Kernel Offloading with Optimized Remote Accesses

Christophe Alias, Alain Darte, Alexandru Plesco

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Kernel Offloading with Optimized Remote Accesses

Christophe Alias — Alain Darte — Alexandru Plesco

N° 7697 — version 1

initial version July 2011 — revised version Juillet 2011

Domaine 2

ISSN 0249-6399 ISRN INRIA/RR--7697--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Kernel Offloading with Optimized Remote Accesses

Christophe Alias* , Alain Darte* , Alexandru Plesco*

Domaine : Algorithmique, programmation, logiciels et architectures
Équipe-Projet COMPSYS

Rapport de recherche n° 7697 — version 1 — initial version July 2011 —
revised version Juillet 2011 — 29 pages

Abstract: The use of hardware accelerators, e.g., with GPGPUs or customized circuits using FPGAs, are particularly interesting for accelerating data- and compute-intensive applications. However, to get high performance when offloading computation kernels to such an acceleration platform, it is mandatory to restructure the application code and to generate adequate communication mechanisms to the external memory, i.e., for remote accesses. In a previous report, we showed, in the context of the high-level synthesis (HLS) of hardware accelerators, how to derive an optimized organization for an accelerator communicating to an external DDR memory. Loop tiling is used to enable block communications, suitable for DDR memories. Pipelined communication processes are generated to overlap communications and computations, thereby hiding some latencies, in a way similar to double buffering. Finally, data reuse among tiles is exploited to avoid remote accesses when data are already available in the local memory.

The goal of this report is to show how to specify the sets of data to be read from and to be written to the external memory so as to reduce communications and reuse data as much as possible in the accelerator. The main difficulty in such a specification arises when some data are (re)defined in the accelerator, and even worse in the case of approximations, when some data *may* be redefined but this is not sure. We show that techniques based on parametric polyhedral optimizations can be used to generate the sets of data to be loaded (resp. stored) just before (resp. after) each tile. An interesting feature is that the case where no approximation is needed appears nicely as a particular case of the general procedure.

Key-words: Polyhedral optimizations, communication optimizations, pipelined processes, DDR memory, hardware accelerators, HLS.

* Compsys, LIP, UMR 5668 CNRS, INRIA, ENS-Lyon, UCB-Lyon

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Déportation de noyau de calcul avec optimisation des accès aux données distantes

Résumé :

Les accélérateurs matériels, comme par exemple via l'utilisation de GPGPUs ou de circuits dédiés sur FPGAs, sont particulièrement intéressants pour accélérer les applications gourmandes en calculs et en accès aux données. En revanche, pour obtenir de bonnes performances quand on déporte un noyau de calcul sur une telle plate-forme d'accélération, il est indispensable de restructurer le code de l'application et de générer des mécanismes de communication adéquats vers la mémoire externe, c'est-à-dire pour les accès distants. Dans un rapport précédent, nous avons montré, dans le contexte de la synthèse de haut niveau (HLS) d'accélérateurs matériels, comment générer une organisation optimisée pour un accélérateur communiquant avec une mémoire externe DDR. Le « tiling » (tuilage) est utilisé pour permettre des transferts par blocs, adaptés aux mémoires DDR. Le pipeline de processus de communication permet le recouvrement des calculs et communications, et donc de masquer certaines latences, selon un principe similaire à la technique du « double buffering » . Finalement, la réutilisation des données entre différentes tuiles est exploitée pour éviter des accès distants quand les données sont déjà disponibles dans la mémoire locale.

Le but de ce rapport est de montrer comment spécifier les ensembles de données a lire depuis la mémoire externe et à écrire dans cette mémoire de façon à réduire les communications et à réutiliser les données, autant que faire se peut, dans l'accélérateur. La difficulté principale dans une telle spécification apparaît lorsque certaines données sont (re)définies dans l'accélérateur, et pire dans le cas d'approximations, lorsqu'elles le sont peut-être mais que ce n'est pas certain. Nous montrons que des techniques d'optimisations polyédrales paramétrées peuvent être utilisées pour générer les ensembles de données à charger (resp. stocker) avant (resp. après) chaque tuile. Une propriété intéressante de notre méthode est que le cas où aucune approximation n'est nécessaire apparaît naturellement comme un cas particulier de la procédure générale.

Mots-clés : Optimisations polyédriques, optimisations des communications, processus pipelinés, mémoire DDR, accélérateurs matériels, synthèse de haut niveau.

Kernel Offloading with Optimized Remote Accesses

Christophe Alias Alain Darte Alexandru Plesco

July 25, 2011

Abstract – The use of hardware accelerators, e.g., with GPGPUs or customized circuits using FPGAs, are particularly interesting for accelerating data- and compute-intensive applications. However, to get high performance when offloading computation kernels to such an acceleration platform, it is mandatory to restructure the application code and to generate adequate communication mechanisms to the external memory, i.e., for remote accesses. In a previous report, we showed, in the context of the high-level synthesis (HLS) of hardware accelerators, how to derive an optimized organization for an accelerator communicating to an external DDR memory. Loop tiling is used to enable block communications, suitable for DDR memories. Pipelined communication processes are generated to overlap communications and computations, thereby hiding some latencies, in a way similar to double buffering. Finally, data reuse among tiles is exploited to avoid remote accesses when data are already available in the local memory.

The goal of this report is to show how to specify the sets of data to be read from and to be written to the external memory so as to reduce communications and reuse data as much as possible in the accelerator. The main difficulty in such a specification arises when some data are (re)defined in the accelerator, and even worse in the case of approximations, when some data *may* be redefined but this is not sure. We show that techniques based on parametric polyhedral optimizations can be used to generate the sets of data to be loaded (resp. stored) just before (resp. after) each tile. An interesting feature is that the case where no approximation is needed appears nicely as a particular case of the general procedure.

1 Introduction

The goal of this report is to show how to optimize the transfer of remote accesses for a computation kernel that has been offloaded to an acceleration platform. It is based on our previous work on source-level communication optimization for high-level synthesis (HLS), described in [2, 20, 3]. More precisely, it is an extension of the "communication coalescing" technique developed in [20, 3], which describe the whole transformation process, including local memory design and code generation. These two additional phases are not addressed here.

We focus on the optimization of hardware accelerators working on a large data set that cannot be completely stored in local memory, but need to be transferred from a DDR memory at the highest possible rate, and possibly stored temporarily locally. For such a memory, the throughput of memory transfers is asymmetric: successive accesses to the same DDR row are pipelined an order of magnitude faster than when the states of the finite-state machine controlling the

*Compsys, LIP, UMR 5668 CNRS, INRIA, ENS-Lyon, UCB-Lyon

DDR must be changed to access different rows. In other words, accessing data by blocks is a direct way of improving the performances: if not, the hardware accelerator, even highly-optimized, keeps stalling and runs at the frequency of the DDR accesses. The same situation occurs when accessing a bus for which burst communications are more efficient, when optimizing communications for GPGPUs or, more generally, when communications, between an external large memory and an accelerator with a limited memory, should be reduced (thanks to data reuse in the accelerator), pipelined, and preferably performed by blocks. This is why we believe that our techniques, although developed for HLS and specialized to Altera C2H [6], can be interesting in other contexts.

Our technique relies on loop tiling to increase the granularity of computations and communications. Each strip of tiles is optimized as follows. Transfers from and to the DDR are pipelined, in a blocking and double-buffering fashion, thanks to the introduction of software-pipelined communicating processes. Data reuse within a strip is exploited by accessing data from the accelerator and not from the DDR when already present. Local memories are automatically generated so as to store communicated data and exploit data reuse. To make this scheme possible, one needs to be able to describe, for each tile T and in a parametric fashion, the set Load(T) of data to be loaded before the execution of the tile T and the set Store(T) of data to be stored just after the execution of T. We give a complete specification of the problem, with the following contributions:

- We characterize in which case the sets Load(T) and Store(T) are valid. Our specification is general enough to handle the case where data used in a tile strip can also be redefined in the strip. It also handles the case where the sets In(T) (data read in the tile T but not written earlier in T) and Out(T) (data written in T) may be approximated.
- When the sets In(T) and Out(T), or their approximations, are described as a polyhedron or a union of polyhedra, linearly parameterized by T, we give an optimization procedure, based on parametric linear programming, that specifies the load and store sets Load(T) and Store(T) so that the live-range of each data in the accelerator is minimized. This should help reducing the size of the local memory.
- Our analysis is parameterized by an abstract "scheduling function" that expresses the tiling of loops and the pipelining of tiles. With this feature, there is no need to generate the tiled program before running the analysis, which is thus independent on the way the tiled code is finally generated. However, tiles are supposed to be executed in sequence, i.e., with no parallelism. The case of parallel programs makes the study more complicated and is left for future work.

We recall some prerequisites in Section 2 related to loop tiling and parametric polyhedral optimizations. Section 3 details the specification for the simplest case where, for each tile, the set of data read in T and the set of data written in T are known exactly. Section 4 addresses the less intuitive case where these sets can be over- and under-approximated. For each situation, we specify conditions for the sets Load(T) and Store(T) to be valid and we give a procedure to compute them so that live-ranges are as short as possible in the local memory, i.e., so that data are loaded from the external memory as late as possible and stored to the external memory as soon as possible.

