Maximal Simplification of Polyhedral Reductions

LOUIS NARMOUR, Colorado State University, USA and University of Rennes, Inria, CNRS, IRISA, France
TOMOFUMI YUKI, Unaffiliated, Japan
SANJAY RAJOPADHYE, Colorado State University, USA

Reductions combine collections of input values with an associative and often commutative operator to produce collections of results. When the *same* input value contributes to *multiple* outputs, there is an opportunity to *reuse* partial results, enabling *reduction simplification*. Simplification often produces a program with lower asymptotic complexity. Typical compiler optimizations yield, at best, a constant fold speedup, but a complexity improvement from, say, cubic to quadratic complexity yields unbounded speedup for sufficiently large problems. It is well known that reductions in polyhedral programs may be simplified *automatically*, but previous methods cannot exploit all available reuse. This paper resolves this long-standing open problem, thereby attaining minimal asymptotic complexity in the simplified program. We propose extensions to prior work on simplification to support any independent commutative reduction. At the heart of our approach is piece-wise simplification, the notion that we can split an arbitrary reduction into pieces and then independently simplify each piece. However, the difficulty of using such piece-wise transformations is that they typically involve an infinite number of choices. We give constructive proofs to deal with this and select a finite number of pieces for simplification.

CCS Concepts: • **Theory of computation** → **Dynamic programming**; **Algebraic semantics**; **Program analysis**; **Abstraction**.

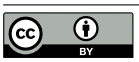
Additional Key Words and Phrases: polyhedral compilation, algorithmic complexity, program transformation

ACM Reference Format:
Louis Narmour, Tomofumi Yuki, and Sanjay Rajopadhye. 2025. Maximal Simplification of Polyhedral Reductions. *Proc. ACM Program. Lang.* 9, POPL, Article 3 (January 2025), 28 pages. https://doi.org/10.1145/3704839

1 Introduction

Computing technology has become increasingly powerful and complex over the years, offering more capability with each new generation of processors. For example, the latest generation of Intel Xeon processors supports configurations of up to 50 cores on a single die with 100 MB of last-level cache. However, using all the available processing power in the presence of complex data dependencies is not always easy. Relying on traditional compilers to produce high-performance code for a given input program is often insufficient. One reason is that general-purpose language compilers must be very conservative in the types of optimizations that can be employed. Consequently, the onus is on the application developer to write the program in such a way that the compiler can successfully detect optimization opportunities. This is challenging because it is often easier to think about problems from a higher level of abstraction, whereas writing efficient code requires lower-level reasoning. Over the years, this has led to the development of a wide variety of Domain Specific Languages (DSLs) and highly specialized frameworks.

Authors' Contact Information: Louis Narmour, Colorado State University, Fort Collins, USA and University of Rennes, Inria, CNRS, IRISA, Rennes, France, louis.narmour@colostate.edu; Tomofumi Yuki, Unaffiliated, Yokohama, Japan, tomofumi.yuki@gmail.com; Sanjay Rajopadhye, Colorado State University, Fort Collins, USA, sanjay.rajopadhye@colostate.edu.



ACM 2475-1421/2025/1-ART3
https://doi.org/10.1145/3704839

The polyhedral model [3, 9, 10, 12, 17, 20, 23, 37–39, 41, 44, 48–50] is one such framework and is a mathematical formalism for specifying, analyzing, and transforming compute- and data-intensive programs. Such programs occur in a wide variety of application domains, like dense linear algebra, signal and image processing, convolutional neural nets, deep learning, back-propagation training, and dynamic programming, to name just a few. In this paper, we study the optimization of programs that can be specified by reductions within the polyhedral model. Reductions are ubiquitous in computing and typically involve applying an associative, often commutative, operator to collections of inputs to produce one or more results. Such operations are interesting because they often require special handling to obtain good performance. The OpenMP C/C++ multithreading API [34] even has directives tailored specifically for parallelizing reductions. However, in some cases, the input program specification may involve *reuse*—the same value contributing to multiple results. When properly exploited, it is possible to improve the asymptotic complexity of such programs. A complexity improvement from, say, $O(N^3)$ to $O(N^2)$ yields unbounded speedup since, asymptotically, N can be arbitrarily large.

Gautam and Rajopadhye [14] previously showed how to reduce, by such polynomial degrees, the asymptotic complexity of commutative polyhedral reductions. They developed a program transformation called *simplification* and outlined a recursive algorithm to automatically simplify polyhedral reductions. Their work (henceforth referred to as GR06) is optimal within its scope of applicability. The simplification algorithm may fail when the reduction operator does not admit an inverse. Such operators are very common in many dynamic programming algorithms. Indeed, polyadic dynamic programming problems [24] are nothing but reductions and are widely used in many bio-informatics algorithms [5–7, 11, 16, 26–29, 33, 51, 54, 55].

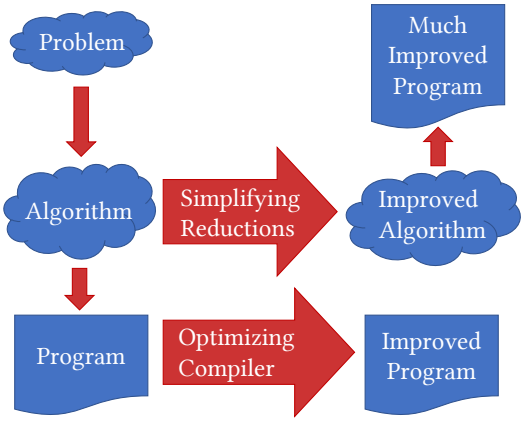


Fig. 1. Simplification improves the asymptotic complexity of the *algorithm*.

This paper proposes a method to extend the simplification algorithm to handle all scenarios, achieving maximal simplification. At the heart of our approach is *piece-wise simplification*, the notion that we can split the problem into a number of pieces and then independently simplify each piece. Such *piece-wise affine transformations*, also known as index-set splitting, have a long history in other polyhedral analyses [4, 15, 40, 42, 46]. The difficulty lies in the fact that, in general, there are infinitely many ways to split a polyhedron. We give constructive proofs showing how to select a finite number of pieces for simplification. In doing so, we make the following contributions,

(1) We propose extensions to the GR06 simplification algorithm to support any arbitrary independent reduction, particularly when the operator does not admit an inverse.
(2) We provide an implementation of our approach as an accompanying software artifact [32].

The remainder of this paper is organized as follows. Section 2 provides several motivating examples of progressively increasing difficulty. In Section 3, we review background material on the polyhedral model and prior work to explain the limitations of simplification. In Sections 4 and 5, we formulate our approach and justify it in Sections 6 and 7. We discuss aspects of our implementation in Section 8. Finally, we review related work in Section 9 and conclude in Section 10.

2 Motivating Examples

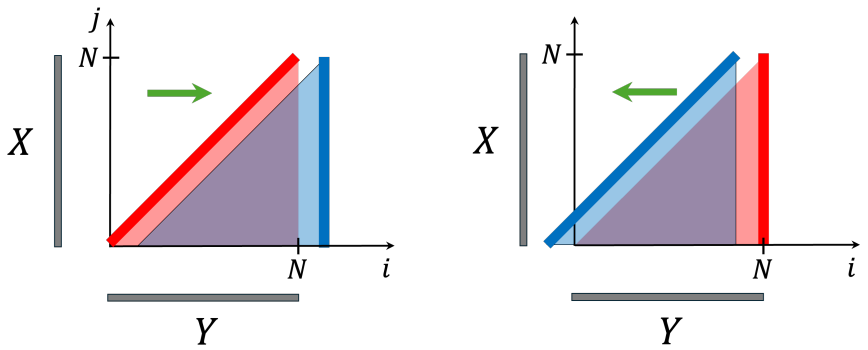
We now show several simple examples to illustrate the intuition of simplification and its limitations.

2.1 Prefix Sum

Consider the prefix sum, which computes an N-element output array from an N-element input array where the i'th element of the output is the sum of the first i elements of the input:

$$Y_i = \sum_{j=0}^{j \leq i} X_j \tag{1}$$

As written, Equation 1 has an asymptotic complexity of $O(N^2)$ because each of the N elements in the output involves a summation over $O(N)$ input values. However, two adjacent elements in the output involve summing many of the *same elements* (i.e., there is *reuse*), which means there is an opportunity to reuse partial results.



(a) Equation 2 expressing Y_i in terms of Y_{i-1}. (b) Equation 3 expressing Y_i in terms of Y_{i+1}.

Fig. 2. Reduction from quadratic to linear asymptotic complexity. The input variable X is oriented along the vertical axis, and the output Y is oriented along the horizontal.

For example, $Y_k = X_0 + X_1 + ... + X_k$ and $Y_{k+1} = X_0 + X_1 + ... + X_k + X_{k+1}$. Once Y_k is computed, there is no need to re-sum the first k elements of X to compute Y_{k+1}. Instead, Y_{k+1} can be expressed in terms of Y_k and only the *new* values of X:

$$Y_{k+1} = (X_0 + X_1 + X_2 + ... + X_k) + X_{k+1}$$
$$Y_{k+1} = Y_k + X_{k+1}$$

This observation allows Equation 1 to be rewritten as either:

$$Y_i = \begin{cases} i = 0 & : \quad X_0 \\ i > 0 & : \quad Y_{i-1} + X_i \end{cases} \tag{2}$$

where Y_i is computed from Y_{i-1}, or equally validly, as:

$$Y_i = \begin{cases} i = N & : \quad \sum_{j=0}^{N} X_j \\ i < N & : \quad Y_{i+1} - X_{i+1} \end{cases} \tag{3}$$

where Y_i is computed from Y_{i+1} instead. The terms are colored to match the corresponding edges in Figure 2. Consequently, both have a smaller, better asymptotic complexity of $O(N)$. The latter requires an initial summation of all N elements of X to produce Y_N, but the overall complexity is still linear because the remaining $N-1$ elements are each computed in constant time.

The arrows in Figure 2 reflect the choice of expressing Y_i as either Y_{i-1} or Y_{i+1}, which has the effect of moving the computation from the 2D domain (triangle) to some of the 1D edges. The correspondance between Figure 2 and Equations 2 and 3 will be made more explicit in Section 3. The goal of GR06's simplification is to explore all such possible rewrites and find the ones with the optimal asymptotic complexity in the presence of reuse. However, as illustrated by the next example, some rewrites may not be possible.

2.2 Prefix Max

Consider the prefix max, which is identical to the previous example, except it uses the max operator instead of addition:

$$Y_i = \max_{j=0}^{j \le i} X_j \tag{4}$$

Everything else is the same, the value produced at Y_i can be used to compute the next value at Y_{i+1}, allowing it to be rewritten with $O(N)$ complexity as:

$$Y_i = \begin{cases} i = 0 & : \quad X_0 \\ i > 0 & : \quad \max(Y_{i-1}, X_i) \end{cases} \tag{5}$$

exactly like Equation 2. However, the key difference here is that only one of the rewrites is possible. There is no way to express an equation analogous to Equation 3 because the max operator does not admit an inverse. This is not an issue here since at least one rewrite does not involve the inverse operation, and thus GR06 can find it. However, in general, a rewrite may not exist that does not involve the inverse operation, and consequently, GR06 may fail, as illustrated by the next example.

2.3 Sliding and Increasing Max Filter

Consider the following equation, which computes an N-element output array from an N-element input array where the i'th element of the output is the max over the sliding, and increasing, window from i to $2i$ on the input:

$$Y_i = \max_{j=i}^{j \le 2i} X_j \tag{6}$$

As written, this has an asymptotic complexity of $O(N^2)$. Like the previous examples, there is also reuse. Across two adjacent answers in the output, $O(N)$ of the same inputs are read:

$$\begin{aligned} Y_k &= \max(X_k, X_{k+1}, X_{k+2}, ..., X_{2k}) \\ Y_{k+1} &= \max(\quad X_{k+1}, X_{k+2}, ..., X_{2k}, X_{2k+1}, X_{2k+2}) \end{aligned}$$

But, neither can Y_{k+1} be expressed in terms of Y_k nor can Y_k be expressed in terms of Y_{k+1} because this would require *removing* the contributions of either X_k or X_{2k+1} and X_{2k+2} which is not possible because the max operator has no inverse.

