Parametric Tiling with Inter-Tile Data Reuse

Alain Darte Alexandre Isoard
Compsys, LIP, UMR 5668 CNRS, INRIA, ENS-Lyon, UCB-Lyon
firstname.lastname@ens-lyon.fr

ABSTRACT

Loop tiling is a loop transformation widely used to improve spatial and temporal data locality, increase computation granularity, and enable blocking algorithms, which are particularly useful when offloading kernels on platforms with small memories. When hardware caches are not available, data transfers must be software-managed: they can be reduced by exploiting data reuse between tiles and, this way, avoid some useless external communications. An important parameter of loop tiling is the sizes of the tiles, which impact the size of the necessary local memory. However, for most analyzes that involve several tiles, which is the case for inter-tile data reuse, the tile sizes induce non-linear constraints, unless they are numerical constants. This complicates or prevents a parametric analysis. In this paper, we show that, actually, parametric tiling with inter-tile data reuse is nevertheless possible, i.e., it is possible to determine, at compile-time and in a parametric fashion, the copy-in and copy-out data sets for all tiles, with inter-tile reuse, as well as the sizes of the induced local memories, without the need to analyze the code for each tile size.

1. INTRODUCTION

Loop tiling is a well-known loop transformation used to improve data locality, increase computation granularity, and control the use and size of local memories for out-of-core computations. We refer to [24] for all details on its semantics, validity conditions, and code generation. It was first introduced as "supernode partitioning" [14], for a set of perfectly nested loops, as a grouping of iterations into *supernodes*. Supernodes are atomic (i.e., can be executed without any communication/synchronization with other supernodes except for live-in/live-out data at beginning/end of a tile execution), identical by translation, bounded, and they form a partition of the whole iteration space. Validity conditions were given in terms of dependence cones and hyperplane partitioning, which define tiles as hyper-rectangles (after some possible change of basis) and establish a link with affine scheduling and the generation of permutable loops. Today, tiling is also used for non-perfectly nested loops [5], thanks to multi-dimensional affine functions: as for the perfect case, some permutable dimensions can be used to perform tiling, even if not all instructions have the same iteration domain. Analysis and code generation may involve more complex sets, but the principles are similar.

Loop tiling can be viewed as a composition of strip-mining and loop interchange, after a preliminary change of basis. It transforms n nested loops into n *tile loops* iterating over the tiles, surrounding n *intra-tile loops* iterating within a tile. Dependence analysis and code generation for loop tiling is well-established in the polyhedral model [10], i.e., for a set of nested `for` loops, writing and reading multi-dimensional arrays and scalar variables, where loop bounds, `if` conditions, and array access functions are affine expressions of surrounding loop counters and structure parameters. In this case, loop iterations can be represented by a *polyhedral iteration domain*. When tile sizes are numerical constants, parametric (in terms of program counters and structural parameters) polyhedral optimizations (e.g., linear programming) can be used although loop tiling transforms n loops into $2n$ loops. Indeed, the image by tiling of a n-dimensional polyhedral iteration domain can be expressed as a $2n$-dimensional polyhedral iteration domain, because the set of points after tiling with fixed sizes can be described by affine inequalities.[1]

In general, *parametric tiling* refers to the case where tile sizes are parameters too. Parametric analysis within a tile is in general feasible as the set of points in a tile is defined with affine constraints from the tile sizes and the *tile origin* (first point in the tile). However, when an analysis involves several tiles, it becomes more intricate, if not unsolvable, as *a priori* expressing the tiled space with tile sizes as parameters induces quadratic constraints. For example, the tiling theory developed in [23], the code generation schemes of [14, 11, 6], the data movement and scratch-pad optimizations of [16, 15, 4, 3, 19] are not parametric. Recently, efficient code generation for parametric tiling [20, 13] as well as some form of symbolic scheduling for tiled codes [7] have been developed.

In this paper, we show that the exact and approximated inter-tile data reuse techniques developed in [3] can be extended to the parametric case. The trick to get around a quadratic formulation is to work with all possible tiles, not just the tiles that are part of the iteration space partitioning and whose origins belong to a lattice, but the difficulty is to make sure that exactness and correctness are maintained.

IMPACT 2014
Fourth International Workshop on Polyhedral Compilation Techniques
Jan 20, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

`http://impact.gforge.inria.fr/impact2014`

[1] However, difficulties due to large coefficients are possible.

2. PREREQUISITES

2.1 Notations and definitions

Vectors are written with arrows such as $\vec{i}$, with components $i_1, \ldots, i_n$. The vector $\vec{0}$ (resp. $\vec{1}$) has all components equal to 0 (resp. 1) and $\vec{a} \circ \vec{b}$ is the product (component-wise) of $\vec{a}$ and $\vec{b}$. We denote by $\preceq$ the lexicographic total order and by $\leq$ the component-wise partial order on vectors: $\vec{i} \leq \vec{j}$ if $\vec{i}$ and $\vec{j}$ have same dimension and $i_k \leq j_k$ for all k.

We will not elaborate on how to build and interpret the different affine functions for tiling non-perfectly nested loops. To simplify the discussion and notations, we only focus on the n dimensions to be tiled. We assume that each statement S with iteration domain $\mathcal{D}_S$ (with iteration vector $\vec{i}$) is tiled, after a first affine mapping $\vec{i} \mapsto \vec{i'} = \theta(S, \vec{i})$, by canonical tiles whose sizes are specified by a vector $\vec{s}$. In other words, a point $\vec{i}$ is mapped to the tile indexed by $\vec{T}$ where $T_k = \lfloor \frac{i'_k}{s_k} \rfloor$, or equivalently $s_k T_k \leq (\theta(S, \vec{i}))_k < s_k(T_k + 1)$, for $k \in [1..n]$. Also, we restrict to the case where both the original program and the tiled program are executed sequentially.[2] Several orders of iterations in the tiled program are possible, we consider that the tiled code is executed following the lexicographic order on the $2n$-dimensional vectors $(\vec{T}, \vec{i'})$. The tiled iteration domain for statement S is then:

$$\mathcal{T}_S = \{(\vec{T}, \vec{i'}) \mid \exists \vec{i} \in \mathcal{D}_S,\ \vec{i'} = \theta(S, \vec{i}),\ \vec{0} \leq \vec{i'} - \vec{s} \circ \vec{T} \leq \vec{s} - \vec{1}\}$$

If θ is a one-to-one mapping between integer points and $\mathcal{D}_S$ is the set of integer points in a polyhedron, $\vec{i}$ can be eliminated and $\mathcal{T}_S$ is also the set of integer points in a polyhedron.

Example. We will illustrate the different concepts and steps of our technique with the kernel `jacobi_1d_imper` from PolyBench [18], with a time loop, and tiled in two dimensions.

```
for (t = 0; t < M; t++) {
  for (i = 1; i < N - 1; i++)
    S1: B[i] = 0.33333 * (A[i-1] + A[i] + A[i+1]);
  for (j = 1; j < N - 1; j++)
    S2: A[j] = B[j];
}
```

The Pluto compiler [17] generates the following mapping:

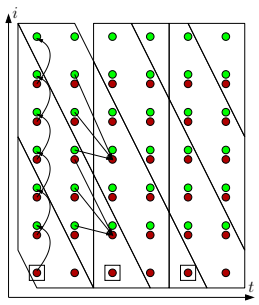
$$\theta(S_1, (t,i)) = (t, 2t+i, 0) \quad \theta(S_2, (t,j)) = (t, 2t+j+1, 1)$$
$$\mathcal{D}_{S_1} = \mathcal{D}_{S_2} = \{(t,i) \mid 0 \leq t \leq M-1,\ 0 \leq i \leq N-2\}$$

This amounts to shifting S_2 by 1 in the j loop, to fusing the i and j loops, as depicted in Fig. 1, then to skewing by 2 the inner loop before tiling (tiles have size 2×3 in Fig. 1). □

2.2 Inter-tile reuse

The inter-tile reuse problem we consider is the kernel offloading with optimized remote accesses presented in [3]. A kernel is tiled and offloaded, tile by tile, to a computing accelerator (a FPGA for [3]). Initially, all data are in remote (external) memory, while all computations are performed on the accelerator. Each tile $\vec{T}$ consists of three *successive* phases: a loading phase where data are copied from external memory to local memory, enabling burst communications,

[2]However, parallelism inside a tile is possible, as well as hierarchical tiling to play with the extent of the tiled domain. Extensions to parallel executions seem also possible by defining a partial execution order, but this is left for future work.



Non-empty 2×3 tiles drawn w.r.t the original space, S_1: red, S_2: green.

Are also shown some flow dependences, due to reads of `B`, at distance $(0,1)$, and reads of `A`, at distance $(1,0)$, $(1,-1)$, $(1,-2)$ in the (t,i) space.

Figure 1: `jacobi1d` kernel and skewed tiling.

then a compute part where the original computations corresponding to the tile are performed on the local memory, and finally a storing phase where data are copied to external memory. In addition, all compute parts are done sequentially on the accelerator, following the lexicographic order on tile indices, and the same is true for loading phases (resp. storing phases). However, loads/stores can be done concurrently with computations of other tiles, enabling pipelining and execution similar to double buffering, even when some data are both read and written, thanks to *inter-tile reuse*. The problem is to define the loading and storing sets $\text{Load}(\vec{T})$ and $\text{Store}(\vec{T})$ for each tile $\vec{T}$ so that a data element is never loaded from external memory if it is already available in local memory, i.e., it has already been loaded or computed (as, in this latter case, the external memory is not necessarily up-to-date). This inter-tile reuse is performed for each *tile strip* (subspace of tiles corresponding to inner tile dimensions). In [3], a tile strip is one-dimensional, but the technique can be applied to multi-dimensional strips. This choice however impacts the size of the local memory.

