IOOpt: Automatic Derivation of I/O Complexity Bounds for Affine Programs

Auguste Olivry, Guillaume Iooss, Nicolas Tollenaere, Atanas Rountev, P Sadayappan, Fabrice Rastello

IOOpt: Automatic Derivation of I/O Complexity Bounds for Affine Programs[*]

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

Guillaume Iooss
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
38000 Grenoble, France

Nicolas Tollenaere
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
38000 Grenoble, France

Atanas Rountev
Ohio State University
Columbus, OH, 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

Evaluating the complexity of an algorithm is an important step when developing applications, as it impacts both its time and energy performance. Computational complexity, which is the number of dynamic operations regardless of the execution order, is easy to characterize for affine programs. Data movement (or, I/O) complexity is more complex to evaluate as it refers, *when considering all possible valid schedules*, to the minimum required number of I/O between a slow (e.g. main memory) and a fast (e.g. local scratchpad) storage location.

This paper presents IOOpt, a fully automated tool that automatically bounds the data movement of an affine (tilable) program. Given a tilable program described in a DSL, it automatically computes: 1. a lower bound of the I/O complexity as a symbolic expression of the cache size and program parameters; 2. an upper bound that allows one to assess the tightness of the lower bound; 3. a tiling recommendation (loop permutation and tile sizes) that matches the upper bound. For the lower bound algorithm which can be applied to any affine program, a substantial effort has been made to provide bounds that are as tight as possible for neural networks: In particular, it extends the previous work of Olivry et al. to handle multi-dimensional reductions and expose the constraints associated with small dimensions that are present in convolutions. For the upper bound algorithm that reasons on the tile band of the program (e.g. output of a polyhedral compiler such as PluTo), the algebraic computations involved have been tuned to behave well on tensor computations such as direct tensor contractions or direct convolutions. As a bonus, the upper bound algorithm that has been extended to multi-level cache can provide the programmer with a useful tiling recommendation.

We demonstrate the effectiveness of our tool by deriving the symbolic lower and upper bounds for several tensor contraction and convolution kernels. Then we evaluate numerically the tightness of our bound using the convolution layers of Yolo9000 and representative tensor contractions from the TCCG benchmark suite. Finally, we show the pertinence of our I/O complexity model by reporting the running time of the recommended tiled code for the convolution layers of Yolo9000.

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

Keywords: Compilation; I/O complexity; Polyhedral model; Convolution; Tensor Contraction

ACM Reference Format:
Auguste Olivry, Guillaume Iooss, Nicolas Tollenaere, Atanas Rountev, P. Sadayappan, and Fabrice Rastello. 2021. IOOpt: Automatic Derivation of I/O Complexity Bounds for Affine Programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454103

1 Introduction

Over the last few decades, the rate of improvement of the peak arithmetic (computational) performance (GFLOPs) of processors has exceeded the rate of improvement of the peak data movement bandwidth from memory. The machine balance of systems, the ratio of peak computational performance to memory performance [9], has steadily increased. Unless a computation has an operational intensity (ratio of number of executed arithmetic operations to number of data

[*]This work was supported in part by the U.S. National Science Foundation through award 2018016, by MIAI Grenoble Alpes (ANR-19-P3IA-0003), and by the Bpifrance Programme d'Investissements d'Avenir (PIA) as part of the ES3CAP project.

```
for(i = 0; i < Ni; i++)
  for(j = 0; j < Nj; j++)
    for(k = 0; k < Nk; k++)
      C[i][j] += A[i][k] * B[k][j];
```

```
for(i1 = 0; i1 < Ni; i1+=Ti)
  for(j1 = 0; j1 < Nj; j1+=Tj)
    for(k = 0; k < Nk; k++)
      for(i = i1; i < i1+Ti; i++)
        for(j = j1; j < j1+Tj; j++)
          C[i][j] += A[i][k] * B[k][j];
```

Listing 1. Matrix-matrix multiplication (untiled and tiled versions)

elements moved from/to main memory) that exceeds the machine balance, the performance achieved will be limited by the memory bandwidth bottleneck.

While the computational complexity of algorithms, that is, the number of arithmetic/logic operations, is well understood, their data movement complexity, that is, the minimum required number of data movement operations to/from memory, is very challenging to characterize. The early work of Hong and Kung [20] devised a methodology for (manually) deriving asymptotic data movement lower bounds (they used the term I/O lower bounds, and we will do so too). Recent work in program analysis [14, 28], using novel mathematical ideas [11], has resulted in tools for the automated derivation of I/O lower bounds for affine computations, as expressions of symbolic problem size parameters and the size of fast memory. Thus, for specific problem sizes and cache size, a lower bound on the minimum required I/O can be computed. However, an open problem has been the assessment of the tightness of the derived lower bounds for programs. Indeed, zero is a valid lower bound on the I/O for any computation and any cache size, but it is a useless and trivial lower bound.

The only way of establishing the tightness of an I/O lower bound is to find a matching upper bound on I/O for the computation, by exhibiting an implementation that attains it. Consider the example of matrix-matrix multiplication. Hong and Kung established that for a fast memory of size S, at least $\Omega\left(\frac{N^3}{\sqrt{S}}\right)$ I/O operations would be required for any valid schedule for the standard $O(N^3)$ algorithm for matrix-matrix multiplication. More recent analysis [36] established a lower bound with scaling constant as $2\frac{N^3}{\sqrt{S}} - 3S$, which exactly matches the highest order term for the data movement of an optimally tiled implementation of matrix-matrix multiplication (see Listing 1).

In this paper, we present the first automated static analysis tool (IOUB) that derives parametric expressions for I/O upper bounds for affine computations. In addition, we present an improvement to the prior state of the art in automated I/O lower bounds (IOLB), of particular relevance to modeling the I/O complexity of convolutional neural networks. IOOpt, which combines the IOLB and IOUB tools, has been used to tightly bound the inherent I/O complexity of two important classes of affine computations that we highlight as case studies in this paper: the family of tensor contraction expressions, and convolutional neural networks. IOOpt can also recommend a tiled code (with appropriate loop permutation and tiles sizes) that minimizes the data movement as modeled in our formulation. The complete flow of our analysis is depicted in Figure 1.

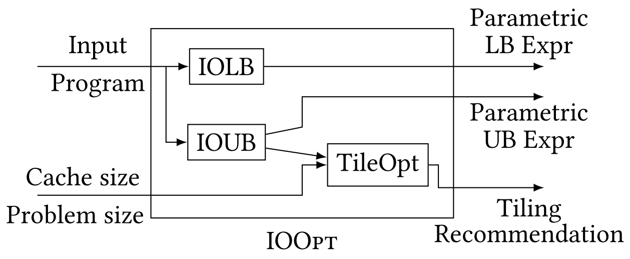

Figure 1. Organisation of the different components of IOOpt

The paper makes the following contributions:

1. Design of the first algorithm for computing a symbolic over-approximation of the data movement for a parametric (multi-dimensional) tiled version of an affine code;
2. Design of the first fully automated scheme for expressing as an operations research problem the minimization of this data movement expression;
3. Extension of the state of the art [28] for the derivation of tight I/O complexity lower bounds in the presence of small dimensions;
4. Integration of these techniques into a tool that computes, for a class of affine computations: 1. an arithmetic complexity; 2. proved lower and upper bounds on I/O complexity; 3. a suggested tiled code that minimizes data movement.

We evaluate our tool by considering both the layers (convolutions) of Yolo9000 [32] and the main benchmarks (tensor contractions) of the TCCG suite [37]. To assess the tightness of our lower and upper data movement bounds, we compute those bounds for a range of cache sizes.

The rest of the paper is organized as follows: Sec. 2 presents a high-level overview of our contributions through an example. Sec. 3 provides some useful formalism along with the required background. Sec. 4 describes our upper bound algorithm. Sec. 5 describes our contributions to the lower bound algorithm. Sec. 6 reports the results of our experiments. Sec. 7 discusses related work, before Sec 8 concludes.

2 Overview

I/O lower and upper bounds. Let us consider a simple memory model with two levels: a small and fast memory (of size S) and a slow but infinite memory. We assume that a piece of data has to be in the fast memory in order to be used in a computation, but we have the possibility of transferring data between the two memory levels. Now, given a program, supposing we can reorder operations and have complete control over data transfers, *how many transfers between these memories do we need to perform at the very least in order to compute it*? This quantity is called the I/O complexity of a program. To compute it exactly is usually intractable, as one would have to simulate every possible valid schedule of operations. Hence the need to find lower and upper bounds on this complexity.

It is possible to derive a lower bound of the I/O complexity by studying the structure of the program through a graph representation called a CDAG, which represents operations and data dependencies between computations [3, 20, 28]. However, we also wish to evaluate how tight this bound is. Thus, we seek an *upper bound* of the smallest volume of I/O required by a computation. This upper bound can be obtained by simply exhibiting a schedule of the computation that has this volume of I/O. Our contribution is a novel algorithm which finds such a schedule, then compares the corresponding upper bound on the I/O to the lower bound, in order to evaluate its tightness. We also improve the state-of-the-art automated lower bound method, allowing us to find matching bounds in more cases.

In the rest of this section, we apply our algorithms informally to the matrix multiplication example described in Listing 1, in order to provide some intuitions on how they work. This program has three parameters N_i, N_j and N_k, corresponding to matrix sizes. Its arithmetic complexity is $N_i N_j N_k$.

Derivation of an upper bound on the I/O complexity. Any valid schedule of a program provides an upper bound on its I/O complexity. For a given schedule, a memory allocation strategy, and fixed numerical values of memory size S and loop bounds, it is sufficient to count loads and stores. However, the goal is to get a symbolic bound.

For the category of programs we consider, the key transformation to get an I/O-efficient schedule is *tiling* [42]. Thus, we first designed a method to compute an upper bound of the required I/O of a given tiled program: For a given tiled program with parametric tile sizes, IOUB automatically computes the corresponding I/O cost (a symbolic expression as a function of program parameters and tile sizes) along with the footprint constraint (an analytical inequality that bounds the tile footprint with the cache size S). This expression can then be used by IOUB to derive an upper bound on the actual I/O complexity, that is, to find the best tile sizes that minimize the I/O cost under the footprint constraint.

