# Exact and Approximated Data-Reuse Optimizations for Tiling with Parametric Sizes

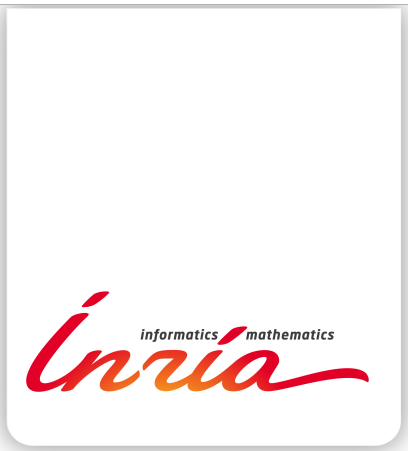
Alain Darte, Alexandre Isoard


# Exact and Approximated Data-Reuse Optimizations for Tiling with Parametric Sizes

**Alain Darte, Alexandre Isoard**

**RESEARCH REPORT**
**N° 8671**
January 2015
Project-Team Compsys

ISSN 0249-6399 ISRN INRIA/RR--8671--FR+ENG




# Exact and Approximated Data-Reuse Optimizations for Tiling with Parametric Sizes

Alain Darte, Alexandre Isoard

Project-Team Compsys

Research Report n° 8671 — January 2015 — 28 pages

This report is an improved version of the work presented at the IMPACT'14 workshop [11] and is published, without the appendix, at the CC'15 conference [12]. Work partially supported by the ManycoreLabs project PIA-6394 led by Kalray.

**RESEARCH CENTRE**
**GRENOBLE – RHÔNE-ALPES**

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

**Abstract:**
Loop tiling is a loop transformation widely used to improve spatial and temporal data locality, to increase computation granularity, and to enable blocking algorithms, which are particularly useful when offloading kernels on computing units with smaller memories. When caches are not available or used, data transfers and local storage must be software-managed, and some useless remote communications can be avoided by exploiting data reuse between tiles. An important parameter of tiling is the sizes of the tiles, which impact the size of the required local memory. However, for most analyses involving 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 with polyhedral optimization techniques.
This paper shows that, when tiles are executed in sequence along tile axes, the parametric (with respect to tile sizes) analysis for inter-tile data reuse is nevertheless possible, i.e., one can 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 sizes for the induced local memories. When approximations of transfers are performed, the situation is much more complex, and involves a careful analysis to guarantee correctness when data are both read and written. We provide the mathematical foundations to make such approximations possible. Combined with hierarchical tiling, this result opens perspectives for the automatic generation of blocking algorithms, guided by parametric cost models, where blocks can be pipelined and/or can contain parallelism. Previous work on FPGAs and GPUs already showed the interest and feasibility of such automation with tiling, but in a non-parametric fashion.

**Key-words:** Loop Tiling, Memory Hierarchy, Software-Managed Caches, Polyhedral Analysis and Optimizations.

# Optimisation pour la réutilisation des données, méthode exacte et méthode approchée, pour le "tiling" avec tailles paramétriques

**Résumé :**

Le tuilage de boucles ("loop tiling") est une transformation de code largement utilisée pour améliorer la localité spatiale et temporelle des données, augmenter la granularité des calculs, et générer des algorithmes par blocs, particulièrement utiles pour déporter des portions de code sur des unités de calcul à petite mémoire. En l'absence de mécanisme automatique de cache, les transferts de données et le stockage local doivent être gérés au niveau logiciel, et certaines de ces communications peuvent être évitées en réutilisant des données entre tuiles. Un paramètre important du tuilage est la taille des tuiles qui influe sur la taille de la mémoire locale requise. Mais, pour la plupart des analyses impliquant plusieurs tuiles, ce qui est le cas pour la réutilisation des données entre tuiles, ces tailles de tuiles induisent des contraintes non-linéaires, sauf si ce sont des constantes numériques. Ceci complique ou empêche une analyse paramétrique par des techniques polyédriques.

Ce rapport montre que, lorsque les tuiles sont exécutées en séquence le long des axes des tuiles, l'analyse paramétrique (au sens des tailles de tuiles), pour la réutilisation des données entre tuiles, est néanmoins possible, c'est-à-dire qu'on peut déterminer, de façon statique et paramétrique, les ensembles de données entrantes et sortantes pour chaque tuile, avec réutilisation entre tuiles, ainsi que des tailles pour les mémoires locales induites. Lorsque des approximations des transferts sont faites, la situation devient plus complexe, et requiert une analyse attentive pour garantir la correction de la transformation dans le cas où des données peuvent être à la fois lues et écrites. Nous apportons des éléments de base théoriques pour rendre de telles approximations possibles. Combinés avec du tuilage hiérarchique, ces résultats ouvrent des perspectives pour la génération automatique d'algorithmes par blocs, guidée par des modèles de coût paramétriques, où les blocs peuvent être pipelinées ou contenir du parallélisme. Des travaux antérieurs pour FPGA et GPU ont déjà montré l'intérêt et la faisabilité d'une telle automatisation par tuilage, mais de façon non-paramétrique.

**Mots-clés :** Tuilage de boucles, hiérarchie mémoire, caches à gestion logicielle, analyses et optimisations polyédriques.

# Exact and Approximated Data-Reuse Optimizations for Tiling with Parametric Sizes

Alain Darte and Alexandre Isoard

Compsys, Computer Science Lab (LIP), CNRS, INRIA, ENS-Lyon, UCB-Lyon
firstname.lastname@ens-lyon.fr

Loop tiling is a loop transformation widely used to improve spatial and temporal data locality, to increase computation granularity, and to enable blocking algorithms, which are particularly useful when offloading kernels on computing units with smaller memories. When caches are not available or used, data transfers and local storage must be software-managed, and some useless remote communications can be avoided by exploiting data reuse between tiles. An important parameter of tiling is the sizes of the tiles, which impact the size of the required local memory. However, for most analyses involving 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 with polyhedral optimization techniques.

This paper shows that, when tiles are executed in sequence along tile axes, the parametric (with respect to tile sizes) analysis for inter-tile data reuse is nevertheless possible, i.e., one can 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 sizes for the induced local memories. When approximations of transfers are performed, the situation is much more complex, and involves a careful analysis to guarantee correctness when data are both read and written. We provide the mathematical foundations to make such approximations possible. Combined with hierarchical tiling, this result opens perspectives for the automatic generation of blocking algorithms, guided by parametric cost models, where blocks can be pipelined and/or can contain parallelism. Previous work on FPGAs and GPUs already showed the interest and feasibility of such automation with tiling, but in a non-parametric fashion.

## 1 Introduction

Todays' hardware diversity increases the need for optimizing compilers and runtime systems. A difficulty when using hardware accelerators (FPGA, GPU, dedicated boards) is to automatically perform kernel/function offloading (a.k.a. outlining as opposed to inlining) between the host and the accelerator, and to organize data transfers between the different memory layers (e.g., in a GPU, from remote to global memory, and from global to shared memory, or even registers). This requires static analysis to identify the kernel input (data read) and output (data produced), and code generation for transfers, synchronizations, and computations. In general, such tasks are done by the programmer who has

to express the communications, to allocate and size the intermediate buffers, and to decompose the kernel into fitting chunks of computation. When each kernel is offloaded in a three-phase process (i.e., upload, compute, store back), such programming remains feasible. For GPUs, developers can use OpenCL or CUDA, or they can rely on higher-level abstractions (e.g., compilation directives as in OpenACC or garbage collector mechanisms as in [9]), static analysis as in OpenMPC [25], runtime approaches as in [24], or mixed compile/runtime optimizations as in [27]. These approaches mainly work at the granularity of variable names, still defined by the programmer, but they can be used to optimize remote transfers when several kernels are successively launched. Things get more complicated when a given kernel is decomposed into smaller kernels (and the initial arrays into array regions) to get blocking algorithms, thanks to *loop tiling*. Indeed, iteration-wise loop analysis and element-wise array analysis are needed to enable intra- and inter-tile data reuse. Moreover, the choice of tile sizes is driven by hardware capabilities such as memory bandwidth, size, and organization, computational power, and such codes are very hard to obtain without automation and some cost model. With this objective, our contribution is a **parametric** (w.r.t. tile sizes) polyhedral analysis technique for **inter-tile data reuse** and a mathematical framework to reason with **approximations** of data accesses and transfers.

Loop tiling is a well-known transformation used to improve data locality [36], increase computation granularity, and control the use and size of local memories for out-of-core computations (see [38] for details on semantics, validity conditions, and code generation). It was first introduced for a set of perfectly nested loops, as a grouping of iterations into *supernodes* [21], which 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 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. Now, tiling is also used for non-perfectly nested loops [7], thanks to multi-dimensional affine loop transformations: as in the perfectly nested 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. Today, loop tiling is still a key loop transformation for performance (speed, memory, locality) and the subject of many new advanced developments, including non-rectangular tiling.

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 [15], 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 sur-

rounding 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 (w.r.t. 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 an $n$-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 corner of 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 [37], the code generation schemes of [21,16,7], the data movement and scratch-pad optimizations of [23,22,6,4,29,35] are not parametric. Recently, efficient code generation for parametric tiling [31,20] as well as some form of symbolic scheduling for tiled codes [8] have been developed.