The procedure developed in [3] is based on parametric linear programming [9]. It consists in performing loads (resp. stores) as late (resp. as soon) as possible, i.e., a data element is loaded just before the first tile that accesses it, if this access is a read, and is stored just after the last tile that writes it. Among all schemes that exploit a full inter-tile reuse, this tends to reduce the size of the local memory. We illustrate this technique on the `jacobi_1d_imper` example.

Example (cont'd). For the tiling of Fig. 1, a 1D tile strip is vertical, indexed by $T_1 = \lfloor \frac{t}{s_1} \rfloor$. To simplify explanations, we only consider the array `A` (the array `B` is not live-in of a tile strip). We compute the first operation (following the order in the tiled code) that accesses `A[m]`, i.e., with $(i_1, i_2) = (t, i)$ and parameters M, N, m, T_1, we compute the lexicographic minimum of $(T_2, i'_1, i'_2, k, i_1, i_2)$ such that:

$$\begin{cases} -1 \leq m - i_2 \leq 1,\ 0 \leq i_1 \leq M-1,\ 1 \leq i_2 \leq N-2,\ k = 0, \\ i'_1 = i_1,\ i'_2 = 2i_1 + i_2,\ 0 \leq i'_1 - 2T_1 \leq 1,\ 0 \leq i'_2 - 3I_2 \leq 2 \end{cases}$$
$$\vee$$
$$\begin{cases} m = i_2,\ 0 \leq i_1 \leq M-1,\ 1 \leq i_2 \leq N-2,\ k = 1, i'_1 = i_1, \\ i'_2 = 2i_1 + i_2 + 1,\ 0 \leq i'_1 - 2T_1 \leq 1,\ 0 \leq i'_2 - 3T_2 \leq 2 \end{cases}$$

The first set of constraints corresponds to reads in S_1 and specifies that `A[m]` is `A[i-1]`, `A[i]`, or `A[i+1]`, that iterations in tiles are valid, i.e., $(T_1, T_2, i'_1, i'_2) \in \mathcal{T}_S$, and $k = 0$ expresses the third component of $\theta(S_1, (t,i))$. The second set of constraints corresponds to writes in S_2 (with $k = 1$). The lexicographic minimum is expressed as a disjunction of

cases (a QUAST [9] or quasi affine solution tree). Then, all solutions (i.e., leaves of the tree) that correspond to a write operation are removed. Here, all first accesses are reads, no simplification is needed. It remains to project out the variables i'_1, i'_2, i_1, i_2, k, to get a relation between tile index $\vec{T}$ and array element m, which describes Load($\vec{T}$) as a union:

$$\{m \mid 0 \le 2T_1 \le M-1,\ 2 \le m \le N-1,\ 1 \le m+4T_1-3T_2 \le 3\}$$
$$\cup$$
$$\{m \mid 0 \le m \le 1,\ 3 \le N,\ 0 \le 2T_1 \le M-1,\ -1 \le 4T_1-3T_2 \le 1\}$$

The second set corresponds to loading the additional `A[0]` and `A[1]` for the unique tile in the tile strip that contains an iteration $(t, 1)$ on its first column (squares in Fig. 1). □

As can be seen from the inequalities involved in the previous example with $\vec{s} = (2, 3)$ (and in the definition of $\mathcal{T}_S$), considering the components of the size vector $\vec{s}$ as parameters generates quadratic constraints. In other words, this formulation is inherently not linear in the tile sizes. The goal of this paper is to show that, surprisingly, the problem can nevertheless be solved, both for exact inter-tile reuse (as in the previous example) and with approximations, thus fully extending the work of [2] to parametric tiling.

3. DEALING WITH UNALIGNED TILES

The first key idea is to represent each tile not with its tile index $\vec{T}$ defined earlier, but with the indices $\vec{I}$ of its *origin* (first element in the tile in the lexicographic order). The first difference is that tiles are scanned with loops with increments equal to $\vec{1}$ when $\vec{T}$ is used and $\vec{s}$ when $\vec{I}$ is used. The second difference is that, when $\vec{I}$ is used instead of $\vec{T}$, the set of elements $\vec{i}$ in a tile is affine in $\vec{s}$: this is the set of all $\vec{i}$ such that $\vec{I} \le \vec{i} \le \vec{I} + \vec{s} - \vec{1}$. In other words, parametric analysis inside a tile is possible. This representation is not new, it is used for analysis in PIPS [12][Fig. 6] and for parametric code generation [20]. Of course, the non-linearity has not disappeared yet. Indeed, the tile origins $\vec{I}$ are restricted to the lattice $\mathcal{L}$ defined by $\vec{I} \in \mathcal{L}$ iff $\vec{I} = \vec{s} \circ \vec{J}$ for some integer vector $\vec{J}$. Without these lattice constraints, the inter-tile reuse problem would be affine in $\vec{s}$. The second key idea is to show how these quadratic constraints can be ignored.

Note that, with standard conditions for tiling (i.e., when all dependence distances are non-negative along the dimensions being tiled [14]), if a tiling is valid, then any translation of it is valid too. In other words, considering all tile origins $\vec{I} = \vec{s} \circ \vec{J} + \vec{I}_0$ for some vector $\vec{I}_0$ defines a valid tiling too. This has the same effect as defining the tiling from the shifted mapping $\vec{i} \mapsto \sigma(S, \vec{i}) - \vec{I}_0$ for all S. We say that two tiles are *aligned* if they belong to the same tiling.

3.1 Exact approach with set equations

The formulation given in Section 2.2 as a linear programming optimization is one possible approach to solve the problem. It was initially formulated in [3] as set equations:

- Load($\vec{T}$) = In($\vec{T}$) \ {In($\vec{T}' \prec \vec{T}$) ∪ Out($\vec{T}' \prec \vec{T}$)}
- Store($\vec{T}$) = Out($\vec{T}$) \ Out($\vec{T}' \succ \vec{T}$)

In($\vec{T}$) and Out($\vec{T}$) are the standard *live-in* and *live-out* sets for tile $\vec{T}$, as defined for example for array region analysis [8]. $X(\vec{T}' \prec \vec{T})$ denotes the union of all sets $X(\vec{T}')$ for all tiles $\vec{T}'$ executed before $\vec{T}$ (lexicographic order) in the same tile strip as $\vec{T}$. Expressing $X(\vec{T}' \prec \vec{T})$ from $X(\vec{T}')$ is done simply by adding the constraint $\vec{T}' \prec \vec{T}$ and specifying that $\vec{T}'$ is in the strip where reuse is exploited. This reuse is obtained thanks to set differences. Intuitively, one would expect to subtract Load($\vec{T}' \prec \vec{T}$) from Load($\vec{T}$) and Store($\vec{T}' \succ \vec{T}$) from Store($\vec{T}$), but such recursive definitions are not usable.

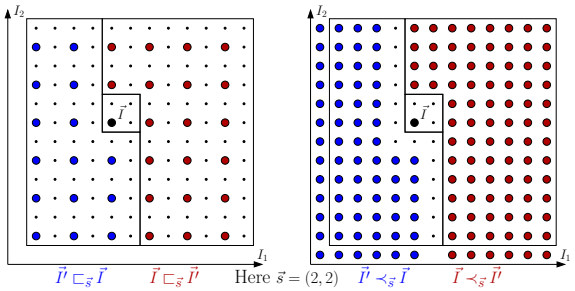


Figure 2: Orders $\sqsubset_{\vec{s}}$ and $\prec_{\vec{s}}$. Points are tile origins.

We now consider all tiles, not just those whose origins belong to the lattice $\mathcal{L}$, but all with the same $\vec{s}$. We define two relations on tiles:

- $\vec{I}' \sqsubset_{\vec{s}} \vec{I}$ iff $\vec{I}' \prec \vec{I}$ and $\vec{I} - \vec{I}' \in \mathcal{L}$.
- $\vec{I}' \prec_{\vec{s}} \vec{I}$ iff, for some $k \in [1..n]$, $I'_i \le I_i$ for all $i < k$ and $I'_k \le I_k - s_k$ where n is the dimension of $\vec{I}$ and $\vec{I}'$.

Their standard reflexive extensions $\sqsubseteq_{\vec{s}}$ and $\preceq_{\vec{s}}$ are partial orders. Fig. 2 shows all tile origins $\vec{I}'$ strictly smaller (in blue) or larger (in red) than the tile origin $\vec{I}$ (in black), for $\sqsubseteq_{\vec{s}}$ and $\preceq_{\vec{s}}$. Tiles comparable for $\sqsubseteq_{\vec{s}}$ are aligned with each other. Also, when $\vec{s} = \vec{1}$, the orders $\preceq$, $\preceq_{\vec{s}}$, and $\sqsubseteq_{\vec{s}}$ are equal.