However, it is still possible to rewrite Equation 6 with an asymptotic complexity of $O(N)$. The solution is to split this reduction into three pieces by the hyperplanes $2i = N$ and $j = N$. Since the reduction is commutative, the order of accumulation does not matter, and as we discuss in

Section 4.5, Equation 6 may be rewritten as,

$$Y_i = \begin{cases} 2i \geq N & : \quad \max\left(\left(\max_{j=i}^{j<N} X_j\right), \left(\max_{j=N}^{j\leq 2i} X_j\right)\right) \\ 2i < N & : \quad \max_{j=i}^{j\leq 2i} X_j \end{cases} \tag{7}$$

In this form, we can see three pieces. The two reductions in the top branch $2i \geq N$ are both instances of a standard suffix max and prefix max in the previous section, 2.2. The reduction in the second branch is the same reduction as Equation 6 but just over a smaller domain. This leads to a recursive simplification strategy, which can be further and similarly split into the same number of pieces. We formulate this more precisely in Section 7, and the rest of this work is dedicated to generalizing this idea to arbitrary input reductions.

3 Background

In this section, we summarize the simplification transformation of GR06 with an example.

3.1 Terminology and Notation

Generally, simplification operates on computations of the following form:

$$Y_{f_p(z)} \quad = \quad \bigoplus_{z \in \mathcal{D}} X_{f_d(z)} \tag{8}$$

where Y is an output variable, and f_p and f_d are affine functions. The *body* is an arbitrary expression involving other variables in the program, provided they do not depend recursively, even transitively, on any instance of Y. Without loss of generality (our tools handle arbitrary expressions), we abstract it as an input variable X. We use the terminology of GR06, summarized below:

- *Polyhedron*: A set of integer points defined by a finite list of inequality and equality constraints.
- *Reduction body* ($\mathcal{D}$): A d-dimensional polyhedron representing the values of the program variable indices involved in the reduction's accumulation.
- *Facet (or face)*: A k-dimensional face of the reduction body described uniquely by a subset of its inequality constraints treated as equalities.
- *Face lattice*: The hierarchical arrangement of faces of the reduction body, see Section 3.2.
- *Write function* (f_p): A rank-deficient affine map from $\mathbb{Z}^d \rightarrow \mathbb{Z}^{d-a}$ defining to which element of the output each point in the reduction body accumulates.[1]
- *Accumulation space* ($\mathcal{A}$): The a-dimensional space characterized by the null space of the write function.
- *Read function* (f_d): A (potentially rank-deficient) affine map from $\mathbb{Z}^d \rightarrow \mathbb{Z}^{d-r}$ characterizing from which element of the input each point in the reduction body reads.
- *Reuse space* ($\mathcal{R}$): The r-dimensional space characterized by the null space of the read function.
- *Reuse vector* ($\vec{\rho}$): Any vector in the reuse space.

Such programs consist of arbitrarily nested loops with affine control and a single statement in the body, plus initialization statements as appropriate. Each statement accumulates values into some element of an output array using the reduction operator $\oplus$. The right-hand sides of the statements are arbitrary expressions evaluated in constant time, reading other array variables via affine access functions. Our reduction operators are associative and commutative, so the execution order of the loops is irrelevant. Every instance of the loop body (for every legal value of the surrounding indices) can be treated as a new dummy variable. Simplification is possible when the reuse space is non-empty (i.e., the read function, f_d, is rank-deficient).

[1] Rank-deficiency implies that $a > 0$.

3.2 Face Lattice

The face lattice [25] is an important data structure for simplification. The face lattice of a polyhedron $\mathcal{D}$ is a graph whose nodes are the *facets* of $\mathcal{D}$. Each face in the lattice is the intersection of $\mathcal{D}$ with one or more *equalities* of the form $\alpha z + \gamma = 0$ for $z \in \mathcal{D}$ obtained by *saturating* one or more of the inequality constraints in $\mathcal{D}$. We refer to each k-dimensional face as a *(k)-face*. More than one constraint may be saturated to yield recursively, facets of facets, or *faces*. In the lattice, faces are arranged level by level, and each face saturates exactly one constraint in addition to those saturated by its immediate ancestors.

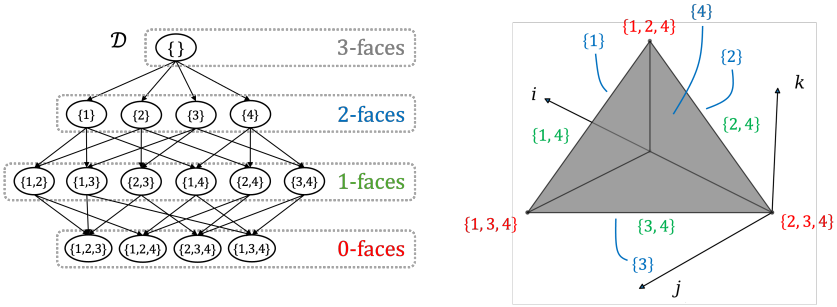


Fig. 3. Face lattice of $\mathcal{D}$ in Equation 9. The numbers inside the labels denote the constraints that, when saturated, describe the face. For example, "{4}" denotes saturating constraint $c_4 : k \leq i - j$, which represents the top oblique 2-face. The back three edges (1-faces) and vertex (0-face) are not labeled.

3.3 Working Example

Consider the reduction which produces an $O(N)$-element array specified by the following equation:

$$Y_i = \sum_{(i,j,k) \in \mathcal{D}} X_k \tag{9}$$

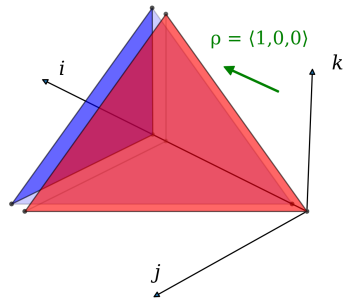
over the 3-dimensional domain:

$$\mathcal{D} = \{[i, j, k] \mid (i <= N) \wedge (0 \leq j) \wedge (0 \leq k) \wedge (k \leq i - j)\} \tag{10}$$

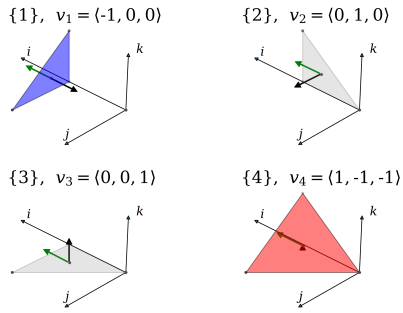
where i, j, and k are indices and N is a parametric size parameter. The write function here is $f_p = \{[i, j, k] \rightarrow [i]\}$ and therefore the accumulation space is the 2-dimensional jk-plane, $\mathcal{A} = \{[i, j, k] \mid i = 0\}$. Similarly, the read function here is $f_d = \{[i, j, k] \rightarrow [k]\}$ and therefore the reuse space is the 2-dimensional ij-plane, $\mathcal{R} = \{[i, j, k] \mid k = 0\}$. This set has four inequality constraints: c_1 $(i \leq N)$, c_2 $(0 \leq j)$, c_3 $(0 \leq k)$, and c_4 $(k \leq i - j)$. Geometrically, the shape of $\mathcal{D}$ is a tetrahedron with 4 faces (2-faces), 6 edges (1-faces), and 4 vertices (0-faces). The face lattice of $\mathcal{D}$ is illustrated in Figure 3. The numbers in each node in the lattice denote which constraints are saturated. For example, the 2-face labeled "{1}" represents the 2-face obtained by saturating c_1 (i.e., the triangular face at $i = N$). Each edge in the lattice can be thought of as saturating one additional constraint.

3.4 Single-Step Simplification

Our working example involves a reduction with the addition operator. The i'th result, Y_i, is the accumulation of the values of the reduction body at points in the i'th triangular slice of the tetrahedron. The complexity of the equation is the number of integer points in $\mathcal{D}$, namely $O(N^3)$. Simplification is possible because the body has redundancy along any vector $\vec{\rho} \in \mathcal{R}$, such as $\vec{\rho} = \langle 1, 0, 0 \rangle$ (green arrow) shown in Figure 4. Any two points $[i, j, k], [i', j', k'] \in \mathcal{D}$ separated by a



(a) Translate $\mathcal{D}$ (shift red pyramid) by $\vec{\rho}$ (green arrow) yielding $\mathcal{D}_s$ (blue pyramid).



(b) Evaluate the residual computation on only the facets of $\mathcal{D}$ and/or $\mathcal{D}_s$.

Fig. 4. Simplification of a cubic equation (computation defined over a tetrahedron) to a quadratic complexity (the residual computation is defined only over the points in the top red triangular face).

scalar multiple of $\vec{\rho}$ read the same value of X because $X_k = X_{k'}$. In other words, the body expression evaluates to the same value at all points along $\langle 1, 0, 0\rangle$. Simplification exploits this reuse to read and compare only the $O(N^2)$ distinct values. A geometric explanation of simplification is:

(1) Translate $\mathcal{D}$ (shift the red pyramid) by $\vec{\rho}$ (green arrow) yielding $\mathcal{D}_s$ (the blue pyramid).
(2) Delete all computations in the intersection of the two.
(3) Evaluate the residual computation on only (a subset of) the facets (here, 2-faces) of $\mathcal{D}$ and/or $\mathcal{D}_s$.

Additionally, some of these facets can be ignored. This is a ***critical*** component of how we will construct splits and is discussed further in Sections 4.2 and 4.3. For now, note that the grey triangular 2-faces "{2}" and "{3}", whose normal vectors are orthogonal to $\vec{\rho}$, were already included in the intersection, and the back blue 2-face "{1}" one at $i = N + 1$ is *external* since it does not contribute to any answer. This leaves a residual computation on only the top red oblique 2-face "{4}". Thus, the $O(N^3)$ computation in Equation 9 can be rewritten as the following $O(N^2)$ equation:

$$Y_i = \begin{cases} i = 0 & : \quad \sum_{j=0}^{j \le i} X_{i-j} \\ i > 0 & : \quad Y_{i-1} + \sum_{j=0}^{j \le i} X_{i-j} \end{cases} \tag{11}$$

obtained from the application of Theorem 5 from GR06, which we do not review in detail here. The intuition is that simplifying Equation 9 along $\vec{\rho} = \langle 1, 0, 0\rangle$ *moves* the computation to the "{4}" face, which is described by the saturated constraint $k = i - j$. Thus the subscript, k, on X from Equation 9 is expressed as X_{i-j} in Equation 11.

3.5 Equivalence Partitioning of Infinite Choices

In this example, the reuse space is multi-dimensional, which means there are infinitely many choices of reuse along which simplification can be performed. However, not all choices of reuse need to be explored. The intuition is that any two reuse vectors resulting in the same combination of residual computation can be viewed as the same candidate choice. In the previous section, the choice of $\vec{\rho} = \langle 1, 0, 0\rangle$ resulted in a residual computation on only the top oblique 2-face. GR06 refers

to the subset of $\vec{\rho}$ that results in the same residual computation as an *equivalance class*. Then, the set of all equivalence classes can be explored with dynamic programming. This guarantees the final simplified program's optimality as long as a single reuse vector from each equivalence class is considered.

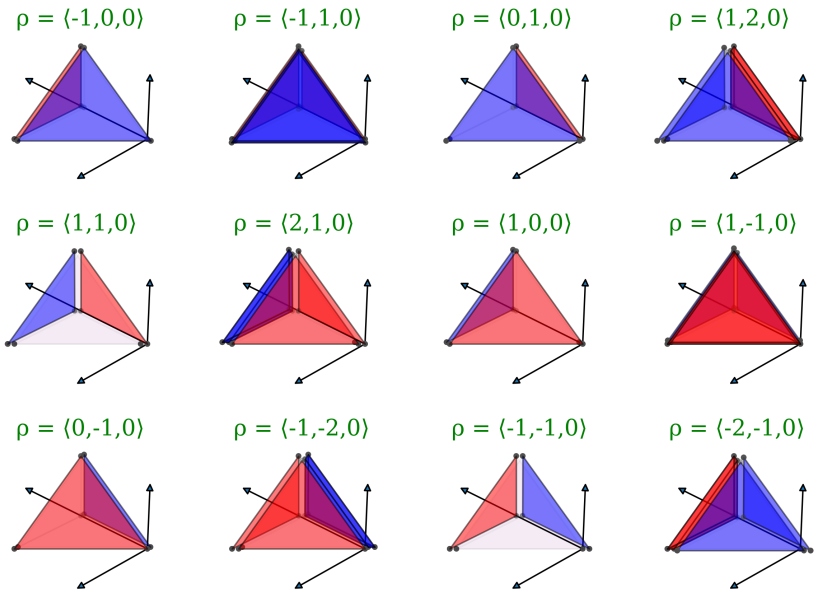


Fig. 5. The 12 equivalence classes for the single-step simplification of Equation 9.