Another key component is the selection of a subset of relevant tiling schemes (loop permutation). The bottom code in Listing. 1 corresponds to the best tiling scheme found by IOUB: here, T_i and T_j are tile sizes along loop dimensions i and j, and tiles have size 1 along dimension k. The corresponding symbolic expressions of the I/O cost and footprint constraint found by IOUB are $IO = N_i N_j N_k \left(\frac{1}{T_i} + \frac{1}{T_j} + \frac{1}{N_k}\right)$, and $T_i T_j + T_i + T_j \leq S$. We would also like to derive a symbolic expression of the I/O complexity upper bound which only depends on program parameters and memory size S, but not on tile sizes. To do so, we designed a procedure to fix tile sizes as functions of S, using computer algebra (see Sec. 6 for more details). For matrix multiplication, the final expression is

$$UB = N_i N_j \left(\frac{2N_k}{\sqrt{S+1} - 1} + 1 \right).$$

As this underlying optimization problem is difficult in general, TileOpt (see Fig. 1) can look for a numerical solution by using fixed given values for $N_{\{i,j,k\}}$ and S. As an example, for $N_i = 2000$, $N_j = N_k = 1500$, and $S = 1024$, IO is minimized for $T_i = T_j = 31$. For those values, $UB = 296322580$.

Lower bound and tightness. Now that we have automatically derived a symbolic upper bound on the I/O cost for matrix multiplication, we can compare it with the lower bound obtained from the IOLB tool [28] to evaluate its tightness. The lower bound we find is:

$$LB = 2N_i + N_k + 2N_j + \frac{2N_i N_j N_k}{\sqrt{S}}$$

where S is the size of the fast memory. We can check that this bound is asymptotically tight (see Sec. 6).

However, these bounds are not always tight. For example, on a convolution (as shown in Sec. 5.4), the lower bound derived by the currently published version of IOLB is several asymptotic orders below the derived upper bound. This lower bound can be improved by taking into account (i) the *reductions* across multiple dimensions, which refines the dependence analysis performed in the first steps of the algorithm, and (ii) *small dimensions*. Small dimensions are dimensions with a much smaller number of iterations than the size of the fast memory S. This property can be used to improve the power factor of the asymptotic bound. Both improvements to the IOLB algorithm are described in Section 5, and allow us to obtain asymptotically tight bounds for tensor contraction and convolution computations.

Lower vs. upper bound method. Both methods are based on polyhedral compilation tools, for abstract program representation and to compute cardinalities of polyhedra with Barvinok's algorithm [40], but the two approaches are fundamentally different. The lower bound algorithm converts the polyhedral program to a graph representation and uses geometric reasoning on this graph to bound the maximum size of a tile that can fit in the fast memory. It does not provide

```
for(c = 0; c < Nc; c++)
  for(f = 0; f < Nf; f++)
    for(x = 0; x < Nx; x++)
      for(w = 0; w < Nw; w++)
        Out[f][x] += Image[x+w][c]
                   * Filter[f][w][c];
```

Listing 2. Running example - 1D Convolution

a schedule, but rather a proof that no schedule can do less than a certain amount of I/O. On the other hand, the upper bound algorithm extends polyhedral tools to compute the data footprint of affine codes, and uses this to get a symbolic expression of its I/O. It also includes heuritics to prune the search space for tiling schemes (loop permutations).

3 Background

In this section, we present the class of programs, its representations, and the memory model we consider in our analysis.

3.1 Class of Programs

Imperfectly nested affine loop programs. We consider *imperfectly nested* loop programs, which can be recursively defined as a sequence of statements and `for` loops, each being itself composed of imperfectly nested loops. A *program parameter* is a symbolic constant during the compilation, whose value will be known during the execution of the program (for example, array sizes). The access functions made by statements to arrays are *affine expressions* of surrounding loop indices and program parameters. We also assume that conditions on loop indices are affine expressions of surrounding loop indices and program parameters [7].

For example, the 1D-convolution described in Listing 2 matches these criteria. This program is even *perfectly nested*: all loops are nested with a single statement inside. Constants `Nc`, `Nf`, `Nx` and `Nw` are the *program parameters*. All loop indices are bounded by affine constraints (e.g. $0 \leq c < \mathtt{Nc}$), and all array subscripts are affine expressions (e.g. $(x + w, c)$ for array `Image`).

Fully-tilable program. The *tiling transformation* [42] is a key loop transformation to improve the data locality of a program. Given a list of consecutive dimensions (called *loop band*), this transformation groups their iterations into tiles, which are executed atomically. Listing 1 shows an example of tiling transformation for the matrix multiplication example: the tiled dimensions are `i` and `j` and the iterations are grouped into rectangles of size `Ti` × `Tj` (*tile shape*). Note that when the tiling shape is rectangular, each tiled dimension (e.g. `i`) is strip-mined, giving rise to the *tile dimension* that will iterate over the tiles (e.g. `i1`), and the *local dimension* that iterates inside a tile (e.g. `i` in the transformed program).

A tiling is *legal* when there is no cycle of dependencies between the computation of different tiles, that is, when the atomicity condition between tiles can be respected by the schedule of the program. The algorithms presented in this paper are described on the assumption that the input program is *fully tilable using rectangular tiles*, which means that all its dimensions can be legally tiled by using rectangular tiles.

In theory, any affine program could be pre-processed using a polyhedral compiler to provide a fully permutable (that is, tilable using rectangular tiles) loop band, but the reality is more complex, as analyzing a non-regular iteration domain involves being able to cope with the simplification and solving of a complex system of symbolic expressions. Nevertheless, several important classes of computation fit our simplified hypothesis, such as a convolution, all kinds of tensor contractions and many linear algebraic kernels, including matrix multiplication. Listing 2 shows another example of such computation, which is a simplified convolution. It will be used as our running example.

3.2 Program Representation

In this paper, we will mainly use two representations of a program. The *CDAG* representation is an unrolled graph representation of the program, and is used as the base formalism for proving lower bounds. The *polyhedral representation* is a mathematical and concise representation of the structure of the program, and corresponds to the intermediate representation manipulated by our algorithms.

Computational Directed Acyclic Graph (CDAG). This representation will be used by our data movement model.

Definition 3.1. *(CDAG) A computational directed acyclic graph $\mathcal{G} = (V, E, O)$ is a graph formed by:*

- *a finite set of nodes V, each one representing an elementary computation, producing a piece of data,*
- *a finite set of edges E, representing data dependencies: there is an edge from a node v to a node w if the computation at w requires the data from v.*

The input data of the program is represented by the nodes with no incoming edge. The output data is represented by a set O of nodes, potentially with no outgoing edges.

We will consider a subset of a CDAG, which is itself a CDAG: Given a sub-graph (V', E'), the corresponding sub-CDAG is (V', E', O') where $O' = O \cap V'$.

Polyhedral representation. The *iteration domain* $\mathcal{I}_S$ of a statement S is the set of integral values that the surrounding loop indices take during execution. Each point of this space is associated with an execution instance of the statement. Due to the hypotheses made on the loop bounds, the iteration domain of a statement is a *$\mathbb{Z}$-polyhedron*, that is, a set of integral points whose coordinates satisfy a set of affine constraints. The computation of the cardinality of a set E (denoted $|E|$ – symbolic expression as a function of the program parameters), can be efficiently done using the Barvinok algorithm[1] [6] available in any polyhedral framework library such as ISL [38].

Each array A is associated with a *memory domain* $\mathcal{M}_A$, which is the multidimensional set of valid array indices for the array A. Given a statement S containing a read or a write to an array A, this occurrence is associated with a *memory access function* $f_A : \mathcal{D}_S \mapsto \mathcal{M}_A$, which is an affine function. For any given sub-domain $D_S \subset \mathcal{I}_S$ defined as a $\mathbb{Z}$-*polyhedron*, the associated data footprint $f_A(D_S)$ is also a $\mathbb{Z}$-*polyhedron* that can be easily computed by polyhedral compilation tools.

For the matrix-matrix multiplication example (Listing 1, untiled version), the dimensions are $\mathcal{D} = \{i, j, k\}$ and the iteration domain is $\mathcal{I} = \{(i, j, k) \mid 0 \leq i < N_i \ \wedge \ 0 \leq j < N_j \ \wedge \ 0 \leq k < N_k\}$. A 3-dimensional rectangular subset of this domain will be defined as $[l_i, u_i] \times [l_j, u_j] \times [l_k, u_k]$, where, for each dimension $d \in \mathcal{D}$, $0 \leq l_d \leq u_d < N_d$. The example code also contains three array accesses:

- for array A, with domain $\mathcal{M}_A = \{x, y \mid 0 \leq x < N_i \ \wedge \ 0 \leq y < N_k\}$ and access function $f_A(i, j, k) = (i, k)$;
- for array B, with domain $\mathcal{M}_B = \{x, y \mid 0 \leq x < N_k \ \wedge \ 0 \leq y < N_j\}$ and access function $f_B(i, j, k) = (k, j)$;
- and for array C, with domain $\mathcal{M}_C = \{x, y \mid 0 \leq x < N_i \ \wedge \ 0 \leq y < N_j\}$ and access function $f_C(i, j, k) = (i, j)$.

3.3 Memory Model

In order to model the amount of data movement needed, we use the *red-white pebble game*, which is a variation of Hong & Kung's red-blue pebble game [20]. It was introduced in Olivry et al.'s work [28] to provide a convenient framework for deriving I/O lower bounds, and in particular to model the *no recomputation assumption*. Since this work also deals with lower bounds, we use this formalism for both upper and lower bounds.

We consider a hierarchy of two memories:

- An infinite slow memory which contains the input data at the beginning of the execution.
- A fast scratchpad of limited capacity S, which must contain the data used by a computation.

In the red-white pebble game, pebbles can be placed on the nodes of the CDAG. A *white pebble* represents a data which has been computed, and a *red pebble* represents a data which is present in the fast memory. To account for the limited size of the fast memory, there are only S red pebbles. There is an unlimited number of white pebbles.