PROPERTY 1. *The strict order $\prec_{\vec{s}}$ can be equivalently defined as follows: $\vec{I}' \prec_{\vec{s}} \vec{I}$ iff, in the tiling induced by $\vec{I}$ (the same is true with $\vec{I}'$), every point in the tile $\vec{I}'$ is executed before any point in the tile $\vec{I}$ (but $\vec{I}$ and $\vec{I}'$ may not be aligned).*

With tile origins, the Load/Store equations can be rewritten:

- Load($\vec{I}$) = In($\vec{I}$) \ (In($\vec{I}' \sqsubset_{\vec{s}} \vec{I}$) ∪ Out($\vec{I}' \sqsubset_{\vec{s}} \vec{I}$)) (1)
- Store($\vec{I}$) = Out($\vec{I}$) \ Out($\vec{I}' \sqsupset_{\vec{s}} \vec{I}$) (2)

The key is to show that these sets can also be defined as:

- Load($\vec{I}$) = In($\vec{I}$) \ (In($\vec{I}' \prec_{\vec{s}} \vec{I}$) ∪ Out($\vec{I}' \prec_{\vec{s}} \vec{I}$)) (3)
- Store($\vec{I}$) = Out($\vec{I}$) \ Out($\vec{I}' \succ_{\vec{s}} \vec{I}$) (4)

This is not obvious as the difference now also involves unaligned tiles that do not belong to the same tiling as $\vec{I}$. Nicely, these sets only involve affine constraints and can thus be computed with a library such as `isl` [21]. Before proving these formulas, we first illustrate their use with our example.

Example (cont'd). The following Load & Store sets were computed thanks to the `isl` calculator `iscc` [22] with the generic script of Fig. 3, for `jacobi_1d_imper` (see Fig. 1).

$$\text{Load}(\vec{I}) =$$
$$\left\{\text{A}(m) \,\middle|\, \begin{array}{l} 1 \le m + 2I_1 - I_2 \le s_2,\ s_1 \ge 1,\ I_1 \ge 0,\ m \ge 1, \\ I_1 \le -1 + M,\ I_2 \ge 2 - s_2 + 2I_1,\ m \le -1 + N,\ N \ge 3 \end{array}\right\}$$
$$\cup$$
$$\left\{\text{A}(m) \,\middle|\, \begin{array}{l} m \ge 1 + I_2,\ m \ge 1,\ M \ge 1,\ m \le -1 + N,\ I_1 \le -1, \\ I_1 \ge 1 - s_1,\ I_2 \ge 2 - s_2,\ N \ge 3,\ m \le s_2 + I_2 \end{array}\right\}$$
$$\cup$$
$$\{\text{A}(1) \mid\ I_2 = 1 + 2I_1 \wedge 0 \le I_1 \le -1 + M,\ N \ge 3,\ s_1 \ge 1,\ s_2 \ge 1\}$$
$$\cup$$
$$\{\text{A}(m) \mid\ 0 \le m \le 1,\ I_2 = 1 \le s_2,\ 1 - s_1 \le I_1 \le -1,\ M \ge 1,\ N \ge 3\}$$
$$\cup$$
$$\{\text{A}(0) \mid\ 0 \le I_1 \le M - 1,\ N \ge 3,\ s_1 \ge 1,\ 1 \le I_2 - 2I_1 \ge 2 - s_2\}$$
$$\cup$$
$$\{\text{A}(0) \mid\ 1 - s_1 \le I_1 \le -1,\ M \ge 1,\ N \ge 3,\ I_2 \ge 2 - s_2,\ I_2 \le 0\}$$

$$
\begin{aligned}
&\mathrm{Store}(\vec{I}) = \\
&\left\{ \mathrm{B}(m) \;\middle|\; \begin{array}{l} m \geq 1,\ m \geq 2 - 2M + s_2 + I_2,\ m \leq -2 + N, \\ I_1 \geq 1 - s_1,\ 2 \leq m + 2s_1 + 2I_1 - I_2 \leq 1 + s_2,\ s_1 \geq 1 \end{array} \right. \\
&\cup \\
&\left\{ \mathrm{B}(m) \;\middle|\; \begin{array}{l} m \geq 1,\ s_1 \geq 1,\ m \leq -2 + N,\ I_1 \leq -1 + M, \\ m \leq 1 - 2M + s_2 + I_2,\ m \geq 2 - 2s_1 - 2I_1 + I_2, \\ I_1 \geq 1 - s_1,\ M \geq 1,\ m \geq 2 - 2M + I_2 \end{array} \right. \\
&\cup \\
&\left\{ \mathrm{A}(m) \;\middle|\; \begin{array}{l} m \geq 1,\ m \geq 1 - 2M + s_2 + I_2,\ m \leq -2 + N, \\ I_1 \geq 1 - s_1,\ 1 \leq m + 2s_1 + 2I_1 - I_2 \leq s_2,\ s_1 \geq 1 \end{array} \right. \\
&\cup \\
&\left\{ \mathrm{A}(m) \;\middle|\; \begin{array}{l} m \geq 1,\ s_1 \geq 1,\ m \leq -2 + N,\ I_1 \leq -1 + M, \\ m \leq -2M + s_2 + I_2,\ m \geq 1 - 2s_1 - 2I_1 + I_2, \\ I_1 \geq 1 - s_1,\ M \geq 1,\ m \geq 1 - 2M + I_2 \end{array} \right.
\end{aligned}
$$

The fact that the array `B` appears in the Store set may be surprising as `B` is recomputed in each tile strip (this is why it does not appear in the Load set). This is because the script of Fig. 3 considers each tile strip in isolation. To be able to remove `B` from the Store set, one would need a similar analysis on tile strips to discover that `B` is actually overwritten by subsequent tile strips. Then, only the last tile strip should store `B`, in case it is live-out of the program.

It can be checked (e.g., with `iscc`) that the set $\mathrm{Load}(\vec{I})$ above is indeed a generalization of the set $\mathrm{Load}(\vec{T})$ derived earlier for the canonical tiling with $\vec{s} = (2, 3)$. This is the complete expression, parameterized by $\vec{s}$, of all cases, including incomplete tiles, and even tilings obtained by translation of $\mathcal{L}$. Note that simply changing `Strip` (see Fig. 3) into `{[I_1,I_2]->[I_1',I_2']}` gives 2D inter-tile reuse, i.e., in the whole space. Constraints on parameters or $\vec{I}$ can also be added as in `Params`, e.g., to get simplified Load/Store sets for complete tiles, or to only consider large tiles, etc. Note however that `isl` uses coalescing heuristics to simplify expressions and, depending on the constraints, the outcome can be simpler or more complicated (although equivalent). Here, replacing $s_1 \geq 0$ by $s_1 > 0$ changes the expression. □

To prove that, when all sets $\mathrm{In}(\vec{I})$ and $\mathrm{Out}(\vec{I})$ are exact, it is equivalent to use $\prec_{\vec{s}}$ instead of $\sqsubset_{\vec{s}}$ in the Load/Store formulas, we define the concept of *pointwise functions*. This is exactly what we need to understand the problems, even more subtle in the case of approximations, related to the equality (or not) of some unions of images of sets (as in Eqs. (1) and (3) for Load, and (2) and (4) for Store). If $\mathcal{A}$ is a set, $\mathcal{P}(\mathcal{A})$ denotes the set of subsets of $\mathcal{A}$ (sometimes also written $2^{\mathcal{A}}$). Below, the function F is typically a function such as Out, which maps a tile, i.e., a subset of the tile strip ($\mathcal{A}$), to a subset of all data elements ($\mathcal{B}$).

Definition 1. *Let $\mathcal{A}$ and $\mathcal{B}$ be two sets, $\mathcal{C} \subseteq \mathcal{P}(\mathcal{A})$. The function $F : \mathcal{C} \to \mathcal{P}(\mathcal{B})$ is* pointwise *iff there exists a function $f : \mathcal{A} \to \mathcal{P}(\mathcal{B})$ such that $\forall X \in \mathcal{C}$, $F(X) = \bigcup_{x \in X} f(x)$.*

If F and G are from $\mathcal{C}$ to $\mathcal{P}(\mathcal{B})$, we write $F \subseteq G$ if $\forall X \in \mathcal{C}$, $F(X) \subseteq G(X)$. The following identifies the "largest" f.

Property 2. *For $F : \mathcal{C} \subseteq \mathcal{P}(\mathcal{A}) \to \mathcal{P}(\mathcal{B})$, let $F_\circ$ be the pointwise function defined from $f_\circ(x) = \bigcap_{Y \in \mathcal{C},\, x \in Y} F(Y)$. Then $F_\circ$ is the largest pointwise under-approximation of F, i.e., $F_\circ \subseteq F$ and, if F' is pointwise, $F' \subseteq F \Rightarrow F' \subseteq F_\circ$. In particular, F is pointwise if and only if $F = F_\circ$.*