In the context of high-level synthesis (HLS), inter-tile data reuse was proposed [2] (then automated [4]), as a source-to-source process on top of Altera C2H HLS tool, to offload small computation kernels to FPGAs while optimizing communications from a remote (in this case external) DDR memory. Similar results with data reuse between two successive tiles only were then demonstrated for AutoESL Xilinx tool [29]. Different (and more restricted) forms of inter-tile data reuse were also designed for programmable accelerators such as GPUs [5,18,35]. None of these approaches are parametric w.r.t. tile sizes. In this paper, we show that maximal inter-tile data reuse can be expressed in the parametric case, even in an approximated situation. 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. Our contributions, mostly at the level of code analysis, are the following:

- When read/write accesses can be described in an exact way using polyhedral representations, we show how to derive, thanks to manipulations of integer sets, the copy-in and copy-out sets for each tile, with parametric tile sizes. This gives a full parametric generalization of the inter-tile data reuse of [4].
- We extend this parametric analysis to handle approximations, which make the analysis more complex when some data may be both read and written by the tiles, as loading too much may not be safe. We introduce the concept of *pointwise functions* for which no additional loss of accuracy is induced.
- Using similar analysis principles, we show how such a parametric analysis can be exploited in the following steps of the compilation, in particular to perform parametric array contraction for the definition of local arrays.

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

## 2 Prerequisites

### 2.1 Notations and Definitions

We write all vectors with bold letters such as $\boldsymbol{i}$, with components $i_1, \ldots, i_n$. The vector $\mathbf{0}$ (resp. $\mathbf{1}$) has all components equal to 0 (resp. 1) and $\boldsymbol{a} \circ \boldsymbol{b}$ is the product (component-wise) of $\boldsymbol{a}$ and $\boldsymbol{b}$. We denote by $\preceq$ the lexicographic total order on vectors of arbitrary size and by $\leq$ the component-wise partial order on vectors with same size, defined by $\boldsymbol{i} \leq \boldsymbol{j}$ if and only if (iff) $i_k \leq j_k$ 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 polyhedral iteration domain $\mathcal{D}_S$ (scanned with the iteration vector $\boldsymbol{i}$) is tiled, after a first affine mapping $\boldsymbol{i} \mapsto \boldsymbol{i'} = \theta(S, \boldsymbol{i})$, by canonical tiles whose sizes are specified by a vector $\boldsymbol{s}$. In other words, a point $\boldsymbol{i}$ is mapped to the tile indexed by $\boldsymbol{T}$ where $T_k = \lfloor \frac{i'_k}{s_k} \rfloor$, or equivalently $s_k T_k \leq (\theta(S, \boldsymbol{i}))_k < s_k(T_k + 1)$, for $k \in [1..n]$, i.e., $0 \leq \theta(S, \boldsymbol{i}) - \boldsymbol{s} \circ \boldsymbol{T} \leq \boldsymbol{s} - \mathbf{1}$. Also, we restrict to the case where the original and the tiled programs are both 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 $(\boldsymbol{T}, \boldsymbol{i'})$. The tiled iteration domain for statement $S$ is then:

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

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

*Example* We illustrate the concepts and steps of our technique with the kernel `jacobi_1d_imper` from PolyBench [30], with a time loop, and tiled in 2D. For the code in Fig. 1, the Pluto compiler [28] generates the following mapping:

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

```
for (t = 0; t < M; t++) {
  for (i = 1; i < N - 1; i++)
    S1: B[i] =
        (A[i-1] + A[i] + A[i+1])/3;
  for (j = 1; j < N - 1; j++)
    S2: A[j] = B[j];
}
```

**Figure 1.** Original kernel.

```
for (t = 0; t < M; t++)
  for (i' = 2t+1; i' < 2t + N; i'++) {
    S0: i = i'-2t;
    S1: if (i<N-1) B[i] =
        (A[i-1] + A[i] + A[i+1])/3;
    S2: if (i>1) A[i-1] = B[i-1];
  }
```

**Figure 2.** transformed kernel.

[2] However, parallelism inside a tile is possible, as well as hierarchical tiling, which enables to play with the extent of the tiled domain. Parallel execution are also possible by defining a partial execution order, if execution follows the axes defining tiles. Other cases seem possible but with additional complications and approximations.

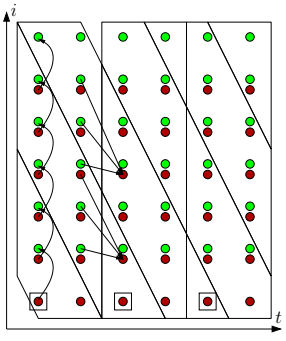
This means shifting $S_2$ by 1 in the $j$ loop, fusing the $i$ and $j$ loops, then skewing by 2 the inner loop, to get the code of Fig. 2. Then, several tiled code generations are possible depending on how iterators are defined and how tiles are aligned, i.e., what the underlying lattice of the tiling is. With the relation $T_k = \lfloor \frac{i_k}{s_k} \rfloor$, tiles are aligned with the canonical basis obtained after the transformation $\theta$ (see Fig. 3 for tiles of size $2 \times 3$, drawn in the original basis to save space). With the "outset" code generation scheme of [31], for tile sizes $s_1 \times s_2$, we get:

```
for (T1 = 0; T1 < M; T1+=s1) {
  lb = 2T1+1-(s2-1); lb = s2*ceiling(lb/s2);
  for (T2 = lb; T2 < 2T1 + N + 2(s1 - 1); T2+=s2)
    for (t=max(0,T1); t<min(M,T1+s1); t++)
      for (i'=max(2t+1,T2); i'<min(2t+N,T2+s2); i'++) {
        S0: i = i'-2t;
        S1: if (i<N-1) B[i] = (A[i-1] + A[i] + A[i+1])/3;
        S2: if (i>1) A[i-1] = B[i-1];
      }
}
```

For our scheme, it would also be valid to shift, after tiling, the inner tile-loop w.r.t. the outer tile-loop, i.e., to move up or down each column in Fig. 3. □

### 2.2 Inter-Tile Data Reuse

The inter-tile reuse problem we formalize here is the kernel offloading with optimized remote accesses presented in [2,4], even if other variations are possible. A kernel is tiled and offloaded, tile by tile, to a computing accelerator (a FPGA in [2,4]). Initially, all data are in remote memory, while all computations are performed on the accelerator. Each tile $\boldsymbol{T}$ consists of three *successive* phases: a *loading* phase where data are copied from remote memory to local memory, enabling burst communications, then a *compute* phase where the original computations corresponding to the tile are performed on the local memory, and finally a *storing* phase where data are copied to remote memory. In addition, all compute (resp. loading and storing) phases are performed in sequence, following the lexicographic order on tile indices. Nevertheless, loads and stores can be done




Non-empty $2 \times 3$ tiles drawn w.r.t the original space. Instruction $S_1$ in red. Instruction $S_2$ in 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 3.** `jacobi1d` kernel and skewed tiling.

concurrently with the computations of other tiles, enabling pipelining, computation/communication overlapping, and execution similar to double buffering. *Inter-tile reuse* makes this possible even when data are both read and written.[3]

Then, the "maximal inter-tile data reuse problem" is to define the loading and storing sets Load($\boldsymbol{T}$) and Store($\boldsymbol{T}$) for each tile $\boldsymbol{T}$ so that a data element is never loaded from remote memory if it is already available in local memory, i.e., if it has already been loaded or computed (as, in this latter case, the remote 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 [4], 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.

Note: there are some similarities with the reuse analysis of [17]. Given a "sliding window" of iterations, one analyzes the data that each iteration needs to bring because they were not already present due to previous iterations in the sliding window. But the communications are not coalesced out of the tile, they are still at the iteration level. In other words, this is a reuse analysis at constant (possibly parametric) distance (the sliding window), but with no granularity or scheduling (through tiling) reorganization, which makes the problem different.

The technique of [4], based on parametric linear programming [14], 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 in a strip, this tends to reduce the size of the local memory. We illustrate this technique again on the `jacobi_1d_imper` example.

*Example (cont'd)* For the tiling of Fig. 3, 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 defined by the tiling) that accesses `A[m]`. This means computing, with $(i_1, i_2) = (t, i)$ and parameters $M$, $N$, $m$, $T_1$, the lexicographic minimum of $(T_2, i'_1, i'_2, k, i_1, i_2)$ in a set defined by a disjunction of two conjunctions of affine inequalities derived from the program (iteration domains and access functions):

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

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 ($(T_1, T_2, i'_1, i'_2) \in \mathcal{T}_S$), and $k = 0$ is the third component of $\theta(S_1, (t, i))$ (i.e., $S_1$ is the first executed statement in the loop body). The second set of constraints corresponds to writes

[3] Without inter-tile reuse, full pipelining of tiles is not always possible if a data is locally written, then read in a subsequent tile. Indeed, one would then need to wait for the data to be stored in remote memory before loading it again. Inter-tile reuse enables to break such a cycle of synchronizations and avoid considering latencies.

in $S_2$ (with $k = 1$, i.e., second executed statement in the loop body). The lexicographic minimum is expressed as a disjunction of cases (a QUAST or quasi affine solution tree [14]). 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 $\boldsymbol{T}$ and array element $m$, which describes $\mathrm{Load}(\boldsymbol{T})$ as a union:

$$\mathrm{Load}(\boldsymbol{T}) = \begin{array}{l} \{m \mid 0 \leq 2T_1 \leq M-1,\ 2 \leq m \leq N-1,\ 1 \leq m + 4T_1 - 3T_2 \leq 3\} \\ \qquad\qquad \cup \\ \{m \mid 0 \leq m \leq 1,\ 3 \leq N,\ 0 \leq 2T_1 \leq M-1,\ -1 \leq 4T_1 - 3T_2 \leq 1\} \end{array}$$

The second set loads the additional `A[0]` and `A[1]` for the unique tile in the strip that contains an iteration $(t, 1)$ on its first column (squares in Fig. 3). □

As can be seen from the inequalities involved in the previous example with $\boldsymbol{s} = (2, 3)$ (and in the definition of $\mathcal{T}_S$), considering the components of the size vector $\boldsymbol{s}$ 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.

