Maximal Atomic irRedundant Sets: a Usage-based Dataflow Partitioning Algorithm

Corentin Ferry
Univ Rennes, CNRS, Inria, IRISA
Rennes, France
cferry@mail.colostate.edu

Steven Derrien
Univ Rennes, CNRS, Inria, IRISA
Rennes, France

Sanjay Rajopadhye
Colorado State University
Fort Collins, CO, USA

Abstract

Programs admitting a polyhedral representation can be transformed in many ways for locality and parallelism, notably loop tiling. Data flow analysis can then compute dependence relations between iterations and between tiles. When tiling is applied, certain iteration-wise dependences cross tile boundaries, creating the need for inter-tile data communication. Previous work [1, 5] computes it as the flow-in and flow-out sets of iteration tiles.

In this paper, we propose a partitioning of the flow-out of a tile into the maximal sets of iterations that are entirely consumed and incur no redundant storage or transfer. The computation is described as an algorithm and performed on a selection of polyhedral programs. We then suggest possible applications of this decomposition in compression and memory allocation.

Keywords: tiling, dependences, flow-out, flow-in, polyhedral, dataflow, redundancy

ACM Reference Format:
Corentin Ferry, Steven Derrien, and Sanjay Rajopadhye. . Maximal Atomic irRedundant Sets: a Usage-based Dataflow Partitioning Algorithm. In *Proceedings of 13th International Workshop on Polyhedral Compilation Techniques (IMPACT'23).* ACM, New York, NY, USA, 9 pages.

1 Introduction

A historical and core usage of computers is the acceleration of computations. The continued demand for precision and speed in signal and data processing algorithms has prompted performance engineers to develop carefully tuned programs for platforms like graphics processors (GPUs), and even domain-specific hardware accelerators (e.g., FPGAs).

The growth of computation volume and complexity pose multiple challenges to application developers: because the computing platforms are massively parallel at multiple levels (nodes, cores, threads, vectors), they must extract enough parallelism from their programs to use all parallelism the platform provides, without over-constraining the memory hierarchy in bandwidth and capacity.

In practice, the parallelism actually used is often limited by the data movement between CPU cores, GPU threads, nodes, as the data goes through complex memory hierarchies that introduce latency, resulting in processor stalls waiting for data.

Program optimization techniques tend to be made available to developers as automatic tools. Certain classes of programs can be analyzed and transformed automatically using *polyhedral* analysis and transformations [6–8, 12–15].

Polyhedral analysis is used to transform programs to improve both parallelism and memory access locality (Bondhugula et al., 2008 [3, 4]). Beyond these transformations, prior work uses polyhedral analyses to reduce the volume of communicated data (Bondhugula, Dathathri et al. [1, 5]).

In this paper, we seek to devise optimal sets of data that are communicated between tiles of a polyhedral program, with a strict condition to not allow *redundancy*, both in terms of write (no data is written more than once into memory) and read (no unused data is read from memory).

This paper is organized as follows: Section 2 introduces the concepts used to construct our sets; Section 3 gives a view of other work related to data movement; Section 4 describes the MARS sets and gives their construction procedure; Section 5 provides examples of constructed sets and analyses them; Section 6 gives possible applications of our work.

2 Background

Our work relies on a large stack of polyhedral techniques, ranging from analysers to schedulers, and in particular relies on loop tiling. This section gives a quick glance at the most important techniques we rely on.

2.1 Polyhedral model

The control flow of part of a program is defined by control structures (e.g., loops and guards), the conditions or bounds of which can be affine. If this is the case, then this control flow admits a *polyhedral representation*: it is possible to semantically equivalently represent these loops by an *iteration space* (the set of all iterations described by the loops and guards) and a *schedule*, respectively as a set of integer points with affine bounds, and an affine map from the iteration space to a multi-dimensional schedule space.

Likewise, memory accesses performed inside the loop nest which access function is an affine function of the surrounding iterators can be represented as affine maps from the iteration space to a collection of *data spaces*.

IMPACT'23, January 16, 2023, Toulouse, France
.

A *polyhedral model* of an imperative program comprises at least the iteration space, schedule and memory access information.

2.2 Dataflow analysis

A polyhedral model of an imperative program bears read and write access functions, from which it is possible to compute an iteration-to-iteration dependence pattern. Dataflow analysis algorithms [6] compute this pattern.

The output of dataflow analysis is a dependence graph, which can be expressed as a *polyhedral reduced dependence graph* (PRDG).

2.3 Loop tiling

In high-performance computing applications, data spaces are too large to fit in a single level of local memory or cache. Therefore, the iteration space is transformed into similarly-shaped, atomic blocks such that the memory footprint of each block fits in a certain level of cache or local memory. This operation is called *loop tiling* [9, 18].