2 Some prerequisites on polyhedral optimizations

Our method can be applied to offload a kernel on which *loop tiling* [21] and polyhedral transformations [4] can be applied, i.e., a set of `for` nested loops, manipulating arrays and scalar variables, whose iterations can be represented by an *iteration domain* using polyhedra. This is the case when loop bounds and `if` conditions are affine expressions of surrounding loops counters and structure parameters. This model can be extended through approximations, as explained in Section 4, for example when access functions are not fully analyzable or when the iteration domain is restricted with some complex `if` conditions.

2.1 Loop tiling and transformation function

Loop tiling is a standard loop transformation, which has proven to be effective for automatic parallelization and data locality improvement. With loop tiling, the iteration domain is partitioned into rectangular blocks (tiles) of iterations to be executed atomically. Loop tiling can be viewed as a composition of strip-mining and loop interchange. Strip-mining introduces two kinds of loops: the *tile loops*, which iterate over the tiles, and the *intra-tile loops*, which iterate in a tile. This step is always legal. Then, loop interchange pushes the intra-tile loops in the inner dimensions of the loop nest. In some cases, preliminary loop transformations, such as loop skewing, are necessary to make the loops tilable (i.e., fully permutable). In this case, "rectangular" has to be understood with respect to this preliminary change of basis.

We call *tile strip* the set of tiles described by the last tile loop, for a given iteration of the outer tile loops. This notion will be widely used in our approach, as most optimizations will be done within a tile strip, parameterized by the counters of the outer tile loops. The approach however is not limited to a one-dimensional tile strip, the same technique could be used to exploit inter-tile data reuse among two or more nested tile loops.

A loop tiling for a statement S, surrounded by n nested loops with iteration domain $\mathcal{D}_S$, can be expressed with an n-dimensional affine function $\vec{i} \mapsto \theta(S, \vec{i})$ and one (to make things simpler) size parameter b, where $\vec{i}$ is the *iteration vector* specifying the iterations of $\mathcal{D}_S$. A tile, defined by n loop counters $I_1, \ldots, I_n$, contains $\vec{i} \in \mathcal{D}_S$ if $bI_k \leq \theta(S, \vec{i}) < b(I_k + 1)$, for $k \in [1..n]$. Adding these constraints, for a fixed value b, to those expressing $\mathcal{D}_S$ gives an iteration domain $\mathcal{D}'_S$ of dimension $2n$. If the transformation θ corresponds to n permutable loops, then a valid sequential schedule of the tiled code is:

$$\theta_{\text{tiled}}(S, I_1 \ldots I_n, \vec{i}) = (I_1, \ldots, I_n, \theta(S, \vec{i}))$$

Main example

Consider the following code that computes in array c the product of two polynomials of degree N, stored in arrays a and b.

```
    for (i=0; i<= 2*N; i++)
S1:    c[i] = 0;

    for (i=0; i<=N; i++)
      for (j=0; j<=N; j++)
S2:     c[i+j] = c[i+j] + a[i]*b[j];
```

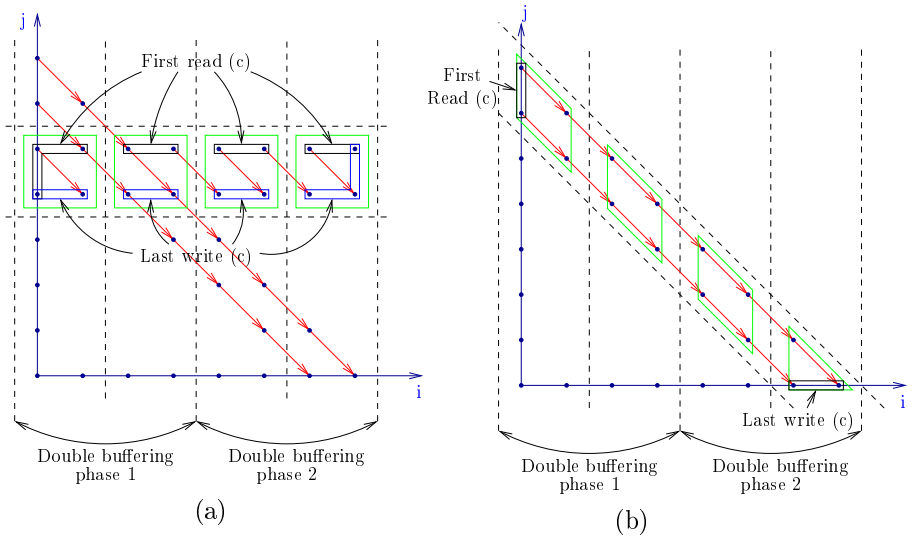

Figure 1: Different tilings induce different communications

We suppose that the offloaded kernel is the set of nested loops containing S_2. They are not permutable, some preliminary loop transformation is needed.

A possible tiling is given by the schedule $(i,j) \mapsto (N-j,i)$, which corresponds to a loop interchange and a loop reversal of the j loop, as depicted in Figure 1(a). For such a tiling, there is maximal inter-tile reuse of b within a tile strip (along the j axis), maximal intra-tile reuse of a within a tile (along the i axis), and some intra- and inter-tile reuse for c between two successive tiles. As an example, the figure shows in grey the set of elements of c that must be loaded by each tile and in blue the set of elements of c that must be stored back by each tile.

With the tiling of Figure 1(b), defined by the schedule $(i,j) \mapsto (i+j,i)$, the data dependences are kept in the tile strip. This way, the loads and stores for array `c` only arise on the first and last tiles of the tile strip. Notice that the loads and stores for the array `a` are the same in both cases. However, the number of transfers for array `b` now increases with this second tiling. The full sequential schedule of iterations, θ_{tiled}, is $(i,j) \mapsto (I,J,i+j,i)$ where $bI \leq i+j \leq bI+(b-1)$ and $bJ \leq i \leq bJ+(b-1)$, i.e., $I = \lfloor \frac{i+j}{b} \rfloor$ and $J = \lfloor \frac{i}{b} \rfloor$. □

Giving S, $\mathcal{D}'_S$, and θ_{tiled} to a polyhedral code generator will generate the tiled code [9, 5]. We point out however that we do not apply such a rewriting as a preliminary transformation as this would complicate our subsequent optimizations. Instead, all analysis and code generation steps are done with respect to the function θ. This function is also used to express the relative schedules of read, write, and computation processes, and to help us synthesize the adequate local buffers in a double-buffering fashion. Actually, "double-buffering" is a language simplification: we do not use two buffers, but one larger buffer. However, two successive blocks of computation in a tile strip are indeed pipelined with two blocks of communications, so as to overlap communications and computations.

The choice of the tiling is left to the user and is specified by means of a function θ. In this report, we do not explain how local buffers are organized, but

only how the sets Load(T) and Store(T) are generated for a given tile indexed by T. We first summarize the assumptions that characterize our scheme:

- Elements in Load(T) are loaded from the external memory before the tile T starts executing, but in any order for a given T.
- Elements in Store(T) are stored to the external memory after the tile T ends executing, but in any order of a given T.
- Tiles are executed in sequence, following a sequential order specified from θ as explained previously, i.e., with increasing T.
- Similarly, Load(T) (resp. Store(T)) is fully transferred before Load(T') (resp. Store(T')) if $T < T'$.

These constraints induce a dependence graph, as depicted in Figure 2, which can be software-pipelined to overlap loads, stores, and computations. Memory reuse however requires additional synchronizations to avoid the worst case where all loads (first row in Figure 2) are performed before all tiles are executed (second row). This is explained in more detail in [20, 3].

2.2 Parametric integer programming

We will make an extensive use of parametric integer programming as defined by Paul Feautrier [13], through the use of the software PIP he developed [16, 19].

A parametric polyhedron $P(\vec{z})$ is a set of the form $\{\vec{x} \mid A\vec{x} + B\vec{z} + \vec{c} \geq 0\}$ where $\vec{x}$ is a vector with n entries, the vector of all unknowns. The vector $\vec{z}$ is the vector built from parameters and has p entries. For a fixed $\vec{z}$, $P(\vec{z})$ is a polyhedron defined by l inequalities if A is a matrix of size $l \times n$, B a matrix of size $l \times p$, and $\vec{c}$ a constant vector of size l. The parameters can themselves be constrained by a set of affine inequalities $M\vec{z} + \vec{h} \geq 0$, called the *context*. The matrices and vectors A, B, M, $\vec{c}$, and $\vec{h}$ are assumed to be integer-valued.

2.2.1 Defining a lexicographic minimum as a QUAST

Depending on the chosen option, PIP finds the lexicographic minimum in $P(\vec{z})$ or the lexicographic minimum of the set of integer points in $P(\vec{z})$. In the original definition, $\vec{x}$ and $\vec{z}$ are supposed to be nonnegative in all entries, but this restriction can be removed. Also, finding the maximum, instead of the minimum, is possible. In all cases, the solution is described by a QUAST (Quasi Affine Selection Tree), which is a tree structure where each internal node describes

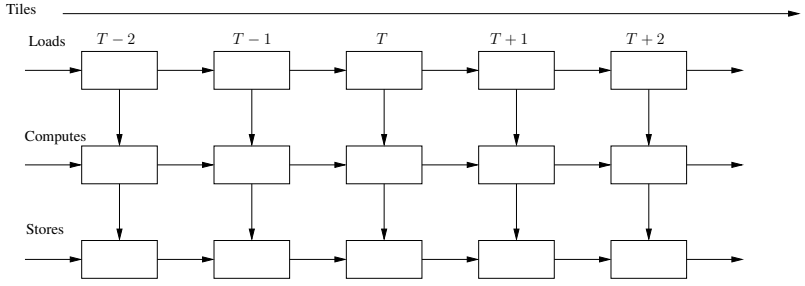

Figure 2: Unrolled dependence graph of load, store, and compute processes

an inequality on parameters and each leaf describes a solution, expressed as an affine function of the parameters, valid if all inequalities found along the path from the root to the leaf are satisfied. In case of the search in a set of integer points, additional parameters (called new parameter) may appear along the path to express integer divisions, and the solution remains affine with respect to all parameters. An additional feature is that the minimum (or maximum) of several QUASTs can also be represented as a QUAST, by merging and simplification rules (see details [14]). We will illustrate this feature later in Section 2.2.2.

