Simplifying Multiple-Statement Reductions with the Polyhedral Model

by

Jianqiao (Cambridge) Yang

B.S., University of California (2017)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

Author . . . .
Department of Electrical Engineering and Computer Science
May 13, 2020

Certified by . . . .
Michael Carbin
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Simplifying Multiple-Statement Reductions with the Polyhedral Model

by

Jianqiao (Cambridge) Yang

Submitted to the Department of Electrical Engineering and Computer Science
on May 13, 2020, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Reduction – an accumulation over a set of values, using an associative and commutative operator – is a common computation in many numerical computations, including scientific computations, machine learning, computer vision, and financial analytics.

Contemporary polyhedral-based techniques make it possible to optimize reductions, such as prefix sum, in which each component of reduction's output potentially shares computation with another component in the reduction. Therefore an optimizing compiler can identify the computation shared between multiple components and generate code that computes the shared computation only once.

These techniques, however, do not support reductions that – when phrased in the language of the polyhedral model – span multiple statements. In such cases, existing approaches can generate incorrect code that violates the data dependencies of the original, unoptimized program.

In this work, we identify and formalize the multiple-statement reduction problem as a bilinear optimization problem. We present a heuristic optimization algorithm for these reductions, and we demonstrate that the algorithm provides optimal complexity for a set of benchmark programs from the literature on probabilistic inference algorithms, whose performance critically rely on simplifying these reductions. Specifically, the complexities for 10 of the 11 programs evaluated improve siginifcantly by factors at least of the sizes of the input data, which are in the range of 10^4 to 10^6 for typical real application inputs. We also confirm the significance of the improvement by showing that the speedups in wall-clock time ranges from 30x to over 10^8x.

Thesis Supervisor: Michael Carbin
Title: Assistant Professor of Electrical Engineering and Computer Science

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A *reduction* – an accumulation over a set of values, using an associative and commutative operator – is a common computation in many numerical computations, including scientific computations, machine learning, computer vision, and financial analytics.

For example, consider the prefix sum (PS) defined mathematically by Equation (1.1): the value at each index i of the array B is the summation of values at indices j before and up to i of array A. Listing 1.1 presents a direct translation of Equation (1.1) to an imperative language with loops. The complexity of Listing 1.1 is $\mathcal{O}(N^2)$: $\mathcal{O}(N)$ for iterating over "$\forall i$", and $\mathcal{O}(N)$ for the summation over j.

$$B[i] = \sum_{j=0}^{j \leq i} A[j] \quad \forall i, 0 \leq i < N \tag{1.1}$$

```
// B is array of ints initialized to all 0
for(i = 0; i < N; i++)
 for(j = 0; j<=i; j++)
  B[i] += A[j]
```

Listing 1.1: Naive PS

```
B[0] = A[0]
for(i = 1; i < N; i++)
 B[i] = B[i−1] + A[i]
```

Listing 1.2: Optimized PS

Optimized Reductions. Listing 1.2 presents a more efficient computation implementation of PS. The complexity of the implementation in Listing 1.2 is $\mathcal{O}(N)$, which is a linear speedup over the naive implementation Listing 1.1. The implementation achieves this speedup by exploiting the fact that consecutive iterations of the loop overlap in their computations. Specifically, for any pair of consecutive iterations, the latter iteration includes the entirety of the former iteration's computation. Therefore, that shared computation only needs to be computed once.

Gautam and Rajopadhye [2006] in the polyhedral model community formalized the above optimizing transformation under array equational language which supports reductions as a first class operation [Yuki et al., 2013], and proposed a set of techniques called *Simplifying Reductions* (SR). The core of SR is called *Simplification Transformation* (ST). At a high level, ST is a transformation in array equational language. ST takes in pointer to a statement with reduction (e.g. Equation (1.1)), and a directed vector along which the reduction's body (e.g. $A[j]$) presents *reuse* as inputs: here *reuse* means that the reduction's body evaluates to the same value along the direction of the vector. Given the inputs, ST transforms the statement in consideration into a set of statements that together is semantically equivalent to the original statement, but exploits the reuse vector to reduce complexity. For example, given Equation (1.1) and a reuse vector $[1, 0]^T$, which satisfies that changing i to $i + 1$ and j to $j + 0$ (i.e. not changing j) does not change the evaluation of $A[j]$, ST outputs Equation (1.2). Translating from Equation (1.2) to an imperative language with loops then produces Listing 1.2.

$$B[0] = A[0] \tag{1.2a}$$

$$B[i] = B[i - 1] + A[i] \quad \forall i, 1 \leq i < N \tag{1.2b}$$

Note that for one application of ST there are usually infinitely many choices for directions that present reuse. For instance, any vector $[c, 0]^T$ with constant c is a valid choice for reuse vector for Equation (1.1), since they all satisfy that changing from i to $i+c$ and not changing j does not change the evaluation of $A[j]$. As a concrete example, applying ST to Equation (1.1) with direction $[-1, 0]^T$ produces Equation (1.3). In Equation (1.3), instead of initializing $B[0]$ and computing $B[i]$s from lower indices to higher indices (i.e. left to right) as in Equation (1.2), Equation (1.3) initializes $B[N-1]$ and computes $B[i]$s from higher to lower indices (i.e. right to left). Complexity of the translation of Equation (1.3) to an imperative language with loops (Listing A.1 in Appendix A) is also $\mathcal{O}(N)$.

$$B[N-1] = \sum_{j=0}^{j<N} A[j] \tag{1.3a}$$

$$B[i] = B[i+1] - A[i] \quad \forall i, 0 \leq i < N-1 \tag{1.3b}$$

Multiple Statement Reductions. However, the SR framework, including ST, proposed by Gautam and Rajopadhye [2006] is restricted to optimizing one single reduction at a time, and it does not consider multiple inter-dependent statements. This is problematic because 1) ST application introduces new dependencies, and 2) the new dependencies introduced by STs together with existing dependencies of the input program may form dependency cycle(s) in the resultant program. To see 1), Equation (1.2b) introduces the dependency from $B[i]$ to $B[i-1]$, i.e., $B[i]$ must be computed after the $B[i-1]$ for any $i \in [1, N)$. To see 2), consider Equation (1.4): here we extended Equation (1.4a) (same as Equation (1.1)) by Equation (1.4b) and obtained a program with multiple statements. If we apply ST to Equation (1.4a) with the reuse vector $[-1, 0]^T$, we will get the program consisting of three statements: Equations (1.3a), (1.3b) and (1.4b), which contains dependency cycles. For example, using "$\xrightarrow{eq}$" to mean a dependency induced by statement eq, we note that the path $B[N-1] \xrightarrow{eq.\ (1.3a)} A[N-1] \xrightarrow{eq.\ (1.4b)} B[N-2] \xrightarrow{eq.\ (1.3b)} B[N-1]$ forms a cycle.

$$B[i] = \sum_{j=0}^{j\leq i} A[j] \quad \forall i, 0 \leq i < N \tag{1.4a}$$

$$A[i+1] = f(B[i]) \quad \forall i, 0 \leq i < N-1 \tag{1.4b}$$

On the other hand, if we apply ST to Equation (1.4a) with reuse vector $[1, 0]^T$, we will get the program consisting of three statements: Equations (1.2a), (1.2b) and (1.4b), which is a valid program without dependency cycle. Listing 1.3 presents a translation of this program to an imperative language with loops, and it correctly computes array A and B with complexity $\mathcal{O}(N)$.

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

Listing 1.3: Optimized PS with multiple statements

In summary to the above observations, the key challenge of optimizing multiple inter-dependent statements with reductions is to consolidate ST with dependency satisfaction.

Contributions. In this work, we term the pattern in Equation (1.4) a *multiple-statement* reduction. We present a new technique to automatically optimize multiple-statement reductions while soundly handling potentially inter-statements dependencies and therefore can automatically generate the code in Listing 1.3. The key idea behind our approach is that our heuristic algorithm uses the original program's affine schedule as a guide to choose among the multiple choices that can be made during the optimization process. One of our key results is that we show that even though the algorithm does not consider other viable choices during optimization, given an affine schedule of the original program and all left-hand-side arrays of reductions, the algorithm is still optimal for reductions with operators that have inverses. To that end, in this work, we present the following contributions:

- We identify multiple-statement reductions, which were not addressed in the Simplifying Reduction [Gautam and Rajopadhye, 2006] where only a single statement was considered.
- We formalize the task of optimizing a multiple statement reduction by combining the insights of the Simplifying Reduction framework with insights from ILP scheduling Pouchet et al. [2011]. We formulate a specification of the problem as a integer bilinear program.

- We propose a heuristic algorithm to solve the above optimization problem.
- We evaluate our proposed method on benchmark suites consisting of standard probabilistic inference algorithms and probabilistic models. Our results show that our approach reduces the complexity of the reductions in our programs to their optimal complexity for all of the 11 programs evaluated. In each 10 out of the 11 programs, the complexity improves by a (multiplicative) factor of at least N, where N is the size of the input data [1]. This is significant because for typical real appplication inputs of the programs in consideration, N is in the range of 10^4 to 10^6 – a factor that subsumes other potential constant factor improvements. We also confirm this significance by showing that the speedups in wall-clock time ranges from 30x to over 10^8x. We also outline the limits of the optimality of our approach, noting that our technique is not optimal if a reduction operator lacks an inverse operation.

In summary, multiple-statement reduction is a key ingredient of probabilistic inference algorithms, which are driving an emerging class of new programming languages and systems Gelman et al. [2015], Daniel Huang [2017], Goodman and Stuhlmüller [2014], Bingham et al. [2018], Cusumano-Towner et al. [2019], Narayanan et al. [2016], Mansingkha et al. [2018], Tran et al. [2017] designed to streamline science and enable new applications. Optimizing these algorithms has historically either been done by hand or has been baked in as a domain/algorithmic-specification optimization for a single problem model Holmes et al. [2012], Liu [1994]. To the best of our knowledge, our results are the first to identify and formulate multiple-statement reductions as a general program pattern, detail their challenges, and propose a technique to optimize their performance.

Road Map. In Chapter 2, We illustrate a heuristic algorithm to address the multiple-statement reduction described in Chapter 1. In addition, to further motivate the problem in the context of existing well-known algorithms, we present another motivating example which will be used for evaluation later in the paper. In Chapters 3

[1]For programs we consider, for example, this is usually the number of data points or the number of words of a text corpus. We include a more detailed review of input sizes for each benchmark in Section 8.1

and 4, we review backgrounds on polyhedral model and SR, respectively. In Chapter 5 we formalize our problem as a integer bilinear program. In Chapter 6 we introduce the proposed heuristic algorithm. In Chapters 7 and 8 we discuss the implementation of our proposed algorithm and its evaluation. In Chapters 9 and 10 we summarize some related work with concluding remarks.

Chapter 2

Example

2.1 Walk Through

In this section, we use the example of Equation (1.4) to 1) illustrate the steps of ST applications given reuse directions, 2) illustrate the valid ST application that leads to dependency cycles, and compare it to the valid ST application, using the algorithm proposed in Gautam and Rajopadhye [2006], and 3) describe the mechanism of our proposed heuristic algorithm, following the intuition we get from the comparison in 2).

Naive Prefix Sum. For ease of comparison and better visualization, we present the input in Equation (1.4) with Figure 2-1, a visual, polyhedral interpretation of the naive prefix sum program in Equation (1.4). In Figure 2-1, the top polyhedron with red dots represents the iteration domain of the reduction statement, $B[i] += A[j]$, with each red dot denoting an iteration instance of the statement. The bottom polyhedron with blue squares represents the iteration domain for the statement $A[i+1] = f(B[i])$. The middle polyhedron with orange diamonds is an additional polyhedron that our technique inserts into the program's polyhedral representation to denote the completion of each reduction B[i].

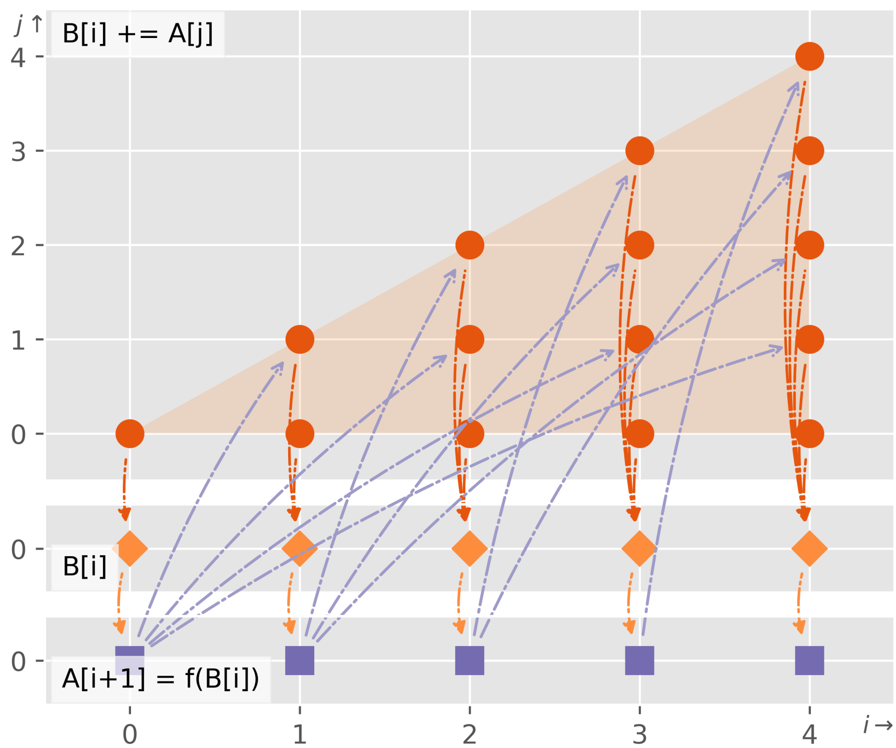


Figure 2-1: Naive prefix sum (Equation (1.4))

Data Dependencies. Each arrow in Figure 2-1 represents a data dependence between iteration instances. An arrow from iteration instance a to instance b represents a data dependence from a to b. The implication is that a needs to execute before b.

There are three sources of data dependencies:

- **Reduction.** Each point in the middle polyhedron depends on all the points in the respective column of the top polyhedron. These dependencies are those of the reduction.

- **Use.** Each point in the bottom polyhedron depends on the point in the respective column of the middle polyhedron. These dependencies are those from the use of the results of the reduction.