Loop tiling consists in breaking the iteration space into *tiles* using families of hyperplanes. Every tiling hyperplane H_i is such that all dependence vectors that are not parallel to it, "traverse" it in the same direction, and there is no dependence going in the opposite direction. This means that the scalar product of all dependences against the normal vector to H_i must have the same sign.

A family of tiling hyperplanes is obtained by repeatedly translating the same hyperplane by the same amount to make a "periodic" tiling. This way, the entire iteration space can be split into similarly-shaped tiles. The distance between two consecutive hyperplanes in a family is the *tile size*. One can note that all hyperplanes in a tiling hyperplane family have the same normal vectors.

In Figure 2, there are two families of hyperplanes: $\{i = 5k : k \in \mathbf{Z}\}$ and $\{i + j = 5k : k \in \mathbf{Z}\}$.

In the rest of this paper, we assume the reader is familiar with loop tiling, and we assume some tiling has been applied to all the programs considered. We will then focus on inter-tile communications.

3 Related Work

The flow-in and flow-out sets have been extensively studied along with a good amount of breakup scenarios by Datharthri et al. (2013) [5] and Bondhugula (2013) [1]. We are focusing on one special case along the lines of the work of Datharthri et al., with a constraint that no point may belong to two communication sets at the same time.

A decomposition of the communicated sets of data may be used for inter-node message passing in MPI to reduce the amount of traffic. Zhao et al. [19] perform a decomposition of the data space of stencils into coarse blocks such that fetch and write operations of each block are contiguous, and blocks are laid out according to the consuming neighbors so that a series of blocks is retrieved in one contiguous message. A supporting graph data structure provides the addresses for each of the blocks. This work seeks an optimal memory layout in terms of number of communications, and does so without the flow-in and flow-out sets or a polyhedral representation. Our work generalizes the idea using the polyhedral framework.

4 MARS: Maximal Atomic irRedundant Sets

This section presents the Maximal Atomic irRedundant Sets (MARS). It is laid out as follows: first, we give a definition of MARS and properties they satisfy. Then, we introduce an algorithm to construct these sets along with an example.

4.1 Notations and hypotheses

To compute the MARS, we need a program with a polyhedral model, to which the tiling transformation is legal along given hyperplanes. We restrict ourselves to the case where the dependences are uniform, and therefore we can consider individual *dependence vectors*. The uniformity of the dependence pattern guarantees that, assuming an infinite iteration space, all tiles of the same shape feature the same MARS. Tile sizes are assumed to be constant, but they can be made runtime parameters using the idea from [16].

We also assume that each statement writes to a single memory location. Therefore, we can interchangeably use an iteration and the value it produces (data).

Additionally, we will use the following notations:

- $\mathcal{D}$ designates the iteration space, of dimension N.
- There are T tiling hyperplane families, H_i for $i \in \{1, \ldots, T\}$. We will
- There are D dependence vectors, $\vec{b}_j$ for $j \in \{1, \ldots, D\}$. The PRDG is the set of all dependence vectors.
- The *non-trivial parts* of a set E are all the non-empty subsets of E. It is noted $\mathcal{P}_n(E)$. For instance:

$$\mathcal{P}_n(\{1,2,3\}) = \{\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$$

- The modulo operator is noted $a \bmod b$ and congruences are noted $a \equiv r\,[b]$.

4.2 Definition

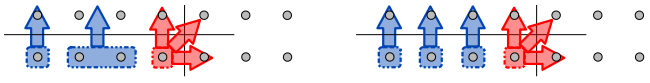
Maximal Atomic irRedundant Sets (MARS) are defined as the maximal sets of iterations that satisfy the following property:

All-consumed per tile (ACT): let S be a set of iterations. If a tile consumes data produced by an iteration $\vec{x} \in S$, then it consumes all the data produced by the same tile as $\vec{x}$.

Figure 1 shows this property is not sufficient for the sets to be unique. However, the maximal such sets are unique.

4.3 Computation

The all-consumed property stated above is equivalent to saying that all data inside a MARS is consumed by exactly the



(a) (b)

Figure 1. Sets of iterations for Smith-Waterman matching (ACT) that are not MARS. In both cases, the iterations used only by the above tile should be merged into a single set to honor (ACT).

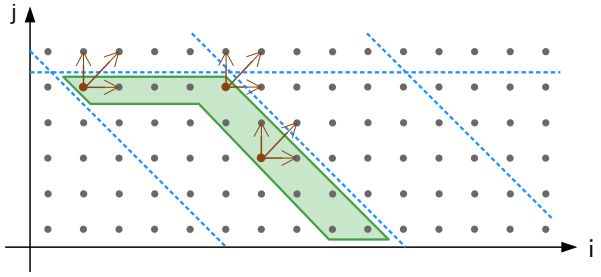


Figure 2. Flow-out set for a skewed tile of a Smith-Waterman kernel. The arrows correspond to the dependence pattern (PRDG).

same tiles. Therefore, if two distinct tiles consume a MARS M, then given the all-consumed property, both consumer tiles use all of the points of M.

