i,j] = max(Y[i,j], X[k,l])
```

Not only is this simplification exceedingly challenging, a number of the reductions in real world code are also non-trivial to simplify. As previously stated, reductions are pervasive in scientific algorithms. These algorithms are often complex and require significant work to simplify by hand, and as these algorithms are often run over large datasets, reducing the computational complexity leads to large performance gains. Due to the challenge inherent in their simplification, optimization of repeated computation in reductions is vital, and we will now explore how the simplifying reductions framework achieves this automatically.

2.2 Automating the Simplification

To understand the intuition of how an automatic simplifier would reach the above simplification, we introduce reuse vectors. We shift our domain by a reuse vector which exposes repeated computations. We will discuss deriving the legal reuse vectors in more depth later, but for this example, we will use the two legal reuse vectors: $[1\ 0]$ and $[-1\ 0]$. We shift our domain by one of these reuse vectors, and the repeated computations are all the points in the intersection of the shifted domain with the original domain. The only points which we have to iterate over lie in what are called the residual domains, which correspond to the faces of our polyhedron and are the union of the two domains minus the intersection. For our two reuse vectors in the example, the only points we compute in the simplified version will belong to the highlighted green boxes in Figure 2.5.[1]

[1]There is no vertical green box in the right example because the points where we would have one lie outside of our data domain, i.e. we don't have to compute X[N].

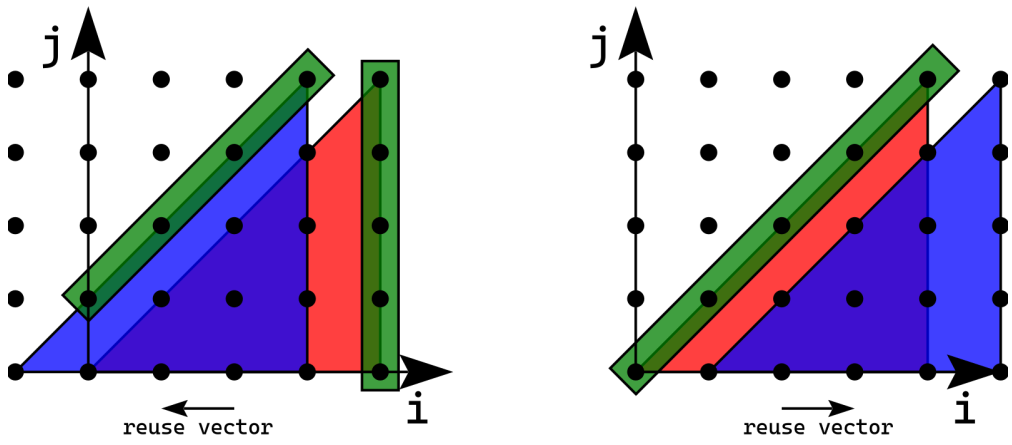

Figure 2.5: The blue area represents the example domains shifted by $[-1\ 0]^T$ and $[1\ 0]^T$ respectively. The green boxes are our residual domains and the overlap of the blue and red represent the points of repeated computation.

Using the reuse vector $[1\ 0]$ would lead to the simplified version of the code presented in the previous section. Simplification using the vector [-1 0] would lead to the following C code:

```
for(int i = 0; i < N; i++) {
    //The computation in the vertical green box
    X[N - 1] += Y[i];
}
for(int i = N - 2; 0 < i; i--) {
    //The computation in the diagonal green box
    X[i] = X[i + 1] - Y[i];
}
```

This simplification highlights one more key idea. In the above code we have used the subtraction operator. Each of our residual domains corresponds to a face of the polyhedron, and to each of these faces, for each reuse vector, we give a labeling of positive, negative, or neutral, meaning we will use the original operator, the inverse operator, or there will be no residual domain associated with that face. The first `for` loop would correspond to a positive face and the second `for`

loop would correspond to an inverse face and that is why the first uses $+$ and the second uses $-$. These labels will be used to partition our reuse vectors, and we will collect all of the faces of our polyhedral domain into a structure called the face lattice.

By moving computation to the residual domains and removing the intersection we have removed repeated calculations and achieved our goal of optimizing the program. With either selection of reuse, we optimize the program complexity from $O(N^2)$ to it's minimal complexity, $O(N)$.[2] The work of Gautam and Rajopadhye provides a framework for optimally reducing programs with reductions if there is any reuse in the original program [6]. Their work does not cover dependent reductions, and so we will now extend our example to highlight the added difficulty of simplifying these programs.

2.3 Extending to a Dependent Reduction

Our reductions might rely on other computations that cannot be fully computed beforehand, in which case we will have a dependent reduction. To demonstrate this, we will alter the prefix sum [29]. We can turn this prefix sum into a dependent reduction by changing the expression in the summation:

$$X[i] = \sum_{j=0}^{j<i} X[j] \tag{2.2}$$

Before the expression in the summation used only the variable Y, which we treated as an input, so we could ignore any dependences. In this above equation, because X is now defined in terms of itself, there are new dependences. In the polyhedral model, dependences tell us how specific points in a computation are related. Specifically, a dependence is a link between two points in our domains that exists whenever one point needs data computed in the other for its own computation, and in Figure 2.6 we see the dependence pattern that arises from the above equation.

[2]For those interested in how to derive the complexity from the domains see [6], but it related to the dimensions of our domain

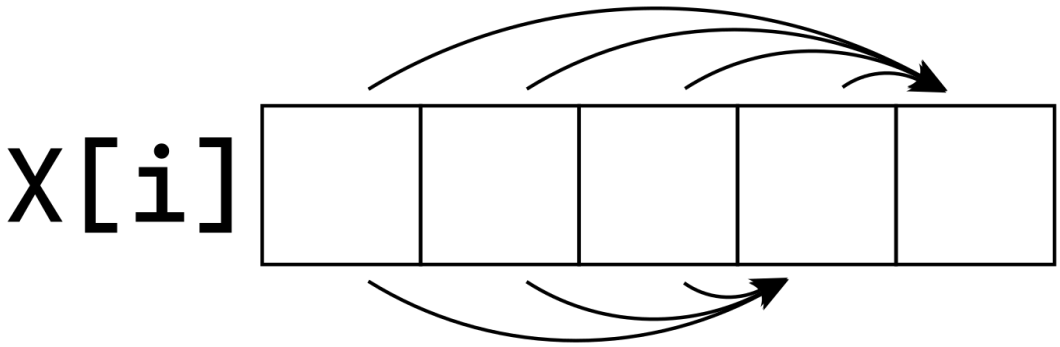

Figure 2.6: Here are the additional dependences that arise from changing our equation to a recursive one. We see that for each value X[i] we require all values X[j] for j < i.

If a point ever depends on itself, it is called a cyclic dependence and a program with a cyclic dependence will not have a schedule and cannot be computed. When we optimize a reduction, our simplification introduces new dependences. Using the original simplification algorithm in the above example could lead to an invalid program as the reuse vector $[-1\ 0]$ will lead to a simplified program with a cyclic dependence. Figure 2.7 highlights one of the cycles that will arise from using the reuse vector $[-1, 0]$. Also highlighted in Figure 2.7, we demonstrate the general pattern that will arise using the reuse vector $[1\ 0]$ which does not lead to cycles, and therefore it is still valid even in the dependent case.