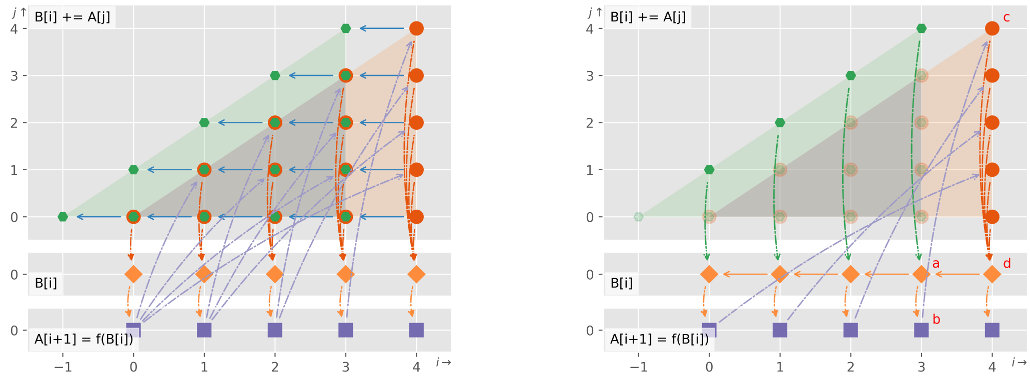
- **Update.** Points in each row of the top polyhedron depend on the point in the bottom polyhedron that is one to the left of the leftmost point of the row. These dependencies are those induced by the update to $A[i + 1]$ in Equation (1.4b) and use by Equation (1.4a).

Incorrect Optimization. Figure 2-2 presents two diagrams that illustrate an incorrect application of ST using Gautam and Rajopadhye [2006] ignoring the dependencies due to multiple-statement reduction.

Instead of using the correct reuse vector $[1, 0]^T$, this application uses the vector $[-1, 0]^T$. This vector maps iteration instances $[i, j]$ to instances $[i-1, j]$. The organization of the diagram Figure 2-2a shows a mapping from the red shaded polyhedron the green shaded polyhedron, with each solid blue arrow represents the mapping between instances of the corresponding polyhedron. The green polygons outlined in red circles are the intersection of the two polyhedrons. Note that because the reuse vector has property that the evaluation of reduction body $A[j]$ are the same for any two points in the same row, the evaluation of a reduction over any column *col* in this intersection triangle must have the same value as the evaluation of a reduction over the column to the left of *col*. Therefore the intersection part of the domain will be eliminated eliminated by ST, by reusing previously computed reductions (i.e. compute $B[i]$ from $B[i-1]$ by incrementalizing using points not in the intersection). Figure 2-2b shows the pruned digram from Figure 2-2a. The top polyhedron now has the red circles at the rightmost column and green polygon dots along the hypotenuse of the shifted domain [1].

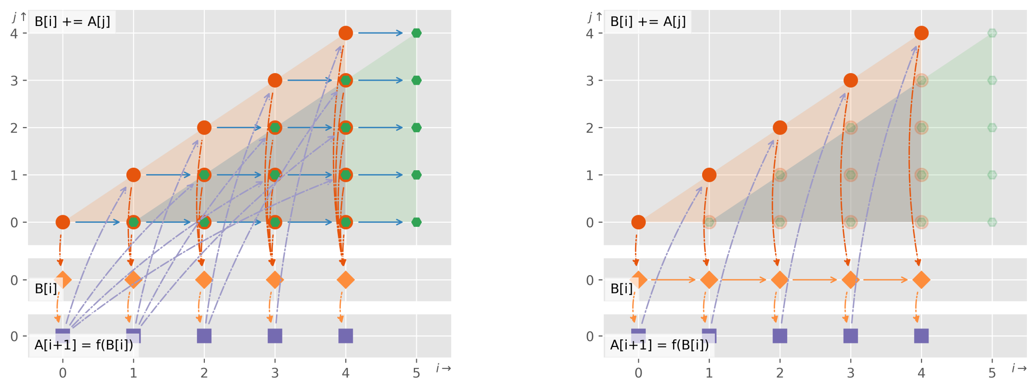
Point d and the red circles column in the top polyhedron in Figure 2-2b correspond to the reduction that initializes B[N-1] in Equation (1.3a). All points in the middle polyhedron except d then correspond to Equation (1.3b), i.e., each B[i] is computed by subtracting the successor point B[i+1] by A[i]. Dependencies in Figure 2-2b are preserved from Figure 2-2a, with the newly introduced dependency along the reuse vector in the middle polyhedron, which is represented by solid orange arrows pointing to the left. However, as mentioned in Chapter 1, Figure 2-2b's dependencies form cycles; for instance, points a, b, c, d forms a cycle. Therefore, the transformed program in Figure 2-2b does not have a valid schedule, and consequently the application of

[1] hypotenuse limited to the domain of projected domain of the reduction (i.e. does not include point $(-1, 0)$)



(a) After shift (b) After transformation

Figure 2-2: Incorrect optimization of prefix sum with multiple statement dependency



(a) After shift (b) After transformation

Figure 2-3: Correct optimization of prefix sum with multiple statement dependency

ST along the reuse vector with mapping $[i, j] \to [i-1, j]$ produces an incorrect optimization.

Correct Optimization. Figure 2-3 presents two diagrams, corresponding to the two steps to correctly applying ST in Gautam and Rajopadhye [2006], respectively. Figure 2-3a illustrates the first step of the algorithm, where the algorithm chooses a reuse vector and shifts the reduction statement's iteration domain along the vector. That is, Figure 2-3a illustrates the shift along the reuse vector, $[1, 0]^T$, which maps iteration instances $[i, j]$ to instances $[i+1, j]$. As shown in this figure, this corresponds to the mapping of top polyhedron, colored in red, to its shifted counterpart, colored in green. Each solid blue arrow represents the mapping from an instance in the red polyhedron to its counterpart in the green polyhedron. The green polygons outlined in red circles are again the points in the intersection of the two polyhedrons, which, same as the previous example of incorrect ST application, will be eliminated by ST.

Figure 2-3b corresponds to the resulting polyhedron and dependence structure after ST eliminates redundant computations, by applying the correct ST with reuse vector $[1, 0]^T$ to Equation (1.4) (Equations (1.2a), (1.2b) and (1.4b)). Each instance in the intersection of the two polyhedrons has been eliminated, along with its induced dependencies. The middle polyhedron also has new dependence edges: an edge has been added between reduction instances along the direction of the reuse vector. This polyhedron denotes the iteration domain of Line 3 in Listing 1.3. Each new dependence edge therefore reflects that each `B[i]` is computed from `B[i - 1]`.

Heuristic for choosing a valid direction. As we have seen from the previous illustration, it is important to choose a valid reuse vector with multiple-statement reductions. In this work, we propose a heuristic algorithm for choosing a valid reuse vector. Notably, one key difference between Figures 2-2 and 2-3 is the dependencies drawn on the middle polyhedron. Specifically, in the middle polyhedron of Figure 2-3, the drawn dependencies on $\mathtt{B}[i]$ respects the scheduled computation order of $\mathtt{B}[i]$ of the original program in Figure 2-1, whereas that of Figure 2-2 disobeys that scheduled order. This observation has inspired the heuristic algorithm that always chooses the reuse vector that is consistent with the scheduled computation order of the LHS of the reduction. We show that the reuse vector chosen with this algorithm is 1) always sound, and 2) guarantees optimality if each reduction operator in the target program has an inverse.

2.2 Motivating Example

Specification and Implementation. Consider the following specification of *Gibbs Sampling [Geman and Geman, 1984] on a two-cluster Gaussian Mixture Model [see for example, Murphy, 2012]* (GS-2GMM). The input to GS-2GMM is a float array

Obs that represents the observations. Informatively, the two-cluster Gaussian Mixture Model (GMM) assumes that each single observation belongs to one of the two clusters, and that each cluster follows a Gaussian distribution. The goal of GS-2GMM is to sample the the array Z that represents the cluster membership of each of the given the observations, following the desired GMM distribution. It achieves this goal by iteratively taking in an old cluster assignment for each observation in turn, and samples a new one by updating the assignments of the remaining observations. This process will produce a stream of Zs which will respect the distribution of cluster assignments. The mathematical specification of GS-2GMM is given in Equation (2.1).

```
int[N] C0L, C1L, C0R, C1R = {0...} // Zero initialize
float[N] S0L, S1L, S0R, S1R = {0...} // Zero initialize
for(i = 1; i < N; i++)
  C0R[0] += (Z[i] == 0 ? 1 : 0)
  C1R[0] += (Z[i] == 1 ? 1 : 0)
  S0R[0] += (Z[i] == 0 ? Obs[i] : 0)
  S1R[0] += (Z[i] == 1 ? Obs[i] : 0)
for(i = 0; i < N; i++)
  // Sample according to Equations (2.1d) to (2.1f)
  Z'[i] = sample(C0L[i] + C0R[i], C1L[i] + C1R[i],
                 S0L[i] + S0R[i], S1L[i] + S1R[i])
  // Incremental updates
  C0L[i] = C0L[i−1] + (Z'[i] == 0 ? 1 : 0)
  C1L[i] = C1L[i−1] + (Z'[i] == 1 ? 1 : 0)
  S0L[i] = S0L[i−1] + (Z'[i] == 0 ? 1 : 0)
  S1L[i] = S1L[i−1] + (Z'[i] == 1 ? 1 : 0)
  C0R[i] = C0R[i−1] − (Z[i] == 0 ? 1 : 0)
  C1R[i] = C1R[i−1] − (Z[i] == 1 ? 1 : 0)
  S0R[i] = S0R[i−1] − (Z[i] == 0 ? Obs[i] : 0)
  S1R[i] = S1R[i−1] − (Z[i] == 1 ? Obs[i] : 0)
```

Listing 2.1: Correct optimized GS-2GMM with multiple-statement dependency

$$P_o(z,i) \overset{abbrev.}{=} P(obs_i|obs_{\setminus i}, Z_{\setminus i}, Z_i = z) \tag{2.1a}$$

$$C_{zi} = \sum_{\forall j \ s.t. j \neq i \wedge Z_j = z} 1 \quad , \forall z, i \tag{2.1b}$$

$$S_{zi} = \sum_{\forall j \ s.t. j \neq i \wedge Z_j = z} obs_i \quad , \forall z, i \tag{2.1c}$$

$$P_o(z,i) = \mathcal{N}(\frac{S_{zi}}{C_{zi}}, (1 + C_{zi})^{-1} + 1) \tag{2.1d}$$

$$P(Z_i = 0|Z_{\setminus i}, obs) = \frac{P_o(0,i)}{P_o(0,i) + P_o(1,i)} \tag{2.1e}$$

$$Z_i \sim P(Z_i|Z_{\setminus i}, obs) \quad \forall i \in [1, N] \tag{2.1f}$$

In Equations (2.1b) and (2.1c), C_{0i} and S_{0i} represent the counts and sums, respectively, of all the observations except the one with index i, for which the current old assignment of cluster membership is 0 (and similarly for C_{1i}, S_{1i}, with membership of 1). Then a distribution $P(Z_i|Z_{\setminus i}, obs)$ is defined by Equations (2.1d) and (2.1e). In this example, the distribution is simply tuple of two floats representing the weights of assiging Z_i to cluster zero or one, respectively. Note that the exact computation required to produce this tuple is not important for understanding the example. The key information is that they are produced by a deterministic pure function that depends only on the counts and sums defined above. Lastly, Equation (2.1f) samples each Z_i in order from this distribution.

Our Approach. Our approach helps to automate the translation from Equation (2.1) to Listing 2.1. For conciseness of presentation, we take the variable S_{zi} and fix $z = 0$ as an example. In this case Equation (2.1c) can be rewritten as sum of two variables $S_{0i} = \texttt{S0L}[i] + \texttt{S0R}[i]$, where $\texttt{S0L}, \texttt{S0R}$ are given by Equations (2.2a) and (2.2b), respectively. The step of rewriting in terms of S0L and S0R is standard in polyhedral model compilation: the original domain with constraint $j \neq i$ is non-convex and it's standard to break it into two convex polyhedrons with constraints $j < i$ and $j > i$. Further, we make the non-affine constraint $Z_j = z$ into an simple if-then-else expression guarding the reduction's body – this is standard and same as the approach proposed by Benabderrahmane et al. [2010] to model non-affine constraints as control predicates.

$$\texttt{S0L}[i] = \sum_{j=0}^{j<i} (\texttt{Z}[j] == 0 \ ? \ \texttt{Obs}[j] : 0) \tag{2.2a}$$

$$\texttt{S0R}[i] = \sum_{j=i+1}^{j<N} (\texttt{Z}[j] == 0 \ ? \ \texttt{Obs}[j] : 0) \tag{2.2b}$$

$$\vdots \quad \text{Other equations...}$$

$$\texttt{Z'}[i] = \texttt{sample}(\texttt{S0L}[i] + \texttt{S0R}[i], ...) \tag{2.2c}$$

Note that Equations (2.2a) and (2.2c) exactly correspond to Equations (1.4a) and (1.4b) respectively. Therefore the technique walked through in Section 2.1 also applies to Equations (2.2a) and (2.2c) for producing efficient complexity specification. Further, our technique is general in that it handles any multiple-statement reduction, including Equation (2.2b) which has constraints $i + 1 \leq j < N$. Lastly, the same analysis can be applied to all cases of C_{zi} and S_{zi} with either $z = 0$ or $z = 1$. The analyses in total produces eight variables, namely `C0L`, `C1L`, `C0R`, `C1R`, `S0L`, `S1L`, `S0R`, `S1R`, which applying our technique and after compiling to exectuable code, produces Listing 2.1.

Discussion. The mathematical formulations in Equation (2.1) are commonly found in fields such as statistics, scientific computing, machine learning, computer vision, molecular biology, and financial analytics [2]. A specification like Equation (2.1) is commonly implemented as an executable program, for example in Listing 2.1. Although Equation (2.1)'s mathematical formulation allows for easier reasoning for humans (e.g. researchers in the above mentioned fields), its executable implementation in Listing 2.1 is very different. Therefore one important and practical research problem

[2] Equation (2.1) is transcribed from Murphy [2012, Section 24.2.4.1], by assuming a simplified version of GMM with only two clusters.

is how do we translate from the mathematical formulations such as Equation (2.1) to executable implementations such as Listing 2.1 automatically. With this goal of automatic translation achieved, researchers in the above fields will only need to specify algorithms in a conceptual high-level DSL similar to Equation (2.1), and then a compiler will automatically generate efficient implementation with complexity comparable to that of a manual implementation. This has many potential benefits including implementation correctness gaurantee and faster iteration of algorithm design. Indeed, many existing and ongoing work in the probabilistic programming community [Goodman et al., 2008, Daniel Huang, 2017, Goodman and Stuhlmüller, 2014, Bingham et al., 2018, Plummer, 2015, Narayanan et al., 2016, Mansingkha et al., 2018, Atkinson et al., 2018] allow the user to code in high level DSLs. Though the details of these systems is out of scope of discussion in this work, one motivation from a practical perspective is the potential of our method to be integrated into these systems for generating efficient complexity code.

Chapter 3

Background: Polyhedral Model

For completeness, we review terminologies in the polyhedral model that we use in this work.