We propose a construction by breaking up the *flow-out set* of each tile. There is an equivalent construction with the *flow-in set* of each tile, and both constructions lead to the maximal sets that respect the (ACT) property.

4.3.1 Flow-out set.

We start by introducing the flow-out set, with a different view than [1]. The flow-out set of a tile is defined as all the iterations which have at least one consumer iteration outside the tile. An example is given in Figure 2 for a Smith-Waterman kernel with skewed tiles. Notably, we see that despite the dependences being all unit or null along each axis, the "thickness" of the flow-out may be greater than one (in this case, with the diagonal dependence).

The tile-wise flow-out can be expressed as follows: given a tile S,

$$\varphi_O(S) = \left\{\vec{x} \in S : \vec{x} + \vec{b}_1 \in \mathcal{D}\backslash S \vee \cdots \vee \vec{x} + \vec{b}_D \in \mathcal{D}\backslash S\right\}$$

However, this formulation is missing information on the tile the dependence vectors lead to. We therefore introduce a finer formulation with the individual contribution of each dependence that traverses a tiling hyperplane, i.e. that crosses tile boundaries. This only requires knowledge of the PRDG (dependence vectors) alongside the tiling hyperplanes and the domain, and can be done as in Algorithm 1.

Algorithm 1: Computing the flow-out set using contributions from each dependence

$\mathcal{D}$ = iteration space,

Input: $B = \left\{\vec{b}_i : i = 1, \ldots, D\right\}$ = PRDG,
$\mathcal{H} = \{H_i : i = 1, \ldots, T\}$ = tiling hyperplane families
Result: φ_O = Flow-out set
for $H \in \mathcal{H}$ **do**
  $\vec{c} = (c_i)_{i=1,\ldots,N}$ normal vector to H;
  $s =$ tile size for hyperplane family H;
  for $\vec{b} \in B$ **do**
    `// Flow-out iterations for dependence` $\vec{b}$ `crossing a hyperplane of` H
    $m = \vec{c} \cdot \vec{b}$;
    $F_{H,\vec{b}} = \left\{\vec{x} = (x_i)_{i=1,\ldots,N} \in \mathcal{D} : s - m \leqslant (\sum_i (c_i x_i)) \bmod s < s \wedge \vec{x} + \vec{b} \in \mathcal{D}\right\}$;
  end
end
$\varphi_O = \bigcup_{H \in \mathcal{H}} \bigcup_{\vec{b} \in B} F_{H,\vec{b}}$;
return φ_O

Following Algorithm 1, the flow-out is then the union of the contributions of all dependences to it:

$$\varphi_O = \bigcup_{H \in \mathcal{H}} \bigcup_{\vec{b} \in B} F_{H,\vec{b}}$$

and then the flow-out of a given tile is given by intersecting the domain of a tile with the flow-out iterations of the entire domain:

$$\varphi_O(S) = \varphi_O \cap S$$

4.3.2 MARS partitioning of the flow-out set.

We explain how our partitioning scheme is done in this subsection.

Principle. The flow-out set can be partitioned into MARS so that, for any given tile, all iterations any given tile produces in a MARS have the exact same consumer tiles. The partitioning idea is illustrated in Figure 3.

The partitioning is done by computing those subsets of the flow-out for which, given select tiling hyperplanes, any dependence crosses all these tiling hyperplanes, and no dependence crosses any other tiling hyperplane.

We browse all possible consumers by applying the above on all combinations of tiling hyperplanes. As there are T tiling hyperplanes, there are $2^T - 1$ possible consumer tiles, and therefore at most $2^{2^T-1} - 1$ MARS.

Example. We can construct the MARS for a Smith-Waterman kernel, which has the following characteristics:

- Domain: $\mathcal{D} = \left\{\begin{pmatrix} i \\ j \end{pmatrix} : (i, j) \in [0, 100]^2\right\}$

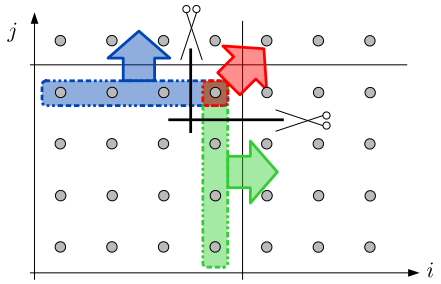


Figure 3. Flow-out set of a tile intersected with each consumers tile's flow-in set. The breakup we propose splits iterations that have different consumer tiles.