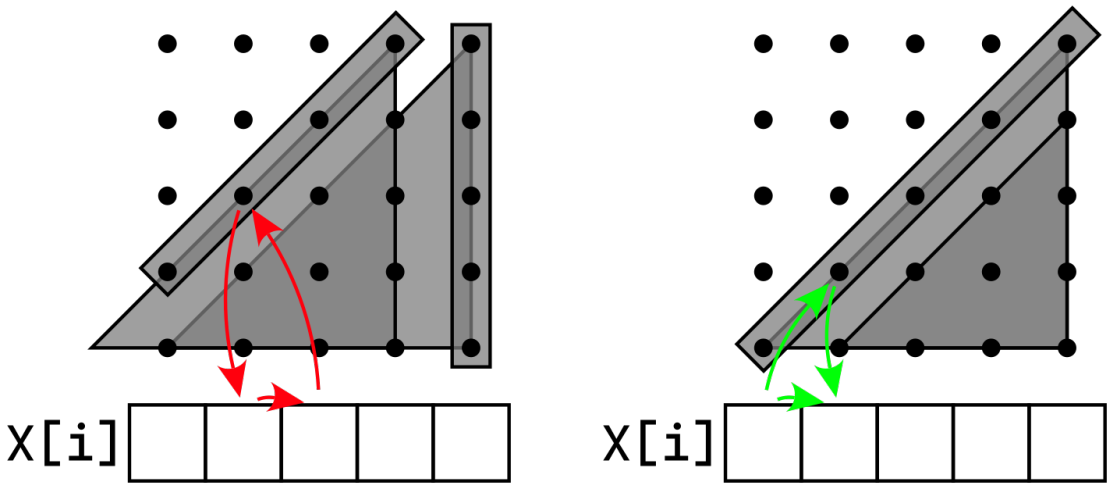

Figure 2.7: Here is one of resulting cycles from using $[-1\ 0]$, as well as the general pattern that arises from using the vector $[1\ 0]$.

Chapter 3

Background

This chapter establishes the necessary mathematical background for the rest of the thesis. The foundation for our work is the polyhedral model. The polyhedral model provides a method for reasoning about programs, along with a number of transformations that optimize those programs. In the model programs are defined over polyhedral domains, and the model is frequently used to reason about nested loop programs. For our purposes, we use the model to view programs as sets of equations defined over these domains.

3.1 Domains

A polyhedral domain is an integer polytope (the n-dimensional extension of a polygon). It can be formulated as all points $\{x \in \mathbb{Z}^n | Ax + b \geq 0\}$. In the polyhedral model, these domains are frequently extended with parameters, i.e., instead of considering the points $0 \leq i < 10$, we consider the points $0 \leq i < n$. This defines an infinite family of polytopes where the parameters determine the size of the polytope. We often consider a parameter as just the addition of another variable/dimension.

We represent both the iterations and the data space for a given program as domains. The iteration domains represent computations and the data domains contain the values used or computed. The reduction body is a specific instance of an iteration domain and corresponds to the set of legal values of loop indices when reductions are implemented in a sequential iterative program. The key to reduction simplification is determining when points in this domain repeat computation. The data domain defines the locations of all the different values our computations use, compute, and store. It can be thought of as the defining the dimensions of an array structure.

3.1.1 The Face Lattice

For a given polyhedral domain, we can create the face lattice. We summarize the definition given by Narmour et al [18]. A face of a polyhedron, D, can be generated by intersecting D with one or more equalities obtained by saturating one or more inequalities. Saturating an inequality, $\alpha \cdot z + \gamma \geq 0$, turns it into an equality of the form $\alpha \cdot z + \gamma = 0$. The face lattice of a given polyhedron, D, is a graph whose nodes are the faces of D. We can divide our face lattice into levels, where all the nodes in each level have the same number of saturated constraints. A facet will be an $N - 1$ dimensional face of D, where N is the dimension of D. Figure 3.1 gives the face lattice for the reduction body of our prefix sum.

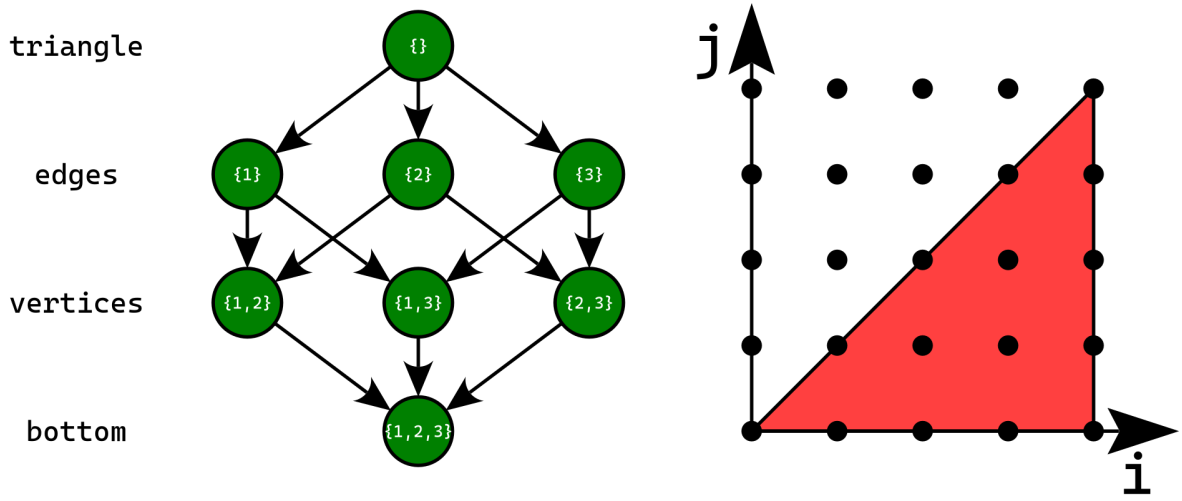

Figure 3.1: The face lattice for our triangular domain. Our domain is constructed from three constraints $0 \leq j$, $j < i$, and $i \leq N$. The numbers in the graph represent which of these constraints have been saturated. The node "1" is the face constructed by setting $0 \leq j$ to $j = 0$, so this face is one of the sides of our triangle. Bottom is the least element in the lattice.

3.2 Dependences

A dependence exists between two points in the domains when the computation occurring at one point requires the value computed the other point. Dependences will be vectors, and specifically a

dependence, d, is $d = i - j$, when the point i, relies on the point j [10]. The dependences can be collected into a set called the dependence polyhedron. To construct the dependence polyhedron, we collect the dependence conditions derived from the program (which points depend on which over what domains), into a set of linear equalities and inequalities, and then project these into the dependence space, projecting from points to dependence vectors, and finally take the convex hull of this projection [30, 10].

3.3 Scheduling

Scheduling is the process of ordering a computation and assigns time stamps to each step in the computation. It is a long-explored problem that derives the computation order based on some optimization metric and is an important part of polyhedral analysis [19, 8, 15, 9, 5, 23]. Just as reduction simplification has an associated search space, there is also an associated search space for scheduling. This search space, the feasible space of the schedule, is derived from the dependences, and collects all valid linear functions from the iteration domain to the time domain. Specifically the constraints respect *causality*, which means for a given schedule, λ, the constraints will have the form, $\lambda(p_1) \geq \lambda(p_2) + 1$, where p_1 and p_2 are two points in the domain and p_1 depends on p_2 [23]. When viewed using dependences, the constraint is $\lambda d > 0$, where d is a dependence vector. A scheduling algorithm will take in an iteration domain along with the dependences and produces a schedule map from the iteration domain to a time step. It produces the schedule by formulating the constraints into a linear programming This time step might be multidimensional and respects an order called the lexicographic order, which is the following relation: ($a \ll b$) where $\exists i(a[1..i] = b[1..i] \wedge a[i+1] < b[i+1])$ [23]. The complete scheduling function is the collection of all these maps with a different map for each variable in the program. Previous work on simplifying dependent reductions relies on formulating the feasible space of the schedule [29, 21].