## 3 Dealing with Unaligned Tiles

The first key idea to break the non-linearity constraint is to represent each tile not with its tile index $\boldsymbol{T}$ defined earlier, but with the index $\boldsymbol{I}$ 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 $\mathbf{1}$ when $\boldsymbol{T}$ is used and equal to $\boldsymbol{s}$ when $\boldsymbol{I}$ is used. The second difference is that, when $\boldsymbol{I}$ is used instead of $\boldsymbol{T}$, the set of elements $\boldsymbol{i}$ in a tile is affine in $\boldsymbol{s}$: this is the set of all $\boldsymbol{i}$ such that $\boldsymbol{I} \leq \boldsymbol{i} \leq \boldsymbol{I} + \boldsymbol{s} - \mathbf{1}$. In other words, parametric analysis inside a tile is possible. This representation is not new, it is used for analysis in PIPS [19, Fig. 6] and for the parametric code generation [31] used for the tiled code of Section 2.1. However, when reasoning with different tiles, the non-linearity is coming back. Indeed, in a given execution, the tile origins $\boldsymbol{I}$ are restricted to the lattice $\mathcal{L}$ defined by $\boldsymbol{I} \in \mathcal{L}$ iff $\boldsymbol{I} = \boldsymbol{s} \circ \boldsymbol{J}$ for some integer vector $\boldsymbol{J}$. The second key idea is to show how these quadratic constraints can nevertheless be ignored, by reasoning on the set of all tiles of size $\boldsymbol{s}$, not just those restricted to $\mathcal{L}$. The inter-tile reuse problem then becomes (piece-wise) affine in $\boldsymbol{s}$ as we will show.

Note that, with standard conditions for tiling (i.e., when all dependence distances are non-negative along the dimensions being tiled [21]), if a tiling is valid, any translation of it is valid too. In other words, considering all tile origins $\boldsymbol{I} = \boldsymbol{s} \circ \boldsymbol{J} + \boldsymbol{I}_0$ for some vector $\boldsymbol{I}_0$ defines a valid tiling too. This has the same effect as defining the tiling from the shifted mapping $\boldsymbol{i} \mapsto \sigma(S, \boldsymbol{i}) - \boldsymbol{I}_0$ for all $S$. Hereafter, we say that two tiles are *aligned* if they belong to the same tiling.

### 3.1 Exact Approach with Set Equations

In Section 2.2, maximal inter-tile data reuse was expressed as a linear programming optimization, following [4]. It can be equivalently formulated with set equations [3], expressed in terms of $\mathrm{In}(\boldsymbol{T})$ and $\mathrm{Out}(\boldsymbol{T})$, the standard *live-in* and *live-out* sets for tile $\boldsymbol{T}$, as defined for example for array region analysis [10]:

$$\mathrm{Load}(\boldsymbol{T}) = \mathrm{In}(\boldsymbol{T}) \setminus \bigcup_{\boldsymbol{T'} \prec \boldsymbol{T}} (\mathrm{In} \cup \mathrm{Out})(\boldsymbol{T'}) = \mathrm{In}(\boldsymbol{T}) \setminus (\mathrm{In} \cup \mathrm{Out})(\boldsymbol{T'} \prec \boldsymbol{T})\}$$

$$\mathrm{Store}(\boldsymbol{T}) = \mathrm{Out}(\boldsymbol{T}) \setminus \bigcup_{\boldsymbol{T'} \succ \boldsymbol{T}} \mathrm{Out}(\boldsymbol{T'}) = \mathrm{Out}(\boldsymbol{T}) \setminus \mathrm{Out}(\boldsymbol{T'} \succ \boldsymbol{T})$$

Here, as indicated in the previous formulas, $X(\boldsymbol{T'} \prec \boldsymbol{T})$ is a shortcut to denote the union of all sets $X(\boldsymbol{T'})$ for all tiles $\boldsymbol{T'}$ executed before $\boldsymbol{T}$ (lexicographic order) in the same tile strip as $\boldsymbol{T}$. Expressing $X(\boldsymbol{T'} \prec \boldsymbol{T})$ from $X(\boldsymbol{T'})$ is done simply by adding the constraint $\boldsymbol{T'} \prec \boldsymbol{T}$ and specifying that $\boldsymbol{T'}$ is in the strip where reuse is exploited. The previous set equations state that we load what is live-in for $\boldsymbol{T}$ and not previously live-in (redundant load) or live-out (defined locally), and we store what is live-out, but not again live-out later (redundant store). One could expect to rather subtract $\mathrm{Load}(\boldsymbol{T'} \prec \boldsymbol{T})$ from $\mathrm{Load}(\boldsymbol{T})$ and $\mathrm{Store}(\boldsymbol{T'} \succ \boldsymbol{T})$ from $\mathrm{Store}(\boldsymbol{T})$, but such recursive implicit definitions are not usable.

We now rephrase these equations when tiles $\boldsymbol{T}$ are represented by their tile origins $\boldsymbol{I}$ as previously explained. We also consider *all* tiles with size $\boldsymbol{s}$, not just those whose origins belong to the lattice $\mathcal{L}$, i.e., even those that will not be executed in a given tiling. These tiles contain valid iterations (which will be executed as part of an *aligned* tile), but their Load and Store sets will not generate transfers during the execution. We define two relations on tiles:

- $\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I}$ iff $\boldsymbol{I'} \prec \boldsymbol{I}$ and $\boldsymbol{I} - \boldsymbol{I'} \in \mathcal{L}$. This is equivalent to the lexicographic order $\boldsymbol{T'} \prec \boldsymbol{T}$ for the corresponding tile indices.
- $\boldsymbol{I'} \prec_{\boldsymbol{s}} \boldsymbol{I}$ iff, for some $k \in [1..n]$, $I'_i \leq I_i$ for all $i < k$ and $I'_k \leq I_k - s_k$ where $n$ is the dimension of $\boldsymbol{I}$ and $\boldsymbol{I'}$. This is a variation of the lexicographic order.

The standard reflexive extensions $\sqsubseteq_{\boldsymbol{s}}$ and $\preceq_{\boldsymbol{s}}$ of these relations are clearly partial orders. Fig. 4 shows all tile origins $\boldsymbol{I'}$ strictly smaller (in blue) or strictly larger (in red) than the tile origin $\boldsymbol{I}$ (in yellow), for the orders $\sqsubseteq_{\boldsymbol{s}}$ and $\preceq_{\boldsymbol{s}}$. Note that tiles comparable for $\sqsubseteq_{\boldsymbol{s}}$ are always aligned with each other. An alternate, maybe more intuitive, definition of $\prec_{\boldsymbol{s}}$ is as follows: $\boldsymbol{I'} \prec_{\boldsymbol{s}} \boldsymbol{I}$ iff, in the tiling induced by $\boldsymbol{I}$ (the same is true with $\boldsymbol{I'}$, this is symmetric), every point in the tile $\boldsymbol{I'}$ is executed before any point in the tile $\boldsymbol{I}$ (but $\boldsymbol{I}$ and $\boldsymbol{I'}$ may not be aligned).
With tile origins, the previous Load/Store equations can be rewritten as:

$$\mathrm{Load}(\boldsymbol{I}) = \mathrm{In}(\boldsymbol{I}) \setminus (\mathrm{In} \cup \mathrm{Out})(\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I}) \quad (1)$$

$$\mathrm{Store}(\boldsymbol{I}) = \mathrm{Out}(\boldsymbol{I}) \setminus \mathrm{Out}(\boldsymbol{I'} \sqsupset_{\boldsymbol{s}} \boldsymbol{I}) \quad (2)$$

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

$$\mathrm{Load}(\boldsymbol{I}) = \mathrm{In}(\boldsymbol{I}) \setminus (\mathrm{In} \cup \mathrm{Out})(\boldsymbol{I'} \prec_{\boldsymbol{s}} \boldsymbol{I}) \quad (3)$$

$$\mathrm{Store}(\boldsymbol{I}) = \mathrm{Out}(\boldsymbol{I}) \setminus \mathrm{Out}(\boldsymbol{I'} \succ_{\boldsymbol{s}} \boldsymbol{I}) \quad (4)$$

This is not obvious as the contribution of unaligned tiles (i.e., not in the same tiling as $\boldsymbol{I}$) is also subtracted, thus the Load/Store sets could now be too small. Nicely, these sets **only involve affine constraints** as the relation $\prec_{\boldsymbol{s}}$ is, by definition, piece-wise affine (this is also the case for a similar "happens-before" relation defined on iteration points). They can thus be computed with a library such as `isl` [33]. Before proving these formulas, we first illustrate their use.

*Example (cont'd)* The following sets were computed thanks to the `isl` calculator `iscc` [34] with the generic script of Fig. 5, for `jacobi_1d_imper` (see Fig. 3).

