Simplifying Dependent Reductions in the Polyhedral Model

CAMBRIDGE YANG, MIT CSAIL, USA
ERIC ATKINSON, MIT CSAIL, USA
MICHAEL CARBIN, MIT CSAIL, USA

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.

Contemporary polyhedral-based compilation techniques make it possible to optimize reductions, such as prefix sums, in which each component of the 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 dependent statements. In such cases, existing approaches can generate incorrect code that violates the data dependences of the original, unoptimized program.

In this work, we identify and formalize the optimization of dependent reductions as an integer bilinear program. We present a heuristic optimization algorithm that uses an affine sequential schedule of the program to determine how to simplfy reductions yet still preserve the program's dependences.

We demonstrate that the algorithm provides optimal complexity for a set of benchmark programs from the literature on probabilistic inference algorithms, whose performance critically relies on simplifying these reductions. The complexities for 10 of the 11 programs 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 speedups in wall-clock time that range from 1.1x to over 10^6x.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Theory of computation** → **Program analysis**; *Program semantics*; *Probabilistic computation*.

Additional Key Words and Phrases: reductions, polyhedral model, program dependence

1 INTRODUCTION

A *reduction* – an accumulation over a set of values, using an associative and commutative operator – is a common computation in many areas, including scientific computations, machine learning, computer vision, and financial analytics. For example, consider the prefix sum defined mathematically by Equation (1) and presented by Listing 1 in an imperative language with loops.

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. The complexity of this naïve prefix sum is $O(N^2)$: $O(N)$ for iterating over "$\forall i$", and $O(N)$ for the summation over j.

Authors' addresses: Cambridge Yang, MIT CSAIL, USA, camyang@csail.mit.edu; Eric Atkinson, MIT CSAIL, USA, eatkinson@csail.mit.edu; Michael Carbin, MIT CSAIL, USA, mcarbin@csail.mit.edu.

$$B[i] = \sum_{j=0}^{j \le i} A[j] \quad \forall i, 0 \le i < N \quad (1)$$

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

Listing 1. Naïve prefix sum

Shared Computation and Reuse. It is possible to implement prefix sum with $O(N)$ complexity, which is a linear speedup over this naïve implementation. The optimization relies on the fact that consecutive iterations of the loop that computes B (indexed by i) *share* equivalent computations: for any pair of consecutive iterations $[i]$ and $[i+1]$, the values of the entire set of computations $\{A[j] \mid j < i\}$ are the same. Therefore, the latter iteration shares the former iteration's entire computation: $\sum_{j=0}^{j<i} A[j]$. In principle, that shared computation can be *reused*: computed once in the former iteration, stored, and then reused in the latter iteration.

$$B[0] = A[0] \quad (2a)$$
$$B[i] = B[i-1] + A[i] \quad \forall i, 1 \le i < N \quad (2b)$$

```
1   // B is array of ints initialized to all 0
2   B[0] = A[0]
3   for(i = 1; i < N; i++)
4       B[i] = B[i−1] + A[i]
```

Listing 2. Optimized prefix sum

Optimized Reductions. Equation (2) – and, correspondingly, Listing 2 – presents an $O(N)$ implementation of prefix sum that reuses the shared computation between iterations. Instead, an iteration calculates its result, stores its result into B (as normal), and the subsequent iteration reuses the result of the previous iteration (via $B[i-1]$) to compute its own result.

1.1 Simplifying Reductions

Gautam and Rajopadhye [2006] developed a suite of polyhedral compilation techniques for *Simplifying Reductions* (SR) that can be applied to automatically transform Equation (1) to Equation (2) for an array equational language that supports reductions as a first class operation [Yuki et al. 2013]. Several of the challenges that these techniques solve are 1) identifying shared computation, 2) identifying if shared computation is *reusable*: if it's possible to transform the program to exploit the reuse), and 3) identifying if reusing shared computation is *profitable*: the transformation reduces the complexity of a program. SR provides a suite of specifications and techniques to identify shared computation as well as identify when shared computation is profitably reusable.[1]

Reuse Vector. The core of SR, the *Simplification Transformation* (ST), codifies the set of profitably reusable computations as a set of *reuse vectors*. For Equation (1), the reuse vector $[1, 0]^\top$ denotes the shared computation: changing i to $i+1$ and j to $j+0$ (i.e. not changing j) does not change the value ofthe reduction body $A[j]$. Given an equational statement and a reuse vector, ST automatically transforms the statement into a set of statements that together is semantically equivalent to the original statement, but reuses shared computation. For example, given Equation (1) and the reuse vector $[1, 0]^\top$, ST transforms Equation (1) to Equation (2).

Choosing a Reuse Vector. The space of valid reuse vectors is infinite in general, e.g., any vector $[c, 0]^\top$ with constant c is a valid choice for the reuse vector for Equation (1), since they all satisfy that changing from i to $i + c$ and not changing j does not change the evaluation of $A[j]$. Moreover,

---

[1]We refer readers to Appendix B.1 for additional explanation of how SR solves these problems. All appendices can be found in the supplementary materials section of the ACM Digital Library.

different reuse vectors result in different output equations by ST and, ultimately, different programs. As a concrete example, applying ST to Equation (1) with the reuse vector $[-1, 0]^\top$ produces Equation (3).

$$B[N - 1] = \sum_{j=0}^{j<N} A[j] \tag{3a}$$

$$B[i] = B[i + 1] - A[i] \quad \forall i, 0 \le i < N - 1 \tag{3b}$$

Instead of initializing $B[0]$ and computing $B[i]$ from lower indices to higher indices (i.e. left to right) as in Equation (2), Equation (3) initializes $B[N - 1]$ and computes $B[i]$ from higher to lower indices (i.e. right to left). However, this still has complexity $O(N)$, just as does Equation (2).

1.2 Dependent Reductions

The SR framework, including ST, only optimizes reductions that are independent. As a point of contrast, consider the following *dependent* reduction:

$$B[i] = \sum_{j=0}^{j\le i} A[j] \quad \forall i, 0 \le i < N \tag{4a}$$

$$A[i + 1] = f(B[i]) \quad \forall i, 0 \le i < N - 1 \tag{4b}$$

Equation (4) extends Equation (1) (equivalent to Equation (4a)) with an additional statement (Equation (4b)). The reduction in Equation (4a) is *dependent*: the value of the reduction $B[i]$ depends on the set of values $\{A[j] | j \le i\}$, while $A[i]$ depends on the previous value of the reduction $B[i-1]$.

Dependent reductions pose a challenge to ST because 1) applying ST introduces new dependences, and 2) the newly introduced dependences together with the program's existing dependences may incorrectly form a dependence cycle in the resultant program.

For example, applying ST to Equation (4a) with the reuse vector $[-1, 0]^\top$ produces a program consisting of three statements: Equations (3a), (3b) and (4b), which, together form a dependence cycle. Specifically, let $E_1 \xrightarrow{S} E_2$ denote a dependence induced by statement S between array entries E_1 and E_2. The path $B[N - 1] \xrightarrow{eq. (3a)} A[N - 1] \xrightarrow{eq. (4b)} B[N - 2] \xrightarrow{eq. (3b)} B[N - 1]$ forms a cycle.

On the other hand, applying ST to Equation (4a) with reuse vector $[1, 0]^\top$ produces a valid program consisting of three statements: Equations (2a), (2b) and (4b), and without any dependence cycle. Listing 3 presents a translation of this program to an imperative language with loops, which correctly computes array A and B with complexity $O(N)$.

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

Listing 3. Optimized prefix sum with dependent reductions

In summary, one key challenge for optimizing the dependent reductions is to augment ST to choose reuse vectors such that the augmented transformation produces programs that have no dependence cycles.

Approach. In this work, present a new technique to automatically optimize dependent reductions while soundly handling dependences that can automatically generate the code in Listing 3. We present a heuristic algorithm whose key idea is to use an affine sequential schedule of the program as a guide to choose among the multiple choices that can be made during the optimization process. Our results show that even though the algorithm does not consider other viable choices during optimization, given an affine sequential schedule of the program and all left-hand-side arrays of reductions, the algorithm is still optimal for reductions with operators that have inverses.

We note that our work relies on the techniques of Gautam and Rajopadhye [2006] to find resuable shared computation – i.e., reuse vectors – for a single reduction.[2] Our work addresses dependent reductions by further constraining reuse vectors to satisfy (intra/inter)-statement(s) dependences.

Applications. Simplifying Reductions is a classic problem in the compiler optimization literature that has reemerged as a primary concern for modern applications. In this work, we study a suite of 11 probabilistic inference algorithms that have been established as widely studied and used algorithms across data science, artificial intelligence, machine learning, computer vision, physics, and medicine. We demonstrate that dependent reductions exist in these algorithms' natural, mathematical specifications. Moreover, delivering efficient implementations of these algorithms by hand – as is current practice – requires solving the simplifying dependent reduction problem by hand, which is a tedious and error-prone endeavor. Our approach shows that it is possible to automatically generate optimized and efficient algorithms from their mathematical specifications alone.

Contributions. In this work, we present the following contributions:

- We identify the problem of simplifying dependent reductions, a problem that was not addressed by the Simplifying Reductions framework [Gautam and Rajopadhye 2006], which did not consider dependences. We illustrate the importance of this problem with examples from real applications.
- We formalize the task of optimizing a dependent reduction by combining the insights of the Simplifying Reductions 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 a benchmark suite 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 that we evaluate. In 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.[3] This is significant because for typical real application 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 1.1x to over 10^6x, with a median of 37x. 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, dependent reductions are a key ingredient of probabilistic inference algorithms, which are driving an emerging class of new programming languages and systems [Bingham et al. 2019; Cusumano-Towner et al. 2019; Daniel Huang 2017; Gelman et al. 2015; Goodman and Stuhlmüller 2014; Mansingkha et al. 2018; Narayanan et al. 2016; Tran et al. 2017] designed to streamline data science and enable new applications. Optimizing these algorithms has historically either been done by hand or has been baked in as a domain/algorithmic-specific 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 the dependent reduction as a general program pattern, detail its challenges, and propose a technique to optimize its performance.

Road Map. In Section 2, we illustrate a heuristic algorithm to address the simplifying dependent reduction problem described in Section 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 Sections 3 and 4, we review background on the polyhedral

---

[2]We include a description of the techniques in Appendix B
[3]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 8.3.2

model and SR, respectively. In Section 5 we formalize our problem as a integer bilinear program. In Section 6 we introduce our heuristic algorithm. In Sections 7 and 8 we discuss the implementation of our algorithm and its evaluation. In Sections 9 and 10 we summarize related work and end with concluding remarks, respectively.

2 EXAMPLE

In this section we give two examples. In Section 2.1, we walk through our approach with the prefix sum example. In Section 2.2, we use a practical application example to further motivate the importance of the dependent reduction problem.

2.1 Walk Through

In this section, we use the example of Equation (4) to 1) illustrate the steps of applying ST given a reuse direction, 2) illustrate an invalid reuse direction that leads ST (using the algorithm of Gautam and Rajopadhye [2006]) to induce a dependence cycle, and compare it to a valid reuse direction, and 3) describe the mechanism of our proposed heuristic algorithm.

Naive Prefix Sum. For ease of comparison and better visualization, we present the input in Equation (4) with Figure 1, a visual, polyhedral interpretation of the naive prefix sum program in Equation (4). In Figure 1, the top polyhedron with round dots represents the iteration domain of the reduction statement, $B[i]$ += $A[j]$, with each round dot denoting an iteration instance of the statement. To the right of the top polyhedron,

Fig. 1. Naive prefix sum (Equation (4))

we have labeled each round dot in the top polyhedron at coordinate (i, j) by the array element $A[j]$ that should be accumulated into $B[i]$. The bottom polyhedron with squares represents the iteration domain for the statement $A[i + 1] = f(B[i])$. The middle polyhedron with diamonds is an additional polyhedron that our technique inserts into the program's polyhedral representation to denote the completion of each reduction $B[i]$, which we label as $\sum A[j]$ in the diagram.

Data Dependences. Each arrow in Figure 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 dependences:

- **Reduction.** Each point in the middle polyhedron depends on all the points in the respective column of the top polyhedron. These dependences are those of the reduction.
- **Use.** Each point in the bottom polyhedron depends on the point in the corresponding column of the middle polyhedron. These dependences are those from the use of the reduction results.
- **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 dependences are those induced by the update to $A[i + 1]$ in Equation (4b) and use by Equation (4a).

Correct Optimization. Figure 2 presents two diagrams, corresponding to an intermediate step and the result of a correct application of ST, respectively.

Figure 2a presents the intermediate step of ST. In this step, the algorithm chooses a reuse vector and shifts the reduction statement's iteration domain along the vector. Figure 2a illustrates the shift along a correct reuse vector, $[1, 0]^\top$, which maps iteration instances $[i, j]$ (round dots in the
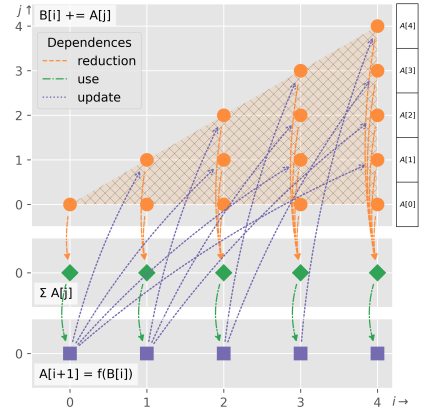
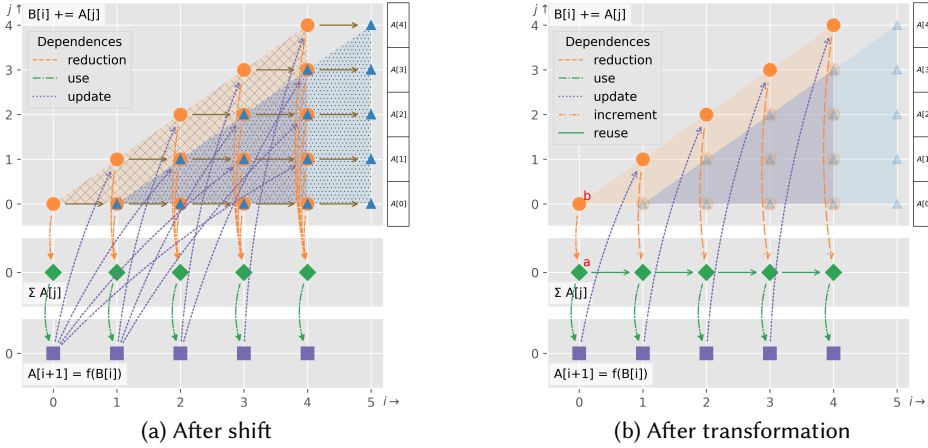(a) After shift   (b) After transformation

Fig. 2. Correct optimization of prefix sum with depedent reduction

crosshatched polyhedron) to instances $[i + 1, j]$ (triangles in the dot-shaded polyhedron). Each solid arrow represents the mapping from an instance in the polyhedron with round dots to its counterpart in the polyhedron with triangles. The triangles outlined in round dots (i.e. overlapped triangles and round dots) are the points corresponding to redundant computations. Specifically, because the reuse vector has the property that the evaluations of the reduction's 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 must have the same value as the evaluation of a reduction over the column to the left of col. ST therefore eliminates this intersection part of the domain by reusing previously computed reductions (i.e. computes $B[i]$ from $B[i - 1]$ by incrementally using points not in the intersection).