3.3.1 Space of Tiling Hyperplanes

Tiling is related to scheduling, and seeks to improve the locality of generated programs by breaking the computation up into tiles [2]. We can construct the space of tiling hyperplanes, and each hyperplane will be defined in terms of its normal vector. Where the schedule must respect *causality*, the space of tiling hyperplanes must respect *atomicity*, implying that no two tiles both depend on each other [2]. This relaxes the constraint slightly, instead of $\lambda d > 0$, where d is a dependence vector, the constraints on the tiling hyperplanes will be $\psi d \geq 0$, again where d is a dependence vector.

3.4 Reductions and Reduction Simplification

For our purposes, reductions have the following syntax [6]:

$$reduce(\oplus, f_w, expr) \tag{3.1}$$

This whole expression computes over the reduction body, where f_w maps from the reduction body into the write space (all the points where the data will be accumulated). The $\oplus$ is an associative, commutative operator. The expression, $expr$, is the computation that will take place at all the points in our reduction body, and these computations are combined with our operator. Also, we derive from a reduction another projection function, the read function. The read function maps from the reduction body to the values required at each point in our computation. The prefix sum example from earlier (2.1) would take the following form:

$$X[i] = reduce(+, (i, j \to i), \{0 \leq j \leq i\} : Y[j]). \tag{3.2}$$

The + from our reduction domain is the += operator in the for loop. The right-hand side of the assignment is the $expr$ ($Y[j]$ in this case). The domain $\{0 \leq j \leq i\}$ tells us which parts of the reduction body are to be combined into each point.

3.4.1 Reduction Simplification

As we have seen, reduction simplification shifts the reduction body along a reuse vector, creates residual domains by subtracting the intersection of the shifted domain with the original domain from their union, and computes the remaining points in the residual domains. This whole process relies on finding the space of all possible reuse vectors. The reuse space is simply the kernel of the read function, $ker(f_r)$, or all the points that f_r sends to zero [6]. Any vectors in the reuse space represent the directions in which we have repeated computation, and thus, shifting the computation domain along one of these vectors can lead to the discovery of a simplified computation. There is possibly an infinite number of reuse vectors to consider so we partition the reuse space into equivalence classes using the face lattice. A label assigns to each facet, positive, negative, or neutral, depending on which operator will be used to combine the computations in that residual domain (neutral means reuse is along that facet, and there will be no residual computation). Each vector that generates the same labelings for each of the facets is considered to be in the same class.

Simplifying reductions is a recursive dynamic programming algorithm that traverses the face-lattice of the reduction body to maximally apply simplification transformations to a program. Starting with the whole reduction body, at each step, we simplify the facets of the current face, F. The key idea is that exploiting reuse along a given reuse vector, ρ, avoids evaluating the reduction expression at most points in F. At each step of the recursion, the asymptotic complexity is reduced by exactly one polynomial degree. Furthermore, at each step, the newly chosen, ρ, is linearly independent of the previously chosen ones. Hence, the method is optimal—all available reuse is fully exploited. Recent work in this area has provided an implementation to automatically simplify reductions, choosing the smallest integer point closest to the origin from any given equivalence class of reuse vectors [18].

In the non-dependent reduction case, we can select a reuse vector arbitrarily from an equivalence class to reach an optimal simplification [6]. However the construction of the equivalence classes does not take program dependences into account, meaning some reuse vectors in an equivalence class might respect a dependency that other vectors in the same equivalence class of reuse

vectors might not. In the case of dependent reductions, if we choose our reuse vector arbitrarily from an equivalence class, that selected reuse vector might prevent a future simplification that would have been possible if a different reuse vector from the same equivalence class was chosen, and this future simplification might be optimal.

3.5 Simplification and Scheduling

When we simplify dependent reductions, we are solving a problem with two unknowns. Recall that the constraints on the schedule have the form $\lambda d > 0$. Normally when solving for the schedule, the dependences are known beforehand, and therefore the constraints are fixed. The schedule can then be solved for using a linear programming approach. The process of simplification, however, introduces new dependences, and the choice of reuse vector changes which new dependences are introduced. Since the dependences introduced by simplification are unknown, both variables in the constraints are unknown rendering the constraints bilinear, and previous work in this area formulates the solution for scheduling and simplification as a bilinear programming problem [29]. To avoid simultaneously optimizing for both, we rely on the fact that the dependences, the feasible space of the schedule, and the space of reuse vectors are all cones. Also that the dual to the dependence cone is the feasible space of the schedule.

3.6 Cones

A convex cone is the positive linear combinations of a set of vectors, called generators.

Definition 1. ***Convex Cone.*** *Given a set of generators, G, the convex cone, C, is $C = \{\sum \alpha_i g_i | \alpha \geq 0$ and $g_i \in G\}$ [3].*

A cone is said to be blunt if it does not contain the origin. For any given cone, we have a set of extreme (or extremal) rays. We will also be able to expand any cone by adding a vector to the set of generators for that cone. Figure 3.2 gives a visual example of a cone.

We usually represent the polyhedral domains in terms of their constraints, but it is also possible to view them as a collection of lines, rays, and vertices. The rays of the polyhedron form a cone.

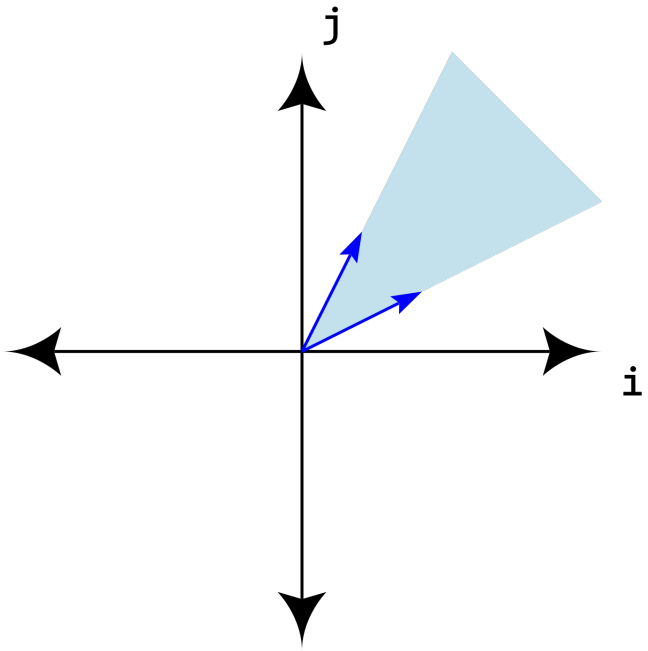

Figure 3.2: A sample cone defined by the extremal rays [1 2] and [2 1]. The blue area represents all the points in our cone

To acquire the representation of the lines, rays, and vertices, we rely on Chernikova's algorithm [12]. This algorithm is it's own dual; when it is run on constraints, it yields the lines, rays, and vertices, and when it is run on the lines, rays, and vertices, it yields the constraints. A number of the objects discussed so far in this chapter are cones:

- The feasible space of the schedule [4]
- The space of tiling hyperplanes [2]
- The reuse space
- The dependences [30]

From the dependence polyhedra we can derive the dependence cone. This cone will be such that if a one point, i, must execute after another point, j, then $i - j$ will be in the dependence cone, i.e. $\sum \alpha_i g_i = i - j$, where $\alpha_i \geq 0$, and g_i is a generator of the dependence cone [10].

3.6.1 Negation

Definition 2. *The **negative cone** is the negation of a dependence cone, $-D$. It is also a cone whose generators are the inverses of the generators in the original cone. See Figure 3.3, for a visualization of this definition.*

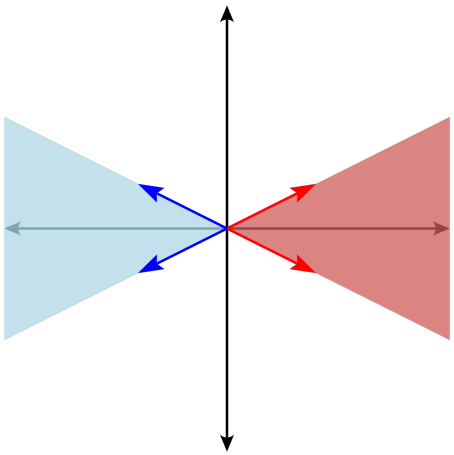

Figure 3.3: The blue area represents our cone, C, and the red area the *negative cone*.