We can place or remove a pebble of a given color according to the following set of rules, assuming that the limit constraint on the number of simultaneously used red pebbles is fulfilled:

[1] In theory, Barvinok's algorithm has an exponential complexity in the number of dimensions of the set. In practice, its execution time is usually less than a few seconds, when applied to the polyhedral sets commonly encountered during program analysis.

```
for(c1 = 0; c1 < Nc; c+=Tc)
  for(f1 = 0; f1 < Nf; f+=Tf)
    for(x = 0; x < Nx; x++)
 ----------------------------------------------- Tile limit
      for(c = c1; c < c1+Tc; c++)
        for(w = 0; w < Nw; w++)
          for(f = f1; f < f1+Tf; f++)
            Out[f][x] += Image[x+w][c]
                       * Filter[f][w][c];
```

Listing 3. Tiled code for 1D-convolution for tiling schedule $\left((w, c, f, x), \{T_c = \mathtt{Tc}, T_f = \mathtt{Tf}, T_x = 1, T_w = \mathtt{Nw}\}\right)$

- **Fetch rule:** A red pebble can be placed on any node that has a white pebble.
- **Spill rule:** A red pebble can be removed from any node.
- **Computation rule:** If a node has no white pebble and all its immediate predecessors have red pebbles, then a red and a white pebble can be placed on this node.

A *valid game sequence* is a list of applications of these rules (moves) on specific nodes and edges of the CDAG, such that:

- At the start of the game, there is a white pebble placed on every input node.
- At the end of the game, all the nodes of the graph are covered by white pebbles,
- At any point of the game, there are at most S nodes with red pebbles.

A game sequence mirrors the data movement that can be performed by a computation. Thus, the amount of loads from the slow memory done by a computation corresponds to the number of times the *fetch rule* is used in a valid game sequence.

4 Upper Bound on Data Movement

In this section, we first present an algorithm that computes an over-approximation of the required I/O for a given parametric tiled program. We then show how to use this symbolic expression to select a permutation and tiles sizes that minimize the amount of I/O. As an actual valid schedule is associated with the computed I/O, this provides a valid upper bound for the data movement complexity. An interesting side effect of the method is that the computed optimizing schedule can be exposed to the user as a suggested loop transformation.

4.1 Loop Permutation and Tiling

Let us consider the example of Listing 1 that reports a tiled and non-tiled code for matrix-matrix multiplication. The original code contains three dimensions i, j, and k. The tiled code contains five loops on dimensions i, j, k, i, j, from outer to inner. The three outermost loops span the entire iteration domain of size Ni×Nj×Nk, while the two innermost ones span a polyhedron of size Ti×Tj. We refer to the innermost (resp. outermost) part as the intra-tile (resp. inter-tile) dimensions. As we will see later, unlike the permutation of the inter-tile dimensions, that of the intra-tile dimensions is not relevant in our model. In other words, while a different schedule with loops ordered as (from outer to inner) (i1,k,j1,i,j) could lead to a different I/O estimation than for the schedule with loops ordered as (i1,j1,k,i,j) (as in Listing. 1), our cost model will not be affected if we permute i and j as in loop order (i1,j1,k,j,i). This motivates the following notation to represent the tiling schedule of Listing. 1: $((i,j,k),\ \{T_i = \mathtt{Ti}, T_j = \mathtt{Tj}, T_k = 1\})$ where the ordered list $\mathcal{P} = (i,j,k)$ describes the permutation of the inter-tile loop dimensions, and $\mathcal{T} = \{T_i = \mathtt{Ti}, T_j = \mathtt{Tj}, T_k = 1\}$ describes the tile sizes. Similarly, for the example in Listing 2, the tiling shown in Listing 3 is represented as $((w,c,f,x),$ $\{T_c = \mathtt{Tc}, T_f = \mathtt{Tf},\ T_x = 1, T_w = \mathtt{Nw}\})$. Note that the loop on w in the inter-tile loops (outermost loops), and the loop on x in the intra-tile loops are both omitted in the code as they would have only one iteration. The permutation of intra-tile loops in the code is arbitrary.

More formally, a *tiling* schedule is defined as a tuple $(\mathcal{P}, \mathcal{T})$ where $\mathcal{P} = (d_j)_{|\mathcal{D}| \geq j \geq 1}$ is a permutation of dimensions $\mathcal{D}$ representing the inter-tile loop order, and $\mathcal{T} = \{T_d\}_{d \in \mathcal{D}}$ represent the tile dimensions. In $\mathcal{P}$, $d_{|\mathcal{D}|}$ represents the outermost loop dimension, while d_1 represents the innermost loop dimension that encloses the tile: the permutation order is outer (leftmost) to inner (rightmost). The tile size for dimension d is T_d.

For an inter-tile loop dimension d_j, we define its *sub-domain* as the set of points in the iteration domain such that the indices of the enclosing loops d_k ($|\mathcal{D}| \geq k \geq j$) have fixed values. Taking the example of the 1D-convolution of Listing 3, the sub-domain $SD_{d_2}(\mathtt{c1,f1})$ for loop dimension $d_j = d_2 = f$ (recall that $d_4 = w,\ d_3 = c,\ d_2 = f,\ d_1 = x$) is $\{(w,c,f,x) \mid 0 \leq x < \mathtt{Nx} \wedge \mathtt{c1} \leq c < \mathtt{c1+Tc} \wedge 0 \leq w < \mathtt{Nw} \wedge \mathtt{f1} \leq f < \mathtt{f1+Tf}\}$.

More generally, if we denote by ik the fixed index value at level k:

$$SD_{d_j}(\mathtt{iD},\ldots,\mathtt{ij}) = \{i_{|\mathcal{D}|},\ldots,i_1 \in \mathcal{I} \mid \forall k \geq j,\ \mathtt{ik} \leq i_k < \mathtt{ik} + T_{d_k}\}$$

A *Sub-Domain data Footprint* for an array A at level j is defined as:

$$SDF_{A,j}(\mathtt{iD},\ldots,\mathtt{ij}) = \left| f_A\left(SD_{d_j}(\mathtt{iD},\ldots,\mathtt{ij})\right)\right|$$

As an example, for the 1D-convolution running example we have:

$$SDF_{\mathtt{Image},2}(\mathtt{c1,f1}) = (\mathtt{Nx{+}Nw{-}1}) \times \mathtt{Tc}$$

Whenever there is a non-empty overlap of data used between two consecutive sub-domains, estimating it allows us to refine our cost model. We define the *inter-Sub-Domain Reuse* for an array A at level j as:

$$SDR_{A,j}(\mathtt{iD},\ldots,\mathtt{ij}) = \Big| f_A\left(SD_{d_j}(\mathtt{iD},\ldots,\mathtt{ij})\right) \cap f_A\left(SD_{d_j}(\mathtt{iD},\ldots,\mathtt{ij} - T_{d_j})\right)\Big|$$

A sharp eye would have observed that the notion of reuse from a "previous" sub-domain is meaningful for all sub-domains but the first one in the loop. To handle this subtlety, for a given loop dimension d_j (assuming its loop index starts at 0), we split the iteration space into two sub-domains: the *front domain*

$$I_{\text{front}} = \{(i_{|\mathcal{D}|},\ldots,i_1) \in \mathcal{I} \mid i_j = 0\}$$

and the *back domain*

$$I_{\text{back}} = \{(i_{|\mathcal{D}|},\ldots,i_1) \in \mathcal{I} \mid i_j \neq 0\}$$

This allows us to define, for a given sub-domain and an array A, its *inverse density* as the ratio:

$$ID_{A,j}(\mathtt{iD},\ldots,\mathtt{ij} \neq 0) = \frac{SDF_{A,j}(\mathtt{iD},\ldots,\mathtt{ij}) - SDR_{A,j}(\mathtt{iD},\ldots,\mathtt{ij})}{\left|SD_{d_j}(\mathtt{iD},\ldots,\mathtt{ij})\right|}$$

$$ID_{A,j}(\mathtt{iD},\ .\ .\ .\ ,\mathtt{0}) = \frac{SDF_{A,j}(\mathtt{iD},\ldots,\mathtt{ij})}{\left|SD_{d_j}(\mathtt{iD},\ldots,\mathtt{ij})\right|}$$

As we will see later, the inverse density is an over-approximation of the best attainable inverse operational intensity (data movement per computation unit). As an example, for the 1D-convolution example,

$$\begin{aligned}
SD_x(\mathtt{c1,f1,x}) = &\quad \{c,f,w,x \mid 0 \leq w < \mathtt{Nw}\ \wedge \\
&\quad x = \mathtt{x}\ \wedge\ \mathtt{c1} \leq c < \mathtt{c1{+}Tc}\ \wedge \mathtt{f1} \leq f < \mathtt{f1{+}Tf}\} \\
|SD_x(\mathtt{c1,f1,x})| = &\quad \mathtt{Nw} \times \mathtt{Tc} \times \mathtt{Tf} \\
SDF_{\mathtt{Image},1}(\mathtt{c1,f1,x}) = &\quad \mathtt{Nw} \times \mathtt{Tc} \\
SDR_{\mathtt{Image},1}(\mathtt{c1,f1,x}) = &\quad \mathtt{Tc} \times (\mathtt{Nw} - 1) \\
ID_{\mathtt{Image},1}(\mathtt{c1,f1,x} \neq 0) = &\quad 1/(\mathtt{Nw} \times \mathtt{Tf}) \\
ID_{\mathtt{Image},1}(\mathtt{c1,f1,0}) = &\quad 1/\mathtt{Tf}
\end{aligned}$$

To avoid complicated expressions for non-rectangular domains, we consider the maximum value for the inverse density that we specialize for the front and the back:

$$ID^{\text{front}}_{A,j} = \max_{\mathtt{iD},\ldots,\mathtt{ij}/\mathtt{ij}=0} ID_{A,j}(\mathtt{iD},\ldots,\mathtt{ij})$$

$$ID^{\text{back}}_{A,j} = \max_{\mathtt{iD},\ldots,\mathtt{ij}/\mathtt{ij}\neq 0} ID_{A,j}(\mathtt{iD},\ldots,\mathtt{ij})$$

In many cases, the sub-domain sizes and intersections do not depend on loop indices, and there is no need to take the maximum.