Figure 2b presents the result of ST, where ST has eliminated the redundant computations. Figure 2b corresponds to the resulting polyhedron and dependences after ST eliminates redundant computations. Specifically, each instance in the intersection of the two polyhedrons has been eliminated, along with its induced dependences.

To map Figure 2b to Equation (2): the point a in the middle polyhedron and the point b in the top polyhedron in Figure 2b correspond to the reduction that initializes $B[0]$ in Equation (2a). All points in the middle polyhedron except a then correspond to Equation (2b), i.e., each $B[i]$ is computed by adding the predecessor point $B[i - 1]$ with $A[i]$.

Dependences in Figure 2b are preserved from Figure 2a for all the non-eliminated instances. Figure 2b introduces the following new dependences:

- **Reuse.** Each instance in the middle polyhedron (except the leftmost instance) now depends on the instance to its left, along the reuse vector. These dependences are those from reusing.
- **Increment.** Each instance in the middle polyhedron now depends on the corresponding instance in the top polyhedron in the same column. These dependences are those from incrementalizing.

Incorrect Optimization. The two diagrams in Figure 3 illustrate an incorrect application of ST using Gautam and Rajopadhye [2006], which ignores the dependences due to dependent reduction. In this case, instead of using the correct reuse vector $[1, 0]^\top$, this application of ST uses the vector $[-1, 0]^\top$. This vector maps iteration instances $[i, j]$ to instances $[i - 1, j]$.

Figure 3a presents the intermediate step of ST. Same as in the correct optimization's case, in this step, the algorithm chooses the a reuse vector and shifts the reduction's domain along the vector. Figure 3a illustrates the shift along the incorrect reuse vector, $[-1, 0]^\top$, which maps iteration instances $[i, j]$ (round dots in the crosshatched polyhedron) to instances $[i - 1, j]$ (triangles in the

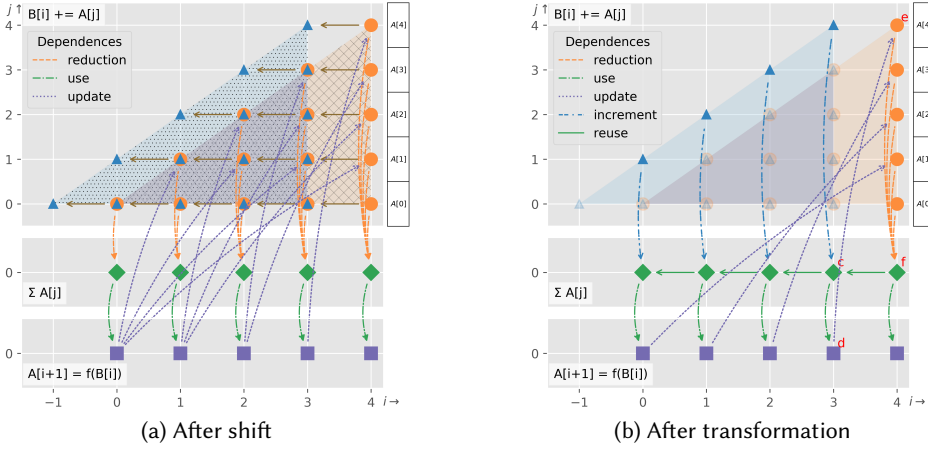(a) After shift   (b) After transformation

Fig. 3. Incorrect optimization of prefix sum with depedent reduction

dot-shaded polyhedron). Each solid arrow again represents the maping between the corresponding instances before and after the shift. The triangles outlined in round dots again are the instances corresponding to redundant computations to be eliminated by ST.

Figure 3b presents the result of ST, where ST has, again, eliminated the redundant computations. The top polyhedron now has the round dots at the rightmost column and the triangles along the hypotenuse of the shifted domain. Note that the hypotenuse is restricted to the domain of projected domain of the reduction and does not include the point $[i, j] = [-1, 0]$.

To map the Figure 3b to Equation (3): the point f and the round dot column in the top polyhedron in Figure 3b correspond to the reduction that initializes $B[N - 1]$ in Equation (3a). All points in the middle polyhedron except d then correspond to Equation (3b), i.e., each $B[i]$ is computed by subtracting the successor point $B[i + 1]$ by $A[i]$.

However, as mentioned in Section 1, Figure 3b's dependences form cycles (e.g., points c, d, e, f form a cycle). Therefore, the transformed program does not have a valid schedule, and consequently the ST application along the reuse vector with mapping $[i, j] \rightarrow [i-1, j]$ is an incorrect optimization.

Heuristic for Choosing a Valid Direction. As we have seen from the previous illustration, it is important to choose a valid reuse vector for dependent reductions. In this work, we propose a heuristic algorithm for choosing a valid reuse vector. Notably, one key difference between Figures 2 and 3 is the dependences drawn on the middle polyhedron. Specifically, in the middle polyhedron of Figure 2, the drawn dependences on B[i] respect a sequential, scheduled computation order of B[i] of the original program in Figure 1, whereas that of Figure 3 disobeys that scheduled order. This observation has inspired the heuristic algorithm, which always chooses the reuse vector that is consistent with a sequential scheduled computation order of the left hand side 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 Simplying Dependent Reductions in Practice

As we later show in Section 8 by studying a variety of benchmarks, dependent reductions appear in the specifications of many problems and algorithms across statistics, artificial intelligence (AI), and machine learning (ML) with applications to computer vision, physics, and medicine. However, the common practice is to develop these algorithms by hand. Therefore, our technique offers the opportunity to automatically translate a specification to an efficient implementation. In this section, we illustrate our technique on a clustering application used across statistics, AI, and ML.

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). This computation is designed to cluster data points such that similar data points, also called *observations*, are assigned to the same cluster. The input to GS-2GMM is a float array Obs that represents the observations. 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 Gibbs sampling procedure samples the array Z that represents the cluster membership of all given observations. It does this by iteratively taking in an old cluster assignment for all observations, and resampling a new assignment by updating the individual assignment of a single observation. This process will produce a stream of samples of Zs that approach the true distribution of Z. The mathematical specification of GS-2GMM is given in Equation (5).

$$C_{zi} = \sum_{\forall j \ \text{s.t.} \ j \neq i \wedge Z_j = z} 1 \quad , \forall z, i \tag{5a}$$

$$S_{zi} = \sum_{\forall j \ \text{s.t.} \ j \neq i \wedge Z_j = z} obs_i \quad , \forall z, i \tag{5b}$$

$$P_o(z, i) = \mathcal{N}\big(\frac{S_{zi}}{C_{zi}}, (1 + C_{zi})^{-1} + 1\big) \stackrel{def.}{=} P(obs_i | obs_{\setminus i}, Z_{\setminus i}, Z_i = z) \tag{5c}$$

$$P_z(i) = \frac{P_o(0, i)}{P_o(0, i) + P_o(1, i)} \stackrel{def.}{=} P(Z_i = 0 | Z_{\setminus i}, obs) \tag{5d}$$

$$Z_i \sim P_z(i) \quad , \forall i \in \{1 \dots N\} \tag{5e}$$

In Equations (5a) and (5b), 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). Equation (5c) defines the function P_o as an abbreviation of a normal distribution that represents the distribution of obs_i given all values of obs except obs_i and all current assignments of Zs except fixing Z_i to be z. We use the notation $\setminus i$ to denote the set $\{j \mid 1 \leq j \leq N \wedge j \neq i\}$. Equation (5d) defines the function P_z as an abbreviation of a probability representing the chance tha Z_i is equal to 0 given all values of obs and all current assignments Z except Z_i. Lastly, Equation (5e) samples each Z_i in order from its distribution. Note that the exact computations required to perform Gibbs sampling are not important for understanding the optimization problem.

Listing 4 gives an efficient implementation of the above mathematical specification – notice that Listing 4 computes the counts and sums incrementally, instead of forming the full reductions of Equations (5a) and (5b). Deriving Listing 4 from Equation (5) requires manually solving the simplifying dependent reductions problem which is tedious and error-prone.

Our Approach. Given an array-based representation of Equation (5), our approach automatically produces Listing 4. For conciseness of presentation, we consider the variable S_{zi} with fixed $z = 0$ as an example. In this case Equation (5b) can be rewritten as sum of two variables $S_{0i} = \text{S0L}[i] + \text{S0R}[i]$, where S0L, S0R are given by Equations (6a) and (6b), respectively.

$$\text{S0L}[i] = \sum_{j=0}^{j<i} (\text{if } \text{Z}[j] == 0 \text{ then } \text{Obs}[j] \text{ else } 0) \tag{6a}$$

$$\text{S0R}[i] = \sum_{j=i+1}^{j<N} (\text{if } \text{Z}[j] == 0 \text{ then } \text{Obs}[j] \text{ else } 0) \tag{6b}$$

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

$$\text{Z'}[i] = \text{sample}(\text{S0L}[i] + \text{S0R}[i], ...) \tag{6c}$$

```
1   int[N] C0L, C1L, C0R, C1R = {0...} // Zero initialize
2   float[N] S0L, S1L, S0R, S1R = {0...} // Zero initialize
3   for(i = 1; i < N; i++)
4     C0R[0] += (Z[i] == 0 ? 1 : 0)
5     C1R[0] += (Z[i] == 1 ? 1 : 0)
6     S0R[0] += (Z[i] == 0 ? Obs[i] : 0)
7     S1R[0] += (Z[i] == 1 ? Obs[i] : 0)
8   for(i = 0; i < N; i++)
9     // Sample according to Equations (5c) to (5e)
10    Z'[i] = sample(C0L[i] + C0R[i], C1L[i] + C1R[i], S0L[i] + S0R[i], S1L[i] + S1R[i])
11    // Incremental updates
12    C0L[i] = C0L[i−1] + (Z'[i] == 0 ? 1 : 0)
13    C1L[i] = C1L[i−1] + (Z'[i] == 1 ? 1 : 0)
14    S0L[i] = S0L[i−1] + (Z'[i] == 0 ? 1 : 0)
15    S1L[i] = S1L[i−1] + (Z'[i] == 1 ? 1 : 0)
16    C0R[i] = C0R[i−1] − (Z[i] == 0 ? 1 : 0)
17    C1R[i] = C1R[i−1] − (Z[i] == 1 ? 1 : 0)
18    S0R[i] = S0R[i−1] − (Z[i] == 0 ? Obs[i] : 0)
19    S1R[i] = S1R[i−1] − (Z[i] == 1 ? Obs[i] : 0)
```

Listing 4. Correct optimized GS-2GMM with dependent reduction; (a ? b : c) denotes if-then-else expression as in the C language

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 is 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 a simple if-then-else expression guarding the reduction's body – this is standard approach and same as the one proposed by Benabderrahmane et al. [2010] to model non-affine constraints as control predicates.

Equations (6a) and (6c) exactly correspond to Equations (4a) and (4b), respectively, since they have the same data flow dependences [4]. Thus the technique walked through in Section 2.1 also applies to Equations (6a) and (6c) to produce a specification with efficient complexity. Further, our technique is general in that it handles any dependent reduction, including Equation (6b) with constraints $i + 1 \leq j < N$, which are the reverse of the constraints in Equation (6a) . Lastly, the same analysis can be applied to all cases of C_{zi} and S_{zi} with $z = 0$ or $z = 1$. The analyses in total produces eight intermediate variables, namely `C0L`, `C1L`, `C0R`, `C1R`, `S0L`, `S1L`, `S0R`, `S1R`, which produce Listing 4 by applying our technique and compiling to executable code.

Results. Our evaluation in Section 8 shows that our technique produces an optimal complexity algorithm for Gibbs Sampling on the Gaussian Mixture Model, matching that of a manually developed implementation, and yielding a 7.1x performance improvement over a naive, unoptimized implementation. These results demonstrate the opportunity to automatically compile high-level specifications that include dependent reductions to efficient implementations.

3 BACKGROUND: POLYHEDRAL MODEL

In this section, we review the terminology from the *polyhedral model* that we use in this work. The polyhedral model represents a program by a set of statements, and for each statement, an associated

---

[4]Although Equation (6c) contains `sample` that is stochastic and Equation (4b) contains `f` that is deterministic, they still have the same data flow dependences.

```
S1 : BTmp[i] += A[j] : {[i, j] : 0 ≤ i < N ∧ 0 ≤ j ≤ i}
S1Fin : B[i] = BTmp[i] : {[i] : 0 ≤ i < N}
S2 : A[i + 1] = f(B[i]) : {[i] : 0 ≤ i < N − 1}
```

Listing 5. Equation (4) in the polyhedral IR

polyhedral set known as the statement's *domain*. Each point in a polyhedral set corresponds to one concrete execution instance of the statement.