Let $\mathcal{F}$ be an arbitrary facet of the reduction body. Let $\mathcal{F}_i$ denote the i'th facet of $\mathcal{F}$ (i.e., its i'th child), and let $\vec{v_i}$ be the linear part of the normal vector of $\mathcal{F}_i$. Let the symbol $\oplus$ be the reduction operator, and $\ominus$ be its inverse if $\oplus$ is invertible. The single-step simplification of $\mathcal{F}$, summarized in Section 3.4, with the reuse vector $\vec{\rho}$, results in a residual computation on some of its facets. The orientation of $\vec{\rho}$ relative to each facet dictates the type of residual computation that occurs on $\mathcal{F}_i$. There are three possibilities, depending on the sign of the dot product between $\vec{\rho}$ and $\vec{v_i}$:

(1) If $\vec{\rho} \cdot \vec{v_i} > 0$ then $\mathcal{F}_i$ contributes with the $\oplus$ operator.
(2) If $\vec{\rho} \cdot \vec{v_i} < 0$ then $\mathcal{F}_i$ contributes with the $\ominus$ operator.
(3) If $\vec{\rho} \cdot \vec{v_i} = 0$ then $\mathcal{F}_i$ does not contribute at all.

Each facet $\mathcal{F}_i$ may be labeled as either an $\oplus$-face, $\ominus$-face, or $\oslash$-face respectively. [2] We say that $\vec{\rho}$ *induces a particular labeling*, $\mathcal{L}$, on the facets of $\mathcal{F}$. Each labeling corresponds to one of the equivalence classes. For a face with m facets, there are a total of 3^m distinct labelings. However, many of these will not be possible. There is no way to mark all facets with the same label, for example, since the set of $\vec{\rho}$ with a positive dot product to all normal vectors in a convex polytope is empty. In this example, the 12 possible labelings (equivalence classes) are shown in Figure 5 with $\oplus$-faces colored in red, $\ominus$-faces in blue, and $\oslash$-faces uncolored.

[2]Labels are pronounced "positive", "negative", and "invariant" faces respectively. The notion of *labelings* are not part of the prior work of GR06, we introduce this language to help explain why simplification can fail in the following sections.

3.6 Recursive Simplification

In the general case, reductions have a d-dimensional reduction body, an a-dimensional accumulation, and an r-dimensional reuse space. The process of Section 3.4 is applied recursively on the face lattice, starting with $\mathcal{D}$. At each step, we simplify the facets of the current face $\mathcal{F}$. The key idea is that exploiting reuse along $\vec{\rho}$ avoids evaluating the reduction expression at most points in $\mathcal{F}$. Specifically, let $\mathcal{F}'$ be the translation of $\mathcal{F}$ along $\vec{\rho}$. Then all the computation in $\mathcal{F} \cap \mathcal{F}'$ is avoided, and we only need to consider the two differences $\mathcal{F}' \backslash \mathcal{F}$ and $\mathcal{F} \backslash \mathcal{F}'$, i.e., the union of some of the facets of $\mathcal{F}$.

At each recursive step down the face lattice, the asymptotic complexity is reduced by exactly one polynomial degree, as facets of $\mathcal{F}$ are strictly smaller dimensional subspaces. Furthermore, at each step, the newly chosen $\vec{\rho}$ is linearly independent of the previously chosen ones. Hence, the method is optimal—all available reuse is fully exploited. This holds regardless of the choice of $\vec{\rho}$ at any level of the recursion, even though there may be infinitely many choices.

Bringing everything together, the residual $O(N^2)$ computation in Equation 11 of our working example,

$$Y_i = \sum_{j=0}^{j \leq i} X_{i-j} \tag{12}$$

can be thought of as a completely new 2-dimensional reduction that may be further simplified. We do not describe this in detail here, as it is very similar to the prefix sum described in Section 2.1.

3.7 Simplification Enhancing Transformations

GR06 proposes several simplification-enhancing transformations that can be used to expand the reuse space. We briefly summarize one of them, which exploits commutativity, called *reduction decomposition* here, as we rely on this heavily in Section 6. Given two functions f_p'' and f_p' such that $f_p = f_p'' \circ f_p'$, a reduction of the form in Equation 8 with multi-dimensional accumulation may be rewritten as the following two reductions,

$$Y_{f_p''(z)} = \bigoplus_{z \in \mathcal{D}} Z_{f_p'(z)} \tag{13}$$

$$Z_{f_p'(z)} = \bigoplus_{z \in \mathcal{D}} X_{f_r(z)} \tag{14}$$

with the introduction of a new variable Z to hold partial answers. We can think of this as decomposing a higher dimensional reduction into a lower dimensional *reduction of reductions*, which is legal because the order of accumulation does not matter. This transformation is useful because it affects which facets can be ignored. We precisely characterize this in Section 4.2.

An Example. Our working example in Equation 9 has two dimensions of accumulation (i.e., along j and k) because $f_p = \{[i, j, k] \longrightarrow [i]\}$. Therefore Equation 9 could be explicitly written as the following double summation,

$$Y_i = \sum_{j=0}^{i} \sum_{k=0}^{i-j} X_k \tag{15}$$

where the inner reduction accumulates over k. We have not explicitly separated this into two separate equations. Still, the inner reduction should be interpreted as a reduction with an accumulation characterized by $f_p' = \{[i, j, k] \longrightarrow [i, j]\}$ (i.e., producing a 2-dimensional intermediate answer) and the outer reduction as one characterized by $f_p'' = \{[i, j] \longrightarrow [i]\}$. Note that there are many other

decompositions available. For example, the inner reduction could be expressed over j instead or any linear combination of j and k, for that matter. We discuss methods for constructing f'_p in Section 6.

4 Splitting to Avoid Simplification Failure

In this section, we present the intuition of our main result with a simple example. The subsequent sections provide the precise formulation and proofs for the general cases.

4.1 How and When Simplification Can Fail: Optimal vs. Maximal

As discussed in the previous section, the GR06 algorithm proceeds recursively down the face lattice, simplifying facets one by one while additional reuse exists. At each step of the recursion, on a particular face $\mathcal{F}$, several candidate choices of reuse are explored, one for each possible labeling of the facets of $\mathcal{F}$. For a particular labeling, each facet is treated as either an $\oplus$-face, $\ominus$-face, or $\oslash$-face. Let us assume now that the reduction operator does not admit an inverse, which means that $\ominus$-faces must be avoided. Recall that some facets may be ignored (e.g., all but the top red oblique 2-face in Figure 4), but some can not; let us refer to these facets that can *not* be ignored as *residual facets*. For a particular labeling, simplification is impossible when two residual facets exist with opposite labels, one $\oplus$-face and one $\ominus$-face. If all residual facets are $\oplus$-faces, then simplification can obviously proceed. Similarly, if they are all $\ominus$-faces, we can simply negate $\vec{\rho}$ to view them as $\oplus$-faces. The single-step simplification of Section 3.4 fails if all candidate choices of reuse induce labelings that involve residual facets with opposite labels.

4.1.1 An Example. Look back at the prefix max from Section 2.2. The reduction body is the triangle, $\mathcal{D} = \{[i, j] \mid 0 \leq j \leq i \leq N\}$ and the reuse space is the 1D space along the i-axis, $\mathcal{R} = \{[i, j] \mid j = 0\}$. This means that there are only two labelings, shown on the left of Figure 6. The labeling induced by $\vec{\rho} = \langle 1, 0\rangle$ involves a single residual $\oplus$-face. In contrast, the labeling induced by $\langle -1, 0\rangle$ involves two residual facets with opposite labels (i.e., one red and one blue coloring). Since at least one labeling exists with no oppositely labeled residual facets, simplification succeeds and enables us to write Equation 5. Note that the vertical patterned blue facet on the left triangle can be ignored because it is a boundary facet, as discussed in Section 4.2.

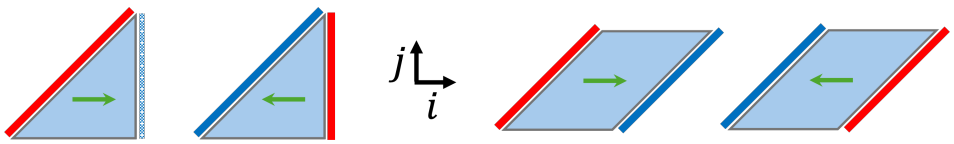


Fig. 6. The two labelings for the prefix max example from Section 2.2 shown by the triangles on the left. The two labelings for the parallelogram in Equation 16 are shown on the right. Residual facets are colored in solid red ($\oplus$-faces) and blue ($\ominus$-faces). The vertical patterned blue facet on the left triangle is an outward boundary facet. Therefore, it can be ignored, and thus, the labeling induced by $\langle 1, 0\rangle$ is possible since this does not involve residual facets with opposite labels.

Now consider the following example:

$$Y_i = \max_{\substack{(j \leq i) \wedge (j \leq N) \\ (0 \leq j) \wedge (i - N \leq j)}} X_j \tag{16}$$

This reduction has a body in the shape of a parallelogram with two possible labelings, as shown on the right of Figure 6. Since all (two) labelings involve at least one pair of residual facets with opposite labels, simplification fails.

4.1.2 Avoid Labelings With Oppositely Labeled Residual Facets. Given this formulation characterizing when simplification fails, our goal is to split the reduction body into pieces so that we can guarantee the existence of at least one labeling in each piece that involves only residual $\oplus$-faces exclusively or $\ominus$-faces, but not both. Before we can describe how to do this, we need to augment GR06's original characterization of non-residual facets, i.e., the facets that may be ignored, which we will call *boundary* and *invariant* facets.

4.2 Boundary Facets

Let $\mathcal{H}$ denote the linear space of the facet $\mathcal{F}$, defined by GR06 as the intersection of the effectively saturated constraints characterizing $\mathcal{F}$. An unsaturated constraint, c_i in $\mathcal{F}$ is characterized as a *boundary* constraint if the following condition holds:

$$\mathcal{H} \cap \mathcal{A} \subseteq \mathcal{H} \cap \ker(c_i) \tag{17}$$

where $\mathcal{A}$ is the accumulation space defined previously, and $\ker(c_i)$ is the null space of the linear part of the constraint. This says that a facet is a boundary if its linear space contains the entire accumulation space, i.e., multiple points *on that face* contribute to one or more answers. Boundary facets are useful because any boundary facet labeled as an $\ominus$-face can be ignored since the "answer(s)" are not needed.

We will make an additional distinction on the degree to which a facet is considered a boundary. Let us further characterize boundary facets as *strong* or *weak* based on the following definitions.

Definition 4.1. Let a facet of $\mathcal{F}$, defined by the effectively saturated constraint c_i, be called a ***strong*** boundary facet if the following condition holds,

$$\operatorname{rank}\big(\mathcal{A} \cap \ker(c_i)\big) = \operatorname{rank}\big(\mathcal{A}\big) \tag{18}$$

Definition 4.2. Let a facet of $\mathcal{F}$, defined by the effectively saturated constraint c_i, be called a ***weak*** boundary facet if the following condition holds,

$$0 < \operatorname{rank}\big(\mathcal{A} \cap \ker(c_i)\big) < \operatorname{rank}\big(\mathcal{A}\big) \tag{19}$$

In other words, a strong boundary facet is one where no other point contributes to the answer(s) to which the points on the facet contribute. In contrast, a facet whose linear space contains *part of* (i.e., has a non-trivial and incomplete intersection with) the accumulation space is said to be a weak boundary. Note that these are mutually exclusive; a facet can not be simultaneously strong and weak.

An example. The distinction between strong and weak boundaries only has meaning in 3D and higher. Look back at the working example from Section 3.3. The accumulation space in this example is the jk-plane. Of its four 2-faces, shown in Figure 4, the "{1}" face, at $i = N$, is a strong boundary because its linear space, the jk-plane contains the entire accumulation space. The other three 2-faces are all weak boundaries because the intersection of their linear spaces with the accumulation space is a 1D subspace.