$$\begin{aligned}
\mathrm{Load}(\boldsymbol{I}) = \{\mathrm{A}(m) \mid\ & 1 \le m + 2I_1 - I_2 \le s_2,\ s_1 \ge 1,\ I_1 \ge 0,\ m \ge 1,\ I_1 \le -1 + M,\\
& I_2 \ge 2 - s_2 + 2I_1,\ m \le -1 + N,\ N \ge 3\}\\
\cup\ \{\mathrm{A}(m) \mid\ & m \ge 1 + I_2,\ m \ge 1,\ M \ge 1,\ m \le -1 + N,\ I_1 \le -1,\\
& I_1 \ge 1 - s_1,\ I_2 \ge 2 - s_2,\ N \ge 3,\ m \le s_2 + I_2\}\\
\cup\ \{\mathrm{A}(1) \mid\ & I_2 = 1 + 2I_1 \wedge 0 \le I_1 \le -1 + M,\ N \ge 3,\ s_1 \ge 1,\ s_2 \ge 1\}\\
\cup\ \{\mathrm{A}(m) \mid\ & 0 \le m \le 1,\ I_2 = 1 \le s_2,\ 1 - s_1 \le I_1 \le -1,\ M \ge 1,\ N \ge 3\}\\
\cup\ \{\mathrm{A}(0) \mid\ & 0 \le I_1 \le M - 1,\ N \ge 3,\ s_1 \ge 1,\ 1 \le I_2 - 2I_1 \ge 2 - s_2\}\\
\cup\ \{\mathrm{A}(0) \mid\ & 1 - s_1 \le I_1 \le -1,\ M \ge 1,\ N \ge 3,\ I_2 \ge 2 - s_2,\ I_2 \le 0\}
\end{aligned}$$

$$\begin{aligned}
\mathrm{Store}(\boldsymbol{I}) = \{\mathrm{B}(m) \mid\ & m \ge 1,\ m \ge 2 - 2M + s_2 + I_2,\ m \le -2 + N,\\
& I_1 \ge 1 - s_1,\ 2 \le m + 2s_1 + 2I_1 - I_2 \le 1 + s_2,\ s_1 \ge 1\}\\
\cup\ \{\mathrm{B}(m) \mid\ & m \ge 1,\ s_1 \ge 1,\ m \le -2 + N,\ I_1 \le -1 + M,\ m \le 1 - 2M + s_2 + I_2,\\
& m \ge 2 - 2s_1 - 2I_1 + I_2,\ I_1 \ge 1 - s_1,\ M \ge 1,\ m \ge 2 - 2M + I_2\}\\
\cup\ \{\mathrm{A}(m) \mid\ & m \ge 1,\ m \ge 1 - 2M + s_2 + I_2,\ m \le -2 + N,\\
& I_1 \ge 1 - s_1,\ 1 \le m + 2s_1 + 2I_1 - I_2 \le s_2,\ s_1 \ge 1\}\\
\cup\ \{\mathrm{A}(m) \mid\ & m \ge 1,\ s_1 \ge 1,\ m \le -2 + N,\ I_1 \le -1 + M,\ m \le -2M + s_2 + I_2,\\
& m \ge 1 - 2s_1 - 2I_1 + I_2,\ I_1 \ge 1 - s_1,\ M \ge 1,\ m \ge 1 - 2M + I_2\}
\end{aligned}$$

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. 5 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 Load($\boldsymbol{I}$) above is indeed a generalization of the set Load($\boldsymbol{T}$) derived earlier for the canonical tiling with $\boldsymbol{s} = (2, 3)$. It is the complete expression, parameterized by $\boldsymbol{s}$, of all cases,

