Automated derivation of parametric data movement lower bounds for affine programs

Auguste Olivry, Julien Langou, Louis-Noël Pouchet, P. Sadayappan, Fabrice Rastello

Automated Derivation of Parametric Data Movement Lower Bounds for Affine Programs*

Auguste Olivry
Univ. Grenoble Alpes,
CNRS, Inria, Grenoble INP, LIG
38000 Grenoble, France

Julien Langou
University of Colorado Denver
Denver, CO, USA

Louis-Noël Pouchet
Colorado State University
Fort Collins, CO, USA

P. Sadayappan
University of Utah
Salt Lake City, UT, USA

Fabrice Rastello
Univ. Grenoble Alpes,
Inria, CNRS, Grenoble INP, LIG
38000 Grenoble, France

Abstract

Researchers and practitioners have for long worked on improving the computational complexity of algorithms, focusing on reducing the number of operations needed to perform a computation. However the hardware trend nowadays clearly shows a higher performance and energy cost for data movements than computations: quality algorithms have to minimize data movements as much as possible.

The theoretical operational complexity of an algorithm is a function of the total number of operations that must be executed, regardless of the order in which they will actually be executed. But theoretical data movement (or, I/O) complexity is fundamentally different: one must consider all possible legal schedules of the operations to determine the minimal number of data movements achievable, a major theoretical challenge. I/O complexity has been studied via complex manual proofs, e.g., refined from $\Omega(n^3/\sqrt{S})$ for matrix-multiply on a cache size S by Hong & Kung to $2n^3/\sqrt{S}$ by Smith et al. While asymptotic complexity may be sufficient to compare I/O potential between broadly different algorithms, the accuracy of the reasoning depends on the tightness of these I/O lower bounds. Precisely, exposing constants is essential to enable precise comparison between different algorithms: for example the $2n^3/\sqrt{S}$ lower bound allows to demonstrate the optimality of panel-panel tiling for matrix-multiplication.

We present the first static analysis to automatically derive non-asymptotic parametric expressions of data movement lower bounds with scaling constants, for arbitrary affine computations. Our approach is fully automatic, assisting algorithm designers to reason about I/O complexity and make educated decisions about algorithmic alternatives.

CCS Concepts: • **Theory of computation → Design and analysis of algorithms**; • **Software and its engineering → Automated static analysis**.

Keywords: Data access complexity; *I/O* lower bounds; Static analysis; Affine programs

1 Introduction

The performance impact of operations and data movement latencies in current architectures can often be effectively masked by using hardware-pipelined implementations. But the volume of data movements required by even an idealized implementation of an algorithm will impose fundamental limits: any implementation of that algorithm will have its performance and energy requirements bounded by this limit [12, 18, 25, 26, 28, 29]. Providing algorithm designers with tools to characterize this fundamental limit is crucial.

Memory movements can be efficiently tracked for a particular algorithm *implementation*, and it is standard practice for performance debugging [1]: Hardware counters can be used to measure cache misses and data traffic. But two different implementations of the same algorithm may have dramatically different memory movement profiles: for example a carefully tiled implementation of matrix multiplication would significantly reduce cache misses versus a naive, untiled one.

In general, determining whether an implementation is sub-optimal or whether the fundamental nature of the algorithm is the limiting factor for the observed cache miss count is crucial. We propose an *automatic* system to answer this question, potentially alleviating the need for the algorithm designers to produce a concrete optimized implementation. As we specifically target the production of non-asymptotic I/O lower bounds, our system also makes it possible for performance experts to reason about the optimality of their implementations with respect to data movement.

Our fully implemented framework IOLB (for *I/O Lower Bounds*) automatically derives parametric lower bounds with

*This work was supported in part by the U.S. National Science Foundation awards 1645514, 1645599, 1750399 and 1816793.

PLDI '20, June 15–20, 2020, London, UK

scaling constants on the data transfer volume, and thus also provides a parametric upper bound on the achievable operational intensity for any possible legal schedule, for regular (affine) programs on a system with a two-level memory hierarchy. IOLB can be viewed as a proof environment, where the input is a C program meeting specific restrictions, and the output is an *I/O* lower bound for this program for any valid schedule of operations. The formal proof itself can be derived, understood, and reviewed from the output of IOLB. The lower bound is parametric, therefore supporting parametric problem sizes as are typically used in loop bound expressions in the input program. This paper describes:

1. The first static analysis for automatic derivation of non-asymptotic I/O lower bounds for affine programs.
2. A complete automated implementation of the IOLB framework, making it accessible to algorithm developers, producing clear parametric formula for the minimal I/O requirement given an input affine program.
3. An extensive evaluation of IOLB on 30 algorithms specified in PolyBench [23], with several first-time I/O lower bounds demonstrations on these algorithms.

The paper is organized as follows. A high-level overview of the approach is presented in Sec. 2. The formalism for data movement lower bounds based on the seminal red-blue pebble game of Hong & Kung [18], along with the core definitions and theorems used to derive our algorithm, are described in Sec. 3. Sec. 4 provides insights on how complex programs can be decomposed to derive tighter bounds. An overview of the complete framework is provided in Sec. 7. It uses two proof techniques, namely the K-partition and the wavefront based proofs that are respectively described in Sec. 5 and Sec. 6. We demonstrate the power of our approach by running it on a full benchmark suite of affine programs: Sec. 8 reports the data movement complexities for the 30 algorithms benchmarked in PolyBench. Related work is discussed in Sec. 9 before concluding.

2 Key Concepts and Overview of Approach

Performance tools such as Intel's Software Development Emulator Toolkit (SDE) and VTune Amplifier (VTune) enable the automated measurement of the achieved operational intensity of a program, that is the number of memory movements (e.g., cache miss) per actual operation executed. This ratio suggests whether a computation is memory-bound or compute-bound for a particular machine, e.g. using the roofline model [35]. But this measures a *particular implementation* of an algorithm: for example such system is used to fine-tune the particular tile size to be used to obtain maximal performance [24]. It cannot provide information on the *minimal number of movements required by any implementation of the algorithm*, and therefore does not bound the achievable operational intensity. We call two codes implementations of the same algorithm if they perform the same atomic operations with potentially different schedules (tiled vs. untiled LU factorization are implementations of the *same* algorithm, while LU with or without pivoting are not).

When facing subpar performance, the designer is left wondering whether the implementation is at fault, and should be better tuned; or whether the implementation is already "optimal", and the performance is bound by a fundamental limit of the implemented algorithm. A simple illustration is matrix-multiplication on dense matrices: a simple (i, j, k) untiled implementation will be memory-bound on most machines, but a carefully tiled one will become compute-bound [35]. One can assess the I/O optimality of an implementation if a (tight) non-asymptotic lower bound on I/O is known. IOLB specializes in automatically computing such non-asymptotic bounds. It enables algorithm designers to reason on the fundamental I/O limits of different algorithmic choices, and enables practitioners to reason on the I/O optimality of their implementation. IOLB works within a two-level memory model: the *I/O* cost of an algorithm is the number of transfers from the slow memory to the fast memory (see Sec. 3.1).

Affine programs. To make automation feasible and producing accurate-enough I/O bounds, we specifically focus in this work on affine (or, polyhedral) programs[14–16] as input to IOLB. This class covers a wide set of key algorithms, as exemplified with the 30 algorithms in PolyBench/C [23] that span popular dense linear algebra, stencils/convolutions, and dynamic programming techniques. Programs are restricted to control-flow that is statically analyzable, where loops and array-based accesses are expressed as affine functions of the (surrounding) loop iterators, and program parameters (i.e., constants unknown at compile-time).

Automating I/O lower bounds computation. We employ two very distinct approaches to finding lower bounds on data movement: one based on the S-Partitioning approach [18], and one based on graph wavefronts [12]. Combining these two approaches is essential for handling of a large class of programs, as they are complementary and work on different data dependence patterns. A lower bound for a program exhibiting a combination of both kind of patterns can combine both. We first present a high-level overview of the S-Partitioning approach, to intuitively familiarize the reader with the reasoning and terminology used.

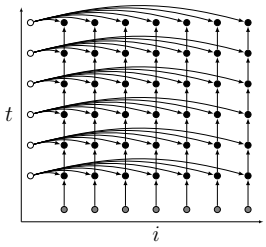
Consider the program in Fig. 1a. For given values of the parameters M and N (e.g., M=6, N=7), the program can be abstracted as a graph called a computational directed acyclic graph (CDAG, cf. Definition 3.1), as shown in Fig. 1c. Vertices in the CDAG represent input values for the computation as well as values computed by all statement instances (the latter are colored black and the former have lighter shades, grey or white). The set of vertices is also called the *iteration space* of the program. Edges in the CDAG capture data flow dependences, that is, relations between producers of data values to

```
Parameters: N, M;
Input: A[N], C[M]; Output: A[N];
for(t=0;t<M;t++)
  for (i=0; i<N; i++)
    A[i] = A[i] * C[t];
```

(a)

```
Parameters: N, M;
Input: A[N], C[M]; Output: S_{M-1}[N];
for (0≤t<M and 0≤i<N)
  if (t==0): S_{0,i} = A[i] * C[0];
  else: S_{t,i} = S_{t-1,i} * C[t];
```

(b)



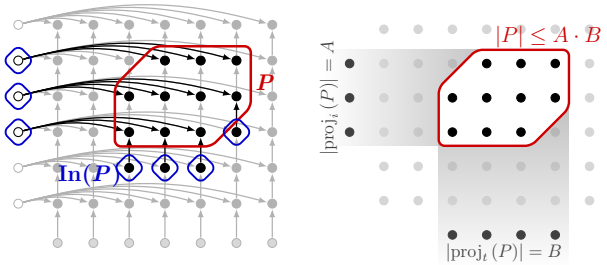
(c)

Figure 1. Example 1. **(a)** C-like code. **(b)** Corresponding single assignment form. **(c)** Corresponding CDAG. Input nodes `A[N]` (resp. `C[N]`) are in grey (resp. white), compute nodes are in black.

consumers. We note that in this abstracted representation of the computation, there is no association of any memory locations with values. Fig. 1b shows a single-assignment form of the same computation as that in Fig. 1a, and both programs have the same CDAG shown on Fig. 1c. The CDAG abstracts all possible valid schedules of execution of the statement instances: the only requirement is that all predecessor vertices in the CDAG must be executed before a given vertex can be executed. Data movement is modeled in a simplified two-level memory hierarchy, with an explicitly controlled fast memory of limited size S (e.g., a set of registers or a scratch-pad), and a slow memory of unlimited capacity. At any point in the execution at most S values corresponding to CDAG vertices may be in fast memory. A computational CDAG vertex can be executed only if the values corresponding to all predecessor vertices are present in fast memory.