.

3.6.2 Dual Cone

The dual cone is the collection of vectors whose dot product with all the points in the original cone in greater than or equal to zero. Figure 3.4 gives an example of a dual cone.

Definition 3. ***Dual Cone.*** *Given a cone, C, the dual cone, C', is $C' = \{\alpha | \alpha \cdot c \geq 0$ and $\forall c \in C\}$.*

Since it is defined in terms of the original cone, any changes to the original cone will lead to changes in the dual cone.

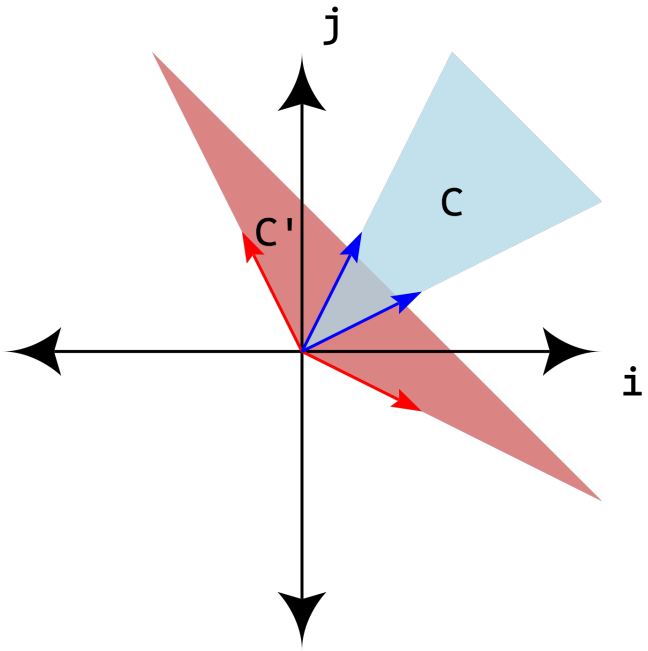

Figure 3.4: A sample cone in blue with the red cone representing the dual.

Chapter 4

Simplifying Dependent Reductions using Duality

We are now ready to discuss our main contributions to the problem of simplifying dependent reductions. Our solution extends the original dynamic programming algorithm by altering the way we select reuse vectors [6]. We give sufficient conditions for selecting the reuse vector so that it has no effect on scheduling, and a heuristic that leads to legal simplification when this condition is not satisfied. To this end, we will develop our method and then describe its integration into the existing simplifying reductions implementation provided by Narmour et al [18].

4.1 Exploiting Duality

As we discussed in Section 3.5, the problem we are trying to solve contains two unknowns, the (new) dependences, and the schedule, which means the causality constraint is bilinear. To avoid a bilinear solution, however, we will view everything through dependences. The first part of this view is reasoning directly about about the dependence introduced when using a particular reuse vector for simplification. When simplifying a reduction, the new dependence in the simplified program is the write function, f_w, applied to the reuse vector, we will call this the *induced dependence*. We will call the entire image of the write function applied to the whole reuse space the *induced dependence space*. The second part of this view uses duality. The dual of the dependence cone is the space of tiling hyperplanes. The constraints on the space of tiling hyperplanes are $\psi d \geq 0$, which is exactly the dual to the dependence cone. Furthermore, the feasible space of the schedule is a subset of the tiling hyperplanes, and therefore a subset of the dual cone. Specifically, the constraints on the schedule are $\lambda d > 0$, which means the set of our schedules is $\{\lambda | \lambda d > 0, \forall d \in D\}$, where D is the dependence cone. As the feasible space of the schedule is defined in terms of the dependences, any changes to the dependence cone will change the schedule space. That is, when we expand the dependence cone, we will be adding new constraints on the schedule.

From the duality relation, we can arrive at two conclusions. First if the induced dependence is already contained in the dependence cone, the new scheduling constraints will be subsumed by the preexisting constraints, and so the new dependence will have no effect on the schedule. Second, a reuse vector that introduces a dependence in the negation of the dependence cone will render the resulting program unscheduable as we will have to satisfy both $\lambda d > 0$ and $-\lambda d > 0$.

4.2 Reuse Selection using the Dependence Cone

We now introduce a way to alter the reuse vector selection using the dependence cone. We will discuss the two cases that arise when we approach the problem from this view. First is our sufficient condition, where our induced dependence space and dependence cone intersect allowing us to simplify without any affect on the feasible space of the schedule, the second case is when they are disjoint and we must add new constraints to the schedule space, and for this we provide a heuristic selection criteria that will ensure legal simplification.

4.2.1 Intersecting Case

When the intersection of dependence cone and induced dependence space is not empty, we will choose our reuse vector from this intersection. This reuse vector respects the program dependences and therefore leads to a legal simplification. Also as the new dependence will already be contained in the dependence cone, any schedule that respects the constraints of the original program, will respect the constraints of the simplified program. Consequently, this will add no new constraints to the schedule space, and therefore no additional constraints on future reuse selection either. In Figure 4.1, we see an example of this selection criteria.

4.2.2 Disjoint Case

When our cones are disjoint we will not be able to guarantee that the reuse vector selected has no impact on the feasible space of the schedule, and we are not able to avoid the bilinear constraint. In our implementation, we provide a heuristic selects one of the extremal rays of our reuse space

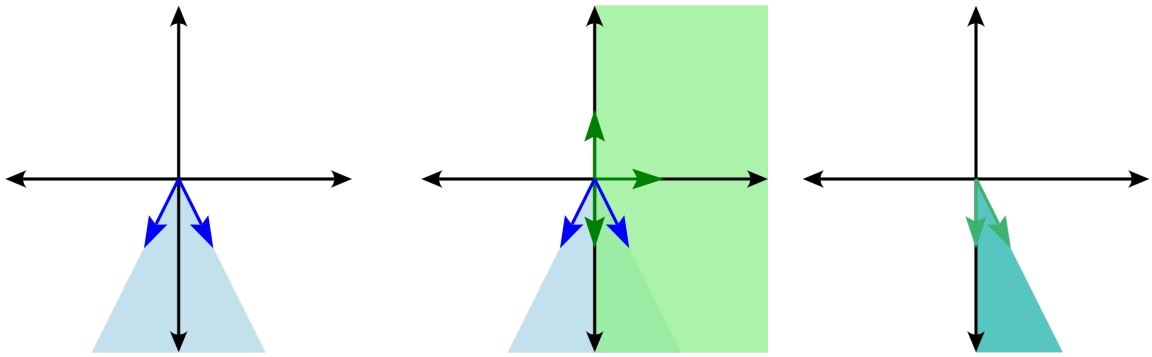

Figure 4.1: In the above diagram, we illustrate the process of refining our reuse space. We start with a dependence cone on the far left, then we combine it in the middle with our reuse space, and the right image is their resulting intersection. We would pick our reuse vector from this space.

as the reuse vectors. However, we will filter our extremal rays by removing vectors that lead to illegal simplifications.

As previously discussed, the reuse vector will induce a dependence in our newly simplified program, and our reuse vector will be invalid if it leads to a cycle in the that new program. If the induced dependence is the inverse of a generator of our dependence cone then the simplification will lead to a cycle. With this in mind, we remove any reuse vectors whose induced dependences are in the negation of the dependence cone from consideration for simplification.