Back to main example

Consider Figure 1(a). Let us find FirstOpRead(m), the operation indexed by (i, j) that is scheduled first with respect to the tiled schedule θ_{tiled}, within a given tile strip, and that accesses a given array cell of c. This search has three parameters: N, the initial loop bound, I, the outer tile index, and m, with $0 \leq m \leq 2N$, the memory cell of c. This amounts to find the lexicographic minimum of (J, ii, jj, i, j) with the following constraints, where b is a constant:

$$\text{Minimize } (J, ii, jj, i, j) \text{ with } \begin{cases} ii = N - j,\ jj = i, \\ bI \leq ii \leq b(I+1) - 1, \\ bJ \leq jj \leq b(J+1) - 1, \quad \text{when } 0 \leq m \leq 2N \\ 0 \leq i \leq N,\ 0 \leq j \leq N, \\ i + j = m \end{cases}$$

This system can be solved with the software tool PIP if b is fixed. For example, with the tile size $b = 10$, we add to the context the constraint $0 \leq m \leq 2N$ and PIP returns the following QUAST:

```
if (−10I + N ≥ 0) /* could be put in the context, too */
  if (−10I + N − m ≥ 0)
    if (10I − N + m + 9 ≥ 0) /* vertical band of elements, first tile, i.e., J = 0 */
      (J, ii, jj, i, j) = (0, N − m, 0, 0, m)
    else ⊥ /* means undefined */
  else
    if (−10I + 2N − m ≥ 0)
      if (−10I + N − m + 9 ≥ 0) /* horizontal band of elements, first tile */
        (J, ii, jj, i, j) = (0, 10I, 10I − N + m, 10I − N + m, N − 10I)
      else with k = ⌊(N+9m+9)/10⌋ /* generic horizontal case */
        (J, ii, jj, i, j) = (I + m − k, 10I, 10I − N + m, 10I − N + m, N − 10I)
    else ⊥ /* undefined */
  else ⊥ /* same, no solution */
```

The first inequality $10I \leq N$ expresses the fact that, if not satisfied, there is no valid iteration. This is found automatically by PIP. In practice, this is better to pre-compute this information by projecting the iteration domain (I, J, ii, jj) onto I and to add it to the context. This makes the QUAST shorter in general. We did not do it here just to illustrate the principles of PIP. Here, if $10I \leq N$ is added to the context, the outer disjunction is not required anymore.

Now, another simplification is possible. Indeed, we are only interested in the variables i and j, while the variables J, ii, and jj were introduced just to specify the schedule (the lexicographic order). If needed, they can be rebuilt from the variables i and j, but there is no need to express them in the solution. In

particular, in this example, the new parameter k is introduced only to specify J and to express the floor function, which is not affine. If we remove these useless variables, the QUAST can first be simplified into:

if $(-10I + N - m \geq 0)$
  if $(10I - N + m + 9 \geq 0)$ /* vertical band of elements, first tile, i.e., $J = 0$ */
    $(i, j) = (0, m)$
  else $\perp$
else
  if $(-10I + 2N - m \geq 0)$
    if $(-10I + N - m + 9 \geq 0)$ /* horizontal band of elements, first tile */
      $(i, j) = (10I - N + m, N - 10I)$
    else with $k = \lfloor \frac{N+9m+9}{10} \rfloor$ /* generic horizontal case */
      $(i, j) = (10I - N + m, N - 10I)$
  else $\perp$ /* means undefined */

Then, this QUAST can be further simplified by applying the following rule

$$\text{if } p \text{ then } x \text{ else } x \equiv x \tag{1}$$

as proposed in [14]. This leads to the following QUAST for FirstOpRead(m).

if $(-10I + N - m \geq 0)$
  if $(10I - N + m + 9 \geq 0)$
    $(i, j) = (0, m)$ /* vertical portion of c */
  else $\perp$
else
  if $(-10I + 2N - m \geq 0)$
    $(i, j) = (10I - N + m, N - 10I)$ /* horizontal portion of c */
  else $\perp$ /* means undefined */

Thanks to this variable elimination and simplification, the final solution has no additional parameter (such as k). This may not be the case in general. □

2.2.2 Simplification and inversion of QUASTs

As mentioned earlier, the minimum of two QUASTs is a QUAST. It can be obtained by applying the following simple combination rules [14]:

$$\min(Q, \text{if } p \text{ then } Q_1 \text{ else } Q_2) \equiv \text{if } p \text{ then } \min(Q, Q_1) \text{ else } \min(Q, Q_2) \tag{2}$$

where Q, Q_1, and Q_2 are QUASTs. The symmetric rule is also used to combine QUASTs. Particular cases can also be exploited as for example:

$$\begin{aligned}\min(&\text{if } p \text{ then } Q_1 \text{ else } Q_2, \text{if } p \text{ then } Q_3 \text{ else } Q_4)\\ &\equiv \text{if } p \text{ then } \min(Q_1, Q_3) \text{ else } \min(Q_2, Q_4)\end{aligned}$$

The previous rules are used to combine internal nodes of the QUASTs. To compare leaves, the following rules are used:

$$\min(\perp, Q) = \min(Q, \perp) = Q \text{ and } \min(\vec{i}, \vec{j}) \equiv \text{if } (\vec{i} \ll \vec{j}) \text{ then } \vec{i} \text{ else } \vec{j} \tag{3}$$

if $\vec{i}$ and $\vec{j}$ are two vector solutions and $\ll$ is the lexicographic order. This lexicographic order is itself expressed as a tree of affine conditions since $\vec{i}$ and $\vec{j}$

are expressed as affine functions of parameters. When these conditions can be statically evaluated within the given context, the right solution leaf is directly plugged. Similarly, dead solutions, i.e., those reached by a path defining an empty polyhedron can be identified and replaced by the symbol $\perp$. More sophisticated simplifications are possible to reduce the redundancy in the parameter conditions (see [17] for more advanced techniques). Combined with the rule of Equation 1, these mechanisms help simplifying the QUAST.

Once built, a QUAST can be interpreted and used in different ways, by changing the role of unknowns and parameters. By construction, the inequalities involved in a QUAST describe the set of parameters as a union of disjoint subsets. Each path to a leaf describes such a subset. If the path does not contain any new parameter, the corresponding subset is the integer points in a polyhedron, otherwise it is a linearly-bounded lattice (LBL), i.e., the projection of the integer points in a polyhedron. In the previous example, the final QUAST decomposes the set of all parameter values into three disjoint subsets: $\{(I,N,m) \mid 0 \leq m \leq 2N, 0 \leq 10I \leq N-m \leq 10I+9\}$, $\{(I,N,m) \mid 0 \leq m \leq 2N, 1 \leq m+10I-N \leq N, 0 \leq 10I \leq N\}$, and the rest for which there is no read to a cell m of array c. Now, considering m as an unknown, in the context $0 \leq 10I \leq N$, this decomposition also specifies the array cells m that are read, as a union of polyhedra parameterized by I and N:

$$\{m \mid \max(0, N-10I-9) \leq m \leq N-10I\} \cup \{m \mid N-10I+1 \leq m \leq 2N-10I\}$$

For each subset, the iteration $\mathrm{FirstOpRead}(m)$ responsible for the first read is specified by an affine function, as given previously.

This trick of changing the status of parameters into unknowns, or the converse, is one of the key of the algorithms we present later. For example, we can define, for each tile indexed by T, the set of memory cells m whose first read occurs in T. This can be expressed by putting the tiling inequalities back. For example, if tiles are defined along the axis that define operations:

$$\begin{aligned}
\mathrm{FirstReadInTile}(T) &= \{m \mid \mathrm{FirstTileRead}(m) = T\} \\
&= \{m \mid \left\lfloor \frac{\mathrm{FirstOpRead}(m)}{b} \right\rfloor = T\} \\
&= \{m \mid bT \leq \mathrm{FirstOpRead}(m) \leq bT+b-1\}
\end{aligned}$$

Back to main example

In Section 2.2.1, we found that $\mathrm{FirstOpRead}(m)$ was defined as:

$$\begin{aligned}
\mathrm{FirstOpRead}(m) = \{(i,j) \mid (i,j) = (0,m), \max(0, N-10I-9) \leq m \leq N-10I\} \\
\cup \{(i,j) \mid (i,j) = (10I-N+m, N-10I), N-10I+1 \leq m \leq 2N-10I\}
\end{aligned}$$

where i and j are the original loop indices. Given the schedule $\theta(i,j) = (N-j, i)$, we now incorporate $J = T = \lfloor \frac{i}{b} \rfloor$ as a parameter, we consider m as an unknown, and we get the following expression for $\mathrm{FirstReadInTile}(T)$:

$$\begin{aligned}
&\{m \mid (i,j) = (0,m), \max(0, N-10I-9) \leq m \leq N-10I, 10T \leq i \leq 10T+9\} \cup \\
&\{m \mid (i,j) = (10I-N+m, N-10I), N+1 \leq m+10I \leq 2N, 10T \leq i \leq 10T+9\}
\end{aligned}$$

and, after simplifications:

$$\begin{aligned}
\mathrm{FirstReadInTile}(T) = \{m \mid \max(0, N-10I-9) \leq m \leq N-10I, T=0\} \\
\cup \{m \mid \max(1, 10T) \leq m+10I-N \leq \min(N, 10T+9)\}
\end{aligned}$$

