Integrating Data Layout Transformations with the Polyhedral Model

Jun Shirako
School of Computer Science
Georgia Institute of Technology
Atlanta, Georgia, USA
shirako@gatech.edu

Vivek Sarkar
School of Computer Science
Georgia Institute of Technology
Atlanta, Georgia, USA
vsarkar@gatech.edu

Abstract

In the polyhedral model, classical loop transformations and statement reordering transformations have been unified and formalized as affine scheduling problems that can be applied to optimization goals such as locality and parallelism. More recently, data layout transformations have shown significant benefits in improving performance by contributing to improved efficiencies in spatial locality, multi-core parallelism and vector parallelism. However, integration of data layout transformations in the polyhedral model has received relatively little attention thus far.

In this paper, we report on our work-in-progress on integrating data layout transformations in the polyhedral model, with a focus on affine representations of data layout transformations. We also demonstrate the potential benefit of performing loop and data layout transformations as an integrated optimization problem, compared to standard decoupled approaches which pick a specific phase order (e.g., data layout transformations before loop transformations) and can thereby miss opportunities to co-optimize both data layout transformations and loop transformations.

Keywords Data layout transformations, loop transformations, parallelism, data locality, polyhedral model

ACM Reference Format:
Jun Shirako and Vivek Sarkar. 2019. Integrating Data Layout Transformations with the Polyhedral Model. In *Proceedings of International Workshop on Polyhedral Compilation Techniques (IMPACT'19).* ACM, New York, NY, USA, 10 pages.

1 Introduction

In recent years, a major focus of optimizing compilers is to transform the input program so as to extract the best granularity of parallelism and data locality that can fit the target architecture. Loop transformations represent a major class of program transformations that have been used to address these objectives. The primary goal of loop transformations is to reorder dynamic statement/instruction execution sequences to optimize parallelism and locality, while satisfying dependence constraints for legality. More recently, data layout transformations have also been receiving attention because of their ability to deliver significant performance improvements due to improved locality and improved support for efficient multi-core and vector parallelism.

There is a large body of past work on loop transformations since the 1980's, e.g., [3, 15, 19, 42, 43]. Syntactic/AST-based loop transformation frameworks automatically select a sequence of individual loop transformations, driven by analytical cost models, to achieve a desired optimization goal [23, 29, 41]. More recently, the polyhedral compilation model has provided significant advances in the unification of affine loop transformations combined with powerful code generation techniques [1, 5, 6, 13, 16, 32]. The benefits of this unified formulation can be seen (for example) in the PLuTo compiler [5, 6], which has been successfully extended and specialized to integrate SIMD constraints [20, 36], and the PPCG system [38, 45], which is capable of modeling the multi-level parallelism and the temporal/spatial locality of multiprocessors and accelerators, generating both OpenMP C and CUDA programs from sequential input.

Data layout transformations represent another class of program transformations that is increasingly being used in compiler optimizations. While loop transformations aim at multiple objectives, e.g., parallelism, temporal locality, and spatial locality, data layout transformations primarily impact spatial locality (which can impact both cache performance and SIMD performance in single-threaded and multithreaded executions). A number of data layout transformation techniques have been proposed, including permutation of array dimensions [18, 36], conversion between Array-of-Structs (AoS) and Struct-of-Arrays (SoA) [9, 31], data skewing, and data tiling [28]. In the polyhedral model, storage optimizations to contract memory space are generally treated as memory allocation problems, e.g., the single assignment form of a given program can be obtained via dataflow analysis [10, 11, 26], or by the nature of input language [27, 40], and iteration points in the single assignment form can be allocated to the same or different memory addresses while preserving program semantics.

There are a number of polyhedral frameworks that addressed the memory allocation problem: [7, 8, 21, 27, 40] for schedule-specific storage optimizations; and [33–35] for schedule-independent and/or unified schedule and storage

IMPACT'19, January 21–23, 2019, Valencia, Spain
2019.

optimizations. As a noteworthy recent achievement for storage optimizations, [4] proposed an intra-array storage optimization algorithm to minimize the dimensionality and storage requirements of arrays in given sequences of loop nests. Furthermore, a few polyhedral frameworks focused on selected data layout transformations, such as joint formulation for loop transformations and array dimension permutations to improve spatial locality for better vectorization efficiency [18] and data tiling combined with loop tiling to minimize inter-node communications on distributed-memory clusters [28].

Despite these important achievements and increasing attention, data layout transformations in the polyhedral model, especially the integration of loop and data layout transformations, have still received relatively little attention compared to the extensive body of literature on polyhedral loop transformations. As a first step to integrating data layout transformations in the polyhedral model, this paper discusses polyhedral extensions to support general data layout transformations including array dimensional permutation, conversion between SoA and AoS, data layout skewing, data tiling, and storage optimizations for both space reduction and parallelism enhancement. We present and discuss the advantages and disadvantages of two types of layout representations, *array-based* – where the unit of mapping/transformation is an array element – and *value-based* – where the unit is the value defined by an individual statement instance. We also demonstrate the potential benefit of integrating loop and data layout transformations as a single optimization problem over decoupled approaches, which pick a specific phase order (often by performing loop transformations before data layout transformations) and can thereby miss opportunities to co-optimize both data layout transformations and loop transformations.

The rest of the paper is organized as follows. Section 2 contains background information on the polyhedral model. Section 3 presents array-based and value-based layout transformations. Section 4 discusses the code generation for layout transformations. Section 5 demonstrate the potential benefit of integrating loop and data layout transformations. Sections 6 and 7 summarize related work and our conclusions.

2 Background

The polyhedral model is a linear algebraic representation for collections of (imperfectly) nested loops whose loop bounds and branch conditions are affine functions of outer loop iterators and runtime constants, which are handled as global parameters [12]. Code regions amenable to this algebraic representation are called *Static Control Parts* and represented in the SCoP format [24]. In this model, a statement consists of three elements: iteration domain, access relation, and schedule. A dynamic instance of a statement is identified by its statement name S and loop iteration vector $\vec{i}$, as $S(\vec{i})$.