3.1 Polyhedral Set Representation

We use the following definition and notation for a polyhedral set; the notation is consistent with integer set library (isl) [Verdoolaege, 2010]'s notation.

Definition 1 (*System of affine inequalities*)**.** A system of affine inequalities is defined as $A \cdot [\vec{x}, 1]^T \geq \vec{0}$: A is an $m \times (n+1)$ constant integer matrix and $\vec{x}$ is length n vector of integer unknowns.

Remark 1. We may also express a system of affine inequalities by conjunction of simple affine inequalities. For example, the system $\begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ N \\ 1 \end{bmatrix} \geq \vec{0}$ is equivalent to $(x \geq 0) \wedge (x \geq N - 1)$ — or simply the short hand $0 \leq x < N$. A simple equality $x = 0$ is short hand for the conjunction of two inequalities $(x \geq 0) \wedge (-x \geq 0)$.

Definition 2 (*Polyhedral set*)**.** A polyhedral set $\mathcal{P}$, defined as $[\vec{p}] \rightarrow \{[\vec{x}] : A \cdot [\vec{x}, \vec{p}, 1]^T \geq 0\}$, contains a tuple of parameters $[\vec{p}]$, a tuple template $[\vec{x}]$ and a system of affine inequalities $A \cdot [\vec{x}, \vec{p}, 1]^T \geq \vec{0}$, where A is a matrix of coefficients. We say $[\vec{p}] \rightarrow \{[\vec{x}]\}$ is the space of $\mathcal{P}$.

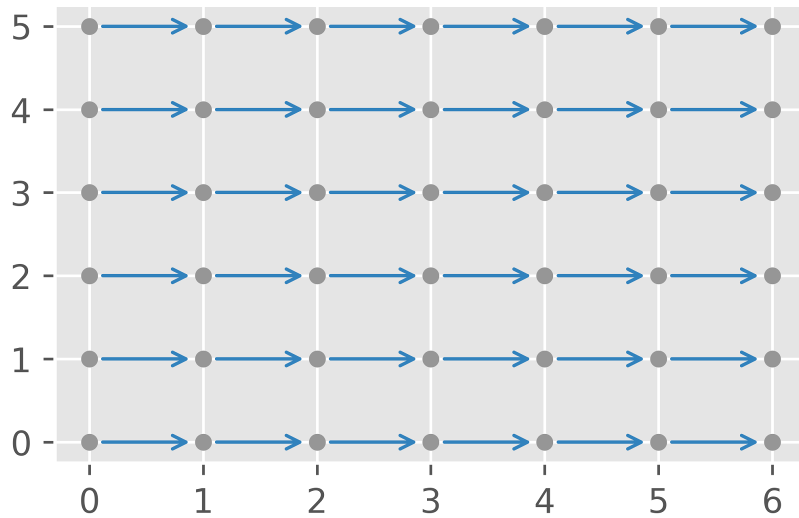


Figure 3-1: Plot of example polyhedral relation

For example, $[N] \to \{[i] : 0 \leq i < N\}$ denotes the set of integers from 0 to $N-1$. The space of this set is $[N] \to \{[i]\}$.

Definition 3 (*Polyhedral relation*)**.** A polyhedral relation $[\vec{p}] \to \{[\vec{x_1}] \to [\vec{x_2}] : A \cdot [\vec{x_1}, \vec{x_2}, \vec{p}, 1]^T \geq \vec{0}\}$ contains a tuple of parameters $[\vec{p}]$, tuple templates $[\vec{x_1}], [\vec{x_2}]$ and a system of affine inequalities $A \cdot [\vec{x_1}, \vec{x_2}, \vec{p}, 1]^T \geq \vec{0}$.

For example, $[N] \to \{[i, j] \to [i+1, j] : 0 \leq i < N, 0 \leq j < N\}$ denotes the relation that maps every integer tuple $[i, j]$ to $[i+1, j]$ within an N-by-N grid. Figure 3-1 visualizes this relation with $N = 5$: the blue arrows map points corresponding to integer tuples to their right successors.

Semantically, the polyhedral set provides an intensional description for a set of tuples, templated by $[\vec{x}]$, so that all tuples in the set satisfy the system of affine inequalities. The set is optionally parametric to $[\vec{p}]$, if $[\vec{p}]$ is not empty.

Similarly, a polyhedral relation describes a set of binary relations mapping from $[\vec{x_1}]$ to $[\vec{x_2}]$, for every $[\vec{x_1}]$-$[\vec{x_2}]$ pair that satisfies the system of affine inequalities; a polyhedral relation can also be parametric to $[\vec{p}]$.

For aesthetic reason, we omit parameter $[\vec{p}]$ when it is clear from the context which identifiers are parameters.

Definition 4 (*Face of polyhedral set*)**.** Let polyhedral set $\mathcal{P} = [\vec{p}] \rightarrow \{[\vec{x}] : A{\cdot}[\vec{x}, \vec{p}, 1] \geq \vec{0}\}$. Let M_i be the i-th row of matrix M. A face of $\mathcal{P}$ is defined as $\mathcal{F} = \mathcal{P} \cap \mathcal{B}$ where $\mathcal{B} = [\vec{p}] \rightarrow \{[\vec{x}] : B \cdot [\vec{x}, \vec{p}, 1] = \vec{0}\}$ and $\forall i \exists j, A_i = B_j$.

In words, a face of $\mathcal{P}$ is $\mathcal{P}$ with some subset of (potentially empty or all) inequalities of $\mathcal{P}$ changed to equality.

3.2 Polyhedral Representation of Program

The *polyhedral model* represents a program by a set of statements, and for each statement, an associating polyhedral set known as the statement's *domain*. Each point in a polyhedral set correspond to one concrete execution instance of the statement.

IR. Following formalization by the original SR work [Gautam and Rajopadhye, 2006, Yuki et al., 2013], we use an equation based representation of program in this work, presented in grammar by Listing 3.1.

```
<prog> := <stmt>+
<stmt> := LHS[<afflist>] ( = | ⊕= ) <expr> : 𝒫
<expr> := <expr>? ⊕ <expr> | ARR[<afflist>] | CONST
<afflist> := (<aff> ,)∗ <aff>
```

Listing 3.1: IR Grammar

We explain each component in turn:

- prog a program consists of multiple statements.
- stmt a statement is left hand side (i.e. LHS<aff>), middle assignment operator (i.e. $=$ or $\oplus=$) , a right hand side expression (i.e. <expr>), and its domain (i.e. $\mathcal{P}$). A statement is a normal assignment statement when the middle assignment operator is plain $=$; a statement is a *reduction* when the middle assignment operator is $\oplus=$.
- expr an expression is either an unary or binary operator applied on expression(s), or an array reference (i.e. ARR[<aff>]).

- aff an affine expression is a kind of expression that applies affine transformation to variables and produces a scalar. It references only variables in $\vec{x}$ or $\vec{p}$, where $[\vec{p}] \to \{[\vec{x}]\}$ is the space of $\mathcal{P}$.

- afflist a list of affine expressions. Array references (i.e. LHS[<afflist>] and ARR[<afflist>]) must have indices that are an affine expressions. An <afflist> of length n can be expressed mathematically as an affine transformation $A \cdot [\vec{x}, \vec{p}, 1]^T$, where A is a constant $n \times (|\vec{x}| + |\vec{p}| + 1)$ integer matrix and $\vec{x}, \vec{p}$ defined same as those for <aff>.

- $\mathcal{P}$ a polyhedral set representing the statement's domain; since each point in the domain corresponds to one concrete execution instance of the statement, if $\mathcal{P}$ is $[p] \to \{[t] : e\}$, then p correspond to parameters of the program, t correspond the set of loop variables of the statement.

3.2.1 IR Semantics

Access Relation. An *access relation* is a polyhedral relation mapping from the space of a statement's domain to the space of an accessed array. An access relation can either b a *write access relation* (in case of LHS<afflist>), or a read access relation (in case of RHS<afflist>). Let ARR[<afflist>] be an array reference for a statement with space $[\vec{p}] \to \{[\vec{x}]\}$ and <afflist> expressed as $A \cdot [\vec{x}, \vec{p}, 1]$, the access relation for this array access is $[p] \to \{[\vec{x}] \to [\vec{y}] : A \cdot [\vec{x}, \vec{p}, 1] = \vec{y}\}$.

Reduction projection. If a statement is a reduction, we define the *projection* of the reduction *proj* as the write access relation of LHS array reference of the reduction.

SSA. Following Gautam and Rajopadhye [2006], our IR requires the program to be in *array static-single-assignment* (Array SSA) form[Feautrier, 1988]; that is, each array element is never written twice during program execution. To our IR, this means for each unique LHS array, and the statements $\mathrm{S}_0 ... \mathrm{S}_k$ that writes to it, $\bigcap_i \mathcal{W}_{\mathrm{S}_i} = \emptyset$, where $\mathcal{W}_{\mathrm{S}_i}$ is the write access relation for S_i.

Semantics. We use usual semantics from array languages [Yuki et al., 2013] for our IR. Specifically, a statement is evaluated under each point of its domain $\mathcal{P}$. An expression is evaluated under a point by substituting the free variables of the expression with the instantiated values of those variables under that point. For example, A[N - i + 1] evaluates to the value of $A[9]$ at point $[N] \rightarrow \{[i] : N = 10 \wedge i = 2\}$. If the statement is a normal assignment, for each point in $\mathcal{P}$ the right hand side expression is evaluated and assigned to the left hand side array. If the statement is a reduction, for each point $p \in \mathcal{P}$ the right hand side expression is evaluated, and its value is accumulated into LHS at point $p' = proj(p)$ using the operator $\oplus$ where *proj* is the projection of the reduction as defined previously.

3.3 Polyhedral Model Scheduling

Scheduling is a step in polyhedral model where a scheduling function assigns each point in a statement's domain a timestamp, denoting the order of all execution instances. This step is essential for multiple-statement programs because the timestamps are assigned to respect the inter-statement and intra-statement dependencies.

3.3.1 Scheduling Function

Definition 5 (Schedule Timestamp)**.** A schedule timestamp is an m-dimensional vector, where m is the upper bound on the dimension of the schedule. For two timestamps T_1 and T_2, $T_1 < T_2$ (T_1 happens before T_2) iff $T_1[i] < T_2[i]$ where i is the first non-equal index between T_1, T_2.

A schedule Θ for a program is a collection of *scheduling functions*, one for each statement. The scheduling function for a statement S is an affine transformation, represented by the matrix Θ_S, that maps statement S's domain to its schedule timestamp. For a statement S with domain in space $[\vec{p}] \rightarrow \{[\vec{x}]\}$, its m dimensional timestamp T_{S}

is given by the $m \times (|\vec{x}| + |\vec{p}| + 1)$ scheduling function Θ_S:

$$T_{\mathrm{S}} = \Theta_{\mathrm{S}} \cdot \begin{bmatrix} \vec{x} \\ \vec{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \theta_{1,1} & \dots & \theta_{1,|\vec{x}|+|\vec{p}|+1} \\ \vdots & \dots & \vdots \\ \theta_{m,1} & \dots & \theta_{m,|\vec{x}|+|\vec{p}|+1} \end{bmatrix} \cdot \begin{bmatrix} \vec{x} \\ \vec{p} \\ 1 \end{bmatrix} \tag{3.1}$$

3.3.2 Dependence Relation

Any two statements S, T must satisfy a dependence relation represented by a polyhedral relation $\mathcal{D}_{\mathrm{S,T}} = [\vec{p}] \to \{[\vec{x_s}] \to [\vec{x_t}] : D_{\mathrm{S,T}} \cdot \left[\vec{x_s}, \vec{x_t}, \vec{p}, 1\right]^T \geq \vec{0}\}$, and $D_{\mathrm{S,T}}$ is the *dependency matrix*. The dependence relation $\mathcal{D}_{\mathrm{S,T}}$ describes the happens before relation between iterations of S and T. For a pair of statements S, T, let S writes to LHS and T's RHS expression reads elements of LHS. The dependence relation $\mathcal{D}_{\mathrm{S,T}}$ is equal to $\mathcal{R}^{-1} \circ \mathcal{W}$, where $\mathcal{R}, \mathcal{W}$ are the read and write access relations for LHS of the two statements respectively, $\mathcal{R}^{-1}$ denotes the inverse of the polyhedral relation $\mathcal{R}$, and $\circ$ denotes composition. Previous work [Collard et al., 1995, Verdoolaege et al., 2013] and textbook [Verdoolaege, 2016] contains a detailed introduction to dependence analysis techniques, which we refer to the reader for a deeper exposure.

3.3.3 ILP formulation of scheduling

The task of scheduling a program in polyhedral model is to find a schedule Θ for the program such that the schedule timestamps for all statements instances satisfy the dependence relations of the program. Pouchet et al. [2011] formalized the scheduling problem for obtaining m-dimensional schedule as the following convex problem:

$$\forall \mathcal{D}_{S,T}, \forall k, \delta_k^{\mathcal{D}_{S,T}} \in \{0, 1\} \tag{3.2a}$$

$$\forall \mathcal{D}_{S,T}, \sum_{i=1}^{m} \delta_k^{\mathcal{D}_{S,T}} = 1 \tag{3.2b}$$

$$\forall \mathcal{D}_{S,T}, \forall k \in [1, m], \forall [\vec{x}_S, \vec{x}_T, \vec{p}] \in \mathcal{D}_{S,T} \tag{3.2c}$$

$$\Theta_S^k \cdot \begin{bmatrix} \vec{x}_S \\ \vec{p} \\ 1 \end{bmatrix} - \Theta_T^k \cdot \begin{bmatrix} \vec{x}_T \\ \vec{p} \\ 1 \end{bmatrix} \geq - \sum_{i=1}^{k-1} \delta_i^{\mathcal{D}_{S,T}} (K\vec{p} + K) + \delta_k^{\mathcal{D}_{S,T}} \tag{3.2d}$$

In words, the formulation creates a binary variable $\delta_k^{\mathcal{P}_{S,T}}$ for each $k \in [1, m]$ dimensions, and each pair of dependence relation in the program. The binary variable is used to model entry by entry comparison of an m dimensional timestamp. Constraint c) finally encodes that the schedule function Θ_S and Θ_T must satisfy that $\vec{x}_S$ is schedule before $\vec{x}_T$, if the dependence $\vec{x}_S \rightarrow \vec{y}_T$ exists — that is, $[\vec{x}_S, \vec{x}_T, \vec{p}]^T \in \mathcal{D}_{S,T}$. The variable K is a known constant obtainable from the original program, and is an upper bound modeling technique to make the problem convex. Pouchet et al. [2011] shows that this problem is equivalent to an ILP thanks to Farkas' Lemma [Schrijver, 1986], solving which produces the desired schedule coefficients Θ in Section 3.3.1

Chapter 4

Background: Simplifying Reduction Framework

Previous work Gautam and Rajopadhye [2006] introduced a core transformation called *simplification transformation* (ST) that can potentially transform a single statement specified in Listing 3.1 to lower its complexity, along with a set of enabling transformations: *reduction decomposition*, *same operator transformation*, *distributivity transformation* and *higher order operator transformation.* For the core transformation, we will use an example from Chapter 1 to illustrate the transformation. For the enabling transformations, we will include a brief description for each transformation. Finally, Gautam and Rajopadhye [2006] combines all the transformations to provide a dynamic programming algorithm to efficiently choose from an infinite set of configurations and orders for the transformations, a sequence of transformations that lead to optimal complexity reduction.