4.3 Invariant Facets

GR06 does not explicitly characterize what we will call invariant facets. We can think of an invariant facet as the dual of a boundary facet, but from the perspective of the reuse space instead of the accumulation space. Let an unsaturated constraint c_i in $\mathcal{F}$ be characterized as an *invariant* constraint if the following condition holds:

$$\mathcal{H} \cap \mathcal{R} \subseteq \mathcal{H} \cap \ker(c_i) \tag{20}$$

where $\mathcal{R}$ is the reuse space. Like boundary facets, invariant facets are useful because the recursion never proceeds into an invariant facet (i.e., invariant facets are always labeled as $\oslash$-faces regardless of the choice of reuse $\vec{\rho}$).

Similarly, we distinguish the extent to which a facet is invariant based on the following definitions.

Definition 4.3. Let a facet of $\mathcal{F}$, defined by the effectively saturated constraint c_i, be called a ***strong*** invariant facet if the following condition holds,

$$\text{rank}\big(\mathcal{R} \cap \ker(c_i)\big) = \text{rank}\big(\mathcal{R}\big) \tag{21}$$

Definition 4.4. Let a facet of $\mathcal{F}$, defined by the effectively saturated constraint c_i, be called a ***weak*** invariant facet if the following condition holds,

$$0 < \text{rank}\big(\mathcal{R} \cap \ker(c_i)\big) < \text{rank}\big(\mathcal{R}\big) \tag{22}$$

Note that a facet may be simultaneously a weak invariant and a weak boundary. This happens when the intersection of accumulation space, reuse space, and linear space of the facet is non-trivial.

An example. Again, looking back at the working example in Section 3.3, the reuse space is the ij-plane. The bottom "{2}" face is„ therefore„ a strong invariant facet, and the other faces are weak invariant facets. In other words, any vector $\vec{\rho} = \langle \rho_i, \rho_j, 0 \rangle \in \mathcal{R}$ labels the bottom 2-face an $\oslash$-face because any such $\vec{\rho}$ is orthogonal to the normal vector of the bottom 2-face, $\langle 0, 0, 1 \rangle$.

4.4 Residual Facets

As mentioned in Section 4.1, we can ignore some of the facets during single-step simplification. Let us refer to the facets which can **not** be ignored as residual facets, which are defined as follows:

Definition 4.5. A facet of $\mathcal{F}$, defined by the effectively saturated constraint c_i, will be called a ***residual*** facet if the following condition holds,

$$\Big(\text{rank}\big(\mathcal{A} \cap \ker(c_i)\big) < \text{rank}\big(\mathcal{A}\big)\Big) \wedge \Big(\text{rank}\big(\mathcal{R} \cap \ker(c_i)\big) < \text{rank}\big(\mathcal{R}\big)\Big) \tag{23}$$

In other words, residual facets are neither strong boundary facets nor strong invariant facets. Residual facets are those into which the recursion may need to explore that can potentially be labeled as $\ominus$-faces.

4.5 Split Reduction

Le Verge [1992] showed that a reduction in the form of Equation 8, with the body $\mathcal{D}$ that can be partitioned into disjoint subsets $\mathcal{D}_1$ and $\mathcal{D}_2$ (i.e., $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2$ and $\mathcal{D}_1 \cap \mathcal{D}_2 = \phi$) may be rewritten as the following:

$$Y_{f_p(z)} = \begin{cases} f_p(\mathcal{D}_1) \cap f_p(\mathcal{D}_2) & : \Big(\bigoplus\limits_{z \in \mathcal{D}_1} X_{f_d(z)}\Big) \oplus \Big(\bigoplus\limits_{z \in \mathcal{D}_2} X_{f_d(z)}\Big) \\ f_p(\mathcal{D}_1) \setminus f_p(\mathcal{D}_2) & : \bigoplus\limits_{z \in \mathcal{D}_1} X_{f_d(z)} \\ f_p(\mathcal{D}_2) \setminus f_p(\mathcal{D}_1) & : \bigoplus\limits_{z \in \mathcal{D}_2} X_{f_d(z)} \end{cases} \tag{24}$$

Even though the domain is only split into *two* pieces, there may be *three* branches because the pieces may overlap in the answer space.

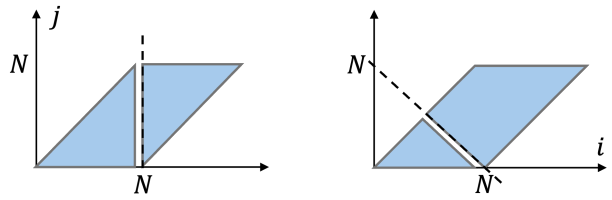


Fig. 7. Two example splits of the reduction body of Equation 16, the split along $i = N$ shown on the left and the split along $i + j = N$ on the right. Both cases have two pieces, but the split by $i + j = N$ results in three branches of the transformed equation because the pieces overlap when projected onto the answer space (i.e., the i-axis).

An example. The reduction body in the example from Equation 16 is the parallelogram, $\mathcal{D} = \{[i, j] \mid (0 \leq j \leq N) \wedge (i - N \leq j \leq i)\}$. Consider one split of $\mathcal{D}$ by the hyperplane $i = N$, where $\mathcal{D}_1 = \mathcal{D} \cap \{[i, j] \mid i < N\}$ and $\mathcal{D}_2 = \mathcal{D} \cap \{[i, j] \mid i \geq N\}$ shown on the left of Figure 7. Spliting the reduction in Equation 16 along $i = N$ yields:

$$Y_i = \begin{cases} 0 \leq i < N & : \max_{j=0}^{j \leq i} X_j \\ N \leq i \leq 2N & : \max_{j=i}^{j \leq N} X_j \end{cases} \tag{25}$$

Equation 25 only has two branches because the projections of the pieces onto the answer space do not overlap (i.e., project the triangles on the left of Figure 7 down onto the i-axis).

Consider another split $\mathcal{D}$ by the hyperplane $i + j = N$, where $\mathcal{D}_1 = \mathcal{D} \cap \{[i, j] \mid i + j < N\}$ and $\mathcal{D}_2 = \mathcal{D} \cap \{[i, j] \mid i + j \geq N\}$ shown on the right of Figure 7, yielding:

$$Y_i = \begin{cases} (N \leq 2i) \wedge (i < N) & : \max\left(\left(\max_{j \in \mathcal{D}_1} X_j\right), \left(\max_{j \in \mathcal{D}_2} X_j\right)\right) \\ 0 \leq 2i < N & : \max_{j \in \mathcal{D}_1} X_j \\ N \leq i \leq 2N & : \max_{j \in \mathcal{D}_2} X_j \end{cases} \tag{26}$$

In this case, there are three branches because the triangle and quadrilateral on the right of Figure 7 overlap from $N/2 \leq i < N$ when projected onto the i-axis.

4.6 Strong Boundary and Invariant Splits

In general, there are infinitely many ways to split the reduction body, $\mathcal{D}$, but not all are useful. Recall that simplification can fail when the reduction operator does not admit an inverse because a pair of oppositely labeled residual facets may be unavoidable. Splitting isolates some of the facets of $\mathcal{D}$ into one piece, making it possible to separate pairs of conflicting facets. Each piece retains some facets of $\mathcal{D}$ and obtains a single new facet characterized by the splitting hyperplane. We want to split $\mathcal{D}$ so that each piece's reuse space is less constrained, thereby enabling simplification.

However, an arbitrary split on $\mathcal{D}$ may undesirably further constrain the available reuse space of each piece because the newly introduced facet changes the set of new labelings permissible in each piece. However, we can avoid this issue by only making splits that introduce a new strong boundary or strong invariant facet. This is useful because strong boundary $\ominus$-faces and invariant facets are never residual and can, therefore, be ignored.

If $\mathcal{D}$ is d-dimensional, then let h be a $(d-1)$-dimensional hyperplane that separates $\mathcal{D}$ into two non-empty pieces. The hyperplane h is characterized by a single equality constraint, c_h. Let $\ker(c_h)$ denote the null space of the linear part of c_h.

Definition 4.6. Let h be called a strong boundary split if $\mathcal{A} \subseteq \ker(c_h)$.

Definition 4.7. Let h be called a strong invariant split if $\mathcal{R} \subseteq \ker(c_h)$.

For example, the vertical split at $i = N$ shown previously in Figure 7 illustrates a strong boundary split because the accumulation space of the reduction in Equation 16 is $\mathcal{A} = \{[i, j] \mid i = 0\}$, which is indeed a subset of the null space of the linear part of $i = N$ (they happen to be the same space in this example). Consequently, the resultant reductions in each branch of Equation 25 can be simplified independently because there exists a labeling involving only residual $\oplus$-faces for each. These are indeed both instances of the prefix max discussed in Section 2.2.

5 Problem Formulation and Hypotheses

As we have discussed, simplification is a powerful transformation that lowers the asymptotic complexity of the underlying computation. However, it is not always possible as motivated by the reduction examples in Equations 6 and 16. In this section, we make our primary claim in the form of Theorem 5.1. Like GR06, we assume that the input program only involves a single size parameter and any reductions present are *independent* (i.e., there is no cycle in the dependence graph among the variables appearing inside the reduction body and the answer variable on the left-hand side of the container equation).

Theorem 5.1. *Given an independent reduction with a d-dimensional body, an a-dimensional accumulation space, and an r-dimensional reuse space, it may always be transformed into an equivalent reduction with an asymptotic complexity that has been decreased by $l = \min(a, r)$ polynomial degrees.*

The proof of Theorem 5.1 will follow from Sections 6 and 7 where we show how to handle all possible combinations of the dimensionalities of the accumulation space, reuse space, and reduction body that can occur. For each case, we will show that the reduction can be split into pieces such that each piece has at least one possible labeling without any residual $\ominus$-faces.

5.1 Assumptions

We make several assumptions about the input reductions to justify Theorem 5.1. We emphasize that this does not introduce any loss of generality and the following assumptions are made solely to facilitate the proofs in the following sections.

5.1.1 Separate Accumulation and Reuse Dimensions Only. We must only consider reductions involving separate accumulation and reuse dimensions, where $a+r = d$. Any reduction where $a+r \neq d$ can be systematically transformed into one or more instances of reductions where $a + r = d$. Therefore, it is sufficient to only consider reductions with accumulation and reuse dimensions.

First, consider the case where $a + r > d$. In such cases, the accumulation and reuse spaces have a non-trivial intersection, which means that the reduction accumulates the *same value* at many points in the body (i.e., along this intersection). GR06 describes special simplification cases that may be applied to remove this intersection, which they call *higher order operator* and *idempotence* simplifications. The working example from Section 3.3 is in an instance of this case; the reduction body is 3-dimensional while the accumulation and reuse spaces are 2-dimensional, though we did not employ these special simplifications.

Second, consider the case where $a + r < d$. Such cases can be viewed as families of reductions, with $d - (a + r)$ *independent parameters*, reading independent slices of the inputs and producing

independent slices of the outputs. For example, the reduction:

$$Y[i, j] = \max_{k=i}^{2i} X[j, k]$$

has a 3D domain, 1D accumulation, and 1D reuse space. However, the index j should be viewed as an independent parameter. Thus, the overall computation can be viewed as $O(N)$ instances of independent 2D reductions, each with a 1D accumulation and 1D reuse space, one for each value of j, embedded in a 3D space. No simplification is possible among the different instances (each of the $O(N)$ reduction instances along j must be computed), but further simplification may be possible within the ik dimensions).

5.1.2 Orthogonal Accumulation and Reuse Along Canonical Axes. Let us assume that the accumulation and reuse spaces are *orthogonal* and are oriented along the canonical axes. Let the reuse space be along the first r canonical axes and the accumulation space be along the last a axes. The program variable domains and the indexing expressions can always be reindexed to put the accumulation and reuse along the canonic axes as described by Le Verge [1992].

5.1.3 Domain of the Reduction Body is a Simplex. We restrict our analysis to domains that are simplexes (i.e., hyper-triangular) based on the following definition, adapted from Gruber [2007].

Definition 5.2. A (d)-simplex is the (d)-dimensional polytope defined as the convex combination of $d + 1$ affinely independent vertices.