2.1 Basic Components

Iteration domain, $\mathcal{D}_S$: The iteration domain of a statement S enclosed by m loops is represented by an m-dimensional polytope, where an element $S(\vec{i}) \in \mathcal{D}_S$ is an instance of statement S. As an example in Figure 1, the iteration domain of statement S is:

$$\mathcal{D}_S = \{S(i,j) \mid 0 \le i < \mathrm{ni} \ \wedge \ 0 \le j < \mathrm{nj}\}$$

Access relation, $\mathcal{A}_{S\to A}$: The array reference(s) to A by statement S is abstracted as an access relation, which maps a statement instance $S(\vec{i})$ to one or more array elements $A(\vec{e})$ to be read/written[1], typically as affine functions [44]. In Figure 1, the write access relation for statement S to array C is:

$$\mathcal{A}^{write}_{S\to C} = \{S(i,j) \to C(e_1,e_2) \mid i = e_1 \ \wedge \ j = e_2\}$$

Schedule, Θ_S: The sequential execution order of a program is captured by the schedule, which maps a statement instance $S(\vec{i})$ to a logical time-stamp vector, expressed as a multidimensional (quasi-)affine function of $\vec{i}$. Statement instances are executed according to the increasing lexicographic order of their time-stamps. Dimensions of schedule may contain loop iterators. A dimension is called a *loop dimension* if it contains one or more iterators; otherwise it is called *scalar dimension*. Schedules to represent the sequential execution order of statements S and T in Figure 1 are:

$$\Theta_S = \{S(i,j) \to (i,j,0)\}$$
$$\Theta_T = \{T(i,j,k) \to (i,j,1,k)\}$$

The first and second dimensions i,j of Θ_S and of Θ_T indicate that i-loop and j-loop are enclosing S and T. The 0 or 1 value in the third dimension indicates that, for the same iteration of the i,j loop nest, the instance of S is executed before any instance of T. The k of Θ_T indicates that T is enclosed in the k loop at the innermost level. While the above affine mapping is a compact representation of schedule, schedule tree [39] directly encodes inclusive relation and ordering of statements and serves as a foundation of compiler/runtime optimizations, including our data layout transformations.

2.2 Legality and Loop Transformations

As with traditional compiler optimizations, the polyhedral compilation computes dependences based on the original schedule and memory accesses, summarized as dependence polyhedra [5]. The polyhedral loop transformations amount to computing new schedules under the legality constraints of dependence polyhedra.

[1] A scalar variable is considered as a degenerate case of an array.

```
  for (i = 0; i < ni; i++) {
    for (j = 0; j < nj; j++) {
S:    C[i][j] *= beta;
      for (k = 0; k < nk; k++)
T:      C[i][j] += alpha * A[i][k] * B[j][k];
  } }
```

Figure 1. gemm input

```
#define _C(x,y) band_0[(x)/32][(y)/64][(x)%32][(y)%64]
#define _A(x,y) band_1[(y)].band_1_0[(x)]
#define _B(x,y) band_1[(x)].band_1_1[(y)]
  for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++)
S:    _C(i,j) *= beta;
  for (k = 0; k < nk; k++)
    for (i = 0; i < ni; i++)
      for (j = 0; j < nj; j++)
T:      _C(i,j) += alpha * _A(i,k) * _B(k,j)
```

Figure 2. array-based layout transformation

Dependence Polyhedra, $\mathcal{P}_{S\to T}$: The dependences between statements S and T are captured by dependence polyhedra — the subset of pairs $(S(\vec{i}), T(\vec{i})) \in \mathcal{D}_S \times \mathcal{D}_T$ that participate in a dependence. Given two statement instances $S(\vec{i})$ and $T(\vec{i})$, $T(\vec{i})$ is said to depend on $S(\vec{i})$ if: 1) they access the same array element where at least one of them is write access; and 2) $S(\vec{i})$ has a lexicographically smaller schedule value than $T(\vec{i})$ – i.e., $\Theta(S(\vec{i})) \prec \Theta(T(\vec{i}))$.

In general, any composition of iteration- and statement-reordering loop transformations (e.g., permutation, skewing, distribution, fusion, and tiling) can be specified by polyhedral schedules. To compute new schedules, polyhedral optimizers rely on integer linear programming where dependence polyhedra are used as legality constraints and their optimization goals are formulated as objectives and/or constraints.

3 Layout Representations

In this section we present two types of layout representations, array-based and value-based, and discuss their relative advantages and disadvantages.

3.1 Array-Based Layout Transformations

A straightforward approach to model data layout transformations is to consider an array element as the unit of representations and transformations. Section 2 showed how the set of statement instances is captured by iteration domains, and loop transformations are implemented by computing new schedules. We apply the same notions to the region of arrays and data layout transformations as described below.

3.1.1 Array Domain

As with statements, let $A(\vec{e})$ denote an element of array A and $\mathcal{D}_A$ as the array domain of array A. Given SCoP region, the upper/lower bound of each dimension of A is an affine function of global parameters, and hence invariant. Therefore, array domain $\mathcal{D}_A$ of m-dimensional array A is a m-dimensional rectangular solid whose dimension sizes are constant at the beginning of the SCoP region at runtime. In Figure 1, the array domains of array C, A, and B are:

$$\mathcal{D}_C = \{C(e_1,e_2) \mid 0 \le e_1 < \text{ni} \ \wedge \ 0 \le e_2 < \text{nj}\}$$
$$\mathcal{D}_A = \{A(e_1,e_2) \mid 0 \le e_1 < \text{ni} \ \wedge \ 0 \le e_2 < \text{nk}\}$$
$$\mathcal{D}_B = \{B(e_1,e_2) \mid 0 \le e_1 < \text{nk} \ \wedge \ 0 \le e_2 < \text{nj}\}$$