4.3 Implementation

As a part of this work, we integrated our new reuse selection into previous work on simplifying reductions [6, 18]. This chapter will give an in depth description of our integration. Our implementation relies on the ISL and PolyLib libraries [26, 28]. We provide an implementation in the AlphaZ which is a compiler for the polyhedral language Alpha. But before we discuss the

integration of our restriction in any more depth, we start with a short description of the original implementation of the simplification algorithm and how it selects reuse vectors.

As previously stated, simplifying reductions is a recursive dynamic programming algorithm. At each recursive step of the algorithm, it selects a list of possible reuse vectors. These are then the vectors used to generate the simplifications, and the algorithm is again applied to those simplifications. The original reuse selection function takes in the reuse space and returns a list of possible reuse vectors, one per labeling. To return that list, it generates the reuse constraints associated with each labeling of the faces and intersects each of those with the reuse space. This generates the equivalence classes of reuse vectors. From each of these equivalence classes, the selection function picks the integer point closest to the origin as the reuse vector [11]. These vectors are collected together and returned as candidates to use for simplification.[3] This is presented in Algorithm 1.

We augment the preexisting process by first building the dependence graph for the program and deriving the dependence cone. The dependence cone is then passed to the simplifying reductions framework and ultimately to the reuse selection function, where it is used as a part of the reuse vector selection. ISL and Polylib give us the set relevant functions, and we use these libraries to find the intersection of dependence cone and the reuse space. If this is not empty then we restrict our space to this intersection and then pass that space along to the original reuse selection function. If there is no intersection, we use these libraries to generate the rays of the reuse space. Once the rays are obtained, they are then filtered for legality using the negative cone. If any of them are members of the negative cone, then they are removed from consideration. This filtered list of extremal rays are returned as the candidate reuse vectors to be used for simplification. In Algorithm 2, we give the new reuse candidate selection function.

[3]We have presented a simplified picture, as the original selection function also generates the face lattice and performs some validity checks as well.

Algorithm 1: Original Reuse Selection

Data: $reuseSpace$ is the reuse space
Data: $labelings$ are all labelings of the facets
Result: A list of reuse vectors

```
reuseEquivalences ← [] ;
for label in labelings do
    labelDomain ← getLabelDomain (face, label) ;
    space ← intersect (labelDomain, reuseSpace) ;
    if !isTrivial (space) then
        reuseEquivalences.add(space) ;
    end
end
validReuseVectors ← [] ;
for domain in reuseEquivalences do
    candidateReuseVector ← integerPointClosestToOrigin (domain) ;
    if testLegality (re, candidateReuseVector) then
        validReuseVectors.add(candidateReuseVector) ;
    end
end
return validReuseVectors ;
```

Algorithm 2: Reuse Vector Selection using the Dependence Cone

```
Data: reuseSpace is the reuse space
Data: dependenceCone is the dependence cone
Data: labelings are all labelings of the facets
Result: A list of reuse vectors
reuseVectors ← [] ;
if intersect (project (reuseSpace), dependenceCone) != ∅ then
    reuseEquivalences ← [] ;
    for label in labelings do
        labelDomain ← getLabelDomain (face, label) ;
        space ← intersect (labelDomain, reuseSpace) ;
        if !isTrivial (space) then
            reuseEquivalences.add(space) ;
        end
    end
    for domain in reuseEquivalences do
        candidateReuseVector ← integerPointClosestToOrigin (domain) ;
        if testLegality (re, candidateReuseVector) then
            reuseVectors.add(candidateReuseVector) ;
        end
    end
    return reuseVectors
end
else
    reuseRays ← generateRays (reuseSpace) ;
    negativeCone ← invert (dependenceCone) ;
    for ray in reuseRays do
        if ! negativeCone.contains(ray) then
            reuseVectors.add(ray) ;
        end
    end
    return reuseVectors
end
```

4.3.1 Revisiting the Example

We will again revisit the dependent prefix scan example to illustrate this method. Recall our dependent prefix scan is defined by the following equation:

$$X[i] = \sum_{j=0}^{j<i} X[j] \tag{4.1}$$

In this example our dependence cone, consists of the space spanned by the single ray [1]. After applying the appropriate projections and intersecting this with our reuse space, we are left with the space spanned by the ray [1 0], allowing us to shift the reduction body along a vector in the positive i direction. This leads to the simplification:

$$X[i] = X[i-1] + X[i-2] \tag{4.2}$$

It also prevents the illegal simplification that would arise from shifting in the negative i direction. Vectors in the negative i direction would be part of our negative cone and therefore they will not be considered as candidates for simplification. The following is the illegal simplification that will be avoided:

$$X[i] = X[i+1] + X[i-2] \tag{4.3}$$

In (4.3) we can see there is no order which would solve the equations because there will be a number of cyclic dependencies.

Chapter 5

Simplifying Gibbs Sampling

We evaluate the implementation by demonstrating the dependent reduction simplification of a real-world problem, namely Gibbs Sampling. We give a brief overview of the problem. We then follow the specific version of the algorithm as given by Yang et al to demonstrate the feasibility of our approach [29].

5.1 Gibbs Sampling

We turn now to Gibbs Sampling, an example of a dependent reduction with broad applicability in a variety of different areas [7, 24]. Gibbs sampling is Markov Based probability sampling algorithm for generating a multivariate probability distribution [7]. Resnik and Hardisty give a good overview of the overall process of Gibbs Sampling [24]. It is a maximum a posterior (MAP) estimation method which allows us to take into account prior knowledge about our distribution. As a Markov Chain mode, the state produced by each iteration of the algorithms only depends on the prior state. Specifically for Gibbs Sampling, instead of choosing all the new states at once, the algorithm updates one state at a time based off of all the other current states, and in this process uses any state that has already been updated in the current iteration.

The overall algorithm for describing this system can be described using dependent reductions. Along the lines of Yang et al, we use a specifically the two-cluster gaussian mixture model as an example of a dependent reduction [29, 16]. This model separates a set of observations into two distinct clusters or classifications. We use the following set of equations. They are a part of a larger system that iterates until it arrives at a stable probability distribution [29].

$$C_{zi} = \sum_{\forall j s.t. j \neq i \wedge Z_j = z} 1, \forall z, i \tag{5.1}$$

$$S_{zi} = \sum_{\forall j s.t. j \neq i \wedge Z_j = z} obs_i, \forall z, i \tag{5.2}$$

$$P_0(z, i) = P(obs_i | obs_j, Z_j, Z_i = z) \forall j s.t. j \neq i \tag{5.3}$$

$$P_z(i) = P(Z_i = 0 | Z_j, obs) \forall j s.t. j \neq i \tag{5.4}$$

$$Z_i \sim P_z(i), \forall i \in \{1...N\} \tag{5.5}$$

5.2 Simplifying the System

Using this mixture model, we restrict our system to two variables and will therefore cluster samples into two groups, (the dimension of Z is equal to 2 and the equations associated with each variable are labeled with a 0 or 1). Figure 5.1 lists the equations that model the update that occurs at each step of the algorithm. Z is the input cluster assignment or the result of the previous iteration, whereas Z' represents a random sampling of the current distributions (which are derived from each S and C equation). Obs will be the observed value. The S variables represent the sums of the observations for each cluster. $S1$ will be the sum of observations in cluster one and $C1$ will be the count of the number of observations in that cluster. We also break the sums and counts into separate variables for the left and right (labeled L and R). The left is used to aggregate these values from the input or the previous sample, while the right variables read from the current distribution. Figure 5.1 gives the set of equations used to describe a single update of the system. The above equations then can be simplified using our system. One possible simplification uses the same reuse vector for each equation, and each equation is simplified similarly to the $S1L$ in Figure 5.2.