This expression gives in a parametric fashion, for each tile T, the set of data m that are accessed as a read for the first time in the tile strip indexed by I. □

We will use such an inversion mechanism to go from an expression such as FirstTileRead(m), which maps memory cells m to tiles T, to an expression such as FirstReadInTile(T), which maps tiles T to memory cells m. Actually, the same mechanism can be used to compute the inverse of any QUAST. Indeed, consider a QUAST $f(\vec{z})$ where the vector of parameters $\vec{z}$ is decomposed in two parts: $\vec{z} = (\vec{u}, \vec{v})$. The inverse of f with respect to $\vec{v}$, for a fixed value of $\vec{u}$, is the set $g(\vec{w}) = \{\vec{v} \mid f(\vec{z}) = \vec{w}\}$ where $(\vec{u}, \vec{w})$ is the vector of parameters. Remember that a QUAST also defines $f(\vec{z})$ through a union of disjoint LBLs:

$$f(\vec{z}) = \cup_{1 \leq i \leq n} \{f_i(\vec{z}, \vec{k}_i) \mid \exists \vec{k}_i \text{ s.t. } P_i(\vec{z}, \vec{k}_i)\}$$

where $\vec{k}_i$ denotes the new parameters introduced in the path leading to the ith leaf, P_i a set of inequalities on this path (i.e., it defines a polyhedron), and f_i is affine. By construction, $\vec{k}_i$ is uniquely defined from $\vec{z}$ (if not, $f(\vec{z})$ is a set and not a singleton) but this feature is not needed for the inversion. We get:

$$g(\vec{w}) = \cup_{1 \leq i \leq n} \{\vec{v} \mid \exists \vec{k}_i \text{ s.t. } P_i(\vec{z}, \vec{k}_i) \text{ and } f_i(\vec{z}, \vec{k}_i) = \vec{w}\}$$

With $\vec{x} = (\vec{u}, \vec{w})$ the new vector of parameters and $Q_i(\vec{x}, \vec{v}, \vec{k}_i)$ the system of inequalities defined by $P_i(\vec{u}, \vec{v}, \vec{k}_i) \cup \{f_i(\vec{u}, \vec{v}, \vec{k}_i) = \vec{w}\}$, which are affine because f_i is affine, $g(\vec{w})$ is defined as a union of LBLs, parameterized by $\vec{x} = (\vec{u}, \vec{w})$:

$$g(\vec{w}) = \cup_{1 \leq i \leq n} \{\vec{v} \mid \exists \vec{k}_i \text{ s.t. } Q_i(\vec{x}, \vec{v}, \vec{k}_i)\}$$

The LBLs composing this union are disjoint because $\{\vec{v} \mid \exists \vec{k}_i \text{ s.t. } Q_i(\vec{x}, \vec{v}, \vec{k}_i)\}$ is a subset of the LBL $\{\vec{v} \mid \exists \vec{k}_i \text{ s.t. } P_i(\vec{u}, \vec{v}, \vec{k}_i)\}$ and these LBLs, by construction of the QUAST, formed a disjoint union in the set of all vectors $\vec{z} = (\vec{u}, \vec{v})$.

To our knowledge, this inversion principle, which derives directly from the definition of a parametric polyhedron (where unknown and parameters are semantically, but not structurally, distinguished) and from the structure of QUASTs, has not been exploited so far. We will use it in Sections 3.3 and 4.

3 Communication coalescing: exact case

This section presents a method to select the array regions to be loaded from and stored to the external DDR memory for each tile. This step impacts two important criteria: a) the amount of communications with the DDR and b) the size of the local memory. At first glance, it may seem that these criteria are antagonistic. Actually, with our scheme, both can be reduced at the same time.

To perform data transfers, several solutions are possible. The most naive one is to access the DDR each time a data access is performed in the code. This solution does not require any local memory but is very inefficient: the latency to the DDR has to be paid for each access, which takes roughly 400 ns on our platform. Accesses must therefore be pipelined (a feature available in Altera HLS tool C2H) so that the accelerator throughput now depends not on the DDR latency, but on its throughput. If successively accessed data are not in the same DDR row, the accelerator is then able to receive 32 bits every 80 ns.

However, if data accesses are reorganized by blocks on the same row, thanks to loop tiling, the accelerator can work at full rate, i.e., it can receive 32 bits every 10 ns. But to sustain this rate and not pay any DDR latency, communications must be fully pipelined. This can be done thanks to *communication coalescing*, which amounts to host transfers out of a tile and regroup the same accesses to eliminate redundancy. Communication coalescing is a common optimization in compilers of parallel languages [7, 8]. However, the form of communication coalescing we develop here is more general as it exploits not only intra-tile reuse but also inter-tile reuse, even if data dependences exist between tiles.

To exploit communication coalescing, several approaches are again possible, depending on when transfers are performed. The first one is to load, just before executing a tile, all the data read in the tile, then to store to the DDR all data written in the tile. This solution, although correct, would not exploit data reuse and, unless no data-flow dependence exists between successive tiles, would forbid to overlap computations and communications. The other extreme solution is to first load all data needed in a tile strip, then to execute all tiles in the strip, before finally storing to the DDR all data produced by the strip. This would exploit data reuse but would require a large local memory to store all needed data. Furthermore, the computations would have to wait for all data to arrive before starting. The strategy we formalize in this section consists in sending load and store requests to the DDR only when needed. For each tile, we load from the DDR the data read for the *first time* in the current tile strip and we store to the DDR the data written for the *last time* in the current tile strip. Meanwhile, the data is kept and used (read and written) in the local memory, exploiting data reuse. As a bonus, the method handles naturally the case where dependences exist between consecutive tiles of a tile strip. Indeed, as the data concerned by the inter-tile data dependences are kept in local memory, the sequential execution of tiles guarantees the correctness of the program.

3.1 General specification

For a tile T, let $\mathrm{In}(T)$ be the data read in T, but not defined earlier in the tile, i.e., used in T and live-in for T, and let $\mathrm{Out}(T)$ be the data written in T. We first assume $\mathrm{In}(T)$ and $\mathrm{Out}(T)$ to be exact. The case where $\mathrm{In}(T)$ is over-approximated does not bring any difficulty. However, the case where $\mathrm{Out}(T)$ is not known exactly is more complex and will be addressed in Section 4.

To simplify set equations, we use a compact notation such as $\mathrm{Load}(t \leq T)$ to express a generic union of sets, here $\cup_{t \leq T} \mathrm{Load}(t)$. The first letter is always the free variable and the second a fixed value. For example, $\mathrm{Out}(t > T)$ stands for $\cup_{t>T} \mathrm{Out}(t)$ but does not stand for $\cup_{t>T} \mathrm{Out}(T)$.

We now specify $\mathrm{Load}(T)$ (resp. $\mathrm{Store}(T)$), the data to be loaded from (resp. stored to) the DDR just before (resp. after) executing the tile T.

Definition 1 (Valid Load)**.** *The function $t \mapsto \mathrm{Load}(t)$ is valid if and only if (iff) the following conditions hold for any tile T.*

(i) $\mathrm{In}(T) \setminus \mathrm{Out}(t < T) \subseteq \mathrm{Load}(t \leq T)$.

(ii) $\mathrm{Out}(t < T) \cap \mathrm{Load}(T) = \emptyset$.

Condition (i) means that all the data needed by the tile T, i.e., those input to T but not produced by a previous tile, are loaded just before T *or earlier*.

Condition (ii) ensures that no data defined by the kernel, thus alive in the local memory, is overwritten by a load. This arises when a data is written in a previous tile before being read in the current tile. Without this condition, some data would be loaded from the DDR and would overwrite the existing value, which is incorrect. Figure 3 illustrates Definition 1. The horizontal axis represents the tile strip, the vertical axis represents the DDR memory. The sets In, Out, and Load are depicted in blue, red, and green respectively. In the tiles $T-1$ and T, some data already loaded in tile $T-2$ are loaded again: this is correct although it is redundant. However, in tile T, loading more would be incorrect as some data previously computed (in red) would be overwritten.

Definition 2 (Valid Store)**.** *The function* $t \mapsto \mathrm{Store}(t)$ *is valid iff the following conditions hold (*$T_{\max}$ *is the last tile of the strip):*

(i) $\mathrm{Out}(t \leq T_{\max}) = \mathrm{Store}(t \leq T_{\max})$.

(ii) $\mathrm{Store}(T) \cap \mathrm{Out}(t > T) = \emptyset$ *for any tile* T.

Definition 2 is illustrated in Figure 4. Conditions (i) and (ii) mean that we expect to store exactly the data locally modified. Condition (ii) means that a data is stored after its last write. This is actually stronger than what is really needed for a validity condition, as a value could be stored several times. But this assumption will simplify the proofs, without hurting the correctness of the whole construction. We could also define more complex schemes, allowing for example to load from the DDR a value modified in the tile strip. But we would need to be able to guarantee that this value was already stored in the DDR and not modified again before the load. This would also imply a combined definition of the Load and Store functions, which we want to avoid.