Listing 5 gives an example of Equation (4) our polyhedral intermediate representation. Each line is a polyhedral statement to which we have affixed a label to the left: S1, S1Fin, and S2, respectively. The polyhedral statement S1 corresponds to Equation (4a), and S2 corresponds to Equation (4b). To aid understanding, we have added a statement, S1Fin, that does not map directly to an equation in Equation (4). The statement denotes the completion of the computation (the reduction) for each element of B and corresponds to the middle polyhedra in Figures 1 to 3.

3.1 Polyhedral Representations

Polyhedral Set. The notation that follows the second colon of each line of Listing 5 denotes the *polyhedral set* that defines the statement's domain. The statement executes once for each point in the set. We introduce the following definition and notation for a polyhedral set; the notation is consistent with the Integer Set Library (ISL) [Verdoolaege 2010]'s notation.

Definition 3.1 (Polyhedral set). A polyhedral set $\mathcal{P}$ is defined as $[\vec{p}] \rightarrow \{[\vec{x}] : M \cdot [\vec{x}, \vec{p}, 1]^\top \geq \vec{0}\}$, which consists of a vector of parameters $[\vec{p}]$, a vector template $[\vec{x}]$, and a *system of affine inequalities* $M \cdot [\vec{x}, \vec{p}, 1]^\top \geq \vec{0}$, where M is an $m \times (|\vec{x}| + |\vec{p}| + 1)$ matrix of constant integers. In addition, $[\vec{p}] \rightarrow \{[\vec{x}]\}$ is the called the space of $\mathcal{P}$.

A polyhedral set provides an intensional description of 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 in $[\vec{p}]$, if $[\vec{p}]$ is not empty. For example, the polyhedral set for statement S1Fin in Listing 5 is

$$[N] \rightarrow \{[i] : \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} i \\ N \\ 1 \end{bmatrix} \geq \vec{0}\}$$ that denotes the set of integer values of i from 0 to $N - 1$.

Specifically, each row of M denotes an inequality. Therefore, the inequalities in this example are $(i \geq 0) \wedge (i \leq N - 1)$ — or simply the shorthand $0 \leq i < N$. An equality $i = 0$ is shorthand for the conjunction of two inequalities $(i \geq 0) \wedge (-i \geq 0)$.

Polyhedral Relation. To give semantics to our polyhedral intermediate representation (e.g. Listing 5), we also introduce the following definition for a *polyhedral relation*, also in accordance with ISL's notation.

Definition 3.2 (Polyhedral relation). A polyhedral relation $[\vec{p}] \rightarrow \{[\vec{x_1}] \rightarrow [\vec{x_2}] : M \cdot [\vec{x_1}, \vec{x_2}, \vec{p}, 1]^\top \geq \vec{0}\}$ contains a vector of parameters $[\vec{p}]$, vector templates $[\vec{x_1}], [\vec{x_2}]$, and a system of affine inequalities $M \cdot [\vec{x_1}, \vec{x_2}, \vec{p}, 1]^\top \geq \vec{0}$.

Fig. 4. Plot of example the polyhedral relation $[N] \rightarrow \{[i, j] \rightarrow [i + 1, j] : 0 \leq i < N, 0 \leq j < N\}$

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 in $[\vec{p}]$. For example, $[N] \rightarrow \{[i, j] \rightarrow [i + 1, j] : 0 \leq i < N \wedge 0 \leq j < N\}$ denotes the relation that maps every integer tuple $[i, j]$ to $[i + 1, j]$ within an N-by-N grid. Figure 4 visualizes this relation for $N = 5$: the arrows map points corresponding to integer tuples to their right successors.
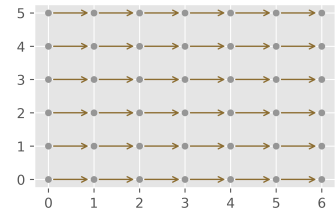
For aesthetic reasons, we omit the parameter $[\vec{p}]$ when it is clear which identifiers are parameters.

3.2 Polyhedral Representation of a Program

$$P := S+$$
$$S := X \ (= \ | \ \oplus=) \ E \ : \ \mathcal{P}$$
$$E := E? \oplus E \ | \ X \ | \ c$$
$$X := a \ [\ (A,)* \ A \ ]$$

Fig. 5. IR Grammar

Syntax. Following the 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 Figure 5. We explain each component in turn:

- P is a program that consists of multiple statements.
- S is a statement that consists of a left hand side array access X, a middle assignment operator (i.e. either = or $\oplus=$) , a right hand side expression (i.e. E), and a domain (i.e. $\mathcal{P}$). A statement is called a normal statement when the middle assignment operator is plain =; it is called a *reduction* when the middle assignment operator is $\oplus=$. The reduction operator $\oplus$ is associative and commutative, and has an identity (e.g. 0 is the identity addition, and 1 is the identify for multiplication).
- E is an expression that is either an unary or binary operator applied on expression(s), an array access (i.e. X), or a constant.
- A is an affine expression, a kind of expression that applies an affine transformation to variables and produces a scalar. It references only variables in $\vec{x}$ or $\vec{p}$, where $[\vec{p}] \rightarrow \{[\vec{x}]\}$ is the space of $\mathcal{P}$.
- X is an affine array access that consists of an array identifier (i.e. a), and a (comma seperated) list of affine expressions (i.e. As). A list of affine expressions of length n can be expressed mathematically as an affine transformation $M \cdot [\vec{x}, \vec{p}, 1]^\top$, where M is a constant $n \times (|\vec{x}| + |\vec{p}| + 1)$ integer matrix and $\vec{x}, \vec{p}$ are defined same as those for an affine expression A.
- $\mathcal{P}$ is a polyhedral set representing the statement's domain. Each point in the domain corresponds to one concrete execution instance of the statement. If $\mathcal{P}$ is $[p] \rightarrow \{[t] : e\}$, then p corresponds to the set of parameters of the program and t corresponds to the loop variables of the statement.

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, an expression a[N - i + j + 1] with domain $[N] \rightarrow \{[i, j] : 0 \leq i < N \land 0 \leq j < N\}$, evaluates to the value of a[9] at the point $[i, j]^\top = [1, 0]^\top$ and given that $N = 10$.

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, we first define the *projection* of a reduction to be a polyhedral relation that maps points in $\mathcal{P}$ to their accessed indices of the left hand side array. For example, for the reduction S1 in Listing 5, the projection is the polyhedral relation $[N] \rightarrow \{[i, j] \rightarrow [i] : 0 \leq i < N \land 0 \leq j \leq i\}$. Then, for each point $p \in \mathcal{P}$ the right hand side expression is evaluated and accumulated into the left hand side array at point $p' = proj(p)$ using the operator $\oplus$, where *proj* is the projection of the reduction.

Computability. In general, programs in the equation-based representation (Figure 5) may not be computable due to cyclic dependences. Saouter and Quinton [1993] showed that detecting cycles for these programs is undecidable.

In this work, we focus on programs in the equational-based representation (Figure 5) that do not have cyclic dependences, admit legal schedules, and are always computable.

3.3 Polyhedral Model Scheduling

Scheduling is a step in the polyhedral model where a scheduling function assigns each point in a statement's domain a timestamp, denoting the order of all execution instances. The task of

scheduling a program in the polyhedral model is to find a schedule Θ for the program such that the schedule timestamps for all statements satisfy the dependence relations of the program.

3.3.1 Dependence Analysis. In scheduling, dependence analysis compute the *dependence relations* of the program. Here we recall concepts in dependence analysis and breifly describe how to compute the dependence relation bewteen two statements in the program.

Access Relation. An *access relation* is a polyhedral relation mapping from the space of the domain of a statement to the space of an accessed array. An access relation can either be a *write access relation* (when the access is a write to an array on the left hand side of a statement), or a read access relation (when the access is a read to an array on the right hand side of a statement). Let $array[A_1, \ldots, A_k]$ be an array reference in a statement with space $[\vec{p}] \rightarrow \{[\vec{x}]\}$, and the list of affine expressions $A_1, \ldots, A_k$ is expressed as $M \cdot [\vec{x}, \vec{p}, 1]^\top$. The access relation for this array access is $[p] \rightarrow \{[\vec{x}] \rightarrow [\vec{y}] : M \cdot [\vec{x}, \vec{p}, 1]^\top = \vec{y}\}$.

For example, in Listing 5, since each iteration instance $[i, j]$ of S1 writes to BTmp$[i]$, the write access relation of S1 is $[N] \rightarrow \{[i, j] \rightarrow [i] : 0 \le i < N \wedge 0 \le j < i\}$. Also, since each iteration instance $[i, j]$ of S1 reads A$[j]$, the read access relation of S1 is $[N] \rightarrow \{[i, j] \rightarrow [j] : 0 \le i < N \wedge 0 \le j < i\}$.

Array 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. This means for each unique left hand side array, and the statements $S_0 \ldots S_k$ that write to it, $\bigcap_i \mathcal{W}_{S_i} = \emptyset$, where $\mathcal{W}_{S_i}$ is the write access relation for S_i.

Dependence Relation. Any two statements S_1, S_2 must satisfy a dependence relation represented by a polyhedral relation $\mathcal{D}_{S_1,S_2} = [\vec{p}] \rightarrow \{[\vec{x}^{S_1}] \rightarrow [\vec{x}^{S_2}] : M^{\mathcal{D}_{S_1,S_2}} \cdot [\vec{x}^{S_1}, \vec{x}^{S_2}, \vec{p}, 1]^\top \ge \vec{0}\}$, where $M^{\mathcal{D}_{S_1,S_2}}$ is the *dependence matrix*. The dependence relation $\mathcal{D}_{S_1,S_2}$ describes the happens before relation between iterations of S_1 and S_2. For a pair of statements S_1, S_2, let S_1 write to an array a and let S_2 read from a. The dependence relation $\mathcal{D}_{S_1,S_2}$ is equal to $\mathcal{R}^{-1} \circ \mathcal{W}$, where $\mathcal{R}$ and $\mathcal{W}$ are the two statements' read and write access relations w.r.t. a, 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 a textbook [Verdoolaege 2016] contain detailed introductions to dependence analysis techniques, which we refer the reader to for deeper exposure.

3.3.2 Scheduling Function. Here we recall the definition of a program's schedule.

Definition 3.3 (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 τ_1 and τ_2, $\tau_1 < \tau_2$ (τ_1 happens before τ_2) iff $\tau_1[i] < \tau_2[i]$ where i is the first non-equal index between τ_1, τ_2.

A schedule Θ for a program is a collection of *scheduling functions*, one for each statement. A scheduling function for a statement S is an affine transformation represented by the matrix Θ^S, which maps statement S's domain to its scheduling timestamp. For a statement S with domain in space $[\vec{p}] \rightarrow \{[\vec{x}^S]\}$, the schedule function is an $m \times (|\vec{x}^S| + |\vec{p}| + 1)$ matrix. The statement's m dimensional timestamp τ^S is given by:

$$\tau^S = \Theta^S \cdot \begin{bmatrix} \vec{x}^S \\ \vec{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \Theta_{1,1} & \cdots & \Theta_{1,|\vec{x}^S|+|\vec{p}|+1} \\ \vdots & \cdots & \vdots \\ \Theta_{m,1} & \cdots & \Theta_{m,|\vec{x}^S|+|\vec{p}|+1} \end{bmatrix} \cdot \begin{bmatrix} \vec{x}^S \\ \vec{p} \\ 1 \end{bmatrix} \tag{7}$$

3.3.3 ILP Formulation Of Scheduling. Early works [Feautrier 1992a,b] gave a greedy algorithm to the scheduling problem and provided the foundation of the scheduling problem. Pouchet et al.

[2011] formalized the scheduling problem for obtaining an m-dimensional schedule as the following single convex problem:

$$\forall \mathcal{D}_{S_1,S_2}, \forall k \in \{1...m\}, \delta_k^{\mathcal{D}_{S_1,S_2}} \in \{0,1\} \tag{8a}$$

$$\forall \mathcal{D}_{S_1,S_2}, \sum_{k=1}^{m} \delta_k^{\mathcal{D}_{S_1,S_2}} = 1 \tag{8b}$$

$$\forall \mathcal{D}_{S_1,S_2}, \forall k \in \{1...m\}, \forall [\vec{x}^{S_1}, \vec{x}^{S_2}, \vec{p}] \in \mathcal{D}_{S_1,S_2}$$

$$\Theta_k^{S_2} \cdot \begin{bmatrix} \vec{x}^{S_2} \\ \vec{p} \\ 1 \end{bmatrix} - \Theta_k^{S_1} \cdot \begin{bmatrix} \vec{x}^{S_1} \\ \vec{p} \\ 1 \end{bmatrix} \geq \delta_k^{\mathcal{D}_{S_1,S_2}} - \sum_{i=1}^{k-1} \delta_i^{\mathcal{D}_{S_1,S_2}} (K\vec{p} + K) \tag{8c}$$

where Θ_k^S denotes the k-th row of the matrix Θ^S.