3.1.2 Layout Mapping

As schedule Θ specifies the relative order of statement instances in time, let layout mapping Φ denote the relative order of array elements in the transformed memory space. Φ_A is the data layout of array A and maps each element $A(\vec{e})$ to a logical address vector, expressed as a multidimensional (quasi-)affine [2] function of $\vec{e}$. There are interesting and intuitive relations between schedule map Θ_S and layout map Φ_A, such as loop permutation and array dimensional permutation; loop fusion/distribution and AoS/SoA conversion; and loop skewing/tiling and data skewing/tiling. All of these transformations can be represented and implemented via affine mappings, as with loop transformations covered by schedule. In contrast, there is a notable difference in that a schedule is unique to a given SCoP region while data layouts can, in general, be changed within a SCoP region, e.g., via data re-distribution [30].

A major advantage of array-based layout transformations is that, so long as the layout is a one-to-one function, there is no legality constraint imposed on layout mapping Φ because each array element $A(\vec{e})$ has a unique location in memory space. In contrast, array-based transformations are not amenable to one-to-many (i.e., expansion) or many-to-one (i.e., contraction) storage optimizations, which, in general, require array dataflow analysis to establish legality.

Figure 1 contains gemm as an input and Figure 2 shows the code with array-based layout transformations specified by the following mapping, which corresponds to: 1) data layout tiling applied to array C with tile sizes of 32 and 64 for the first and second dimensions respectively; 2) array dimensional permutation applied to array A; and 3) array dimensional fusion applied at the outer dimensions of arrays A and B, resulting in an Array-of-Struct-of-Array layout.

$$\Phi_C = \{C(e_1,e_2) \to (0,\ e_1/32,\ e_2/64,\ e_1\%32,\ e_2\%64)\}$$
$$\Phi_A = \{A(e_1,e_2) \to (1,e_2,0,e_1)\}$$
$$\Phi_B = \{B(e_1,e_2) \to (1,e_1,1,e_2)\}$$

In Figure 2, loop transformations (distribution and permutation) are simultaneously implemented with the following schedule. For efficient tiled data access, loop tiling should also be applied in conjunction with data tiling but we omitted that in the interest of readability.

$$\Theta_S = \{S(i,j) \to (0,i,j)\}$$
$$\Theta_T = \{T(i,j,k) \to (1,k,i,j)\}$$

```
  for (i = 1; i < ni; i++) {
    for (j = 1; j < nj; j++) {
S:    A[i][j] = C[i][j-1];
T:    B[i][j] = A[i][j] + C[i-1][j];
U:    C[i][j] = B[i][j];
  } }
```

Figure 3. Sample input

```
#define _insS(i,j) band_0
#define _insT(i,j) band_0
#define _insU(i,j) band_1[(j)]
  for (i = 1; i < ni; i++) {
    for (j = 1; j < nj; j++) {
S:    _insS(i,j) = _insU(i,j-1);
T:    _insT(i,j) = _insS(i,j) + _insU(i-1,j);
U:    _insU(i,j) = _insT(i,j);
  } }
```

Figure 4. value-based layout transformation

3.2 Value-Based Layout Transformations

Storage optimizations that contract or expand the memory space require array dataflow analysis to capture which statement instances (consumers) use the value defined by a given statement instance (producer). The layout transformations need to be performed so as to ensure legality with respect to liveness (for many-to-one layouts) and consistency (for one-to-many layouts). The notion of value-based layout transformation is fundamentally equivalent to partial data expansion [21]. In contrast to that past work, we aim to also study legality constraints in cases where the schedule is not fixed, thereby enabling exploration of an optimization space in which both the schedule and layout maps are computed simultaneously.

3.2.1 Total data expansion

As with [21] and [4], we assume the input static control program is converted into functionally equivalent single-assignment form, and each statement instance $S(\vec{i})$ has a unique memory location for its defined value – i.e., array $InsS(\vec{i})$ in [21]. We consider this location in the single-assignment form as value identifier, which is the unit of interest in value-based layout transformations.

3.2.2 Layout Mapping

Let layout mapping Φ_S denote the data layout to store the value defined by statement S in the transformed memory space. Φ_S maps each element $S(\vec{i})$ to logical address vector expressed as a multidimensional (quasi-)affine function of $\vec{i}$. Analogous to the array-based layout representation, the value-based representation is capable of modeling arbitrary layout transformations to enhance spatial data locality, e.g., array permutation, AoS/SoA conversion, data skewing, and data tiling, in addition to storage optimizations to contract/-expand memory space.

Our ultimate goal is to compute both schedule Θ and layout Φ simultaneously as a single optimization problem. The legality constraints are imposed by value-based dependences. Let $flow_k$ denote k-th dataflow, i.e., value-based Read-After-Write (RAW) dependence.

$$flow_k = \{S_k(\vec{i}) \rightarrow T_{k,1}(\vec{j_1}), ..., T_{k,n_k}(\vec{j_{n_k}})\}$$

Producer-consumer constraints: The producer of a value $S_k(\vec{i})$ must precede any consumers of the value $T_{k,1}(\vec{j_1})$, ..., $T_{k,n_k}(\vec{j_{n_k}})$.

$$\Theta(S_k(\vec{i})) \prec lex\_min(\Theta(T_{k,1}(\vec{j_1})), \ ..., \ \Theta(T_{k,n_k}(\vec{j_{n_k}})))$$

Liveness constraints: The memory location of a value must not be overwritten until the last use of the value. In other words, given two value-based RAW dependences $flow_k$ and $flow_l$, they must satisfy either of: livenesses of $flow_k$ and $flow_l$ do not overlap; or memory locations identified by $\Phi(S_k(\vec{i}))$ and $\Phi(S_l(\vec{i}))$ are different.