4.2 Cost Model

Let us start with the case where there is a single array A. We consider a tiling $(\mathcal{P} = (d_{|\mathcal{D}|}, \ldots, d_1), \mathcal{T} = \{T_d\}_{d \in \mathcal{D}})$.

First, the data footprint of a tile has to be smaller than the cache capacity S:

$$SDF_{A,1} \leq S.$$

Second, let us define the *outermost reuse dimension* as the "first" (smallest possible l) dimension d_l in $\mathcal{P}$ such that the sub-domain data footprint of A is less than the cache capacity S. That is, l is the leftmost index in $\mathcal{P}$ such that

$$SDF_{A,l} \leq S.$$

The total I/O cost for array A derived from the inverse density as follows constitutes an upper bound on the cost of an optimal red-white pebble game:

$$IO_A = ID^{\text{front}}_{A,l} \times |I_{\text{front}}| + ID^{\text{back}}_{A,l} \times |I_{\text{back}}|$$

Indeed, by construction, each sub-domain can be executed by bringing only once (before starting execution) all the required data. Every sub-domain in the back can reuse the data used in the previous sub-domain and the corresponding I/O can thus be saved. This means that the optimal inverse operational intensity for any sub-domain in the front (resp. in the back) is no more than $ID^{\text{front}}_{A,l}$ (resp. $ID^{\text{back}}_{A,l}$).

When there are multiple arrays $A_1, \ldots, A_s$, we "cut" the cache into array-specific regions of sizes $S_1, \ldots, S_s$, such that $S_1 + \cdots + S_s = S$. The conditions are the same as above, replacing A and S by A_j and S_j.

On our running example, for array Image, supposing the outermost reuse dimension is x:

$$\begin{aligned} IO_{\texttt{Image}} &= ID^{\text{front}}_{\texttt{Image},1} \times |I_{\text{front}}| + ID^{\text{back}}_{\texttt{Image},1} \times |I_{\text{back}}| \\ &= \frac{1}{\texttt{Tf}} \times \texttt{Nw.Nc.Nf} + \frac{1}{\texttt{Nw.Tf}} \times \texttt{Nw.Nc.Nf.(Nx} - 1) \\ &= \frac{\texttt{Nc.Nf.(Nx + Nw} - 1)}{\texttt{Tf}} \end{aligned}$$

The same computation for arrays Out and Filter yields (assuming the outermost reuse dimensions are respectively x and f):

$$\begin{aligned} IO_{\texttt{Out}} &= \texttt{Nc.Nf.Nx/Tc} \\ IO_{\texttt{Filter}} &= \texttt{Nc.Nf.Nw} \end{aligned}$$

The red-white pebble game and the corresponding lower bound reasoning frameworks can be extended to multi-level memory hierarchies [34]. The above computation of I/O can also be extended by simply considering one tiling band per cache level and independently applying the previous reasoning to each level.

4.3 Loop Permutation Selection

We have only focused on computing the required I/O cost for a given permutation of the inter-tile loops. Having a symbolic expression as a function of the tile sizes allows the use of a Non-Linear optimization Problem (NLP) solver such as IPOPT [41] to find optimizing tile sizes for this particular permutation. Unfortunately, the search space of all possible permutations grows exponentially with the number of dimensions and with the memory depth.

However, it is straightforward to see that many permutations are equivalent in terms of I/O cost: for example, switching the two outer loops `c1` and `f1` in the conv-1D example in Listing 3 would not change the I/O costs. Moreover, some permutations are better than others: for example, it is not worth exploring the permutation whose innermost dimensions do not allow any data reuse between two successive tiles. In this section, we provide some insights into how to select a subset of relevant permutations for a given code.

First, we define a notion of *reuse*: for a given dimension d and an array A, there is *reuse* for array A on dimension d if, when putting dimension d innermost (that is, setting $d_1 = d$), the sub-domain data footprint at level 2 for A *does not increase much* compared to level 1:

$$SDF_{A,2} - SDF_{A,1} \ll SDF_{A,1}$$

This allows us to decide which dimensions are worth putting at the innermost levels.

In simple cases, this notion is not ambiguous: for array Out in the conv-1D example, c is a reuse dimension as setting it innermost would lead to $SDF_{\texttt{Out}_2} = SDF_{\texttt{Out}_1} = \texttt{Tf} \times \texttt{Tx}$, while f is not a reuse dimension as setting it innermost would lead to $SDF_{\texttt{Out}_2} - SDF_{\texttt{Out}_1} = (\texttt{Nf} - \texttt{Tf}) \times \texttt{Tx}$ (unless $\texttt{Nf} = \texttt{Tf}$ but in that case it means that f iterates only once). For more complex subscript expressions, such as for array Image, the criterion is less clear: setting w as innermost and assuming it is not part of the tile would lead to comparing $\texttt{Tw} - 1$ with $\texttt{Tx}$ (as $SDF_{\texttt{Image}_2} = (\texttt{Tx} + \texttt{Tw} - 1) \times \texttt{Tc}$, $SDF_{\texttt{Image}_1} = \texttt{Tx} \times \texttt{Tc}$). In other words, a reuse criterion requires an "oracle" (such as the user) to provide information about small and large dimensions.

Assuming such an oracle and a reuse criterion, Algorithm 1 selects a set of representative permutations. This algorithm builds a permutation from the innermost to the outermost dimensions while keeping track of the set of remaining dimensions (variable $\mathcal{D}'$), and of the set S of arrays having a potential reuse along each dimension d (variable R). When a dimension is chosen, the set of potential reuse is updated. When there is no more potential reuse, an arbitrary permutation of the remaining dimensions is chosen.

Figure 2 illustrates the steps of the algorithm on the 1D convolution (Listing 2), and how it narrows the loop permutation space to three permutations (one of them is the one used for the running example in Listing 3). Note that some permutations are pruned during the selection process (red cross in Figure 2), because they have strictly less reuse potential than others. For example, selecting dimension c as the innermost dimension allows us to have reuse on array Out, but dimension w allows us to have reuse on one more array (Out and Image).

```
input  : dimensions $\mathcal{D}$, arrays $\mathcal{A}$, reuse oracle
output : list of permutations $\mathcal{P}$
function genPerm($\mathcal{D}'$, $R$)
    if $\mathcal{D}' = \emptyset$ then
        return $\{()\}$;
    if $S = \emptyset$ for all $(d, S) \in R$ then
        return $\{P\}$ where $P$ is an arbitrary
          permutation of $\mathcal{D}'$;
    $\mathcal{P} := \emptyset$;
    forall $(d, S) \in R$ such that $\nexists (d', S') \in R,\ S \subsetneq S'$
     do
        $\mathcal{D}'_d := \mathcal{D}' \setminus \{d\}$;
        $R_d := \{(d', S' \cap S), (d', S') \in R \setminus \{(d, S)\}\}$;
        $\mathcal{P}_d :=$ genPerm($\mathcal{D}'_d$, $R_d$);
        $\mathcal{P} := \mathcal{P} \cup \{(P, d), P \in \mathcal{P}_d\}$;
    return $\mathcal{P}$;
$R := \{(d, \{A \in \mathcal{A}, \text{reuse}(A, d)\}), d \in \mathcal{D}\}$;
return genPerm ($\mathcal{D}$, $R$);
```

Algorithm 1: Generation of the list of loop permutations that maximize reuse

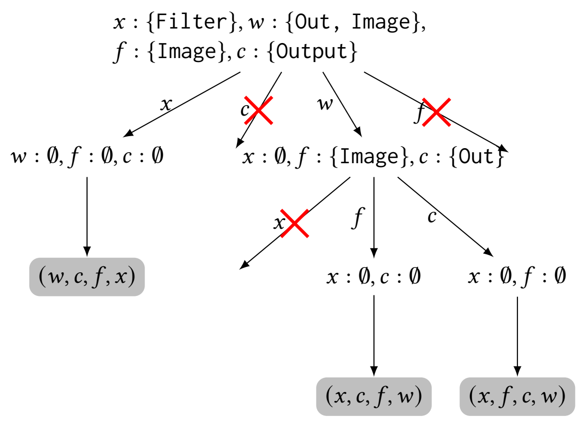

Figure 2. Application of Algorithm 1 to the 1D convolution kernel

4.4 Putting It All Together

The IOUB tool combines the permutation selection and I/O cost computation as follows: it first selects a list of permutations and generates the corresponding set of tiled versions (with tile sizes as parameters) using Algorithm 1. For each permutation, it computes a symbolic expression (as a function of program parameters, tile sizes, and cache size) that represents the required I/O, using the method described in Sec. 4.1 and 4.2, as well as constraints on tile sizes. This is implemented using the ISL [38] and Sympy [27] libraries. Then, this can be fed to TileOpt, which uses an NLP solver (IPOPT [41]) to find the best tile sizes for each permutation. The final I/O cost is the minimum over all versions, and TileOpt also provides a basic tiled code that implements the corresponding tiling scheme. It can also be useful to have a symbolic bound that does not depend on tile sizes, but only on program parameters and cache size. A discussion on how to derive such an expression for the examples we used in experiments can be found in Sec. 6.

5 Lower Bound on Data Movement

In this section, we consider the problem of finding a symbolic lower bound on the volume of loads needed to perform an affine computation. We first present the main intuitions behind the partitioning method, which is one of the state-of-the-art techniques to derive a symbolic lower bound [11, 20, 28]. We then provide two improvements on this method, namely reductions and small dimensions. These are needed to find asymptotically tight bounds for some programs, in particular convolutions. Indeed, let us consider the 2D convolution from Figure 3a. For the sake of examining the asymptotic behavior of the bound, let us assume that all program parameters are equal to N and greater than the small memory size S. With the state-of-the-art method, the lower bound that is found is $O(N^4)$. If we detect the reduction and adapt the dependence analysis before applying the partitioning method (Section 5.3), the bound is improved to $O(N^7/S)$. We can further improve this bound by exploiting the fact that H and W are usually small parameters that are much smaller than S. By refining the partitioning method (Section 5.2), we obtain $O(\sqrt{HW}N^5/\sqrt{S})$ as a lower bound, and this bound is asymptotically tight.