In practice, decomposing the domain into simplices may not always be necessary. However, we use properties of simplices to prove that simplification is always possible. There may be heuristic solutions to decide how to split non-simplices, but we do not consider any such approaches here. Therefore, in the remaining discussion, we assume that any reductions appearing in the input program have been preprocessed and initially decomposed into simplices.

Our maximal simplification result directly carries over to general polyhedral sets because any (d)-dimensional parametric polytope can be decomposed into the union of (d)-dimensional simplices. Triangulating multi-dimensional polytopes is common in computer graphics [13, 31], for example. Simplices have useful properties used in our proofs:

- Any (k)-face of an (d)-simplex is itself a (k)-simplex. The number of (k)-faces of a (d)-simplex are given by the binomial coefficients, $\binom{d+1}{k+1}$ unique combinations of its $d + 1$ vertices.
- A (d)-simplex can be split into two (d)-simplices by adding one new affinely independent vertex. See Lemma 5.3.

5.2 Simplex-Preserving Strong Boundary or Invariant Splits

Repeatedly trying to make arbitrary splits runs the risk of falling into an endless loop. We need a way to guarantee that the process of splitting will terminate. Recall that simplification of a reduction with the body $\mathcal{D}$ fails when every possible labeling involves one or more pairs of oppositely labeled *residual* facets. Therefore, if we can repeatedly split $\mathcal{D}$ in such a way that *both* preserves the total number of facets in each piece *and* strictly decreases the number of residual facets, then it will be possible to guarantee that each piece has a labeling with no conflicting residual facets.

We combine the following two ideas. First, we will use the fact that splitting a d-simplex through any of its $(d - 2)$-faces produces two d-simplices per Lemma 5.3 and therefore preserves the total number of facets in each piece. Second, we will use the notion of a strong boundary or invariant split, which introduces a single new *non-residual* facet, as described in Section 4.6. Combining these two ideas, by making strong boundary or invariant splits through $(d - 2)$-faces of simplices, guarantees that the process of splitting will produce pieces with strictly fewer residual facets because the

number of total facets in each piece remains the same. Such splits will be called Simplex-Preserving Strong Boundary (SPB) or Invariant (SPI) splits.

When the accumulation space is one-dimensional, an SPB split can be constructed by infinitely extending one of the $(d-2)$ faces along it. Similarly, when the reuse space is 1-dimensional, extending a $(d-2)$-face along it yields an SPI split. Since there are finitely many $(d-2)$-faces, there are finitely many candidate SPB and SPI splits to process.

Lemma 5.3. *Let a splitting hyperplane of a (d)-dimensional polytope be any $(d-1)$-dimensional hyperplane that has points on both sides of the hyperplane. Any splitting hyperplane that saturates a (d-2)-face of a (d)-simplex produces two (d)-simplices.*

Proof. By definition, an (d)-simplex is the convex combination of $(d+1)$ vertices. Additionally, every (k)-face is itself a (k)-simplex, which is just the simplex formed by the $(k+1)$ vertices of the (k)-face. Any splitting $(d-1)$-dimensional hyperplane that saturates a $(d-2)$-face contains its $(d-1)$ vertices. There are two remaining vertices, and these, by definition, can not be part of the splitting hyperplane. This is because the hyperplane is itself a $(d-1)$-simplex and if it contained one of these additional two vertices, then it would saturate an entire $(d-1)$-face of the domain. Therefore, we can consider these other two vertices as separate from the hyperplane. We will refer to these as vertex A and vertex B. The convex combination of vertices A and B forms a (1)-simplex (i.e., a (1)-face or 1-dimensional linear subspace that connects the vertices). Let point C be any point in this (1)-simplex. We can construct the following two new sets.

(1) The convex combination of the $(d-1)$ vertices on the $(d-2)$-face, vertex A, and point C.
(2) The convex combination of the $(d-1)$ vertices on the $(d-2)$-face, vertex B, and point C. Then take the set difference of this set with the previous set.

Since both are convex combinations of $(d+1)$ vertices, they are both, by definition, (d)-simplices. □

5.2.1 A 2D Example. The vertical split at $i = N$ illustrated previously in Figure 7 is analogous to an SPB split in the sense that it is a strong boundary split through a vertex. However, a parallelogram is not a simplex, of course. Regardless, we can compute the constraint $i = N$ as follows. First, compute the linear space, H, of the bottom right vertex of the parallelogram in Figure 7, as the intersection of the saturated constraints describing this vertex:

$$H = \{[i, j] \mid (j = 0) \wedge (j = i - N)\}$$

Next, construct the relation characterizing the 1-dimensional basis, b, of the accumulation space from the kernel of the write function, $f_p = \{[i, j] \rightarrow [i]\}$ in this example, to obtain the relation:

$$b = \{[i, j] \rightarrow [i, j + 1]\}$$

Then compute its transitive closure, b^*, to obtain the relation:

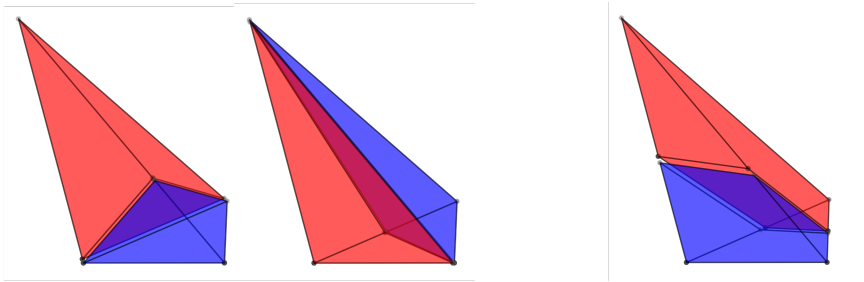
$$b^* = \{[i, j] \rightarrow [i, j']\}$$

Then apply b^* to the linear space of the vertex, H, to obtain the set:

$$\{[i, j] \mid i = N\}$$

Finally, the single equality constraint is used to characterize the splitting hyperplane $i = N$. Extending H by b^* effectively removes one of the equality constraints because extending it increases its dimensionality by one. Since we started from a set with two equality constraints (i.e., because H came from a $(d-2)$-face), its extension is guaranteed to have a single equality constraint. These operations are available in the integer set library, `isl` [47].

5.2.2 A 3D Example. In three dimensions, a simplex-preserving split is any plane passing through an edge of a tetrahedron. Any such plane splits it into two tetrahedra (one of which may be empty) as illustrated in Figure 8.



(a) Two boundary splits, each through an edge. (b) Arbitrary boundary split.

Fig. 8. *Tetrahedron*-preserving strong boundary splits (shown on left). Axes are not shown, but imagine that the accumulation space is 1D along the k-dimension pointing upwards. This means each split here is a strong boundary split per Section 4.6, but if the split does not pass through an edge, then the resulting pieces are not guaranteed to be tetrahedra, as shown on the right. Splits passing through edges, however, result in two tetrahedra, as shown on the left.

6 3-Dimensional and Higher Reductions

In this section, we consider the single-step simplification of Section 3.4 for 3-dimensional and higher reductions where all possible labelings involve at least one pair of oppositely labeled facets. Therefore, splitting the reduction body may be required. Our goal is to show that pairs of oppositely labeled residual facets can always be avoided. We will show that all but three residual facets can *either* be transformed into strong boundary facets by reduction decomposition *or* ignored by choosing a reuse vector that labels them as $\oslash$-faces. Two of the remaining three residual facets may involve opposite labels. Whenever this happens, we will split the domain so that all $\oplus$-faces are on one side of the split and all $\ominus$-faces are on the other. Then, in the piece containing the $\ominus$-face(s), we will negate $\vec{\rho}$ to view them as $\oplus$-faces.

We rely on the fact that any $(d-1)$-face has a non-trivial intersection with subsets of the accumulation and reuse space oriented along the canonical axes, per Lemma 6.1.

Lemma 6.1. *The linear space of any $(d-1)$-face has a non-trivial intersection with the linear subspace spanned by any two or more canonic axes.*

Proof. The $(d-1)$-dimensional linear space describing any $(d-1)$-face involves d indices and one equality constraint. The linear subspace spanned by q canonic axes involves d indices and $(d-q)$ equality constraints and is q-dimensional. Their intersection involves $(1+d-q)$ equality constraints. Therefore, it is $(q-1)$-dimensional. When $q > 1$, this intersection is non-trivial. □

6.1 Avoid Some Residual Facets With Reduction Decomposition

The reduction decomposition transformation described in Section 3.7 is useful because it has the effect of transforming a weak boundary into a strong boundary. This happens because the accumulation space of the inner reduction (i.e., the null space of f'_p in Equation 13) is strictly smaller after decomposition. Recall that a weak boundary, per Definition 4.2, is a facet that partially intersects the accumulation space. By constructing the inner accumulation space to only involve the

dimensions contained by a weak facet, the facet becomes a strong boundary in the inner reduction. Multiple weak boundary facets can simultaneously be transformed into strong boundary facets by constructing the inner accumulation as the subset of the accumulation space that they share.

Lemma 6.2. *Given a dimensions of accumulation, any set of a residual facets can be transformed into $a-1$ strong boundary facets via reduction decomposition.*

Proof. Let $[\mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}_k]$ be a list of k residual facets. Take the first residual facet, $\mathcal{F}_1$, and let $\mathcal{A}_1$ be the intersection of its linear subspace $\mathcal{H}_1$ and the a-dimensional accumulation space $\mathcal{A}$,

$$\mathcal{A}_1 = \mathcal{H}_1 \cap \mathcal{A} \tag{27}$$

$\mathcal{A}_1$ is a $(a-1)$-dimensional subspace per Lemma 6.1. Now let $\mathcal{A}_{p-1}$ denote the intersection of the first $(p-1)$ residual faces and the accumulation space,

$$\mathcal{A}_{p-1} = \mathcal{H}_1 \cap \mathcal{H}_2 \cap ... \cap \mathcal{H}_{p-1} \cap \mathcal{A} \tag{28}$$

$\mathcal{A}_{p-1}$ is a $(a-p+1)$-dimensional subspace. Now we decompose the reduction, per Section 3.7, where the inner reduction's accumulation space is precisely along $\mathcal{A}_{p-1}$. This is done for up to a residual facets since $\mathcal{A}_{p-1}$ is at least 1-dimensional when $p=a$. The inner reduction now has a 1D accumulation space, and $p-1$ of the residual facets are subsequently strong boundaries (i.e., non-residual). □

6.2 Avoid Some Residual Facets With Appropriate Reuse Selection

Any facets labeled as an $\oslash$-face can be ignored. Multiple facets can be labeled as an $\oslash$-face by selecting a reuse vector orthogonal to all of the facet normal vectors. The following proof is analogous to that of Theorem 6.2.

Lemma 6.3. *Given r dimensions of reuse, any set of $r-1$ residual facets can be labeled as $\oslash$-faces by choosing a reuse vector in their combined intersection with the reuse space.*

Proof. Let $[\mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}_k]$ be a list of k residual facets. Take the first facet $\mathcal{F}_1$ and let $\mathcal{R}_1$ be the intersection of its linear space $\mathcal{H}_1$ with the r-dimensional reuse space $\mathcal{R}$,

$$\mathcal{R}_1 = \mathcal{H}_1 \cap \mathcal{R} \tag{29}$$

$\mathcal{F}_1$ is an $(r-1)$-dimension subspace. Let $\mathcal{R}_{p-1}$ denote the intersection of the first $(p-1)$ residual facets and the reuse space,

$$\mathcal{R}_{p-1} = \mathcal{H}_1 \cap \mathcal{H}_2 \cap ... \cap \mathcal{H}_{p-1} \cap \mathcal{R} \tag{30}$$

$\mathcal{R}_{p-1}$ is a $(r-p+1)$-dimensional subspace. Any reuse vector $\vec{\rho} \in \mathcal{R}_{p-1}$ is orthogonal to the normal vectors of all $p-1$ facets and therefore labels all as $\oslash$-faces.

□

6.3 All Possible Scenarios

Now consider an arbitrary reduction over a d-dimensional simplicial domain with a dimensions of accumulation and r dimensions of reuse. There are $d+1$ facets on the reduction body because it is a simplex. We just showed, in the previous subsections, how to avoid up to $(a-1)+(r-1) = d-2$ of them using reduction decomposition and an appropriate choice of reuse. Consequently, there are only two possible scenarios that must be considered.