The main idea of the S-partitioning approach for proving lower bounds can be understood as follows. Consider any valid schedule for the execution of the vertices of a CDAG, expressed as a sequence of instructions: load, store, or operation execution (Op). A valid schedule must ensure that values corresponding to predecessor vertices are available in fast memory when the operation corresponding to each CDAG vertex is executed. The sequence of instructions of the schedule is partitioned into contiguous maximal sub-sequences such that the total number of load instructions in any sub-sequence (except the last one) is exactly equal to a specified limit T (whose value will be chosen later in the reasoning). Let us suppose (as explained shortly) that no more than U Ops can be provably present within any of the partitioned sub-sequences. Let V denote all computational vertices in the CDAG. There must be at least $\lfloor |V|/U \rfloor$ sub-sequences with T loads, leading to a lower bound on the number of loads of $Q_{\text{low}} = T \cdot \lfloor |V|/U \rfloor$.

We next use the simple example of Fig. 1c to explain how an upper bound for U can be computed. The automated analysis based on partitioning in IOLB is centered around the use of geometric inequalities that relate the cardinality of a set of points in a multi-dimensional space to cardinalities of lower-dimensional projections of those points. The set of points here (P) are the computational vertices (Ops) in one of the partitioned sub-sequences (SS) with T load instructions. The In-set $\text{In}(P)$ of P is the set of all predecessors of the vertices in P that do not belong to P. Clearly, $\text{In}(P)$ represents values that were not computed in the current sub-sequence SS containing P. Since all values in $\text{In}(P)$ must be in fast memory in order to execute the Ops corresponding to P, they must either have already been in fast memory at the beginning of the sub-sequence SS or must have been explicitly loaded within SS. No more than S values from $\text{In}(P)$ could have been present at the beginning of SS, and T values were loaded in SS. Thus the size of $\text{In}(P)$ must be less than $(S+T)$.



(a) (b)

Figure 2. In-set, projections and geometric inequality

In our simple example, vertices corresponding to the loop statement are naturally represented as points in a two-dimensional lattice. With that representation, it may be observed that the size of the In-set of a vertex set of this particular graph must be greater than or equal to the cardinality of the orthogonal projections of P onto the vertical and horizontal axes (i.e. the height and width of P). As illustrated on Fig. 2b, the size of the vertex set in the two-dimensional space is bounded by the product of the sizes of its two 1D projections. This result can be generalized to arbitrary dimensions and any set of (not necessarily orthogonal) projections, and is called the Brascamp-Lieb inequality. Setting $T = S$, a vertex set with an In-set of size at most $2S$ cannot have projections of size more than $2S$, and therefore cannot itself be

greater than $U = 4S^2$. This implies that any valid ordering of the operations for this computation will result in at least $S \cdot \lfloor MN/4S^2 \rfloor \approx MN/4S$ load operations[1].

Lower bounds: algorithms vs. problems. The reader may be familiar with the external memory model [2], or cache-oblivious algorithms [17]. The memory model is very similar to the one we use in this paper, except that the granularity of memory transfers to and from fast memory is that of a block of several words (typically hundreds), instead of individual words. In these models, researchers are interested in designing algorithms that minimize memory transfers, either for a fixed fast memory size (external memory algorithms, or cache-aware algorithms) or for any size (cache-oblivious algorithms), as well as proving theoretical lower bounds on the number of such memory transfers.

The fundamental difference with the work presented here is that lower bounds in these models [2, 17] are *over all possible algorithms solving a certain problem*, while the lower bounds provided by IOLB are *over all possible valid schedules for a specific algorithm* (i.e. a fixed set of partially ordered operations). As an example there exists many algorithms for performing matrix-matrix multiplication. The lower bound provided by IOLB gives information on what could possibly be achieved by rescheduling the operations of the usual $O(N^3)$ algorithm, but gives no information on other algorithms such as Strassen's [31].

The goal of IOLB is to provide information on whether *a given implementation of an algorithm* might be improved with respect to data movement or if it is fundamentally *I/O*-bound.

Overview of our contributions. To automate and generalize this geometric reasoning on arbitrary affine programs, we need to: (1) Generalize the geometric upper-bounding for any number of projections with arbitrary dimensionality (Sec. 3.3); (2) Build (derive from array accesses) a compact representation (DFG) of the data-flow dependencies of the program that is suitable for reasoning about reuse directions (Sec. 3.4); (3) Analyze this representation to extract reuse directions (represented as DFG-paths – Sec. 3.4); (4) Generalize the geometric reasoning for a perfectly nested loop with one statement to any combination of loops with arbitrary number of statements (embedding – Sec. 5).

The goal of IOLB is to go even further and automatically derive parametric bounds that are as tight as possible (including maximization of the scaling constants). For this purpose, the developed algorithm: (5) Enables the combination (and tightening) of constraints associated with different projections, even with an arbitrary number of them with lower dimensionality; (6) Handles non-orthogonal projections even if they are not linearly independent; (7) Develops a new reasoning strategy inspired from the wavefront reasoning of Elango et al. [12] (Sec. 6); (8) Allows the combination of individual complexities (obtained through potentially different methods) of overlapping program regions (Def. 4.1, Lemma 4.2) even for an unbounded number of regions (parameterized regions inside loops – Sec. 4.3);

[1]It is actually possible to improve this bound by a factor of 4 with more advanced techniques, as shown in Sec. 5

3 Foundations

In this section, we present some background and discuss prior results needed for the developments in this paper.

3.1 CDAG and *I/O* Complexity

The formalism and methodology we use to derive schedule-independent data movement lower bounds for execution of an algorithm on a processor with a two-level memory hierarchy is strongly inspired by the foundational work of Hong & Kung [18]. In this formalism, an algorithm is abstracted by a graph — called a CDAG —, where vertices model execution instances of arithmetic operations and edges model data dependencies among the operations. We formalize the data movement (or *I/O*) complexity of a CDAG via the red-white pebble game (a variation we designed of Hong & Kung's red-blue pebble game). In this game, a vertex of a CDAG can hold red and white pebbles. Red pebbles represent values in the fast memory (typically a cache or scratchpad), and their total number is limited. White pebbles represent computed values, that can be loaded into the fast memory. A value can be computed only when all its operands reside in the fast memory: a red pebble can be placed on a vertex in the CDAG if all its predecessors hold a red pebble, a white pebble is placed alongside the red. Values that have been computed can be loaded in and discarded from the fast memory at any time: a red pebble can be placed or removed from a vertex holding a white pebble. However a value can only be computed once: once a vertex holds a white pebble, it cannot be removed. The *I/O* cost of an execution of the game is the number of loads into the fast memory: the number of times a red pebble is placed alongside a white one.

Contrary to Hong & Kung's original model, our formalism *does not allow recomputation* of the value at a vertex. This follows many previous efforts [4, 5, 8, 9, 11–13, 19, 28]. This assumption is necessary to be able to derive bounds for complex CDAGs by decomposing them into subregions. Another slight difference of IOLB with prior work is that it only models *loads and not stores* — this means the generated bounds are clearly also valid lower bounds for a model that counts both loads and stores. Since the number of loads dominates stores for most computations, the tightness of the lower bounds is not significantly affected. We provide formal definitions below.

Definition 3.1 (Computational Directed Acyclic Graph)**.** A *Computational Directed Acyclic Graph (CDAG)* is a tuple $G = (V, E, I)$ of finite sets such that (V, E) is a directed acyclic

graph, $I \subseteq V$ is called the *input set* and every $v \in I$ has no incoming edges.

Definition 3.2 (Red-White Pebble Game)**.** Given a CDAG $G=(V,E,I)$, we define a complete S-red-white pebble game (S-RW game for short) as follows: In the initial state, there is a white pebble on every input vertex $v \in I$, S red pebbles and an unlimited number of white pebbles. Starting from this state, a complete game is a sequence of steps using the following rules, resulting in a final state with white pebbles on every vertex.

(R1) A red pebble may be placed on any vertex that has a white pebble.

(R2) If a vertex v does not have a white pebble and all its immediate predecessors have red pebbles on them, a red pebble may be placed on v. A white pebble is placed alongside the red pebble.

(R3) A red pebble may be removed from any vertex.

The *cost* of a S-RW game is the number of applications of rule (R1), corresponding to the number of transfers from slow to fast memory.

Here, red pebbles mark operations whose results are currently stored in fast memory, and white pebbles mark operations whose results have been computed. A result resides in fast memory immediately after it has been computed, and we consider that it is always present in slow memory as well (since stores are not taken into account). Computation can happen at most once due to rule ($R2$), and this is the fundamental difference with Hung & Kung's model.

Definition 3.3 (*I/O* complexity)**.** The *I/O* (or *data movement*) *complexity* of a CDAG G for a fast memory capacity S, denoted $Q(G)$, is the minimum cost of a complete S-RW game on G.

This quantity is the fundamental measure for which this work tries to establish lower bounds.

3.2 Partitioning

One key idea from Hong & Kung was the design of a mapping between any valid sequence of moves in the red-blue pebble game and a partition of the vertices of a CDAG and thereby the assertion of an *I/O* lower bound for any valid schedule in terms of the minimum possible count of the disjoint vertex-sets in any valid $2S$-partition (see below) of the CDAG.

The argument is the following: any execution can be decomposed into consecutive segments doing exactly (but for the last one) S loads. There are at most S vertices in fast memory before the start of each segment. Considering the set of computed vertices in one of these segments, we can bound the size of its "frontier" (or *In-set*) by $2S$: there can be at most S vertices in fast memory before the execution of the segment, and by construction there are exactly S loads.

Smith et al. [30] introduced a generalization of this argument, leading to tighter bounds in many cases. The idea is to decompose the execution into segments with T (not necessarily tied to be equal to S) loads . This leads to a $(S+T)$-partitioning lemma instead of the original $2S$.