5.1 Background: The Partitioning Method for IOLB

The partitioning method considers a red-white pebble game over the CDAG of a computation (both notions were introduced in Section 3). A complete mathematical formalization of the content of this subsection can be found in [28].

Another key notion is a *K-bounded set*, which is a set of nodes of the CDAG with at most K predecessors. If we consider a $(S+T)$-bounded set (where S is still the size of the fast memory and $T > 0$), then we need to load at least T pieces of data in the fast memory to be able to perform the computation. This gives a lower bound on the amount of communication needed to compute this set. Any valid sequence of moves in the red-white pebble game can be decomposed into a $(S+T)$*-partition*, i.e., a partition of the CDAG into $(S+T)$-bounded sets. Therefore, a lower bound on the total number of loads can be deduced from an upper bound on the number of sets in a $(S+T)$-partition.

When the dependencies of the computation are affine, we can find an upper bound on the size of a $(S+T)$-bounded set by using a geometrical argument called the *Brascamp-Lieb inequality*. Its mathematical formulation is:

Theorem 5.1 (Brascamp-Lieb inequality). *Let d and d_j be non-negative integers and $\phi_j : \mathbb{Z}^d \mapsto \mathbb{Z}^{d_j}$ be group homomorphisms for all $1 \le j \le m$.*

If we have a set of $s_j \in [0, 1]$ such that, for all subgroups $\mathcal{H}$ of $\mathbb{Z}^d$:

$$rank(\mathcal{H}) \le \sum_{j=1}^{m} s_j \cdot rank(\phi_j(\mathcal{H}))$$

Then, for all nonempty finite sets $E \in \mathbb{Z}^d$:

$$|E| \le \prod_{j=1}^{m} |\phi_j(E)|^{s_j}$$

Here, $rank(G)$ is the subgroup rank, defined as the smallest cardinality of a generating set for G. This can be seen as an equivalent to the dimension of a vector space in a discrete setting.

This result can intuitively be seen as a generalization of the following inequality for 3-dimensional vector spaces:

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 $|V| \le C^{3/2}$.

Vertices in the 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 $(S+T)$-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 $S + T$". Finding a bound on the size of a $(S + T)$-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 the cardinality bounds on some of its projections.

This theorem is applied to a K-bounded set E (where $K = S + T$). We select the group homomorphisms ϕ_j such that their images are included in the input set of E. Indeed, by definition of a K-bounded set, we will have $|\phi_j(E)| \le K$, for all $1 \le j \le m$. Therefore, we will obtain the upper bound: $|E| \le K^\sigma$ where $\sigma = \sum_j s_j$.

To select such homomorphisms ϕ_j, we study the paths of affine dependence of the CDAG involving E that originate with a vertex in the input set of E. These paths are associated by a *path relation*, which is the composition of the affine functions of the dependencies composing a path. We build a homomorphism from each path relation.

Then, we need to find a set of rationals s_j that satisfy the linear constraints from Theorem 5.1, and minimize σ. In order to find the subgroups $\mathcal{H}$ that lead to the tightest constraints on s_j, we use bases of $Ker(\phi_j)$. We select a combination of vectors of these bases, and consider the subgroup generated by these vectors to build a constraint on s_j.

Once we have a symbolic upper bound of the size of a $(S + T)$-bounded set, we obtain a symbolic lower bound on the volume of data communications by the program, for any schedule.

5.2 Adapting the Partitioning Method to Small Dimensions

We assume that some of the dimensions of the iteration domain are *small*, i.e., their size is orders of magnitude smaller than C. We note the product of all the sizes of the small dimensions N_{sd}. We adapt the partitioning method by considering an additional group homomorphism ϕ_{sd} to the application of Theorem 5.1. This group homomorphism is a projection of the space on the small dimensions, which gives us a much tighter bound $|\phi_{sd}(E)| \le N_{sd}$.

Assuming a set of s_j and s_{sd} (coefficient associated with ϕ_{sd}) satisfying the constraints of Theorem 5.1, we obtain the following upper bound:

$$|E| \le \prod_j |\phi_j(E)|^{s_j} \cdot |\phi_{sd}(E)|^{s_{sd}} \le K^\sigma \cdot N_{sd}^{s_{sd}}$$

where $\sigma = \sum_j s_j$, not including s_{sd}.

Note that the constraints on the s_j are relaxed due to the contribution of s_{sd}, giving us an opportunity to find smaller values of s_j, compared to the usual method. Moreover, in order to tighten this upper bound, we should minimize σ first, then s_{sd} next.

5.3 Using Reductions

A *reduction* is the successive application of an associative and commutative binary operator over a set of values. For example, in the convolution kernel described in Figure 3, we have a reduction over 3 dimensions (c, h and w) with the addition operation. We only consider reductions that sum all the values along some dimensions, called *reduced dimensions*. This property can be used to sum the values of a set in any order, instead of having a fixed sequence of summation.

Because the program is expressed as a loop nest, its reduction is sequential. Thus, when we examine the CDAG, we have a chain of dependencies linking the nodes along the reduced dimensions. When there are more than 2 reduced dimensions, it impacts negatively our study of the paths of dependencies of a CDAG, thus the quality of the extracted homomorphisms ϕ_j and the upper bound found on the size of E.

We detect a simple class of reduction by performing a pattern-matching on the affine dependencies of the CDAG. We impose that the reduced dimensions are defined over a rectangular domain and that the summation is performed in a lexicographic order over a permutation of these reduced dimensions. Once a reduction is detected, we replace the sequential chain of dependencies (between the nodes of the reduction) by two sets of broadcast dependencies: (i) starting from the computations using the final result of the reduction to all the nodes of the reduction, and (ii) starting from the initialization of the reduction to the nodes of the reduction.

```
for (b = 0; b < B; b++)
  for (c = 0; c < C; c++)
    for (f = 0; f < F; f++)
      for (x = 0; x < X; x++)
        for (y = 0; y < Y; y++)
          for (h = 0; h < H; h++)
            for (w = 0; w < W; w++) {
               Out[f,x,y,b] += Image[x+h,y+w,c,b]
                                 * Filter[f,h,w,c];
}
```

(a) Convolution kernel.

$\phi_1(b,c,f,x,y,h,w) = (b,0,f,x,y,0,0)$	(from reduction)
$\phi_2(b,c,f,x,y,h,w) = (x+h,y+w,c,b)$	(from Image)
$\phi_3(b,c,f,x,y,h,w) = (f,h,w,c)$	(from Filter)
$\phi_{sd}(b,c,f,x,y,h,w) = (h,w)$	(small dimensions)

(b) Homomorphisms used in Brascamp-Lieb inequality.

$(\vec{f})$	$1 \le s_1 + s_3$	
$(\vec{b})$	$1 \le s_1 + s_2$	
$(\vec{c})$	$1 \le s_2 + s_3$	
$(\vec{x})$	$1 \le s_1 + s_2$	
$(\vec{w})$	$1 \le s_2 + s_3$	$+s_{sd}$
$(\vec{w} - \vec{x})$	$1 \le s_1 + s_3$	$+s_{sd}$
$(\vec{w}, \vec{x})$	$2 \le s_1 + s_2 + s_3$	$+s_{sd}$
$(\vec{y})$	$1 \le s_1 + s_2$	
$(\vec{h})$	$1 \le s_2 + s_3$	$+s_{sd}$
$(\vec{h} - \vec{y})$	$1 \le s_1 + s_3$	$+s_{sd}$
$(\vec{h}, \vec{y})$	$2 \le s_1 + s_2 + s_3$	$+s_{sd}$

(c) Constraints on s_j from Brascamp-Lieb inequality.

- No small dimensions: $s_j = \frac{2}{3}$, $\sigma = 2$.
- Small dimensions: $s_j = \frac{1}{2}$, $s_{sd} = \frac{1}{2}$, $\sigma = \frac{3}{2}$.

(d) Solution obtained for the s_j, minimizing $\sigma = \sum_j s_j$.

Figure 3. Partitioning method - Brascamp-Lieb inequality applied to a convolution, without and with small dimensions (H and W).

On the convolution example of Figure 3, if we do detect the reduction, the dependence on Out is along the dimension s, and we will find as a corresponding homomorphism $\phi_1(b,c,f,x,y,h,w) = (b,c,f,x,y,h,0)$. If we detect the reduction, the homomorphism of the new path becomes $\phi_1(b,c,f,x,y,h,w) = (b,0,f,x,y,0,0)$, which leads to a bigger kernel and better constraints on s_j.

5.4 Example - Convolution

To illustrate this method, we consider a convolution computation, described in Figure 3a.

Derivation with no small dimensions. This derivation corresponds to the one found in Demmel and Dihn [12]. By examining the affine dependencies of the program, we find three homomorphisms ϕ_1, ϕ_2 and ϕ_3, described in Figure 3b.

As stated by Theorem 5.1, we consider the subgroups $\mathcal{H}$ generated from the combination of kernel vectors of the ϕ_j. We obtain the constraints shown in Figure 3c, ignoring the rightmost column. Their corresponding subgroups are shown in the leftmost column.

Finally, we solve this system of linear inequality to obtain a solution for the s_j, which corresponds to the upper bound on the size of a K-bounded set: $|E| \le K^2$.

Derivation with small dimensions. Now, we assume that the product of the problem size parameters H and W is small compared to the size of the small memory (S). Compared to the first derivation, we have an additional homomorphism ϕ_{sd}, shown in Figure 3b. The presence of this new homomorphism adds a new coefficient s_{sd} to the constraints on the s_j, as shown in Figure 3c. We obtain a better solution for the s_j, which corresponds to the upper bound on the size of a K-bounded set: $|E| \le K^{\frac{3}{2}} \cdot (H \cdot W)^{\frac{1}{2}}$. This is a tighter upper bound.