In words, Equation (8a) creates a binary variable $\delta_k^{\mathcal{D}_{S_1,S_2}}$ for each dependence relation in the program and for each of the $k \in \{1...m\}$ dimensions of the schedule.[5] Each variable models the comparison between the two m-dimensional timestamps of statements S_1 and S_2 at the k-th dimension. Equation (8b) specifies that $\delta_k^{\mathcal{D}_{S_1,S_2}}$ is only strongly satisfied once for all k. Equation (8c) encodes the constraint that the schedule functions Θ^{S_1} and Θ^{S_2} must satisfy that $\vec{x}^{S_1}$ is scheduled before $\vec{x}^{S_2}$, if the dependence $\vec{x}^{S_1} \to \vec{x}^{S_2}$ exists. Specifically, the dependence $\vec{x}^{S_1} \to \vec{x}^{S_2}$ exists if $[\vec{x}^{S_1}, \vec{x}^{S_2}, \vec{p}]^\top \in \mathcal{D}_{S_1,S_2}$. 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] show that this problem is equivalent to an ILP thanks to Farkas' Lemma [Schrijver 1986]. Specifically, let $M^{\mathcal{D}_{S_1,S_2}}$ denote the constraint matrix of $\mathcal{D}_{S_1,S_2}$ given in Definition 3.2 (i.e. $\mathcal{D}_{S_1,S_2}$ is $[\vec{p}] \to \{[\vec{x}^{S_1}] \to [\vec{x}^{S_2}] : M^{\mathcal{D}_{S_1,S_2}} \cdot [\vec{x}^{S_1}, \vec{x}^{S_2}, \vec{p}, 1]^\top \geq \vec{0}\}$). Farkas' Lemma makes it possible to introduce a vector of integer variables, $\Lambda_k^{\mathcal{D}_{S_1,S_2}}$, of length n, where n is one plus the number of rows of $M^{\mathcal{D}_{S_1,S_2}}$, for all $\mathcal{D}_{S_1,S_2}$ and $k \in \{1...m\}$. It is then possible to expand Equation (8c) to the following constraints.

$$\forall \mathcal{D}_{S_1,S_2}, \forall k \in \{1...m\},$$

$$\Lambda_k^{\mathcal{D}_{S_1,S_2}} \geq 0 \tag{9a}$$

$$\Theta_k^{S_2} \cdot \begin{bmatrix} \vec{x}^{S_2} \\ \vec{p} \\ 1 \end{bmatrix} - \Theta_k^{S_1} \cdot \begin{bmatrix} \vec{x}^{S_1} \\ \vec{p} \\ 1 \end{bmatrix} - \delta_k^{\mathcal{D}_{S_1,S_2}} + \sum_{i=1}^{k-1} \delta_i^{\mathcal{D}_{S_1,S_2}} (K\vec{p}+K) = \left(\Lambda_k^{\mathcal{D}_{S_1,S_2}}\right)^\top \cdot \begin{bmatrix} M^{\mathcal{D}_{S_1,S_2}} \\ \vec{0} & 1 \end{bmatrix} \cdot \begin{bmatrix} \vec{x}^{S_1} \\ \vec{x}^{S_2} \\ \vec{p} \\ 1 \end{bmatrix} \tag{9b}$$

By equating the left and right side of Equation (9b) for all coefficients of $\vec{x}^{S_1}, \vec{x}^{S_2}, \vec{p}$ and the constant terms produces the desired affine constraints in the ILP formulation of scheduling. Solving the above formulation produces the desired schedule coefficients Θ in Section 3.3.2.

3.3.4 Example. We give an example of the above ILP formulation of scheduling, using the example from Section 1 (Equation (4) and its IR form in Listing 5). Let S_1 refer to statement S1 and S_2 refer to statement S2 in Listing 5. The dependence relations between S_1 and S_2 are:

$$\mathcal{D}_{S_1,S_2} = [N] \to \{[i^{S_1}, j^{S_1}] \to [i^{S_2}] : i^{S_1} = i^{S_2} \wedge 0 \leq i^{S_1} < N - 1 \wedge 0 \leq j^{S_1} \leq i^{S_1}\} \tag{10a}$$

$$\mathcal{D}_{S_2,S_1} = [N] \to \{[i^{S_2}] \to [i^{S_1}, j^{S_1}] : i^{S_1} = i^{S_2} + 1 \wedge 0 \leq i^{S_2} < N - 1 \wedge 0 \leq j^{S_1} \leq i^{S_2}\}. \tag{10b}$$

The corresponding constraint matrices for $\mathcal{D}_{S_1,S_2}$ and $\mathcal{D}_{S_2,S_1}$ are given in Equation (11).

---

[5]If the dependence relation between S_1, S_2 is a union of polyhedral relations, then we consider each piece of the union as distinct and set up the constraints in Equation (8) for each piece of the union.

$$M^{\mathcal{D}_{S_1,S_2}} = \begin{bmatrix} -1 & 0 & 0 & 1 & -2 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \end{bmatrix} \quad \text{(11a)} \qquad M^{\mathcal{D}_{S_2,S_1}} = \begin{bmatrix} -1 & 0 & 0 & 1 & -2 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 & -1 \\ 1 & -1 & 0 & 0 & 1 \end{bmatrix} \quad \text{(11b)}$$

Assuming the schedule timestamp's dimension is $m = 3$ and K is a large enough constant, we formulate the dependence constraints as in Equations (8c) and (9) by Equation (12).

$$0 \le \delta_k^{\mathcal{D}_{S_1,S_2}} \le 1 \qquad 0 \le \delta_k^{\mathcal{D}_{S_2,S_1}} \le 1 \qquad \forall k \in \{1...3\} \qquad \text{(12a)}$$

$$\sum_{k=1}^{3} \delta_k^{\mathcal{D}_{S_1,S_2}} = 1 \qquad \sum_{k=1}^{3} \delta_k^{\mathcal{D}_{S_2,S_1}} = 1 \qquad \text{(12b)}$$

$$\Lambda_k^{\mathcal{D}_{S_1,S_2}} \ge 0 \qquad \Lambda_k^{\mathcal{D}_{S_2,S_1}} \ge 0 \qquad \forall k \in \{1...3\} \qquad \text{(12c)}$$

$$\begin{cases} i^{S_1} : & -\Theta_{k,1}^{S_1} = \Lambda_k^{\mathcal{D}_{S_1,S_2}} \cdot [-1,0,1,-1,1,0] \\ j^{S_1} : & -\Theta_{k,2}^{S_1} = \Lambda_k^{\mathcal{D}_{S_1,S_2}} \cdot [0,1,-1,0,0,0] \\ i^{S_2} : & \Theta_{k,1}^{S_2} = \Lambda_k^{\mathcal{D}_{S_1,S_2}} \cdot [0,0,0,1,-1,0] \qquad \forall k \in \{1...3\} \\ N : & \Theta_{k,2}^{S_2} - \Theta_{k,3}^{S_1} + K \cdot \sum_{i=1}^{k-1} \delta_i^{\mathcal{D}_{S_1,S_2}} = \Lambda_k^{\mathcal{D}_{S_1,S_2}} \cdot [1,0,0,0,0,0] \\ 1 : & \Theta_{k,3}^{S_2} - \Theta_{k,4}^{S_1} - \delta_k^{\mathcal{D}_{S_1,S_2}} + K \cdot \sum_{i=1}^{k-1} \delta_i^{\mathcal{D}_{S_1,S_2}} = \Lambda_k^{\mathcal{D}_{S_1,S_2}} \cdot [-2,0,0,0,0,1] \end{cases} \quad \text{(12d)}$$

$$\begin{cases} i^{S_2} : & -\Theta_{k,1}^{S_2} = \Lambda_k^{\mathcal{D}_{S_2,S_1}} \cdot [-1,0,1,-1,1,0] \\ i^{S_1} : & \Theta_{k,1}^{S_1} = \Lambda_k^{\mathcal{D}_{S_2,S_1}} \cdot [0,0,0,1,-1,0] \\ j^{S_1} : & \Theta_{k,2}^{S_1} = \Lambda_k^{\mathcal{D}_{S_2,S_1}} \cdot [0,1,-1,0,0,0] \qquad \forall k \in \{1...3\} \\ N : & \Theta_{k,3}^{S_1} - \Theta_{k,2}^{S_2} + K \cdot \sum_{i=1}^{k-1} \delta_i^{\mathcal{D}_{S_2,S_1}} = \Lambda_k^{\mathcal{D}_{S_2,S_1}} \cdot [1,0,0,0,0,0] \\ 1 : & \Theta_{k,3}^{S_1} - \Theta_{k,4}^{S_2} - \delta_k^{\mathcal{D}_{S_2,S_1}} + K \cdot \sum_{i=1}^{k-1} \delta_i^{\mathcal{D}_{S_2,S_1}} = \Lambda_k^{\mathcal{D}_{S_2,S_1}} \cdot [-2,0,0,-1,1,1] \end{cases} \quad \text{(12e)}$$

To briefly summarize, Equations (12a) to (12c) set up the variables related to the dependence constraints. Equations (12d) and (12e) handle the case of $\mathcal{D}_{S_1,S_2}$ and $\mathcal{D}_{S_2,S_1}$ respectively for Equation (9). Specifically, Equations (12d) and (12e) apply Farkas' Lemma for the coefficients of $i^{S_1}, j^{S_1}, i^{S_2}, N$ and for the constant terms. We have labeled each equational constraint in Equations (12d) and (12e) with the corresponding symbolic term (i.e. one of $i^{S_1}, j^{S_1}, i^{S_2}, N$) or constant term (i.e. denoted by 1) on the left side of the equation.

3.3.5 Non-Sequential Scheduling. We first define a *sequential schedule*.

Definition 3.4 (Sequential Schedule). A schedule Θ is sequential if for each statement S and its scheduling function Θ^S we have:

$$\Theta^S \cdot [\vec{x}_1^S, \vec{p}, 1]^\top \ne \Theta^S \cdot [\vec{x}_2^S, \vec{p}, 1]^\top \quad \forall \vec{x}_1^S, \vec{x}_2^S. \ \vec{x}_1^S \ne \vec{x}_2^S. \qquad \text{(13)}$$

Intuitively, a schedule is sequential if it maps distinct iteration vectors to distinct timestamps.

A valid, dependence-satisfying schedule that satisfies Equation (8) may be *non-sequential* in that it permits multiple statement instances to execute in the same timestep. Therefore, it is possible for a schedule to demand an unbounded number of statement instances to execute at the same timestamp, which does not directly map to physical machines with finite resources [Gupta et al. 2002; Redon and Feautrier 1994].

In the context of reductions, a reduction may be scheduled to accumulate an unbounded number of values at the same timestamp [Redon and Feautrier 1994]. For example, the (one-dimensional) schedule that assigns S1$[i, j]$ to $[i]$ is not realistic, since it assigns the same timestamp to S1$[i, j]$

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

Listing 6. ST in the polyhedral IR for the example in Section 1 (Equation (4)), given the reuse vector $[1, 0]^\top$

for all j, and therefore requires accumulation of a potentially unbounded number of values. Gupta et al. [2002] demonstrate a scheduling approach that bounds the total accumulations per timestep to target physical machines.

In this paper, we do not extend Pouchet et al. [2011]'s formalization with physical scheduling constraints. While this consideration is important for practical scheduling of reductions, we only use their scheduling formalization to support our formalization of the dependent reduction scheduling problem in Section 5. Our heuristic algorithm in Section 6.2 does not require a schedule to have been computed using their scheduling formalization. Our algorithm instead relies on a sequential schedule that can be computed via any means, including a scheduling algorithm that adopts realistic scheduling constraints. In Section 6.1 we discuss sequential scheduling in more detail.

4 BACKGROUND: THE SIMPLIFYING REDUCTIONS FRAMEWORK

Previous work introduced a core transformation called the *simplification transformation* (ST) that can transform a single reduction specified in in the polyhedral representation (Figure 5) to lower its complexity [Gautam and Rajopadhye 2006]. The work also introduced a set of enabling transformations that make available opportunities to simplify reductions.

For the core transformation, we use an example from Section 1 to illustrate the transformation. For the enabling transformations, we include a brief description of each transformation in Appendix C. Finally, Gautam and Rajopadhye [2006] combine 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 the example from Section 2.1 to illustrate how the simplification transformation reduces the complexity of a reduction. We provide a more complete specification of ST in Appendix B.

Listing 6 illustrates the example of applying ST to Listing 5 (i.e., the IR form of Equation (4)) to produce the optimized version in Equations (2a), (2b) and (4b). As we mentioned before, core ST operates on a single statement only and only produces a correct result for a dependent reduction if provided with a correct reuse vector.

Original Reduction. Above the arrow, Listing 6 presents the reduction in Equation (4a) in the polyhedral IR as the statement S1 with domain $\mathcal{P} = [N] \rightarrow \{[i, j] : 0 \leq i < N \wedge 0 \leq j \leq i\}$. The right hand side expression is 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. Below the arrow, the optimized prefix sum consists of three distinct computations: seeding the initial values of the reduction, computing the root of the reduction, and computing the core of the reduction via reuse.

The statement S1Add computes BTmpAdd, which serves to hold the unique seeds of the reduction: the uniquely computed value introduced into the reduction at each iteration i. The statement computes over the full space of i and sets BTmpAdd[i] to equal A[i]. The statement S1AddOnly computes the root of the reduction, the first (by dependence order) value computed into BTmp[0] and for which the reduction use not reuse any previous computation. Hence, S1AddOnly sets BTmp[0] to equal BTmpAdd[i], the unique element for this iteration. The statement S1AddReuse computes the core of the reduction incrementally, summing the previous computation BTmp[i - 1] to the unique element for this iteration.

To identify this optimization opportunity and generate the optimized code, the Simplification Transformation identifies a *reuse vector* by which shifting the original, unoptimized polyhedron ($\mathcal{P}$) makes plain that consecutive iterations of the polyhedron share the same computation and can be incrementalized. The reuse vector then drives the transformation of the code. To elaborate ST's mechanics, we first present how ST leverages a reuse vector to transform the program before then explaining how ST identifies appropriate reuse vectors.

4.2 From Reuse Vector to Transformed Program

ST takes a reuse vector and manipulates the domain of the reduction to compute the subset of iteration instances that can be transformed to incrementally compute their results using previously computed results.

Fig. 6. Visualization of the algorithm on prefix sum example.

Consider the *reuse vector* $\vec{r} = [1,0]^\top$, which can also be represented by the polyhedral relation $\{[i,j] \rightarrow [i+1,j] : \forall i,j\}$. Conceptually, the reuse vector therefore denotes a shift of all points $[i,j]$ to $[i+1,j]$. Given the reuse vector $\vec{r} = [1,0]^\top$, ST performs the following steps:

- **Shift.** The transformation first shifts S1's polyhedron along the direction of the reuse vector, transforming $\{[i,j] : 0 \le i < N \wedge 0 \le j \le i\}$ into $\{[i,j] : 1 \le i < N+1 \wedge 0 \le j \le i-1\}$. Figure 6 illustrates the two polyhedrons, with the pre-transformation polyhedron presented as the crosshatched polyhedron with round dots and the post-shift polyhedron presented as the dot-shaded polyhedron with triangles.
- **Intersect.** The transformation next computes the intersection of the shifted polyhedron with its original polyhedron, yielding $\{[i,j] : 1 \le i < N \wedge 0 \le j \le i-1\}$ (triangles outlined in round dots in Figure 6). This polyhedron 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. We note that the intersection computation itself does not detect the equivalence of expressions across indices. Instead, ST's reuse vector selection process (Section 4.3) ensures that instances within the intersection denote equivalent expressions. i
- **Project.** Finally, the transformation projects the result onto the space of the polyhedron that represents the indices of the left hand side of the array BTmp. Concretely, the transformation applies the projection represented by the polyhedral relation $\{[i,j] \rightarrow [i] : \forall i,j\}$), yielding the polyhedron $\{[i] : 1 \le i < N\}$. This final polyhedron is exactly the domain of elements of BTmp that exhibits reuse along the reuse vector $\vec{r}$. This polyhedron corresponds to the domain of the statement S1AddReuse, computes BTmp[i] with BTmp[i] = BTmp[i − 1] + BTempAdd[i].