Definition 3.4 (In-set, K-bounded set, K-partition)**.** Let $G=(V,E)$ be a CDAG, $P \subseteq V$ be a vertex set in G.

The *In-set* of P is the set of vertices outside P with a successor inside P.

A vertex set $P \subseteq V$ is called *K-bounded* if $\mathrm{In}(P) \leq K$.

A *K-partition* of G is a partition of V into subsets with no cyclic dependencies, such that every subset has an In-set of size $\leq K$.

Lemma 3.5 ($(S+T)$-Partitioning *I/O* lower bound, no input case [12])**.** *Let S be the capacity of the fast memory, let $G=(V,E,\emptyset)$ be a CDAG, and let h be the minimum number of subsets in a $(S+T)$-partition of $G_I=(V,E,I=\mathrm{Sources}(V))$ for some $T>0$. Then, the minimum I/O for G satisfies:*

$$Q(G) \geq T \cdot (h-1) - |\mathrm{Sources}(V)|.$$

where $\mathrm{Sources}(V)$ *is the set of vertices with no predecessors in G.*

3.3 Using Projection to Bound the Cardinality of K-Bounded Sets

The key idea behind the automation of I/O lower bound computation is the use of geometric inequalities through an appropriate program representation. Vertices of a CDAG are mapped to points in a multidimensional geometric space $\mathcal{E} \simeq \mathbb{Z}^d$ through some mapping ρ (where dimensions are typically loop indices), and regular data dependencies in the CDAG are represented as projections on a lower-dimensional space.

The condition "set of vertices $P \subset V$ is K-bounded" in the CDAG corresponds to a condition of the form "the size of the projections of $\rho(P)$ in $\mathcal{E}$ is bounded by K". Finding a bound on the size of a K-bounded set in a CDAG can thus be reduced to: finding a bound on the size of a set E in a geometric space, given cardinality bounds on some of its projections. This correspondence is developed in Sec. 5.

The mathematical result we use is a discrete version of the Brascamp-Lieb inequality, introduced by Christ et al. [10].

Intuitively, this inequality is a generalization of the following, 3-dimensional continuous one:

If the surfaces of all three projections of a 3-dimensional volume V on planes $x=0$, $y=0$, $z=0$ are bounded by some constant C, then $|V| \leq C^{3/2}$.

This result can be generalized to arbitrary dimensions (giving $|V| \leq C^{d/(d-1)}$), and further to any set of projections, even on lower-dimensional subspaces.

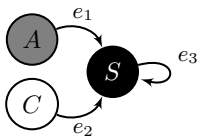
More details are provided in the full technical report [22].

3.4 A Compact Representation of the CDAG: the Data-flow Graph

A CDAG (see Fig. 1c) represents a single dynamic execution of a program, and can be very large. To be able to analyze programs of realistic size with reasonable resources, we use

```
for (0≤t<M and 0≤i<N)
  if (t==0): S[0,i]=A[i]*C[0];
  else: S[t,i]=S[t-1,i]*C[t];
```

(a) Single assignment form



(b) DFG

$D_A = [N] \to \{A[i] : 0 \le i < N\}$
$D_C = [N] \to \{C[t] : 0 \le t < M\}$
$D_S = [M,N] \to \{S[t,i] : 0 \le t < M \wedge 0 \le i < N\}$
$|D_S| = MN$

(c) Node domains

$R_{e_1} = [N] \to \{A[i] \to S[0,i] : 1 \le i < N\}$
$R_{e_2} = [M,N] \to \{C[t] \to S[t,i] : 0 \le t < M \wedge 0 \le i < N\}$
$R_{e_3} = [M,N] \to \{S[t,i] \to S[t+1,i] : 0 \le t < M-1 \wedge 0 \le i < N\}$

(d) Edge relations

Figure 3. DFG for Example 1

a compressed representation called a Data-flow graph (DFG). Another advantage of such a representation is that it is *parametric*, i.e. a single DFG can represent CDAGs of different sizes, depending on program parameters. A DFG represents an *affine* computation, which is the class of programs that can be handled by the *polyhedral model* [14–16]. We use the terminology and syntax from the ISL library [32], and illustrate them with the example of Fig. 1. Formal definitions can be found in the manual [33].

Vertex domains. As one can see on Fig. 1c, to each loop is associated a "geometric" space dimension (t and i here) so that each *vertex of the CDAG* lives in a multidimensional iteration space, its *domain*, that can be algebraically represented as a union of parametric $\mathbb{Z}$-polyhedra (sets of integer points in a multidimensional space bounded by affine inequalities). A domain is an *set* (in ISL terms) for which standard operations (union, intersection, difference,…) are available, as well as a *cardinality* operation (denoted $|D|$). As an example (see Fig. 3c), the *domain* D_S of statement S is a $\mathbb{Z}$-polyhedron with parameters M and N made up of all integer points (t,i) such that $0 \le t < M$ and $0 \le i < N$. The number of points in this set (cardinality) is $|D_S| = MN$. Note that the space within which all the points of a statement (S here) live is identified with the name of the statement, using the notation $S[t,i]$.

Edge relations. A set of *edges* of the CDAG is represented using a *relation* (*map* in ISL), which is a set of pairs between two spaces, from the *domain* space to the *image* space. As an example (see Fig. 3d), the data flow from statement $S[t,i]$ (definition of $A[i]$ in S) to statement $S[t+1,i]$ (use of $A[i]$ in S) is represented using the relation R_{e_3}. In addition to standard set operations, ISL can compute the transitive closure of a relation, denoted R^* (this will be needed is Sec. 6). Binary relations are also supported: image of a domain D through a relation R (denoted $R(D)$), and composition of two relations R_1 and R_2, denoted $R_1 \circ R_2$ (this is left composition, going the opposite way from usual functional notation). Composition restricts the image domain of the resulting relation to points where the composition relation makes sense: $\mathrm{Dom}(R_1 \circ R_2) = R_1^{-1}(\mathrm{Im}(R_1) \cap \mathrm{Dom}(R_2))$, $\mathrm{Im}(R_1 \circ R_2) = R_2(\mathrm{Im}(R_1) \cap \mathrm{Dom}(R_2))$. As with domains, we will sometimes manipulate unions of such relations.

Data-flow graph (DFG). A DFG is a graph $G = (\mathcal{S}, \mathcal{D})$. Each vertex $S \in \mathcal{S}$ of the graph represents a (static) statement or an input array of the program. Each vertex S is associated with a parametric iteration domain D_S and a list of enclosing loops (empty for input arrays). Each edge $d = (S_a, S_b) \in \mathcal{D}$ represents a flow dependency between statements or input arrays. Each edge is associated with an affine relation R_d between the coordinates of the source and sink vertices. The DFG is a compact (exact) representation of the dynamic CDAG where a single vertex/edge of the DFG represents several vertices/edges of the dynamic CDAG. While all the reasoning and proofs can be done by visualizing a CDAG, the actual heuristic described in this paper manipulates its compact representation, allowing to translate graph methods [13] into geometric reasoning. Fig. 3b shows the DFG for our simple stencil code.

DFG-paths. A fundamental object in our lower bound analysis is a DFG-path, which is simply a directed path in a DFG. The relation R_p of a DFG-path $p = (e_1, \ldots, e_k)$ is the composition of the relations of its edges: $R_p = R_{e_1} \circ \cdots \circ R_{e_k}$. We are only interested in two specific types of DFG-paths, depending on their relation:

- *chain circuits*, which are cycles from one DFG-vertex S to itself, such that the path relation R_p is a translation $S[\vec{x}] \to S[\vec{x} + \vec{b}]$.
- *broadcast S_a, S_b-paths*, which are elementary paths (from a S_a to S_b – S_b possibly equal to S_a) in which all DFG-edges but the first one are injective edges, such that the inverse of the corresponding relation R_p is an affine function $S_b[\vec{x}] \to S_a[A \cdot \vec{x} + \vec{b}]$, where A is not full-rank.

Intuitively, a chain circuit corresponds in the CDAG to "iterative" dependencies, for instance every statement $S_{i,j}$ in a 2-dimensional loop depending on the result of statement $S_{i-1,j}$. Broadcast paths correspond to a same data being reused multiple times, for instance a variable x being used by every statement S_i in a one-dimensional loop. The dimension of the kernel of A in the definition above corresponds to the dimension of the set of statements that use a single piece of data: it is of dimension d if it is used in every iteration of a d-dimensional loop. In both cases, these are regular data reuse patterns that can be exploited by our geometric approach.

In Fig. 3, path $p = (e_3)$ is a chain circuit, going from S to itself with translation vector $\vec{b} = (1,0)$. Path $p' = (e_2)$ is

a broadcast path, with relation $R_{p'} = R_{e_2} = \{C[t] \rightarrow S[t,i] : 0 \leq t < M \ \wedge \ 0 \leq i < N\}$. The inverse relation is the linear function $\vec{I} \mapsto A \cdot \vec{I} + \vec{b}$, with $A = (1\ 0), \vec{b} = (0)$. The kernel of A is $\{(0,i), i \in \mathbb{R}\}$.

4 CDAG Decomposition

To derive data movement lower bounds for a complex program, it is essential to be able to decompose it into subregions for which we can compute lower bounds, and then sum the complexity for each subregion. The *no recomputation* condition (see Sec. 3.1) is necessary for such a decomposition. Under this hypothesis, it is quite straightforward to see that a decomposition into disjoint subregions is sufficient. In this section, we provide a more general decomposition lemma, using the fact that vertices of a subregion that will not be counted as loads can also be part of another subregion. We then explain how it is applied on the DFG representation, distinguishing two cases: combining a fixed number of program regions (see example in Fig. 5); and summing over all iterations of a loop (see example in Fig. 4), which amounts to combining an unbounded (parametric) number of program regions. We stress that the CDAG partitioning method (Sections 3.2 and 3.3) and the CDAG decomposition method (this section) are two distinct things, used at different stages in the global algorithm.

4.1 Non-disjoint Decomposition Lemma

Definition 4.1 (sub-CDAG, no-spill set)**.** Let $G = (V,E,I)$ be a CDAG, and $V_i \subset V$. The sub-CDAG $G_{|V_i}$ of G is the CDAG with vertices V_i, edges $E_i = E \cap (V_i \times V_i)$ and input vertices $I_i = I \cap V_i$.