- PRDG: $B = \left\{\vec{b}_1 = \begin{pmatrix}1\\0\end{pmatrix}, \vec{b}_2 = \begin{pmatrix}0\\1\end{pmatrix}, \vec{b}_3 = \begin{pmatrix}1\\1\end{pmatrix}\right\}$ $(D = 3)$
- Tiling hyperplane families:
  $\mathcal{H} = \{\{\mathcal{H}_1 : i \equiv 0[4]\}, \{\mathcal{H}_2 : j \equiv 0[4]\}\}$ $(T = 2)$
- Normal vectors: $\left\{\vec{c}_1 = \begin{pmatrix}1\\0\end{pmatrix}, \vec{c}_2 = \begin{pmatrix}0\\1\end{pmatrix}\right\}$
- Tile sizes: $\{4, 4\}$

We can first notice that dependence $\vec{b}_1$ does not cross hyperplane $\mathcal{H}_2$, and likewise dependence $\vec{b}_2$ does not cross hyperplane $\mathcal{H}_1$.

Let us start by considering those dependences that cross any hyperplane of the $\mathcal{H}_1$ family and none of $\mathcal{H}_2$.

To have a dependence cross hyperplane $\mathcal{H}_1$, the source iteration $\vec{x} = \begin{pmatrix}i\\j\end{pmatrix}$ must be such that, if $\vec{b} = \begin{pmatrix}b_i\\b_j\end{pmatrix}$, then $(i \bmod 4) + b_i \geqslant 4$ or $(i \bmod 4) + b_i < 0$. Because all $\vec{b}$s only have positive coordinates, let us only consider the case $(i \bmod 4) + b_i \geqslant 4$.

From dependence $\vec{b}_1$, we get $i \bmod 4 \geqslant 3$; from dependence $\vec{b}_3$, we also get $i \bmod 4 \geqslant 3$. Therefore, the set of points such that *any* dependence crosses $\mathcal{H}_1$ is:

$$\left\{\vec{x} = \begin{pmatrix}i\\j\end{pmatrix} \in \mathcal{D} : i \bmod 4 \geqslant 3\right\}$$

or equivalently

$$\left\{\vec{x} = \begin{pmatrix}i\\j\end{pmatrix} \in \mathcal{D} : i \equiv 3[4]\right\}$$

We compute the subset of these points for which $\mathcal{H}_2$ is crossed. The condition to cross $\mathcal{H}_2$ is that, if $\vec{x} = \begin{pmatrix}i\\j\end{pmatrix}$ and $\vec{b} = \begin{pmatrix}b_i\\b_j\end{pmatrix}$, then $(j \bmod 4) + b_j \geqslant 4$, which means $j \bmod 4 \geqslant 3$ with both dependences $\vec{b}_1$ and $\vec{b}_2$. We therefore get that the points from which $\mathcal{H}_1$ is crossed and not $\mathcal{H}_2$ is:

$$\left\{\vec{x} = \begin{pmatrix}i\\j\end{pmatrix} \in \mathcal{D} : i \equiv 3[4] \wedge \neg(j \equiv 3[4])\right\}$$

We can do the same procedure to cross only $\mathcal{H}_2$, and both of $\mathcal{H}_1$ and $\mathcal{H}_2$, which yield the following sets:

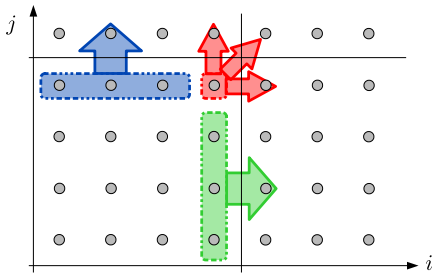


Figure 4. MARS and their consumers for Smith-Waterman using square tiling.

$$\left\{\vec{x} = \begin{pmatrix}i\\j\end{pmatrix} \in \mathcal{D} : \neg(i \equiv 3[4]) \wedge (j \equiv 3[4])\right\}$$

and

$$\left\{\vec{x} = \begin{pmatrix}i\\j\end{pmatrix} \in \mathcal{D} : i \equiv 3[4] \wedge j \equiv 3[4]\right\}$$

Those sets are the MARS we were looking for, and correspond to those in Figure 4.

Algorithm. Algorithm 2 gives the computation procedure to construct all MARS for all tiles.

In this algorithm, crossing a hyperplane is a shortcut for the property used in Algorithm 1. Assume $\vec{c} = (c_i)$ is the normal vector to a hyperplane H, s is the tile size along that hyperplane, $\vec{b}$ is a dependence vector, and $\vec{x} = (x_i) \in \mathcal{D}$. Let $m = \vec{c} \cdot \vec{b}$ assuming $m > 0$. Then:

$$\vec{x} + \vec{b} \text{ crosses } H \Leftrightarrow s - m \leqslant (\textstyle\sum_i (c_i x_i)) \bmod s < s$$

By intersecting the MARS obtained from Algorithm 2 with individual tiles, we obtain a decomposition of every tile's flow-out set into tile-wise MARS. These then satisfy the following two properties:

- Each tile-wise MARS is composed of iterations from a single tile,
- Each tile-wise MARS is entirely consumed by every of its consumer tiles.

4.4 Dual view: flow-in

Equivalently to partitioning the flow-out set into MARS, it is possible to compute the partitioning of the flow-in set of each tile into MARS. The flow-in set is computed with the same algorithm as the flow-out, using the opposite of the dependence vectors.