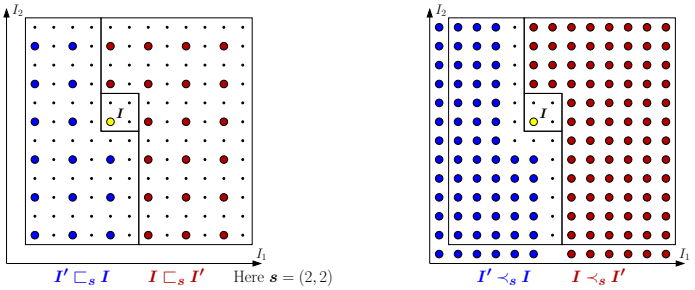



**Figure 4.** Orders $\sqsubset_{\boldsymbol{s}}$ and $\prec_{\boldsymbol{s}}$. Points are tile origins.

```
# 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 5.** Script `iscc` for the Jacobi1D example.

including incomplete tiles, and even tilings obtained by translation of $\mathcal{L}$. Note that simply changing the object `Strip` (see Fig. 5) from `{[I_1,I_2]->[I_1,I_2']}` to `{[I_1,I_2]->[I_1',I_2']}` gives 2D inter-tile reuse, i.e., in the whole space, as the first dimension is not a fixed parameter anymore. The strict order $\prec_s$ is defined by `TiledPrev` while `Load` and `Store`, at the end of the script, express Eq. (3) and (4). Constraints on parameters or on $\boldsymbol{I}$ can be added in `Params`, e.g., to get simplified Load/Store sets for complete tiles, for 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 final expression. □

To prove that we can use $\prec_{\boldsymbol{s}}$ (in Eq (3) and (4)) instead of $\sqsubset_{\boldsymbol{s}}$ (in Eq (1) and (2)), we define the concept of *pointwise functions*. This is a bit more than what we need for the proofs, but this concept makes easier to understand the underlying problems, related to the equality (or not) of some unions of images of sets, which will be even more subtle when dealing with approximations.

### 3.2 Pointwise Functions

If $\mathcal{A}$ is a set, $\mathcal{P}(\mathcal{A})$ denotes the set of subsets of $\mathcal{A}$ (sometimes also written $2^{\mathcal{A}}$). Hereafter, 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 $f : \mathcal{A} \to \mathcal{P}(\mathcal{B})$ such that $\forall X \in \mathcal{C}$, $F(X) = \bigcup_{x \in X} f(x)$.*

In other words, a function $F$ is pointwise if the image of any set where $F$ is defined (not necessarily all sets) can be summarized by the contributions (through $f$) of the points it contains. In our case, $\mathcal{A}$ is the set of iterations in the tile strip to be analyzed and $\mathcal{C}$ is the set of all tiles (aligned or unaligned) intersected with $\mathcal{A}$.

If all written values are live-out, $\mathrm{Out}(\boldsymbol{I}) = \mathrm{Write}(\boldsymbol{I})$, the values written in $\boldsymbol{I}$. Otherwise, this set should be intersected with Liveout, the set of all elements live-out of the tile strip. The function Write is, by definition, pointwise, because it is the union, for all points $\boldsymbol{i}$ in $\boldsymbol{I}$, of the set of values write($\boldsymbol{i}$) written at iteration $\boldsymbol{i}$. Also, even if $\boldsymbol{I} \mapsto \mathrm{In}(\boldsymbol{I})$ may not be pointwise, any element read but not written in $\boldsymbol{I}$ is live-in for $\boldsymbol{I}$, thus $(\mathrm{In} \cup \mathrm{Write})(\boldsymbol{I}) = (\mathrm{Read} \cup \mathrm{Write})(\boldsymbol{I})$, which is pointwise, by introducing read($\boldsymbol{i}$) the set of points read at iteration $\boldsymbol{i}$. We get:

$$\begin{aligned}\mathrm{Load}(\boldsymbol{I}) = \mathrm{In}(\boldsymbol{I}) \setminus (\mathrm{In} \cup \mathrm{Write})(\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I}) &= \mathrm{In}(\boldsymbol{I}) \setminus \bigcup_{\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I}} \bigcup_{\boldsymbol{i} \in \boldsymbol{I'}} (\mathrm{read} \cup \mathrm{write})(\boldsymbol{i}) \\ &= \mathrm{In}(\boldsymbol{I}) \setminus \bigcup_{\boldsymbol{I'} \prec_{\boldsymbol{s}} \boldsymbol{I}} \bigcup_{\boldsymbol{i} \in \boldsymbol{I'}} (\mathrm{read} \cup \mathrm{write})(\boldsymbol{i}) = \mathrm{In}(\boldsymbol{I}) \setminus (\mathrm{In} \cup \mathrm{Write})(\boldsymbol{I'} \prec_{\boldsymbol{s}} \boldsymbol{I})\end{aligned}$$

This is because $\cup_{\boldsymbol{I'} \prec_{\boldsymbol{s}} \boldsymbol{I}} \boldsymbol{I'} = \cup_{\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I}} \boldsymbol{I'}$. Indeed, since all tiles aligned with $\boldsymbol{I}$ form a partition of $\mathcal{A}$, the points covered by the two unions are the same: these are all the points executed before any point in $\boldsymbol{I}$. The same is true for Store($\boldsymbol{I}$), which is equal to Liveout $\cap$ (Write($\boldsymbol{I}$) $\setminus$ Write($\boldsymbol{I'} \sqsupset_{\boldsymbol{s}} \boldsymbol{I}$)), or equivalently equal to Liveout $\cap$ (Write($\boldsymbol{I}$) $\setminus$ Write($\boldsymbol{I'} \succ_{\boldsymbol{s}} \boldsymbol{I}$)). This concludes the proof in the exact case.

In summary, because tiles represent points exactly and because the "happens-before" relation (the fact that a point, resp. a tile, happens, during tiled execution, before another point, resp. tile) can be represented by a piece-wise affine relation, it is possible to perform a parametric analysis of inter-tile data reuse.

The equality of the unions of the images for $\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I}$ and for $\boldsymbol{I'} \prec_{\boldsymbol{s}} \boldsymbol{I}$ is actually a general property, and even a characterization, of pointwise functions. As the following theorem shows, pointwise functions are exactly those that induce the desired "stability" property on union of sets, i.e., if two unions of sets cover

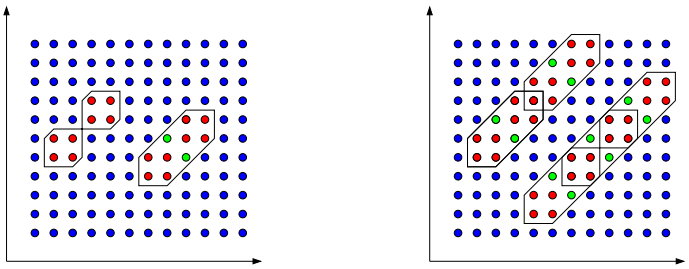


**Figure 6.** "Double squares" (red), F (image of red & green), non pointwise situations.

the same points, then the union of their contributions through $F$ are the same. This a more general property than *distributive functions* (for $\cup$), those for which $F(A \cup B) = F(A) \cup F(B)$ because, in our case, $F(A \cup B)$ may not be defined.

**Theorem 1.** *$F : \mathcal{C} \to \mathcal{P}(\mathcal{B})$ is pointwise if and only if $\forall \mathcal{C}' \subseteq \mathcal{C}$, $\forall \mathcal{C}'' \subseteq \mathcal{C}$, $\bigcup_{X \in \mathcal{C}'} X = \bigcup_{X \in \mathcal{C}''} X \Rightarrow \bigcup_{X \in \mathcal{C}'} F(X) = \bigcup_{X \in \mathcal{C}''} F(X)$.*

Note that the previous property on unions is equivalent to $\forall X \in \mathcal{C}$, $\forall \mathcal{C}' \subseteq \mathcal{C}$, $X \subseteq \bigcup_{X' \in \mathcal{C}'} X' \Rightarrow F(X) \subseteq \bigcup_{X' \in \mathcal{C}'} F(X')$, i.e., if a set is covered by a union of sets, then its image is contained in the union of the images of these sets.

A third equivalent characterization is possible, which explicitely builds a function $f$ for a pointwise function $F$. If $F$ and $G$ are from $\mathcal{C}$ to $\mathcal{P}(\mathcal{B})$, we write $F \subseteq G$ if $\forall X \in \mathcal{C}$, $F(X) \subseteq G(X)$. Theorem 2 also identifies the "largest" pointwise under-approximation of $F$. All missing proofs are provided in the appendix.

**Theorem 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 $F = F_\circ$.*

To get the intuition for these concepts, it is simpler to consider objects more general than rectangular tiles. Let $\mathcal{C}$ be the set of all possible "double squares" (in 2D) defined as two diagonally-neighboring squares as depicted on the left of Fig. 6 (red points in two boxes). Suppose each point $\boldsymbol{i}$ has an image $f(\boldsymbol{i})$. If $F(\boldsymbol{I})$ is defined for a "double-square" $\boldsymbol{I}$ as the union of all $f(\boldsymbol{i})$ for $\boldsymbol{i} \in \boldsymbol{I}$, it is pointwise by definition. Now, suppose $F(\boldsymbol{I})$ is defined as the union of all $f(\boldsymbol{i})$ for $\boldsymbol{i}$ in the convex hull of $\boldsymbol{I}$ (red + green points). The first situation on the right of Fig. 6 shows that each point $\boldsymbol{i}$ is included in two "double-squares" whose images by $F$ have only $f(\boldsymbol{i})$ in common. Thus $F_0$ is not equal to $F$ (the image of green points are missing) unless $f$ has some additional property and, according to Theorem 2, $F$ is not pointwise. The second situation on the right of Fig. 6 shows that a "double-square" is fully contained in two "double-squares", but the image of its green points (if $f$ is injective) is not covered by the image of these two "double-squares" so, according to Theorem 1, $F$ is not pointwise.

### 3.3 The Case of Approximations

We will use the previous properties of pointwise functions for approximations. There are at least four reasons why approximations of the various sets In, Out, Load, and Store may be used in an automatic code analyzer and optimizer.

- The execution of $S$ at iteration $\boldsymbol{i}$ is not guaranteed, for example when it depends on a non-analyzable (e.g., data-dependent) `if` condition.
- The access functions are not fully analyzable (e.g., indirect accesses).
- The In/Out sets are approximated on purpose (e.g., they are restricted to polyhedra or hyper-rectangles) due to the algorithms used for analysis.
- The Load/Store sets are approximated to make them simpler, or to get transfer sets of some special form (e.g., vector/array communications).

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 $\mathrm{In} \cup \mathrm{Out}$ is not pointwise anymore. We first recall and extend the principles stated in [3] for approximations, assuming that the sets $\overline{\mathrm{In}}$, $\underline{\mathrm{Out}}$, and $\overline{\mathrm{Out}}$ are given such that $\mathrm{In}(\boldsymbol{I}) \subseteq \overline{\mathrm{In}}(\boldsymbol{I})$ and $\underline{\mathrm{Out}}(\boldsymbol{I}) \subseteq \mathrm{Out}(\boldsymbol{I}) \subseteq \overline{\mathrm{Out}}(\boldsymbol{I})$. Here, the under-approximations (that could benefit from [10,32]) are not used for correctness, only for accuracy.

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

$$\mathrm{Store}(\boldsymbol{I}) = \mathrm{Liveout} \cap (\overline{\mathrm{Out}}(\boldsymbol{I}) \setminus \overline{\mathrm{Out}}(\boldsymbol{I'} \sqsupset_{\boldsymbol{s}} \boldsymbol{I})) \tag{5}$$

Then, any over-approximation $\overline{\mathrm{Store}}(\boldsymbol{I})$ of $\mathrm{Store}(\boldsymbol{I})$ can be used. Eq. (5) means that a possibly-defined element is always stored to remote memory, in case it is indeed written at runtime. But what if this is not the case? We add it to the set of input elements so that its initial value is stored back instead of garbage:

$$\overline{\mathrm{In}}'(\boldsymbol{I}) = \overline{\mathrm{In}}(\boldsymbol{I}) \cup (\overline{\mathrm{Store}}(\boldsymbol{I}) \setminus \underline{\mathrm{Out}}(\boldsymbol{I})) \tag{6}$$

Following [3, Thm. 3], loads are defined, as exactly as possible, from the sets $\underline{\mathrm{Out}}$, $\overline{\mathrm{Out}}$, and $\overline{\mathrm{In}}'$ (i.e., after $\overline{\mathrm{Store}}$ is defined). They are valid if for any tile $\boldsymbol{I}$:

$$\mathrm{Load}(\boldsymbol{I'} \sqsubseteq_{\boldsymbol{s}} \boldsymbol{I}) \text{ contains } \overline{\mathrm{Ra}}(\boldsymbol{I}) = \overline{\mathrm{In}}'(\boldsymbol{I}) \setminus \underline{\mathrm{Out}}(\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I}) \tag{7}$$

$$\mathrm{Load}(\boldsymbol{I}) \cap \overline{\mathrm{Out}}(\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I}) = \emptyset \tag{8}$$