Keen observers might notice that each of our equations in the series of equations look like the prefix sum example, and they are in fact very similar. In each case we can choose our reuse vector from two equivalence classes where one equivalence class leads to a valid simplification and the other to illegal simplifications. Using our implementation, the only program returned by

```
S0R[i] = reduce(+, (i, j -> i), {: 0 <= i < j <= N}:
    if(Z[j] = 0[]) then obs[j] else 0[]);
S0L[i] = reduce(+, (i, j -> i), {: 0 <= j < i <= N}:
    if(Z'[j] = 0[]) then 1[] else 0[]);
S1R[i] = reduce(+, (i, j -> i), {: 0 <= i < j <= N}:
    if(Z[j] = 1[]) then obs[j] else 0[]);
S1L[i] = reduce(+, (i, j -> i), {: 0 <= j < i <= N}:
    if(Z'[j] = 1[]) then 1[] else 0[]);
C0R[i] = reduce(+, (i, j -> i), {: 0 <= i < j <= N}:
    if(Z[j] = 0[]) then 1[] else 0[]);
C1R[i] = reduce(+, (i, j -> i), {: 0 <= i < j <= N}:
   if(Z[j] = 1[]) then 1[] else 0[]);
C1L[i] = reduce(+, (i, j -> i), {: 0 <= j < i <= N}:
    if(Z'[j] = 1[]) then 1[] else 0[]);
C0L[i] = reduce(+, (i, j -> i), {: 0 <= j < i <= N}:
    if(Z'[j] = 0[]) then 1[] else 0[]);
Z'[i] = case {
 {: i = 0}: sample(C1R[i], C0R[i], S0R[i], S1R[i]);
 {: 0 < i}: sample(C1R[i] + C1L[i],
                   S0R[i] + S0L[i],
                   C0L[i] + C0R[i],
                   S1L[i] + S1R[i]);
};
```

Figure 5.1: The equations in Alpha for the distribution in Gibbs Sampling.

```
S1L[i] = case  {
 {: i = 1 } : S1L_pos[i];
    {: 2 <= i <= N } : (S1L_pos[i] + S1L[i-1]);
};

S1L_pos[i0] = if (Z'[i0-1] = 1[]) then 1[] else 0[];
```

Figure 5.2: An examples simplification for one of the equations.

simplification was the one that chose from the legal set of reuse vectors at each step. Overall using our simplification technique, we were able to generate an optimized program that reduced this update state at each iteration from an $O(N^2)$ operation to an $O(N)$ operation. This gives a positive demonstration that our implementation works as intended and demonstrates its viability on a real world algorithm. Especially an algorithm with an optimal version that is challenging to arrive at by hand [29].

Chapter 6

Related Work

As seen throughout this thesis, there is a long strain of research contributing to the polyhedral model. Because it can encapsulate so many different aspects of computation, there are many other parts that have not been explored in this thesis, such as tiling and parallelization. There is a large amount of directly related work which we can divide into three separate areas: reduction simplification, scheduling, and simplifying dependent reductions.

6.1 Simplifying Reductions

The simplifying reduction framework laid out in this thesis originates in the work of Gautam and Rajopadhye [6]. Since it's original publication, there have been a few extensions. Recent work has provided an implementation, and this implementation and provides a method for selecting a specific reuse vector from a given equivalence class of reuse vectors. Specifically it selects the closest integer point to the origin [11]. Other recent work includes expanding the simplifying reductions framework to expanding to another set of programs not handled by the original algorithm. It does this by breaking of the reduction body using index-set splitting, which is a piece-wise affine transformation that divides the reduction body into a number of pieces which can be simplified independently [17].

6.2 Scheduling

As scheduling plays a vital role in polyhedral research there a large body of work that goes into scheduling. Early works give the initial formulation of the scheduling problem as a linear programming problem, and describe how to derive scheduling constraints that can be passed to a PIP solver [15, 5, 22, 20]. More recent work fully formulates the problem of multidimensional scheduling as a single linear programming problem [19]. There has also been work on scheduling reductions specifically [8, 23].

6.3 Simplifying Dependent Reductions

As discussed throughout this thesis, we are not the first to explore the problem of simplifying dependent reductions. The first work we found in the area was the work of Yang et al [29]. Their solution simultaneously optimizes for the schedule and reuse using a bilinear approach. They also provide a heuristic algorithm that greatly simplifies the computational complexity of their approach. Rajopadhye develops a validity check that determines if a given reuse vector is valid [21]. This check uses the generators of the feasibility space of the schedule.

Chapter 7

Future Work

The major piece of future work is a direct extension of this work. That is to explore further the disjoint case where we rely on a heuristic. This could include finding a new way to partition the reuse space to take dependences into account. This could ensure that an optimal solution is always reached.

There are also other extensions to the original simplifying reductions algorithm. The initial formulation of simplifying reductions handles single parameters [6]. This is sufficient for many computations, but would require a user to hard code certain values that might otherwise rely on parameters. The challenge is deducing the simplification with the smallest complexity, i.e. how does one choose a program that is $O(N^2)$ vs $O(N * M)$ vs $O(M^2)$. Future work could explore how to simplify a program with multiple parameters. Also the original algorithm only ensures optimal complexity, and interesting work might seek to extend the algorithm to ensure optimal complexity even with respect to constant coefficients.

Chapter 8

Conclusion

In this work we have explored the removal of repeated computation via the technique of simplifying reductions. We presented an alternative view of the simplifying dependent reductions problem that relies solely on program dependences by using duality. This allowed us provide sufficient conditions for simplification without adding new constraints on the schedule as well as a heuristic for selecting reuse when new constraints on the schedule must be introduced. We were able to integrate it within the original recursive dynamic programming algorithm to simplify reductions [6]. We provided an implementation of this work and demonstrated its feasibility on a real world problem. This gives us a method to optimize complex algorithms automatically which can provide performance increases of a polynomial degree or more.

Bibliography

[1] Eli Bingham et al. “Pyro: deep universal probabilistic programming”. In: *J. Mach. Learn. Res.* 20.1 (Jan. 2019), 973–978. ISSN: 1532-4435.

[2] Uday Bondhugula et al. “A practical automatic polyhedral parallelizer and locality optimizer”. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’08. Tucson, AZ, USA: Association for Computing Machinery, 2008, 101–113. ISBN: 9781595938602. DOI: 10.1145/1375581.1375595. URL: https://doi.org/10.1145/1375581.1375595.

[3] Alain Darte, Yves Robert, and Frédéric Vivien. *Scheduling and Automatic Parallelization*. Birkhäuser Boston, 2000. Chap. Parallelism Detection in Nested Loops. ISBN: 978-0-8176-4054-0. DOI: 10.1007/978-1-4612-1362-8. URL: https://doi.org/10.1007/978-1-4612-1362-8.

[4] Jean-Marc Delosme and Ilse C. F. Ipsen. “Systolic array synthesis: Computability and time cones”. In: *International Workshop on Parallel Algorithms and Architectures*. Ed. by M. Cosnard et al. Elsevier Science, North Holland, Apr. 1986, pp. 295–312.

[5] Paul Feautrier. “Some efficient solutions to the affine scheduling problem. I. One-dimensional time”. In: *International Journal of Parallel Programming* 21.5 (1992), pp. 313–347. ISSN: 1573-7640. DOI: 10.1007/BF01407835. URL: https://doi.org/10.1007/BF01407835.

[6] Gautam and S. Rajopadhye. “Simplifying reductions”. In: *SIGPLAN Not.* 41.1 (Jan. 2006), 30–41. ISSN: 0362-1340. DOI: 10.1145/1111320.1111041. URL: https://doi.org/10.1145/1111320.1111041.

[7] Stuart Geman and Donald Geman. “Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-6.6 (1984), pp. 721–741. DOI: 10.1109/TPAMI.1984.4767596.