Intersecting the MARS created by Algorithm 2 (not broken up into individual tile-wise MARS) with the obtained tile-wise flow-in set then gives a breakup into MARS. This partitioning can be used to figure out which MARS every tile should fetch from other tiles as an input.

Algorithm 2: Computing the MARS

$\mathcal{D}$ = iteration space,

Input: $B = \left\{\vec{b}_i : i = 1, \ldots, D\right\}$ = PRDG,

$\mathcal{H} = \{H_i : i = 1, \ldots, T\}$ = tiling hyperplane families

Result: $\mathcal{M}$ = partition of flow-out into MARS

$\mathcal{M} = \emptyset$;

$\mathcal{T} = \mathcal{P}_n(\mathcal{H})$; `/* All neighboring tiles */`

for $I \in \mathcal{P}_n(\mathcal{T})$ **do**

  $E = \mathcal{T} \backslash I$;

  `//` A`: all tiles in` I `must be reached by` $\geqslant 1$ `dependence`

  for $CT \in I$ **do**

    $NCT = \mathcal{H} \backslash CT$;

    for $\vec{b} \in B$ **do**

      `//` $P_{CT,\vec{b}}$: $\vec{b}$ `crosses all hyperplanes of` CT `and no others`

      $$P_{CT,\vec{b}} = \left\{\vec{x} \in \mathcal{D} : \bigwedge_{H \in CT} \left(\vec{x} + \vec{b} \text{ crosses } H\right) \wedge \bigwedge_{H \in NCT} \neg \left(\vec{x} + \vec{b} \text{ crosses } H\right) \wedge \vec{x} + \vec{b} \in \mathcal{D}\right\};$$

    end

  end

  $A = \bigcap_{CT \in I} \bigcup_{\vec{b} \in B} P_{CT,\vec{b}}$;

  `//` S`: no tiles in` E `may be reached by any dependence`

  for $\vec{b} \in B$ **do**

    for $CT \in E$ **do**

      $NCT = \mathcal{H} \backslash CT$;

      `//` $Q_{CT,\vec{b}}$: $\vec{b}$ `crosses a hyp. in` NCT `or doesn't cross a hyp. in` CT

      $$Q_{CT,\vec{b}} = \left\{\vec{x} \in \mathcal{D} : \left[\bigvee_{H \in CT} \neg \left(\vec{x} + \vec{b} \text{ crosses } H\right) \vee \bigvee_{H \in NCT} \left(\vec{x} + \vec{b} \text{ crosses } H\right)\right] \wedge \vec{x} + \vec{b} \in \mathcal{D}\right\};$$

    end

  end

  $S = \bigcap_{CT \in E} \bigcap_{\vec{b} \in B} Q_{CT,\vec{b}}$;

  $\mathcal{M} = \mathcal{M} \cup \{A \cap S\}$;

end

return $\mathcal{M}$

5 Implementation and Analysis

It is possible to express all MARS using polyhedral tools (ISL) provided the tile sizes are constant. However, using the idea from [16], it is possible to use parametric tile sizes by adding those sizes as additional parameters. We have implemented a MARS calculator in Python using ISLPy.

5.1 Usage of the MARS calculator

A MARS calculator is available at https://github.com/cferr/mars.git.

To compute the MARS for a given program and tiling hyperplanes, it needs input that can be computed using publicly available tools:

- The polyhedral model of a program, to be extracted for instance using PET [17];
- Dependence vectors, obtained using array dataflow analysis e.g. using `iscc`;
- Legal tiling hyperplanes, found for instance by calling PLuTo; the standard equation and the normal vectors to these hyperplanes are to be provided.

The MARS calculator then runs Algorithm 2. A visualization of the MARS is given with `islplot` when the iteration space is two- or three-dimensional. For two-dimensional iteration spaces, the MARS in the entire iteration space can be visualized; for three-dimensional spaces, a sample tile needs to be provided and the MARS specific to that tile will be shown.

5.2 Results

We have run the MARS calculator against a series of uniform dependence benchmarks. This section evaluates the result on the following questions:

- How many MARS are there per tile?
- What is the dimensionality of the result MARS? In particular, how many singleton MARS are there?