Eq. (7) means that all data possibly defined outside of the tile strip – the remote accesses $\overline{\mathrm{Ra}}(\boldsymbol{I})$ – have to be loaded before $\boldsymbol{I}$. Eq. (8) means that data possibly defined earlier in the tile strip should not be loaded, as this could overwrite some valid data. Eq. (9) below gives a non-recursive definition of $\mathrm{Load}(\boldsymbol{I})$, simpler (and more usable) than the formula of [3, Thm. 6] (although it is equivalent):

$$\mathrm{Load}(\boldsymbol{I}) = \overline{\mathrm{Ra}}_{\boldsymbol{I}} \cap ((\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\boldsymbol{I}) \setminus (\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I})) \tag{9}$$

where $\overline{\mathrm{Ra}}_{\boldsymbol{I}}$ denotes all remote accesses for the tile strip w.r.t. $\boldsymbol{I}$, i.e., the union of all $\overline{\mathrm{Ra}}(\boldsymbol{I'})$, as defined in Eq. (7), for all $\boldsymbol{I'}$ that belong to the same tiling as $\boldsymbol{I}$.

The mechanism of Eq. (9) is actually simple: unlike for the exact case, a remote access live-in for $\boldsymbol{I}$ (i.e., in $\overline{\text{In}}'(\boldsymbol{I})$) cannot be loaded just before $\boldsymbol{I}$ if it *may* be written earlier (i.e., in $\overline{\text{Out}}(\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I})$). Otherwise, the load will erase the right value if, at runtime, it was indeed written earlier. Instead, the trick is to load the element before the first tile $\boldsymbol{I'}$ that may write it. This way, either the value is defined locally and the read in $\boldsymbol{I}$ gets this value, or it is not defined and the read gets the original value. Thm. 3 (proof in the appendix) states more formally the correctness and exactness of Eq. (9). Then, any over-approximation $\overline{\text{Load}}(\boldsymbol{I})$ of this "exact" Load($\boldsymbol{I}$) can be used (even if it may induce some useless loads) as long as it still satisfies $\overline{\text{Load}}(\boldsymbol{I}) \cap \overline{\text{Out}}(\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I}) = \emptyset$, as required by Eq. (8).

**Theorem 3.** *Eq. (9) defines valid loads, which are "exact" w.r.t. the* $\overline{\text{In}}'$, $\underline{\text{Out}}$, *and* $\overline{\text{Out}}$ *sets (no useless or redundant loads) and performed as late as possible.*

We write $\Delta F$ the function defined from $F$ by $\Delta F(\boldsymbol{I}) = F(\boldsymbol{I}) \setminus F(\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I})$. Then, with $F = \overline{\text{In}}' \cup \overline{\text{Out}}$, we get Load($\boldsymbol{J}$) $= \overline{\text{Ra}}_{\boldsymbol{I}} \cap \Delta F(\boldsymbol{J})$ for all $\boldsymbol{J}$ aligned with $\boldsymbol{I}$.

**Parametric Case.** Our goal is to reformulate Eq. (5) and (9) so that the Store and Load sets can be computed with the tile sizes $\boldsymbol{s}$ as parameter. Can we just replace the order $\sqsubseteq_{\boldsymbol{s}}$ by $\preceq_{\boldsymbol{s}}$ as in the exact case (Section 3.1)? No. Doing so may, in general, be incorrect, resulting in missing loads or stores for $\boldsymbol{I}$, if subtracting the contribution of unaligned tiles (i.e., those that will not be executed) remove additional elements. This is where pointwise functions come, again, into play.

The easy case is when approximations are at the level of iterations, i.e., the accesses of each iteration $\boldsymbol{i}$ are approximated with $\underline{\text{write}}(\boldsymbol{i}) \subseteq \text{write}(\boldsymbol{i}) \subseteq \overline{\text{write}}(\boldsymbol{i})$ and $\text{read}(\boldsymbol{i}) \subseteq \overline{\text{read}}(\boldsymbol{i})$, resulting in pointwise functions $\underline{\text{Write}}$, $\overline{\text{Write}}$, and $\overline{\text{Read}}$. If the sets $\overline{\text{Out}}$, $\overline{\text{In}}$, then Store are derived from $\overline{\text{Write}}$ and $\overline{\text{Read}}$ with no further approximation, then, as for the exact case, $\overline{\text{Out}}$ and $\overline{\text{In}}' \cup \overline{\text{Out}}$ are pointwise too. Thus, a Store($\boldsymbol{I}$) can be computed with Eq. (5), in a parametric way, with $\succ_{\boldsymbol{s}}$ instead of $\sqsupset_{\boldsymbol{s}}$. The same is true for the central part of Load($\boldsymbol{I}$) in Eq. (9) with $\prec_{\boldsymbol{s}}$ instead of $\sqsubset_{\boldsymbol{s}}$. It remains to compute $\overline{\text{Ra}}_{\boldsymbol{I}}$ from $\overline{\text{Ra}}(\boldsymbol{I}) = \overline{\text{In}}'(\boldsymbol{I}) \setminus \underline{\text{Out}}(\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I})$. As the tiles in $\mathcal{L}$ cover the whole iteration space, $\overline{\text{Ra}}_{\boldsymbol{I}}$ 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 $\boldsymbol{I}$. But if the mapping $\theta$ used for tiling was considered legal with the same pointwise approximation of reads and writes, then any shifted tiling (with standard validity conditions) preserves anti, flow, and output dependences, thus $\overline{\text{Ra}}_{\boldsymbol{I}}$ does not depend on $\boldsymbol{I}$. It is even equal to the live-in data for the tile strip when considering the original order of the code and, thus, can be computed, independently on $\boldsymbol{s}$.

The previous approach can be used when Load/Store sets are computed "exactly" but from a pointwise approximation of accesses. We now consider the 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, $\overline{\text{Store}}(\boldsymbol{I})$ 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 $\overline{\mathrm{Out}}$ in Eq. (5) and $\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}}$ in Eq. (9), which, *a priori*, have no reason to be pointwise. We deal with the computation of $\overline{\mathrm{Ra}}_{\boldsymbol{I}}$ later.

We first mention an interesting intermediate situation that works with no further difficulties, even if the approximations are not pointwise. If a pointwise function $F$ is over-approximated through its domain (the iterations) instead of its range (the data), i.e., $\overline{F}(\boldsymbol{I}) = F(\overline{\boldsymbol{I}})$ with $\boldsymbol{I} \subseteq \overline{\boldsymbol{I}}$, then it may be the case that, when computing the unions (either with $\sqsubset_{\boldsymbol{s}}$ or $\prec_{\boldsymbol{s}}$), no new iterations are added with the approximated domains. This is what happens with the approximated "double-squares" of Fig. 6, typical from parallel tiles. Then $\overline{F}(\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I})$ equals:

$$\bigcup_{\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I}} \bigcup_{\boldsymbol{i} \in \overline{\boldsymbol{I'}}} f(\boldsymbol{i}) = \bigcup_{\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I}} \bigcup_{\boldsymbol{i} \in \boldsymbol{I'}} f(\boldsymbol{i}) = \bigcup_{\boldsymbol{I'} \prec_{\boldsymbol{s}} \boldsymbol{I}} \bigcup_{\boldsymbol{i} \in \boldsymbol{I'}} f(\boldsymbol{i}) = \bigcup_{\boldsymbol{I'} \prec_{\boldsymbol{s}} \boldsymbol{I}} \bigcup_{\boldsymbol{i} \in \overline{\boldsymbol{I'}}} f(\boldsymbol{i}) = \overline{F}(\boldsymbol{I'} \prec_{\boldsymbol{s}} \boldsymbol{I})$$

In this case, even without pointwise functions, parametric approximations can be designed, with a careful analysis of the "shape" (the sets $\overline{\boldsymbol{I}}$) of approximations. But, this situation does not cover the case where approximations are made in the range of $F$ and cannot be converted into approximations in the domain of $F$, as it is the case for pointwise functions. We now address this general case.

The key point for approximation is that loading earlier and storing later always keeps correctness. As noticed earlier, $\mathrm{Load}(\boldsymbol{I})$ has the form $\overline{\mathrm{Ra}}_{\boldsymbol{I}} \cap \Delta F(\boldsymbol{I})$ with $\Delta F(\boldsymbol{I}) = F(\boldsymbol{I}) \setminus F(\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I})$, thus $\Delta F(\boldsymbol{I'} \sqsubseteq_{\boldsymbol{s}} \boldsymbol{I}) = F(\boldsymbol{I'} \sqsubseteq_{\boldsymbol{s}} \boldsymbol{I})$. If we define $F^\circ$ pointwise such that $F \subseteq F^\circ$, then $\Delta F(\boldsymbol{I'} \sqsubseteq_{\boldsymbol{s}} \boldsymbol{I}) \subseteq \Delta F^\circ(\boldsymbol{I'} \sqsubseteq_{\boldsymbol{s}} \boldsymbol{I})$, i.e., possibly more data are loaded (but no load is delayed), thus the validity condition of Eq. (7) is satisfied with $\overline{\mathrm{Ra}}_{\boldsymbol{I}} \cap \Delta F^\circ$. The same is true for $\mathrm{Store}(\boldsymbol{I})$ with $\sqsupseteq_{\boldsymbol{s}}$: possibly more data are stored but no store is advanced. Finally, Eq. (8) is satisfied too as $\overline{\mathrm{Out}}(\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I}) \subseteq F(\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I}) \subseteq F^\circ(\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I})$, which is subtracted in $\Delta F^\circ$. Thus, such an over-approximation mechanism (making $F$ bigger) is always valid.

Thm. 4 below shows how to build such a function $F^\circ$ with the additional property that loads in $\Delta F$ that correspond to "pointwise loads" are still loaded for the same tile with $\Delta F^\circ$, i.e., not earlier (thus with no lifetime increase). 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.

**Theorem 4.** *Let $\mathcal{C}$ be the set of all tiles of size $\boldsymbol{s}$ and $F : \mathcal{C} \to \mathcal{P}(\mathcal{B})$. Define $F^\circ$ by $F^\circ(\boldsymbol{I}) = \cup_{\boldsymbol{J},\, \boldsymbol{I} \in \boldsymbol{J}} F(\boldsymbol{J})$, where $\boldsymbol{I} \in \boldsymbol{J}$ means that $\boldsymbol{I}$ is in the tile with origin $\boldsymbol{J}$. Then $F \subseteq F^\circ$ and $F^\circ$ is pointwise. Moreover, if $y$ is such that $\forall \boldsymbol{I},\ y \in F(\boldsymbol{I}) \Rightarrow y \in F_\circ(\boldsymbol{I})$ ($F_\circ$ is defined in Thm. 2), then $\forall \boldsymbol{I},\ y \in \Delta F^\circ(\boldsymbol{I}) \Rightarrow y \in \Delta F(\boldsymbol{I})$, i.e., over-approximating $F$ by $F^\circ$ does not load "pointwise" elements earlier.*

The same technique can be used for $\mathrm{Store}(\boldsymbol{I})$ but with an expression such as $F^\circ(\boldsymbol{I}) = \cup_{\boldsymbol{J},\boldsymbol{J} \in \boldsymbol{I}} F(\boldsymbol{J})$. It remains to see what to do with the set $\overline{\mathrm{Ra}}_{\boldsymbol{I}}$. We can compute, with $\boldsymbol{s}$ as parameter, $\overline{\mathrm{Ra}}(\boldsymbol{I}) = \overline{\mathrm{In}}(\boldsymbol{I}) \setminus \underline{\mathrm{Out}}(\boldsymbol{I'} \prec_{\boldsymbol{s}} \boldsymbol{I})$, thus replacing $\sqsubset_{\boldsymbol{s}}$ by $\prec_{\boldsymbol{s}}$. We get *a priori* a smaller set, which could be problematic because of the intersection in Eq. (9). However, it is still correct and, actually, even more precise. Indeed, as Out is exact, we have $\overline{\mathrm{In}}'(\boldsymbol{I}) \setminus \mathrm{Out}(\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I}) = \overline{\mathrm{In}}'(\boldsymbol{I}) \setminus \mathrm{Out}(\boldsymbol{I'} \prec_{\boldsymbol{s}} \boldsymbol{I})$ and what is actually important in Eq. (7) is that this set is indeed