Proof. Let $X \in \mathcal{C}$ and $y \in F_\circ(X) = \cup_{x \in X} f_\circ(x)$: $\exists x_y \in X$ such that $y \in f_\circ(x_y)$. With $Y = X$ in the definition of $f_\circ$, we get $f_\circ(x_y) \subseteq F(X)$, thus $y \in F(X)$, and $F_\circ \subseteq F$. If F' is pointwise and $F' \subseteq F$, then $f'(x) \in F'(Y) \subseteq F(Y)$ for all $Y \in \mathcal{C}$ such that $x \in Y$. Thus $f'(x) \subseteq f_\circ(x)$ by definition of $f_\circ$. Finally, if F is pointwise, $F \subseteq F_\circ$, thus $F = F_\circ$ since $F_\circ \subseteq F$. Conversely, if $F = F_\circ$, F is pointwise with $f_\circ$. □

```
# Inputs
Params := [M, N, s_1, s_2] -> { : s_1 >= 0 and s_2 >= 0 };
Domain := [M, N] -> { # Iteration domains
  S_1[i_1, i_2] : 1 <= i_2 <= N-2 and 0 <= i_1 <= M-1;
  S_2[i_1, i_2] : 1 <= i_2 <= N-2 and 0 <= i_1 <= M-1;
} * Params;
Read := [M, N] -> { # Read access functions
  S_1[i_1, i_2] -> A[m] : -1 + i_2 <= m <= 1 + i_2;
  S_2[i_1, i_2] -> B[i_2]; } * Domain;
Write := [M, N] -> { # Write access functions
  S_1[i_1, i_2] -> B[i_2];
  S_2[i_1, i_2] -> A[i_2]; } * Domain;
Theta := [M, N] -> { # Preliminary mapping
  S_1[i_1, i_2] -> [i_1, 2 i_1 + i_2, 0];
  S_2[i_1, i_2] -> [i_1, 1 + 2 i_1 + i_2, 1]; };

# Tools for set manipulations
Tiling := [s_1, s_2] -> { # Two dimensional tiling
  [[I_1, I_2] -> [i_1, i_2, k]] -> [i_1, i_2, k] :
      I_1 <= i_1 < I_1 + s_1 and I_2 <= i_2 < I_2 + s_2 };
Coalesce := { [I_1, I_2] -> [[I_1, I_2] -> [i_1, i_2, k]] };
Strip := { [I_1, I_2] -> [I_1, I_2'] };
Prev := { # Lexicographic order
  [[I_1, I_2] -> [i_1, i_2, k]] -> [[I_1, I_2] -> [i_1', i_2', k']] :
      i_1' <= i_1 - 1 or (i_1' <= i_1 and i_2' <= i_2 - 1)
      or (i_1' <= i_1 and i_2' <= i_2 and k' <= k - 1) };
TiledPrev := [s_1, s_2] -> { # Special ''lexicographic'' order
  [I_1, I_2] -> [I_1', I_2'] : I_1' <= I_1 - s_1 or
      (I_1' <= I_1 and I_2' <= I_2 - s_2) } * Strip;
TiledNext := TiledPrev^-1;
TiledRead  := Tiling.(Theta^-1).Read;
TiledWrite := Tiling.(Theta^-1).Write;

# Set/relation computations
In  := Coalesce.(TiledRead - (Prev.TiledWrite));
Out := Coalesce.TiledWrite;
Load  := In - ((TiledPrev.In) + (TiledPrev.Out));
Store := Out - (TiledNext.Out);
print coalesce (Load % Params);
print coalesce (Store % Params);
```

Figure 3: Script `iscc` for the Jacobi1D example.

Property 3. *$F : \mathcal{C} \to \mathcal{P}(\mathcal{B})$ is pointwise iff $\forall \mathcal{C}', \mathcal{C}'' \subseteq \mathcal{C}$, $\bigcup_{X \in \mathcal{C}'} X = \bigcup_{X \in \mathcal{C}''} X \Rightarrow \bigcup_{X \in \mathcal{C}'} F(X) = \bigcup_{X \in \mathcal{C}''} F(X)$.*

Proof. Let $A = \bigcup_{X \in \mathcal{C}'} X$ and $B = \bigcup_{X \in \mathcal{C}''} X$. If F is pointwise, $\bigcup_{X \in \mathcal{C}'} F(X) = \bigcup_{X \in \mathcal{C}'} \bigcup_{x \in X} f(x) = \bigcup_{x \in A} f(x)$, the same for B. Thus, if $A = B$, the two unions are equal.

Now suppose that F is not pointwise. Property 2 shows that there exist $X \in \mathcal{C}$ and $y \in F(X) \setminus F_\circ(X)$, where $F_\circ(X) = \bigcup_{x \in X} \bigcap_{Y \in \mathcal{C}, x \in Y} F(Y)$, i.e., $\forall x \in X$, $\exists Y_x \in \mathcal{C}$ such that $x \in Y_x$ and $y \notin F(Y_x)$. By construction, $X \subseteq \bigcup_{x \in X} Y_x$ thus $\bigcup_{x \in X} Y_x = X \cup (\bigcup_{x \in X} Y_x)$. But $y \notin \bigcup_{x \in X} F(Y_x)$ while $y \in F(X)$ thus $y \in F(X) \cup (\bigcup_{x \in X} F(Y_x))$, contradiction. □

Property 3 is exactly what we need to prove the formulas. The set $\mathcal{A}$ is the set of iterations in the tile strip to be analyzed. The set $\mathcal{C}$ is the set of all tiles (aligned or unaligned) intersected with $\mathcal{A}$. Since all tiles aligned with $\vec{I}$ form a partition of $\mathcal{A}$, $\cup_{\vec{I}' \prec_{\vec{s}} \vec{I}} \vec{I}' = \cup_{\vec{I}' \sqsubset_{\vec{s}} \vec{I}} \vec{I}'$: it is the set of all points executed before any point in $\vec{I}$. In [3], all written values are supposed to be live-out, so $\mathrm{Out}(\vec{I}) = \mathrm{Write}(\vec{I})$, the values written in $\vec{I}$. In general, if Liveout is the set of all elements live-out of the tile strip, then the right formulas are:

- $\mathrm{Load}(\vec{I}) = \mathrm{In}(\vec{I}) \setminus (\mathrm{In}(\vec{I}' \sqsubset_{\vec{s}} \vec{I}) \cup \mathrm{Write}(\vec{I}' \sqsubset_{\vec{s}} \vec{I}))$ (5)
- $\mathrm{Store}(\vec{I}) = \mathrm{Liveout} \cap (\mathrm{Write}(\vec{I}) \setminus \mathrm{Write}(\vec{I}' \sqsupset_{\vec{s}} \vec{I}))$ (6)

The function Write is, by definition, pointwise, as it is the union, for all points $\vec{i}$ in $\vec{I}$, of the set of values written at iteration $\vec{i}$. Thus, in Eq. (6), one can replace $\sqsupset_{\vec{s}}$ by $\succ_{\vec{s}}$. Also, even if $\vec{I} \mapsto \mathrm{In}(\vec{I})$ may not be pointwise, any element read but not written in $\vec{I}$ is live-in for $\vec{I}$, thus $\mathrm{In}(\vec{I}) \cup \mathrm{Write}(\vec{I}) = \mathrm{Read}(\vec{I}) \cup \mathrm{Write}(\vec{I})$. As the function Read $\cup$ Write is pointwise, $\sqsubset_{\vec{s}}$ can be safely replaced by $\prec_{\vec{s}}$ in Eq. (5) too.

This concludes the proof for the exact case.

3.2 The case of approximations

There are at least four reasons why approximations of the various sets In, Out, Load, Store may be used.

- The execution of S at iteration $\vec{i}$ is not sure, e.g., when it depends on a non-analyzable `if` condition.
- The access functions are not fully analyzable.
- The In/Out sets are approximated on purpose (e.g., they are restricted to polyhedra or hyper-rectangles).
- The Load/Store sets are approximated to make them simpler, or to get transfer sets of some special form.

In the first two cases, the approximation is pointwise, so the Read/Write functions remain pointwise. In the last two cases, it is more likely that the function In $\cup$ Out is not pointwise anymore. We now address these two situations. We first recall the principles stated in [3] to handle approximations, assuming that the sets $\overline{\mathrm{In}}$, $\underline{\mathrm{Out}}$, and $\overline{\mathrm{Out}}$ are given such that $\mathrm{In}(\vec{I}) \subseteq \overline{\mathrm{In}}(\vec{I})$ and $\underline{\mathrm{Out}}(\vec{I}) \subseteq \mathrm{Out}(\vec{I}) \subseteq \overline{\mathrm{Out}}(\vec{I})$.

3.2.1 Non-parametric case

The first step is to define the Store sets, as exactly as possible from the $\overline{\mathrm{Out}}$ sets, i.e., data possibly written:

- $\mathrm{Store}(\vec{I}) = \mathrm{Liveout} \cap (\overline{\mathrm{Out}}(\vec{I}) \setminus \overline{\mathrm{Out}}(\vec{I}' \sqsupset_{\vec{s}} \vec{I}))$ (7)

Then, any over-approximation $\overline{\mathrm{Store}}(\vec{I})$ of $\mathrm{Store}(\vec{I})$ can be used. Eq. (7) means that an element considered live-out of the tile strip and possibly defined is always stored to external memory, in case it is written at runtime. As some elements which are stored may not be actually defined during the execution, they are added to the set of input elements so that their initial values are stored back instead of garbage:

- $\overline{\mathrm{In}}'(\vec{I}) = \overline{\mathrm{In}}(\vec{I}) \cup (\overline{\mathrm{Store}}(\vec{I}) \setminus \underline{\mathrm{Out}}(\vec{I}))$ (8)

The Load sets are then defined, as exactly as possible, from the approximated $\overline{\mathrm{In}}'$, $\underline{\mathrm{Out}}$, and $\overline{\mathrm{Out}}$ sets. Following [3][Thm. 3], approximated loads are valid if for any tile $\vec{I}$:

- $\overline{\mathrm{Ra}}(\vec{I}) = \overline{\mathrm{In}}'(\vec{I}) \setminus \underline{\mathrm{Out}}(\vec{I}' \sqsubset_{\vec{s}} \vec{I}) \subseteq \mathrm{Load}(\vec{I}' \sqsubseteq_{\vec{s}} \vec{I})$. (9)
- $\overline{\mathrm{Out}}(\vec{I}' \sqsubset_{\vec{s}} \vec{I}) \cap \mathrm{Load}(\vec{I}) = \emptyset$. (10)

The first condition means that all data possibly read from outside of the tile strip – the remote accesses $\overline{\mathrm{Ra}}(\vec{I})$ – have to be loaded earlier. The second condition means that data possibly defined earlier in the tile strip should not be loaded, as this could overwrite some valid data. The following equation gives a non-recursive definition of $\mathrm{Load}(\vec{I})$, simpler than the formula given in [3][Thm. 6] (although equivalent):

- $\overline{\mathrm{Ra}}_{\vec{I}} \cap ((\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I}) \setminus (\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I}' \sqsubset_{\vec{s}} \vec{I}))$ (11)

where $\overline{\mathrm{Ra}}_{\vec{I}}$ denotes the set of all remote accesses for the tile strip, i.e., the union of all $\overline{\mathrm{Ra}}(\vec{I}')$, as defined in Eq. (9), where $\vec{I}'$ and $\vec{I}$ belong to the same tiling. Prop. 4 proves that the formula of Eq. (11) defines the loads as expected.