The polyhedron $\{[i] : 1 \le i < N\}$ does not cover the full domain of the original reduction. Specifically, it is missing BTmp[i] on the domain $\{[i] : i = 0\}$ – that is, exactly when $i = 0$. The value of BTmp[0] should be equal to A[0] and the transformation generates the statement S1AddOnly to perform this computation.
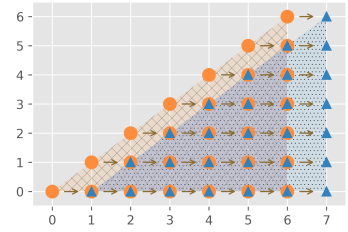
4.3 Selecting a Reuse Vector

The simplifying transformation work offers a fully automated technique to identify a reuse vector for a reduction. The techniques must specifically (1) identify if computation is *shared* among iterations of the reduction, (2) identify if the shared computation is *reusable*: if it is possible to transform the program reuse the shared computation), and 3) identify if reusing shared computation is *profitable*: the transformation reduces the complexity of a program.

Shared. ST includes an algorithm to determine the *share space*: the set of all vectors such that, for each vector, the evaluations of right hand side expression of the reduction are the same if shifted along the reuse vector.

For the prefix sum example in Section 4.2, the right hand side expression $(A[j])$) is the same along the direction $[1, 0]^\top$. However, the right hand side expression is also the same along the direction $[-1, 0]^\top$ as well as $[2, 0]^\top$. In fact, any reuse vector $[n, 0]$ for a given integer n is in the share space of this reduction. Therefore, generally, the set of reuse vectors is potentially infinite in size.

For a statement S, let $\mathcal{S}(S)$ denote the statement's share space. Gautam and Rajopadhye [2006, Section 5.2] demonstrated how to compute $\mathcal{S}(S)$ from S.

Inverse. If $\oplus$ does not have an inverse, we require that applying ST along a vector $\vec{r}$ does not introduce statements that require 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] introduce 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 requires that $\vec{r}$ must be pointing out of (instead of pointing into) the boundaries of $\mathcal{P}$ corresponding to the Boundary Constraints.

Fig. 7. Visualization of the Inverse constraint

For example, consider a modification to the prefix sum example (Listing 1), where the summation is replaced with $\max$. The ST application on this modified example along the reuse vector $[-1, 0]^\top$ is invalid, since $\max$ does not have an inverse but the transformed program requires an inverse operation in order to compute $B[i]$ from $B[i + 1]$ (i.e. Equation (3b)). Figure 7 illustrates this constraint: for the boundary of the domain orthogonal to the projection (the right side of the triangular domain), the inverse constraint requires the reuse vectors to point out of the domain (i.e. to the dotted arrow's direction).

For a statement S, let $\mathcal{I}(S)$ denote the set of vectors $\vec{r}$ that satisfy the inverse constraint. Gautam and Rajopadhye [2006, Section 5.4] demonstrated how to compute $\mathcal{I}(S)$ for a statement S.

Profitable. Applying ST with a given reuse vector is profitable only if the transformed program has lower complexity than the original program. Of note, 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, Listing 2) but with the input parameter N fixed to some constant – say $N = 4$. The complexities before and after ST will be the same – $O(1)$ – since both programs will perform a fixed number of computations.

For a reduction statement S with domain $\mathcal{P}$, let $\mathcal{L}(\mathcal{P})$ denote the set of reuse vectors that reduce the complexity of the reduction. Gautam and Rajopadhye [2006, Section 4.2] demonstrated how to compute $\mathcal{L}(\mathcal{P})$ given a domain $\mathcal{P}$.
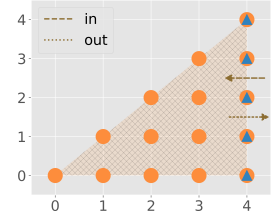
Valid. In general, the set of *valid* reuse vectors are those that satisfy the above properties (i.e., shared, reusable, and profitable). Formally, if we let the notation S.*domain* denote the domain of a statement S, then the set of valid reuse vectors is $\mathcal{R}(S) = \mathcal{S}(S) \cap \mathcal{I}(S) \cap \mathcal{L}(S.domain)$. Therefore, any $\vec{r} \in \mathcal{R}(S)$ is a valid reuse vector for applying ST.

4.4 Residual Reduction

Notice in Listing 6 that the statement S1Add still contains a reduction. Although for this example S1Add 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.[6]

Even though the space of valid reuse vectors (Section 4.3) is potentially infinite and cannot be simply enumerated, Gautam and Rajopadhye [2006] presented an dynamic programming algorithm that partitions this space into equivalence classes. The algorithm then enumerates those equivalence classes to choose reuse vectors. The algorithm always terminates and produces a resulting program with optimal complexity (i.e. minimal polynomial degree).

5 SIMPLIFYING DEPENDENT REDUCTIONS PROBLEM

In this section, we state the Simplifying Dependent Reductions (SDR) problem. In particular, we focus on the core of the Simplifying Reductions approach – the Simplification Transformation in Section 4.1 – and do not consider the Simplifying Reductions framework's additional *enabling transformations*. These transformations increase available simplification opportunities; we briefly touch on enabling transformations in Section 5.4.

5.1 Problem Statement

Ideally we would formulate the Simplifying Dependent Reduction (SDR) as Equation (14).

$$
\begin{aligned}
&\textbf{minimize} \quad \text{complexity}(prog') &&\text{(14a)}\\
&\textbf{subject to} &&\\
&\quad prog^0 = prog, \ prog' = prog^n &&\text{(14b)}\\
&\quad \forall i \in \{1...n\}: \ prog^i = \text{ST}_{S_i, \vec{r}_i}(prog^{i-1}) &&\text{(14c)}\\
&\qquad \vec{r}_i \in \mathcal{S}(S_i) \cap \mathcal{I}(S_i) \cap \mathcal{L}(S_i.domain) &&\text{(14d)}\\
&\quad \exists \text{ schedule } \Theta \text{ of } prog^n, &&\\
&\qquad \text{s.t. } \Theta \text{ satisfies dependence}(prog^n) &&\text{(14e)}\\
&\textbf{given} \quad prog, \ \text{dependence}(prog) &&\text{(14f)}\\
&\textbf{variables} \quad S_1, \ldots, S_n, \ \vec{r}_1, \ldots, \vec{r}_n &&\text{(14g)}
\end{aligned}
$$

Equation (14) states that given a program *prog*, and all pairwise dependences between those statements, dependence (*prog*), apply a sequence of n ST transformations, $\text{ST}_{S_1, \vec{r}_1}, \ldots, \text{ST}_{S_n, \vec{r}_n}$ that minimize the complexity of the resulting program, *prog'*. Here we use $\text{ST}_{S, \vec{r}}(prog)$ to denote an ST that is applied on a statement S in *prog* along the reuse vector $\vec{r}$. Each S_i is a variable that refers to a reduction in $prog^{i-1}$. Note that S_i can either be a reduction that is in $prog^0$, the initial input program, or it can be reduction that is introduced by any of the previous $i - 1$ ST applications (i.e. recursive ST in Section 4.4). Further, Equation (14d) requires each r_i to satisfy the constraints (i.e. complexity, inverse and sharing, denoted by $\mathcal{S}(.), \mathcal{I}(.), \mathcal{L}(.)$ respectively) as stated in Section 4.3.

Unfortunately, there are two issues with Equation (14): 1) it has infinite space for $\vec{r}_i$ 2) it has impractically large space for S_i.

---

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

Firstly, if we assume that n is given and bounded, [7] 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 reuse vectors cannot be simply enumerated (however it can be partitioned in to equivalence classes as proposed by Gautam and Rajopadhye [2006]).

Secondly, also assuming n is given and bounded, the program relies on a sequence of statements, $(S_1, \ldots, S_n)$, to specify on which statement in $prog^i$ to perform ST. Although, unlike the case of $\vec{r}_i$, the number of 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 $(S_1, \ldots, S_n)$ has at least $|S_1|!$ 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 i-th ST application has $n - i + 1$ remaining alternative choices of S_i (i.e. $|S_i| = n - i + 1$). Therefore the search space of S_i is also not practical to navigate with enumerative search.

5.2 Per-face ST Application

We will resolve these issues with a correct formalization in the rest of Section 5. Specifically, we show, for a program, a one-to-one correspondence between all its potential ST applications and all *faces* of its reductions' domains. This correspondance allows a construction of an Integer Bilinear Programming (IBP) formulation to SDR, which avoids the explicit enumerative searchs in the above issues of Equation (14).

Definition 5.1 (Face of polyhedral set). Let the polyhedral set $\mathcal{P} = [\vec{p}] \rightarrow \{[\vec{x}] : M \cdot [\vec{x}, \vec{p}, 1] \geq \vec{0}\}$. 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 $\forall i \exists j, B_i = M_j$.

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

We first make the following observation of ST on a 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 4.4, and this is exactly the root problem of the incorrect formulation Equation (14): this recursion appears non-terminating. We will solve this issue by stating and proving Lemma 5.3 — to this end, we first recall Lemma 5.2 from Gautam and Rajopadhye [2006] that we will use in our proof. We then state Lemma 5.3 and give a proof.

Lemma 5.2 (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 $\mathcal{P} - \mathcal{P}' = \uplus_i \mathcal{P}_i$. That is, $\mathcal{P} - \mathcal{P}'$ is a finite union of $\mathcal{P}_i$s. Further, there exists a one-to-one map from i to faces of $\mathcal{P}$ such that each $\mathcal{P}_i$ corresponds uniquely to a $(n-1)$-dimensional face of $\mathcal{P}$.*

Lemma 5.3 (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 5.2, they decompose into convex polyhedral domains, each corresponding to a unique $(n-1)$-dimensional face of the n-dimensional polyhedron $\mathcal{P}$.

---

[7]The total number of ST applications n must be a finite given the fact that the complexity of the input program is finite, and that each ST application reduces the complexity.

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 $\mathcal{P}$. Therefore, the entire recursion is 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 5.3, 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*. Specifically, if we think about the algorithm that recursively applies ST to an input program to get the final optimized program as is done in Gautam and Rajopadhye [2006], the recursion forms a computation tree, where each node of the tree corresponds to a choice of reuse vector of ST. Lemma 5.3 says that 1) this recursion is bounded, and 2) each node in this computation tree corresponds one-to-one to a face in the input program's statements' domains. Under the per-face application view, an algorithm may first choose a reuse vector for each face of each statement in the program up-front (i.e. assigns the choice of each node in the computation tree up-front), instead of recursively requesting the reuse vectors.

5.3 Integer Bilinear Program Formulation

With the per-face application view of ST in Section 5.2, we are now ready to give the correct formulation of SDR. The basic idea behind this formulation is to combine previous work on ST for a single statement [Gautam and Rajopadhye 2006, Section 5] previous work on the integer linear program formulation of polyhedral model scheduling [Pouchet et al. 2008, 2007, 2011], and the per-face application view of ST (Section 5.2). We first revisit Equation (14) and give the correct high level formulation as Equation (15).

This optimization problem minimizes the complexity of *prog'* (Equation (15a)), which is a version of *prog* transformed by a composition of STs applied to each face (Equation (15b)). The reuse vectors $\{r_1, \ldots, r_n\}$ that drive each ST must lie in the prescribed set that presents sharing ($\mathcal{S}(.)$), satisfies the inverse condition ($\mathcal{I}(.)$) and reduces complexity ($\mathcal{L}(.)$), as in Section 4.3 (Equation (15c)). Lastly, there must exist a schedule Θ that satisfies the dependences in *prog'* (Equation (15d)).

$$\textbf{minimize} \quad \text{complexity}(prog') \tag{15a}$$
$$\textbf{subject to}$$
$$prog' = (\text{ST}_{f_1, \vec{r}_1} \circ \ldots \circ \text{ST}_{f_n, \vec{r}_n})(prog) \tag{15b}$$
$$\forall i \in \{1...n\} : \ \vec{r}_i \in \mathcal{S}(f_i.stmt) \cap \mathcal{I}(f_i.stmt) \cap \mathcal{L}(f_i) \tag{15c}$$
$$\exists \text{ schedule } \Theta \text{ of } prog', \\ \text{s.t. } \Theta \text{ satisfy dependence}(prog') \tag{15d}$$
$$\textbf{given} \quad prog, \ \text{dependence}(prog) \tag{15e}$$
$$\textbf{variables} \quad \vec{r}_1, \ldots, \vec{r}_n \tag{15f}$$

This high level formulation is similar to Equation (14), 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, and 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 (15c)).

5.3.1 Transformed Program. The program *prog'* is a version of *prog* transformed by the composition of STs applied to each face (Equation (15b)) following the insight from Section 5.2.