loaded. Thus, considering $\overline{\mathrm{Ra}}(\boldsymbol{I}) = \overline{\mathrm{In}}(\boldsymbol{I}) \setminus \underline{\mathrm{Out}}(\boldsymbol{I'} \prec_{\boldsymbol{s}} \boldsymbol{I})$ in Eq. (7) is fine as it is a superset. Finally, to compute $\overline{\mathrm{Ra}}_{\boldsymbol{I}} = \bigcup_{\boldsymbol{J},\boldsymbol{J}-\boldsymbol{I}\in\mathcal{L}} \overline{\mathrm{Ra}}(\boldsymbol{J})$, we drop the lattice constraint. If $\overline{\mathrm{Ra}}$ 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 but it still provides some general mathematical means to reason on the correctness of approximations for parametric tiling. A possible approximation (to reduce complexity) 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 [10]. This is left for future work. We point out however that generalizing such a parametric inter-tile reuse to more general tilings, where tiles (rectangular or not) are not executed following the axes that define them, will be more difficult if the iteration space covered by tiles that "happen before" a given tile cannot be defined by a piece-wise affine relation. One can still define approximations, even not necessarily pointwise, as long as $(\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\boldsymbol{I'} \prec_{\boldsymbol{s}} \boldsymbol{I}) = (\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I})$ (and similar equalities), as illustrated with the "double-squares" of Fig. 6. However such approximations are more difficult to define systematically and may require unacceptable (i.e., too rough) additional over-approximations.

## 4 Next Step: Deriving Local Memory Sizes

One of the interests of computing the Load/Store sets in a parametric fashion is that, now, the size of the resulting local memory (e.g., obtained by bounding boxes or lattice-based array contraction [13]) can also be computed in a parametric fashion. Such a parametric scheme seems almost mandatory in a context such as described in [4,29], for HLS from C to FPGA. Indeed, as explained in [4], 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. Also, as explained in [29], identifying the right tile sizes may require executions of multiple scenarios. Parametric code generation would help speeding up such a design space exploration. With this parametric inter-tile reuse, combined with parametric code generation [31] and buffer sizing [1], one should be able to derive 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.

To illustrate such applications, we extended the buffer sizing of [1] – which requires lifetime information of array elements to use memory reuse for array contraction – to the case where $\boldsymbol{s}$ is a parameter, and for partial orders of computations, e.g., those expressing pipeline executions. As for inter-tile reuse, we consider all tiles, not just those aligned w.r.t. 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 illustration. The first schedule performs all computations in sequence: tiles are serialized and each tile performs its loads, then its computations, then its stores before a new tile is computed. The second one is a double-buffering-style schedule (in each tile strip) defined with the following precedences: a) if $\boldsymbol{I}_1$, $\boldsymbol{I}_2$, $\boldsymbol{I}_3$ are three successive tiles for $\sqsubseteq_{\boldsymbol{s}}$, transfer requests are serialized as $\text{Load}(\boldsymbol{I}_2) \to \text{Store}(\boldsymbol{I}_1) \to \text{Load}(\boldsymbol{I}_3) \to \text{Store}(\boldsymbol{I}_2) \to \ldots$, b) tile computations are done sequentially following $\sqsubseteq_{\boldsymbol{s}}$, and c) each tile $\boldsymbol{I}$ loads its set $\text{Load}(\boldsymbol{I})$, then computes, then stores its set $\text{Store}(\boldsymbol{I})$. All other overlappings (in particular parallelism between computations and transfers) can arise at runtime, achieving a kind of double-buffering-style computation.

*Example (cont'd)* The `jacobi_1d_imper` code of Fig. 1 has two parameters $N$ and $M$ defining the loop bounds. The proposed tiling has also two tile size parameters $s_1$ and $s_2$. There could be a 5th parameter to specify each tile strip, but we chose to derive mappings valid for all tile strips (as for all examples hereafter). After Load/Store analysis and memory folding with modulos, we get (after simplification) to following sizes for `A` and `B`, for the sequential schedule:

- $\text{size}(\texttt{B}) = \min(N-2, 2M+s_2-1, 2s_1+s_2-1)$.
- $\text{size}(\texttt{A}) = \min(N, 2M+s_2, 2s_1+s_2)$.

and, with the pipeline schedule:

- $\text{size}(\texttt{B}) = \min(N-2, 2M+2s_2-2, 2s_1+2s_2-2)$.
- $\text{size}(\texttt{A}) = \min(N, 2M+2s_2, 2s_1+2s_2)$.

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 analyzed in [4]. □

We are currently working on an automated implementation of the described algorithm with `isl`, with an integration into PPCG [35], an optimizer for GPUs. For the moment, we manually adapted an `iscc` script for each `PolyBench` [30] example. The results are given in Table 1 (see appendix). The transformations $\theta$ were given by the `isl` scheduler, which gives results similar to those of Pluto [28]. 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"[4] – at least without preliminary transformations such as array expansion –, or simply because they had too many instructions[5] or variables[6] and will not fit in the table anyway (these examples were not tried: they may – but maybe not – reveal complexity issues, which will be explored with the automatic implementation in `isl`, as well as different approximation schemes). Moreover, in Table 1,

[4] Kernels `durbin`, `ludcmp`, `cholesky`, and `symm`
[5] Kernels `adi`, `fdtd-apml`, `gramschmidt`, `2mm`, `3mm`, `correlation`, and `covariance`
[6] Kernels `bicg`, `gemver`, and `gesummv`

parameters were restricted so that each kernel domain contains at least one strip with at least two consecutive full tiles, and tile sizes are at least 2: this avoids many special cases (their generation is possible however) that, again, would not fit in the table.

The results shown in Table 1 are the array sizes after memory folding. We computed a memory allocation compatible for all tile strips, depending on the program parameters 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, using the successive modulo approach of Lefebvre and Feautrier [26]. We are still working on the approximations, not provided in the table, as well as on techniques to speed-up and simplify both the expressions of intermediate sets such as $\overline{\text{In}}'$ and the final ones such as $\overline{\text{Load}}$ 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, thus causing 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 $\texttt{path}[i,k]$ and $\texttt{path}[k,j]$ differently from the accesses to $\texttt{path}[i,j]$. More generally, it is more likely that automating such schemes, with pipelining, parallelism, and hierarchical transfers, will require more advanced communication and allocation strategies.

## 5 Conclusion

This work provides the first parametric solution for generating memory transfers with data reuse 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. In this case, when a value has been loaded or defined in a previous tile, it is read from the local memory and not loaded from the remote memory, which is not yet up-to-date. Our solution is parametric in the sense that we can derive the copy-in/copy-out sets for each tile, exploiting both intra- and inter-tile data reuse, with tile sizes as parameters. Such a result is quite surprising as parametric tiling is often considered as necessarily involving quadratic constraints, i.e., not analyzable within the polyhedral model. We solve it in an affine way with a different reasoning that considers, in the analysis, all (unaligned) possible tiles obtained by translation and not just the tiles of a given tiling. A similar technique can be used to parameterize the computations of local memory sizes, thanks to parametric lifetime analysis and array contraction with parametric modulos (or bounded boxes), even for pipeline schedules similar to double buffering.

This reasoning can also be extended in the case of approximations, which are needed when dealing with kernels that are not fully affine, or because approximations of communications are desired for code simplicity, complexity issues, or architectural constraints (e.g., vector communication). The main difficulty with approximation is that, when some data can be both read and written, loading blindly from remote memory, in an over-approximate way, is not safe as it may not be up-to-date. We address the problem thanks to the introduction of the concept of pointwise functions, well suited to deal with unaligned tiles. This concept may be useful for other applications linked to extensions of the polyhedral model as it turns out to be fairly powerful. For the moment, our study provides the mathematical foundations to discuss the correctness of approximation techniques that still need to be designed, even if some simple schemes are already possible. The full implementation, from the analysis down to code generation, is still a development challenge. Full experiments will be needed to validate the approach and help designing cost models for tile size selection. Nevertheless, the different performance studies with inter-tile data reuse for GPUs [17,18,35] or FPGAs [4,29], for non-parametric tile sizes, already demonstrate its interest.

"Guessing" the right size of the tiles can be laborious, especially when dealing with multi-level tiling and multi-level caches. The search space can become so wide that even iterative compilation might not be sufficient. As said, our parametric technique provides a direct expression of the copy-in/copy-out sets for each tile, and can then be used for performing array contraction on the accelerator still in a parametric fashion. It is only with such a parametric description that we can hope to design cost models for compile-time tile size selection in the context of tiling with inter-tile data reuse. Such static compilation techniques could then be integrated on top of intermediate languages such as OpenACC or OpenCL, or directly generate lower-level code, providing an automatic way to derive blocking algorithms for accelerators. Other applications are certainly possible, as soon as data reuse among tiles or pages has to be analyzed.

**Acknowledgements** We 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 [4].

## References

1. Christophe Alias, Fabrice Baray, and Alain 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, June 2007.