4.1 Simplification Transformation

Here we use take the example from Section 2.1 to illustrate the core simplification transformation that reduces complexity of a reduction. The full specification of ST can be found in Appendix B and Gautam and Rajopadhye [2006].

Listing 4.1 illustrates the example of applying ST (Section 3.2) to Equation (1.4)

$\texttt{S1} : \texttt{BTmp}[i] \mathrel{+}= \texttt{A}[j] : \{[i, j] : 0 \leq i < N \wedge 0 \leq j \leq i\}$
$\texttt{S1Fin} : \texttt{B}[i] = \texttt{BTmp}[i] : \{[i] : 0 \leq i < N\}$
$\texttt{S2} : \texttt{A}[i + 1] = \texttt{B}[i] : \{[i] : 0 \leq i < N\}$
$\Downarrow$
$\texttt{S1Add} : \texttt{BTmpAdd}[i] \mathrel{+}= \texttt{A}[j] : \{[i, j] : 0 \leq i < N \wedge i = j\}$
$\texttt{S1AddOnly} : \texttt{BTmp}[i] = \texttt{BTmpAdd}[i] : \{[i] : i = 0\}$
$\texttt{S1AddReuse} : \texttt{BTmp}[i] = \texttt{BTmp}[i - 1] + \texttt{BTmpAdd}[i] : \{[i] : 1 \leq i < N\}$
$\texttt{S1Fin} : \texttt{B}[i] = \texttt{BTmp}[i] : \{[i] : 0 \leq i < N\}$
$\texttt{S2} : \texttt{A}[i + 1] = \texttt{B}[i] : \{[i] : 0 \leq i < N\}$

Listing 4.1: ST in polyhedral IR for the example in Chapter 1 (Equation (1.4)), given a correct reuse vector $[1, 0]^T$

and producing the optimized version (Equations (1.2a), (1.2b) and (1.4b)) in our IR. As we mentioned before, core ST operates on single statement only, and produces correct result for multiple statements if a correct reuse vector is given.

Original Reduction. Specifically, Listing 4.1 presents the reduction in Equation (1.4a) in the polyhedral IR as the statement `S1` with domain $\mathcal{P} = [N] \to \{[i, j] : 0 \leq i < N \wedge 0 \leq j \leq i\}$ in Listing 4.1. The right hand side expression is $\texttt{A}[j]$, and i is not a bound variable – this means given a fixed j, the right hand side's values are the same for different values of i.

Optimized Reduction. The optimized prefix sum is symmetric to that in code, with the addition of a statement `S1Add`, which provides the contents of `BTmpAdd`. Specifically, `S1Add` is a polyhedral over the full space of i that sets `BTmpAdd[i]` to equal `A[i]`. `TAddOnly` next initializes `BTmp[0]` (as on Line 1 of Listing 1.3) and `S1AddReuse` incrementally computes the remaniningg values of `BTmp` (as on Line 3 of Listing 1.3).

Algorithm (Reuse Vector). To identify this optimization opportunity and generate the optimized code, the Simplifying Reduction transformation identifies a *reuse vector* by which shifting the original, unoptimized polyhedral ($\mathcal{P}$) makes plain that consecutive iterations of the polyhedral overlap and can therefore be incrementalized.

Consider the *reuse vector* $\vec{r} = [1, 0]^T$, that shifts all points $[i, j]$ to $[i + 1, j]$; $\vec{r}$ can also be represented by the polyhedral relation $\{[i, j] \rightarrow [i + 1, j] : \forall i, j\}$. The arrows in Figure 4-1 visualizes $\vec{r}$ over the domain of the original reduction (red dots in the shaded red triangle).

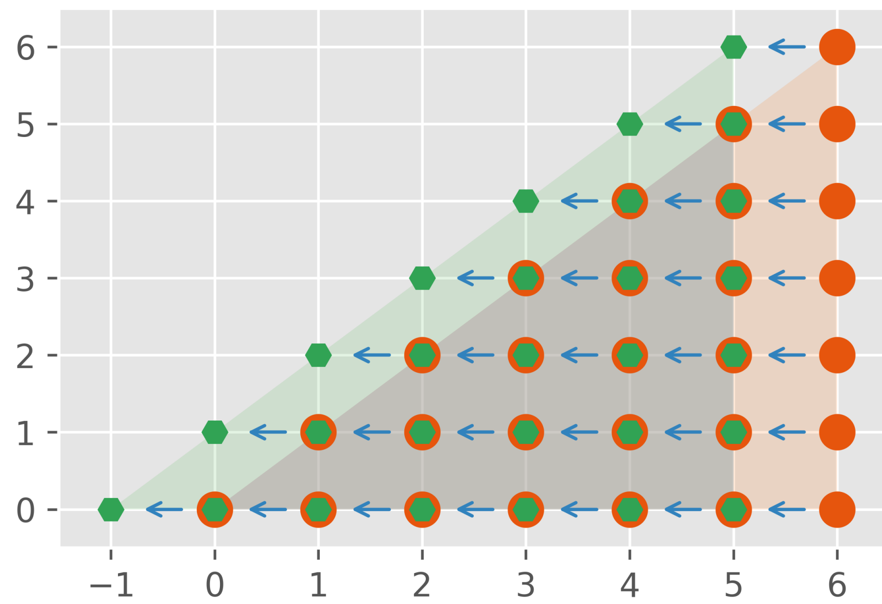


Figure 4-1: Visualization of algorithm on prefix sum example

Given this reuse vector, STthe Simplifying Reduction transformation performs the following steps:

- **Shift.** The transformation first shifts `T`'s polyhedral along the direction of the reuse vector, transforming $\{[i, j] : 0 \leq i < N \wedge 0 \leq j \leq i\}$ (red dots in Figure 4-1) into $\{[i, j] : 1 \leq i < N + 1 \wedge 0 \leq j \leq i - 1\}$ (green hexagon points in the shaded green triangle in Figure 4-1) .

- **Intersect.** The transformation next computes the intersection of the shifted polyhedral with its original polyhedral, yielding $\{[i, j] : 1 \leq i < N \wedge 0 \leq j \leq i - 1\}$ (overlapped points in the shaded purple triangle in Figure 4-1). This polyhedral denotes the subset of points of the original domain $\mathcal{P}$, whose value can be reused from the predecessor points as indicated by the reuse vector.

- **Project.** Finally, the transformation projects the result onto the space of polyhedral that represents the indices of left hand side array `S`. Concretely, the transformation applies the projection represented by the polyhedral relation $\{[i,j] \rightarrow [i] : \forall i,j\}$), yielding the polyhedral $\{[i] : 1 \leq i < N\}$.

This final polyhedral is exactly the domain of elements of S that exhibits reuse along the reuse vector $\vec{r}$. For this example, this means that instead of computing $\texttt{S}[i]$ with the original accumulation $\texttt{S}[i]$ += $\texttt{A}[i]$, the transformation computes $\texttt{S}[i]$ with $\texttt{S}[i] = \texttt{S}[i-1] + \texttt{A}[i]$ on the final polyhedral domain $\{[i] : 1 \leq i < N\}$.

Completion. The polyhedral $\{[i] : 1 \leq i < N\}$ does not cover the full domain of the original reduction. Specifically, it is missing $\texttt{S}[i]$ on the domain $\{[i] : i = 0\}$ – that is, exactly when $i = 0$. The value of $\texttt{S}[0]$ should be equal to $\texttt{A}[0]$. Therefore, we should expect two IR statements: one statement for initializing $\texttt{S}[0]$ to $\texttt{A}[0]$, and one statement incrementing while reusing a previous value of $\texttt{S}[i-1]$.

The transformed code in Listing 4.1 captures this intuition: `T-add-only` for initialization and `T-add-reuse` for incrementing and reusing; however, instead of directly using $\texttt{A}[j]$, the transformation uses an auxiliary array `S_ADD`, which is useful for generalizing to cases where initialization or incrementalization requires more than one value from the right hand side expression (i.e. A).

4.2 Configuration of Simplification Transformation

A fully automated optimizing compiler should automatically identify a reuse direction $\vec{r}$ and apply ST. There are several considerations when comes to choosing $\vec{r}$.

- **Complexity**: performing ST along $\vec{r}$ reduces the complexity of the computation.
- **Inverse**: if $\oplus$ does not have an inverse, performing ST along $\vec{r}$ should not require an inverse operator.
- **Sharing**: RHS expression presents *sharing* (defined below) along $\vec{r}$.

Each of the requirement prescribes a set, potentially an infinite set, of vectors $\vec{r}$, which we explain below:

Complexity. We require that applying ST along reuse vector $\vec{r}$ reduces the complexity of the program. The complexity of a program will not increase after applying ST for any $\vec{r}$; however, the complexity can stay the same if $\vec{r}$ is chosen along a direction where the original polyhedral domain $\mathcal{P}$ has constant thickness – that is, the extent of $\mathcal{P}$ is bounded by some constant not parameterized by the input parameters of the program. For example, consider an extreme case of the prefix sum example (Listing 1.1, Listing 1.2) but with the input parameter N fixed to some constant – say $N = 4$. The complexities before and after ST will be the same – $\mathcal{O}(1)$ – since both programs will perform a fixed number of computations. For a statement S with domain $\mathcal{P}$, we use $\mathcal{L}(\mathcal{P})$ to denote the set of vectors $\vec{r}$ that satisfies the complexity condition.

Inverse. If $\oplus$ does not have an inverse, we require that applying ST along a vector $\vec{r}$ will not introduce statements that requires the inverse operator of $\oplus$. For example, if $\oplus$ is min() or max(), it does not have an inverse; in such cases, Gautam and Rajopadhye [2006] introduces the concept of *Boundary Constraints* – which in short is the set of constraints of the domain $\mathcal{P}$ that are orthogonal to the projection *proj* – and require that $\vec{r}$ must be pointing out of (instead of pointing into) the boundaries of $\mathcal{P}$ corresponding to the Boundary Constraints. For a statement S, we use $\mathcal{I}(\text{S})$ to denote the set of vectors $\vec{r}$ that satisfies the inverse condition.

Sharing. We require that applying ST along a vector $\vec{r}$ where the right hand side expression of the considered reduction presents sharing along $\vec{r}$. For example in Section 4.1, we analyzed that the prefix sum example has sharing along direction of i. Gautam and Rajopadhye [2006] introduced an algorithm to determine the share space, the space formed by all reuse directions, given an equationally specified reduction. For a statement S, we use $\mathcal{S}(\text{S})$ to denote the set of vectors $\vec{r}$ that satisfies the inverse condition.

In general, for a statement S, denote its domain as S.*domain*; we would like to find the intersection $\mathcal{R}(\text{S}) = \mathcal{L}(\text{S}.domain) \cap \mathcal{I}(\text{S}) \cap \mathcal{S}(\text{S})$, so that any $\vec{r} \in \mathcal{R}(\text{S})$ is a valid reuse vector to perform ST.

4.3 Recursive ST

Notice in Listing 4.1 that statement Tadd still contains a reduction. Although for this example Tadd does not have further ST opportunities, in general, the residual reduction might still have available ST opportunities so that ST can be applied recursively to all introduced reductions [1].

[1] In general, ST can also introduce more than one reduction; we include a full description of ST in Appendix B.

Chapter 5

Multiple-Statement SR Problem

In this section, we state the Multiple-Statement Simplifying Reduction (MSSR) problem. In particular, we focus on the core of the Simplifying Reduction approach – the Simplification Transformation in Section 4.1 – and do not consider the Simplifying Reduction framework's additional *enabling transformations*. These transformations increase available simplification opportunities; we briefly touch on enabling transformations in Section 5.3.

5.1 Problem Statement

Ideally we'd formulate the Multiple Statement Simplifying Reduction (MSSR) as follows:

$$\textbf{minimize} \;\; \text{complexity}(\mathit{prog}') \tag{5.1a}$$
$$\textbf{subject to}$$
$$\mathit{prog}^1 = \mathit{prog},\; \mathit{prog}' = \mathit{prog}^n \tag{5.1b}$$
$$\forall i \in [2, n]\colon \mathit{prog}^i = \mathrm{ST}_{\mathrm{S}_i, \vec{r_i}}(\mathit{prog}^{i-1}) \tag{5.1c}$$
$$\vec{r_i} \in \mathcal{L}\left(\mathrm{S}_i.\mathit{domain}\right) \cap \mathcal{I}\left(S_i\right) \cap \mathcal{S}\left(S_i\right) \tag{5.1d}$$
$$\exists \text{ schedule } \Theta \text{ of } \mathit{prog}^n, \tag{5.1e}$$
$$\text{s.t. } \Theta \text{ satisfies dependence}(\mathit{prog}^n) \tag{5.1f}$$
$$\textbf{given} \;\; \mathit{prog}, \text{ dependence}(\mathit{prog}) \tag{5.1g}$$
$$\textbf{variables} \;\; \mathrm{S}_1, \mathrm{S}_2, ..., \mathrm{S}_n,\; \vec{r_1}, \vec{r_2}, ..., \vec{r_n} \tag{5.1h}$$

This states that given a program *prog*, and all pairwise dependencies between those statements, dependence (*prog*), apply a sequence of n ST transformations, $\mathrm{ST}_{\mathrm{S}_1,\vec{r}_1}, ...$ $, \mathrm{ST}_{\mathrm{S}_n,\vec{r}_n}$ that minimizes the complexity of the resulting program, *prog′*. Here we use $\mathrm{ST}_{\mathrm{S},\vec{r}}(\mathit{prog})$ to denote an ST that is applied on a statement S in *prog* along the reuse vector $\vec{r}$. Further, Equation (5.1d) requires each r_i to satisfy the constraints (i.e. complexity, inverse and sharing, denoted by $\mathcal{L}(.), \mathcal{I}(.), \mathcal{S}(.)$ respectively) as stated in Section 4.2.

Unfortunately, Equation (5.1) has three issues: 1) it is not a well-defined formulation due to the unknown n 2) it has infinite space for $\vec{r_i}$ 3) it has impractically large space for S_i.

First, it is not a well-defined formulation: to define its variables, the problem relies on an oracle to produce n, the total number of ST applications – even though there is no readily apparent bound on that number. To elaborate, each ST application removes one statement, and introduces zero to two reductions that are potentially applicable for further ST applications – thus one needs justification that recursively applying ST always terminates in order to bound n.