$$lex\_max(\Theta(T_{k,1}(\vec{j_1})), \ ..., \ \Theta(T_{k,n_k}(\vec{j_{n_k}}))) \preceq \Theta(S_l(\vec{i}))$$
$$\vee \ lex\_max(\Theta(T_{l,1}(\vec{j_1})), \ ..., \ \Theta(T_{l,n_l}(\vec{j_{n_l}}))) \preceq \Theta(S_k(\vec{i}))$$
$$\vee \ \Phi(S_k(\vec{i})) \ \neq \ \Phi(S_l(\vec{i}))$$

Figures 3 and 4 respectively show a sample input code and the code with value-based data layout transformations to contract the memory space. After converting the input program into single-assignment form (e.g, `A[i][j]` into `_instS(i,j)`), the following layout mapping is used to specify the location to store the defined values in the transformed memory space.

$$\Phi_S = \{S(i,j) \rightarrow (0)\}$$
$$\Phi_T = \{T(i,j) \rightarrow (0)\}$$
$$\Phi_U = \{U(i,j) \rightarrow (1,j)\}$$

4 Code Generation for Transformed Layout

4.1 Code-level Layout Specification

The affine mapping representations of layout transformations are straightforward to be used at the source code level. We use the following `layout` directive to specify layout transformations as an extension to the C language:

`#pragma layout map(`$transfer : map : dom$`) ...`

where one or more `map` clauses, each of which specifies the layout transformations of a single array, can be included in directive. A `map` clause contains three components: $transfer$ is the type of data transfers among original and transformed layouts; *map* is the affine mapping of array element $A(\vec{e})$ to multidimensional layout vector; and *dom* is the array domain.

There are four types of transfers: `in` – i.e. copy-in from the original to transformed layouts, `out` – i.e., copy-out from

the transformed to original layouts, `inout` – i.e., both copy-in and copy-out, and `redist` – i.e., re-distribution of data layout. The layout directive with `in`, `out`, and `inout` types may appear only at the beginning of a SCoP region while the directive with `redist` type is used in the middle of the SCoP region, to specify the point where the data stored in an old layout is transferred into the new layout[2]. We employ the iscc [37] expression for the layout mapping. Assuming the lower array bounds in C language are always 0, let $type[size_1, size_2, \ldots]$ denote the array domain of A, where *type* is the data type (e.g., int and double) and $size_i$ is the i-th dimension size represented as the affine combination of global parameters. To simplify the code generation presented in Section 4.2, *type* must be same across all `map` clauses in the current implementations.

Figure 6 shows an example corresponding to the layout mapping example from Section 3.1.2 (data tiling is omitted for simplicity). Arrays A and B are read-only (`in`) while C is read then written (`inout`).

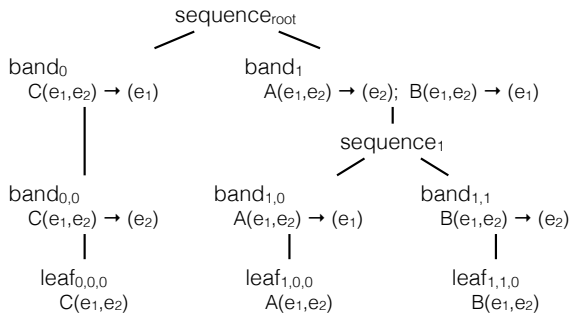

Figure 5. Schedule tree for layout of Figure 6

4.2 Code Generations for New Layout

Once new layouts are computed in IR-level by compilers or specified in source-level by users, the next challenge is how to implement the new layouts in the output code. This section presents two approaches to implement the data layout transformations specified as array domain and layout mapping. The same approaches are used for value-based layout transformations after converting input code into the single-assignment form as discussed in Section 3.2.1. For the data layout code generation, we first convert the multidimensional form of layout mapping into schedule tree [39], which is more straightforward form to capture the arbitrary nested structures of data layouts. Figure 5 shows the schedule tree representation of the layout mapping in Figure 6. We use this layout mapping as the running example in this section.

4.3 Linearized field

The first approach is to map the schedule tree form of layout mapping into a one-dimensional memory field. In order to compute the total field size and the offset to individual array element in the linearized field, we first compute the size of tree node in the following manner.

- Size of sequence node is the total size of child nodes.

$$size(sequence_k) = \sum_{i=0}^{\#children} size(child\_node_{k,i})$$

- Size of band node is its child node size × dimension length, which is given by the maximum value of the dimension with optional padding ($pad_k \geq 1$).

$$size(band_k) = length_k \times size(child\_node_{k(,0)})$$

$$length_k = max(range(band_k)) + pad_k$$

- Size of leaf node is 1 if all arrays are same type. Otherwise leaf size corresponds to the data type of the array.

$$size(leaf_k) = 1 \ (or \ \texttt{sizeof}(type))$$

In the code generation phase, the schedule tree is traversed in depth-first order and the expressions of dimension lengths and node sizes are defined respectively, according to the above rules (Figure 7).

The total field size is equivalent to the root node of the schedule tree, e.g., $size(sequence_{root})$ in the running example. The offset to array element $A(\vec{e})$ is equivalent to the linearized form of layout mapping Φ_A and computed as the summation of contributions from individual dimensions: a) in case of scalar dimension, its contribution is the total size of preceding nodes in the schedule tree; and b) in case of loop dimension, its contribution is the child node size × the expression of the loop dimension, e.g., `e2` and `e1` for the first and second loop dimensions of array `A` (Figure 6). The code generation phase also defines the expressions of total field size and offsets, and inserts the statement to allocate the linearized array field and macros to access individual array elements.

Finally, the copy-in/out data transfers between original and transformed layouts, as with the re-distribution of transformed layouts, are simply implemented as assignments using the access macros, e.g., the doubly nested loop to copy the original `A` into the transformed layout via macro `_A(e1,e2)` at the bottom of Figure 7. Also, the loop code generation phase uses the access macros for each array reference.