The *no-spill set* of $G_{|V_i}$ is the subset of vertices of $V_i \setminus I_i$ with either:

1. no outgoing edges in E_i, or
2. no incoming edges in E_i and at most one outgoing edge in E_i

The *may-spill set* of $G_{|V_i}$ is the complement of its no-spill set in V_i.

Lemma 4.2 (CDAG decomposition)**.** *Let $G = (V,E,I)$ be a CDAG. Let $V_1, V_2, \ldots, V_k$ be subsets of V such that for any $i \neq j$, the may-spill sets of $G_{|V_i}$ and $G_{|V_j}$ are disjoint.*

Then, the I/O complexity of G is bounded by the I/O complexities of the sub-CDAGs $G_{|V_i}$:

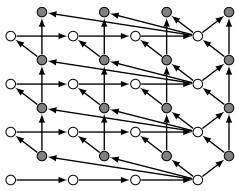
$$Q(G) \geq \sum_{i=1}^{k} Q(G_{|V_i}).$$

To prove this lemma, it suffices to show that it is possible to build a RW-game for G from RW-games for subgraphs $G_{|V_i}$, and that this game is valid and has a cost greater than the sum of the individual games. This poses no great difficulty, and the full proof can be found in the technical report [22].

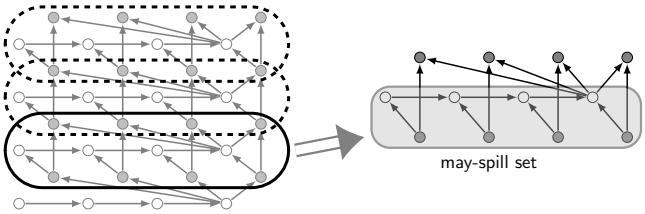
IOLB implements two different mechanisms that make use of the non-disjoint decomposition lemma. The basic one

```
  for(t=0; t<M; t++) {
      s = 0;
      for(i=0; i<N; i++)
S1:       s += A[j];
      for(i=0; i<N; i++)
S2:       A[j] += s;
  }
```

(a) Code



(b) CDAG for M=4, N=4. White vertices correspond to S1, gray vertices to S2.



(c) Decomposition of the CDAG

Figure 4. Example 2

(bounded combination – Sec. 4.2) simply decomposes the CDAG into a bounded number of sub-CDAGs (e.g. corresponding to different sub-regions of the code), computes the corresponding *I/O* complexities, and combines them. The more complex one (loop parametrization – Sec. 4.3), decomposes the CDAG into an unbounded number of sub-CDAGs by "slicing" the iteration space of a loop nest. IOLB combines the two mechanisms. The following example illustrates the decomposition lemma for loop parametrization.

Illustrating example. Consider Example 2 in Fig. 4. The CDAG can be decomposed into $M-1$ identical subgraphs, as shown on Fig. 4c (each subgraph $G_{|V_t}, t = 1,\ldots,M-1$ corresponds to iteration t of the loop around S_1, and iterations $t-1$ and t of the loop around S_2). On each of these subgraphs, the may-spill set contains the two "bottom" rows (because vertices in the "top" row have no successor in the sub-CDAG). Thus the may-spill sets of these subgraphs are pairwise disjoint and the *I/O* for the whole CDAG is greater than the sum of the individual *I/O* for each subgraph by Lemma 4.2.

On each subgraph $G_{|V_t}$, the wavefront method (Sec. 6) can be applied, giving a lower bound on *I/O* of $Q(G_{|V_t}) \geq N - S$. As the may-spill set of the different subgraphs do not intersect, the individual complexities can be summed over $t = 1,\ldots,M-1$, providing a lower bound for the whole CDAG:

$$Q(G) \geq (M-1)(N-S).$$

4.2 Bounded Combination

The main procedure of IOLB (Sec. 7.3) selects a bounded set of (possibly overlapping) sub-CDAGs and computes their individual complexities. The objective of this procedure is

to combine (sum) as many non-interfering (disjoint may-spill sets) complexities as possible. It does so using a greedy approach: Assume there are two sub-CDAGs both with a "high" complexity but with non-disjoint may-spill sets. The procedure will select the one with the highest complexity, recompute the complexity of the second after removing the intersecting part, and then sum them up. The overall set of sub-CDAGs is iteratively processed this way (and the complexities summed-up) until empty or negligible complexities remain. The comparison (what is "higher") is done using *instances of parameter values*, simply evaluating the corresponding symbolic expressions. It should be emphasized that the final bound is a valid lower bound for *any* parameter values, the instances of parameter values are only used for heuristics.

Let us have a look at the example on Fig. 5. In the original code (5a), notice that k is the outer loop index, meaning that A[k] will have been modified either in the current loop iteration or the previous one depending on the order between i and k (Floyd-Warshall exhibits the same pattern, with three loops instead of two). This is made clear in the single-assignment form (5b), and can be visualized in the CDAG representation (5d). The dependences on input values are grayed out in (5b) and omitted in (5d), and we will ignore them in the discussion to keep the explanations simple.

Considering only the statement vertex S in the DFG, the dependency analysis gives the following relations:

$$R_1 = \{S[k-1,i] \rightarrow S[k,i] : \ 1 \leq k < N \ \wedge \ 0 \leq i < N\}$$
$$R_2 = \{S[k-1,k] \rightarrow S[k,i] : \ 1 \leq k < N \ \wedge \ 0 \leq i < k\}$$
$$R_3 = \{S[k,k] \ \rightarrow \ S[k,i] : \ 0 \leq k < N \ \wedge \ k < i < N\}$$

The image domains of R_2 and R_3 provide a natural decomposition of the CDAG into two non-interfering sub-CDAGs, as shown in (5e). On each part, the pattern is similar to that of Example 1 on page 3 the geometric approach gives a lower bound (omitting lower order terms) $Q(G_i) \geq \frac{N^2}{2S}$. Since they do not interfere, the procedure will return their sum $Q(G) \geq \frac{N^2}{S}$, independently of the parameter instance.

4.3 Loop Parametrization

As done on the example above, IOLB can compute the *I/O* complexity of some inner loop nests of a bigger enclosing loop nest and sum them. To this end, our scheme performs what we call *loop parameterization*. Loop parameterization considers each individual sub-CDAGs where the outermost indices are fixed (our algebraic formulation obviously allows to consider such indices as parameters without the need to explicitly enumerate them) enriched by their input vertices. This is formalized in the full technical report [22].

5 K-Partition Bound Derivation

In this section, we explain how to apply the geometric reasoning of Sec. 3.3 on a CDAG $G = (V,E)$, using its compact representation as a DFG. To apply Lemma 3.5 on G, we need to find a lower bound on the minimum number of subsets h in any K-partition of G. The general reasoning is as follows: First, we embed V in a geometric space through a map $\rho : V \rightarrow \mathbb{Z}^d$, such that two disjoint subsets of V are mapped to disjoint subsets of $\mathbb{Z}^d$, and $|\rho(P)| \leq |P|$.

Second, we use the DFG representation to find a subset $V' \subseteq V$ and a set of projections (group homomorphisms) $\phi_1,...,\phi_m$ with the property that:

$$\text{Any } K\text{-bounded set } P \subseteq V' \setminus \text{Sources}(V') \text{ satisfies } \left|\phi_j(\rho(P))\right| \leq K. \tag{1}$$

To do so, we primarily use broadcast and chain circuit structures (cf. Sec. 3.4). These are two special cases that are easy to detect from the DFG, common in applications, and convenient from the stand point of (1). Third, using the Brascamp-Lieb inequality [10], we derive an upper bound U on $|\rho(P)|$ for any $(S+T)$-bounded P. This provides a lower bound $\left\lceil \frac{|V' \setminus \text{Sources}(V)|}{U} \right\rceil$ on the number h of disjoint $(S+T)$-bounded sets in $V' \setminus \text{Sources}(V')$.

DFG-paths and projections. Let S_k be some fixed DFG-vertex (corresponding to one program statement).

Let $Q_1,...,Q_m$ be DFG-paths all ending in S_k.

Definition 5.1 (embedded projections)**.** For a given path Q, the associated projection ϕ_Q is defined as:

- $\phi_Q(i_1,\ldots,i_d) = (j_1,\ldots,j_{d'})$, for a broadcast path with relation $R_Q = \{S_j[j_1,...,j_{d'}] \rightarrow S_k[i_1,...,i_d] : ...\}$,
- the orthogonal projection on the hyperplane in $\mathbb{Z}^d$ defined by vector $\delta = (\delta_1,...,\delta_d)$, for a chain circuit with $R_Q = \{S_k[i_1,...,i_d] \rightarrow S_k[i_1+\delta_1,...,i_d+\delta_d] : ...\}$.

It is straightforward to check that ϕ_Q satisfies (1) in both cases.

Summing projections. In some cases, the parts of the In-set of a vertex set associated with two given path relations are actually disjoint. Let Q_1 and Q_2 be two such paths, such that $R_{Q_1}^{-1}(P) \cap R_{Q_2}^{-1}(P) = \emptyset$ for any $P \subseteq V \setminus \text{Sources}(V)$. If these are two broadcast paths, then since $R_{Q_i}^{-1}(P) \subset \text{In}(P)$, any K-bounded set P satisfies the stronger inequality:

$$\left|\phi_{Q_1}(\rho(P))\right| + \left|\phi_{Q_2}(\rho(P))\right| \leq K$$

The same holds if Q_1 is a chain circuit and $R_{Q_1}^{-1}(P) \cap R_{Q_2}^{-1}(P) = \emptyset$, by a similar argument. When this is the case, we say the two paths are *independent*.