5.2.1 Evaluated applications.

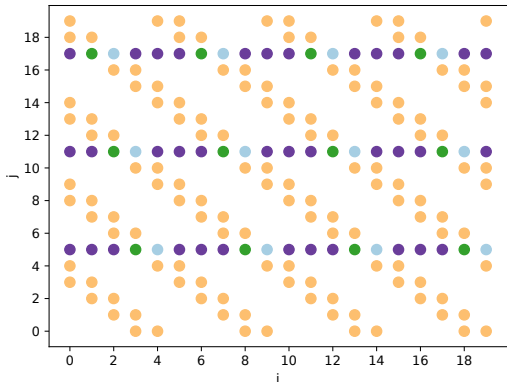
The MARS calculator has been used on the following applications:

- `sw`: Smith-Waterman dynamic programming algorithm for sequence alignment;
- `jacobi-1d`: Jacobi 1D stencil;
- `canonical-3d`: Artificial 3-dimensional example that has a dependence along each canonical axis;
- `gemm`: GEMM (BLAS) implementation from PolyBench [11];
- `seidel-2d`: Seidel 2D stencil implementation from PolyBench;
- `jacobi-2d`: Jacobi 2D stencil implementation from PolyBench;

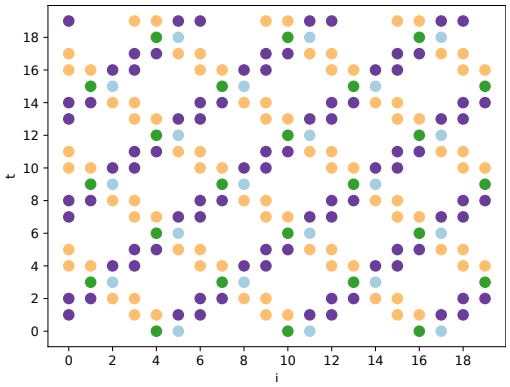
The `jacobi-2d` benchmark is exploited twice, with different tiling schemes: one is rectangular tiling combined with skewing, the other one is diamond tiling [2]. We will refer to them respectively as `jacobi-2d-r` and `jacobi-2d-d`.

Table 1 shows the results obtained by running the MARS calculator on the selected applications, and Figure 5 shows the MARS for a single tile of every application.

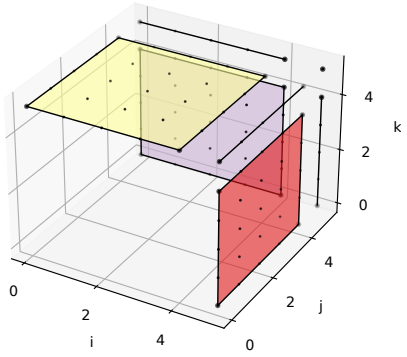
Table 1 gives the number of dimensions of the iteration space, the dependence pattern, tiling hyperplanes, the number of consumer tiles per tile (*# Cons. Tiles*), the number of



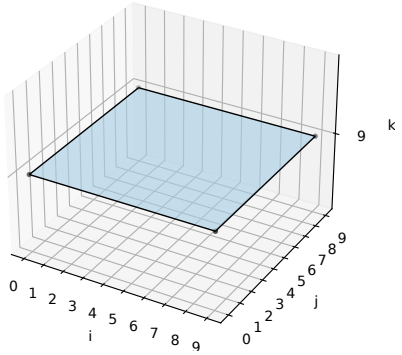
(a) `sw`



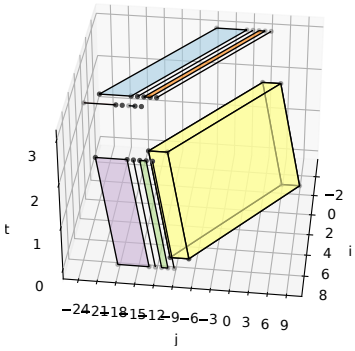
(b) `jacobi-1d`



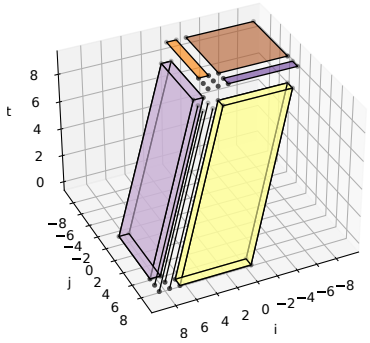
(c) `canonical-3d`



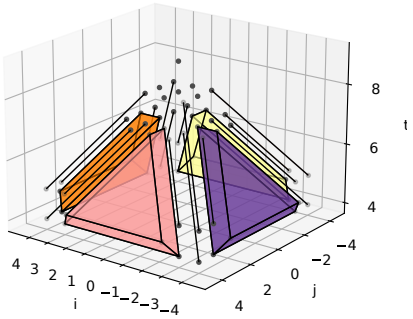
(d) `gemm`



(e) `seidel-2d`



(f) `jacobi-2d-r` (skewing + rectangular tiling)



(g) `jacobi-2d-d` (diamond tiling)

Figure 5. Visualization of MARS generated using the MARS calculator (https://github.com/cferr/mars.git). Note that $\vec{x} + \vec{b} \in \mathcal{D}$ isn't checked, the iteration space being assumed to be infinite by the computer.