Property 4. *The function $\vec{I} \mapsto Load(\vec{I})$ of Eq. (11) defines valid loads, "exact" w.r.t. the $\overline{In}'$, $\underline{Out}$, and $\overline{Out}$ sets (no useless or redundant loads), and done as late as possible.*

In the next proof and later, we write ΔF the function defined from a function F by $\Delta F(\vec{I}) = F(\vec{I}) \setminus F(\vec{I}' \sqsubset_{\vec{s}} \vec{I})$. By induction, for all $\vec{I}$, $\Delta F(\vec{I}' \sqsubseteq_{\vec{s}} \vec{I}) = F(\vec{I}' \sqsubseteq_{\vec{s}} \vec{I})$ (but the first one is a disjoint union) and, similarly, $\Delta F(\vec{I}' \sqsubset_{\vec{s}} \vec{I}) = F(\vec{I}' \sqsubset_{\vec{s}} \vec{I})$. This implies the recursive relation $\Delta F(\vec{I}) = F(\vec{I}) \setminus \Delta F(\vec{I}' \sqsubset_{\vec{s}} \vec{I})$. Also, $\Delta F(\vec{I}) = F(\vec{I}' \sqsubseteq_{\vec{s}} \vec{I}) \setminus F(\vec{I}' \sqsubset_{\vec{s}} \vec{I})$.

Proof. We now prove Property 4. We first prove that the loads are valid. Eq. (10) is satisfied since $\overline{\mathrm{Out}}(\vec{I}' \sqsubset_{\vec{s}} \vec{I})$ is subtracted in Eq. (11). By defining $F = \overline{\mathrm{In}}' \cup \overline{\mathrm{Out}}$, we get $\mathrm{Load}(\vec{J}) = \overline{\mathrm{Ra}}_{\vec{I}} \cap \Delta F(\vec{J})$ for all $\vec{J}$ aligned with $\vec{I}$, thus $\mathrm{Load}(\vec{J}' \sqsubseteq_{\vec{s}} \vec{J}) = \overline{\mathrm{Ra}}_{\vec{I}} \cap \Delta F(\vec{J}' \sqsubseteq_{\vec{s}} \vec{J}) = \overline{\mathrm{Ra}}_{\vec{I}} \cap F(\vec{J}' \sqsubseteq_{\vec{s}} \vec{J})$. As $\overline{\mathrm{Ra}}(\vec{J}) \subseteq \overline{\mathrm{Ra}}_{\vec{I}}$ and $\overline{\mathrm{Ra}}(\vec{J}) \subseteq \overline{\mathrm{In}}'(\vec{J}) \subseteq F(\vec{J})$, then $\overline{\mathrm{Ra}}(\vec{J}) \subseteq \overline{\mathrm{Ra}}_{\vec{I}} \cap F(\vec{J}' \sqsubseteq_{\vec{s}} \vec{J})$, thus Eq. (9) is satisfied too. Note that the intersection with $\overline{\mathrm{Ra}}_{\vec{I}}$ in $\mathrm{Load}(\vec{I})$ is not needed for correctness but it makes sure there are no useless loads. Also, $\mathrm{Load}(\vec{J}) = \overline{\mathrm{Ra}}_{\vec{I}} \cap (F(\vec{J}) \setminus \Delta F(\vec{J}' \sqsubset_{\vec{s}} \vec{J})) = (\overline{\mathrm{Ra}}_{\vec{I}} \cap F(\vec{J})) \setminus \mathrm{Load}(\vec{J}' \sqsubset_{\vec{s}} \vec{J})$, thus there are no redundant loads. Finally, if $y \in \mathrm{Load}(\vec{J})$, either $y \in \overline{\mathrm{In}}'(\vec{J})$ and y must be loaded before $\vec{J}$ as it may be read in $\vec{J}$, or $y \in \overline{\mathrm{Out}}(\vec{J})$ and it cannot be loaded later or it will overwrite the value possibly written in $\vec{J}$. Loads are thus done as late as possible. □

The mechanism implicit in Eq. (11) is finally simple: unlike for the exact case, a remote access considered as live-in for $\vec{I}$ (i.e., in $\overline{\mathrm{In}}'(\vec{I})$) cannot be loaded just before $\vec{I}$ if it *may* be written earlier (i.e., in $\overline{\mathrm{Out}}(\vec{I}' \sqsubset_{\vec{s}} \vec{I})$). Otherwise, the load will erase the right value if, at runtime, it is actually written earlier. Instead, the trick is to load the element before the first tile $\vec{I}'$ that may write it. This way, either the value is defined locally and the read in $\vec{I}$ gets this value, or it is not and the read gets the original value. Then any over-approximation $\overline{\mathrm{Load}}(\vec{I})$ of this "exact" $\mathrm{Load}(\vec{I})$ can be used (even if it may generate some useless loads) as long as it still satisfies $\overline{\mathrm{Load}}(\vec{I}) \cap \overline{\mathrm{Out}}(\vec{I}' \sqsubset_{\vec{s}} \vec{I}) = \emptyset$.

3.2.2 Parametric case

Now, the goal is to reformulate Eq. (11) so that it can be computed with $\vec{s}$ as parameter. The situation is much more complex than for the exact case (Section 3.1) but, nevertheless, all situations can be handled thanks to an extensive use of the concept of pointwise function.

We first consider the case where the accesses of each iteration $\vec{i}$ are approximated with $\underline{\mathrm{write}}(\vec{i}) \subseteq \mathrm{write}(\vec{i}) \subseteq \overline{\mathrm{write}}(\vec{i})$ and $\mathrm{read}(\vec{i}) \subseteq \overline{\mathrm{read}}(\vec{i})$, with the corresponding pointwise functions $\underline{\mathrm{Write}}$, $\overline{\mathrm{Write}}$, and $\overline{\mathrm{Read}}$. If $\overline{\mathrm{Out}}$, $\overline{\mathrm{In}}$, then Store are directly derived from $\overline{\mathrm{Write}}$ and $\overline{\mathrm{Read}}$, then, as for the exact case, $\overline{\mathrm{Out}}$ and $\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}}$ are pointwise too. Thus, a parametric $\mathrm{Store}(\vec{I})$ can be computed with Eq. (7) with $\succ_{\vec{s}}$ instead of $\sqsupset_{\vec{s}}$. The same is true for the central part of $\mathrm{Load}(\vec{I})$ in Eq. (11) with $\prec_{\vec{s}}$ instead of $\sqsubset_{\vec{s}}$. It remains to compute $\overline{\mathrm{Ra}}_{\vec{I}}$ from $\overline{\mathrm{Ra}}(\vec{I}) = \overline{\mathrm{In}}'(\vec{I}) \setminus \underline{\mathrm{Out}}(\vec{I}' \sqsubset_{\vec{s}} \vec{I})$. As the tiles in $\mathcal{L}$ cover the whole iteration space, $\overline{\mathrm{Ra}}_{\vec{I}}$ is the set of all data that are maybe read (or written for stores) and possibly not written before (i.e., live-in for the tile strip), for the schedule induced by the tiling aligned with $\vec{I}$. But if the mapping θ selected for tiling was considered legal with the same pointwise approximation of reads and writes, then anti, flow, and output dependences are preserved for any shifted tiling, thus $\overline{\mathrm{Ra}}_{\vec{I}}$ does not depend on $\vec{I}$ and is even equal to the live-in data for the tile strip when considering the original order of the code. Thus, it can be easily computed, independently on $\vec{s}$.

The previous approach can be used when Load/Store sets are computed "exactly" but from a pointwise approximation of accesses. We now consider the general case where, in addition to this pointwise approximation, even the sets $\overline{\text{Out}}$, $\overline{\text{In}}$, $\overline{\text{Store}}$, and $\overline{\text{Load}}$ can be over-approximated further, for whatever reason. For example, $\overline{\text{Store}}(\vec{I})$ can contain data that are not even in $\overline{\text{Out}}$ or $\overline{\text{In}}$, and thus not remote in the strict sense. However, transfers still need to be correct. We first consider how to handle $\overline{\text{Out}}$ in Eq. (7) and $\overline{\text{In}}' \cup \overline{\text{Out}}$ in Eq. (11), which, *a priori*, have no reason to be pointwise. We deal with the computation or approximation of $\overline{\text{Ra}}_{\vec{I}}$ later.

The key point is that loading earlier and storing later always keeps correctness. As exploited for Prop. 4, $\text{Load}(\vec{I})$ has the form $\overline{\text{Ra}}_{\vec{I}} \cap \Delta F$ with $\Delta F(\vec{I}) = F(\vec{I}) \setminus F(\vec{I}' \sqsubset_{\vec{s}} \vec{I})$, thus $\Delta F(\vec{I}' \sqsubseteq_{\vec{s}} \vec{I}) = F(\vec{I}' \sqsubseteq_{\vec{s}} \vec{I})$. If we define F° pointwise such that $F \subseteq F^\circ$, then $\Delta F(\vec{I}' \sqsubseteq_{\vec{s}} \vec{I}) \subseteq \Delta F^\circ(\vec{I}' \sqsubseteq_{\vec{s}} \vec{I})$, i.e., possibly more data are loaded (and no load occurs later), thus the first validity condition of Eq. (9) is satisfied with $\overline{\text{Ra}}_{\vec{I}} \cap \Delta F^\circ$. The same is true for $\text{Store}(\vec{I})$ with $\sqsupseteq_{\vec{s}}$, i.e., possibly more data are stored but no store occurs earlier. Finally, Eq. (10) is satisfied too as $\overline{\text{Out}}(\vec{I}' \sqsubset_{\vec{s}} \vec{I}) \subseteq F(\vec{I}' \sqsubset_{\vec{s}} \vec{I}) \subseteq F^\circ(\vec{I}' \sqsubset_{\vec{s}} \vec{I})$, which is subtracted in ΔF°. Thus, such an over-approximation mechanism is always valid.