In Definition 1, Condition (i) was a simple inclusion, which characterizes all valid solutions, including those that load data possibly several times and even those that load useless data. The next definition specifies the *exact* solutions, i.e., those that avoid useless loads. Note that Condition (i) can be rewritten into an equivalent condition into $\cup_{t \leq T} \{\mathrm{In}(t) \setminus \mathrm{Out}(t' < t)\} \subseteq \mathrm{Load}(t \leq T)$.

Definition 3 (Exact Load)**.** *The function* $t \mapsto \mathrm{Load}(t)$ *is exact iff the following conditions hold (*$T_{\max}$ *is the last tile of the strip):*

(i) The function $t \mapsto \mathrm{Load}(t)$ *is valid.*

(ii) $\cup_{t \leq T_{\max}} \{\mathrm{In}(t) \setminus \mathrm{Out}(t' < t)\} = \mathrm{Load}(t \leq T_{\max})$.

(iii) $\mathrm{Load}(T) \cap \mathrm{Load}(T') = \emptyset$ *for any tile* $T \neq T'$.

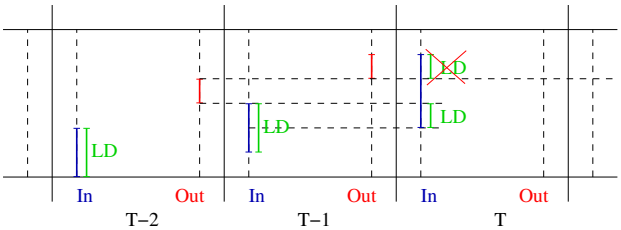

Figure 3: Valid loads for a 2D example.

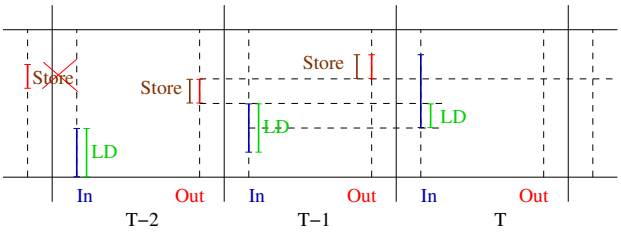

Figure 4: Valid stores for a 2D example.

The equality in Condition (ii) means that no useless data is loaded. Condition (iii) means that all Load(T) are disjoint and thus forbids redundant loads, i.e., data loaded several times, as illustrated in Figure 5. This may increase the size of the local memory as "holes" in the lifetime of an array element are not exploited (the corresponding memory location cannot be reused from the first access to the last). But, again, this assumption simplifies our general scheme. Finally, note that Definition 3 does not yet fully specify *when* a data is loaded.

Note that for an exact load, Condition (ii) of Definition 1 is redundant. Indeed, suppose that there exists x such that $x \in \text{Load}(T) \cap \text{Out}(t < T)$. As $x \in \text{Load}(T)$, Condition (ii) of Definition 3 implies that there exists T' such that $x \in \text{In}(T')$ and $x \notin \text{Out}(t' < T')$. Since $x \in \text{Out}(t < T)$, $T' < T$. Now, because of Condition (i) of Definition 1 applied to T', there exists $T'' \leq T'$ such that $x \in \text{Load}(T'')$. This contradicts Condition (iii) of Definition 3 with T'' and T.

Definition 4 (Exact Store)**.** Store(t) *is exact iff the following condition hold.*

(i) The function $t \mapsto \text{Store}(t)$ *is valid.*

(ii) $\text{Store}(T) \cap \text{Store}(T') = \emptyset$ *for any tiles* $T \neq T'$*.*

Unlike for loads where the equality in Condition (ii) of Definition 3 is an optimization choice, the equality in Condition (i) of Definition 2 is always required, not just for the exact case: it cannot be an over-approximation otherwise the execution of the tile strip would store an undefined value to the DDR, possibly leading to an incorrect code if one of the extra stores overwrites a meaningful value. As for Condition (ii) of Definition 4, it means that all Store(T) are disjoint, i.e., a value defined by the tile strip is stored only once. Again, as for Definition 3, this definition does not specify exactly when the stores occur.

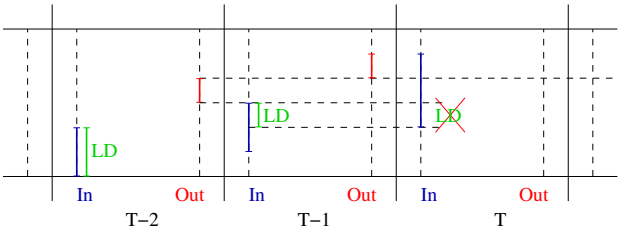

Figure 5: Valid loads refined with exact loads for a 2D example.

3.2 Exact formulation with reduced lifetimes

The previous definitions do not explicit the Load and Store operators for a given tile T. We now give a constructive solution where loads are performed as late as possible and stores as soon as possible. This has the effect of reducing the lifetime of data in the local memory, which tends to reduce its size. We start with a description involving sets and set operations. We then give an algorithm, based on parametric integer programming to build the Load and Store sets.

Theorem 1. *The functions* $T \mapsto \mathrm{Load}(T)$ *and* $T \mapsto \mathrm{Store}(T)$ *defined by*

- $\mathrm{Load}(T) = \mathrm{In}(T) \setminus \{\mathrm{In}(t < T) \cup \mathrm{Out}(t < T)\}$
- $\mathrm{Store}(T) = \mathrm{Out}(T) \setminus \mathrm{Out}(t > T)$

are exact load and store operators that reduce the lifetimes in the local memory.

Following Condition (i) of Definition 1 and Condition (iii) of Definition 3, a natural definition of $\mathrm{Load}(T)$ would be $\{\mathrm{In}(T) \setminus \mathrm{Out}(t < T)\} \setminus \mathrm{Load}(t < T)$. But this gives a recursive definition of Load. The definition above is more direct. Intuitively, $\mathrm{Load}(T)$ contains all the data read in the tile T, while excluding $\mathrm{In}(t < T)$, the data already read, and $\mathrm{Out}(t < T)$, the data already defined. As for $\mathrm{Store}(T)$, it is exactly the data written for the last time in T, i.e., $\mathrm{Out}(T)$, the data written in the current tile, excluding $\mathrm{Out}(t > T)$, those written later.

Proof. Let $x \in \mathrm{In}(T) \setminus \mathrm{Out}(t < T)$ and let t_0 be the smallest tile index such that $x \in \mathrm{In}(t_0)$. By definition, $t_0 \leq T$ and $x \notin \mathrm{In}(t < t_0)$. Also, $x \notin \mathrm{Out}(t < t_0)$ since $x \notin \mathrm{Out}(t < T)$. Thus x belongs to $\mathrm{Load}(t_0)$ and, finally, to $\mathrm{Load}(t \leq T)$. This proves Condition (i) of Definition 1, i.e., all needed data are loaded.

Actually, this also proves that $\cup_{t \leq T} \{\mathrm{In}(t) \setminus \mathrm{Out}(t' < t)\} \subseteq \mathrm{Load}(t \leq T)$. Conversely, by definition, $\mathrm{Load}(t \leq T) = \cup_{t \leq T} \{(\mathrm{In}(t) \setminus \mathrm{In}(t' < t)) \setminus \mathrm{Out}(t' < t)\}$ is a subset of $\cup_{t \leq T} \{\mathrm{In}(t) \setminus \mathrm{Out}(t' < t)\}$. Therefore, both sets are equal, i.e., $\mathrm{Load}(t \leq T) = \cup_{t \leq T} \{\mathrm{In}(t) \setminus \mathrm{Out}(t' < t)\}$. In particular, this is true for $T_{\max}$, which proves Condition (ii) of Definition 3, i.e., only the needed data are loaded.

Condition (iii) of Definition 3 holds from the fact that $\mathrm{Load}(T) \cap \mathrm{Load}(T')$, for $T' < T$, is a subset of $(\mathrm{In}(T) \setminus \mathrm{In}(t < T)) \cap \mathrm{Load}(T')$, thus a subset of $(\mathrm{In}(T) \setminus \mathrm{In}(T')) \cap \mathrm{Load}(T')$ and, finally, a subset of $(\mathrm{In}(T) \setminus \mathrm{Load}(T')) \cap \mathrm{Load}(T')$, which is empty. In other words, no data is loaded more than once.

As for the Store operator, it is clear that $\mathrm{Store}(T) \cap \mathrm{Store}(T') = \emptyset$ for $T \neq T'$ and $\mathrm{Store}(t \leq T_{\max}) = \mathrm{Out}(t \leq T_{\max})$. This proves Condition (ii) of Definition 4 (exact Store) and Condition (i) of Definition 2 (valid Store). Condition (ii) of Definition 2 is also satisfied as $\mathrm{Out}(t > T)$ is removed from $\mathrm{Store}(T)$.