6 Experiments

Using our new I/O cost computation method, and our improvements to the lower bound algorithm for computing I/O complexity, IOOpt is able to provide tight bounds for both tensor contractions (even with small dimensions) and convolutions.

Benchmarks. We illustrate this by running it on representative benchmarks: For convolutions, we considered 11 different layers of Yolo9000 [32]. The parameter values for each layer are shown in Fig. 4. For tensor contractions, we considered the ones from TCCG [37].

The benchmark Python script from TCCG source code gathers 73 tensor contraction kernels (originating from various sources), that can be reduced to 49 different kernels, once synonyms are identified. For each kernel, TCCG selects the sizes of every dimension so that they are multiple of 8, roughly equal, and so that the product of all sizes is around 200×2^{20}.

We can further reduce the number of relevant kernels that need to be considered in our analysis, by grouping them according to the number of dimensions of each array, and the number of dimensions shared between them. Indeed, the array layouts (dimension order) do not impact our analysis. This yields eight classes of tensor contraction kernels, described in Figure 5.

Layer	F	C	X	Y	W	H
Yolo9000-0	32	3	544	544	3	3
Yolo9000-2	64	32	272	272	3	3
Yolo9000-4	128	64	136	136	3	3
Yolo9000-5	64	128	136	136	1	1
Yolo9000-8	256	128	68	68	3	3
Yolo9000-9	128	256	68	68	1	1
Yolo9000-12	512	256	34	34	3	3
Yolo9000-13	256	512	34	34	1	1
Yolo9000-18	1024	512	17	17	3	3
Yolo9000-19	512	1024	17	17	1	1
Yolo9000-23	28272	1024	17	17	1	1

Figure 4. Parameter values for convolutional layers of Yolo9000

Kernel	dim.	s. d.	Problem sizes
abcde-efbad-cf	5 5 2	4 1 1	48/32/24/32/48/32
abcd-dbea-ec	4 4 2	3 1 1	72/72/24/72/72
abc-bda-dc	3 3 2	2 1 1	312/312/296/312
abcdef-dega-gfbc	6 4 4	3 3 1	24/16/16/24/16/16/24
abc-adec-ebd	3 4 3	2 1 2	72/72/72/72/72
ab-cad-dcb	2 3 3	1 1 2	312/296/312/312
ab-ac-cb	2 2 2	1 1 1	5136/5136/5120
abcd-aebf-fdec	4 4 4	2 2 2	72/72/72/72/72/72

Figure 5. Classes of Tensor Contraction kernels from the TCCG benchmarks. The classes are determined from the number of dimensions of the arrays (Out / In1 / In2), and the number of shared dimensions (s. d.) between arrays (Out+In1 / Out+In2 / In1+In2)

Parametric lower bound expressions. Figure 6 shows the parametric lower bound found for the tensor contraction kernels and the 2D convolution kernels. For each line of this array, the first term of the maximum function is the sum of the volumes of the input arrays used by the computation, which is trivially a lower bound of the number of I/O. Because the lower bound derived is still sound, even if the small dimension hypothesis is not satisfied, we have to combine various lower bounds for different small dimension scenarios together.

For the tensor contraction kernels, we considered $8 = 2^3$ small dimension scenarios. Dimensions shared between two arrays are grouped together, and every combination of small/regular dimensions for those three groups is examined. For convolution kernels, we considered 5 small dimension scenarios, based on array sizes: (i) no small parameters, (ii) H and W small parameters, (iii) H, W and B small parameters, (iv) H, W, X, Y and B small parameters, and (v) C, H, W and B small parameters. Among those bounds, only the first three scenarios lead to interesting bounds. They correspond to the last three rows in the expression. The previous version of the algorithm (without reduction management or small dimensions) fails to find an interesting bound, and returns the sum of array sizes (first row in the expression).

Symbolic upper bound expressions. The method presented in Sec. 4 provides a symbolic expression for the I/O of a program, as a function of program parameters and tile sizes. To compare them with the parametric lower bound expressions, we would like to remove tile sizes from the expression, and express them as functions of the cache size instead. In general, this is too complicated to solve. However, for the benchmarks we consider, this is possible by using only a few assumptions.

Let us start with the matrix multiplication example from Listing 1, with tiling $\big((i,j,k), \{T_i = \mathtt{Ti}, T_j = \mathtt{Tj}, T_k = 1\}\big)$. The IO cost expression and the constraints on tile sizes are:

$$IO = IO_A + IO_B + IO_C = \mathtt{Ni}\cdot\mathtt{Nj}\cdot\mathtt{Nk}\cdot\left(\frac{1}{\mathtt{Ti}}+\frac{1}{\mathtt{Tj}}+\frac{1}{\mathtt{Nk}}\right) \quad (1)$$

$$SDF_{A,1} + SDF_{B,1} + SDF_{C,1} = \mathtt{Ti} + \mathtt{Tj} + \mathtt{Ti}\cdot\mathtt{Tj} \le S \quad (2)$$

Then, we consider square tiles, by assuming that `Ti` and `Tj` are equal to the same value T. We also assume that the tile completely fills the cache, which means that we consider that inequality (2) is actually an equality. While this may not be the best solution, these hypotheses will still provide a valid bound. We obtain the following equality:

$$2T + T^2 = S.$$

It has a unique positive solution:

$$T = \sqrt{S+1} - 1.$$

We can then plug this value back into expression (1) to get our symbolic bound:

$$IO = \mathtt{Ni}\cdot\mathtt{Nj}\cdot\left(\frac{2\mathsf{Nk}}{\sqrt{S+1}-1}+1\right)$$

Note that the dominant term matches the dominant term of the I/O lower bound for matrix multiplication.

Here, we chose the tiling scheme manually, but this can be automated using TileOpt with some relevant numerical values of parameters. It will return a permutation and numerical tile sizes, and we can use the information on which dimensions have $T_d = 1$ or $T_d = N_d$ to guide our symbolic solving. If the problem sizes are significantly bigger than $\sqrt{S}$, the best tiling found by the solver will precisely be this one.

To generalize this reasoning to tensor contraction kernels, we only need to add some extra information on the expected order of magnitude of tile sizes.

Dimensions in a tensor contraction computation can be divided into three groups such that after "merging" the dimensions in each group, the computation is equivalent to a matrix multiplication. This corresponds to the "shared dimensions" in Fig. 5. The condition we impose is that the

Kernel	I/O bounds: lower (LB) and upper (UB)
TC abcde-efbad-cf	$\mathrm{UB} = \frac{2ABCDEF}{\sqrt{S+1}-1} + CF$
	$\mathrm{LB} = \max\left(ABCDE + EFBAD + CF, -2 + F - S + 2C + 2ABDE + \frac{2ABCDE(F-1)}{\sqrt{S}}\right)$
TC abcd-dbea-ec	$\mathrm{UB} = \frac{2ABCDE}{\sqrt{S+1}-1} + CE$
	$\mathrm{LB} = \max\left(ABCD + DBEA + EC, -2 + E - S + 2C + 2ABD + \frac{2ABCD(E-1)}{\sqrt{S}}\right)$
TC abc-bda-dc	$\mathrm{UB} = \frac{2ABCD}{\sqrt{S+1}-1} + CD$
	$\mathrm{LB} = \max\left(ABC + BDA + DC, -2 + D - S + 2C + 2AB + \frac{2ABC(D-1)}{\sqrt{S}}\right)$
TC abcdef-dega-gfbc	$\mathrm{UB} = \frac{2ABCDEFG}{\sqrt{S+1}-1} + BCFG$
	$\mathrm{LB} = \max\left(ABCDEF + DEGA + GFBC, -2 + G - S + 2ADE + 2BCF + \frac{2ABCDEF(G-1)}{\sqrt{S}}\right)$
TC abc-adec-ebd	$\mathrm{UB} = \frac{2ABCDE}{\sqrt{S+1}-1} + BDE$
	$\mathrm{LB} = \max\left(ABC + ADEC + EBD, -3 + DE - S + 3AC + 3B - ABC + \frac{2ABC(DE-1)}{\sqrt{S}}\right)$
TC ab-cad-dcb	$\mathrm{UB} = \frac{2ABCD}{\sqrt{S+1}-1} + AB$
	$\mathrm{LB} = \max\left(AB + CAD + DCB, -3 + CD - S + 3A + 3B - AB + \frac{2AB(CD-1)}{\sqrt{S}}\right)$
TC ab-ac-cb	$\mathrm{UB} = \frac{2ABC}{\sqrt{S+1}-1} + BC$
	$\mathrm{LB} = \max\left(AB + AC + CB, -2 + C - S + 2A + 2B + \frac{2AB(C-1)}{\sqrt{S}}\right)$
TC abcd-aebf-fdec	$\mathrm{UB} = \frac{2ABCDEF}{\sqrt{S+1}-1} + CDEF$
	$\mathrm{LB} = \max\left(ABCD + AEBF + FDEC, -3 + EF - S + 3AB + 3CD - ABCD + \frac{2ABCD(EF-1)}{\sqrt{S}}\right)$
2D Convolution	$\mathrm{UB} = CFHWXY\left(\frac{1}{XY} + \frac{1}{H\Delta W} + \frac{(H+\Delta-1)(W+X-1)}{H\Delta^2 WX}\right)$ where $\Delta = \frac{-HW+W+\sqrt{H^2W^2+4HSW-2HW^2+4SW+4S+W^2}}{2(HW+W+1)}$
	$\mathrm{LB} = \max\Big(BC(Y+H-1)(X+W-1) + BFXY + FCHW,$ $-2 - S + C + 4F + BY + BX + 2BXY - 2BXYF + \frac{BFXY(WHC-1)}{S},$ $-2 - S + C + 4F + BY + BX + 2BXY - 2BXFY + \frac{2BXYCF\sqrt{HW}}{\sqrt{S}} - \frac{2BXYF}{\sqrt{HWS}},$ $-2 - S + C + 4F + BY + BX + 2BXY - 2BXFY + \frac{2XYCF\sqrt{BHW}}{\sqrt{S}} - \frac{2XYF\sqrt{B}}{\sqrt{HWS}}\Big)$

Figure 6. Combined parametric I/O bounds of tensor contraction (TC) and 2D convolution kernels. S is the small memory size and other uppercase letters are problem sizes (for TC kernels A…F are tensor dimensions, for convolution see Fig. 3a).

products of tile sizes inside each group are equal. For example `abc-adec-ebd`, the three groups are $\{a, c\}$, $\{b\}$ and $\{d, e\}$, and the condition is $T_a T_c = T_b = T_d T_e$.