Example. Listing 7 illustrates applications of ST to all the faces of the program for Equation (4). Specifically, the top of Listing 7 shows the program *prog* in the polyhedral IR of Equation (4), and the bottom of Listing 7 shows the program *prog'* after the sequence of ST transformations. Each introduced reduction, by the insight from Section 5.2, corresponds to a face of the reduction S1's

S1 : BTmp$[i]$ += A$[j]$: $\{[i,j] : 0 \le i < N \wedge 0 \le j \le i\}$
S1Fin : B$[i]$ = BTmp$[i]$: $\{[i] : 0 \le i < N\}$
S2 : A$[i+1]$ = $f($B$[i])$: $\{[i] : 0 \le i < N - 1\}$
  $\Downarrow$
S1 : BTmp$[i]$ += $A[j]$: $\{[i,j] : r_i = 0 \wedge r_j = 0 \wedge i < N \wedge 0 \le j \le i\}$
S1AddR1Pos : BTmpAdd$[i]$ += A$[j]$: $\{[i,j] : r_i > 0 \wedge r_j = 0 \wedge i < N \wedge i - r_i < j \wedge 0 \le j \le i\}$
S1AddR1Neg : BTmpAdd$[i]$ += A$[j]$: $\{[i,j] : r_i < 0 \wedge r_j = 0 \wedge N + r_i \le i < N \wedge 0 \le j \le i\}$
S1SubRNeg : BTmpSub$[i + r_i]$ += A$[j]$: $\{[i,j] : r_i < 0 \wedge r_j = 0 \wedge -r_i \le i < N \wedge i + r_i < j \le i\}$
S1AddOnlyR1Pos : BTmp$[i]$ = BTmpAdd$[i]$: $\{[i] : r_j = 0 \wedge 0 \le i < r_i\}$
S1AddOnlyR1Neg : BTmp$[i]$ = BTmpAdd$[i]$: $\{[i] : r_j = 0 \wedge N + r_i \le i < N\}$
S1AddReuse : BTmp$[i]$ = BTmp$[i - r_i]$ + BTmpAdd$[i]$: $\{[i] : r_i > 0 \wedge r_j = 0 \wedge r_i \le i < N\}$
S1SubRNegReuse : BTmp$[i]$ = BTmp$[i - r_i]$ − BTmpSub$[i]$: $\{[i] : r_i < 0 \wedge r_j = 0 \wedge 0 \le i < N + r_i\}$
S1Fin : B$[i]$ = BTmp$[i]$: $\{[i] : 0 \le i < N\}$
S2 : A$[i+1]$ = $f($B$[i])$: $\{[i] : 0 \le i < N - 1\}$

Listing 7. Applying ST to all the faces of the polyhedral IR program of Equation (4), given the reuse vector $\vec{r}_1 = [r_i, r_j]^\top$ consisting of integer variables r_i and r_j. The reuse vectors $\{\vec{r}_2, \ldots, \vec{r}_8\}$ and statements with empty domains are omitted.

domain. The reduction S1 has domain $\{[i,j] : r_i = 0 \wedge r_j = 0 \wedge i < N \wedge 0 \le j \le i\}$. Equation (16) gives all the faces f_i of this domain.

$$f_1 = \{[i,j] : 0 \le i < N \wedge 0 \le j \le i\} \quad \text{(16a)} \qquad f_5 = \{[i,j] : i = 0 \wedge j = 0\} \quad \text{(16e)}$$

$$f_2 = \{[i,j] : 0 \le i < N \wedge j = 0\} \quad \text{(16b)} \qquad f_6 = \{[i,j] : i = N - 1 \wedge j = 0\} \quad \text{(16f)}$$

$$f_3 = \{[i,j] : 0 \le i < N \wedge j = i\} \quad \text{(16c)} \qquad f_7 = \{[i,j] : i = N - 1 \wedge j = N - 1\} \quad \text{(16g)}$$

$$f_4 = \{[i,j] : i = N - 1 \wedge 0 \le j \le i\} \quad \text{(16d)} \qquad f_8 = \{[i,j] : 1 = 0\} \quad \text{(16h)}$$

We briefly explain the statements in $prog'$ and associate the introduced reductions to the faces.

(1) S1 in $prog'$ is the same as S1 in the original $prog$, but with the extra constraints on its domain that $r_i = 0 \wedge r_j = 0$. This corresponds to the case that $\vec{r}_1 = [0, 0]^\top$, which means that we do not apply ST at all. This reduction corresponds to the face f_1.
(2) S1AddR1Pos corresponds to f_3 (the top polyhedron in Figure 2b) for the case when $r_1 > 0$. S1AddR1Neg corresponds to f_3 (the top polyhedron in Figure 3b) for the case when $r_1 < 0$.
(3) S1SubRNeg is a residual reduction that is required for incrementally computing B via subtraction. It is only non-empty for the case when $r_1 < 0$. This reduction corresponds to the face f_2.
(4) S1AddOnlyR1Pos initializes BTmp from the result of S1AddR1Pos for the case when $r_1 > 0$. S1AddOnlyR1Neg initializes BTmp from the result of S1AddR1Neg for the case when $r_1 < 0$.
(5) S1AddReuse and S1SubRNegReuse compute B incrementally by respectively adding or subtracting from the previous value of B$[i - r_i]$.
(6) SFin and S2 remain the same as in the original $prog$.

5.3.2 *Reuse Constraints.* The reuse constraints enforce that each $\vec{r}_i$ is chosen from $\mathcal{S}(f_i.stmt) \cap \mathcal{I}(f_i.stmt) \cap \mathcal{L}(f_i)$. Gautam and Rajopadhye [2006] demonstrate how to compute each of these sets from f_i (Section 4.3). The intersection is a union of polyhedral sets. We use disjunction to constrain $\vec{r}_i$ to belong to one of the polyhedral sets. For each polyhedral set, encoding that $\vec{r}_i$ belongs to the polyhedral set is then just a simple affine inequality constraint.

Example. We illustrate reuse constraints for the example of Listing 7. First, $\mathcal{S}(\text{S1}) = [N] \rightarrow \{[i,j] : j = 0\}$ since reduction's right hand side $A[j]$'s values are the same for different i for any fixed j, for $\mathcal{S}(\text{S1})$. Second, $\mathcal{I}(\text{S1})$ is the universe set (i.e. does not impose any constraint), since the operator of S1 is the addition operator (i.e. +) and it has a well defined inverse operator,

subtraction (i.e. −). Third, we give $\mathcal{L}(f_i)$ in Equation (1) in Appendix A. Therefore, the intersection $\mathcal{S}(\text{S1}) \cap \mathcal{I}(\text{S1}) \cap \mathcal{L}(f_i)$ is equal to $[N] \to \{[i,j] : j = 0\}$ for f_1 and f_2, and is equal to $[N] \to \{[i,j] : i = 0 \land j = 0\}$ for $f_{3...8}$. In summary, we impose the constraint that $\vec{r}_1, \vec{r}_2 \in [N] \to \{[r_i, r_j] : r_j = 0\}$, and $\vec{r}_i \in [N] \to \{[r_i, r_j] : r_i = 0 \land r_j = 0\}$ for all $i \geq 3$.

5.3.3   Dependence Constraints. The dependence constraints enforce that Θ satisfies the dependences of $prog'$. Specifically, it requires that for each pair of statements S_1 and S_2 that potentially occur in $prog'$, their scheduling functions $\Theta^{S_1}, \Theta^{S_2}$ satisfy the dependence relation $\mathcal{D}_{S_1, S_2}$. At a high level, we set up the dependence constraints just the same as in Equation (8); however, with Equation (15) we allow the dependence matrix D_{S_1, S_2} to contain entries with (linear) terms with unknowns from $\vec{r}_1 ... \vec{r}_n$. An informative argument for why D_{S_1, S_2} contains these unknown entries is: if we look from the recursive ST view, each application of ST introduces a reuse vector variable $\vec{r}_i$, 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 the reuse vectors introduced by the previous ST application. The residual reductions' domains then have space extended by $\vec{r}_1 ... \vec{r}_n$.

Example. We formulate the dependence constraints (Equation (15d)) for $prog'$ in Listing 7. This step is the same as in Section 3.3.3 except that the dependence relations' constraint matrices, instead of being constant matrices, now contain r_1 and r_2 as integer variables. For brevity of presentation, we will give the example of formulating the dependence constraint (i.e. Equations (8) and (9)) for statement S1AddReuse to itself. The rest of the dependence constraints between other pairs of statements can be formulated in the same way.

For this example, we let statement S be S1AddReuse. For this statement, the dependence relation $\mathcal{D}_{S,S}$ between instances of S, is $[N] \to \{[i] \to [i'] : i' = i + r_i \land r_i > 0 \land r_j = 0 \land r_i <= i < N - r_i\}$. Equation (17) presents the constraint matrix $M^{\mathcal{D}_{S,S}}$ of this polyhedral relation. Notice that $M^{\mathcal{D}_{S,S}}$ contains r_i and r_j as variables in its last column. We can

$$M^{\mathcal{D}_{S,S}} \cdot \begin{bmatrix} i \\ i' \\ N \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & r_i-1 \\ 1 & 0 & 0 & -r_i \\ -1 & 0 & 1 & -r_i-1 \\ -1 & 1 & 0 & -r_i \\ 1 & -1 & 0 & r_i \\ 0 & 0 & 0 & r_j \\ 0 & 0 & 0 & -r_j \end{bmatrix} \cdot \begin{bmatrix} i \\ i' \\ N \\ 1 \end{bmatrix} \quad (17)$$

then formulate the dependence constraint as in Equations (8) and (9) and get Equation (18).

$$0 \leq \delta_k^{\mathcal{D}_{S,S}} \leq 1 \ \forall k \in \{1...3\} \quad (18a) \qquad \sum_{k=1}^{3} \delta_k^{\mathcal{D}_{S,S}} = 1 \quad (18b) \qquad \Lambda_k^{\mathcal{D}_{S,S}} \geq 0 \ \forall k \in \{1...3\} \quad (18c)$$

$$\begin{cases} i: & -\Theta_{k,1}^S = \Lambda_k^{\mathcal{D}_{S,S}} \cdot [0, 1, -1, -1, 1, 0, 0, 0] \\ i': & \Theta_{k,1}^S = \Lambda_k^{\mathcal{D}_{S,S}} \cdot [0, 0, 0, 1, -1, 0, 0, 0] \\ N: & K \cdot \sum_{i=1}^{k-1} \delta_i^{\mathcal{D}_{S,S}} = \Lambda_k^{\mathcal{D}_{S,S}} \cdot [0, 0, 1, 0, 0, 0, 0, 0] \\ 1: K \cdot \sum_{i=1}^{k-1} \delta_i^{\mathcal{D}_{S,S}} - \delta_k^{\mathcal{D}_{S,S}} = \Lambda_k^{\mathcal{D}_{S,S}} \cdot [r_i - 1, -r_i, -r_i - 1, -r_i, r_i, r_j, -r_j, 1] \end{cases} \quad \forall k \in \{1...3\} \quad (18d)$$

Notice that Equation (18d) refers to the integer variables r_i and r_j, multiplying them with $\Lambda_k^{\mathcal{D}_{S,S}}$ makes the problem bilinear constrained.

5.3.4   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 Ehrhardt polynomial [Ehrhardt 1967] 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 $O(M^2N)$ and $O(MN^2)$ 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 $O(M^2N^2)$, a total ordering such as $O(1) < O(M) < O(N) < O(MN) < O(M^2N) < O(MN^2) < O(M^2N^2)$ 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 [Nemhauser and Wolsey 1988], to encode an indicator variable $y \in \{0, 1\}$ for the constraint $x = 0$ so that $y = 1$ iff $x = 0$.

Example. To formulate the objective by encoding the complexity of the *prog′* in Listing 7, we first find the asymptotic complexity of each face by counting its cardinality. For this example, we have *complexity*$(f_1) = O(N^2)$, *complexity*$(f_2) = O(N)$, *complexity*$(f_3) = O(N)$ and *complexity*$(f_i) = O(1), \forall i \geq 4$. Next, we assign an indicator variable to indicate if ST is disabled along a certain face – a face with disabled ST implies that the domain that corresponds to the face is non-empty and therefore incurs cost in the objective. Take the face f_1 for example, ST is disabled along this face and incurs cost iff $\vec{r}_1 = [0, 0]^\top$. We use $1_{\vec{r}_i = \vec{0}} \in \{0, 1\}$ to denote this indicator variable. Assuming the ordering $O(1) < O(N) < O(N^2)$, we may assign the integers $0, 1, 2$ to each complexity term. The maximum number of statements in the program is $|S| = 9$, and we use a base-$|S|$ encoding for summing the final objective. The final objective then becomes

$$\sum_i |S|^{complexity(f_i)} \cdot 1_{\vec{r}_i = \vec{0}} = 9^2 \cdot 1_{\vec{r}_1 = \vec{0}} + 9^1 \cdot 1_{\vec{r}_2 = \vec{0}} + 9^1 \cdot 1_{\vec{r}_3 = \vec{0}} + 9^0 \cdot 1_{\vec{r}_4 = \vec{0}} + ...9^0 \cdot 1_{\vec{r}_8 = \vec{0}} \quad (19)$$

5.4 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 3.3.1) 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.

6 SDR HEURISTIC ALGORITHM

The problem formulation we present in Section 5.3 is a full characterization of the SDR 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. The size of the formulation (i.e. total number of constraints and number of variables) in Section 5.3 is proportional 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 of dependence resolution is based on an 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 SDR. The key idea behind our heuristic approach is that for a program that has an affine sequential schedule, [8] 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 in Section 5.3), 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 dependences of the program.

6.1 Insights

The key insights of our algorithm are that 1) choosing any valid reuse vector for a given ST results in the same final algorithmic complexity for the program (implied by Gautam and Rajopadhye [2006]) and 2) for any valid reuse vector, the direction itself or its negation adheres to a sequential schedule of the left hand side of the reduction. We demonstrate these two insights with the following lemmas.

LEMMA 6.1. *For any application of ST, the complexity decrease is always the same regardless of the actual choice of reuse vector.*

Lemma 6.1 is implied by (though not enunciated in) Gautam and Rajopadhye [2006, Theorem 3 and Section 7]. We state Lemma 6.1 explicitly for the ease of its reference and give a self-contained proof in Appendix D.

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 a scheduling function from a reduction statement to the left hand side of a reduction in a given program as follows. First the program is augmented by adding to the program a new redirect statement $A[\vec{x}] = A'[\vec{x}]$ with the same domain as the domain of A, where A' is a fresh symbol which replaces the left hand side array A of the program. Then the scheduling function of the left hand side of the reduction is simply the scheduling function of the newly introduced redirect statement of the left hand side in the schedule of the augmented program.