It remains to prove that the lifetimes of data are as short as possible in the local memory, in other words that a data is loaded (resp. stored) as late (resp. soon) as possible, i.e., loaded just before its first read and stored right after its last write. This is true because $\mathrm{Load}(T) \subseteq \mathrm{In}(T)$, i.e., the data is required for a read in the tile T, thus it cannot be loaded later, and $\mathrm{Store}(T) \subseteq \mathrm{Out}(T)$, i.e., the data is defined in the tile T, thus it cannot be stored earlier. □

Theorem 1 gives an explicit formulation of the sets $\mathrm{Load}(T)$ and $\mathrm{Store}(T)$ if differences of parametric sets can be computed. We illustrate the previous formulas with our initial example, the product of two polynomials.

Back to main example

Consider the array c again and suppose the sets $\mathrm{In}(T)$ and $\mathrm{Out}(T)$ are available. Remember that $\mathrm{In}(T)$ is the set of array elements read in tile T but not previously written in the same tile. In this example, any array element accessed is first read before being written. Also, $\mathrm{In}(T) = \mathrm{Out}(T)$. Thus, with $T = J$:

$$\mathrm{In}(T) = \mathrm{Out}(T) = \{m \mid m = i + j,\ ii = N - j,\ jj = i,\ 0 \leq i \leq N,\ 0 \leq j \leq N,\\ bI \leq ii \leq b(I+1) - 1,\ bT \leq jj \leq b(T+1) - 1\}$$

In this particular case, the sets can be simplified, by substitution, into:

$$\mathrm{In}(T) = \mathrm{Out}(T) = \{m \mid m = jj + N - ii,\ 0 \leq ii \leq N,\ 0 \leq jj \leq N,\\ bI \leq ii \leq b(I+1) - 1,\ bT \leq jj \leq b(T+1) - 1\}$$

and, by projection, into:

$$\{m \mid bT \leq m \leq 2N - bI,\ N - b(I+1) + 1 + bT \leq m \leq N - bI + b(T+1) - 1\}$$

$\mathrm{In}(t < T)$ and $\mathrm{Out}(t < T)$ are obtained by adding the constraint $0 \leq t < T$:

$$\mathrm{In}(t < T) = \mathrm{Out}(t < T) = \{m \mid 0 \leq t \leq T - 1,\ bt \leq m \leq 2N - bI,\\ N - b(I+1) + 1 + bt \leq m \leq N - bI + b(t+1) - 1\}$$

In this example, this can be proved to be equal, for $T \geq 1$, to:

$$\mathrm{In}(t < T) = \{m \mid 0 \leq m \leq 2N - bI,\ N - b(I+1) + 1 \leq m \leq N - bI + bT - 1\}$$

Finally, according to Theorem 1, $\mathrm{Load}(T)$ is equal to $\mathrm{In}(T) \setminus \mathrm{In}(t < T)$, thus

$$\mathrm{Load}(0) = \{m \mid 0 \leq m \leq 2N - bI,\ N - b(I+1) + 1 \leq m \leq N - bI + b - 1\}$$

and, for $T \geq 1$:

$$\mathrm{Load}(T) = \{m \mid m \leq 2N - bI,\ N - bI + bT \leq m \leq N - bI + b(T+1) - 1\}$$

We retrieve the grey boxes (first reads of array c) in Figure 1(a). □

Note that, in general, for a program in the polytope model [15], the approach we used in the previous example, derived from Theorem 1, requires to be able to compute the difference of parametric LBLs. A tool such as the Omega Calculator [18] could be used, again when the block size is a constant. For the previous example, with $b = 10$, this tool outputs the following load operator (we removed the inequalities $0 \leq 10I \leq N$ and $0 \leq T$ that were generated):

$$\begin{aligned} \mathrm{Load}(T) \quad &= \quad \{m \mid 10T + N \leq 10I + m \leq 2N,\ 10I + m \leq 9 + 10T + N\} \\ &\cup \quad \{m \mid T = 0,\ 10I + m \leq N - 1,\ N \leq 9 + 10I + m,\ 0 \leq m\} \end{aligned}$$

which is exactly the expression found at the end of Section 2.2.2 for the first reads in the tile strip that occur in tile T.

It is sometimes not clear with the Omega Calculator when approximations are made in the output, in particular for operations such as set intersections or differences, and how to simplify the expressions. Also, we may not need in practice the full power of Presburger arithmetic. In the next section, we propose an alternative approach based on parametric integer programming, following the principles developed in Section 2.2.2. This technique always builds exact Load and Store operators, with no need to compute differences of sets. The simplification of the solutions relies on QUAST simplifications.

3.3 Using PIP to compute exact loads and stores

Theorem 1 provides a specification of optimized data transfers, based on set operations, which is more general than state-of-the-art communication vectorization and coalescing. Indeed, standard communication optimizations can eliminate redundant successive reads and redundant successive writes, but a data with a sequence of alternating reads and writes in the tile strip will generate only remote accesses. With our technique, the data is loaded and stored back only once: in the meantime, it is locally stored and possibly modified, thus reducing communications and increasing local reuse. We now show how we compute the Load and Store sets using parametric integer linear programming, in particular when the kernel fits in the polytope model, i.e., with affine loop bounds and affine access functions . How to map the data to local memory and how to generate the corresponding communication code is explained in [20, 3].

For a memory cell $\vec{m}$, we define:

- FirstTileAccess($\vec{m}$) the first tile that accesses $\vec{m}$, as a read or a write.
- FirstTileReadBeforeWrite($\vec{m}$) the first tile that accesses $\vec{m}$, if it is a read.
- LastTileWrite($\vec{m}$) the last tile that accesses m as a write.

Theorem 1 can then be reformulated as follows:

Theorem 2. *The operators of Theorem 1 can be defined as:*

$$\begin{aligned} \text{Load}(T) &= \{\vec{m} \mid \text{FirstTileReadBeforeWrite}(\vec{m}) = T\} \\ \text{Store}(T) &= \{\vec{m} \mid \text{LastTileWrite}(\vec{m}) = T\} \end{aligned}$$

In other words, Load(T) *contains the data accessed for the first time in T if this access is a read, and* Store(T) *contains the data written for the last time in T.*

Proof. Following Theorem 1, Load(T) contains all data in In(T), i.e., read in T but not written earlier in T, except those read or written in an earlier tile. Thus, Load(T) contains all data m whose first access is in T and this first access is a read, i.e., such that FirstTileReadBeforeWrite($\vec{m}$) $= T$. Similarly, Store(T) contains all data written in T, except those written later, thus all data $\vec{m}$ whose last write is in T, i.e., such that LastTileWrite($\vec{m}$) $= T$. □

If In(T) and Out(T) are available as (unions of) polyhedra or LBLs, the set FirstTileAccess(m) can be defined by a QUAST obtained by minimization:

$$\text{FirstTileAccess}(\vec{m}) = \min(\min\{T \mid \vec{m} \in \text{In}(T)\}, \min\{T \mid \vec{m} \in \text{Out}(T)\}) \quad (4)$$

When combining these two QUASTs as explained in Section 2.2.2, we can, in addition, tag each solution with the set it is coming from (in case of equality, we tag with the first set, i.e., the set of reads). Then, if we replace each solution coming from the second set by $\perp$, we get a QUAST that specifies, for each $\vec{m}$, the first tile that accesses it, keeping only those corresponding to a read, i.e., FirstTileReadBeforeWrite($\vec{m}$). Then, using the inversion mechanism of Section 2.2.2, we invert this mapping to get Load(T). As for Store(T), it is obtained by inverting the mapping LastTileWrite($\vec{m}$), obtained by maximization.

Actually, in practice, In(T) is computed from read and write accesses in the program. Also In(T) is not just the set of data read in T, but the set of data

read in T and not yet defined in T. However, pre-computing such a set $\mathrm{In}(T)$ is not necessary: FirstTileAccess($\vec{m}$) can be directly computed from the first read and write operations, instead of tiles. For that, for a memory cell $\vec{m}$, we define:

- FirstOpRead($\vec{m}$) the first operation in the tile strip that reads $\vec{m}$.
- FirstOpWrite($\vec{m}$) the first operation in the tile strip that writes $\vec{m}$.

FirstOpRead(m) is obtained by first extracting the set of operations reading $\vec{m}$. Then, we compute the read which is scheduled first (with respect to θ_{tiled}, the tiled schedule) in the tile strip, which boils down to compute the lexicographic minimum in a union of polytopes, as for exact array data-flow analysis [14]. Additional leaf modifications remove useless variables in the solution, as we illustrated in Section 2.2.1, which enables more QUAST simplifications.