Dims	Application	Dependences	Tiling hyperplanes	# Cons. tiles	Nb MARS	Singletons
2	`sw`	$(1,0),(0,1),(1,1)$	$i+j, j$	3	4	2
2	`jacobi-1d`	$(1,-1),(1,0),(1,1)$	$t+i, t-i$	3	4	2
3	`canonical-3d`	$(1,0,0),(0,1,0),(0,0,1)$	i, j, k	3	7	1
3	`gemm`	$(0,1,0)$	i, j, k	1	1	0
3	`seidel-2d`	$(0,1,1),(0,0,1),(1,-1,1),$ $(0,1,0),(1,0,0),(1,-1,0),$ $(0,1,-1),(1,0,-1),$ $(1,-1,-1)$	$t, t+i, 4t+2i+j$	7	13	2
3	`jacobi-2d-r`	$(1,0,1),(1,1,0),(1,0,0),$ $(1,-1,0),(1,0,-1)$	$t, t+i, t+j$	7	13	4
3	`jacobi-2d-d`	$(1,0,1),(1,1,0),(1,0,0),$ $(1,-1,0),(1,0,-1)$	$t+i, t+j, t-i, t-j$	15	34 (26 non-empty)	6

Table 1. Results obtained from the MARS calculator

computed MARS (*Nb MARS*) and the number of singleton MARS (that have a single point per tile).

5.2.2 Analysis. Two observations can be made out of the MARS, on their number and the tiles that consume them. As a general rule, the consumer tiles of a MARS are adjacent to each other, and the more cutting hyperplanes surrounding a MARS, the fewer dimensions it has. One notable case is `seidel-2d` (Figure 5e) where a two-dimensional MARS is surrounded by two one-dimensional ones, close to the $t+i$ and $4t+2i+j$ hyperplanes intersection, and close to the t and $4t+2i+j$ intersection.

In 2-dimensional iteration spaces, two tiling hyperplanes are enough to tile all dimensions, in which case there are a maximum of $2^{2^2-1}-1=7$ MARS per tile. Our examples, `sw` and `jacobi-1d`, only exhibit 4 MARS, each MARS being consumed by two adjacent tiles.

In 3-dimensional iteration spaces, the number of MARS goes up to 34 per tile on `jacobi-2d-d` out of a maximum 32767 (due to the 15 consumer tiles). Computing the MARS took more than two hours for `jacobi-2d-d` on an *Intel Core i7-8665U* CPU. As one can expect given the size of $\mathcal{P}_n(\mathcal{P}_n(\mathcal{H}))$, the complexity of the computation is such that our computer will not find the MARS in a matter of hours if there are 5 or more tiling hyperplanes. This is a strong call to prune the search space.

There are several possible optimizations. The first one is to figure out the actual consumer tiles instead of enumerating $\mathcal{P}_n(\mathcal{P}_n(\mathcal{H}))$. This is implemented in the MARS computer.

Then, not all tuples of consumer tiles will yield a MARS; in particular, very specific dependence patterns will yield MARS consumed by non-adjacent tiles (for instance, this is the case in `canonical-3d`, and more generally in any application where every dependence is orthogonal to a tiling hyperplane). Figuring out precisely when this happens would drastically reduce the search space.

For the `jacobi-2d-d` instance, there were 34 MARS computed, but only 26 of them would not be empty when intersecting them with the tile being shown. This is due to the fact that the tiling hyperplanes are not linearly independent, and create in this case three different tile shapes; not all tile shapes yield all MARS. To correctly generate code with MARS input / output, we will need to compute the MARS per tile instead of for the entire iteration space.

6 Possible Applications

MARS can be used in a variety of applications where fine-grain knowledge of the tile's flow-in origin and flow-out destination is known. In this paper, we detail two applications: compression, and memory allocation.

6.1 Compression

The fact that MARS are not redundant makes them suitable for compression: in general, decompression of an entire block of data is needed to access part of it. When using MARS, all the data that is decompressed is actually needed, and therefore there is no compression-induced redundancy. Works such as Ozturk et al. [10] could be extended with data tiles of different sizes, where each data tile is actually a MARS.

Singleton and low-dimensional MARS are, however, going to be detrimental to compression. The volume of each MARS is a function of at least one tile size, with the exception of singleton MARS which volume will not change as tile size grows.

6.2 Memory Allocation

MARS can be used to construct a memory allocation for inter-tile communication, similarly to what Zhao et al. [19] have done with coarser-grain blocks. The idea is similar: allocate contiguous blocks of memory for each MARS, and find a suitable layout.

6.2.1 A case for merging MARS. In some cases, as it can be observed in Figure 5, the no-redundancy property yielding

singletons may cause performance penalties. This is notably the case when creating access functions at the granularity of MARS: such access functions will read or write the exact access data for each tile, but unless singleton accesses are merged with other accesses, these will incur a bandwidth waste.

To alleviate the performance issues, we need to relax the no-redundancy property, and allow for MARS to be merged according to an objective function. This merge process yields an intermediate partitioning between very fine-grain MARS and the entire flow-out, which would be the result of merging all MARS together.