4.4 Nested Structures

The second approach is to allocate the C structs that correspond to the schedule tree. Although this is even more straightforward than the first approach, the resulting structure is not guaranteed to strictly preserve the relative order of the layout mapping. We first determine the data type of each node in the following manner.

[2]This point must be at the top-level, i.e., not enclosed in any loops.

```
#pragma layout map(inout: C(e1,e2) -> (0, e1, e2) :    double[ni, nj]) \
               map(in:    A(e1,e2) -> (1, e2, 0, e1) : double[ni, nk]) \
               map(in:    B(e1,e2) -> (1, e1, 1, e2) : double[nk, nj])
```

Figure 6. Layout directive to specify layout transformations in Section 3.1.2 (data tiling is omitted for simplicity)

```
// Dimension length
int len_0_0 = nj + padding;
int len_0 = ni;
int len_1_0 = ni + padding;
int len_1_1 = nj + padding;
int len_1 = max(nk, nk);

// Node size definition
int band_0_0 = len_0_0 * 1;
int band_0 = len_0 * band_0_0;
int band_1_0 = len_1_0 * 1;
int band_1_1 = len_1_1 * 1;
int seq_1 = band_1_0 + band_1_1;
int band_1 = len_1 * seq_1;
int seq_root = band_0 + band_1;

// Allocation for new layout
double *field = malloc(seq_root * sizeof(double));

// Accesses for new layout
#define _C(e1,e2) field[0 + (e1) * band_0_0 + (e2)]
#define _A(e1,e2) field[band_0 + (e2) * seq_1 + 0 + (e1)]
#define _B(e1,e2) field[band_0 + (e1) * seq_1 + band_1_0 + (e2)]

// Data transfer
for (int e1 = 0; e1 < ni; e1++)
  for (int e2 = 0; e2 < nj; e2++)
    _A(e1,e2) = A[e1][e2];
...
```

Figure 7. Codegen via linearized field

```
// Dimension length
... (same as linearized field)

// Node struct definition
typedef struct _seq_1 {
  double *band_1_0, *band_1_1;
} Seq_1;

// Allocation for new layout
int szd = sizeof(double);
double **band_0 = malloc(len_0 * sizeof(double*));
band_0[0] = malloc(len_0 * len_0_0 * szd);
for(i = 0; i < len_0; i++) {
  band_0[i] = (*band_0 + len_0_0 * i);
}
Seq_1 *band_1 = malloc(len_1 * sizeof(Seq_1));
for (i = 0; i < len_1; i++) {
  band_1[i].band_1_0 = malloc(len_1_0 * szd);
  band_1[i].band_1_1 = malloc(len_1_1 * szd);
}

// Accesses for new layout
#define _C(e1,e2) band_0[(e1)][(e2)]
#define _A(e1,e2) band_1[(e2)].band_1_0[(e1)]
#define _B(e1,e2) band_1[(e1)].band_1_1[(e2)]

// Data transfer
... (same as linearized field)
```

Figure 8. Codegen via nested structs

- The data type of sequence node is defined as a struct whose members correspond to child nodes of the sequence node.

  ```
  typedef struct _seq_k {
    type_k_0 child_k_0;  type_k_1 child_k_1; ...
  } Seq_k;
  ```

- The data type of band node is the pointer type of the child node so as to allocate an array of child node type.

  ```
  type_k_0 *band_k;
  band_k = malloc(len_k * sizeof(type_k_0));
  ```

- The data type of leaf node is the array's data type.

In the code generation phase, the schedule tree is traversed in depth-first order and the data type of each node is determined. The structs of sequence nodes except for the root node are explicitly defined in the output code (Figure 8).

The kind of root node can be either of band or sequence. If the root is a band node, the code generation phase declares the variable corresponding to the root band node. Otherwise, it declares the variable(s) corresponding to the members of the root sequence node, e.g., "`double **band_0`" and "`Seq_1 *band_1`" in Figure 8. The arrays for band nodes are recursively allocated in the nested structure. The macro to access each array element is simply computed from the root-to-leaf path in the schedule tree. In our running example, the pass from $sequence_{root}$ to $leaf_{1,0,0}$ gives the access macro: `#define _A(e1,e2) band_1[(e2)].band_1_0[(e1)]`.

5 Potential Impact of Integrating Loop and Data Layout Transformations

We use the generalized matrix-multiply (gemm) from PolyBench 4.2 [25] along with the default benchmark dataset, to motivate the potential impact of integrating loop and data layout transformations in a single optimization problem. As with many of other PolyBench benchmarks, gemm has high computational intensity, i.e., 2-dimensional arrays were accessed in 3-dimensional loop nests. The execution time measured in the following experiments includes 2-D arrays' data copy-in and copy-out overhead, which is indeed ignorable on all the tested systems. We observed that 22 of 29 PolyBench benchmarks are also computationally intensive, i.e., n-dimensional arrays in m-dimensional loop nests where $n < m$, and can be good candidates for the performance study of the data layout transformations.

Three Linux-based SMP systems are used: a 12-core (dual 6-core) 2.8GHz Intel Xeon Westmere, a 12-core 2.1GHz Broadwell, and a 24-core (dual 12-core) 3.0GHz IBM POWER8. On Xeon, all experimental variants were compiled using the Intel C/C++ compiler v15.0 (Westmere) and v17.0 (Broadwell) with the "-O3 -xHOST" options for sequential runs

```
#pragma omp parallel for ...
  for (i = 0; i < ni; i++) {
    for (j = 0; j < nj; j++)
S:    C[i][j] *= beta;
      for (k = 0; k < nk; k++) {
T:      C[i][j] += alpha
         * A[i][k] * B[j][k];
  } }
```