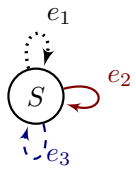
Example. Consider paths $p_1 = (e_2)$ and $p_2 = (e_3)$ in Fig. 3. p_1 is a broadcast path with relation $\{C[t] \rightarrow S[t,i]\}$, so the corresponding projection is $\phi_1(t,i) = (t)$. p_2 is a chain path with relation $\{S[t,i] \rightarrow S[t+1,i]\}$, so the corresponding projection is $\phi_2(t,i) = \text{proj}_{(1,0)}(t,i) = (0,i)$ (see Fig. 2). It is straightforward to check that paths p_1 and p_2 are independent, so a K-bounded set P actually satisfies $|\phi_1(P)| + |\phi_2(P)| \leq K$.

```
Parameters: N;
Input: A[N]; Output: A[N];
for(k=0;k<N;k++)
  for (i=0; i<N; i++)
    A[i] = f(A[i],A[k]);
```

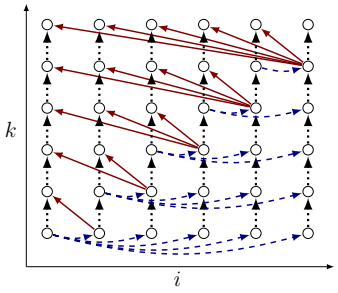
(a) C-like code

```
Parameters: N;
Input: A[N]; Output: S_{N-1}[N];
for (0≤k<N and 0≤i<N)
  if (k==i==0): S_{0,i} = f(A[0],A[0]);
  else if (k==0): S_{0,i} = f(A[i],S_{0,0});
  else if (i<=k): S_{k,i} = f(S_{k-1,i}, S_{k-1,k});
  else if (i>k): S_{k,i} = f(S_{k-1,i}, S_{k,k});
```

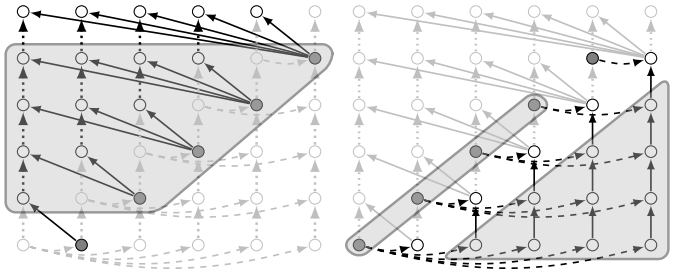
(b) Corresponding single assignment form



(c) DFG



(d) CDAG for N=5. Dotted, plain and dashed edges respectively correspond to DFG-edges e_1,e_2 and e_3.



(e) Decomposition into two non-interfering sub-CDAGs. Sources are in gray. May-spill sets are encircled.

Figure 5. Example 3

Here the geometric inequality gives:

$$|P| \leq |\phi_1(P)||\phi_2(P)|.$$

Setting $a=|\phi_1(P)|$,$b=|\phi_2(P)|$, the following optimization problem gives a bound on $|P|$:

$$\begin{aligned} &\text{Minimize} && ab \\ &\text{such that} && a+b \leq (S+T) \end{aligned}$$

This is minimal for $a=b=(S+T)/2$, giving

$$|P| \leq ((S+T)/2)^2.$$

We can then set $T=S$ (this is optimal for this case), getting:

$$|P| \leq S^2.$$

The iteration domain is of cardinality MN and the frontier is of size $N+M$, so in the end Lemma 3.5 gives:

$$Q \geq \left\lfloor \frac{MN}{S^2} \right\rfloor \times S - N - M.$$

This summing argument can be generalized to an arbitrary set of projections (see the full technical report [22]).

The whole procedure can be automated and applied to any parametrized DFG, with an arbitrary number of projection constraints of any dimensionality, that do not need to be orthogonal or even linearly independent. The full algorithm is provided in the full technical report [22].

6 Wavefront Bound Derivation

An alternative way to derive data movement lower bounds in the no-recomputation model is the *wavefront* abstraction introduced by Elango et al. in [12]. At any point in an execution of a RW-game, the wavefront is the set of vertices that have been computed but whose result is still needed by some successor (sometimes called the set of *live* vertices). If the size of the wavefront at some point in the execution is greater than the size of the fast memory, then vertices have to be spilled to the slow memory and thus loaded (using rule ($R1$)).

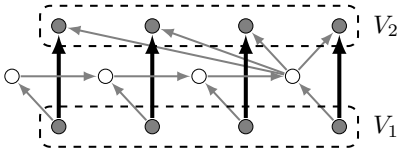


Figure 6. Application of the wavefront method.

There are many possible ways of finding a bound on the size of the minimum wavefront in a CDAG. In this work we use the following simple characterization, which is sufficient to obtain very strong bounds in cases like Example 2 (Fig. 5).

Let us start with an example: Fig 6 shows one of the subgraphs of Fig. 5e. In this CDAG, every vertex in the top row V_2 (indirectly) depends on every vertex in the bottom row V_1. Furthermore there is a one-to-one correspondence between V_1 and V_2 through bold edges. So all vertices in V_1 have to be computed before any vertex in V_2 can be computed, and every single vertex in V_1 is a direct dependency of a vertex in V_2. Thus, right before the computation of the first vertex in V_2, all vertices in V_1 belong to the wavefront, making the *I/O*:

$$Q \geq |V_1| - S = N - S.$$

This argument can be generalized to any CDAG exhibiting a similar structure, and this structure can be discovered in a parametric DFG using operations on polyhedral relations, in particular by computing the transitive closure of the dependence relation between two "layers" of the graph. A complete algorithm as well as proper formalizations are provided in the full technical report [22].

As in our example, the common case to use this technique to get a strong data movement lower bound is to combine it with the parametric CDAG decomposition (Sec. 4.3).

7 Complete Framework

7.1 DFG Construction

Our front end (PET [34]) takes as input a program in C where the to-be analyzed regions (SCoPs – Static Control Parts) are delimited by `#pragma scop` and `#pragma endscop` annotations. For PET, all array accesses are supposed not to alias with one another. Any scalar data is assumed to be atomic and all of the same size: our CDAG is not weighted (which is a limitation of our implementation and not a conceptual limitation of the approach). As illustrated by the example of Fig. 1 and 3 (multidimensional-)array accesses are affine expressions of static parameters and loop indices. A static parameter can be the result of any complex calculation but has to be a fixed value for the entire execution of the region. Loop bounds and more generally control tests follow the same rules (affine expressions). As a consequence, the iteration space is a union of (parametric) polyhedra, and memory accesses (read and writes) are piecewise affine functions. This representation of the region execution that fits into the polyhedral framework [16] allows to compute data dependencies using standard data-flow analyses.

PET outputs a polyhedral representation of the input C program, from which we extract a *Data-flow graph (DFG)* $G=(\mathcal{S},\mathcal{D})$ (see Sec. 3.4).

7.2 Instances of Parameter Values

As briefly explained in Sec. 4, to generate bounds that are as tight as possible, our heuristic needs to make decisions. Such decisions are based on our ability to compare the size of two different domains sizes or even the complexity of two different sub-CDAGs. The overall framework being parametric (it provides complexities that are functions of parameter values and cache size), a total order is obtained by considering a specific instance of parameter values, taken as an additional input alongside the C program. One needs to outline that a specific instance of parameter values is *not* considered by the algorithm as a precondition: for a given instance, the computed lower bound expression is universal i.e. is correct for *any* parameter values. For completeness, several instances are considered, and to each instance I is associated a complexity Q^I. As we have $Q \geq Q^I$ for any instance, denoting $\mathcal{I}$ the set of all considered instances, they are simply combined as: $Q = \max_{I\in\mathcal{I}}(Q^I)$.

7.3 Main Algorithm

Alg. 1 contains the skeleton of the main part of IOLB, with links to corresponding subsections.

```
1  function program_Q
     input  : Data-flow graph G=(S,D), an instance I
     output : lower bound Q_low
2    Q = ∅;
3    Let D be the max loop depth;
4    foreach loop level 0 ≤ d < D do                                   → Sec. 4.3
5      foreach S ∈ S surrounded by at least d+1 loops do
6        Ω_d := [I_1,...,I_d] → {S[i_1,...,i_D] : i_1=I_1 ∧ ... i_d=I_d};
7      Let G' be a copy of G;
8      while elapsedTime < timeout do
9        Let D_S be the parametrized domain of S in G';
10       P := ∅, L := ∅;
11       foreach P_i ∈ genpaths(G',S,Ω_d) do
12         if |D_S ∩ Dom(P_i)| ≥ γ·|D_S| then
13           K_i := Ker(P_i);
14           if L := subspace_closure(B,K_i)
               changed then
15             D_S := D_S ∩ Dom(P_i);
16             P := P ∪ P_i;                                           Sec. 5
17       if P = ∅ then exit while loop;
18       (Q,G') = combine_paramQ(Q,G',
                  sub_paramQ_bypartition(P,D_S,L,Ω_d));
19     (Q,G') = combine_paramQ(Q,G',                                   Sec. 6
                sub_paramQ_bywavefront(S,Ω_d));
20   Q_low = input_size(G) + max(0, combine_subQ(Q));                  Sec. 4.2
21 function combine_paramQ
     input  : set of global
              bounds Q, DFG G', parametrized bound Q(Ω)
     output : updated Q, G'
22   if [Ω ≠ Ω' ⇒ Q.interf(Ω) ∩ Q.interf(Ω') = ∅] then
23     Q := ∑_Ω Q(Ω);
24     Q.may-spill := ⋃_Ω Q.may-spill(Ω);
25   Q = Q ∪ {Q};
26   G' := G' \ Q.may-spill;
27
```

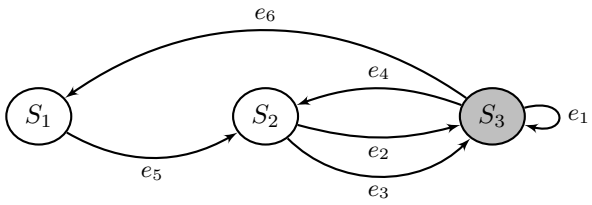
Algorithm 1: Main procedure that computes Q_{low} for the program by combining lower bound of sub-CDAGs obtained through K-partition or wavefront reasoning

To make it concrete, we first show a step-by-step execution of the algorithm on the `cholesky` kernel. The pseudo-code and associated DFG for `cholesky` are reported in Fig. 7.