LEMMA 6.2. *Given a sequential 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 sequential schedule of the original program and does not introduce a dependence cycle.*

PROOF. Consider a reduction S with projection *proj* and left hand side array A. Suppose A has an affine sequential schedule Θ_A, then we have that $\Theta_A \cdot [\vec{x}, \vec{p}, 1]^\top$ is the schedule time for $A[\vec{x}]$. Let the vector $\vec{r}$ be in the same space as the domain of S, and we shift the domain of S along

---

[8]We rely on this for the soundness of our heuristic algorithm. We acknowledge that enforcing sequentiality may limit available parallelism. In practice, our algorithm works with an affine sequential schedule computed by any algorithm.

$\vec{r}$; let the projected vector of $\vec{r}$ onto the domain of A be $\vec{r_A} = proj(\vec{r})$. Consider $\vec{x}$ and $\vec{x} + \vec{r_A}$. Their scheduled timestamps are $\Theta_A \cdot [\vec{x}, \vec{p}, 1]^\top$ and $\Theta_A \cdot [\vec{x} + \vec{r_A}, \vec{p}, 1]^\top$, respectively. For all $\vec{x}$, $\Theta_A \cdot [\vec{x} + \vec{r_A}, \vec{p}, 1]^\top - \Theta_A \cdot [\vec{x}, \vec{p}, 1]^\top = \Theta_A \cdot [\vec{r_A}, \vec{0}, 0]^\top$ is a constant that does not depend on $\vec{x}$. Since Θ_A is sequential, the difference $\Theta_A \cdot [\vec{r_A}, \vec{0}, 0]^\top$ is non-zero for all non-zero $\vec{r_A}$. Therefore, $A[\vec{x}]$ is scheduled either always before $A[\vec{x} + \vec{r_A}]$ or always after $A[\vec{x} + \vec{r_A}]$. We describe these two cases in detail as follows.

Case 1. When the first non-zero entry (in accordance with the timestamp comparison in Definition 3.3) of $\Theta_A \cdot [\vec{r_A}, \vec{0}, 0]^\top$ is positive, then $A[\vec{x}]$ is always scheduled before $A[\vec{x} + \vec{r_A}]$. In this case, applying ST with the reuse vector $\vec{r}$ will not introduce any dependence cycle, since the newly introduced dependence is always consistent with the original schedule.

Case 2. When the first non-zero entry of $\Theta_A \cdot [\vec{r_A}, \vec{0}, 0]^\top$ is negative, $A[\vec{x}]$ is always scheduled after $A[\vec{x} + \vec{r_A}]$. In this case, applying ST with the reuse vector $-\vec{r}$ will not introduce any dependence cycle, for the same reason as in Case 1.

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 dependence that can be subsumed by an enforced dependence 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 8.

---

(1) Schedule the augmented program to obtain an initial sequential schedule Θ for all statements and left hand side 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.

Fig. 8. SDR heuristic algorithm

To test if $\vec{r}$ is consistent with Θ, one can compute $\Theta_A \cdot [\vec{r_A}, \vec{0}, 0]^\top$, with Θ_A and $\vec{r_A}$ defined as in Lemma 6.2, and then test if the first non-zero entry is positive.

Example. Section 2 shows an example of the above heuristic algorithm. Specifically, the middle polyhedron in Figure 2a illustrates the augmented redirect statement with the same domain as the left hand side of the reduction (i.e. $[N] \rightarrow \{[i] : 0 \leq i < N\}$). A valid two dimensional sequential schedule for the left hand side of the reduction is $\tau_B = \Theta_B \cdot [i, N, 1]^\top = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \cdot [i, N, 1]^\top = [1, i]^\top$. Then, for the reuse vector $\vec{r} = [1, 0]^\top$, we have $\Theta_B \cdot \vec{r_B} = \Theta_B \cdot [1, 0, 0]^\top = [0, 1]^\top$. Since the first non-zero entry of $[0, 1]^\top$ is positive, ST with the reuse vector $[1, 0]^\top$ will not introduce a dependence cycle (i.e. consistent with the correct optimization in Section 2).

6.3 Algorithm Analysis

Heuristic Scheduling. One advantage of the heuristic algorithm in Figure 8 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 Bondhugula et al. [2008]; Feautrier [1992a,b]; Gupta et al. [2007]. Most of these algorithms, such as the PLUTO scheduler [Bondhugula et al. 2008], provide scalable solutions to the polyhedral scheduling problem and thus the algorithm in Figure 8 does not present bottleneck due to scheduling.

Optimality Guarantee. The algorithm is optimal for the SDR problem if all the 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 of ST applications among all faces, if all reduce operators have inverses. For any ST application along a face, the complexity decrease is always the same regardless of the 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 it is possible for our algorithm to produce 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 vector to point out of certain boundaries of the domain for a reduction if there is no 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 $\vec{r}$ 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.

7 IMPLEMENTATION

We implemented our IR as in Section 3.2 and the heuristic algorithm as in Section 6 using Python. We used Integer Set Library (ISL) [Verdoolaege 2010] for manipulation of polyhedral sets and relations. We use ISL's scheduler and code generator and generate Cython code [Behnel et al. 2011], which is then compiled to C++. To obtain an affine sequential schedule for the original augmented program, we use a PLUTO-like scheduler [Bondhugula et al. 2008] built into ISL.

8 EVALUATION

The algorithm presented in this work is particularly effective on optimizing unoptimized implementations of *probabilistic inference* procedures into efficient implementations, where the inference procedures have mathematical specifications that naturally translate to our IR. The inference procedures are also *iterative*, so they contain dependent reductions that are not addressed by previous work [Gautam and Rajopadhye 2006].

Research Question. The goal of this section is to evaluate how effective the heuristic algorithm is at improving the performance on benchmarks consisting of algorithms described above.

8.1 Evaluation Metrics

We considered the following two aspects to evaluate the effectiveness of the heuristic algorithm.

8.1.1 Complexity. We first evaluate asymptotic complexity of the algorithms. This is an important metric because it determines how well these algorithms scale to large data sets, and hence how widely they can be applied.

Optimality is defined regarding programs realizable through transformations presented in this work. We evaluated our implementation in Section 7 using unoptimized implementations of probabilistic inference procedures. We present their algorithmic complexities before and after optimization, as well as the optimal complexities achievable with transformations in this work, by

solving the problem formulation in Section 5.3 exactly. In addition, we also report the complexities of manual implementations using transformations that are not in this work.

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

8.1.2 Runtime. Note that a real performance gain is not necessarily implied by a complexity improvement because the asymptotic complexity comparison ignores constant factors. The constant factors can be caused by, for example, auxillary variables overhead or memory/cache effect induced by ST, and the constant factors could change. Therefore, we further validate the runtime performance gains due to lowered compliexites, by measuring wall-clock time improvements between the optimized and unoptimized implementations.

8.2 Benchmarks

A subset of the benchmark algorithms considered are identified as "model-algorithm" pairs, where the model refers to a generative probabilistic model, and the algorithm refers to a class of algorithm to perform inference on the model. We considered 3 models and 3 algorithms. For models, we considered 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 considered Gibbs Sampling (GS) [Geman and Geman 1984], Metropolis Hasting (MH) [Hastings 1970; Metropolis et al. 1953] and Likelihood Weighting

Table 1. Benchmarks: Parameters and Meanings

Benchmark(s)	Param	Meaning
GMM-*	N	# of observations
	K	# of clusters
LDA-*	W	# of words in corpus
	K	# of topics
DMM-*	W	# of words in corpus
	K	# of topics
	D	# of documents in corpus
	A	size of alphabet of corpus
	L	max length of document
LBP-Stereo	N	# of total pixels
	K	max stereo displacement
	D	# of neighbors of a pixel
CoxPH	N	# of observations
	K	dim of a single observation

(LW) [Fung and Chang 1989]. Thus our benchmarks contain a total of 9 "model-algorithm" pairs.

Models and algorithms above have broad applications in the literature. 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. The models for GMM [Daniel Huang 2017; Walia et al. 2019], LDA [Daniel Huang 2017; Walia et al. 2019], and DMM [Walia et al. 2019] have been used as benchmarks for probabilistic inference systems. Gibbs sampling [Geman and Geman 1984], Metropolis-Hastings [Hastings 1970; Metropolis et al. 1953], and Likelihood Weighting [Fung and Chang 1989] are all widely used inference algorithms in 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).

In addition, we included two other benchmarks, namely the Loopy Belief Propagation on a 2D grid model for the application of Stereo matching (LBP-Stereo [Grauer-Gray and Cavazos 2011; Jian Sun et al. 2003]), and the Cox proportional hazards model (CoxPH) [Cox 1972; Therneau 2013]. Loopy Belief Propagation [Bishop 2006] is an iterative approximate inference algorithm, and its instantiation on the 2D grid model has applications in fields such as vision [Grauer-Gray and Cavazos 2011; Jian Sun et al. 2003] and physics [Kikuchi 1951]. CoxPH is a well known statistical model, which is typically combined with Newton's method, an iterative optimization algorithm, for

Table 2. Benchmarks: Comparison of Complexities

Benchmark	Original	Optimized (Heuristic)	SDR-Optimal	Manual
GMM-GS	$O(N^2K^2)$	$O(NK)$	$O(NK)$	$O(NK)$
GMM-MH	$O(N(N+K))$	$O(N)$	$O(N)$	$O(N)$
GMM-LW	$O(N(N+K))$	$O(N)$	$O(N)$	$O(N)$
LDA-GS	$O(W^2K^2)$	$O(WK)$	$O(WK)$	$O(WK)$
LDA-MH	$O(W^2K)$	$O(WK)$	$O(WK)$	$O(WK)$
LDA-LW	$O(W^2K)$	$O(W)$	$O(W)$	$O(W)$
DMM-GS	$O(WADK^2 + D^2K^2)$	$O((W+A)KD)$	$O((W+A)KD)$	$O(AKD)$
DMM-MH	$O(D^2K^2 + D(W+A))$	$O((K+W+A)D)$	$O((K+W+A)D)$	$O((K+L+A)D)$
DMM-LW	$O((WA+K)D)$	$O((K+W+A)D)$	$O((K+W+A)D)$	$O((K+L+A)D)$
LBP-Stereo	$O(NKD^2)$	$O(NKD)$	$O(NKD)$	$O(NKD)$
CoxPH	$O(K^2N^2)$	$O(K^2N)$	$O(K^2N)$	$O(K^2N)$

inference on the model's parameters. CoxPH is commonly found in medical applications [Collett 1993; White et al. 2016], and mechanical systems [Susto et al. 2015].

All of the benchmarks have a common feature that they are iterative methods specialized to a generative probabilistic model. The parameters for these benchmarks are listed in Table 1.

8.3 Results

As described in Section 8.1, we first evaluated our method by analyzing the asymptotic performance improvements and reported the results in Section 8.3.1. Then, we validated the runtime improvements of the benchmarks in Section 8.3.2.

8.3.1 Complexity Results. Table 2 summarizes the results on comparison of complexities, expressed in terms of the corresponding parameters of each benchmark.

The column "Original" gives the complexity of the original program for the benchmarks. The column "Optimized (Heuristic)" (later abbreviated as "Optimized") gives the complexity of the transformed program using the heuristic implementation in Section 7. The column "SDR-Optimal" gives the complexity of the transformed program by potentially solving the problem formulation in Section 5.3 exactly [9]; and this is the optimal complexity one can achieve using techniques presented in this work. The column "Manual" gives the complexity of a potential optimized manual implementation written by a developer; this means that the complexity reduction potentially comes from transformations not present in this work.

Comparing the "Original" and "Optimized" columns, our approach reduces the complexities for all benchmarks. Comparing the "Optimized" and "SDR-Optimal" columns, our heuristic approach produces complexities same as solving SDR optimally for all benchmarks. Comparing the "Optimized" and "Manual" columns, our approach produces 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 did not consider in this work, which is a direction of future research.

8.3.2 Runtime Validation. So far we have evaluated our heuristic algorithm using algorithmic complexity as the primary factor, which ignores constant factors caused by, for example, auxiliary variables overhead or memory/cache effect induced by ST. In this section, we validate our hypothesis that asymptotic complexity improvements dominates potential constant factors improvements for

---

[9]For all of our benchmarks, it turns out that our heuristic algorithm is able to assign a non-zero reuse vector to each non-degenerate face (i.e. one whose cardinality is not $O(1)$). This implies that our heuristic algorithm's solution is already the optimal solution to SDR, since it does not have further complexity reduction opportunities. Therefore, through this argument we obtain the SDR-optimal complexities without having to actually construct and solve the bilinear program.

Table 3. Benchmarks: Parameter Sizes

Benchmark(s)	Parameter	Size
GMM-*	N	10000
	K	10
LDA-*	W	$466,000$
	K	50
DMM-*	W	$570,000$
	K	4
	D	278
	A	129
	L	3202
LBP-Stereo	N	$110,592$
	K	16
	D	4
CoxPH	N	424
	K	11

Table 4. Runtime evaluations

Benchmark	Original	Optimized (Heuristic)	Speedup
GMM-GS	$29.2\,\text{ms}$	$4.1\,\text{ms}$	$7.1\,\textbf{x}$
GMM-MH	$0.94\,\text{ms}$	$0.72\,\text{ms}$	$1.3\,\textbf{x}$
GMM-LW	$2.5\,\text{s}$	$1.7\,\text{s}$	$1.5\,\textbf{x}$
LDA-GS	**timeout**	$7.1\,\text{ms}$	$> 6.1 \times 10^6\,\textbf{x}$
LDA-MH	**timeout**	$0.69\,\text{s}$	$> 63 \times 10^3\,\textbf{x}$
LDA-LW	**timeout**	$41.1\,\text{s}$	$> 1.1 \times 10^3\,\textbf{x}$
DMM-GS	$2.2\,\text{s}$	$0.54\,\text{s}$	$4.1\,\textbf{x}$
DMM-MH	$529\,\text{ms}$	$14.3\,\text{ms}$	$37\,\textbf{x}$
DMM-LW	$71\,\text{s}$	$1.23\,\text{s}$	$58\,\textbf{x}$
LBP-Stereo	$16.9\,\text{s}$	$15.1\,\text{s}$	$1.1\,\textbf{x}$
CoxPH	$69.1\,\text{ms}$	$8.4\,\text{ms}$	$8.2\,\textbf{x}$

the parameters of these benchmarks by timing our benchmarks and comparing the runtimes of the naive implementations with the optimized implementations. We generate C++ code for the IR programs corresponding to the Original and Optimized (Heuristic) columns in Table 2 . We ran these implementations and report timeouts for benchmarks that ran for 12+ hours.