Figure 9. gemm using PLuTo (minimum reuse distance schedule) + manual best layout search (which resulted in transposing the dimensions of array B)

```
#pragma omp parallel for ...
  for (i = 0; i < ni; i++) {
    for (j = 0; j < nj; j++)
S:    C[i][j] *= beta;
    for (k = 0; k < nk; k++)
      for (j = 0; j < nj; j++)
T:      C[i][j] += alpha
         * A[i][k] * B[k][j];
  }
```

Figure 10. gemm using PolyAST + manual best layout search (which resulted in no change to the original data layout)

```
#pragma omp parallel for private(j)
  for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++)
S:    C[i][j] *= beta;
#pragma omp parallel for private(i, j) \
            reduction(+: C[0:ni][0:nj])
  for (k = 0; k < nk; k++)
    for (i = 0; i < ni; i++)
      for (j = 0; j < nj; j++)
T:      C[i][j] += alpha
         * A[k][i] * B[k][j];
```

Figure 11. gemm using our framework (which resulted in transposing the dimensions of array A, and also a different loop transformation)

and the "-O3 -xHOST -openmp" options for the output from automatic parallelization. On POWER8, all variants were compiled using the IBM XL C/C++ compiler 13.1 with the "xlc -O5" command for sequential runs and the "xlc_r -O5 -qsmp=omp" command for the output from automatic parallelization by PLuTo, PolyAST, and our framework.

Figure 9 is the output of the PLuTo polyhedral compiler [5, 6], whose objective function is to minimize the temporal distance between two accesses to the same memory location, i.e., *minimum reuse distance*. This is also a core objective of many polyhedral optimizers to achieve maximal temporal data locality and outermost forall parallelism. As an example of the phase-ordered approach, after loop transformations we manually tried all possible ($2^3 = 8$) array dimensional permutation candidates on three platforms, and found the best layout: permuting 2-D array `B` so that the original `B[k][j]` was changed into `B[j][k]` with better spatial locality. As discussed later, the original `B[k][j]` layout was selected when enabling intra-tile permutation by PLuTo.

We also applied PolyAST [32], a hybrid framework to integrate polyhedral and AST-based loop transformations, and obtained the output shown in Figure 10. PolyAST employs analytical memory cost models, which guide loop transformations to implement better temporal and spatial data locality on the given data layout. As a result, the best layout found by manual array dimensional permutation on three platforms is same as the original layout. Note that we omit loop tiling in Figures 9–11 for readability, though tiling was performed to obtain the performance numbers in Table 1.

Figure 11 is the output of our on-going work, cost-based iterative compilation to integrate data layout and loop transformations. This approach first generates candidate layouts Φ, which are all 8 combinations of array permutation in this example, and applies PolyAST transformations to all candidates and generates corresponding schedules Θ. Finally, it selects the pair of Φ and Θ with the minimum estimated cost as the final output. Our framework selected very different loop transformations from others, e.g., statements S and T are completely distributed and the k-loop is parallelized as an array reduction for statement T. Integrated with these loop transformations, the data layout selected by our framework is: permuting 2-D array `A` so that the original `A[i][k]` reference was changed to `A[k][i]`. Although the reduction parallelism needs extra overhead to compute the final values, our cost analysis detected that the overhead is sufficiently small and more than overcome by the benefits of reduced memory cost. The selected data layout minimizes the array read costs on `A` and `B`, while the partial sum of array reduction on `C` is accumulated into thread-local storage, thereby incurring no inter-thread communications except the final sum.

Table 1 shows the speedups of the all experimental variants with tiling, relative to the original program. It clearly shows the effectiveness of the integrated loop and data layout transformations over the phase-ordered strategies using PLuTo and PolyAST polyhedral loop optimizers. The 'PLuTo' and 'Min dist + layout' variants have the same inter-tile loop structures as shown in Figure 9, while PLuTo additionally permutes intra-tile loops to enhance vectorizations. Because of the intra-tile loop permutation to locate j-loop innermost, the original layout `B[k][j]` was selected as the best for the PLuTo variant. To summarize, proper integration of data layout and loop transformations can find the optimal solution which may lie in a space not covered by phase-ordered approaches.

	12-core Westmere	12-core Broadwell	24-core POWER8
PLuTo + layout	4.96×	4.44×	9.62×
Min dist + layout	5.09×	4.01×	8.62×
PolyAST + layout	5.72×	4.48×	11.58×
Integrated	7.50×	4.97×	14.96×

Table 1. Speedup over the original sequential. The manual best layout search selected the original layouts for PLuTo and PolyAST respectively due to intra-tile loop permutation and temporal/spatial locality-aware affine scheduling.

6 Related Work

There is an extensive body of literature on loop and data layout transformations. In this section, we focus on past contributions that are most closely related to this paper.

[18] proposed a combined loop and data layout transformation framework for improving the cache performance of sequential codes. They first apply loop transformations to optimize the locality of a given loop nest, and then apply data transformations for arrays for which the array references do not exhibit good spatial locality after the loop transformations, i.e., loop-first approach.

[40] first tackled the memory allocation problem for polyhedral programs in the context of the ALPHA language, where variables have single assignment form by nature. Given a schedule, they gave necessary and sufficient conditions for the legality of a memory allocation.

[33] addressed the problem of finding memory allocations for a perfect loop nest with stencil-like uniform dependences. They proposed universal occupancy vector, which provides a schedule-independent storage reuse pattern and enables storage reduction while keeping the flexibility in loop scheduling.

[34, 35] was the first to address the unification of affine scheduling and storage optimization with consideration for the optimal tradeoff between parallelism and storage space. They proposed a mathematical framework that unifies the techniques of one-dimensional affine scheduling and occupancy vector analysis, which determines a good storage mapping for a given schedule, a good schedule for a given storage mapping, and a good storage mapping that is valid across a range of schedules.

[27] provided constructive algorithms for asymptotically optimal memory allocation functions for a given schedule. This work builds on top of [40] and the analysis targets static control flow programs in a single assignment form.