In this example, the K-partition method is the method that yields the strongest bound. To keep things tractable, we will detail only the parts of the algorithm that contribute to this bound: the iteration of the outer loops (lines 4 and 5) for which $d=0$ and $S=S_3$, and only the K-partition part (lines 8 to 18, corresponding to Sec. 5). High-level insights of the

```
for(k = 0; k < n; k++)
   A[k][k] = sqrt(A[k][k]);              //S1
   for(i = k+1; i < n; i++)
      A[i][k] /= A[k][k];                //S2
   for(i = k+1; i < n; i++)
      for(j = k+1; j <= i; j++)
          A[i][j] -= A[i][k] * A[j][k]; //S3
```

(a) Source code



(b) DFG (input nodes are omitted)

$$R_{e_1} = \{S_3[k-1,i,j] \to S_3[k,i,j] : 1 \le k < N \ \wedge\ k+1 \le i < N \ \wedge\ k+1 \le j \le i\}$$
$$R_{e_2} = \{S_2[k,j] \to S_3[k,i,j] : 0 \le k < N \ \wedge\ k+1 \le i < N \ \wedge\ k+1 \le j \le i\}$$
$$R_{e_3} = \{S_2[k,i] \to S_3[k,i,j] : 0 \le k < N \ \wedge\ k+1 \le i < N \ \wedge\ k+1 \le j \le i\}$$
$$R_{e_4} = \{S_3[k-1,i,k] \to S_2[k,i] : 1 \le k < N \ \wedge\ k+1 \le i < N\}$$
$$R_{e_5} = \{S_1[k] \to S_2[k,i] : 0 \le k < N \ \wedge\ k+1 \le i < N\}$$
$$R_{e_6} = \{S_1[k-1,k,k] \to S_1[k] : 1 \le k < N \ \wedge\ k+1 \le i < N\}$$

(c) Edge relations

Figure 7. Cholesky decomposition

rest of the algorithm are provided at the end of this section, and more complete explanations are available in [22].

The DFG contains three statement vertices $\{S_1,S_2,S_3\}$ (the vertex corresponding to input array A and the corresponding dependences are omitted as they do not play a role in the lower bound derivation). The main loop of Alg. 1 iterates on those statements and computes some lower bound complexities for each of them.

Procedure genpaths (called at line 11 in Alg. 1) traverses the DFG, searching for chain or broadcast paths ending in S (cf. Sec. 3.4). Here, it finds three "interesting paths" for statement S_3. These are the three paths pointing to S_3, namely: $P_1 = (e_1)$ (chain path), $P_2 = (e_2)$ and $P_3 = (e_3)$ (broadcast paths). Their relations are: $R_{P_1} = R_{e_1}$, $R_{P_2} = R_{e_2}$, $R_{P_3} = R_{e_3}$ (cf. Fig. 7c).

The corresponding projections and kernels are:

$$\phi_1(k,i,j) = \text{proj}_{(1,0,0)}(k,i,j) = (0,i,j) \qquad K_1 = \text{Ker}(\phi_1) = \langle(1,0,0)\rangle$$
$$\phi_2(k,i,j) = (k,j) \qquad K_2 = \text{Ker}(\phi_2) = \langle(0,1,0)\rangle$$
$$\phi_3(k,i,j) = (k,i) \qquad K_3 = \text{Ker}(\phi_3) = \langle(0,0,1)\rangle$$

The domain D_S is initialized at line 9 to

$$D_S := D_{S_3} = \{S_3[k,i,j] : 0 \le k < N \ \wedge\ k+1 \le i < N \ \wedge\ k+1 \le j \le i\}.$$

The **foreach** loop then iterates over $\{P_1,P_2,P_3\}$. Here the three paths have domains that almost span the entire domain of S_3, so the condition at line 12 is always true (γ is a constant between 0 and 1). They also have pairwise orthogonal kernels, so the condition at line 14 is also true at each iteration, and at the end of the **foreach** loop:

$$\begin{aligned} D_S &= ((D_{S_3} \cap \text{Dom}(P_1)) \cap \text{Dom}(P_2)) \cap \text{Dom}(P_3) \\ &= \{S_3[k,i,j] : 1 \le k < N \ \wedge\ k+1 \le i < N \ \wedge\ k+1 \le j \le i\} \\ \mathcal{P} &= \{P_1,P_2,P_3\} \end{aligned}$$

At line 18, function sub_paramQ_bypartition derives a lower bound from the set of paths $\mathcal{P}$, which is then added to the set of lower bounds $\mathcal{Q}$ by function combine_paramQ.

Without delving into details, sub_paramQ_bypartition derives an upper bound on the size of a K-bounded set in the CDAG, using paths in $\mathcal{P}$ as geometric constraints. A high-level description is given in Sec. 5, and a thorough explanation (detailing in particular the use of the lattice of subgroups $\mathcal{L}$) is available in the full technical report [22].

We can check that P_1 is independent from P_2 and P_3, but P_2 and P_3 are not (that is $R_{P_1}^{-1}(D) \cap R_{P_2}^{-1}(D) = \emptyset$, $R_{P_1}^{-1}(D) \cap R_{P_3}^{-1}(D) = \emptyset$ and $R_{P_2}^{-1}(D) \cap R_{P_3}^{-1}(D) \neq \emptyset$).

Thus the following inequality holds for any K-bounded set E in the CDAG:

$$|\phi_1(E)| + \frac{1}{2}|\phi_2(E)| + \frac{1}{2}|\phi_3(E)| \le K. \tag{2}$$

Let s_1,s_2,s_3 be the solutions to the following optimization problem:

$$\text{Minimize } \sigma := \sum_j s_j \qquad \text{s.t.} \qquad \begin{array}{l} 0 \le s_1,s_2,s_3 \le 1 \\ 1 \le s_2 + s_3 \\ 1 \le s_1 + s_3 \\ 1 \le s_1 + s_2 \end{array} \tag{3}$$

The discrete Brascamp-Lieb theorem [10], combined with Lemma 5.2 from [22], applied on projections ϕ_i, guarantee that, for any K-bounded set E:

$$|E| \le K^{\sigma} \left(\frac{2s_1}{\sigma}\right)^{s_1} \left(\frac{2s_2}{\sigma}\right)^{s_2} \left(\frac{s_3}{\sigma}\right)^{s_3}. \tag{4}$$

The solution to (3) is $s_1 = s_2 = s_3 = \frac{1}{2}$, so

$$|E| \le 2 \cdot (K/3)^{3/2}.$$

Lemma 3.5 tells us that, if U is an upper bound on the size of a $(S+T)$-bounded-set in G, then:

$$Q(G) \ge T \cdot \left\lfloor \frac{|V \setminus \text{Sources}(V)|}{U} \right\rfloor - |\text{Sources}(V)|.$$

Here $V = D \cup R_{P_1}^{-1}(D) \cup R_{P_2}^{-1}(D) \cup R_{P_3}^{-1}(D)$, giving:

$$\begin{aligned} V \setminus \text{Sources}(V) &= \{S_3[k,i,j] : 1 \le k < N \\ &\qquad \wedge\ k+1 \le i < N \ \wedge\ k+1 \le j \le i\} \\ \text{Sources}(V) &= \{S_3[0,i,j] : 1 \le i < N \ \wedge\ 1 \le j \le i; \\ &\qquad S_2[k,i] : 1 \le k < N \ \wedge\ k+1 \le i < N\} \end{aligned}$$

So $|V \setminus \text{Sources}(V)| = \frac{N^3}{6}$ and $|\text{Sources}(V)| = N^2$. [2] Taking for U our upper bound on $|E|$ provides the following inequality for which the objective is to set a value for T that maximizes its right hand side:

$$Q \geq T \times \left\lfloor \frac{N^3/6}{2 \cdot (K/3)^{3/2}} \right\rfloor - N^2 \approx \frac{T}{(S+T)^{3/2}} \times \frac{N^3/6}{2 \cdot (1/3)^{3/2}}.$$

Setting $T = 2S$ (so $K = S + T = 3S$) leads to the following lower bound on Q:

$$Q^{\infty}_{\text{low}} = (2S) \times \frac{N^3/6}{2S^{3/2}} = \frac{N^3}{6\sqrt{S}}.$$

Concerning the parts of the algorithm that were not detailed here: the outermost loop (Line 4) corresponds to the loop parametrization detailed in Sec. 4.3: for each loop depth d, outermost indices are fixed (as parameter Ω_d – Line 6), and parametrically computed lower bounds are summed (when not interfering – Line 22) over all iterations (Line 23 in combine_paramQ). The loop on statements S (Line 5) allows to decompose the full CDAG into as many "S-centric" sub-CDAGs. The so-obtained bounded set of lower bounds $\mathcal{Q}$ are combined using procedure combine_subQ (Line 20) as described in Sec. 4.2. To take compulsory misses into account, the size of the input data of the program is added to the expression.

For each statement S, both techniques (K-partition and wavefront resp. Line 18 and Line 19) generate lower bounds. As opposed to the implicitly considered "S-centric" sub-CDAGs for the wavefront reasoning, an "S-centric" sub-CDAG for the K-partition reasoning (which is built by finding a set $\mathcal{P}$ of DFG-paths that terminate at S – Lines 10-17 through function genpaths) does not necessarily span all the S-vertices (D_S) of the CDAG. So several (non-intersecting) sub-CDAGs can be built until no more interesting lower bound can be derived (Line 17).

8 Experimental Evaluation

IOLB was implemented in C, using ISL-0.13 [33], barvinok-0.37 [6] and PET-0.05 [34]. We also used GiNaC-1.7.4 [7] for the manipulation of symbolic expressions, and PIP-1.4.0 [14] for linear programs. IOLB takes as input an affine C program and outputs a symbolic expression for a lower bound on *I/O* complexity as a function of the problem size parameters of the program and capacity of fast memory.

IOLB was applied to all programs in the PolyBench/C-4.2.1 benchmark suite [23]. For each kernel, our tool outputs an *I/O* lower bound expression Q_{low}, from which we derive an upper bound on operational intensity OI_{up} by forming the ratio of the number of operations and Q_{low}. To evaluate the quality of the results produced by IOLB, we manually generate tiled versions of each kernel, then manually compute parametric data-movement costs as a function of tile sizes and cache size, then manually find the optimal tile sizes and thereby, finally, derive a manually optimized data-movement cost for this kernel. By forming the ratio of the total number of operations and the data-movement cost, we then generate OI_{manual}. In this derivation, we assume that we have explicit control of the cache. Then OI_{manual} is compared with an operational intensity upper-bound obtained by forming the ratio of the number of operations and the data movement lower bound generated by IOLB: OI_{up}.