[8] Gautam Gupta, Sanjay Rajopadhye, and Patrice Quinton. “Scheduling reductions on realistic machines”. In: *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’02. Winnipeg, Manitoba, Canada: Association for Computing Machinery, 2002, 117–126. ISBN: 1581135297. DOI: 10.1145/564870.564888. URL: https://doi.org/10.1145/564870.564888.

[9] Gautam Gupta, Sanjay Rajopadhye, and Patrice Quinton. “Scheduling reductions on realistic machines”. In: *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’02. Winnipeg, Manitoba, Canada: Association for Computing Machinery, 2002, 117–126. ISBN: 1581135297. DOI: 10.1145/564870.564888. URL: https://doi.org/10.1145/564870.564888.

[10] F. Irigoin and R. Triolet. “Supernode partitioning”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’88. San Diego, California, USA: Association for Computing Machinery, 1988, 319–329. ISBN: 0897912527. DOI: 10.1145/73560.73588. URL: https://doi.org/10.1145/73560.73588.

[11] Ryan Job. “Automatically Simplifying Reductions”. MS. Colorado State University, 2024.

[12] Hervé Le Verge. *A Note on Chernikova’s algorithm*. Research Report RR-1662. INRIA, 1992. URL: https://inria.hal.science/inria-00074895.

[13] R B Lyngso, M Zuker, and C N Pedersen. “Fast evaluation of internal loops in RNA secondary structure prediction.” In: *Bioinformatics* 15.6 (June 1999), pp. 440–445. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/15.6.440. eprint: https://academic.oup.com/bioinformatics/article-pdf/15/6/440/48835148/bioinformatics_15_6_440.pdf. URL: https://doi.org/10.1093/bioinformatics/15.6.440.

[14] Christophe Mauras. “Alpha : un langage equationnel pour la conception et la programmation d’architectures paralleles synchrones”. Thèse de doctorat dirigée par Quinton, Patrice Infor-

matique Rennes 1 1989. PhD thesis. 1989, 1 vol. (109 p.) URL: http://www.theses.fr/1989REN10116.

[15] Christophe Mauras et al. “Scheduling affine parameterized recurrences by means of variable dependent timing functions”. In: RR-1204. 1990. URL: https://inria.hal.science/inria-00075354.

[16] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. Cambridge, Massachusetts: MIT Press, 2012.

[17] Louis Narmour, Tomofumi Yuki, and Sanjay Rajopadhye. “Maximal Simplification of Polyhedral Reductions”. In: *Proc. ACM Program. Lang.* 9.POPL (Jan. 2025). DOI: 10.1145/3704839. URL: https://doi.org/10.1145/3704839.

[18] Louis Narmour et al. *Simplification of Polyhedral Reductions in Practice*. 2024. arXiv: 2411.17498 [cs.PL]. URL: https://arxiv.org/abs/2411.17498.

[19] Louis-Noël Pouchet et al. “Loop transformations: convexity, pruning and optimization”. In: vol. 46. 1. New York, NY, USA: Association for Computing Machinery, Jan. 2011, 549–562. DOI: 10.1145/1925844.1926449. URL: https://doi.org/10.1145/1925844.1926449.

[20] Patrice Quinton and Vincent van Dongen. “The mapping of linear recurrence equations on regular arrays”. In: *Journal of VLSI signal processing systems for signal, image and video technology* 1.2 (1989), pp. 95–113. ISSN: 0922-5773. DOI: 10.1007/BF02477176. URL: https://doi.org/10.1007/BF02477176.

[21] Sanjay V. Rajopadhye. “Simplifying Dependent Reductions”. In: 2021. URL: https://api.semanticscholar.org/CorpusID:231766485.

[22] Sanjay V. Rajopadhye, S. Purushothaman, and Richard M. Fujimoto. “On synthesizing systolic arrays from Recurrence Equations with Linear Dependencies”. In: *Foundations of Software Technology and Theoretical Computer Science*. Ed. by Kesav V. Nori. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 488–503. ISBN: 978-3-540-47239-1.

[23] Xavier Redon and Paul Feautrier. "Scheduling reductions". In: *Proceedings of the 8th International Conference on Supercomputing*. ICS '94. Manchester, England: Association for Computing Machinery, 1994, 117–125. ISBN: 0897916654. DOI: 10.1145/181181.181319. URL: https://doi.org/10.1145/181181.181319.

[24] Philip Resnik and Eric Hardisty. *Gibbs Sampling for the Uninitiated*. Technical Report. Accessed: 2025-02-17. University of Maryland, 2010. URL: http://hdl.handle.net/1903/10058.

[25] T.F. Smith and M.S. Waterman. "Identification of common molecular subsequences". In: *Journal of Molecular Biology* 147.1 (1981), pp. 195–197. ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(81)90087-5. URL: https://www.sciencedirect.com/science/article/pii/0022283681900875.

[26] Sven Verdoolaege. "isl: An Integer Set Library for the Polyhedral Model". In: *Mathematical Software – ICMS 2010*. Ed. by Komei Fukuda et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 299–302. ISBN: 978-3-642-15582-6.

[27] Benjamin W. Wah and Guo-jie Li. "Systolic processing for dynamic programming problems". In: *Circuits, Systems and Signal Processing* 7.2 (1988), pp. 119–149. ISSN: 1531-5878. DOI: 10.1007/BF01602094. URL: https://doi.org/10.1007/BF01602094.

[28] Doran K. Wilde. *A Library for doing polyhedral operations*. Tech. rep. RR-2157. inria-00074515. INRIA, 1993.

[29] Cambridge Yang, Eric Atkinson, and Michael Carbin. "Simplifying dependent reductions in the polyhedral model". In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434301. URL: https://doi.org/10.1145/3434301.

[30] Yi-Qing Yang, Corinne Ancourt, and François Irigoin. "Minimal data dependence abstractions for loop transformations: Extended version". In: vol. 23. 4. 1995, pp. 359–388. DOI: 10.1007/BF02577771. URL: https://doi.org/10.1007/BF02577771.

[31] Tomofumi Yuki et al. *AlphaZ: A System for Design Space Exploration in the Polyhedral Model*. Tech. rep. Berlin, Heidelberg, 2013, pp. 17–31.

Appendix A

Scheduled Code Generator

In this appendix, we discuss our implementation of a code-generator built as a part of this work. Specifically, we give a brief overview of Alpha, the language of the AlphaZ compiler, then we describe the dependence graph, the data structure from which we derive the schedule, and finally we discuss the implementation followed by some performance evaluation.

A.1 Alpha

Alpha is a polyhedral equational language, and AlphaZ is a compiler for it. AlphaZ has been developed jointly at INRIA and Colorado State University. The compiler admits programs in the form of specifications of polyhedral equations. Users can then perform a number of optimizing transformations on their input program and generate C code as an output. This compiler is where we provide our implementation of dependent reduction simplification. The language itself is declarative, and allows for a straightforward implementation of many mathematical equations required in scientific computing. In the language, reductions are first class operators and a number of the aforementioned transformations can be used on them. We provide an example of the prefix sum written in Alpha in Figure A.1, but fuller descriptions of the language can be found in [14, 31].

A.2 Dependence Graph

The (reduced) dependence graph it is a directed multigraph with a node for each variable in the program [15, 31]. In this graph, there is an edge when a variable needs data from another in consumer/producer relationship. Each edge is associated with an affine map that characterizes this dependence relation as this map connects points in our domain to the points they require to compute. Specifically it maps between context domains. Since our variables can be broken up into different cases and perform different computations at each point in its domain, we restrict the

```
affine prefix [N] -> {: N > 0}
    outputs
        X: {[i]: 0 <= i <= N}
    let
        X[i] = case {
            {: i = 0 }: 10[];
            {: 0 < i <= N}: reduce(+, [j], {: 0 <= j < i}: X[j]);
        };
.