[7, 8] presented lattice based memory allocation for schedule-specific storage optimization in the context of the polyhedral model. They formulated modular memory allocations in terms of integer lattices, where the basis of the lattice is akin to a set of occupancy vectors that are applied simultaneously. The size of the allocated storage is equivalent to the determinant of the lattice, and they proposed several heuristics for minimizing this quantity.

[22] proposed a temporal and spatial data locality optimization framework of nested loops, where unimodular transformations are used to implement good temporal locality while aggressive data layout transformations with special attention on TLB effectiveness are used to improve spatial locality. Their data layout transformation would be closely related to the array-based layout transformation presented in Section 3.1. An important difference is that their approach aims to implement contiguous memory access for TLB efficiency and represents transformed layouts in one-dimensional polynomial functions (e.g., $i \times (i-1)$) while we are interested in general layout transformations that can be represented as multi-dimensional (quasi-)affine functions.

[36] proposed a joint formulation for loop transformations and data layout permutations based on the optimization algorithms of the R-Stream polyhedral compiler. The proposed extension allows additional flexibility of permuting arrays per statement, aiming at improved spatial locality for better vectorization efficiency. The data layout transformations considered with scheduling have so far been limited to permutations and the scheduling problem requires many boolean decision variables, which can certainly be improved in their future work.

[14] formulated the memory stream alignment problem, a fundamental performance bottleneck for stencil computations on short-vector SIMD architectures, and develop an approach to overcoming the problem via data layout transformations for improved vectorization.

[31] proposed an automatic data layout selection algorithm built on a source-to-source layout transformation tool. Given an input program and target machine specification, this approach recommend a good SoA/AoS data layout. [9] introduced selection of optimized SoA/AoS generation for CPU+GPU hybrid architectures. Neither [31] nor [9] included loop transformations in the scope of their work.

[17] presented the design and evaluation of Brainy, a program analysis tool for optimized data structure selection based on dynamic profiling. Given program, inputs, and target architecture, it generates machine-learning based models to predict the best data structure implementation. Loop transformations were not included in the scope of this work.

[28] addressed the minimization of inter-node communications on distributed-memory clusters by combining data tiling transformations with loop tiling. On top of polyhedral loop transformations to enable locality optimizations (including tiling), they successfully introduced additional constraints for data tiling to minimize communications.

7 Conclusions

In this paper, we introduced our on-going study on integrating data layout transformations in the polyhedral model. We presented two types of layout representations, array-based and value-based, and discussed the advantages and disadvantages of both representations.

The array-based layout transformation is amenable to spatial data locality optimizations without imposing additional legality constraints while not straightforward to implement storage contraction and expansion. In contrast, value-based layout transformation is capable of enabling a broader range of transformations including storage contraction/expansion while it affects legality constraints and brings new challenges to compute both schedule and layout simultaneously.

As an optimization framework using the array-based layout representations, we are developing a cost-based iterative compilation algorithm to find the best pair of loop and data layout transformations. The potential benefit of this integrated approach was demonstrated on three SMP systems using GEMM in this paper. Integrating the value-based layout transformation, whose effectiveness was demonstrated by a wide range of past researches, in our optimization framework is another important direction of the future work.

Acknowledgments

We are very thankful to the IMPACT Program Committees for their detailed and positive feedback on this paper.

References

[1] The Polyhedral Compiler Collection. http://www.cs.ucla.edu/~pouchet/software/pocc/.

[2] Polyhedral Extraction Tool.

[3] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '84, pages 233–246, New York, NY, USA, 1984. ACM. ISBN 0-89791-139-3. doi: 10.1145/502874.502897. URL http://doi.acm.org/10.1145/502874.502897.

[4] S. G. Bhaskaracharya, U. Bondhugula, and A. Cohen. Automatic storage optimization for arrays. *ACM Trans. Program. Lang. Syst.*, 38(3): 11:1–11:23, Apr. 2016. ISSN 0164-0925. doi: 10.1145/2845078. URL http://doi.acm.org/10.1145/2845078.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proc. of PLDI '08*, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375595. URL http://doi.acm.org/10.1145/1375581.1375595.

[6] U. Bondhugula, A. Acharya, and A. Cohen. The pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests. *ACM Trans. Program. Lang. Syst.*, 38(3):12:1–12:32, Apr. 2016. ISSN 0164-0925. doi: 10.1145/2896389. URL http://doi.acm.org/10.1145/2896389.

[7] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. *IEEE Trans. Computers*, 54(10):1242–1257, 2005. doi: 10.1109/TC.2005.167. URL https://doi.org/10.1109/TC.2005.167.

[8] A. Darte, A. Isoard, and T. Yuki. Extended lattice-based memory allocation. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 218–228, 2016. doi: 10.1145/2892208.2892213. URL https://doi.org/10.1145/2892208.2892213.

[9] R. B. Deepak Majeti, Kuldeep S. Meel and V. Sarkar. Automatic Data Layout Generation and Kernel Mapping for CPU+GPU Architectures. In *25th International Conference on Compiler Construction*, Mar 2016.

[10] P. Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing*, pages 429–441, 1988.

[11] P. Feautrier. Dataflow analysis of scalar and array references. 20(1): 23–53, Feb. 1991.

[12] P. Feautrier and C. Lengauer. Polyhedron Model. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1581–1592. Springer US, 2011. ISBN 978-0-387-09765-7 978-0-387-09766-4.

[13] T. Grosser, A. Größlinger, and C. Lengauer. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 22(4), 2012. URL http://dblp.uni-trier.de/db/journals/ppl/ppl22.html#GrosserGL12.

[14] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 225–245, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19860-1. URL http://dl.acm.org/citation.cfm?id=1987237.1987255.

[15] F. Irigoin and R. Triolet. Supernode Partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 319–329, New York, NY, USA, 1988. ACM. ISBN 978-0-89791-252-5. doi: 10.1145/73560.73588.