IOLB runs in less than one second on each of these kernels on a standard computer.

Let us use `jacobi-1d` as an example to illustrate this. IOLB computes a lower bound expression Q_{low} on the number of loads needed for any schedule of the `jacobi-1d` kernel:

$$Q_{\text{low}} = 2 + N + \max\left(0, \frac{TN}{4S} - N - T - \frac{1}{4}\frac{N}{S} - \frac{3}{4}\frac{T}{S} - S + 5\right).$$

The first term is the input data size, and the second term is obtained by the partitioning technique. Since the expression of Q_{low} can be quite large, we automatically simplify to Q^{∞}_{low} by only retaining the asymptotically dominant terms, assuming all parameters $N, M, \ldots$ and cache size S tend to infinity, and $S = o(N, M, \ldots)$. Finally, from Q^{∞}_{low} and the fact that the `jacobi-1d` kernel performs $6TN$ operations, we compute an upper bound for the OI of any schedule of the `jacobi-1d` kernel:

$$Q^{\infty}_{\text{low}} = \frac{TN}{4S} \qquad OI_{\text{up}} = \frac{6TN}{Q^{\infty}_{\text{low}}} = 24S$$

8.1 Parametric Bounds for *OI*

Table 1 reports, for each kernel in PolyBench:

- the size of the input data and the number of operations;
- the simplified *I/O* lower bound Q^{∞}_{low} from IOLB;
- the parametric lower and upper bounds on operational intensity OI_{manual} and $OI_{\text{up}} = \frac{\#\text{ ops}}{Q^{\infty}_{\text{low}}}$, and their ratio;
- the best known published OI_{up}, when it exists.

The 30 reported benchmarks can be divided into four categories, corresponding to table divisions:

1. (19 kernels) The ratio $\frac{\#\text{ ops}}{\#\text{ input data}}$ is high, and we manually find that high OI is achievable through tiling. IOLB gives a non-trivial OI upper bound that is within a constant of the manually obtained OI lower bound OI_{manual}. The bound is asymptotically tight for 9 of them, and within a factor of 2 for an additional 5. Except for matrix multiplication (`gemm`), where it matches the best published bound, these are all improvements over previously published results.
2. (7 kernels) The ratio $\frac{\#\text{ ops}}{\#\text{ input data}}$ is a constant: clearly, these cases do not provide enough operations to enable data reuse. The reported lower bound by IOLB is # input data, which is asymptotically tight for 5 of them, and within a factor of 2 for 1 more.

[2] From here on, we omit lower-order additive terms to keep things readable. The full formula output by IOLB is available in App. C of the full report [22].

Table 1. Results on PolyBench benchmarks

kernel	# input data	# ops	Q^{∞}_{low}	$OI_{\text{manual}} \le OI \le OI_{\text{up}}$	ratio	Published OI_{up}
2mm	$N_iN_k+N_kN_j$ $+N_jN_l+N_iN_l$	$2(NiN_jN_k$ $+N_iN_jN_l)$	$2(N_iN_jN_k$ $+N_iN_jN_l)/\sqrt{S}$	$\sqrt{S} \le OI \le \sqrt{S}$	1	–
3mm	$N_iN_k+N_kN_j$ $+N_jN_m+N_mN_l$	$2(NiN_jN_k+N_jN_lN_m$ $+N_iN_jN_l)$	$2(N_iN_jN_k+N_iN_jN_l$ $+N_jN_lN_m)/\sqrt{S}$	$\sqrt{S} \le OI \le \sqrt{S}$	1	–
cholesky	$\frac{1}{2}N^2$	$\frac{1}{3}N^3$	$\frac{1}{6}N^3/\sqrt{S}$	$\sqrt{S} \le OI \le 2\sqrt{S}$	2	$8\sqrt{S}$ [3]
correlation	MN	M^2N	$\frac{1}{2}M^2N/\sqrt{S}$	$\sqrt{S} \le OI \le 2\sqrt{S}$	2	–
covariance	MN	M^2N	$\frac{1}{2}M^2N/\sqrt{S}$	$\sqrt{S} \le OI \le 2\sqrt{S}$	2	–
doitgen	$N_pN_qN_r+N_p^2$	$2N_p^2N_qN_r$	$2N_p^2N_qN_r/\sqrt{S}$	$\sqrt{S} \le OI \le \sqrt{S}$	1	–
fdtd-2d	$3N_xN_y+T$	$11N_xN_yT$	$\frac{2}{3\sqrt{3}}N_xN_yT/\sqrt{S}$	$\frac{11}{24}\sqrt{3}\sqrt{S} \le OI \le \frac{33}{2}\sqrt{3}\sqrt{S}$	36	–
floyd-warshall	N^2	$2N^3$	$2N^3/\sqrt{S}$	$\sqrt{S} \le OI \le \sqrt{S}$	1	$8\sqrt{S}$ [3]
gemm	$N_iN_j+N_jN_k+N_iN_k$	$2N_iN_jN_k$	$2N_iN_jN_k/\sqrt{S}$	$\sqrt{S} \le OI \le \sqrt{S}$	1	$\sqrt{S}$ [30]
heat-3d	N^3	$30N^3T$	$\frac{3}{8}\sqrt[3]{2}N^3T/\sqrt[3]{S}$	$\frac{5}{2}\sqrt[3]{S} \le OI \le 40\cdot 2^{2/3}\sqrt[3]{S}$	$16\cdot 2^{2/3}$	–
jacobi-1d	N	$6NT$	$\frac{1}{4}NT/S$	$\frac{3}{2}S \le OI \le 24S$	16	$48S$ [12]
jacobi-2d	N^2	$10N^2T$	$\frac{2}{3\sqrt{3}}N^2T/\sqrt{S}$	$\frac{5}{4}\sqrt{S} \le OI \le 15\sqrt{3}\sqrt{S}$	$12\sqrt{3}$	$40\sqrt{2}\sqrt{S}$ [12]
lu	N^2	$\frac{2}{3}N^3$	$\frac{2}{3}N^3/\sqrt{S}$	$\sqrt{S} \le OI \le \sqrt{S}$	1	$8\sqrt{S}$ [3]
ludcmp	N^2	$\frac{2}{3}N^3$	$\frac{2}{3}N^3/\sqrt{S}$	$\sqrt{S} \le OI \le \sqrt{S}$	1	$8\sqrt{S}$ [3]
seidel-2d	N^2	$9N^2T$	$\frac{2}{3\sqrt{3}}N^2T/\sqrt{S}$	$\frac{9}{4}\sqrt{S} \le OI \le \frac{27\sqrt{3}}{2}\sqrt{S}$	$6\sqrt{3}$	–
symm	$\frac{1}{2}M^2+2MN$	$2M^2N$	$2M^2N/\sqrt{S}$	$\sqrt{S} \le OI \le \sqrt{S}$	1	$8\sqrt{S}$ [3]
syr2k	$\frac{1}{2}N^2+2MN$	$2MN^2$	$MN^2/\sqrt{S}$	$\sqrt{S} \le OI \le 2\sqrt{S}$	2	$8\sqrt{S}$ [3]
syrk	$\frac{1}{2}N^2+MN$	MN^2	$\frac{1}{2}MN^2/\sqrt{S}$	$\sqrt{S} \le OI \le 2\sqrt{S}$	2	$8\sqrt{S}$ [3]
trmm	$\frac{1}{2}M^2+MN$	M^2N	$M^2N/\sqrt{S}$	$\sqrt{S} \le OI \le \sqrt{S}$	1	$8\sqrt{S}$ [3]
atax	MN	$4MN$	MN	$4 \le OI \le 4$	1	–
bicg	MN	$4MN$	MN	$4 \le OI \le 4$	1	–
deriche	HW	$32HW$	HW	$\frac{16}{3} \le OI \le 32$	6	–
gemver	N^2	$10N^2$	N^2	$5 \le OI \le 10$	2	–
gesummv	$2N^2$	$4N^2$	$2N^2$	$2 \le OI \le 2$	1	–
mvt	N^2	$4N^2$	N^2	$4 \le OI \le 4$	1	–
trisolv	$\frac{1}{2}N^2$	N^2	$\frac{1}{2}N^2$	$2 \le OI \le 2$	1	–
adi	N^2	$30N^2T$	N^2T	$5 \le OI \le 30$	6	–
durbin	N	$2N^2$	$\frac{1}{2}N^2$	$\frac{2}{3} \le OI \le 4$	6	–
gramschmidt	MN	$2MN^2$	$MN^2/\sqrt{S}$	$1 \le OI \le 2\sqrt{S}$	$2\sqrt{S}$	–
nussinov	$\frac{1}{2}N^2$	$\frac{1}{3}N^3$	$\frac{1}{6}N^3/\sqrt{S}$	$1 \le OI \le 2\sqrt{S}$	$2\sqrt{S}$	–

3. (2 kernels) The ratio $\frac{\#\text{ ops}}{\#\text{ input data}}$ is high which does not discard potential for tiling and high OI. Our best manual schedule leads to a constant OI which is arbitrarily far from this optimistic ratio. IOLB proves that the code is not tileable, the best achievable OI is a constant. IOLB finds this upper bound on OI thanks to the wavefront technique. This is better by at least a factor of $\sqrt{S}$ than any bound that could be obtained by geometric reasoning.
4. (2 kernels) There is an arbitrarily large discrepancy between OI_{up} and OI_{manual}. Visual examination shows that, for these cases, IOLB is too optimistic. These codes are actually not tileable in all dimensions, and we believe that it is possible, using more advanced techniques that are currently out of the scope of IOLB, to prove a matching OI upper bound.

The complete symbolic expressions output by IOLB are available in the full technical report [22] (Appendix C).

9 Related Work

The seminal work of Hong & Kung [18] was the first to present an approach to derive lower bounds on data movement for any valid execution schedule of operations in a computational DAG. Their work modeled data movement in a two-level memory hierarchy and presented manually derived decomposability factors (asymptotic order complexity, without scaling constants) for a few algorithms like matrix multiplication and FFT. Several efforts have sought to build on the fundamental lower bounding approach devised by Hong & Kung, usually targeting one of two objectives: i) generalizing the cost model to more realistic architecture hierarchies [8, 9, 28], or ii) providing an *I/O* complexity with (tight)

constant for some specific class of algorithms (sorting/FFT [2, 26], relaxation [27], or linear algebra [4, 5, 11, 19]).