6.3.1 Base Case: $d-1$ or Fewer Residual Facets. In this case, $a-1$ residual facets can be transformed into strong boundaries per Lemma 6.2 and $r-1$ facets can be labeled as $\oslash$-faces per Lemma 6.3. This leaves at most one remaining residual facet because $(d-1)-(a-1)-(r-1) \leq 1$. At least two residual facets must be present for simplification to fail; therefore, simplification succeeds in this case.

6.3.2 General Case: d or More Residual Facets. Like the base case, $a - 1$ residual facets can be transformed into strong boundaries, and $r - 1$ facets can be labeled $\oslash$-faces. In this case, however, there can be up to three remaining residual facets since $(d + 1) - (a - 1) - (r - 1) \leq 3$. Among the remaining residual facets, two of them may involve opposite labels. In other words, there may be one $\oplus$-face and two $\ominus$-faces, or two $\oplus$-faces and one $\ominus$-face. In this case, a single SPB or SPI split can be made through one of their $(d - 2)$-faces to separate the conflicting facets.

7 2-Dimensional Reductions: Fractal Simplification of Triangles

Only one type of reduction can occur in two dimensions: a reduction with a 1D accumulation and 1D reuse space. Per Section 5.1.2, let the reuse space be oriented horizontally along the i-axis and accumulation vertically along the j-axis. This means that vertical and horizontal edges are boundary and invariant edges.

There are only three types of triangles that can occur:

(1) 1 residual edge (i.e., a right triangle, which is an instance of a standard scan, e.g., Section 2.2)
(2) 2 residual edges (some of which require fractal simplification)
(3) 3 residual edges, (can be split into two disjoint instances, each with 2 residual edges)

The following sections describe how to make 2-dimensional versions of SBP or SBI splits, previously described in Section 5.2, to obtain instances of the first case.

7.1 Base Case: Right Triangles

Any triangle with only one residual edge can always be simplified. If the residual edge monotonically increases, we exploit reuse along $\vec{\rho} = \langle 1, 0 \rangle$. This yields a standard scan (e.g., prefix-max, prefix-min, etc.). Otherwise, we exploit reuse along $\langle -1, 0 \rangle$, producing a backward scan (suffix-max, suffix-min, etc.)

7.2 Two Residual Edges

In this case, there is either one boundary (vertical) or one invariant (horizontal) edge. Let the three vertices of the triangle be, $\{\mathcal{V}_0, \mathcal{V}_1, \mathcal{V}_2\}$, of which, $\mathcal{V}_0$, which we call the corner vertex, is the intersection of the two residual edges. There are two sub-cases depending on the relative orientation of the vertices.

7.2.1 Covered Corner. Let the corner vertex (the vertex between the two residual edges) be called *covered* if it lands between the other two vertices when all vertices are projected onto either of the axes. As illustrated by the red point in Figure 9, and by the definition of $\mathcal{V}_0$ being covered, its projection on the other edge is a point on that edge. Hence, an SPB or SPI split through $\mathcal{V}_0$ yields two right triangles. One can be simplified as a forward-scan and the other as a backward-scan.

7.2.2 Corner Vertex Is Not Covered. There is no loss of generality in assuming that the entire triangle is in the positive quadrant and the corner vertex is at the origin (this can be accomplished by a simple reindexing of one or both of the input/output arrays). Let us first consider that the non-residual edge is vertical, and let $\mathcal{V}_1$ be *below* $\mathcal{V}_2$. If n is sufficiently large, we make one horizontal cut through the lower vertex, $\mathcal{V}_1$, and let $\mathcal{V}_2'$ be its intersection with the upper residual edge. Next, we make a vertical cut through $\mathcal{V}_2'$, intersecting the lower residual edge at, say, $\mathcal{V}_1'$. We now have three triangles, $\Delta_1 = [\mathcal{V}_0, \mathcal{V}_1', \mathcal{V}_2']$, $\Delta_2 = [\mathcal{V}_1', \mathcal{V}_2', \mathcal{V}_1]$ and $\Delta_3 = [\mathcal{V}_2', \mathcal{V}_1, \mathcal{V}_2]$. Of these, the latter two are right triangles, and Δ_1 is geometrically similar to the original triangle as illustrated on the left side of Figure 10.

Theorem 7.1. *The reduction over $[\mathcal{V}_0, \mathcal{V}_1, \mathcal{V}_2]$ can be simplified, i.e., computed with $O(1)$ reduction operations per element of the answer variable Y.*

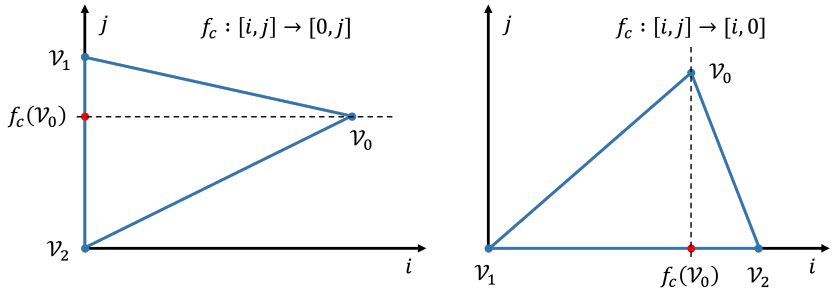


Fig. 9. Covered corner vertex: The projection of $\mathcal{V}_0$ (shown by the red point) is between $\mathcal{V}_1$ and $\mathcal{V}_2$. A single horizontal or vertical cut through the corner produces two right triangles.

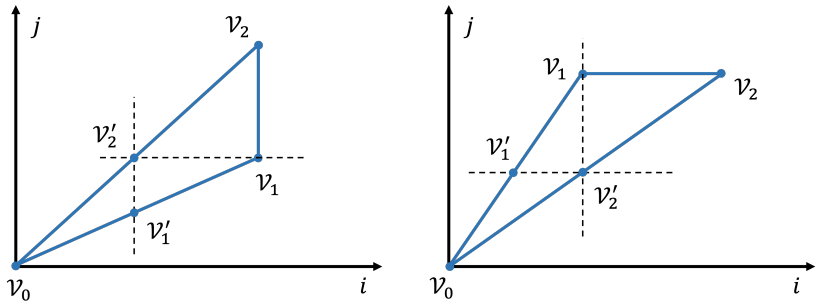


Fig. 10. Non-covered corner vertex: $\mathcal{V}_0$ lands *outside* of the other two vertices when projected onto the linear space of the vertical edge (left) or when projected onto the space of the invariant edge (right). Each case requires a single vertical and horizontal cut to produce two good right triangles in the base case and a third triangle, which is a homothetic scaling of the original triangle.

Proof. By Induction. As the base case, consider $n \leq c$ for some constant c. For such sufficiently small triangles, computing each element of Y needs only $O(1)$ reduction operations. In general, when n is sufficiently large, let us assume by induction that Δ_1 can be simplified to compute the section of the result until $\mathcal{V}_1'$ in $O(1)$ time per element of Y. Another backward scan and a forward scan on the appropriate section of X will yield the next section of Y from $\mathcal{V}_1'$ to $\mathcal{V}_1$. □

The above argument can be carried over mutatis mutandis to the case where the non-residual edge is horizontal. Indeed, as illustrated on the right side of Figure 10, the first cut of the previous case yields the second one.

The proof of Theorem 7.1 suggests a simple code generation strategy (again, explained for the case when the non-residual edge is vertical). Note that vertices $\mathcal{V}_1$ and $\mathcal{V}_2$ are given as affine functions of the parameter, n, and this is known statically. Let Let $\mathcal{V}_1 = [an, bn]$ and $\mathcal{V}_2 = [an, b'n]$ for some scalars a, b and b'. We can compute the values of a_1, b_1 and b_1' the factors that specify $\mathcal{V}_1'$ and $\mathcal{V}_2'$, and from that, the scaling factor, $\frac{a}{a_0} = \frac{b}{b_0'} = \frac{b'}{b_0'}$. This leads to the code structure shown in Figure 11. It is important to note that this code is recursive, not polyhedral, and takes us out of the polyhedral model. Nevertheless, all our analyses are polyhedral, and in order to generate it, we need only a fixed number of splits. Also, note that the recursive function calls can be optimized using standard tail recursion optimization techniques.

```
void fractal(int *Y, int *X, int L, int U) {
    if (U < threshold) {                        // do the full input reduction
        for (i=L; i<U; i++)
            for (j=i; j<=2*j; j++)
                Y[i] = max(Y[i], X[j]);
        return;
    }
    for (i=U; i>=U/2; i--)                      // backward scan on U/2<=i<=U
        Y[i] = max(Y[i+1], X[i]);
    for (i=U/2; i<=U; i++)                      // foward scan on U/2<=i<=U
        Y[i] = max(Y[i-1], X[2*i], X[2*i-1]);
    fractal(Y, X, L, U/2);                      // recurse on L<=i<U/2
}
```

Fig. 11. Recursive pseudo-code for fractal simplification of the example from Section 2.3. The complete working code can be found in the artifact [32].

7.3 Three Residual Edges

In this case any vertical cut through a covered vertex when projected onto the i-axis will produce two triangles in case 2. Alternatively, and equivalently, any horizontal cut through a covered vertex when projected on the j-axis will achieve the same thing.

8 Simplification With Splitting

In this section, we summarize the extended optimal simplification algorithm that incorporates our previously proposed splitting techniques.

8.1 Maximal Simplification Algorithm

Given an input reduction in the form of Equation 8, the maximal optimal simplification algorithm is summarized below in Algorithm 1. This should be understood precisely as the dynamic programming algorithm of GR06 (Algorithm 2 in their work) with an additional dynamic programming step to carry out splitting when necessary. Additionally, we require that the input reduction be initially preprocessed and separated into the union of disjoint simplices. We assume that has already been done or viewed as an additional preprocessing step as needed.

As stated previously, arbitrarily splitting reductions can lead to an endless loop; however, we can guarantee termination by only making simplex preserving strong boundary (SPB) and invariant (SPI) splits. Furthermore, we only do so when no other valid candidate choices exist. This is because the dynamic programming algorithm enumerates a finite number of choices based on the number of facets at the target reductions. Such splits strictly reduce the number of choices that need to be explored. Each candidate split in step 6 in Algorithm 1 consists either of a single split, for 3-dimensions and higher per Section 6, or two splits that expose a repeating pattern, in 2-dimensions per Section 7.

8.2 Implementation

We provide a proof-of-concept implementation of the individual components of our approach using the Alpha language [22, 30] and the AlphaZ system [53]. The Alpha language is an equational language that separates the specification of a program from its execution plan. Additionally, it supports modeling reduction operations as first-class objects with explicit representations of the write and read functions, f_p and f_d, characterizing the accumulation and reuse space, respectively.

Algorithm 1: Maximal Optimal Simplification

Input: d-dimensional reduction with a-dimensional accumulation and r-dimensional reuse. Assume that the body is a simplex and that the other preprocessing steps of GR06 have been done to expose the r dimensions of reuse.

Output: Equivalent optimal $\big(d - \min(a, r)\big)$-dimensional reduction(s)

```
while there are residual reductions with reuse do
    foreach residual reduction with body 𝒟 do
        Construct the set of candidate single-step simplifications among facets of 𝒟
        Construct the set of candidate reduction decompositions to transform one (or more)
          facets of 𝒟 into strong boundaries
        if there are no possible single-step simplification and decomposition candidates then
            Construct the set of candidate SPB and SPI splits
        end
        Use dynamic programming to optimally choose:
            (a) A single-step simplification of 𝒟 along candidate ρ⃗
            (b) A reduction decomposition candidate
            (c) A SPB or SPI candidate split to produce two new residual reductions
    end
end
```

Program variable domains are represented as polyhedral sets using `isl` [47] (the integer set library), which naturally supports the constraint representations we use to describe our splitting hyperplanes.

Our results in this paper are primarily theoretical. We provide, in our artifact [32], complete working code for the components of the maximal simplification algorithm, including constructing the set of all possible labelings and corresponding candidate splits. A complete push-button tool that implements the extended dynamic programming algorithm and automatically and optimally simplifies any reduction requires addressing several practical issues. Specifically, it is necessary to handle two issues: managing the combinatorially large number of possible solutions and developing methods to compare them using the constant factors, and the existence of parallel schedules with scalable locality. Putting all this together is outside the scope of this paper and left as future work.