6.2.2 Global memory allocation. The case of global memory allocation uses the number of transactions as a metric: the runtime of memory accesses is a function of the number of transactions and the volume of each transaction. Because global memories are behind shared buses, each transaction costs a fixed initiation penalty caused by arbitration, plus the number of cycles the actual transfer takes; the longer the transfer, the more profitable it is and the better usage of the bandwidth.

An optimization problem can be formulated with the following elements:

- Minimize the number of transactions,
- Maximize every transaction's length (and therefore bandwidth usage),
- Find an intra-MARS layout,
- Find an inter-MARS layout, possibly allowing interleaving MARS from different tiles.

This is left for future work.

7 Conclusion

In this work, we have introduced an element of program analysis, MARS, to determine sets of data communicated between tiles without redundancy. These sets can be computed for certain programs with uniform dependence patterns, and their computation can be automated.

A number of questions remain open with respect to the use of these sets. The MARS are a fine-grain data structure in terms of usage per tile, which means that they can yield the minumum amount of inter-tile communcation; however, the presence of singletons with high transfer cost means that further manipulations on MARS are needed to make them suitable for memory transfers or compression.

References

[1] Uday Bondhugula. 2013. Compiling Affine Loop Nests for Distributed-Memory Parallel Architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM. https://doi.org/10.1145/2503210.2503289

[2] Uday Bondhugula, Vinayaka Bandishti, and Irshad Pananilath. 2017. Diamond Tiling: Tiling Techniques to Maximize Parallelism for Stencil Computations. *IEEE Transactions on Parallel and Distributed Systems* 28, 5 (may 2017), 1285–1298. https://doi.org/10.1109/tpds.2016.2615094

[3] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. 2008. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *International Conference on Compiler Construction (ETAPS CC)*. https://doi.org/10.1007/978-3-540-78791-4_9

[4] 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/10.1145/1379022.1375595

[5] Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. 2013. Generating Efficient Data Movement Code for Heterogeneous Architectures with Distributed-Memory. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. IEEE. https://doi.org/10.1109/PACT.2013.6618833

[6] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (feb 1991), 23–53. https://doi.org/10.1007/BF01407931

[7] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem. Part I. One-dimensional Time. *International Journal of Parallel Programming* 21, 5 (1992), 313–347. https://doi.org/10.1007/BF01407835

[8] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time. *International Journal of Parallel Programming* 21, 6 (1992), 389–420. https://doi.org/10.1007/BF01379404

[9] F. Irigoin and R. Triolet. 1988. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '88*. ACM Press. https://doi.org/10.1145/73560.73588

[10] O. Ozturk, M. Kandemir, and M.J. Irwin. 2009. Using Data Compression for Increasing Memory System Utilization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 6 (jun 2009), 901–914. https://doi.org/10.1109/tcad.2009.2017430

[11] Louis-Noël Pouchet and Tomofumi Yuki. 2016. PolyBench/C 4.2.1. http://polybench.sf.net

[12] P. Quinton and V. Van Dongen. 1989. The Mapping of Linear Recurrence Equations on Regular Arrays. *Journal of VLSI Signal Processing* 1, 2 (1989), 95–113. https://doi.org/10.1007/BF02477176

[13] S. V. Rajopadhye. 1986. *Synthesis, Optimization and Verification of Systolic Architectures*. Ph. D. Dissertation. University of Utah, Salt Lake City, Utah 84112.

[14] S. V. Rajopadhye. 1989. Synthesizing Systolic Arrays with Control Signals from Recurrence Equations. *Distributed Computing* 3 (May 1989), 88–105. https://doi.org/10.1007/BF01558666

[15] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto. 1986. On Synthesizing Systolic Arrays from Recurrence Equations with Linear Dependencies. In *Proceedings, Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer Verlag, LNCS 241, New Delhi, India, 488–503. https://doi.org/10.1007/3-540-17179-7_30

[16] Lakshminarayanan Renganarayanan, Daegon Kim, Michelle Mills Strout, and Sanjay Rajopadhye. 2012. Parameterized loop tiling. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 34, 1 (2012), 3. https://doi.org/10.1145/2160910.2160912

[17] Sven Verdoolaege and Tobias Grosser. 2012. Polyhedral Extraction Tool. In *Second Int. Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. Paris, France.

[18] Michael E. Wolf and Monica S. Lam. 1991. A data locality optimizing algorithm. *ACM SIGPLAN Notices* 26, 6 (jun 1991), 30–44. https://doi.org/10.1145/113446.113449

[19] Tuowen Zhao, Mary Hall, Hans Johansen, and Samuel Williams. 2021. Improving communication by optimizing on-node data movement with data layout. In *Proceedings of the 26th ACM SIGPLAN Symposium*

on Principles and Practice of Parallel Programming. ACM. https://doi.org/10.1145/3437801.3441598

Acknowledgments

The authors would like to thank Tomofumi Yuki for the scientific insight and valuable discussions that led to the construction of MARS.