2. Christophe Alias, Alain Darte, and Alexandru Plesco. Optimizing DDR-SDRAM communications at C-level for automatically-generated hardware accelerators. An experience with the Altera C2H HLS tool. In *21st IEEE Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP'10)*, pages 329–332, Rennes, July 2010. IEEE Computer Society.

3. Christophe Alias, Alain Darte, and Alexandru Plesco. Kernel offloading with optimized remote accesses. Technical Report RR-7697, Inria, July 2011.
4. Christophe Alias, Alain Darte, and Alexandru Plesco. Optimizing remote accesses for offloaded kernels: Application to HLS for FPGA. In *Design, Automation and Test in Europe (DATE'13)*, pages 575–580, Grenoble, March 2013.
5. Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'08)*, pages 1–10, 2008.
6. Muthu Manikandan Baskaran, Nicolas Vasilache, Benoît Meister, and Richard 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, February 2012.
7. Uday Bondhugula, Albert 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, June 2008.
8. Srinivas Boppu, Franck Hannig, and Jürgen Teich. Loop program mapping and compact code generation for programmable hardware accelerators. In *24th Int. Conference on Application-Specific Systems, Architectures and Processors (ASAP'13)*, pages 10–17, Washington, DC, June 2013.
9. Mathias Bourgoin, Emmanuel Chailloux, and Jean Luc Lamotte. Efficient abstractions for GPGPU programming. *International Journal of Parallel Programming*, 42(4):583–600, 2014.
10. Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC'96)*, volume 1033 of *LNCS*, pages 46–60. Springer, 1996.
11. Alain Darte and Alexandre Isoard. Parametric tiling with inter-tile data reuse. In Sanjay Rajopadhye and Sven Verdoolaege, editors, *4th International Workshop on Polyhedral Compilation Techniques (IMPACT'14)*, Vienna, Austria, January 2014.
12. Alain Darte and Alexandre Isoard. Exact and Approximated Data-Reuse Optimizations for Tiling with Parametric Sizes. In *24th International Conference on Compiler Construction (CC'15), part of ETAPS'15*, London, United Kingdom, April 2015.
13. Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, October 2005.
14. Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988. Corresponding software tool PIP: `http://www.piplib.org/`.
15. Paul Feautrier and Christian Lengauer. The polyhedron model. In David Padua, editor, *Encyclopedia of Parallel Programming*. Springer, 2011.
16. Georgios I. Goumas, Maria Athanasaki, and Nectarios Koziris. An efficient code generation technique for tiled iteration spaces. *IEEE Transactions on Parallel and Distributed Systems*, 14(10):1021–1034, 2003.
17. Armin Größlinger. Precise management of scratchpad memories for localising array accesses in scientific codes. In *18th International Conference on Compiler Construction (CC'09)*, pages 236–250, 2009.
18. Serge Guelton, Mehdi Amini, and Béatrice Creusillet. Beyond do loops: Data transfer generation with convex array regions. In Hironori Kasahara and Keiji

Kimura, editors, *International Workshop on Languages and Compilers for Parallel Computing (LCPC'13)*, volume 7760 of *LNCS*, pages 249–263. Springer, 2013.

19. Serge Guelton, Ronan Keryell, and François 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.
20. Albert Hartono, Muthu Manikandan 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, April 2010.
21. François Irigoin and Rémi Triolet. Supernode partitioning. In *15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)*, pages 319–329, San Diego, California, 1988. ACM.
22. Ilya Issenin, Erik Borckmeyer, Miguel Miranda, and Nikil 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), April 2007. Article 15.
23. Mahmut Kandemir, Ismail Kadayif, Alok Choudhary, J. Ramanujam, and Ibrahim Kolcu. Compiler-directed scratch pad memory optimization for embedded multiprocessors. *IEEE Transactions on VLSI Systems*, 12(3):281–287, March 2004.
24. Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In *16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*, pages 277–288. ACM, 2011.
25. Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, pages 1–11. IEEE Computer Society, 2010.
26. Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24:649–671, 1998.
27. Sreepathi Pai, R. Govindarajan, and Matthew J. Thazhuthaveetil. Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme. In *21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*, pages 33–42. ACM, 2012.
28. PLUTO: An automatic polyhedral parallelizer and locality optimizer for multicores. `http://pluto-compiler.sourceforge.net`.
29. Louis-Noël Pouchet, Peng Zhang, P. Sadayappan, and Jason 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.
30. Louis-Noël Pouchet. PolyBench/C, the polyhedral benchmark suite. `http://sourceforge.net/projects/polybench/`.
31. Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay V. Rajopadhye, and Michelle Mills Strout. Parameterized tiled loops for free. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)*, pages 405–414, San Diego, June 2007.
32. Ramakrishna Upadrasta and Albert Cohen. Sub-polyhedral scheduling using (unit-) two-variable-per-inequality polyhedra. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*, pages 483–496, Roma, Italy, January 2013.

33. Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *LNCS*, pages 299–302. Springer, 2010. `http://freecode.com/projects/isl/`.
34. Sven Verdoolaege. Counting affine calculator and applications. In *1st International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, April 2011.
35. Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.
36. Michael Wolf and Monica Lam. A data locality optimizing algorithm. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, pages 30–44, Toronto, Ontario, Canada, 1991. ACM.
37. Jingling Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.
38. Jingling Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, 2000.

## A Proofs for Pointwise Functions

We first recall the definition of pointwise functions.

**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 $f : \mathcal{A} \to \mathcal{P}(\mathcal{B})$ such that $\forall X \in \mathcal{C}$, $F(X) = \bigcup_{x \in X} f(x)$.*

Thus, $F$ is pointwise if the image of any set where $F$ is defined can be summarized by the contributions (through $f$) of the points it contains. We first prove Theorem 2, then Theorem 1 (theorems are presented in the opposite order in the text). If $F$ and $G$ are from $\mathcal{C}$ to $\mathcal{P}(\mathcal{B})$, we write $F \subseteq G$ if $\forall X \in \mathcal{C}$, $F(X) \subseteq G(X)$.

**Theorem 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 the function $F$ is pointwise, $F \subseteq F_\circ$, thus $F = F_\circ$ since $F_\circ \subseteq F$. Conversely, if $F = F_\circ$, $F$ is pointwise with $f_\circ$. □

**Theorem 1.** *$F : \mathcal{C} \to \mathcal{P}(\mathcal{B})$ is pointwise if and only if $\forall \mathcal{C}' \subseteq \mathcal{C}$, $\forall \mathcal{C}'' \subseteq \mathcal{C}$, $\bigcup_{X \in \mathcal{C}'} X = \bigcup_{X \in \mathcal{C}''} X \Rightarrow \bigcup_{X \in \mathcal{C}'} F(X) = \bigcup_{X \in \mathcal{C}''} F(X)$.*

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

Now suppose that $F$ is not pointwise. Theorem 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. □

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

**Theorem 3.** *Eq. (9) defines valid loads, which are "exact" w.r.t. the $\overline{\mathrm{In}}'$, $\underline{\mathrm{Out}}$, and $\overline{\mathrm{Out}}$ sets (no useless or redundant loads) and performed as late as possible.*

*Proof.* We first prove that the loads are valid. First, Eq. (8) is satisfied since $\overline{\mathrm{Out}}(\boldsymbol{I'} \sqsubset_{\boldsymbol{s}} \boldsymbol{I})$ is subtracted in Eq. (9). By defining $F = \overline{\mathrm{In}}' \cup \overline{\mathrm{Out}}$, we get $\mathrm{Load}(\boldsymbol{J}) = \overline{\mathrm{Ra}}_{\boldsymbol{I}} \cap \Delta F(\boldsymbol{J})$ for all $\boldsymbol{J}$ aligned with $\boldsymbol{I}$, thus $\mathrm{Load}(\boldsymbol{J'} \sqsubseteq_{\boldsymbol{s}} \boldsymbol{J}) = \overline{\mathrm{Ra}}_{\boldsymbol{I}} \cap$

$\Delta F(\boldsymbol{J'} \sqsubseteq_{\boldsymbol{s}} \boldsymbol{J}) = \overline{\mathrm{Ra}}_{\boldsymbol{I}} \cap F(\boldsymbol{J'} \sqsubseteq_{\boldsymbol{s}} \boldsymbol{J})$. As $\overline{\mathrm{Ra}}(\boldsymbol{J}) \subseteq \overline{\mathrm{Ra}}_{\boldsymbol{I}}$ and $\overline{\mathrm{Ra}}(\boldsymbol{J}) \subseteq \overline{\mathrm{In}}'(\boldsymbol{J}) \subseteq F(\boldsymbol{J})$, then $\overline{\mathrm{Ra}}(\boldsymbol{J}) \subseteq \overline{\mathrm{Ra}}_{\boldsymbol{I}} \cap F(\boldsymbol{J'} \sqsubseteq_{\boldsymbol{s}} \boldsymbol{J})$, thus Eq. (7) is satisfied too. Note that the intersection with $\overline{\mathrm{Ra}}_{\boldsymbol{I}}$ in $\mathrm{Load}(\boldsymbol{I})$ is not needed for correctness but it makes sure there are no useless loads. Also, $\mathrm{Load}(\boldsymbol{J}) = \overline{\mathrm{Ra}}_{\boldsymbol{I}} \cap (F(\boldsymbol{J}) \setminus \Delta F(\boldsymbol{J'} \sqsubset_{\boldsymbol{s}} \boldsymbol{J})) = (\overline{\mathrm{Ra}}_{\boldsymbol{I}} \cap F(\boldsymbol{J})) \setminus \mathrm{Load}(\boldsymbol{J'} \sqsubset_{\boldsymbol{s}} \boldsymbol{J})$, thus there are no redundant loads. Finally, if $y \in \mathrm{Load}(\boldsymbol{J})$, either $y \in \overline{\mathrm{In}}'(\boldsymbol{J})$ and $y$ must be loaded before $\boldsymbol{J}$ as it may be read in $\boldsymbol{J}$, or $y \in \overline{\mathrm{Out}}(\boldsymbol{J})$ and it cannot be loaded later or it will overwrite the value possibly written in $\boldsymbol{J}$. Loads are thus done as late as possible. □

**Theorem 4.** *Let $\mathcal{C}$ be the set of all tiles of size $\boldsymbol{s}$ and $F : \mathcal{C} \to \mathcal{P}(\mathcal{B})$. Define $F^\circ$ by $F^\circ(\boldsymbol{I}) = \cup_{\boldsymbol{J},\, \boldsymbol{I} \in \boldsymbol{J}} F(\boldsymbol{J})$, where $\boldsymbol{I} \in \boldsymbol{J}$ means that $\boldsymbol{I}$ is in the tile with origin $\boldsymbol{J}$. Then $F \subseteq F^\circ$ and $F^\circ$ is pointwise. Moreover, if $y$ is such that $\forall \boldsymbol{I},\, y \in F(\boldsymbol{I}) \Rightarrow y \in F_\circ(\boldsymbol{I})$ ($F_\circ$ is defined in Thm. 2), then $\forall \boldsymbol{I},\, y \in \Delta F^\circ(\boldsymbol{I}) \Rightarrow y \in \Delta F(\boldsymbol{I})$, i.e., over-approximating $F$ by $F^\circ$ does not load "pointwise" elements earlier.*

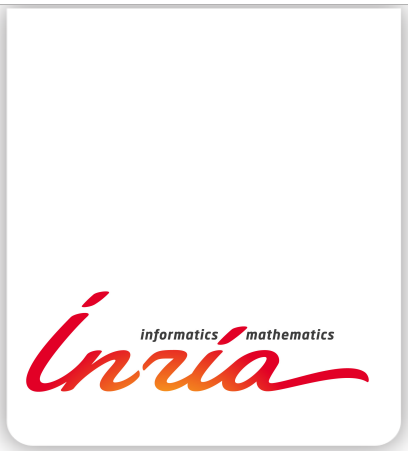
*Proof.* Depending of the context, we use $\boldsymbol{I}$ to represent a point in $\mathbb{Z}^n$ but also the tile with origin $\boldsymbol{I}$. Of course $F \subseteq F^\circ$ since $\boldsymbol{I} \in \boldsymbol{I}$. Now, let $f^\circ : \mathbb{Z}^n \to \mathcal{P}(\mathcal{B})$ with $f^\circ(\boldsymbol{J})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

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

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



**Table 1.** Memory sizes for different arrays, with sequential & pipelined schedules (`PolyBench` examples).




**RESEARCH CENTRE**
**GRENOBLE – RHÔNE-ALPES**

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399