More precisely, in the polytope model, all reads to an array c are as follows:

$$S : \vec{i} \in D : \ldots = \ldots c[u(\vec{i})] \ldots$$

where D is the iteration domain of the statement S, $\vec{i}$ is an iteration vector, and u is an *affine function*. The reads of $\vec{m} = c(\vec{i_0})$ in S is the set of operations $(S, \vec{i})$ that read $\vec{m}$, i.e., $u(\vec{i}) = \vec{i_0}$, and that are actually executed, i.e., $\vec{i} \in D$:

$$\mathrm{Read}(\vec{m}, S) = \{\vec{i} \in D \mid u(\vec{i}) = \vec{i_0}\}$$

Here, we assumed that c occurs only once in S, otherwise, one needs to distinguish each individual access. Now, S is given an affine schedule θ_S, specified by the user, as discussed in Section 2.1. We extend the definition of Read by incorporating the execution date of $\vec{i}$, $(\vec{I}, \vec{ii}) = (\lfloor \theta_S(\vec{i}) \rfloor, \theta_S(\vec{i}))$ to get:

$$\mathrm{Read}(\vec{m}, S) = \{(\vec{I}, \vec{ii}, \vec{i}) \mid \vec{ii} = \theta_S(\vec{i}) \wedge b\vec{I} \leq \vec{ii} < b(\vec{I} + \vec{1}) \wedge u(\vec{i}) = \vec{i_0} \wedge \vec{i} \in D\}$$

Then, we use PIP to get the lexicographic minimum of $\mathrm{Read}(\vec{m}, S)$. As in Section 2.2.1, we keep in the vector expressing the solution only the components corresponding to $\vec{i}$ and we simplify the QUAST. We proceed the same way for every assignment reading c, then we compute the global minimum by combining the QUASTs. In other words, if $S_1, \ldots, S_n$ denote the assignments reading c:

$$\mathrm{FirstOpRead}(\vec{m}) = \min_{\ll}(\mathrm{Read}(\vec{m}, S_1) \cup \ldots \cup \mathrm{Read}(\vec{m}, S_n))$$

which is computed by combining QUASTS as:

$$\mathrm{FirstOpRead}(\vec{m}) = \min_{\ll}(\min_{\ll} \mathrm{Read}(\vec{m}, S_1), \ldots, \min_{\ll} \mathrm{Read}(\vec{m}, S_n))$$

Similarly, we can obtain FirstOpWrite($\vec{m}$). As we did previously at the granularity of tiles, we can compute:

$$\mathrm{FirstOpAccess}(\vec{m}) = \min_{\ll}(\mathrm{FirstOpRead}(\vec{m}), \mathrm{FirstOpWrite}(\vec{m}))$$

If we replace all leaves that correspond to a write by $\perp$, we get an expression of FirstOpReadBeforeWrite($\vec{m}$), the first operation that accesses $\vec{m}$, if it is a read. Finally, as we did in Section 2.2.2, we can go easily from the expression of FirstOpReadBeforeWrite($\vec{m}$) to FirstTileReadBeforeWrite($\vec{m}$) as the tile indices $\vec{I}$ of a given operation $(S, \vec{i})$ is given by the relation $\vec{I} = \lfloor \theta_S(\vec{i}) \rfloor$, or equivalently $b\vec{I} \leq \lfloor \theta_S(\vec{i}) \rfloor < b(\vec{I} + \vec{1})$ (remember that T is the innermost tile index). It remains to inverse the resulting QUAST to get $\mathrm{Load}(T)$. Similarly, $\mathrm{Store}(T)$ is obtained by maximization, through the computation of LastOpWrite($\vec{m}$), then of LastTileWrite($\vec{m}$), and finally the inversion of LastTileWrite($\vec{m}$).

4 A conservative approximation formulation

If the sets $\mathrm{In}(t)$ and $\mathrm{Out}(t)$ are not precisely defined, we have two options.

(i) Restrict to a subset of programs for which an exact computation of $\mathrm{In}(T)$ and $\mathrm{Out}(T)$ is possible. This is the option we chose for the moment in our current implementation. We restrict to kernels in the polytope model.

(ii) Deal with approximation. In this case, we need to express the validity conditions relating the Load and Store operators to the approximated In and Out, and then to exhibit such operators.

We now discuss this second option. Note that approximations can also be used to restrict to sets that are simpler to manipulate and to simplify the final code generation, even if exact analysis is possible. For example, one may prefer to only manipulate simple polyhedra instead of complicated unions of LBLs.

4.1 General specification

In the following, we assume that the set $\mathrm{In}(T)$ is over-approximated by the set $\overline{\mathrm{In}}(T)$, i.e., $\mathrm{In}(T) \subseteq \overline{\mathrm{In}}(T)$, and that the set $\mathrm{Out}(T)$ is under- and over-approximated by the sets $\underline{\mathrm{Out}}(T)$ and $\overline{\mathrm{Out}}(T)$, i.e., $\underline{\mathrm{Out}}(T) \subseteq \mathrm{Out}(T) \subseteq \overline{\mathrm{Out}}(T)$. The set $\underline{\mathrm{Out}}(T)$ will help reducing communications and lifetimes. It can be empty in the worst case. Compared to the exact case of Section 3, there are now several complications, but only due to the over-approximation of $\mathrm{Out}(T)$. Indeed, if $\mathrm{Out}(T)$ is exact, over-approximating $\mathrm{In}(T)$ only causes additional useless loads but does not affect the execution of the kernel. The procedure of Section 3 can then be applied at no risk, with $\overline{\mathrm{In}}(T)$ instead of $\mathrm{In}(T)$. Also, over-approximating $\mathrm{Load}(T)$ itself is safe as long as Condition (ii) of Definition 1 is guaranteed. In other words, one should not load a value that has already been written in the tile strip, otherwise this load will overwrite a valid value.

Theorem 3 (Valid approximated Load)**.** *The function $t \mapsto \mathrm{Load}(t)$ is valid, with respect to the sets $\overline{\mathrm{In}}$, $\underline{\mathrm{Out}}$, and $\overline{\mathrm{Out}}$, iff:*

(i) $\overline{\mathrm{In}}(T) \setminus \underline{\mathrm{Out}}(t < T) \subseteq \mathrm{Load}(t \leq T)$ *for any tile T.*

(ii) $\overline{\mathrm{Out}}(t < T) \cap \mathrm{Load}(T) = \emptyset$ *for any tile T.*

Proof. It is sufficient to prove that these conditions imply the correctness conditions given by Definition 1. Let us first check that Condition (i) of Definition 1 holds. This is clear because $\mathrm{In}(T) \setminus \mathrm{Out}(t < T) \subseteq \overline{\mathrm{In}}(T) \setminus \underline{\mathrm{Out}}(t < T)$, thus in $\mathrm{Load}(t \leq T)$. Similarly, since $\mathrm{Out}(T) \subseteq \overline{\mathrm{Out}}(T)$, Condition (ii) of Definition 1 is verified. Conversely, if Condition (i) of Theorem 3 is not satisfied, some data element that may be needed in tile T is not loaded on time. Also, if Condition (ii) of Theorem 3 is not satisfied, some data element is loaded that may overwrite a valid value. In both cases, the resulting code may be incorrect. □

Theorem 3 generalizes Definition 1. Condition (i) means that all data read by T, except those previously written for sure, are loaded before T or earlier. The fact that $\overline{\mathrm{In}}(T)$ is an over-approximation can cause useless data to be read, but thanks to Condition (ii), those that may be previously defined cannot be over-written. They must be loaded earlier (see Figure 6). Redundant loads can be avoided if the condition $\mathrm{Load}(T) \cap \mathrm{Load}(T') = \emptyset$ is satisfied for all $T \neq T'$.

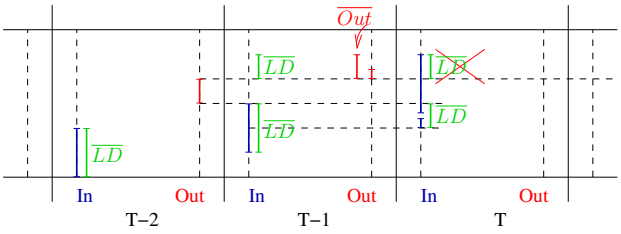

Figure 6: Valid approximated loads for a 2D example.

This solution is fully static. One could also use the same scheme as in Section 3, with $\underline{\text{Out}}(T)$ instead of $\text{Out}(T)$, and rely on a dynamic test to guarantee Condition (ii) of Definition 1. However, this would require an additional bit memory to mark, at run-time, the data that are indeed written. If a load is performed before the memory bit is set, the load would occur without altering the correctness. If the load is performed after, the bit will prevent the data to be overwritten. The problem is when the load is performed in parallel with a possible write. Unless an operation can guarantee an atomic access to both the bit and the memory cell, this overlap between communication and computation may lead to incorrect code. Furthermore, although such a dynamic test can allow to load as late as possible, therefore to reduce lifetime at run-time, this will not help reducing the size of the local memory if a static memory mapping is used, as we do with lattice-based memory allocation [12, 1]. Indeed, the memory location still needs to be reserved from the first time where it may be written. This is why we preferred a fully static scheme.

Now, consider the Store operator. Dealing with approximation seems more tricky at first glance. Indeed, if $\text{Store}(T)$ is over-approximated, extra data may be stored to the DDR, causing useful data in the DDR to be crushed. Conversely, if $\text{Store}(T)$ is under-approximated, we may forget to store useful outputs defined by the tile strip. Both approximations may cause an incorrect execution. Again, a dynamic solution is possible that would prevent, at run-time, to store a data not defined in the tile strip. But we can also design a fully-static solution as follows. An over-approximation of the Store operator can be correct if the extra data to be stored are exactly those already present in the DDR, as illustrated in Figure 7. This mechanism keeps the program semantics.

Theorem 4 (Valid approximated Store)**.** *The function $t \mapsto \text{Store}(t)$ is valid, with respect to the sets $\overline{\text{In}}$, $\underline{\text{Out}}$, and $\overline{\text{Out}}$, iff (with $T_{\max}$ last tile of the strip):*

(i) $\overline{\text{Out}}(t \leq T_{\max}) \subseteq \text{Store}(t \leq T_{\max})$.

(ii) $\text{Store}(T) \cap \overline{\text{Out}}(t > T) = \emptyset$ *for any tile* T.

(iii) $\text{Store}(T) \setminus \underline{\text{Out}}(t \leq T) \subseteq \text{Load}(t \leq T)$ *for any tile* T.