[16] ISL. Integer set library. http://isl.gforge.inria.fr.

[17] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: Effective selection of data structures. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 86–97, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993509. URL http://doi.acm.org/10.1145/1993498.1993509.

[18] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, pages 285–297, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press. ISBN 1-58113-016-3. URL http://dl.acm.org/citation.cfm?id=290940.290999.

[19] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, pages 301–320, 1993.

[20] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When Polyhedral Transformations Meet SIMD Code Generation. volume 48, pages 127–138, New York, NY, USA, June 2013. ACM. doi: 10.1145/2499370.2462187. URL http://doi.acm.org/10.1145/2499370.2462187.

[21] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Comput.*, 24(3-4):649–671, May 1998. ISSN 0167-8191. doi: 10.1016/S0167-8191(98)00029-5. URL http://dx.doi.org/10.1016/S0167-8191(98)00029-5.

[22] V. Loechner, B. Meister, and P. Clauss. Precise datalocality optimization of nested loops. *The Journal of Supercomputing*, 21(1):37–76, 2002.

[23] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving Data Locality with Loop Transformations. 18(4):424–453, July 1996. ISSN 0164-0925. doi: 10.1145/233561.233564.

[24] OpenScop. Openscop specification and library. http://icps.u-strasbg.fr/ bastoul/development/openscop/.

[25] PolyBench. The polyhedral benchmark suite. http://www.cse.ohio-state.edu/~pouchet/software/polybench/.

[26] W. Pugh and D. Wonnacott. An evaluation of exact methods for analysis of value-based array data dependences. In *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, pages 546–566. Springer-Verlag LNCS 768, Aug. 1993.

[27] F. Quilleré and S. V. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Trans. Program. Lang. Syst.*, 22(5):773–815, 2000. doi: 10.1145/365151.365152. URL https://doi.org/10.1145/365151.365152.

[28] C. Reddy and U. Bondhugula. Effective automatic computation placement and data allocation for parallelization of regular programs. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 13–22, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2642-1. doi: 10.1145/2597652.2597673. URL http://doi.acm.org/10.1145/2597652.2597673.

[29] V. Sarkar. Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers. *IBM J. Res. & Dev.*, 41(3), May 1997.

[30] V. Sarkar and L. A. Vazquez. Automatic localization for distributed-memory multiprocessors using a shared-memory compilation framework. In *Proc. of the Twenty-Seventh Hawaii International Conference on System Sciences*, Jan 1994.

[31] K. Sharma, I. Karlin, J. Keasler, J. R. McGraw, and V. Sarkar. Data layout optimization for portable performance. In *Euro-Par 2015: Parallel Processing*, pages 250–262. Springer, 2015.

[32] J. Shirako, L.-N. Pouchet, and V. Sarkar. Oil and water can mix: An integration of polyhedral and ast-based transformations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 287–298, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-5500-8. doi: 10.1109/SC.2014.29. URL http://dx.doi.org/10.1109/SC.2014.29.

[33] M. M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independent storage mapping for loops. In *ASPLOS-VIII Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 3-7, 1998.*, pages 24–33, 1998. doi: 10.1145/291069.291015. URL https://doi.org/10.1145/291069.291015.

[34] W. Thies, F. Vivien, J. Sheldon, and S. P. Amarasinghe. A unified framework for schedule and storage optimization. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 232–242, 2001. doi: 10.1145/378795.378852. URL https://doi.org/10.1145/378795.378852.

[35] W. Thies, F. Vivien, and S. P. Amarasinghe. A step towards unifying schedule and storage optimization. *ACM Trans. Program. Lang. Syst.*, 29(6):34, 2007. doi: 10.1145/1286821.1286825. URL https://doi.org/10.1145/1286821.1286825.

[36] N. Vasilache, B. Meister, M. Baskaran, and R. Lethin. Joint scheduling and layout optimization to enable multi- level vectorization. In *IMPACT-2: 2nd International Workshop on Polyhedral Compilation Techniques, Paris, France, January*, Paris, France, Jan 2012. URL https://www.researchgate.net/publication/230759922_Joint_Scheduling_and_Layout_Optimization_to_Enable_Multi-Level_Vectorization?ev=prf_pub.

[37] S. Verdoolaege. Counting Affine Calculator and Applications. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, Apr. 2011. doi: 10.13140/RG.2.1.2959.5601.

[38] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, Jan. 2013. ISSN 1544-3566. doi: 10.1145/2400682.2400713. URL http://doi.acm.org/10.1145/2400682.2400713.

[39] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen. Schedule Trees. In *IMPACT - 4th Workshop on Polyhedral Compilation Techniques, associated with HiPEAC*, Vienna, Austria, Jan. 2014. ACM. URL https://hal.inria.fr/hal-00911894.

[40] D. Wilde and S. V. Rajopadhye. Memory reuse analysis in the polyhedral model. In *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume I*, pages 389–397, 1996. doi: 10.1007/3-540-61626-8_51. URL https://doi.org/10.1007/3-540-61626-8_51.

[41] M. Wolf, D. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 274–286, 1996.

[42] M. Wolfe. Loop skewing: The wavefront method revisited. *Int. J. Parallel Program.*, 15(4):279–293, Oct. 1986. ISSN 0885-7458. doi: 10.1007/BF01407876. URL http://dx.doi.org/10.1007/BF01407876.

[43] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics. ISBN 0-89871-228-9. URL http://dl.acm.org/citation.cfm?id=645818.669220.

[44] D. G. Wonnacott. *Constraint-based Array Dependence Analysis*. PhD thesis, College Park, MD, USA, 1995. UMI Order No. GAX96-22167.

[45] O. Zinenko, S. Verdoolaege, C. Reddy, J. Shirako, T. Grosser, V. Sarkar, and A. Cohen. Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 3–13, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5644-2. doi: 10.1145/3178372.3179507. URL http://doi.acm.org/10.1145/3178372.3179507.