The expressions are shown in Figure 6. They correspond to the "general case" where the input size is not the bottleneck, i.e., when parameters are sufficiently large compared to S.

The expression for convolution is quite complex, but an asymptotic analysis shows that the highest order term is $\frac{2CFXY\sqrt{HW}}{\sqrt{S}}$ when C, F, X and Y are sufficiently large, which matches the third term in the lower bound (the B factor does not appear since it is equal to 1 in all our benchmarks).

Comparison of upper and lower bounds for different cache sizes. Figure 7 shows a comparison between lower and upper bounds for the considered benchmarks. For several cache sizes (from 16 kB to 4MB), the lower bound is computed by plugging the actual values in the symbolic expressions shown above, and the upper bound is obtained by running TileOpt on the optimization problems generated by IOUB.

As a sanity check, we confirm that the upper bound found is always above the lower bound, which was not trivial for the convolution bounds, and that both curves are decreasing or are constant along S. Both bounds are close to each other, ranging from at most a factor of 3 between them for Yolo9000-2 on the smallest value of S, to almost the same values for the *ab-ac-cb* TC kernel, which is also known as matrix multiplication. Moreover, the upper bound and the lower bound come closer for the largest values of S, which shows a match between the asymptotic dominant term of both bounds. When the cache size becomes very large, the dominant cost becomes the initial loading of the input data,

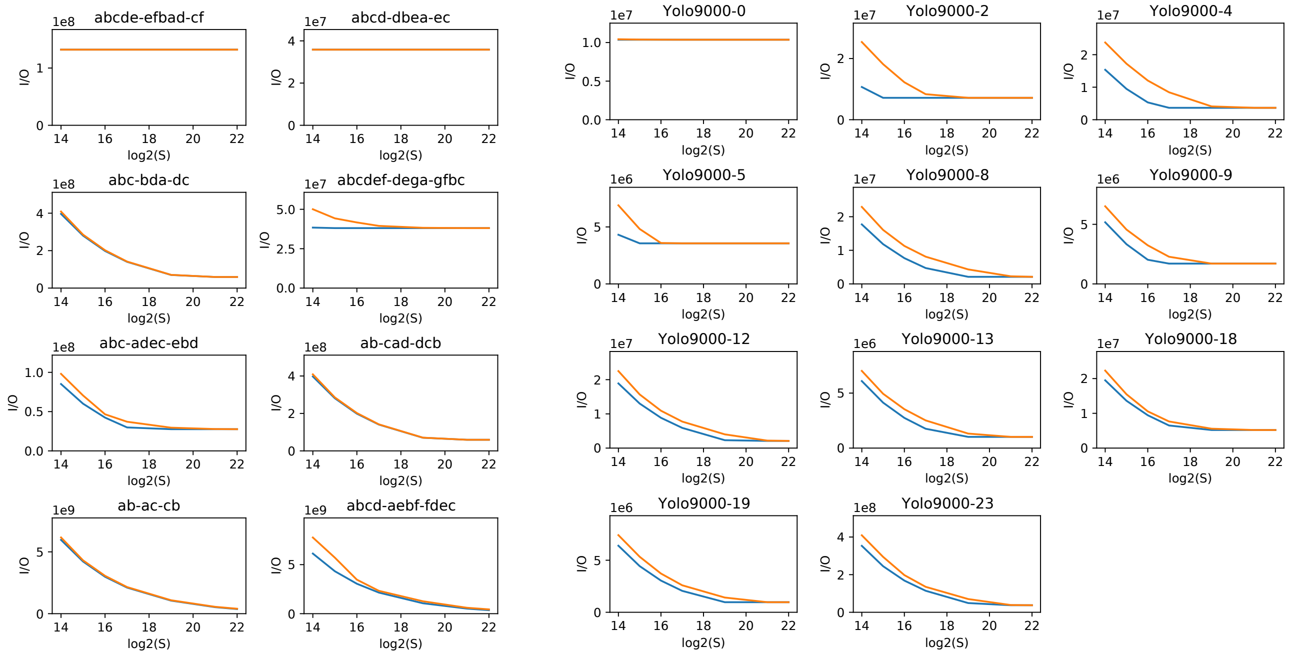

Figure 7. I/O bounds (lower is blue and upper is orange) of tensor contractions (TCCG) and convolutions (Yolo) for different cache sizes

hence the matching bounds and the horizontal line for some benchmarks.

Evaluation of the tiling recommendation. We evaluate the efficiency of the tiling recommendation we produce on the Yolo-9000 benchmark (Figure 4). The tiling recommendation has been obtained using a simple multi-level extension of the cache model where the objective function minimizes the weighted (with available measured inverse bandwidth) sum of data movements between the different cache levels. We fix the innermost dimension of the permutation in order to force vectorization on dimension f. Note that fixing the innermost dimensions of our tiling recommendation does not impact the rest of our model.

In Figure 8, we compare the original code (no tiling), the tiled code based on our tiling recommendation, with the convolution from OneDNN [21], a state-of-the-art library. We ran our experiment on an Intel i9-7940X Skylake-X machine with AVX-512 vector instructions, a 32kB L1 cache, a 1MB L2 cache and a shared 20MB L3 cache. Each code is run 200 times, and the median is taken.

As expected, our implementation does not outperform OneDNN (with the exception of the first layer), but is still reasonably fast. Indeed, many optimizations such as register tiling that expose instruction level parallelism without stressing too much register pressure, packing, versioning, etc., which are implemented by OneDNN, are not present in our "naive" implementation. Thus, our tiling strategy can be improved in order to obtain a good performance.

Kernel	No Tiling	OneDNN	Tiling reco
Yolo9000-0	8%	27%	31%
Yolo9000-2	14%	52%	28%
Yolo9000-4	18%	64%	36%
Yolo9000-5	14%	77%	48%
Yolo9000-8	14%	76%	34%
Yolo9000-9	15%	84%	35%
Yolo9000-12	14%	77%	34%
Yolo9000-13	8%	74%	27%
Yolo9000-18	13%	66%	33%
Yolo9000-19	6%	73%	6%
Yolo9000-23	3%	72%	3%

Figure 8. Comparison of the efficiency of our tiling recommendation with a state-of-the-art convolution implementation. The numbers are percentages of performance compared to the theoretical machine peak.

Limitations. While the methodology presented here is very general on the theoretical side, practical implementation has some limitations. When iteration domains get complex, for instance when loop counters depend on one another, or loop bounds are more involved functions of parameters, cardinalities computed by Barvinok's algorithm get more complex, with several possible expressions depending on relations between parameters. This can greatly increase the size of the optimization problem, making it currently intractable. Another limit is the method to generate symbolic upper bounds: currently we rely on solving a polynomial equation, which is not generally doable when the degree exceeds four. This could be addressed by relaxing the problem of finding the precise expression of a tile size that completely fills the cache to only finding a size that does not exceed the cache capacity.

7 Related Work

I/O complexity lower bound. Most papers on computing lower bounds are handmade proofs for specific algorithms. While early works including the seminal paper of Hong and Kung [20] focused on deriving asymptotic order lower bounds without bothering about the constant factor, several more recent contributions have finally derived tight (up to lower order terms) lower bounds with scaling constants for a few specific classes of algorithms [1–3, 13, 22, 23, 30, 31, 36]. The early work of Chris et al. [11], followed by that of Elango et al. [14], paved the way to algorithmic approaches for automatic derivation of I/O complexity lower bounds for the subclass of affine programs. The first tool able to actually automate the process and apply it to a full benchmark suite is IOLB from Olivry et al. [28]. One important issue of IOLB with regard to neural networks is the looseness of the obtained bound for convolutions. This is due to its inability to exploit the presence of small dimensions, as opposed to the recent work of Demmel et al. [12] which uses a similar trick to that of Elango et al. in [14] for deriving tight bounds for CNN. The main limitation of [12, 14] resides in the inability to provide scaling constants. The algorithm presented in Sec. 5 extends the work of Olivry et al. for exploiting the presence of small dimensions without losing its main advantage: being able to provide an exact (not restricted to asymptotic order inequality) bound with scaling constant.

I/O complexity upper bound, cache miss prediction & tile size selection . Computing an I/O complexity upper bound for an algorithm is the most reasonable way to assess the tightness of a lower bound. While this computation is usually done by hand using ad hoc techniques specific to each studied algorithm [1, 12, 23, 28, 31, 36], Fauzia et al. [15] proposed a heuristic that directly reasons on the CDAG, which unfortunately does not scale to real programs. Finding an upper bound for a fixed architecture can also be viewed as finding an optimized program transformation that minimizes data movement costs, which also implies being able to evaluate this cost. Thus, restricting the analysis to affine programs and using the polyhedral framework appears to be appropriate for this problem. However, while the seminal scheduling algorithm from PluTo [8] is able to expose tilable loops and generate tiled code, it can only handle fixed tile sizes. This is because parametric tiling is not an affine transformation: a tiled affine code with parametric tile sizes is no longer affine. A consequence is that existing polyhedral tools cannot be used to evaluate the I/O cost of such a code. These tools include PolyFeat [4], which computes an approximation of the number of capacity misses for arbitrary affine programs, as well as other algorithms [5, 10, 18] which focus on precisely modeling conflict misses. Some of these algorithms are restricted to a small class of programs, and most of them can only model a one-level cache. They all need to consider fixed parameters and fixed tile sizes. Other works [24, 35, 43] implemented ad hoc computation of this cost function for very restricted sub-classes of programs, but our algorithm is the first to be able to automatically generate a symbolic expression of it for arbitrary parametric tiled affine programs, in a multi-level cache setting.