Secondly, even if we assume that n is given and bounded, the formulation does not readily translate to an executable algorithm. Specifically, enumeratively searching all possible $\vec{r_i}$ combinations is not feasible: each $\vec{r_i}$ alone is chosen from an infinite set of vectors, and the entire search space is also infinite; therefore the search space of $\vec{r_i}$s is impossible to navigate with enumerative search.

Thirdly, also assuming n is given and bounded, the program relies on a sequence of S_is, to specify on which statement in $prog^i$ to perform ST. Although, unlike the case of $\vec{r_i}$, the number choices for each S_i is finitely bounded (i.e. by the number of ST applicable reductions in the program), the combinations of all possible $(\mathrm{S}_1, ...\mathrm{S}_n)$ has at least $|\mathrm{S}_1|!$ possibilities: assuming the best case scenario where each ST applications removes one reduction and introduces zero reductions that are potentially applicable for further ST applications, which imples the ith ST application has $n-i+1$ remaining alterantive choices of S_i (i.e. $|\mathrm{S}_i| = n - i + 1$). Therefore the search space of S_i is also not practical to navigate with enumerative search.

We will resolve these issues with a correct formalization in the rest of Chapter 5. Specifically, we show, for a program, an one-to-one correspondance between all its potential ST applications and all faces of its reductions' domains. This correspondance resolves the first issue by bounding the number of ST applications to the number of faces of the program. This correspondance also allows a construction of an Integer Bilinear Programming (IBP) formulation to MSSR, which avoids the explicit enumerative search in the second and third issue.

5.1.1 Per-face ST application

We first make the following observation of ST on single statement S with domain $\mathcal{P}$: if we apply ST on S, we can then recursively apply ST on the newly introduced reductions, as in Section 4.3, and this is exactly the root problem of the incorrect formulation Equation (5.1): this recursion appears non-terminating. We will solve this issue by stating and proving Lemma 2 — to this end, we first recall Lemma 1 from Gautam and Rajopadhye [2006] that we will use in our proof. We then state Lemma 2 and give a proof.

Lemma 1 (Local Face Correspondance [Gautam and Rajopadhye, 2006, Theorem 3])**.** Let $\mathcal{P}'$ be the translation of an n-dimensional $\mathcal{P}$ along $\vec{r}$, then $\mathcal{P} - \mathcal{P}' = \uplus \mathcal{P}_i$, and there exist an one-to-one map such that each $\mathcal{P}_i$ correspond uniquely to a $(n-1)$-dimensional face of $\mathcal{P}$.

Lemma 2 (Global Face Correspondance)**.** Each recursive application of ST is on a subset (a polyhedral set) of $\mathcal{P}$, and all subsets correspond exactly one-to-one to all faces of $\mathcal{P}$.

Proof. Given a statement S with domain $\mathcal{P}$, ST performs a shift of $\mathcal{P}$ along a given reuse vector to $\mathcal{P}'$; new reduction statements are introduced over domains $\mathcal{P} - \mathcal{P}'$ and $\mathcal{P}' - \mathcal{P}$. Note that these two domains are non-convex half shells around the original domain $\mathcal{P}$, and together form a full shell around $\mathcal{P}$. The two shells are both non-convex, however by Lemma 1, they decompose into convex polyhedral domains, each corresponding to a unique $(n-1)$-dimensional face of the n-dimensional $\mathcal{P}$.

Since ST is applied recursively on these decomposed $(n-1)$-dimensional faces and then on the sequence of $(n-i)$-dimensional faces until the recursion hits the vertices of $\mathcal{P}$. The entire recursion is therefore a procedure that enumerates through all faces of a statement S's full domain $\mathcal{P}$ and assigns a reuse vector to each face. □

With Lemma 2, the recursive ST application always terminates since the number of faces of $\mathcal{P}$ is finite. Further, this introduces a *per-face application view of ST* — under this view, the algorithm first chooses a reuse vector for each face of $\mathcal{P}$ up-front; it then uses the same recursive ST application starting at $\mathcal{P}$ same as before; however, for each sub-domain's ST application, it uses the reuse vector assigned for the face corresponding to that sub-domain. Lastly, note that the reuse vector assigned to each face is parallel to the face because the residual domain corresponding to the face already has constant thickness orthogonal to that face – therefore shifts not parallel to the face does not change complexity and disobeys the rule stated for Complexity in Section 4.2.

5.2 Integer Bilinear Program Formulation

With the per-face application view of ST in Section 5.1.1, we are now ready to give the correct formulation of MSSR. The basic idea behind this formulation is to combine previous work on SR for a single statement [Gautam and Rajopadhye, 2006], previous work on integer linear program formulation of polyhedral model scheduling Pouchet et al. [2011, 2007, 2008] and the per-face application view of ST presented in Section 5.1.1. We first revisit Equation (5.1) and give the correct high level formulation as follows:

$$\textbf{minimize} \;\; \text{complexity}(\textit{prog}') \tag{5.2a}$$

$$\textbf{subject to}$$

$$\textit{prog}' = (\text{ST}_{f_1,\vec{r}_1} \circ ... \circ \text{ST}_{f_n,\vec{r}_n})(\textit{prog}) \tag{5.2b}$$

$$\forall i \in [1, n] : \; \vec{r}_i \in \mathcal{L}(f_i) \cap \mathcal{I}(f_i.\textit{stmt}) \cap \mathcal{S}(f_i.\textit{stmt}) \tag{5.2c}$$

$$\exists \text{ schedule } \Theta \text{ of } \textit{prog'}, \tag{5.2d}$$

$$\text{s.t. } \Theta \text{ satisfy dependence}(\textit{prog}') \tag{5.2e}$$

$$\textbf{given} \;\; \textit{prog}, \text{ dependence}(\textit{prog}) \tag{5.2f}$$

$$\textbf{variables} \;\; \vec{r_1}, \vec{r_2}, ..., \vec{r_n} \tag{5.2g}$$

For Equation (5.2), $\{f_1...f_n\}$ denotes the set of all faces of domains of all statements in *prog*. Following the per-face view of ST in Section 5.1.1, the function composition $\text{ST}_{f_1,\vec{r}_1} \circ ... \circ \text{ST}_{f_n,\vec{r}_n}$ denotes applying per-face ST with the assigned reuse directions $\vec{r}_1...\vec{r}_n$ (for all faces $f_1...f_n$). We use $f.\textit{stmt}$ to denote a face f's corresponding statement (i.e. the statement which has the domain f).

This high level formulation is similar to Equation (5.1), except that now 1) each reuse vector $\vec{r}_i$ is in one-to-one correspondence with a face f_i — we thus have a bounded number of unknown variables for reuse vectors 2) the variables S_i are eliminated, as the new formulation uses the per-face ST view, instead of the recursive ST application view. Lastly, each reuse vector is still constrained to satisfy the validity constraints (i.e. Equation (5.2c)).

5.2.1 Variables

As we mention above, the unknown variables contains reuse vectors $\vec{r}_1 ... \vec{r}_n$. Note that, moreover, the existential quantification over Θ (Equation (5.2d)) implies that Θ is also an unknown. As in Section 3.3.1, Θ is a collection of scheduling functions Θ_{S}, one for each statement S in the final program *prog′*; each Θ_{S} is an $m \times n$ dimension matrix of integer unknowns, where m is the schedule dimension, and n is one plus the sum of the number of dimension of S.*domain* and the number of parameters of the program.

5.2.2 Constraints

Equation (5.2) contains two main categories of constraints: *reuse constraints* in Equation (5.2c) and *dependency constraints* in Equation (5.2f).

Reuse constraints. The reuse constraints enforces that each $\vec{r}_i$ is chosen from $\mathcal{L}(f_i) \cap \mathcal{I}(f_i.stmt) \cap \mathcal{S}(f_i.stmt)$ — this later set is a union of polyhedral sets computable from f_i. Since it's a union of polyhedral sets, we use disjunction to constrain $\vec{r}_i$ to belong to one of the polyhedral sets. For each the polyhedral set, encoding that $\vec{r}_i$ belongs to the polyhedral set is then just a simple affine inequality constraint.

Dependency constraints. The dependency constraints enforces that Θ satisfies the dependency of *prog′*. Specifically, it requires that for each pair of statements S and T that potentially occur in *prog′*, their scheduling functions $\Theta_{\mathrm{S}}, \Theta_{\mathrm{T}}$ satisfy the dependence relation $\mathcal{D}_{\mathrm{S,\,T}}$. On the high level, we set up the constraints just the same as in Equation (3.2). However, the dependence matrix $D_{\mathrm{S,\,T}}$ now contains entrie(s) with (linear) terms with unknowns from $\vec{r}_1 ... \vec{r}_n$. An informative argument for why $D_{\mathrm{S,\,T}}$ contains these unknown entries is: if we look from the recursive ST view, each application of ST introduces a reuse direction unknown $\vec{r}$, and the algorithm recurses down to the residual reductions – for the next recursive application, we can think of it as taking in a program with both the original program's parameters, and also the reuse vector unknowns introduced by the previoys ST application. The residual reductions' domains then have space extended by $\vec{r}$. not finished

5.2.3 Objective: complexity

Since we would like to minimize the overall complexity, we need to express our integer bilinear program's objective as the complexity of the transformed program. We can compute complexity of each face by counting the cardinality of each face's domain Verdoolaege et al. [2007]. The cardinality of a face is an Ehrhart polynomial Ehrhardt [2009] in terms of the program parameters.

Encoding. If the program only has one parameter, then the degree of the polynomial is a natural choice of a scalar that represents the complexity of the program.

If the program has multiple parameters, then one needs to be careful about comparing complexities: it is necessary to be able to compare between $\mathcal{O}(M^2N)$ and $\mathcal{O}(MN^2)$in order to minimize complexity. To this end, we assume that a total ordering is given for all possible polynomial terms of global parameters as a sequence of increasing scalars. For example, with two global parameters M, N, and maximum possible complexity $\mathcal{O}(M^2N^2)$, a total ordering such as $\mathcal{O}(1) < \mathcal{O}(M) < \mathcal{O}(N) < \mathcal{O}(MN) < \mathcal{O}(M^2N) < \mathcal{O}(MN^2) < \mathcal{O}(M^2N^2)$ is given, and integers 0...6 are assigned to each big-O term in the previous sequence.

Summing scalar encodings. Either the program has a single global parameter or has multiple global parameters, we have a mapping from complexities, which are polynomials in terms of global parameters, to their scalar encodings. Since the final objective is the total complexity of the full transformed program, we need to sum the scalar encoding of complexities for all statements, without losing the ability to compare the resultants' degrees. To that end, we propose to use a simple base-$|S|$ encoding method where $|S|$ is the maximum number of statements in the program: for a complexity encoded as scalar c, we use $|S|^c$ as a term in the final objective. As an example, to sum two complexities represented in scalar c_1 and c_2, we compute

$|S|^{c_1} + |S|^{c_2}$. We define the base-$|S|$ sum of c_i as $\sum |S|^{c_i}$.

Indicator variable. In the formulation, we require indicator variables to indicate if ST is disabled along a certain face – in which case no complexity reduction should be applied for the corresponding domain. We can use the big-M method, a well-known ILP modeling trick, to encode an indicator variable $y \in \{0, 1\}$ for the constraint $x = 0$ so that $y = 1$ iff $x = 0$.

5.3 Discussion

The above formulation is an integer objective bilinear constrained program. The objective is linear because it is an affine combination of the indicator variables. The problem is bilinear constrained because: in the original ILP formulation scheduling, the dependence matrix (defined in Section 3.3.2) is multiplied by a vector of unknowns to form a linear constraint; however by introducing the unknown reuse vectors $\vec{r_i}$, the dependence matrix contains entries that depends on $\vec{r_i}$, thereby making the constraints bilinear.

Enabling transformations. The enabling transformations presented in the original SR paper [Gautam and Rajopadhye, 2006] can be incorporated into our formulation by the use of binary decision variables and technique of encoding logical constraints as integer linear constraints.

Chapter 6

MSSR Heuristic Algorithm

The problem formulation we present in Section 5.2 is a full characterization of the MSSR problem. In this work we consider this formulation only as a specification instead of a complete solution — solving an integer linear objective bilinear constrained program is NP-hard. As far as we know, none of well-known solvers can solve this problem out of the box, though it is possible to reformulate such problem into mixed integer linear programming (MILP) [Gupte et al., 2013]. However, the size of the formulation(i.e. total number of constraints and number of variables) in Section 5.2 is portional to the number of statements, number of faces per statement and the maximal complexity of the program – either one of which could potentially lead to exponential blow up in the size of the formulation. Further, our formulation on dependency resolution is based on ILP formulation of multidimensional scheduling, which by itself already introduces a tractability challenge as pointed out in Pouchet et al. [2011].

For these reasons, we propose here a sound heuristic solution to MSSR. The key idea behind our heuristic approach is that for a program with an affine schedule, we can leverage the schedule itself to choose a reuse vector for each ST that we apply to the program. Specifically, for any reuse vector that is valid for a given face (according the constraints that we specify in Section 5.2), our algorithm chooses either the reuse vector itself or its negation as the reuse vector for the ST. This algorithm – though simple – is still optimal for reductions that have inverses – which spans a broad class of programs – and always preserves the original dependencies of the program.

6.1 Insights

The key insights that guide our algorithm are that 1) choosing any valid reuse vector for a given ST results in the same final algorithmic complexity for the program and 2) for any valid reuse vector, the direction itself or its negation adheres to the program's original affine schedule of the LHS of the reduction. We demonstrate these two insights with the following lemmas.

Lemma 3. For any application of ST, the complexity decrease is always the same regardless of the actual choice of reuse vector.

Proof. For any ST application, the reduction's n-dimensional domain $\mathcal{P}$ is reduced to the two half shells $\mathcal{P} - \mathcal{P}'$ and $\mathcal{P}' - \mathcal{P}$. The two half shells decompose into convex polyhedral sets corresponding to all $(n-1)$-dimensional faces of $\mathcal{P}$. Further, for each decomposed convex polyhedral set, the thickness of the set, which is defined as the spanned width of the set orthogonal to its corresponding face, is a constant dependent solely on the ST's reuse vector and the face's orientation. Therefore, the cardinality of each decomposed polyhedral set is just the cardinality of the face multiplied by some constant. It then follows that for any two STs with two non-zero reuse vectors, their resultant residual reductions' complexities are always the same, and equal the sum of the cardinalities of all the faces of $\mathcal{P}$ multiplied by some constant. □

Before introducing the next lemma, we first introduce an extended definition of scheduling functions. Recall that the scheduling function of a reduction statement is an affine function from the reduction's domain to the timestamp. We extend the context of scheduling function from a reduction statement to the LHS of a reduction in a given program as follows. First the program is augmented by adding to the program a new redirect statement $\texttt{A}[\vec{x}] = \texttt{A}'[\vec{x}]$ with the same domain as the domain of $\texttt{A}$, where $\texttt{A}'$ is a fresh symbol which replaces the LHS array $\texttt{A}$ of the program. Then the scheduling function of the LHS of the reduction is simply the scheduling function of the newly introduced redirect statement of the LHS in the schedule of the augmented program.