In the context of linear algebra, Irony et al. [19] were the first to use the Loomis-Whitney inequality [21] to find a lower bound on data movement. This was in the context of `gemm` (one of the kernels of PolyBench). The asymptotic upper bound on *OI* from this paper is $4\sqrt{2}\sqrt{S}$. IOLB returns $\sqrt{S}$. This result was then extended in [3] to 7 more kernels of PolyBench: `cholesky`, `floyd-warshall`, `lu`, `symm`, `syr2k`, `syrk`, and `trmm`, where their upper bound on *OI* is $8\sqrt{S}$ for all of them. IOLB returns $\sqrt{S}$ for 4 of these kernels, and $2\sqrt{S}$ for the other 3. The method presented in [3] is limited to a few algorithms. Kwasniewski et al. [20] implemented an algorithm for parallel matrix-matrix multiplication that matches the communication lower bound for any combinations of matrix dimensions, processors counts and memory sizes. See discussion on [10] for more details on these limitations.

The studies that are the most related to this paper are those from Christ et al. [10], and Elango et al. [12, 13].

The idea of using a variant of the Brascamp-Lieb inequality to derive bounds for arbitrary affine programs comes from Christ et al. [10]. However, the approach they propose suffers from several limitations: 1. The model is based on association of operations with data elements and does not capture data dependencies in a computational DAG. Consequently, it can lead to very weak lower bounds on data movement for computations such as Jacobi stencils. 2. There is no way to (de-)compose the CDAG, and they view all the statements of the loop body (that has to be perfectly nested) as an atomic statement. As a consequence, it is incorrect to use this approach for loop computations where loop fission is possible. 3. The lower bounds modeling is restricted to S-partitioning, leading to very weak lower bounds for algorithms such as `adi` or `durbin`. 4. Obtaining scaling constants, in particular with non-orthogonal reuse directions, is difficult, and only asymptotic order complexity bounds can be derived. 5. No automation of the lower bounding process was proposed, but manually worked out examples of asymptotic complexity as a function of fast memory capacity (without scaling constants) were presented.

Elango et al. [12] used a variant of the red-blue pebble game without recomputation, enabling the composition of several sub-CDAGs, and the use of a lower-bounding approach based on wavefronts in the DAG. Manual application of the approach for parallel execution was done on specific examples, but no approach to automation was proposed.

The later work of Elango et al. [13] was the first to make the connection between paths in the data-flow graph and regular data reuse patterns and to propose an automated compiler algorithm for affine programs. However, their proposed approach suffers from several limitations: 1. Only asymptotic $\Omega(\ldots)$ data movement bounds were obtainable, without any scaling constants. In contrast, IOLB generates meaningful non-asymptotic parametric *I/O* lower bound expressions. From these expressions, we can derive asymptotic lower bounds with scaling constants, critical for use in deducing upper limits on *OI* for a roofline model. 2. Since they were only trying to provide asymptotic bounds without constants, they did not address (de-)composition (asymptotic bounds can be safely summed up even if they interfere). Also, they only considered enumerative decomposition, and not dimension decomposition through loop parameterization that is necessary to obtain a tight bound for their Matmult-Seidel illustrative example. They also only considered the simple non-overlapping notion of interference, and did not allow decomposition of the same statement, required in order to obtain a tight bound for computations like `floyd-warshall`. 3. Finally, their approach only used the S-partitioning paradigm for lower bounds but not the wavefront-based paradigm, thus leading to very weak bounds for benchmarks such as `adi` or `durbin`.

10 Conclusion

This paper presents the first compile-time analysis tool to automatically compute a non-asymptotic parametric lower bound on the data movement complexity of an affine program. For a cache/scratchpad of limited size S, the minimum required data movement in the two-level memory hierarchy is expressed as a function of S and program parameters. As a result, the tool enables, for a representative class of algorithms that fit in the polyhedral model, to automatically derive a bound on the best achievable operational intensity for all possible valid schedule of a given algorithm. Its effectiveness has been illustrated on a full benchmark suite of dense algorithms, the PolyBench suite, with results matching or improving over the current state of the art for many of them. We believe our automated tool has strong potential to assist algorithm developers reasoning and understanding the fundamental memory movement limits of their algorithms, by alleviating the need to manually reason and prove I/O lower bounds.

References

[1] Laksono Adhianto, S. Banerjee, Michael W. Fagan, Mark Krentel, Gabriel Marin, John M. Mellor-Crummey, and Nathan R. Tallent. 2010. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[2] Alok Aggarwal and Jeffrey S. Vitter. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31 (1988), 1116–1127. Issue 9.

[3] Grey Ballard, Erin Carson, James Demmel, Mark Hoemmen, Nick Knight, and Oded Schwartz. 2014. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica* 23 (2014), 1–155.

[4] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2011. Minimizing Communication in Numerical Linear Algebra. *SIAM J. Matrix Analysis Applications* 32, 3 (2011), 866–901.

[5] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2012. Graph expansion and communication costs of fast matrix multiplication. *J. ACM* 59, 6 (2012), 32.

[6] Alexander I. Barvinok. 1994. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research* 19, 4 (1994), 769–779.

[7] Christian Bauer, Alexander Frink, and Richard Kreckel. 2002. Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language. *J. Symbolic Computation* 33 (2002), 1–12.

[8] Gianfranco Bilardi and Enoch Peserico. 2001. A characterization of temporal locality and its portability across memory hierarchies. *Automata, Languages and Programming* (2001), 128–139.

[9] Gianfranco Bilardi, Michele Scquizzato, and Francesco Silvestri. 2012. A Lower Bound Technique for Communication on BSP with Application to the FFT. In *Euro-Par 2012 Parallel Processing - 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings*. 676–687.

[10] Michael Christ, James Demmel, Nicholas Knight, Thomas Scanlon, and Katherine Yelick. 2013. *Communication Lower Bounds and Optimal Algorithms for Programs That Reference Arrays — Part 1.* EECS Technical Report EECS–2013-61. UC Berkeley.

[11] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. 2012. Communication-optimal Parallel and Sequential QR and LU Factorizations. *SIAM J. Scientific Computing* 34, 1 (2012), A206–A239.

[12] Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2014. On characterizing the data movement complexity of computational DAGs for parallel execution. In *Proc. of the 26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, Prague, Czech Republic - June 23 - 25, 2014*. 296–306.

[13] Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2015. On Characterizing the Data Access Complexity of Programs. In *Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 567–580.

[14] Paul Feautrier. 1988. Parametric integer programming. *RAIRO Recherche Opérationnelle* 22, 3 (1988), 243–268.

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

[16] Paul Feautrier and Christian Lengauer. 2011. Polyhedron model. In *Encyclopedia of Parallel Computing*. 1581–1592.

[17] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-Oblivious Algorithms. In *Proc. of the 40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*. 285–298.

[18] Jia-Wei Hong and H. T. Kung. 1981. I/O complexity: The red-blue pebble game. In *Proc. of the 13th Annual ACM Symposium on Theory of Computing (STOC '81), May 11-13, 1981, Milwaukee, Wisconsin, USA* (Milwaukee, Wisconsin, United States). 326–333.

[19] Dror Irony, Sivan Toledo, and Alexandre Tiskin. 2004. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel and Distrib. Comput.* 64, 9 (2004), 1017–1026.

[20] Grzegorz Kwasniewski, Marko Kabic, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. 2019. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*. 24:1–24:22.

[21] Lynn H. Loomis and Hassler Whitney. 1949. An inequality related to the isoperimetric inequality. *Bull. Am. Math. Soc.* 55 (1949), 961–962.

[22] Auguste Olivry, Julien Langou, Louis-Noël Pouchet, P. Sadayappan, and Fabrice Rastello. 2019. Automated Derivation of Parametric Data Movement Lower Bounds for Affine Programs. arXiv:1911.06664 [cs.CC]

[23] Louis-Noël Pouchet and Tomofumi Yuki. 2015. PolyBench/C 4.2. http://polybench.sf.net/.

[24] J. Ramanujam and P. Sadayappan. 1992. Tiling multidimensional iteration spaces for multicomputers. *J. Parallel and Distrib. Comput.* 16, 2 (1992), 108–230.

[25] Desh Ranjan, John E. Savage, and Mohammad Zubair. 2010. Upper and Lower I/O Bounds for Pebbling *r*-Pyramids. In *Combinatorial Algorithms - 21st International Workshop, IWOCA 2010, London, UK, July 26-28, 2010, Revised Selected Papers*. 107–120.

[26] Desh Ranjan, John E. Savage, and Mohammad Zubair. 2011. Strong I/O Lower Bounds for Binomial and FFT Computation Graphs. In *Computing and Combinatorics*. LNCS, Vol. 6842. 134–145.

[27] Desh Ranjan, John E. Savage, and Mohammad Zubair. 2012. Upper and lower I/O bounds for pebbling r-pyramids. *J. Discrete Algorithms* 14 (2012), 2–12.

[28] John E. Savage. 1995. Extending the Hong-Kung model to memory hierarchies. In *Computing and Combinatorics*. LNCS, Vol. 959. 270–281.

[29] John E. Savage and Mohammad Zubair. 2008. A unified model for multicore architectures. In *Proc. of the 1st international forum on Next-generation multicore/manycore technologies, IFMT 2008, Cairo, Egypt, November 24-25, 2008*. 9.

[30] Tyler Michael Smith, Bradley Lowery, Julien Langou, and Robert A. van de Geijn. 2019. A Tight I/O Lower Bound for Matrix Multiplication. arXiv:1702.02017v2

[31] Volker Strassen. 1969. Gaussian elimination is not optimal. *Numerische mathematik* 13, 4 (1969), 354–356.

[32] Sven Verdoolaege. 2010. ISL: An integer set library for the polyhedral model. In *Mathematical Software–ICMS 2010*. 299–302.

[33] Sven Verdoolaege. 2018. Integer Set Library: Manual. http://isl.gforge.inria.fr/manual.pdf.

[34] Sven Verdoolaege and Tobias Grosser. 2012. Polyhedral Extraction Tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*.

[35] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.