This can be used to produce simplified Alpha programs for all of the examples discussed above. Additionally, AlphaZ can be used to generate C code that can be subsequently compiled and run. More information can be found in the accompanying artifact [32].

8.3 A Complete Higher Dimensional Example

Consider the $O(N^2)$ simplification of the following $O(N^4)$ reduction over the simplicial domain $\mathcal{D} = \{[i, j, k, l] \mid j \leq N \text{ and } i \leq k \leq 2i \text{ and } i + j \leq l \leq 2j\}$ with 2-dimensional accumulation and reuse,

$$Y[i, j] = \max_{[i,j,k,l] \in \mathcal{D}} X[k, l] \tag{31}$$

This is an interesting example because our splitting extensions are necessary, and among all possible simplification choices, with splitting, the fractal simplification step of Section 7 is unavoidable. Since 4-dimensional spaces are difficult to visualize, we illustrate each step by showing concrete loops. Due to space constraints, we only describe the details relevant to the key effect of each step. Complete working code can be found with the accompanying artifact [32].

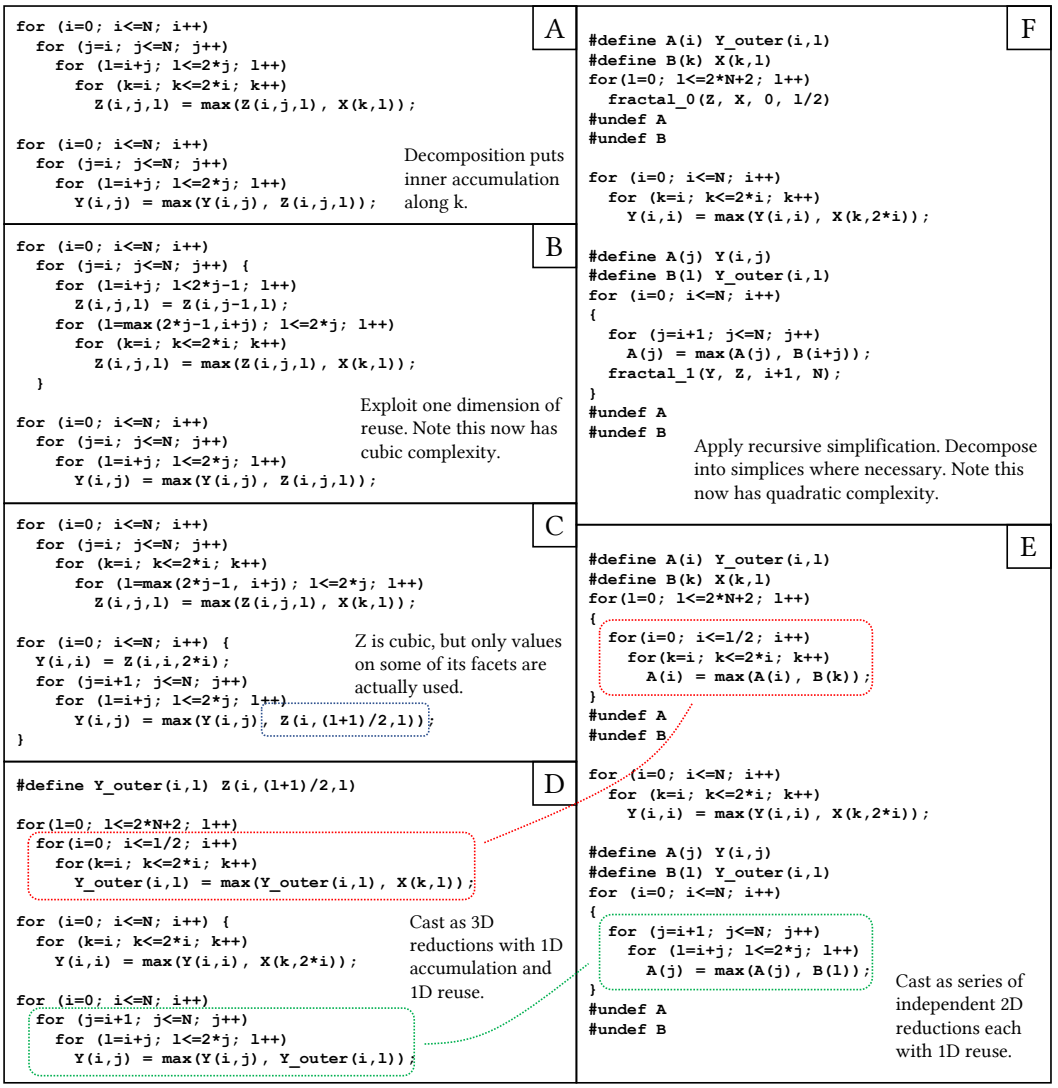


Fig. 12. Illustrating the $O(N^2)$ simplification procedure of the $O(N^4)$ reduction in Equation 31.

Step A - *reduction decomposition to put the inner accumulation along k.* The set of candidate single-step simplifications (step 3 of Algorithm 1) is empty. The only option is to explore reduction decomposition choices. Choosing a decomposition that puts the inner reduction along k and storing the partial answer in a temporary intermediate variable Z results in the loops shown in box A of Figure 12.

Step B - *simplification of the inner reduction.* The set of candidate single-step simplifications on the inner reduction is now non-empty. Applying single-step simplification using the reuse vector $\vec{\rho} = [0, 1, 0, 0]$ results in the $O(N^3)$ loops shown in box B.

Step C - *recover the residual reductions.* The single-step simplification by $\vec{\rho} = [0, 1, 0, 0]$ in this instance results in computing a 3-dimensional intermediate answer but GR06 describes and handles

the post-processing required to read only the needed values in the outer reduction. Z is cubic, but only values on some of its facets are actually used.

Step D *- preprocess to cast as series of independent reductions.* The residual reductions now have 3-dimensional bodies with 1-dimensional accumulation and reuse, which corresponds to the fact that we are left with a series of 2D reductions embedded in a 3D space, as described in Section 5.1.2.

Step E *- cast as series of independent reductions.* There are two residual 2D reductions embedded in a 3D space. Both of these involve triangular domains that require the fractal simplification of Section 7.

Step F *- Apply recursive fractal simplification.* Finally, computing the inner 2D reductions recursively results in the loops shown in box F. We don't explicitly show the definitions of `fractal_0` and `fractal_1` here, but they have the same structure as the code in Figure 11. The complete concrete code for this example can be found in the accompanying artifact [32].

9 Related Work

Simplification has garnered renewed interest recently. Asymptotic inefficiencies are present, even in deployed codes. Ding and Shen [8] noted that nine of the 30 benchmarks in Polybench 3.0 and two deployed PDE solvers have such inefficiencies. Separately, Yang et al. [52] showed that simplification is helpful for many algorithms in statistical learning like Gibbs Sampling (GS), Metropolis Hasting (MH), and Likelihood Weighting (LW). Their benchmarks include Gaussian Mixture Models (GMM), Latent Dirichlet Allocation (LDA), and Dirichlet Multinomial Mixtures (DMM). See their paper for details of benchmarks, algorithms, size parameters, machine specs, etc. They formulate and solve a more general problem: the reduction body may use (possibly transitively, through other variables) the output variable Y. So, it is necessary to solve the combined problem of simplification and scheduling. They propose a simple heuristic that handles all the examples they encounter and only applies to reduction operators that admit an inverse. Addressing these limitations, possibly using the techniques we present here, is an open problem.

Simplification is related but complementary to the problems of *marginalization of product functions* (MPF) and its discrete version, *tensor contraction* (TC). Such problems arise in many domains. MPF can be optimally computed using Pearl's "summary passing" or "message passing" algorithm [35] for Bayesian inference, or the generalized distributive law [2, 19, 45]. Similarly, there is a long history of research on optimal implementations of TC [18, 36, 43]. In this problem, the sizes of the tensors in each dimension are known, and we seek the implementation with the minimum number of operations.

To explain the problem, first note that the cost of multiplying three matrices, A, B, and C, is affected by how associativity is exploited: if A and C are short-stout, and B is tall-thin, $(AB)C$ is better than $A(BC)$, and if A and C are tall-thin and B is short-stout, the latter parenthesization is. Indeed, optimizing this for a sequence of matrices is a classic textbook problem used to illustrate dynamic programming. However, the underlying matrix multiplication uses the standard cubic algorithm. TC extends this to multiple chains of products (requiring us to optimally identify and exploit common subexpressions) and to tensors rather than matrices (exposing opportunities to exploit simplification by inserting new variables).

Specifically, consider the following system of equations:

$$X_{i,l} = \sum_{j,k=1}^{N} A_{i,j,l} \times B_{i,k} \qquad\qquad Y_j = \sum_{i,k,j=1}^{N} A_{i,j,l} \times B_{i,k} \tag{32}$$

Naively, each equation would have $O(N^4)$ complexity since each result is the accumulation of a triple summation, and there are $O(N)$ answers. However, if we define two new variables:

$$T_{i,l} = \sum_{j=1}^{N} A_{i,j,l} \qquad T'_i = \sum_{k=1}^{N} B_{i,k} \tag{33}$$

then the following is an equivalent simplified system of equations whose complexity is only $O(N^3)$:

$$X_{i,l} = T_{i,l} \times T'_i \qquad Y_j = \sum_{i,l=1}^{N} A_{i,j,l} \times T'_i \tag{34}$$

However, the reuse space, the accumulation space, and the domain constraints (loop bounds) are simple, and simple transformations like loop permutation can explore the space of all possible simplifications. The space of choices may be combinatorially large but not infinite, as tackled by the GR06 algorithm. There may be a need for more sophisticated) simplifications if the tensors have a special structure. Nevertheless, the reduction operation here admits an inverse, and there is no need for our results in this paper.

10 Conclusions and Open Questions

The ultimate goal is to enable compilers to take a high-level application program specification and carry it out in the most efficient way possible, preferably *automatically* and *optimally*. This work takes a step in that direction to enable users (i.e., application scientists and programmers) to focus less on the *engineering* aspects (the *how*) of their algorithms and more on the problems (the *what*) that their algorithms are intended to solve. In this work, we have studied reuse-based simplification of polyhedral reductions. The simplification transformation proceeds recursively down the face lattice of the domain of the reduction body and attempts to exploit one dimension of available reuse at each level. Previously, at some point along this traversal, it may not have been possible to employ the simplification transformation without requiring inverse operations. However, as we have shown, it is always possible to split the residual reduction at the problematic node in the lattice in such a way that we can guarantee that the simplification transformation will not fail. We have provided the mathematical proofs which enable us to make this claim.

We showed how to maximally simplify any arbitrary independent commutative reduction to obtain the optimal asymptotic complexity. We provided a proof-of-concept implementation of the individual components of our approach as an accompanying software artifact [32] that can be used to generate code. Along these same lines, additional work concerning traditional compile-time scheduling and efficient code generation is still needed. For example, we rely on the fact that any polyhedral set can be decomposed into a union of disjoint simplices, but we do not discuss the implications of the choice of decomposition on the efficiency of the resulting code. At this point, for example, it is not obvious why we should prefer one decomposition over another. It may be the case that one particular simplicial decomposition scheme leads to code that is more amenable to vectorization. These are interesting questions that require future exploration.

Acknowledgments

The work of the first author was partially supported by the National Science Foundation under Grant No. 2318970.

References

[1] 2007. Convex Polytopes. In *Convex and Discrete Geometry*, Peter M. Gruber (Ed.). Springer, Berlin, Heidelberg, 243–351. https://doi.org/10.1007/978-3-540-71133-9_3

[2] S. M. Aji and R. J. McEliece. 2000. The Generalized Distributive Law. *IEEE Transactions on Information Theory* 46, 2 (March 2000), 325–343. Number: 2.

[3] S. Amarasinghe and M. Lam. 1993. Communication Optimization and Code Generation for Distributed Memory Machines. In *The ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*. ACM Press, Albuquerque, N.M., 126–138.