We now show how to build such a function F° with an additional property that means that loads in ΔF that correspond to "pointwise loads" are still loaded for the same tile with ΔF°, i.e., not earlier. Indeed, the goal is to try to avoid the naive solution where all data are loaded (resp. stored) before (resp. after) the whole computation of the tile strip.

PROPERTY 5. *Let $\mathcal{C}$ be the set of all tiles of size $\vec{s}$ in $\mathbb{Z}^n$ and $F : \mathcal{C} \to \mathcal{P}(\mathcal{B})$. Let F° defined by $F^\circ(\vec{I}) = \cup_{\vec{J},\, \vec{I} \in \vec{J}} F(\vec{J})$, where $\vec{I} \in \vec{J}$ means that $\vec{I}$ belongs to the tile with origin $\vec{J}$. Then $F \subseteq F^\circ$ and F° is pointwise. Moreover, if y is such that $\forall \vec{I},\ y \in F(\vec{I}) \Rightarrow y \in F_\circ(\vec{I})$ ($F_\circ$ is defined in Prop. 2), then $\forall \vec{I},\ y \in \Delta F^\circ(\vec{I}) \Rightarrow y \in \Delta F(\vec{I})$, i.e., over-approximating F by F° does not load "pointwise" elements earlier.*

PROOF. Depending of the context, we use $\vec{I}$ to represent a point in $\mathbb{Z}^n$ but also the tile with origin $\vec{I}$. Of course $F \subseteq F^\circ$ since $\vec{I} \in \vec{I}$. Now, let $f^\circ : \mathbb{Z}^n \to \mathcal{P}(\mathcal{B})$ with $f^\circ(\vec{J}) = F(\vec{J} - \vec{s} + \vec{1})$: $\vec{J}$ is the opposite corner in the tile whose origin is $\vec{J} - \vec{s} + \vec{1}$. Then, $\forall \vec{I} \in \mathbb{Z}^n$, $\cup_{\vec{J} \in \vec{I}} f^\circ(\vec{J}) = \cup_{\vec{J} \in \vec{I}} F(\vec{J} - \vec{s} + \vec{1})$. But $\vec{J} \in \vec{I}$ iff $\vec{I} \in \vec{J}' = \vec{J} - \vec{s} + \vec{1}$. Thus, the previous union is equal to $\cup_{\vec{J}', \vec{I} \in \vec{J}'} F(\vec{J}') = F^\circ(I)$, i.e., F° is pointwise.

Now, suppose that $\forall \vec{I},\ y \in F(\vec{I}) \Rightarrow y \in F_\circ(\vec{I})$. If $y \in F^\circ(\vec{I}' \sqsubseteq_{\vec{s}} \vec{I}) = \cup_{\vec{I}' \sqsubseteq_{\vec{s}} \vec{I}} \cup_{\vec{J},\, \vec{I}' \in \vec{J}} F(J)$, then $y \in F(\vec{J})$ for some $\vec{J}$ and $\vec{I}'$ such that $\vec{I}' \sqsubseteq_{\vec{s}} \vec{I}$, $\vec{I}' \in \vec{J}$. Thus $y \in F_\circ(\vec{J})$ and $y \in f_\circ(x)$ for some $x \in \vec{J}$ because $F_\circ$ is pointwise. Since $F_\circ \subseteq F$ and since the union of tiles $\cup_{\vec{I}' \sqsubseteq_{\vec{s}} \vec{I}} \cup_{\vec{J},\, \vec{I}' \in \vec{J}} \vec{J}$ spans the same set of points as the union of tiles $\cup_{\vec{I}' \sqsubseteq_{\vec{s}} \vec{I}} \vec{I}'$, this shows that $y \in F(\vec{I}' \sqsubseteq_{\vec{s}} \vec{I})$. Remember that for any function G, $\Delta G(\vec{I}) = G(\vec{I}' \sqsubseteq_{\vec{s}} \vec{I}) \setminus G(\vec{I}' \sqsubset_{\vec{s}} \vec{I})$. Thus if $y \in \Delta F^\circ(\vec{I})$, $y \in F^\circ(\vec{I}' \sqsubseteq_{\vec{s}} \vec{I}) \setminus F^\circ(\vec{I}' \sqsubset_{\vec{s}} \vec{I})$, which implies $y \in F(\vec{I}' \sqsubseteq_{\vec{s}} \vec{I})$ (as we just showed) and $y \notin F(\vec{I}' \sqsubset_{\vec{s}} \vec{I})$ (because $F \subseteq F^\circ$). Thus $y \in \Delta F(\vec{I})$. □

The same technique can be used for $\text{Store}(\vec{I})$ but with an expression such as $F^\circ(\vec{I}) = \cup_{\vec{J}, \vec{J} \in \vec{I}} F(\vec{J})$. It remains to see what to do with the set $\overline{\text{Ra}}_{\vec{I}}$. We can compute, with $\vec{s}$ as parameter, $\overline{\text{Ra}}(\vec{I}) = \overline{\text{In}}(\vec{I}) \setminus \underline{\text{Out}}(\vec{I}' \prec_{\vec{s}} \vec{I})$, thus replacing $\sqsubset_{\vec{s}}$ by $\prec_{\vec{s}}$. We get *a priori* a smaller set, which could be problematic because of the intersection in Eq. (11). However, it is still correct and actually, even more precise. Indeed, as Out is exact, $\overline{\text{In}}'(\vec{I}) \setminus \text{Out}(\vec{I}' \sqsubset_{\vec{s}} \vec{I}) = \overline{\text{In}}'(\vec{I}) \setminus \text{Out}(\vec{I}' \prec_{\vec{s}} \vec{I})$ and what is actually important in Eq. (9) is that this set is indeed loaded. Thus, it is enough to consider $\overline{\text{Ra}}(\vec{I}) = \overline{\text{In}}(\vec{I}) \setminus \underline{\text{Out}}(\vec{I}' \prec_{\vec{s}} \vec{I})$ in Eq. (9) as it is a superset. Finally, to compute $\overline{\text{Ra}}_{\vec{I}} = \bigcup_{\vec{J}, \vec{J} - \vec{I} \in \mathcal{L}} \overline{\text{Ra}}(\vec{J})$, we just drop the constraint on the lattice $\mathcal{L}$. If $\overline{\text{Ra}}$ is not pointwise, we get a possibly larger set: this is suboptimal, but correct.

This completes the theory for parametric tiling with inter-tile reuse and approximations. In practice, it needs to be adapted to each approximation scheme. A possible approximation consists in removing in all intermediate computations such as Out, Store, In′, all existential variables (projection) and to manipulate only integer points in polyhedra. Another possibility is to rely on array region analysis techniques [8]. This is left for future work.

4. DOWN TO LOCAL MEMORY SIZES

The interest of computing the Load/Store sets in a parametric fashion is that, now, the size of the resulting local memory (as bounding box with modulo) can also be computed in a parametric fashion. This is almost mandatory in a context such as the one described in [3], for high-level synthesis (HLS) from C to FPGA. Indeed, some manual (though systematic) changes must be done to the tiled code so that it is accepted by the HLS tool. Doing these changes for all interesting tile sizes is not reasonable. Now, with this parametric inter-tile reuse, combined with parametric code generation [20], and buffer sizing [1], one should be able to have a fully automatic scheme, with parametric tile sizes. This also makes the design and use of analytical cost models possible, in particular to explore hierarchical tiling, which impacts the local memory size.

For buffer sizing, we also extended the approach of [1], which requires lifetime information of array elements to be able to compute memory mappings with memory reuse, to the case where $\vec{s}$ is a parameter, and for partial orders of computations, for example those expressing pipeline executions. As for inter-tile reuse, we take into account all tiles, not just those aligned with respect to a given lattice. Again, one can make sure that no rough approximation is performed that would result in an over-estimated memory size. These results are out of the scope of this paper. We only report here some examples, for two schedules, as an illustration. The first one performs all computations in sequence: tiles are serialized and each performs its loads, then its computations, then its stores before a new tile is computed. The second one is a double-buffering-style schedule on each tile strip defined as follows: if $\vec{I}_1$, $\vec{I}_2$, $\vec{I}_3$ are three successive tiles for $\sqsubseteq_{\vec{s}}$, transfers are fully serialized as $\text{Load}(\vec{I}_2) \to \text{Store}(\vec{I}_1) \to \text{Load}(\vec{I}_3) \to \text{Store}(\vec{I}_2) \to \ldots$ in addition to the fact that tile computations are done sequentially following $\sqsubseteq_{\vec{s}}$, and each tile $\vec{I}$ of course loads its set $\text{Load}(\vec{I})$, then computes, then stores its set $\text{Store}(\vec{I})$.