Lemma 4. Given the affine schedule for the augmented program, then for any ST application on a reduction whose operator has an inverse and for any valid reuse vector $\vec{r}$, either $\vec{r}$ or $-\vec{r}$ agrees with the schedule of the original program and does not introduce a dependency cycle.

Proof. Consider a reduction statement S with projection *proj* and LHS array `A`. Suppose `A` has an affine schedule $\Theta_{\texttt{A}}$ then we have $\Theta_{\texttt{A}} \cdot [\vec{x}, \vec{p}, 1]^T$ is the schedule time for $\texttt{A}[\vec{x}]$. Let vector $\vec{r}$ be in same space of the domain of S, and we shift the domain of S along $\vec{r}$; let the projected vector of $\vec{r}$ onto the domain of `A` be $\vec{r_{\texttt{A}}} = proj(\vec{r})$. Consider $\vec{x}$ and $\vec{x} + \vec{r_{\texttt{A}}}$. Their scheduled timestamps are $\Theta_{\texttt{A}} \cdot [\vec{x}, \vec{p}, 1]^T$ and $\Theta_{\texttt{A}} \cdot [\vec{x} + \vec{r_{\texttt{A}}}, \vec{p}, 1]^T$. Since $\Theta_{\texttt{A}} \cdot [\vec{x} + \vec{r_{\texttt{A}}}, \vec{p}, 1]^T - \Theta_{\texttt{A}} \cdot [\vec{x}, \vec{p}, 1]^T = \Theta_{\texttt{A}} \cdot [\vec{r_{\texttt{A}}}, \vec{0}, 0]^T$ is a constant not dependent on $\vec{x}$, it must be the case that for all $\vec{x}$, either $\texttt{A}[\vec{x}]$ is always scheduled before $\texttt{A}[\vec{x} + \vec{r_{\texttt{A}}}]$, or vice versa. Specifically, if the first non-zero entry (in accordance with the timestamp comparison in Definition 5) of $\Theta_{\texttt{A}} \cdot [\vec{r_{\texttt{A}}}, \vec{0}, 0]$ is positive, then $\texttt{A}[\vec{x}]$ is always schedule before $\texttt{A}[\vec{x} + \vec{r_{\texttt{A}}}]$, otherwise, $\texttt{A}[\vec{x}]$ is always schedule after $\texttt{A}[\vec{x} + \vec{r_{\texttt{A}}}]$. If $\texttt{A}[\vec{x}]$ is scheduled before $\texttt{A}[\vec{x} + \vec{r_{\texttt{A}}}]$, then applying ST with reuse vector $\vec{r}$ will not introduce any dependence cycle, since the newly introduced dependency is always consistent with the original schedule; on the other hand, if $\texttt{A}[\vec{x}]$ is scheduled after $\texttt{A}[\vec{x} + \vec{r}]$, then applying ST with reuse vector $-\vec{r}$ will not introduce any dependence cycle.

Further, since $\vec{r}$ chosen this way is always consistent with the original schedule, a previous application of ST will not affect a later application of ST — intuitively, a previously applied ST introduces a dependency that can be subsumed by an enforced dependency according to the original program's schedule; thus later a application of ST, as long as it is also consistent with original schedule, will not be affected. □

6.2 Algorithm

With justification in Section 6.1, we now introduce the heuristic algorithm in Figure 6-1.

1. Schedule the augmented program to obtain an initial schedule Θ for all statements and LHS of reductions
2. Apply ST to all faces of all reduction statement's domains; choose the direction that is consistent with Θ by:
   (a) First pick any valid reuse vector $\vec{r}$ from the candidate set.
   (b) Test if $\vec{r}$ is consistent with Θ, if not consistent, set $\vec{r} \leftarrow -\vec{r}$, if $-\vec{r}$ is also a valid reuse vector; otherwise, do not apply the current ST.

Figure 6-1: SSSR heuristic algorithm

To test if $\vec{r}$ is consistent with Θ, one can compute $\Theta_{\mathtt{A}} \cdot \vec{r_{\mathtt{A}}}$ (with $\Theta_{\mathtt{A}}, \vec{r_{\mathtt{A}}}$ defined as in Lemma 4) and test if the first non-zero entry is positive. Alternatively, a naive way is just to attempt to reschedule the original program with the introduced dependency along $\vec{r}$ and test if the program is schedulable.

6.3 Algorithm Analysis

Heuristic scheduling. One advantage of the heuristic algorithm in Figure 6-1 is that the schedule Θ does not need to be obtained from forming and solving the ILP formulation as in Section 3.3, and one is free to choose any scheduling algorithm in the polyhedral literature such as Gupta et al. [2007], Feautrier [1992a,b], Bondhugula et al. [2008]. Most of these algorithms, such as the PLUTO scheduler [Bondhugula et al., 2008] provides scalable solution to the polyhedral scheduling problem and thus algorithm in Figure 6-1 does not present bottleneck due to scheduling.

Optimality Guarantee. The algorithm is optimal for the MSSR problem if all reduction operators have inverses. This is because the algorithm considers a basis direction of reuse, and picks the direction along that basis that is consistent with the original schedule. As long as all reduction operators have inverses, the heuristic algorithm will assign a non-zero reuse vector to each face that has valid reuse opportunities — in other words, the heuristic algorithm maximizes the total number ST applications among all faces, if all reduce operators have inverses. For any application of ST along a face, the complexity decrease is always the same regardless of the actual choice of reuse vector, therefore, maximizing the number of ST applications among all faces minimizes the total complexity.

Lastly, if a reduction operator does not have an inverse, thereby restricting the candidate set of directions, then its possible for our algorithm produces a non-optimal solution. Specifically, if an operator does not have an inverse, the valid reuse vector for that operator will be restricted to a one sided direction (since ST requires the reuse direction to point out of certain boundaries of the polyhedral domain if the operator does not have an inverse), instead of both directions of the basis. It is possible that the original program does not have an unique valid schedule. Consider the following scenario: one schedule is consistent with $\vec{r}$, while another schedule is consistent with $-\vec{r}$; since the operator does not have an inverse, only the positive direction $\vec{r}$ is valid. Therefore, the initial schedule will affect whether this ST is applied or not – which in turn leads to the suboptimality of the algorithm.

Chapter 7

Implementation

We implemented our IR as in Section 3.2 and the heuristic algorithm as in Chapter 6 using Python. We use Integer Set Library (ISL) [Verdoolaege, 2010] for manipulation of polyhedral set and relations.

For obtaining the original program schedule, we use a PLUTO-like scheduler built-in of ISL. To test if a reuse vector is consistent with the original schedule, we simply attempt to introduce a new dependency along the reuse vector and perform a full scheduling — note that this is not necessary, and can be potentially eliminated by the method of computing $\Theta_{\mathrm{S}} \cdot \vec{r}$ following Section 6.2. However, in our case simply attempting to reschedule the program is easier to implement and the method is agnostic to the underlying scheduling algorithm.

Chapter 8

Evaluation

The algorithm presented in this work is particularly effective at optimizing straightforward, unoptimized implementations of *probabilistic inference* procedures into efficient implementations. The inference procedures have mathematical specifications that naturally translate to our IR. The inference procedures are also *iterative*. In this section, we evaluate our heuristic algorithm's effectiveness to improve the asymptotic complexity of benchmarks consisting of such inference algorithms.

Methodology. We evaluate our implementation in Chapter 7 using unoptimized implementations of probabilistic inference procedures. We present their algorithmic complexities, before and after optimization using our heuristic implementation from Chapter 7, and the optimal complexities achievable with transformations in this work, by potentially solving the problem formulation in Section 5.2 exactly. We also report complexities of manual implementations with possible transformations not in this work.

We collect the complexities before and after by counting the cardinality of the resultant polyhedral domains using library implementations in Verdoolaege et al. [2007]. We collect the optimal complexity by inspecting the benchmarks and deriving the optimal complexities manually. We collect complexities of manual implementations by either finding an existing implementation of the algorithm if one exists in the literature or, otherwise, by manually deriving the best known.

Model	Original	Optimized	MSSR-Optimal	Manual	#IR	#SR
GMM-GS	$\mathcal{O}(N^2K^2)$	$\mathcal{O}(NK)$	$\mathcal{O}(NK)$	$\mathcal{O}(NK)$	16	24
GMM-MH	$\mathcal{O}(N(N+K))$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	16	24
GMM-LW	$\mathcal{O}(N(N+K))$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	5	7
LDA-GS	$\mathcal{O}(W^2K^2)$	$\mathcal{O}(WK)$	$\mathcal{O}(WK)$	$\mathcal{O}(WK)$	20	28
LDA-MH	$\mathcal{O}(W^2K)$	$\mathcal{O}(WK)$	$\mathcal{O}(WK)$	$\mathcal{O}(WK)$	42	65
LDA-LW	$\mathcal{O}(W^2K)$	$\mathcal{O}(W)$	$\mathcal{O}(W)$	$\mathcal{O}(W)$	7	11
DMM-GS	$\mathcal{O}(WADK^2+D^2K^2)$	$\mathcal{O}((W+A)KD)$	$\mathcal{O}((W+A)KD)$	$\mathcal{O}(AKD)$	40	46
DMM-MH	$\mathcal{O}(D^2K^2+D(W+A))$	$\mathcal{O}((K+W+A)D)$	$\mathcal{O}((K+W+A)D)$	$\mathcal{O}((K+L+A)D)$	82	142
DMM-LW	$\mathcal{O}((WA+K)D)$	$\mathcal{O}((K+W+A)D)$	$\mathcal{O}((K+W+A)D)$	$\mathcal{O}((K+L+A)D)$	10	12
LBP	$\mathcal{O}(N^2K^2D^2)$	$\mathcal{O}(N^2K^2D)$	$\mathcal{O}(N^2K^2D)$	$\mathcal{O}(N^2K^2D)$	3	1
CoxPh	$\mathcal{O}(K^2N^2)$	$\mathcal{O}(K^2N)$	$\mathcal{O}(K^2N)$	$\mathcal{O}(K^2N)$	6	5

Table 8.1: Benchmarks Table

Benchmarks. A subset of the benchmark algorithms we consider are identified as a "model-algorithm" pair, where the model refers to a generative probabilistic model, and the algorithm refers to a class of algorithm to perform inference on the model. For models, we consider the Gaussian Mixture Model (GMM) Murphy [2012], Latent Dirichlet Allocation (LDA) Blei et al. [2003] and Dirichlet Multinomial Mixture (DMM) Holmes et al. [2012]. For algorithms, we consider Gibbs Sampling (GS) Geman and Geman [1984], Metropolis Hasting (MH) Metropolis et al. [1953], Hastings [1970] and Likelihood Weighting (LW) Fung and Chang [1989]. Thus we have a total of 9 benchmark algorithms of this kind.

The models for LDA [Blei et al., 2003, Griffiths and Steyvers, 2004] and DMM [Holmes et al., 2012] are popular for existing data science problems. In addition, the models for GMM [Daniel Huang, 2017, Walia et al., 2018], LDA [Daniel Huang, 2017, Walia et al., 2018], and DMM [Walia et al., 2018] have been used as benchmarks for probabilistic inference systems.We chose Gibbs sampling [Geman and Geman, 1984], Metropolis-Hastings [Metropolis et al., 1953, Hastings, 1970], and Likihood Weighting [Fung and Chang, 1989] because they are all common inference algorithms from the literature. LDA and DMM are particularly valuable benchmarks because there are published Gibbs sampling algorithms that researchers have manually optimized (Griffiths and Steyvers [2004] and Resnik and Hardisty [2010], respectively).

We also include another two benchmarks: Loopy Belief Propagation on a grid Ising model (LBP-Ising) and the Cox proportional hazards model (CoxPh) [Therneau, 2013, Cox, 1972]. Loopy Belief Propagation Bishop [2006] is an iterative approximate inference algorithm, and its instantiation on the Ising model has applications in fields such as vision [Grauer-Gray and Cavazos, 2011] and physics Kikuchi [1951]. CoxPh is a well known statistical model, which is typically combined with Newton's method, an iterative optimization algorithm, for optimization. CoxPh is commonly found in medical applications [Collett, 1993, White et al., 2016], and mechanical systems [Susto et al., 2015].

The benchmarks all have the common feature that they are iterative methods specialized to a generative probabilistic model.

At these scales, the asymptotic complexity outweighs constant factor improvements such as memory accesses and loop order, and thus suitable to analyze the asymptotic performance.

Results. Table 8.1 summarizes the results of our approach. Each symbol in the complexity columns corresponds to a parameter of the benchmark.

The column "Original" gives the complexity of the original program for the benchmarks. The column "Optimized" gives the complexity of the transformed program using the heuristic implementation in Chapter 7. The column "MSSR-Optimal" gives the complexity of the transformed program by potentially solving the problem formulation in Section 5.2 exactly; this is the optimal complexity one can get using techniques presented in this work. The column "Manual" gives the complexity of a potential optimized manual implementation written by a diligent developer; this means that the complexity reduction potentially comes from transformations not present in this work.

The column "#IR" counts the number of IR statements for the benchmark; the column "#SR" counts the simplifying reductions attempted.

Comparing the "Original" and "Optimized" columns, we can see that our approach can reduce the complexity for all benchmarks. Comparing the "Optimized" and "MSSR-Optimal" columns, our approach can generate algorithms that have the same complexity as that of optimal implementation for all benchmarks. Optimality

Model	Original	Optimized	Speedup
GMM-GS	1.8×10^2 ms	1.3 ms	**138x**
GMM-MH	6×10^1 ms	1.7×10^{-1} ms	**353x**
GMM-LW	23 s	7.5×10^{-1} s	**30x**
LDA-GS	**timeout**	6.1×10^{-3} s	$> \mathbf{7 \times 10^6 x}$
LDA-MH	**timeout**	1.1×10^{-1} ms	$> \mathbf{3 \times 10^8 x}$
LDA-LW	**timeout**	45.3 s	> **953x**
DMM-GS	2.1×10^2 s	1.1 s	**191x**
DMM-MH	3.5×10^1 s	6.8×10^{-1} s	**51x**
DMM-LW	**timeout**	30 s	> **1440x**

Table 8.2: Runtime evaluations

is defined regarding programs realizable through transformations presented in this work. Comparing the "Optimized" and "Manual" columns, our approach can generate algorithms with complexities the same as manual implementations for 8 out of 11 benchmarks. We identified that the 3 benchmarks related to DMM require additional data layout modifications which we do not consider in this work, and lies in the direction of future research.