```

Figure A.1: Here is an example alpha program for the dependent prefix sum. We see there is not a lot of translation that needs to happen between our equations and the Alpha program.

domain for our map to only the points relevant to that dependence, which are called the context domains. From the dependence graph we can construct our schedule constraints. In Figure A.2, we see an example dependence graph for the dependent prefix sum.

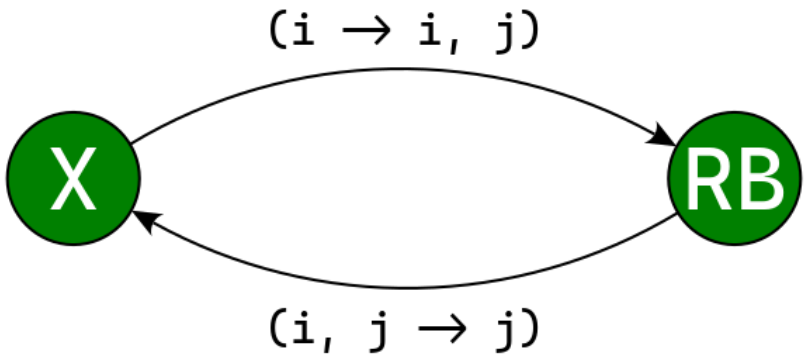

Figure A.2: RB in the above graph represents our reduction body. Associated with each arrow is the dependence mapping from one domain to another. In this case the top arrow arises from equation 3.4 and the bottom arrow comes from equation 3.5.

A.3 Scheduled Code Generation

As a part of this work, we implemented a scheduled code-generator. This takes an input Alpha program and transpiles it to C code. It provides an alternative code-generator to the a previous implementation of a demand driven code generator [11]. By creating a schedule and generating the code that respects that schedule, the generated code skips a number of repeated recursive calls that might arise in the demand driven code-generator. The demand driven code-generator creates a program that iterates through points in each variables' domain in an arbitrary order. This order may or may not respect the existing dependences in the program, therefore each computation must look up the values it depends on, hence the name demand driven. This is a memoized process so it either computes the value if it still has not been computed or reads a lookup table if it has been. This is a recursive process, so if the required point also requires another point, a similar look up will occur. While this is a sufficient implementation for obtaining working C code from an Alpha program, it is leads to many repeated function calls which, as we will see, can significantly slowdown the code.

The scheduled code generator requires more upfront processing but generates more efficient code. There are two parts to this implementation, first it must generate a legal schedule and then use that schedule to generate the loops. To accomplish this, it builds the dependence graph from the program AST, and then turns that into the appropriate form to pass to the ISL scheduler. The ISL Scheduler is a Parametric Integer Linear Programming solver and will generate a set of legal scheduling coefficients [26]. This schedule is then taken and used to generate the C code. Since legal schedules will respect the program dependences, there is no need to check whether a value will have been computed and removes the need for repeated function calls, replacing them with array look ups. This gives another completely automated pipeline for the generation of C code from the Alpha language using the AlphaZ compiler.

Table A.1: Performance comparison of the two code generators, the numbers in parenthesis represent the problem size. [4]

Program	Demand Driven (ms)	Scheduled (ms)	Times Speedup
LU Decomposition (1000)	2639.0548	1534.0212	1.72
3D Heat Equation (100, 100)	16330.505	4585.5388	3.56
Matrix vector product (1000)	3.8378	3.7423333	1.03
Jacobi 1D Stencil (1000, 1000)	35.4967	12.4099	2.86
Jacobi 2D Stencil (100, 1000)	626.5523	219.6664	2.85
TRMM (1000)	2184.7307	2162.3622	1.01
Floyd Warshall (100)	29.2229	9.1801	3.18
SYR2k (100, 1000)	35.9442	34.5156	1.04
SYMM (100, 100)	3.1727	2.8245	1.12
COV (1000, 100)	61.5399	33.3575	1.84

A.4 Performance Evaluation

In order to compare our code generator with the original code generator implemented in AlphaZ, we ran both on a number of programs from the PolyBench benchmark of programs. These are mostly linear algebra and stencil computations. As stated previously we would expect the scheduled code to be more optimal by its very nature and the timings of these test programs demonstrate the performance gained by using the scheduled code generator. Each of these was run twelve times with the fastest and slowest values removed. The remaining ten were then averaged and the runtimes are presented in Table A.1. The table lists each of the programs run and the numbers next to them represent the parameters used for each. These parameters are specific to each problem but in general represent the size of the input data or the number of steps to be performed.

As we can see from the table we were able to achieve an average speedup of 2.02, cutting the execution time in half on average. There is even more room for speed up as well as the scheduled code generator used in this comparison still generates some function calls that can be fully inlined, and the fully inlined version of this code would likely lead to even better performance increases.

[4]The experiments were run on 12th Gen Intel(R) Core(TM) i7-12700K with 20 cores

Appendix B

Full Gibbs Sampling Program

Before giving the whole specification, we repeat the brief description of each variable. These equations model the update that occurs at each step of the Gibbs Sampling algorithm. Z is the input cluster assignment or the result of the previous iteration, whereas Z' represents a random sampling of the current distributions (which are derived from each S and C equation). Obs will be the observed value. The S variables represent the sums of the observations for each cluster. $S1$ will be the sum of observations in cluster 1 and $C1$ will be the count of the number of observations in that cluster. We have variables for the left and right (labeled L and R). The left is used to aggregate these values from the input or the previous sample, while the right variables read from the current distribution. The following is the full specification:

Figure B.1: The full Alpha program for the update step of the Gibbs Sampling algorithm

```
external sample(4)
affine GMMgs [N] -> {: 0 < N}
 inputs
  Z, obs: {[i]: 0 <= i <= N}
 outputs
  Z' : {[i]: 0 <= i < N}
 locals
  S0R : {[i]: 0 <= i < N}
  S0L : {[i]: 0 < i <= N}
  S1R : {[i]: 0 <= i < N}
  S1L : {[i]: 0 < i <= N}
  C0R : {[i]: 0 <= i < N}
  C0L : {[i]: 0 < i <= N}
  C1R : {[i]: 0 <= i < N}
  C1L : {[i]: 0 < i <= N}
 let
  S0R[i] = reduce(+, (i, j -> i), {: 0 <= i < j <= N}:
             if(Z[j] = 0[]) then obs[j] else 0[]);
  S0L[i] = reduce(+, (i, j -> i), {: 0 <= j < i <= N}:
             if(Z'[j] = 0[]) then 1[] else 0[]);
  S1R[i] = reduce(+, (i, j -> i), {: 0 <= i < j <= N}:
             if(Z[j] = 1[]) then obs[j] else 0[]);
  S1L[i] = reduce(+, (i, j -> i), {: 0 <= j < i <= N}:
             if(Z'[j] = 1[]) then 1[] else 0[]);
  C0R[i] = reduce(+, (i, j -> i), {: 0 <= i < j <= N}:
             if(Z[j] = 0[]) then 1[] else 0[]);
  C1R[i] = reduce(+, (i, j -> i), {: 0 <= i < j <= N}:
             if(Z[j] = 1[]) then 1[] else 0[]);
  C1L[i] = reduce(+, (i, j -> i), {: 0 <= j < i <= N}:
             if(Z'[j] = 1[]) then 1[] else 0[]);
  C0L[i] = reduce(+, (i, j -> i), {: 0 <= j < i <= N}:
             if(Z'[j] = 0[]) then 1[] else 0[]);
  Z'[i] = case {
   {: i = 0}: sample(C1R[i], C0R[i], S0R[i], S1R[i]);
   {: 0 < i}: sample(C1R[i] + C1L[i],
                        S0R[i] + S0L[i],
                        C0L[i] + C0R[i],
                        S1L[i] + S1R[i]);
  };
.
```