Example (cont'd). Remember the `jacobi_1d_imper` code. It has two parameters N and M that define the loop bounds. With the proposed tiling, there are also two tile size parameters s_1 and s_2. There could be a fifth parameter to specify

each tile strip, but we chose to derive mappings valid for all tile strips (the same for all examples hereafter). After Load/Store analysis, followed by memory folding with modulos, we get (after simplification) the following sizes for `A` and `B`, for the sequential schedule:

- size(`B`) = $\min(N-2, 2M+s_2-1, 2s_1+s_2-1)$
- size(`A`) = $\min(N, 2M+s_2, 2s_1+s_2)$.

and, with the pipeline schedule:

- size(`B`) = $\min(N-2, 2M+2s_2-2, 2s_1+2s_2-2)$.
- size(`A`) = $\min(N, 2M+2s_2, 2s_1+2s_2)$.

These expressions are actually expressed as disjunctions, each term that contributes to the minimum being specified by conditions on parameters. One can also of course easily retrieve (this time in a parametric fashion) the expression of the memory size for the product of 2 polynomials of [3]. □

We are working on a fully-automated implementation of the described algorithm with `isl`. For the moment, we manually adapted an `iscc` script for each `PolyBench` [18] example. The results are given in Table 1. The transformations θ were given by the `isl` scheduler, which gives results similar to those of Pluto [17]. We tiled the largest consecutive tilable dimensions (underlined in Table 1) for which dependences are nonnegative. Some examples were omitted, either because the schedule provided by `isl` did not exhibit any "tileability"[3] – at least without preliminary transformations such as array expansion –, or simply because they had too many instructions[4] or variables[5] to fit in the table. Moreover, for each example, parameters were restricted so that the domain contains at least one strip with at least two consecutive full tiles, and tile sizes are at least 2 (to avoid many special cases that, again, would not fit in the table).

The results shown in the table are the array sizes after memory folding. We computed a memory allocation compatible for all tile strips, depending on the parameters of the program and the counters of the loops surrounding the tiled loops. Another choice could have been to compute a memory allocation depending on the strip, potentially saving space for boundary strips. The memory size was computed for both sequential and pipelined (double buffering) execution with inter-tile data reuse. We are still working on the approximations, not provided in the table, as well as on techniques to speed-up and simplify the expressions that are obtained, i.e., both the expressions of intermediate sets such as $\overline{\mathrm{In}}'$ and the final ones such as $\overline{\mathrm{Load}}$ and memory sizes.

Double buffering, as expected, usually doubles the local memory size in terms of the innermost tile size. Some arrays require almost all data to be live during a strip, and thus cause the whole array to be stored into local memory (e.g., `x` in `trisolv`). Furthermore, modulo allocation has limitations. It is really apparent on `floyd_warshall` where memory conflicts are spread in such a way that only a modulo bigger than $k+1$ and $n-k$ on both dimensions is valid. Thus, while the number of conflicting memory addresses is proportional to the tile area, the allocation is not. A tighter memory allocation could be obtained with a piecewise modulo allocation scheme, allocating accesses to `path`$[i,k]$ and `path`$[k,j]$ differently from the accesses to `path`$[i,j]$.

[3] `durbin`, `ludcmp`, `cholesky` and `symm`

[4] `adi`, `fdtd-apml`, `gramschmidt`, `2mm`, `3mm`, `correlation`, and `covariance`

[5] `bicg`, `gemver`, and `gesummv`

5. CONCLUSION

In this work, we provided the first parametric solution for generating the memory transfers needed when a kernel is offloaded to a distant accelerator, tile by tile after loop tiling, and when all intermediate results are stored locally on the accelerator. For such computations, there is a complete decoupling between loads and stores, and when a value has been defined in a previous tile, it has to be loaded from the local memory and not from the distant memory as this memory is not yet up-to-date. In other words, inter-tile reuse is mandatory. This also saves external communications.

Our solution is parametric in the sense that we derive the set of loads and stores from and to the distant memory with the tile sizes as parameters. Although the direct formulation is quadratic, we can still solve it in an affine way by developing techniques that consider, in the analysis, all (unaligned) possible tiles obtained by translation and not just those that belong to a tiling (partitioning) of the iteration space. We were able to use a similar technique to also parameterize the computations of local memory sizes, thanks to parametric lifetime analysis and folding with modulos, even for pipeline schedules similar to double buffering.

Also, the whole analysis can handle approximations thanks to the introduction of the concept of pointwise functions, well suited to deal with unaligned tiles. We believe that this technique can be used for other applications linked to the extension of the polyhedral model as it turns out to be fairly powerful. Our future work will be to derive efficient approximation techniques, either because the program cannot be fully analyzable, or because approximations can speed-up or simplify the results of the analysis without losing much in terms of memory transfers and/or memory sizes.

Acknowledgements

We would like to thank Sven Verdoolaege for his help in using `isl` and `iscc` as well as for his suggestion that set differences and relations could solve the non-parametric problem as efficiently as linear programming optimizations as in [3].

6. REFERENCES

[1] C. Alias, F. Baray, and A. Darte. Bee+Cl@k: An implementation of lattice-based array contraction in the source-to-source translator Rose. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, San Diego, USA, June 2007.

[2] C. Alias, A. Darte, and A. Plesco. Kernel offloading with optimized remote accesses. Technical Report RR-7697, Inria, July 2011.

[3] C. Alias, A. Darte, and A. Plesco. Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for FPGA. In *Design, Automation and Test in Europe (DATE'13)*, pages 575–580, Grenoble, France, Mar. 2013.

[4] M. M. Baskaran, N. Vasilache, B. Meister, and R. Lethin. Automatic communication optimizations through memory reuse strategies. In *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*, pages 277–278, New Orleans, Louisiana, Feb. 2012.

[5] U. Bondhugula, M. M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan.