8.1 Runtime Validation

So far we have evaluated our heuristic algorithm using algorithmic complexity as the primary factor, which ignores constant factors. In this section, we validate our hypothesis that asymptotic complexity dominates potential constant factors for the parameters of these benchmarks by timing a subset of our benchmarks and comparing the runtimes of naive implementations with optimized implementations.

Parameter sizes. We collect the typical instantiated values for global parameters from the corresponding literature. For GMM we use Daniel Huang [2017], for LDA we use Newman [2008], for DMM we use Turnbaugh et al. [2008], for LBP-Ising we use Grauer-Gray and Cavazos [2011] and for CoxPh we use INVESTIGATORS [1989]. Based on these prior works, we collected the following parameters for each model:

GMM $N = 10000, K = 10$

LDA $W = 466,000, K = 50$

DMM $W = 570,000, K = 4, D = 278, A = 129, L = 3202$

LBP-Ising $N = 168750, K = 64, D = 4$

CoxPH $N = 15792, K = 6$

Results. Due to limitations on time, we evaluate only on the 9 benchmarks in Table 8.2. We use Python implementations that match the naive and optimized complexity that we report in Table 8.1. We ran these implementations and report timeouts for benchmarks that ran for longer than 12 hours.

It's evident from Table 8.2 that all 9 benchmarks have non-trivial speedups, which supports our previous hypothesis that for these benchmarks and our technique, complexity dominates constant-factor concerns.

Chapter 9

Related Work

Simplifying Reductions. Previous works on simplifying reduction are Liu et al. [2005] and Gautam and Rajopadhye [2006]. Liu et al. [2005] proposed a loop based transformation algorithm for reducing complexities on loop programs. The algorithm uses the Omega calculator [Padua, 2011] for analysis on *contributing set*. The method is general in that any set calculation method, potentially methods that work for even non-polyhedral sets, can be used. Method in Liu et al. [2005] uses only the direction of loop increment to decrease the complexity. Gautam and Rajopadhye [2006] generalized the method in Liu et al. [2005]; one of the advances was that it formalized the notion of reuse space and proposed to use directions in the reuse space to decrease complexity.

Incrementalization in Probabilistic Programming. The problem of incrementalization occurs in probabilistic programming system (PPS), and is known as incremental inference. Existing work such as Kiselyov [2016], Ritchie et al. [2016], Wu et al. [2016], Nori et al. [2015], Yang et al. [2014], Zhang and Xue [2019] attempt to address the problem of incremental inference inc PPS. However, these techniques are variants/combinations of 1) tracing JITs, 2) specialization and caching/memoization, 3) dynamic dependence analysis, 4) dynamic program slicing, or 5) runtime symbolic analysis – in summary, dynamic optimizations. These techniques introduce significant runtime overhead for storing dependency graph/traces (which is of size proportional to the number of the executed statement instances) and/or performing analysis on those graphs/traces dynamically. Our technique can be applied to PPS to solve the incremental inference problem; however, our technique is a static compilation techinique which do not suffer from runtime overhead.

Reductions. Previous work [Doerfert et al., 2015, Ginsbach and O'Boyle, 2017] proposed techiniques to detect reductions from loop based code; these techniques can be used as front-ends to our technique for conversion into our reduction based IR. Previous work [Rauchwerger and Padua, 1999, Doerfert et al., 2015, Reddy et al., 2016, Ginsbach and O'Boyle, 2017] optimize reductions in the polyhedral model for considerations such as privatization and parallelization. They do not optimize reductions' complexities; however, they can be used as optimizing backend for generating efficient code for reduction after applying our method.

ILP scheduling. Previous work [Pouchet et al., 2011, 2007, 2008] give ILP formulation of the scheduling problem. Specifically, Pouchet et al. [2011] shows how to construct constraints for a convex ILP problem to find an m-dimensional schedule for a program. Moreover, this formulation of constraints allows one to incorporate a desired objective to be optimized — in this work, we used the complexity of the final transformed program as the objective and we showed how to encode such objective as a affine expression in Section 5.2.3.

Heuristic scheduling. There are also other scheduling methods such as the ones in Feautrier [1992a,b], Bondhugula et al. [2008] that use heuristics to schedule a program. These methods are usually more scalable than an ILP formulation. In this work, we use ideas from the ILP formulation to formulate the MSSR problem, while our provided heuristic algorithm does not depend on using the ILP formulation for scheduling, and we also resort to the PLUTO [Verdoolaege, 2010, Bondhugula et al., 2008] heuristic scheduling algorithm in our implementation.

Chapter 10

Conclusion

In this work, we introduce the multiple-statement reduction problem and provide a heuristic algorithm that is optimal for reduction operators that have inverses. These reductions have otherwise only appeared as domain/algorithm-specific optimizations as described in the published description of standard probabilistic inference algorithms. Our hope is that this work formally outlines a key general-purpose optimization opportunity that can be delegated to the compiler, rather than being a significant piece of manual implementation that stands between the elaboration of a new probabilistic inference algorithm and its high performance implementation. Our results hold the promise that emerging language and systems for probabilistic programming could see significant performance improvements by incorporating our techniques.

Appendix A

Extra Listings

```
for(i = 0; i < N; i++)
 B[N - 1] += A[i]
for(i = N-2; i >= 0; i--)
 B[i] = B[i+1] - A[i]
```

Listing A.1: Alternative optimized PS (right-to-left)

Appendix B

Simplifying Reduction

A key opportunity that we've identified is the integration of the histogram transformation with the Simplifying Reduction transformation [Gautam and Rajopadhye, 2006].

B.1 Simplifying Reduction in Polyhedral Model

Consider an IR statement for which the set of non-affine equality predicates $\mathbf{Q}$ is empty:

$$\texttt{label:}\quad \mathsf{LHS}[\mathbf{u}] \oplus\!= \mathit{expr}\ :\ \emptyset\ \&\ \mathcal{P}$$

Simplifying reduction (SR) transforms this statement into an equivalent form as in Figure B-1. The transformation takes in one parameter, a nonzero constant vector $\vec{r}$, representing the direction of reuse, which we will explain shortly.

We first define some notations:

`l-add-only:`	$\mathsf{LHS}[\mathbf{u}] = \mathsf{ADD}[\mathbf{u}]$	$: \mathcal{P}^{\mathbf{u}}_{\text{add}} - \mathcal{P}^{\mathbf{u}}_{\text{int}}$
`l-reuse-only:`	$\mathsf{LHS}[\mathbf{u}] = \mathsf{LHS}[T^{\mathbf{u}}_r(\mathbf{u})]$	$: \mathcal{P}^{\mathbf{u}}_{\text{int}} - (\mathcal{P}^{\mathbf{u}}_{\text{add}} \cup \mathcal{P}^{\mathbf{u}}_{\text{sub}})$
`l-add-reuse:`	$\mathsf{LHS}[\mathbf{u}] = \mathsf{ADD}[\mathbf{u}] \oplus \mathsf{LHS}[T^{\mathbf{u}}_r(\mathbf{u})]$	$: \mathcal{P}^{\mathbf{u}}_{\text{add}} \cap (\mathcal{P}^{\mathbf{u}}_{\text{int}} - \mathcal{P}^{\mathbf{u}}_{\text{sub}})$
`l-reuse-sub:`	$\mathsf{LHS}[\mathbf{u}] = \mathsf{LHS}[T^{\mathbf{u}}_r(\mathbf{u})] \ominus \mathsf{SUB}[\mathbf{u}]$	$: \mathcal{P}^{\mathbf{u}}_{\text{sub}} \cap (\mathcal{P}^{\mathbf{u}}_{\text{int}} - \mathcal{P}^{\mathbf{u}}_{\text{add}})$
`l-add-reuse-sub:`	$\mathsf{LHS}[\mathbf{u}] = \mathsf{ADD}[\mathbf{u}] \oplus \mathsf{LHS}[T^{\mathbf{u}}_r(\mathbf{u})] \ominus \mathsf{SUB}[\mathbf{u}]$	$: \mathcal{P}^{\mathbf{u}}_{\text{sub}} \cap \mathcal{P}^{\mathbf{u}}_{\text{int}} \cap \mathcal{P}^{\mathbf{u}}_{\text{add}}$
`ladd:`	$\mathsf{ADD}[\mathbf{u}] \oplus\!= \mathit{expr}$	$: \mathcal{P}_{\text{add}}$
`lsub:`	$\mathsf{SUB}[\mathbf{u}] \oplus\!= \mathit{subst}(\mathit{expr}, T_r(\mathit{freevars}(\mathit{expr})))$	$: (\mathcal{P}^{\mathbf{u}}_{\text{int}})^{\mathbf{s}} \cap \mathcal{P}_{\text{sub}}$

Figure B-1: Simplifying Reduction in the Polyhedral Model

- we use $p^{\mathbf{a}}$ to denote projecting p onto space $\mathbf{a}$; the superscript acts effectively as an projection function; p can either be a point, an affine transformation or a polyhedral set of points.

- $T_r(x)$ is an affine translation transformation (under homogeneous coordinates). That is, if x is a vector $\vec{x}$ representing a point, T_r shifts $\vec{x}$ to $\vec{x} + \vec{r}$. If x is a polyhedron $\mathcal{P}$, T_r shifts all points in $\mathcal{P}$ by $+\vec{r}$.

Then, let $\mathcal{P}' = T_{-r}(\mathcal{P})$, i.e. $\mathcal{P}'$ is $\mathcal{P}$ shifted by $-\vec{r}$, we define the following symbols in Figure B-1:

$$\mathcal{P}_{\text{add}} = \mathcal{P} - \mathcal{P}' \qquad \mathcal{P}_{\text{sub}} = \mathcal{P}' - \mathcal{P} \qquad \mathcal{P}_{\text{int}} = \mathcal{P} \cap \mathcal{P}'$$

Explanation. The core intuition behind ST is to realize *reuse* of the RHS *expr*. Specifically, we require a choice of $\vec{r}$ so that it presents *sharing* for the RHS expression, that is:

$$[\![\mathit{subst}(\mathit{expr}, T_r(\mathit{freevars}(\mathit{expr}))) = \mathit{expr}]\!]$$

. In other words, the value of *expr* is the same for any point $\mathbf{v}$ and its shifted counterpart $T_r(\mathbf{v})$. This way, we can avoid evaluation of *expr* by simply copying from $\mathit{subst}(\mathit{expr}, T_r(\mathit{freevars}(\mathit{expr})))$, whenever possible. The first five statements `l-add-only` through `l-add-reuse-sub` computes LHS this way and reuse $\mathit{subst}(\mathit{expr}, T_r($ $\mathit{freevars}(\mathit{expr})))$ along $\vec{r}$. The domains of the five statements prescribe the set of points according to each statement's semantics.

To make this concrete, first notice the following:

- $\mathcal{P}^{\mathbf{u}}_{\text{add}}$ is the set of indices that receives *expr*'s values evaluated in $\mathcal{P}_{\text{add}}$

- $\mathcal{P}^{\mathbf{u}}_{\text{sub}}$ is the set of indices that receives *expr*'s values evaluated in $\mathcal{P}_{\text{sub}}$

- $\mathcal{P}^{\mathbf{u}}_{\text{int}}$ is the set of indices that receives *expr*'s values evaluated in $\mathcal{P}_{\text{int}}$. Receiving value from the intersection means that it is possible to reuse from the index point shifted by $\vec{r}$.

We then explain each of the first five statements in turn:

Reuse only. Consider the domain of `l-reuse-only`, $\mathcal{P}^{\mathbf{u}}_{\text{int}} - (\mathcal{P}^{\mathbf{u}}_{\text{add}} \cup \mathcal{P}^{\mathbf{u}}_{\text{sub}})$, can be read as: the set of indices that receive value from intersection, but does not receive from ADD or SUB, and this is precisely the set of points that can be directly copied along $\vec{r}$. Thus, `l-reuse-only` performs just this copy operation: $\mathsf{LHS}[\mathbf{u}] = \mathsf{LHS}[T^{\mathbf{u}}_{r}(\mathbf{u})]$.

Add Only. `l-add-only`'s domain $\mathcal{P}^{\mathbf{u}}_{\text{add}} - \mathcal{P}^{\mathbf{u}}_{\text{int}}$ can be read as: the set of indices that receive value from ADD, but does not recieve value from intersection. One can verify that $\mathcal{P}^{\mathbf{u}}_{\text{add}} - \mathcal{P}^{\mathbf{u}}_{\text{int}} = \mathcal{P}^{\mathbf{u}}_{\text{add}} - \mathcal{P}^{\mathbf{u}}_{\text{int}} - \mathcal{P}^{\mathbf{u}}_{\text{sub}}$ this also implies that the set also does not recieve value from SUB. Therefore, the statement just copies from ADD

Add and Reuse. `l-add-reuse`'s domain, $\mathcal{P}^{\mathbf{u}}_{\text{add}} \cap (\mathcal{P}^{\mathbf{u}}_{\text{int}} - \mathcal{P}^{\mathbf{u}}_{\text{sub}})$, can be read as: the set of indices that receive value from ADD and the intersection, but does not receive value from SUB. Therefore the statement reuses value along $\vec{r}$, and increments with value calculated from ADD.

Sub and Reuse. `l-reuse-sub`'s domain, $\mathcal{P}^{\mathbf{u}}_{\text{sub}} \cap (\mathcal{P}^{\mathbf{u}}_{\text{int}} - \mathcal{P}^{\mathbf{u}}_{\text{add}})$, can be read as: the set of indices that recieves value from SUB and the intersection, but does not receive value from ADD. Therefore the statement reuses value along $\vec{r}$, and decrements with value calculated from SUB.

Add, Reuse and Sub. `l-add-reuse-sub`'s domain, $\mathcal{P}^{\mathbf{u}}_{\text{sub}} \cap \mathcal{P}^{\mathbf{u}}_{\text{int}} \cap \mathcal{P}^{\mathbf{u}}_{\text{add}}$. can be read as: the set of indices that receive value from both ADD, the intersection and SUB. Therefore, the statement reuse along $\vec{r}$, increments with ADD and decrements with SUB.

Residual Reductions. The statements `ladd` and `lsub` are themselves reductions, and we will call them *residual reductions* after SR transformation. They compute additional values that are requested by the top five statements. The residual reduction accumulates the same right-side expression as the original reduction, but with domains that are subsets of the original domain.

B.2 Configuration of Simplifying Reduction

As mentioned in Section 4.2, we need to consider three constraints – complexity, sharing and dependence – when choosing the reuse vector $\vec{r}$. Here we discuss the constraints in more detail.

Complexity. A program's complexity is a function over its input parameters.

The complexity after one SR transformation is equal to the total sum of all cardinalities of domain sizes of the statements after the transformation. The complexity of the first five statements combined together is equivalent to iterating points of LHS array, and therefore it will always remain unchanged, since we will always need to compute answer for each point of the LHS. As shown in Gautam and Rajopadhye [2006], in order for one step of SR to be meaningful, in the sense that it decreases the complexity, we need that $|\mathcal{P}_{\text{add}}| + |(\mathcal{P}^{\mathbf{u}}_{\text{int}})^{\mathbf{s}} \cap \mathcal{P}_{\text{sub}}| < |\mathcal{P}|$.