We showed how we use this cost function to find the best loop permutations and tile sizes for a given architecture, using operations research. Once again, this is out of the scope of polyhedral analysis, as cost functions are not affine. So the usual optimization strategies used by polyhedral compilers [8, 19, 39], mostly based on parametric integer programming [16], cannot be used. Some literature exists on tile size selection, but it mostly deals with performance model design and is either handcrafted for specific kernels and access function patterns [25, 29] or uses machine learning [43] and even cache simulators [26]. Polyhedral analysis is only used for the tiling transformation. Renganarayana and Rajopadhye [33] showed that most performance models for tile size selection used in the literature are polynomials that are operations research-friendly. Our automatically generated cost function, which matches the distinct-access model promoted by Ferrante et al. [17], fits into this category.

8 Conclusion

We have presented an algorithm which computes an upper bound on the data movement complexity of a polyhedral program. It outputs a symbolic formula on the problem sizes and the cache size, and advises a loop permutation and tiling which minimizes it. Additionally, we have improved an algorithm which computes the lower bound of the data movement complexity, by exploiting the fact that some parameters are much smaller than the cache size. Both algorithms were implemented as a fully automatic tool and evaluated on multiple convolution and tensor contraction kernels. We have evaluated the numerical tightness of these bounds by evaluating them for the problem sizes of tensor contraction and convolution benchmarks.

References

[1] Alok Aggarwal and Jeffrey S. Vitter. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31 (1988), 1116–1127. Issue 9. https://doi.org/10.1145/48529.48535

[2] 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. https://doi.org/10.

1137/090769156

[3] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2013. Graph Expansion and Communication Costs of Fast Matrix Multiplication. *Journal of the ACM* 59, 6, Article 32 (Jan. 2013), 23 pages. https://doi.org/10.1145/2395116.2395121

[4] Wenlei Bao, Changwan Hong, Sudheer Chunduri, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. 2016. Static and Dynamic Frequency Scaling on Multicore CPUs. *ACM Transactions on Architecture and Code Optimization* 13, 4 (2016), 26 pages. https://doi.org/10.1145/3011017

[5] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, and P. Sadayappan. 2018. Analytical modeling of cache behavior for affine programs. *Proceeding of the ACM on Programming Languages* 2, POPL (2018), 26 pages. https://doi.org/10.1145/3158120

[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. https://doi.org/10.1287/moor.19.4.769

[7] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. 2003. Putting Polyhedral Loop Transformations to Work. In *Languages and Compilers for Parallel Computing, 16th International Workshop (LCPC) (Lecture Notes in Computer Science, Vol. 2958)*. Springer, 209–225. https://doi.org/10.1007/978-3-540-24644-2_14

[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1375581.1375595

[9] David Callahan, John Cocke, and Ken Kennedy. 1988. Estimating Interlock and Improving Balance for Pipelined Architectures. *J. Parallel and Distrib. Comput.* 5, 4 (1988), 334–358. https://doi.org/10.1016/0743-7315(88)90002-0

[10] Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. 2001. Exact Analysis of the Cache Behavior of Nested Loops. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 286–297. https://doi.org/10.1145/378795.378859

[11] 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. arXiv:1308.0068v1 [math.CA]

[12] James Demmel and Grace Dinh. 2018. Communication-Optimal Convolutional Neural Nets. arXiv:1802.06905v2 [cs.DS]

[13] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. 2012. Communication-optimal Parallel and Sequential QR and LU Factorizations. *SIAM Journal on Scientific Computing* 34, 1 (2012), A206–A239. https://doi.org/10.1137/080731992

[14] Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2015. On Characterizing the Data Access Complexity of Programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 567–580. https://doi.org/10.1145/2676726.2677010

[15] Naznin Fauzia, Venmugil Elango, Mahesh Ravishankar, J. Ramanujam, Fabrice Rastello, Atanas Rountev, Louis-Noël Pouchet, and P. Sadayappan. 2013. Beyond Reuse Distance Analysis: Dynamic Analysis for Characterization of Data Locality Potential. *ACM Transaction on Architecture and Code Optimization* 10, 4 (Dec. 2013), 29 pages. https://doi.org/10.1145/2541228.2555309

[16] Paul Feautrier. 1988. Parametric Integer Programming. *RAIRO Recherche Opérationnelle* 22, 3 (1988), 243–268.

[17] Jeanne Ferrante, Vivek Sarkar, and W. Thrash. 1991. On Estimating and Enhancing Cache Effectiveness. In *Proceedings of the Languages and Compilers for Parallel Computing (LCPC), Fourth International Workshop (Lecture Notes in Computer Science, Vol. 589)*. Springer, 328–343. https://doi.org/10.1007/BFb0038674

[18] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. 1999. Cache Miss Equations: a Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Transaction on Programming Languages and Systems* 21, 4 (1999), 703–746. https://doi.org/10.1145/325478.325479

[19] Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letter* 22, 4 (2012). https://doi.org/10.1142/S0129626412500107

[20] Jia-Wei Hong and H. T. Kung. 1981. I/O complexity: The Red-Blue Pebble Game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC)* (Milwaukee, Wisconsin, United States). ACM, 326–333. https://doi.org/10.1145/800076.802486

[21] Intel. 2020. OneAPI Deep Neural Network Library (oneDNN). https://01.org/oneDNN.

[22] 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. https://doi.org/10.1016/j.jpdc.2004.03.021

[23] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 24 pages. https://doi.org/10.1145/3295500.3356181

[24] Rui Li, Aravind Sukumaran-Rajam, Richard Veras, Tze Meng Low, Fabrice Rastello, Atanas Rountev, and P. Sadayappan. 2019. Analytical Cache Modeling and Tilesize Optimization for Tensor Contractions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Michela Taufer, Pavan Balaji, and Antonio J. Peña (Eds.). ACM, 13 pages. https://doi.org/10.1145/3295500.3356218

[25] Junyi Liu, John Wickerson, and George A. Constantinides. 2017. Tile Size Selection for Optimized Memory Reuse in High-level Synthesis. In *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*, Marco D. Santambrogio, Diana Göhringer, Dirk Stroobandt, Nele Mentens, and Jari Nurmi (Eds.). IEEE, 1–8. https://doi.org/10.23919/FPL.2017.8056810

[26] Sanyam Mehta, Gautham Beeraka, and Pen-Chung Yew. 2013. Tile Size Selection Revisited. *ACM Trans. Archit. Code Optim.* 10, 4 (2013), 35:1–35:27. https://doi.org/10.1145/2541228.2555292

[27] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. 2017. SymPy: Symbolic Computing in Python. *PeerJ Computer Science* 3 (Jan. 2017), e103. https://doi.org/10.7717/peerj-cs.103

[28] Auguste Olivry, Julien Langou, Louis-Noël Pouchet, P. Sadayappan, and Fabrice Rastello. 2020. Automated Derivation of Parametric Data Movement Lower Bounds for Affine Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 808—-822. https://doi.org/10.1145/3385412.3385989

[29] Nirmal Prajapati, Waruna Ranasinghe, Sanjay V. Rajopadhye, Rumen Andonov, Hristo Djidjev, and Tobias Grosser. 2017. Simple, Accurate, Analytical Time Modeling and Optimal Tile Size Selection for GPGPU Stencils. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*, Vivek Sarkar and Lawrence Rauchwerger (Eds.). ACM, 163–177. https://doi.org/10.1145/3018743.3018744

[30] Desh Ranjan, John E. Savage, and Mohammad Zubair. 2011. Strong I/O Lower Bounds for Binomial and FFT Computation Graphs. In

Computing and Combinatorics - 17th Annual International Conference, COCOON 2011, Dallas, TX, USA, August 14-16, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6842), Bin Fu and Ding-Zhu Du (Eds.). Springer, 134–145. https://doi.org/10.1007/978-3-642-22685-4_12

[31] 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. https://doi.org/10.1016/j.jda.2011.12.005

[32] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: Better, Faster, Stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 6517–6525. https://doi.org/10.1109/CVPR.2017.690

[33] Lakshminarayanan Renganarayanan and Sanjay V. Rajopadhye. 2008. Positivity, Posynomials and Tile Size Selection. In *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC)*. IEEE/ACM, 55. https://doi.org/10.1109/SC.2008.5213293

[34] John E. Savage. 1995. Extending the Hong-Kung Model to Memory Hierarchies. In *Proceedings of the First Annual International Conference on Computing and Combinatorics (COCOON '95)*. Springer-Verlag, Berlin, Heidelberg, 270–281. https://doi.org/10.1007/BFb0030842

[35] Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan, and Vivek Sarkar. 2012. Analytical Bounds for Optimal Tile Size Selection. In *Proceedings of Compiler Construction - 21st International Conference (CC) (Lecture Notes in Computer Science, Vol. 7210)*. Springer, 101–121. https://doi.org/10.1007/978-3-642-28652-0_6

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

[37] Paul Springer and Paolo Bientinesi. 2016. Design of a High-Performance GEMM-like Tensor-Tensor Multiplication. arXiv:1607.00145

[38] Sven Verdoolaege. 2010. ISL: An Integer Set Library for the Polyhedral Model. In *Mathematical Software–ICMS 2010*. 299–302.

[39] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimization* 9, 4, Article 54 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

[40] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. 2007. Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions. *Algorithmica* 48, 1 (2007), 37–66. https://doi.org/10.1007/s00453-006-1231-0

[41] Andreas Wächter and Lorenz T. Biegler. 2006. On the Implementation of an Interior-Point Filter Line-Search Algorithm for large-scale nonlinear programming. *Math. Program.* 106, 1 (2006), 25–57. https://doi.org/10.1007/s10107-004-0559-y

[42] Jingling Xue. 2000. *Loop Tiling for Parallelism*. Kluwer International Series in Engineering and Computer Science, Vol. 575. Kluwer. https://doi.org/10.1007/978-1-4615-4337-4

[43] Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E. Eichenberger, and Kevin O'Brien. 2010. Automatic Creation of Tile Size Selection Models. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. ACM, 190–199. https://doi.org/10.1145/1772954.1772982