Sample	Schedule	Sequential Memory Size	Pipelined Memory Size
Stencils			
fdtd-2d	$S_0(t,j)\mapsto(\underline{t,t,t+j},0)$ $S_1(t,i,j)\mapsto(\underline{t,t+i,t+i+j},1)$ $S_2(t,i,j)\mapsto(\underline{t,t+i,t+i+j},3)$ $S_3(t,i,j)\mapsto(\underline{t,t+i+1,t+i+j+1},2)$	hz$[s_1{+}s_2,\min(s_1,s_2){+}s_3]$ ex$[s_1{+}s_2,\min(s_1,s_2){+}s_3]$ ey$[s_1{+}s_2,\min(s_1,s_2{-}1){+}s_3]$ _fict_$[\min(s_1,s_2)]$	hz$[s_1{+}s_2,\min(s_1,s_2){+}2s_3]$ ex$[s_1{+}s_2,\min(s_1,s_2){+}2s_3]$ ey$[s_1{+}s_2,\min(s_1,s_2){+}2s_3]$ _fict_$[\min(s_1,s_2)]$
jacobi-1d-imper	$S_0(t,i)\mapsto(\underline{t,2t+i},0)$ $S_1(t,j)\mapsto(\underline{t,2t+j+1},1)$	A$[2s_1{+}s_2]$ B$[2s_1{+}s_2{-}1]$	A$[2s_1{+}2s_2]$ B$[2s_1{+}2s_2{-}2]$
jacobi-2d-imper	$S_0(t,i,j)\mapsto(\underline{t,2t+i,2t+i+j},0)$ $S_1(t,i,j)\mapsto(\underline{t,2t+i+1,2t+i+j+1},1)$	A$[2s_1{+}s_2,\min(2s_1,s_2{+}1){+}s_3]$ B$[2s_1{+}s_2{-}1,\min(2s_1,s_2){+}s_3{-}1]$	A$[2s_1{+}s_2,\min(2s_1,s_2{+}1){+}2s_3]$ B$[2s_1{+}s_2{-}1,\min(2s_1,s_2{+}1){+}2s_3{-}2]$
seidel-2d	$S_0(t,i,j)\mapsto(\underline{t,t+i,2t+i+j})$	A$\begin{bmatrix}s_1{+}s_2{+}1,\\ \min(2s_1{+}2,s_1{+}s_2,2s_2{+}2){+}s_3\end{bmatrix}$	A$\begin{bmatrix}s_1{+}s_2{+}1,\\ \min(2s_1{+}2,s_1{+}s_2,2s_2{+}2){+}2s_3\end{bmatrix}$
Medley			
floyd-warshall	$S_0(k,i,j)\mapsto(k,\underline{i,j})$	path$\begin{bmatrix}\max(k{+}1,n{-}k),\\ \max(k{+}1,n{-}k)\end{bmatrix}$	path$\begin{bmatrix}\max(k{+}1,n{-}k),\\ \max(k{+}1,n{-}k,2s_2)\end{bmatrix}$
reg-detect	$S_0(t,j,i,cnt)\mapsto(\underline{t,j{-}i,t{+}i,t{+}cnt},2)$ $S_1(t,j,i)\mapsto(\underline{t,j{-}i,t{+}i,t},4)$ $S_2(t,j,i,cnt)\mapsto(\underline{t,j{-}i,t{+}i,t{+}cnt},3)$ $S_3(t,j,i)\mapsto(\underline{t,j{-}i,t{+}i,len{+}t},0)$ $S_4(t,i)\mapsto(\underline{t,{-}i,t{+}i,len{+}t},5)$ $S_5(t,j,i)\mapsto(\underline{t,j{-}i,t{+}i,len{+}t},1)$	diff$\begin{bmatrix}s_1{+}s_2{+}s_3{-}3,\\ \min(s_1{+}s_3{-}2,s_2),\\ \min(s_1,s_3){+}s_4{-}1)\end{bmatrix}$ path$\begin{bmatrix}\min(s_1{-}1,s_4){+}s_2{+}s_3{-}1,\\ \min(s_1{+}s_3{-}1,s_2,s_3{+}s_4)\end{bmatrix}$ mean$\begin{bmatrix}s_2{+}s_3{-}1,\\ \min(s_2,s_3{-}1)\end{bmatrix}$ sum_tang$\begin{bmatrix}s_1{+}s_2{+}s_3{-}2,\\ \min(s_1{+}s_3{-}1,s_2)\end{bmatrix}$ sum_diff$\begin{bmatrix}s_1{+}s_2{+}s_3{-}2,\\ \min(s_1{+}s_3{-}1,s_2),\\ \min(s_1,s_3){+}s_4\end{bmatrix}$	diff$\begin{bmatrix}s_1{+}s_2{+}s_3{-}3,\\ \min(s_1{+}s_3{-}2,s_2),\\ \min(s_1,s_3){+}s_4{-}1)\end{bmatrix}$ path$\begin{bmatrix}\min(s_1,2s_4){+}s_2{+}s_3{-}1,\\ \min(s_1{+}s_3,s_2,s_3{+}2s_4)\end{bmatrix}$ mean$\begin{bmatrix}s_2{+}s_3{-}1,\\ \min(s_2,s_3{-}1)\end{bmatrix}$ sum_tang$\begin{bmatrix}s_1{+}s_2{+}s_3{-}2,\\ \min(s_1{+}s_3{-}1,s_2)\end{bmatrix}$ sum_diff$\begin{bmatrix}s_1{+}s_2{+}s_3{-}2,\\ \min(s_1{+}s_3{-}1,s_2),\\ \min(s_1,s_3){+}s_4\end{bmatrix}$
Linear algebra solvers			
dynprog	$S_0(iter,i,j)\mapsto(iter,\underline{i,0,j},4)$ $S_1(iter,i,j)\mapsto(iter,\underline{i,0,j},3)$ $S_2(iter,i,j,k)\mapsto(iter,\underline{k,j,i{+}j},1)$ $S_3(iter,i,j)\mapsto(iter,\underline{j,j,i{+}j},2)$ $S_4(iter)\mapsto(iter,\underline{len,len,len},0)$	sum_c$\begin{bmatrix}\min(s_1,s_2{+}s_3{-}1),\\ s_2{+}s_3{-}2,\\ st\end{bmatrix}$ W$\begin{bmatrix}\min(s_1,s_2){+}s_3{-}1,\\ \min(s_1,s_2,s_3)\end{bmatrix}$ c$[len{-}1,len{-}2]$	sum_c$\begin{bmatrix}\min(s_1,s_2{+}2s_3{-}1),\\ s_2{+}2s_3{-}3,\\ st\end{bmatrix}$ W$\begin{bmatrix}\min(s_1,s_2){+}2s_3{-}1,\\ \min(s_1,s_2,2s_3)\end{bmatrix}$ c$[len{-}1,len{-}2]$
lu	$S_0(t,i)\mapsto(\underline{k,k,j},1)$ $S_1(t,i,j)\mapsto(\underline{k,i,j},0)$	A$[n,n]$	A$[n,n]$
Linear algebra kernels			
atax	$S_0(i)\mapsto(\underline{0,i},2)$ $S_1(i)\mapsto(\underline{i,0},0)$ $S_2(i,j)\mapsto(\underline{i,j},1)$ $S_3(i,j)\mapsto(\underline{i,ny{+}j},3)$	A$[s_1,ny]$ x$[s_2]$ y$[ny]$ tmp$[s_1]$	A$[s_1,ny]$ x$[2s_2]$ y$[ny]$ tmp$[s_1]$
doitgen	$S_0(r,q,p)\mapsto(\underline{r,q,p,0},0)$ $S_1(r,q,p,s)\mapsto(\underline{r,q,p{+}s,s},1)$ $S_2(r,q,p)\mapsto(\underline{r,q,p{+}np,np},2)$	A$[s_1,s_2,np]$ sum$[s_1,s_2,s_3{+}s_4{-}1]$ C4$[s_4,s_3]$	A$[s_1,s_2,np]$ sum$[s_1,s_2,s_3{+}2s_4{-}1]$ C4$[2s_4,s_3]$
gemm	$S_0(i,j)\mapsto(\underline{i,j,0},0)$ $S_1(i,j,k)\mapsto(\underline{i,j,k},1)$	A$[s_1,s_3]$ B$[s_3,s_2]$ C$[s_1,s_2]$	A$[s_1,2s_3]$ B$[2s_3,s_2]$ C$[s_1,s_2]$
mvt	$S_0(i,j)\mapsto(1,\underline{i,j})$ $S_1(i,j)\mapsto(0,\underline{i,j})$	for S_0 for S_1 A$[s_1,s_2]$ A$[s_2,s_1]$ x1$[s_1]$ x2$[s_1]$ y_1$[s_2]$ y_2$[s_2]$	for S_0 for S_1 A$[s_1,2s_2]$ A$[2s_2,s_1]$ x1$[s_1]$ x2$[s_1]$ y_1$[2s_2]$ y_2$[2s_2]$
syr2k	$S_0(i,j)\mapsto(\underline{i,j,0},0)$ $S_1(i,j,k)\mapsto(\underline{i,j,k},1)$ $S_2(i,j,k)\mapsto(\underline{i,j,k},2)$	A$[ni,s_3]$ B$[ni,s_3]$ C$[s_1,s_2]$	A$[ni,2s_3]$ B$[ni,2s_3]$ C$[s_1,s_2]$
syrk	$S_0(i,j)\mapsto(\underline{i,j,0},0)$ $S_1(i,j,k)\mapsto(\underline{i,j,k},1)$	A$[ni,s_3]$ C$[s_1,s_2]$	A$[ni,2s_3]$ C$[s_1,s_2]$
trisolv	$S_0(i)\mapsto(\underline{0,i},0)$ $S_1(i,j)\mapsto(\underline{j,i},1)$ $S_2(i)\mapsto(\underline{i,i},2)$	A$[s_2,s_1]$ x$[n]$ c$[s_2]$	A$[2s_2,s_1]$ x$[n]$ c$[2s_2]$
trmm	$S_0(i,j,k)\mapsto(i,\underline{j{+}k,j})$	A$[1,\min(k,s_1{+}s_2{-}1)]$ B$\begin{bmatrix}\max(ni{-}k,k{+}1),\\ \min(ni,s_1{+}k,s_2{+}k)\end{bmatrix}$	A$[1,\min(k,s_1{+}2s_2)]$ B$\begin{bmatrix}\max(ni{-}k,k{+}1),\\ \min(ni,s_1{+}k,2s_2{+}k)\end{bmatrix}$

Table 1: Examples

Automatic transformations for communication minimized parallelization and locality optimization in the polyhedral model. In *17th International Conference on Compiler Construction (CC'08)*, pages 132–146, Budapest, Hungary, Mar. 2008.

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM International Conference on Programming Languages Design and Implementation (PLDI'08)*, pages 101–113, Tucson, Arizona, June 2008.

[7] S. Boppu, F. Hannig, and J. Teich. Loop program mapping and compact code generation for programmable hardware accelerators. In *24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP'13)*, pages 10–17, Washington, DC, June 2013.

[8] B. Creusillet and F. Irigoin. Interprocedural array region analyses. In *Workshop on Languages and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 1996.

[9] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988. Corresponding software tool PIP: `http://www.piplib.org/`.

[10] P. Feautrier and C. Lengauer. The polyhedron model. In D. Padua, editor, *Encyclopedia of Parallel Programming*. Springer, 2011.

[11] G. I. Goumas, M. Athanasaki, and N. Koziris. An efficient code generation technique for tiled iteration spaces. *IEEE Transactions on Parallel and Distributed Systems*, 14(10):1021–1034, 2003.

[12] S. Guelton, R. Keryell, and F. Irigoin. Compilation pour cible hétérogènes: automatisation des analyses, transformations et décisions nécessaires. In *20ème Rencontres Françaises du Parallélisme (Renpar'11)*, Saint Malo, France, May 2011.

[13] A. Hartono, M. M. Baskaran, J. Ramanujam, and P. Sadayappan. DynTile: Parametric tiled loop generation for parallel execution on multicore processors. In *24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10)*, pages 1–12, Atlanta, Georgia, Apr. 2010.

[14] F. Irigoin and R. Triolet. Supernode partitioning. In *15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)*, pages 319–329, San Diego, California, 1988. ACM.

[15] I. Issenin, E. Borckmeyer, M. Miranda, and N. Dutt. DRDU: A data reuse analysis technique for efficient scratch-pad memory management. *ACM Transactions on Design Automation of Electronics Systems (ACM TODAES)*, 12(2), Apr. 2007. Article 15.

[16] M. Kandemir, I. Kadayif, A. Choudhary, J. Ramanujam, and I. Kolcu. Compiler-directed scratch pad memory optimization for embedded multiprocessors. *IEEE Transactions on VLSI Systems*, 12(3):281–287, Mar. 2004.

[17] PLUTO: An automatic polyhedral parallelizer and locality optimizer for multicores. `http://pluto-compiler.sourceforge.net`.

[18] L.-N. Pouchet. PolyBench/C, the polyhedral benchmark suite. `http://sourceforge.net/projects/polybench/`.

[19] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-based data reuse optimization for configurable computing. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'13)*, pages 29–38. ACM, 2013.

[20] L. Renganrayanan, D. Kim, S. V. Rajopadhye, and M. M. Strout. Parameterized tiled loops for free. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)*, pages 405–414, San Diego, California, June 2007.

[21] S. Verdoolaege. isl: An integer set library for the polyhedral model. In K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer, 2010. `http://freecode.com/projects/isl/`.

[22] S. Verdoolaege. Counting affine calculator and applications. In *1st International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, Apr. 2011.

[23] J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

[24] J. Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, 2000.