Sharing. Fully determining all possible $\vec{r}$ that presents sharing for the right-hand side expression is not decidable: for an arbitrary RHS expression *expr* as an uninterpreted function: we can encode the problem as $\exists \vec{r} \forall \mathbf{v}. [\![\mathit{expr}]\!] = [\![\mathit{subst}(\mathit{expr}, T_r(\mathit{freevars}(\mathit{expr})))]\!]$, and this is not decidable in general. However, we can still heuristically deduce valid reuse vectors, if we know the internals structure of *expr*. Gautam and Rajopadhye [2006] proposed a heuristical approach by computing a polyhedral set $\hat{\mathcal{S}}_2$ called *share space*, which is the intersection of the nullspaces of the dependence functions of all the subexpressions of *expr*, and selecting any vector $r \in \hat{\mathcal{S}}_2$.

B.3 Choosing a reuse vector

Complexity reduction. We chose $\vec{r}$ to be in the *linealty space* $\mathcal{L}_{\mathcal{P}}$, that is defined informally as the subspace of $\mathcal{P}$ where $\mathcal{P}$ extends infinitely as the sizes the parameters of $\mathcal{P}$ tends to infinity. Intuitively, this means that we only want to reuse computation along directions of $\mathcal{P}$ that can grow asymptotically with the parameters, instead of directions that are bounded by fixed constants.

Sharing. We use a simple yet effective heuristic in our implementation: find the set of variables $\mathbf{fv}$ in the left hand side $\mathbf{u}$ that is not bound in the right hand side $\mathbf{v}$: i.e. $\mathbf{fv} = \mathbf{s} - \mathbf{v}$ (recall that $\mathbf{s}$ is the space of the statement's domain $\mathcal{P}$). For any variable $x \in \mathbf{fv}$, we can find its constant unit vector $\vec{r}_x$ (a vector under space $\mathbf{s}$ that is 1 only along direction x), we must have $\vec{r}_x$ satisfying the above criterion: $\forall \mathbf{v}. \mathit{expr} = \mathit{expr}(T_{r_x}(\mathbf{v}))$. In fact, any linear scaling of $\vec{r}_x$ is a valid choice. In summary, for one IR statement, there can be $|\mathbf{fv}|$ dimensions of reuse, in the sense that each $\vec{r}_x \forall x \in \mathbf{fv}$ are orthogonal to each other and thus forms different dimensions; for each dimension, there can be infinitely many valid reuse vectors, that are the different scalings of the unit direction for that dimension.

Dependence. Applying SR can introduce new dependencies along the reuse vector that was not in the original program. For example, in Listing 4.1, the transformed statement `T-add-reuse` introduces a new dependence that now `S`$[i]$ depends on `S`$[i-1]$. We require that applying SR along a vector $\mathbf{r}$ does not introduce any dependency cycle in transformed program so that it remains valid.

Appendix C

Enabling Transformations

In this section we briefly review the enabling transformations introduced in Gautam and Rajopadhye [2006]. Since these transformations are important to fully utilize ST for a single reduction, we encourage readers to find more details of these transformations in Gautam and Rajopadhye [2006].

Reduction Decomposition. For reduction with projection function *proj*, we can potentially decompose $proj = proj1 \circ proj2$, where $\circ$ denotes function composition. It is possible to break the reduction into two statements: the first statement with projection *proj2* produces an intermediate output, followed by a reduction with projection *proj1* that returns the original output. The first statement could lead to a larger share space than the original reduction, and therefore RD enables enable ST.

Same Operator Transformation. It's possible to lift inner expressions out of reductions to increase share space.

Distributivity Transformation. It's possible to utilizie distributivity of an operator to lift inner expression out of reductions to increase share space.

Higher Order Operator Transformation. It's possible to collapse along the entire reuse space, if the reduce operator $\oplus$ has an higher order operator $\otimes$.

Bibliography

Eric Atkinson, Cambridge Yang, and Michael Carbin. Verifying handcoded probabilistic inference procedures. In *arXiv e-prints*, 2018.

Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-11969-7, 978-3-642-11969-9. doi: 10.1007/978-3-642-11970-5_16. URL `http://dx.doi.org/10.1007/978-3-642-11970-5_16`.

Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. Pyro: Deep universal probabilistic programming. *arXiv preprint arXiv:1810.09538*, 2018.

Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. In *JMLR*, volume 3, 2003.

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, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375595. URL `http://doi.acm.org/10.1145/1375581.1375595`.

Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy array dataflow analysis. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, page 92–101, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917006. doi: 10.1145/209936.209947. URL `https://doi.org/10.1145/209936.209947`.

D Collett. *Modelling Survival Data in Medical Research. New York: Chapman and Hall/CRC*. 1993.

D. R. Cox. Regression models and life-tables. *Journal of the Royal Statistical Society: Series B (Methodological)*, 34(2):187–202, 1972. ISSN 0035-9246.

Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 221–236. ACM, 2019.

Greg Morisett Daniel Huang, Jean-Baptiste Tristan. Compiling markov chain monte carlo algorithms for probabilistic modeling. In *PLDI*, 2017.

Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. Polly's polyhedral scheduling in the presence of reductions. In *International Workshop on Polyhedral Compilation Techniques*, Amsterdam, Netherlands, Jan 2015.

E Ehrhardt. Sur un problème de géométrie diophantienne linéaire. ii. *Journal für die reine und angewandte Mathematik*, 1967, 12 2009. doi: 10.1515/crll.1967.227.25.

P. Feautrier. Array expansion. In *Proceedings of the 2Nd International Conference on Supercomputing*, ICS '88, pages 429–441, New York, NY, USA, 1988. ACM. ISBN 0-89791-272-1. doi: 10.1145/55364.55406. URL `http://doi.acm.org/10.1145/55364.55406`.

Paul Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *International Journal of Parallel Programming*, 21(5): 313–347, Oct 1992a. ISSN 1573-7640. doi: 10.1007/BF01407835. URL `https://doi.org/10.1007/BF01407835`.

Paul Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International Journal of Parallel Programming*, 21 (6):389–420, Dec 1992b. ISSN 1573-7640. doi: 10.1007/BF01379404. URL `https://doi.org/10.1007/BF01379404`.

Robert M. Fung and Kuo-Chu Chang. Weighing and integrating evidence for stochastic simulation on bayesian networks. In *UAI*, 1989.

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, 2006. ACM. ISBN 1-59593-027-2. doi: 10.1145/1111037.1111041. URL `http://doi.acm.org/10.1145/1111037.1111041`.

Andrew Gelman, Daniel Lee, and Jiqiang Guo. Stan: A probabilistic programming language for bayesian inference and optimization. *Journal of Educational and Behavioral Statistics*, 40(5):530–543, 2015.

Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1984.

Philip Ginsbach and Michael F. P. O'Boyle. Discovery and exploitation of general reductions: A constraint based approach. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 269–280. IEEE Press, 2017. ISBN 9781509049318.

Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. 2014. Accessed: 2016-10-7.

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *UAI*, 2008.

Scott Grauer-Gray and John Cavazos. Optimizing and auto-tuning belief propagation on the gpu. In Keith Cooper, John Mellor-Crummey, and Vivek Sarkar, editors, *Languages and Compilers for Parallel Computing*, pages 121–135, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-19595-2.

T. Griffiths and M. Steyvers. Finding scientific topics. In *PNAS*, volume 101, 2004.

Gautam Gupta, Kim Daegon, and Sanjay Rajopadhye. Scheduling in the z-polyhedral model. pages 1–10, 01 2007. doi: 10.1109/IPDPS.2007.370229.

Akshay Gupte, Shabbir Ahmed, Myun Cheon, and Santanu Dey. Solving mixed integer bilinear problems using milp formulations. *SIAM Journal on Optimization*, 23, 04 2013. doi: 10.1137/110836183.

W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. In *Biometrika*, volume 57, 1970.

Ian Holmes, Keith Harris, and Christopher Quince. Dirichlet multinomial mixtures: Generative models for microbial metagenomics. In *PLOS One*, 2012.

THE ARIC INVESTIGATORS. THE ATHEROSCLEROSIS RISK IN COMMUNIT (ARIC) STUDY: DESIGN AND OBJECTIVES. *American Journal of Epidemiology*, 129(4):687–702, 04 1989. ISSN 0002-9262. doi: 10.1093/oxfordjournals.aje.a115184. URL https://doi.org/10.1093/oxfordjournals.aje.a115184.

Ryoichi Kikuchi. A theory of cooperative phenomena. *Phys. Rev.*, 81:988–1003, Mar 1951. doi: 10.1103/PhysRev.81.988. URL https://link.aps.org/doi/10.1103/PhysRev.81.988.

Oleg Kiselyov. Probabilistic programming language and its incremental evaluation. pages 357–376, 11 2016. ISBN 978-3-319-47957-6. doi: 10.1007/978-3-319-47958-3_19.

Jun S. Liu. The collapsed gibbs sampler in bayesian computations with applications to a gene regulation problem. In *Journal of the American Statistical Association*, volume 89, 1994.

Yanhong A. Liu, Scott D. Stoller, Ning Li, and Tom Rothamel. Optimizing aggregate array computations in loops. *ACM Trans. Program. Lang. Syst.*, 27(1): 91–125, January 2005. ISSN 0164-0925. doi: 10.1145/1053468.1053471. URL `http://doi.acm.org/10.1145/1053468.1053471`.

Vikash Mansingkha, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. Probabilistic programming with programmable inference. In *PLDI*, 2018.

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. In *Journal of Chemical Physics*, volume 21, 1953.

Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, Cambridge, Massachusets, 2012.

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. Probabilistic inference by program transformation in hakaru (system description). In *FLOPS*, 2016.

David Newman. Bag of words dataset. In *UCI Machine Learning Respository*, 2008.

Aditya V. Nori, Sherjil Ozair, Sriram K. Rajamani, and Deepak Vijaykeerthy. Efficient synthesis of probabilistic programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 208–217, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686. doi: 10.1145/2737924.2737982. URL `https://doi.org/10.1145/2737924.2737982`.

David Padua, editor. *Omega Calculator*, pages 1355–1355. Springer US, Boston, MA, 2011. ISBN 978-0-387-09766-4. doi: 10.1007/978-0-387-09766-4_2303. URL `https://doi.org/10.1007/978-0-387-09766-4_2303`.

Martyn Plummer. *JAGS Version 4.0.0 user manual*. Addison-Wesley, Reading, Massachusetts, 2015.

Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *IEEE/ACM Fifth International Symposium on Code Generation and Optimization (CGO'07)*, pages 144–156, San Jose, California, March 2007. IEEE Computer Society press.

Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, Tucson, Arizona, June 2008. ACM Press.

Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, Ponnuswamy Sadayappan, and Nicolas Vasilache. Loop transformations: Convexity, pruning and optimization. *ACM SIGPLAN Notices*, 46:549–562, 05 2011. doi: 10.1145/1925844.1926449.

L. Rauchwerger and D. A. Padua. The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, Feb 1999. ISSN 2161-9883. doi: 10.1109/71.752782.

C. Reddy, M. Kruse, and A. Cohen. Reduction drawing: Language constructs and polyhedral compilation for reductions on gpus. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 87–97, Sept 2016. doi: 10.1145/2967938.2967950.

Philip Resnik and Eric Hardisty. Gibbs sampling for the uninitiated. In *UMIACS Technical Report*, June 2010.

Daniel Ritchie, Andreas Stuhlmüller, and Noah Goodman. C3: Lightweight incrementalized mcmc for probabilistic programs using continuations and callsite caching. In Arthur Gretton and Christian C. Robert, editors, *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, volume 51 of *Proceedings of Machine Learning Research*, pages 28–37, Cadiz, Spain, 09–11 May 2016. PMLR. URL `http://proceedings.mlr.press/v51/ritchie16.html`.

Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1986. ISBN 0-471-90854-1.

G. A. Susto, A. Schirru, S. Pampuri, S. McLoone, and A. Beghi. Machine learning for predictive maintenance: A multiple classifier approach. *IEEE Transactions on Industrial Informatics*, 11(3):812–820, June 2015. ISSN 1941-0050. doi: 10.1109/TII.2014.2349359.

Patricia M Therneau, Terry M.;Grambsch. *Modeling Survival Data: Extending the Cox Model*. Springer, New York, 2013. ISBN 9780387987842.

Dustin Tran, Matthew D Hoffman, Rif A Saurous, Eugene Brevdo, Kevin Murphy, and David M Blei. Deep probabilistic programming. *arXiv preprint arXiv:1701.03757*, 2017.

Peter Turnbaugh, Micah Hamady, Tanya Yatsunenko, Brandi Cantarel, Alexis Duncan, Ruth Ley, Mitchell Sogin, Joe Jones, Bruce A Roe, Jason Affourtit, Michael Egholm, Bernard Henrissat, Andrew C Heath, Rob Knight, and Jeffrey I Gordon. A core gut microbiome in obese and lean twins. 457:480–4, 12 2008.

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*, pages 299–302, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15582-6.

Sven Verdoolaege. Presburger formulas and polyhedral compilation, 01 2016.

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. ISSN 1432-0541. doi: 10.1007/s00453-006-1231-0. URL `https://doi.org/10.1007/s00453-006-1231-0`.

Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. On demand parametric array dataflow analysis. 01 2013. doi: 10.13140/RG.2.1.4737.7441.

Rajan Walia, Jacques Carette, Praveen Narayanan, Chung-chieh Shan, and Sam Tobin-Hochstadt. Efficient compilation of array probabilistic programs. In *arXiv e-prints*, 2018.

Nicola White, Fiona Reid, Adam Harris, Priscilla Harries, and Patrick Stone. A systematic review of predictions of survival in palliative care: How accurate are clinicians and who are the experts? *PLOS ONE*, 11(8):1–20, 08 2016. doi: 10.1371/journal.pone.0161407. URL `https://doi.org/10.1371/journal.pone.0161407`.

Yi Wu, Lei Li, Stuart Russell, and Rastislav Bodik. In *IJCAI*, 2016.

Lingfeng Yang, Patrick Hanrahan, and Noah Goodman. Generating Efficient MCMC Kernels from Probabilistic Programs. In Samuel Kaski and Jukka Corander, editors, *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, volume 33 of *Proceedings of Machine Learning Research*, pages 1068–1076, Reykjavik, Iceland, 22–25 Apr 2014. PMLR. URL `http://proceedings.mlr.press/v33/yang14d.html`.

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*, pages 17–31, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37658-0.

Jieyuan Zhang and Jingling Xue. Incremental precision-preserving symbolic inference for probabilistic programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 237–252, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314623. URL `https://doi.org/10.1145/3314221.3314623`.