[4] Uday Bondhugula, Vinayaka Bandishti, Albert Cohen, Guillain Potron, and Nicolas Vasilache. 2014. Tiling and optimizing time-iterated computations over periodic domains. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 39–50. https://doi.org/10.1145/2628071.2628106

[5] Michal J. Boniecki, Grzegorz Lach, Wayne K. Dawson, Konrad Tomala, Pawel Lukasz, Tomasz Soltysinski, Kristian M. Rother, and Janusz M. Bujnicki. 2016. SimRNA: a coarse-grained method for RNA folding simulations and 3D structure prediction. *Nucleic Acids Research* 44, 7 (April 2016), e63. https://doi.org/10.1093/nar/gkv1479

[6] Karl Bringmann, Fabrizio Grandoni, Barna Saha, and Virginia Vassilevska Williams. 2019. Truly Subcubic Algorithms for Language Edit Distance and RNA Folding via Fast Bounded-Difference Min-Plus Product. *SIAM J. Comput.* 48, 2 (Jan. 2019), 481–512. https://doi.org/10.1137/17M112720X Publisher: Society for Industrial and Applied Mathematics.

[7] Hamidreza Chitsaz, Rolf Backofen, and S.Cenk Sahinalp. 2009. biRNA: Fast RNA-RNA Binding Sites Prediction. In *Workshop on Algorithms in Bioinformatics (WABI) (LNBI, Vol. 5724)*, S.L. Salzberg and T. Warnow (Eds.). Springer-Verlag, Berlin, Heidelberg, 25–36.

[8] Yufei Ding and Xipeng Shen. 2017. GLORE: generalized loop redundancy elimination upon LER-notation. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), 74:1–74:28. https://doi.org/10.1145/3133898 Number: OOPSLA.

[9] P. Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (Feb. 1991), 23–53. Number: 1.

[10] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming* 21, 6 (Dec. 1992), 389–420. https://doi.org/10.1007/BF01379404 Number: 6.

[11] C. Flamm, W. Fontana, I. L. Hofacker, and P. Schuster. 2000. RNA folding at elementary step resolution. *RNA* 6 (March 2000), 325–338.

[12] J. A. B. Fortes and D. I. Moldovan. 1984. Data broadcasting in linearly scheduled array processors. *ACM SIGARCH Computer Architecture News* 12, 3 (Jan. 1984), 224–231. https://doi.org/10.1145/773453.808186 Number: 3.

[13] S. Ganapathy and T. G. Dennehy. 1982. A new general triangulation method for planar contours. *SIGGRAPH Comput. Graph.* 16, 3 (July 1982), 69–75. https://doi.org/10.1145/965145.801264

[14] Gautam and S. Rajopadhye. 2006. Simplifying reductions. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '06)*. Association for Computing Machinery, New York, NY, USA, 30–41. https://doi.org/10.1145/1111037.1111041

[15] M. Griebl, P. Feautrier, and C. Lengauer. 2000. Index Set Splitting. *IJPP: International Journal of Parallel Programming* 28, 6 (Dec. 2000), 607–631. https://doi.org/10.1023/A:1007516818651 Number: 6.

[16] Liang Huang, He Zhang, Dezhong Deng, Kai Zhao, Kaibo Liu, David A Hendrix, and David H Mathews. 2019. LinearFold: linear-time approximate RNA folding by 5'-to-3' dynamic programming and beam search. *Bioinformatics* 35, 14 (July 2019), i295–i304. https://doi.org/10.1093/bioinformatics/btz375

[17] F. Irigoin and R. Triolet. 1988. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '88)*. Association for Computing Machinery, New York, NY, USA, 319–329. https://doi.org/10.1145/73560.73588

[18] Martin Kong, Raneem Abu Yosef, Atanas Rountev, and P. Sadayappan. 2023. Automatic Generation of Distributed-Memory Mappings for Tensor Computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3581784.3607096

[19] F. R. Kschischang, B. J. Frey, and H-A. Loeliger. 2001. Factor Graphs and the Sum-Product Algorithm. *IEEE Transactions on Information Theory* 47, 2 (Feb. 2001), 498–519. Number: 2.

[20] M. S. Lam. 1989. *A Systolic Array Optimizing Computer*. Kluwer Academic (Springer). https://doi.org/10.1007/978-1-4613-1705-0

[21] HERVE LE VERGE. 1992. *Un environnement de transformations de programmes pour la synthese d'architectures regulieres*. These de doctorat. Rennes 1. https://theses.fr/1992REN10139

[22] H. Le Verge, C. Mauras, and P. Quinton. 1991. The ALPHA Language and its use for the Design of Systolic Arrays. *Journal of VLSI Signal Processing* 3, 3 (Sept. 1991), 173–182. Number: 3.

[23] Christian Lengauer. 1993. Loop parallelization in the polytope model. In *CONCUR'93*, Eike Best (Ed.). Springer, Berlin, Heidelberg, 398–416. https://doi.org/10.1007/3-540-57208-2_28

[24] Guo-Jie Li and Benjamin W. Wah. 1985. Systolic Processing for Dynamic Programming Problems. In *International Conference on Parallel Processing, ICPP'85*. IEEE Computer Society Press, 434–441.

[25] Vincent Loechner and Doran K. Wilde. 1997. Parameterized Polyhedra and Their Vertices. *IJPP: International Journal of Parallel Programming* 25, 6 (Dec. 1997), 525–549. https://doi.org/10.1023/A:1025117523902 Number: 6 Publisher: Springer Verlag.

[26] Ronny Lorenz, Stephan H. Bernhart, Christian Höner zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F. Stadler, and Ivo L. Hofacker. 2011. ViennaRNA Package 2.0. *Algorithms for Molecular Biology* 6, 1 (Nov. 2011), 26. https://doi.org/10.1186/1748-7188-6-26 Number: 1.

[27] Ronny Lorenz, Ivo L. Hofacker, and Peter F. Stadler. 2016. RNA folding with hard and soft constraints. *Algorithms for Molecular Biology* 11, 1 (April 2016), 8. https://doi.org/10.1186/s13015-016-0070-z

[28] R B Lyngsø, M Zuker, and C N Pedersen. 1999. Fast evaluation of internal loops in RNA secondary structure prediction. *Bioinformatics* 15, 6 (June 1999), 440–445. https://doi.org/10.1093/bioinformatics/15.6.440 Number: 6.

[29] David H Mathews and Douglas H Turner. 2006. Prediction of RNA secondary structure by free energy minimization. *Current Opinion in Structural Biology* 16, 3 (June 2006), 270–278. https://doi.org/10.1016/j.sbi.2006.05.010 Number: 3.

[30] Christophe Mauras. 1989. *Alpha : un langage equationnel pour la conception et la programmation d'architectures paralleles synchrones*. These de doctorat. Rennes 1. https://theses.fr/1989REN10116

[31] Atul Narkhede and Dinesh Manocha. 1995. VII.5 - Fast Polygon Triangulation Based on Seidel's Algorithm. In *Graphics Gems V*, Alan W. Paeth (Ed.). Academic Press, Boston, 394–397. https://doi.org/10.1016/B978-0-12-543457-7.50059-0

[32] Louis Narmour. 2024. Maximal Simplification of Polyhedral Reductions. https://doi.org/10.5281/zenodo.13943008

[33] R Nussinov and A B Jacobson. 1980. Fast algorithm for predicting the secondary structure of single-stranded RNA. *Proceedings of the National Academy of Sciences of the United States of America* 77, 11 (Nov. 1980), 6309–6313. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC350273/ Number: 11.

[34] OpenMP Architecture Review Board. 2021. {OpenMP} Application Program Interface Version 5.2. 124–140. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

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

[36] Robert N. C. Pfeifer, Jutho Haegeman, and Frank Verstraete. 2014. Faster identification of optimal contraction sequences for tensor networks. *Physical Review E* 90, 3 (Sept. 2014), 033315. https://doi.org/10.1103/PhysRevE.90.033315 Publisher: American Physical Society.

[37] William Pugh. 1991. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing (Supercomputing '91)*. Association for Computing Machinery, New York, NY, USA, 4–13. https://doi.org/10.1145/125826.125848

[38] P. Quinton and V. Van Dongen. 1989. The Mapping of Linear Recurrence Equations on Regular Arrays. *Journal of VLSI Signal Processing* 1, 2 (1989), 95–113. Number: 2 Publisher: Kluwer Academic Publishers, Boston.

[39] S. V. Rajopadhye. 1989. Synthesizing Systolic Arrays with Control Signals from Recurrence Equations. *Distributed Computing* 3 (May 1989), 88–105. Publisher: Elsevier Science, North Holland.

[40] S. V. Rajopadhye, L. Mui, and S. Kiaei. 1992. Piecewise Linear Schedules for Recurrence Equations. In *VLSI Signal Processing, V*, K. Yao, R. Jain, W. Przytula, and J. Rabbaey (Eds.). IEEE Signal Processing Society, Napa, 375–384.

[41] J. Ramanujam and P. Sadayappan. 1990. Nested Loop Tiling for Distributed Memory Machines. In *Proceedings of the Fifth Distributed Memory Computing Conference, 1990.*, Vol. 2. IEEE, Charleston, SC, USA, 1088–1096. https://doi.org/10.1109/DMCC.1990.556321

[42] Harenome Razanajato, Vincent Loechner, and Cédric Bastoul. 2017. Splitting Polyhedra to Generate More Efficient Code. https://inria.hal.science/hal-01505764

[43] Gerald Sabin and P. Sadayappan. 2021. *Tensor Contraction and Operation Minimization forExtreme Scale Computational Chemistry*. Technical Report DOE-RNET-20616. RNET Technologies. https://www.osti.gov/biblio/1782724

[44] Robert Schreiber and Jack J. Dongarra. 1990. *Automatic blocking of nested loops*. Technical Report NASA-CR-188874. RIACS. https://ntrs.nasa.gov/citations/19910023530 Issue: NASA-CR-188874 NTRS Author Affiliations: Research Inst. for Advanced Computer Science, Tennessee Univ. NTRS Document ID: 19910023530 NTRS Research Center: Legacy CDMS (CDMS).

[45] G. R. Shafer and P. P. Shenoy. 1990. Probability Propagation. *Annals of Mathematics and Artificial Intelligence* 2, 1-4 (March 1990), 327–351. Number: 1-4.

[46] Nicolas Vasilache, Albert Cohen, and Louis-Noel Pouchet. 2007. Automatic Correction of Loop Transformations. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. 292–304. https://doi.org/10.1109/PACT.2007.4336220

[47] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software – ICMS 2010*, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Springer, Berlin, Heidelberg, 299–302. https://doi.org/10.1007/978-3-642-15582-6_49

[48] M. Wolf and M. Lam. 1991. Loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems* 2, 4 (Oct. 1991), 452–471. Number: 4.

[49] Michael E. Wolf and Monica S. Lam. 1991. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation - PLDI '91*. ACM Press, Toronto, Ontario, Canada, 30–44. https://doi.org/10.1145/113445.113449

[50] Michael Wolfe. 1987. Iteration Space Tiling for Memory Hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, USA, 357–361.

[51] Michael T. Wolfinger, Andreas, Christoph Flamm, Ivo L. Hofacker, and Peter F. Stadler. 2004. Efficient computation of RNA folding dynamics. *Journal of Physics A: Mathematical and General* 37, 17 (2004), 4731–4741. Number: 17.

[52] Cambridge Yang, Eric Atkinson, and Michael Carbin. 2021. Simplifying dependent reductions in the polyhedral model. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 20:1–20:33. https://doi.org/10.1145/3434301 Number: POPL.

[53] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2013. AlphaZ: A System for Design Space Exploration in the Polyhedral Model. In *Languages and Compilers for Parallel Computing*, Hironori Kasahara and Keiji Kimura (Eds.). Springer, Berlin, Heidelberg, 17–31. https://doi.org/10.1007/978-3-642-37658-0_2

[54] Joseph N Zadeh, Conrad D Steenberg, Justin S Bois, Brian R Wolfe, Marshall B Pierce, Asif R Khan, Robert M Dirks, and Niles A Pierce. 2011. NUPACK: Analysis and design of nucleic acid systems. *Journal of computational chemistry* 32, 1 (2011), 170–173. Number: 1 Publisher: Wiley Subscription Services, Inc., A Wiley Company Hoboken.

[55] M Zuker and P Stiegler. 1981. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Research* 9, 1 (Jan. 1981), 133–148. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC326673/ Number: 1.

Received 2024-07-11; accepted 2024-11-07