Parameter Sizes. We collected the typical instantiated values for global parameters from the corresponding literature. Specifically, for GMM we use Daniel Huang [2017], for LDA we use Newman [2008], for DMM we use Turnbaugh et al. [2008], for LBP-Stereo we use Jian Sun et al. [2003] and for CoxPH we use Therneau [2013, Appendix D2]. Based on these prior works, we collected the following parameters for each model in Table 3.

Results. In Table 4, all benchmarks have non-trivial speedups. In particular, for LDA benchmarks all the unoptimized implementations timeout. This is because, in Table 2, the complexities of LDA benchmarks all improve by a factor of $W \times K$. With our instantiated parameter values this factor is $466,000 \times 50 = 2.3 \times 10^7$ – the largest factor across all benchmarks. This large a factor unsurprisingly leads to the timeout of the unoptimzied implementations. On the other hand, in Table 2 LBP-Stereo's complexity only improves by a factor of D, the number of neighbors of a pixel, which is set to 4 (i.e. number immediate neighbors of a pixel) in our parameter setting. Nonetheless, we observe a speedup of 1.1x for this benchmark. We also note that this speedup scales with the specification of the LBP model – setting to 8 neighboring pixels (i.e. nearby 8 pixels for a pixel at the center of a 3 by 3 square) would lead to speed up of of 1.4x. In sum, the observed speedups validates that for these benchmarks and our technique, complexity dominates constant-factor concerns.

9 RELATED WORK

Reductions Doerfert et al. [2015]; Ginsbach and O'Boyle [2017] introduced techniques 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. Doerfert et al. [2015]; Ginsbach and O'Boyle [2017]; Rauchwerger and Padua [1999]; Reddy et al. [2016] optimized reductions in the polyhedral model for considerations such as privatization and parallelization. They do not optimize for complexities; however, they can be used as optimizing backends for generating efficient code for reductions after applying our method. Iooss et al. [2014] proposed a semi-algorithm that decides equivalence between programs with reductions; it can be used to check correctness of a program transformation.

Simplifying Reductions Liu et al. [2005] introduced a loop based transformation algorithm for reducing complexities on loop programs. The algorithm uses the Omega calculator [Padua 2011] for analysis on a *contributing set*. The contributing set based anaylsis is general in the sense that it works for non-polyhedral sets as well. Liu et al. [2005] used only the direction of loop increment to decrease the complexity. Gautam and Rajopadhye [2006] generalized the method in Liu et al. [2005] and introduced the simplifying reductions framework; the framework is more general in that it uses reuse vectors not limited to just the direction of loop increments.

Scheduling We review related works on polyhedral scheduling in Appendix E.1.

Program Complexity Rubiano [2017] and Alias et al. [2010] studied methods to infer the complexity of programs. These works can reduce the effort for manually analyzing the hand optimized benchmarks, which is particularly helpful for enlarging the benchmark suite.

Incrementalization in Probabilistic Programming The problem of incrementalization occurs in probabilistic programming system (PPS), and is known as incrementalized inference. Existing work [Kiselyov 2016; Nori et al. 2015; Ritchie et al. 2016; Wu et al. 2016; Yang et al. 2014; Zhang and Xue 2019] attempted to address the problem of incrementalized inference in 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 dependence 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 incrementalized inference problem; however, our technique is a static compilation techinique which do not suffer from runtime overhead.

Many existing and ongoing work [Atkinson et al. 2018; Bingham et al. 2019; Daniel Huang 2017; Goodman et al. 2008; Goodman and Stuhlmüller 2014; Mansingkha et al. 2018; Narayanan et al. 2016; Plummer 2015] allow the user to code in high level DSLs. Though the details on these systems are out of the scope of the paper, our method can be potentially integrated into these systems for generating code with efficient complexity, which we discuss more in Appendix E.2.

10 CONCLUSION

In this work, we introduce the simplifying dependent reductions problem and provide a heuristic algorithm that — given an affine sequential schedule for the program — is optimal for reduction operators that have inverses. These reductions have otherwise only appeared as domain- or 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 languages and systems for this increasingly important class of computations could see significant performance improvements by incorporating our techniques.

ACKNOWLEDGMENTS

REFERENCES

Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Static Analysis Symposium.* https://hal.inria.fr/inria-00523298

Eric Atkinson, Cambridge Yang, and Michael Carbin. 2018. Verifying Handcoded Probabilistic Inference Procedures. In *arXiv e-prints.*

Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The Best of Both Worlds. *Computing in Science and Engg.* 13, 2 (March 2011), 31–39.

Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The Polyhedral Model is More Widely Applicable Than You Think. In *European Conference on Theory and Practice of Software, International Conference on Compiler Construction.*

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* 20, 28 (2019).

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

David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *Journal of Machine Learning Research* 3 (Jan. 2003), 993–1022.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Conference on Programming Language Design and Implementation.*

Jean-François Collard, Denis Barthou, and Paul Feautrier. 1995. Fuzzy Array Dataflow Analysis. In *Symposium on Principles and Practice of Parallel Programming.*

D Collett. 1993. *Modelling Survival Data in Medical Research.* Chapman & Hall, New York.

D. R. Cox. 1972. Regression Models and Life-Tables. *Journal of the Royal Statistical Society: Series B (Methodological)* 34, 2 (1972), 187–202.

Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. 2019. Gen: a General-purpose Probabilistic Programming System with Programmable Inference. In *Conference on Programming Language Design and Implementation.*

Greg Morisett Daniel Huang, Jean-Baptiste Tristan. 2017. Compiling Markov Chain Monte Carlo Algorithms for Probabilistic Modeling. In *Conference on Programming Language Design and Implementation.*

Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. 2015. Polly's Polyhedral Scheduling in the Presence of Reductions. In *International Workshop on Polyhedral Compilation Techniques.*

E. Ehrhardt. 1967. Sur un problème de Géométrie Diophantienne Linéaire. II. *Journal für die Reine und Angewandte Mathematik* 1967 (1967), 25–49. Issue 227.

P. Feautrier. 1988. Array Expansion. In *International Conference on Supercomputing.*

Paul Feautrier. 1992a. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International Journal of Parallel Programming* 21, 5 (Oct. 1992), 313–347.

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

Robert M. Fung and Kuo-Chu Chang. 1989. Weighing and Integrating Evidence for Stochastic Simulation on Bayesian Networks. In *Conference on Uncertainty in Artificial Intelligence.*

Gautam and S. Rajopadhye. 2006. Simplifying Reductions. In *Symposium on Principles of Programming Languages.*

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

Stuart Geman and Donald Geman. 1984. Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of images. *Transactions on Pattern Analysis and Machine Intelligence* 6 (Nov. 1984), 721–741. Issue 6.

Philip Ginsbach and Michael F. P. O'Boyle. 2017. Discovery and Exploitation of General Reductions: A Constraint Based Approach. In *International Symposium on Code Generation and Optimization.*

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A language for generative models. In *Conference on Uncertainty in Artificial Intelligence.*

Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2020-10-30.

Scott Grauer-Gray and John Cavazos. 2011. Optimizing and Auto-tuning Belief Propagation on the GPU. In *Workshop on Languages and Compilers for Parallel Computing.*

T. Griffiths and M. Steyvers. 2004. Finding Scientific Topics. *Proceedings of the National Academy of Sciences* 101, suppl. 1 (April 2004), 5228–5235.

Gautam Gupta, Kim Daegon, and Sanjay Rajopadhye. 2007. Scheduling in the Z-Polyhedral Model. *International Parallel and Distributed Processing Symposium.*

Gautam Gupta, Sanjay Rajopadhye, and Patrice Quinton. 2002. Scheduling Reductions on Realistic Machines. In *Symposium on Parallel Algorithms and Architectures*.

W. K. Hastings. 1970. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika* 57, 1 (April 1970), 97–109.

Ian Holmes, Keith Harris, and Christopher Quince. 2012. Dirichlet Multinomial Mixtures: Generative Models for Microbial Metagenomics. *PLOS ONE* 7 (2 2012). Issue 2.

Guillaume Iooss, Christophe Alias, and Sanjay Rajopadhye. 2014. On Program Equivalence with Reductions. In *International Static Analysis Symposium*.

Jian Sun, Nan-Ning Zheng, and Heung-Yeung Shum. 2003. Stereo matching using belief propagation. *Transactions on Pattern Analysis and Machine Intelligence* 25, 7 (July 2003), 787–800.

Ryoichi Kikuchi. 1951. A Theory of Cooperative Phenomena. *Physical Review* 81 (March 1951), 988–1003. Issue 6.

Oleg Kiselyov. 2016. Probabilistic Programming Language and its Incremental Evaluation. In *Asian Symposium on Programming Languages and Systems*.

Jun S. Liu. 1994. The Collapsed Gibbs Sampler in Bayesian Computations with Applications to a Gene Regulation Problem. *J. Amer. Statist. Assoc.* 89, 427 (Sept. 1994), 958–966.

Yanhong A. Liu, Scott D. Stoller, Ning Li, and Tom Rothamel. 2005. Optimizing Aggregate Array Computations in Loops. *ACM Transactions on Programming Languages and Systems* 27, 1 (Jan. 2005), 91–125.

Vikash Mansingkha, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic Programming with Programmable Inference. In *Conference on Programming Language Design and Implementation*.

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. 1953. Equation of State Calculations by Fast Computing Machines. *Journal of Chemical Physics* 21, 6 (1953), 1087–1092.

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

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *International Symposium on Functional and Logic Programming*.

George L. Nemhauser and Laurence A. Wolsey. 1988. Integer and Combinatorial Optimization.

David Newman. 2008. Bag of Words Dataset. In *UCI Machine Learning Respository*.

Aditya V. Nori, Sherjil Ozair, Sriram K. Rajamani, and Deepak Vijaykeerthy. 2015. Efficient Synthesis of Probabilistic Programs. In *Conference on Programming Language Design and Implementation*.

David Padua (Ed.). 2011. *Omega Calculator.* Springer US, Boston, MA, 1355–1355. https://doi.org/10.1007/978-0-387-09766-4_2303

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

Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. 2008. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *Conference on Programming Language Design and Implementation*.

Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. 2007. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *International Symposium on Code Generation and Optimization*.

Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. 2011. Loop Transformations: Convexity, Pruning and Optimization. In *Symposium on Principles of Programming Languages*.

L. Rauchwerger and D. A. Padua. 1999. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. *Transactions on Parallel and Distributed Systems* 10, 2 (Feb. 1999), 160–180.

C. Reddy, M. Kruse, and A. Cohen. 2016. Reduction drawing: Language constructs and polyhedral compilation for reductions on GPUs. In *International Conference on Parallel Architecture and Compilation Techniques*.

Xavier Redon and Paul Feautrier. 1994. Scheduling Reductions. In *International Conference on Supercomputing*.

Philip Resnik and Eric Hardisty. 2010. *Gibbs Sampling for the Uninitiated.* Technical Report. University of Maryland Institute of Advanced Computer Studies.

Daniel Ritchie, Andreas Stuhlmüller, and Noah Goodman. 2016. C3: Lightweight Incrementalized MCMC for Probabilistic Programs using Continuations and Callsite Caching. In *International Conference on Artificial Intelligence and Statistics*.

Thomas Rubiano. 2017. *Implicit Computational Complexity and Compilers.* Ph.D. Dissertation.

Yannick Saouter and Patrice Quinton. 1993. Computability of Recurrence Equations. *Theoretical Computer Science* 116, 2 (Aug. 1993), 317–337.

Alexander Schrijver. 1986. *Theory of Linear and Integer Programming.* John Wiley & Sons, Inc., New York, NY, USA.

G. A. Susto, A. Schirru, S. Pampuri, S. McLoone, and A. Beghi. 2015. Machine Learning for Predictive Maintenance: A Multiple Classifier Approach. *Transactions on Industrial Informatics* 11, 3 (June 2015), 812–820.

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

Dustin Tran, Matthew D Hoffman, Rif A Saurous, Eugene Brevdo, Kevin Murphy, and David M Blei. 2017. Deep probabilistic programming. In *International Conference on Learning Representations*.

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. 2008. A core gut microbiome in obese and lean twins. *Nature* 457 (12 2008), 480–4.

Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *International Congress on Mathematical Software*.

Sven Verdoolaege. 2016. Presburger Formulas and Polyhedral Compilation. https://doi.org/10.13140/RG.2.1.1174.6323

Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. 2013. On Demand Parametric Array Dataflow Analysis. In *International Workshop on Polyhedral Compilation Techniques*.

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

Rajan Walia, Praveen Narayanan, Jacques Carette, Sam Tobin-Hochstadt, and Chung-chieh Shan. 2019. From High-level Inference Algorithms to Efficient Code. In *International Conference on Functional Programming*.

Nicola White, Fiona Reid, Adam Harris, Priscilla Harries, and Patrick Stone. 2016. A Systematic Review of Predictions of Survival in Palliative Care: How Accurate Are Clinicians and Who Are the Experts? *PLOS ONE* 11, 8 (08 2016), 1–20.

Yi Wu, Lei Li, Stuart Russell, and Rastislav Bodik. 2016. In *International Joint Conferences on Artificial Intelligence*.

Lingfeng Yang, Patrick Hanrahan, and Noah Goodman. 2014. Generating Efficient MCMC Kernels from Probabilistic Programs. In *International Conference on Artificial Intelligence and Statistics*.

Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2013. AlphaZ: A System for Design Space Exploration in the Polyhedral Model. In *Workshop on Languages and Compilers for Parallel Computing*.

Jieyuan Zhang and Jingling Xue. 2019. Incremental Precision-Preserving Symbolic Inference for Probabilistic Programs. In *Conference on Programming Language Design and Implementation*.