Proof. The first two conditions show that $\text{Store}(T)$ is an over-approximation that verifies Conditions (i) and (ii) of Definition 2 (valid Store), because of the approximation $\text{Out}(t) \subseteq \overline{\text{Out}}(t)$ for each tile t. Condition (iii) ensures the correctness of Store: any data stored from the local memory to the DDR and not known to be previously defined in the tile strip (T included) is loaded earlier from the DDR. This way, the DDR is always written by a correct value. □

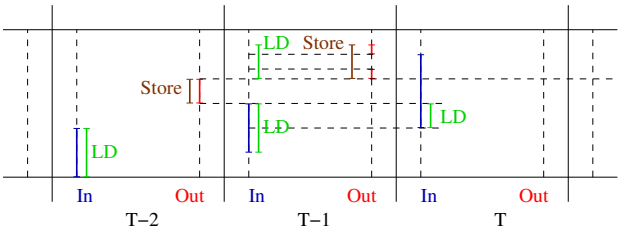

Figure 7: Valid approximated stores for a 2D example.

4.2 Approximated formulation with reduced lifetimes

We now exhibit Load and Store operators verifying these conditions. A naive solution is to load, before the first tile, all data potentially read in the tile strip, i.e., $\text{Load}(T_{\min}) = \overline{\text{In}}(t \leq T_{\max})$ and $\text{Load}(t > T_{\min}) = \emptyset$, and to store, after the last tile, all data potentially written in the tile strip, i.e., $\text{Store}(T_{\max}) = \overline{\text{Out}}(t \leq T_{\max})$ and $\text{Store}(t < T_{\max}) = \emptyset$. According to Theorems 3 and 4, this scheme is valid if $\overline{\text{Out}}(t \leq T_{\max}) \subseteq \text{Load}(T_{\min}) \cup \underline{\text{Out}}(t \leq T_{\max})$, which means that we should also pre-load all data that cannot be proved to be defined in the tile strip, i.e., $\text{Load}(T_{\min}) = \overline{\text{In}}(t \leq T_{\max}) \cup \big(\overline{\text{Out}}(t \leq T_{\max}) \setminus \underline{\text{Out}}(t \leq T_{\max})\big)$. This solution is obviously not suitable to reduce communications, overlap them with computations, and reduce the size of the local memory.

First note that a data in $\text{Store}(T)$ behaves exactly as a read that occurs just after tile T. Both require to be loaded in a similar way, as shown by Condition (i) of Theorem 3 and Condition (iii) of Theorem 4. In other words, we can define $\text{Store}(T)$ exactly as for the exact case, but with $\overline{\text{Out}}(T)$ instead of $\text{Out}(T)$. The Load operator will then take care of pre-loading all the necessary data. For that, $\text{Load}(T)$ will be defined, not from $\overline{\text{In}}(T)$, but from $\overline{\text{In}}(T) \cup (\text{Store}(T) \setminus \underline{\text{Out}}(T))$.

Theorem 5. *The function* $T \mapsto \text{Store}(T)$ *defined by*

$$\text{Store}(T) = \overline{\text{Out}}(T) \setminus \overline{\text{Out}}(t > T) \text{ for any tile } T$$

is an exact store operator with respect to the sets $\overline{\text{Out}}(T)$*, provided that the load operator is valid with respect to the sets* $\overline{\text{In}}'(T) = \overline{\text{In}}(T) \cup (\text{Store}(T) \setminus \underline{\text{Out}}(T))$.

Proof. Again, by construction, it is clear that $\text{Store}(T) \cap \overline{\text{Out}}(t > T) = \emptyset$, $\text{Store}(T) \cap \text{Store}(T') = \emptyset$ for $T \neq T'$, and $\text{Store}(t \leq T_{\max}) = \overline{\text{Out}}(t \leq T_{\max})$. This means that only the data that may be defined are stored, that there are no redundant stores, and that these stores are done as soon as possible, i.e., just after the last tile that possibly defines such data. As for Condition (iii) of Theorem 4, it is guaranteed if the Load operator is valid with respect to the sets $\overline{\text{In}}'(T) = \overline{\text{In}}(T) \cup (\text{Store}(T) \setminus \underline{\text{Out}}(T))$. Condition (i) of Theorem 3 ensures that $\overline{\text{In}}'(T) \setminus \underline{\text{Out}}(t < T) \subseteq \text{Load}(t \leq T)$ and thus, as $\overline{\text{In}}'(T)$ contains $\text{Store}(T) \setminus \underline{\text{Out}}(T)$, it ensures $\text{Store}(T) \setminus \underline{\text{Out}}(t \leq T) \subseteq \text{Load}(t \leq T)$. Note that we added $\text{Store}(T) \setminus \underline{\text{Out}}(T)$ to $\overline{\text{In}}'(T)$ instead of $\text{Store}(T) \setminus \underline{\text{Out}}(t \leq T)$ (which is smaller). This makes equations simpler without changing the result. □

Note that $\text{Store}(T)$ itself can even be over-approximated if needed. The Load operator will then have to take care of these possibly additional useless stores. The tricky part is now to find a formula for $\text{Load}(T)$ as, unlike for the exact case,

a data loaded at T is not always read in T. It may be pre-loaded at T because it is read later but may be overwritten in the meantime. Thus, defining $\mathrm{Load}(T)$ by $\overline{\mathrm{In}}'(T)\setminus(\overline{\mathrm{In}}'(t<T)\cup\overline{\mathrm{Out}}(t<T))$ or by $\overline{\mathrm{In}}'(T)\setminus(\overline{\mathrm{In}}'(t<T)\cup\underline{\mathrm{Out}}(t<T))$ does not work. We could define $\mathrm{Load}(T)$ by $(\overline{\mathrm{In}}'(T)\cup\overline{\mathrm{Out}}(T))\setminus(\overline{\mathrm{In}}'(t<T)\cup\overline{\mathrm{Out}}(t<T))$. This gives a valid solution. However, some useless data are loaded, in particular all data that are possibly written, then written for sure, then read. Those do not need a remote access. The right formula is given by the following theorem:

Theorem 6. *The function* $T\mapsto\mathrm{Load}(T)$ *defined, together with* $\mathrm{Store}(T)$*, by*

- $\overline{\mathrm{In}}'(T)=\overline{\mathrm{In}}(T)\cup(\mathrm{Store}(T)\setminus\underline{\mathrm{Out}}(T))$ *(all data that are "read").*
- $\overline{\mathrm{Ra}}(T)=\overline{\mathrm{In}}'(T)\setminus\underline{\mathrm{Out}}(t<T)$ *(all data that need a remote access).*
- $\mathrm{Load}(T)=\Big(\overline{\mathrm{In}}'(T)\cup(\overline{\mathrm{Out}}(T)\cap\overline{\mathrm{Ra}}(t>T))\Big)\setminus\Big(\overline{\mathrm{In}}'(t<T)\cup\overline{\mathrm{Out}}(t<T)\Big)$.

give valid operators that are "exact" with respect to the sets $\overline{\mathrm{In}}$*,* $\underline{\mathrm{Out}}$*, and* $\overline{\mathrm{Out}}$ *(no useless or redundant transfers) and that reduce the lifetimes in local memory.*

Proof. We first prove that the Load operator is, with respect to the set $\overline{\mathrm{In}}'(T)$, valid, exact (i.e., does not load useless data), and performs loads as late as possible. Condition (ii) of Theorem 3 is satisfied as $\overline{\mathrm{Out}}(t<T)$ is removed from $\mathrm{Load}(T)$. Now consider Condition (i). Let $x\in\overline{\mathrm{Ra}}(T)=\overline{\mathrm{In}}'(T)\setminus\underline{\mathrm{Out}}(t<T)$. Let t_0 be the smallest tile index such that $x\in\overline{\mathrm{In}}'(t_0)$ or $x\in\overline{\mathrm{Out}}(t_0)$. By definition, $x\notin\overline{\mathrm{In}}'(t<t_0)$ and $x\notin\overline{\mathrm{Out}}(t<t_0)$. Also, since $x\in\overline{\mathrm{In}}'(T)$, $t_0\leq T$ and $x\in\overline{\mathrm{Ra}}(t'>t_0)$. Thus $x\in\mathrm{Load}(t_0)$ and, finally, $x\in\mathrm{Load}(t\leq T)$. This proves Condition (i) of Definition 3, i.e., all needed data are loaded.

Actually, this also proves that $\overline{\mathrm{Ra}}(t\leq T)\subseteq\mathrm{Load}(t\leq T)$ and, in particular, $\overline{\mathrm{Ra}}(t\leq T_{\max})\subseteq\mathrm{Load}(t\leq T_{\max})$. Conversely, consider $x\in\mathrm{Load}(T)$. Either $x\in\overline{\mathrm{Ra}}(t>T)$, i.e., there exists t such that $x\in\overline{\mathrm{In}}'(t)\setminus\underline{\mathrm{Out}}(t'<t)$, or $x\in\overline{\mathrm{In}}(T)$ and, since $x\notin\overline{\mathrm{Out}}(t<T)$, we conclude that $x\in\overline{\mathrm{In}}(T)\setminus\underline{\mathrm{Out}}(t<T)$. This proves that $\mathrm{Load}(t\leq T_{\max})=\overline{\mathrm{Ra}}(t\leq T_{\max})$. In other words, all needed data are loaded and only those. The Load operator defined this way is exact with respect to the sets $\overline{\mathrm{In}}$, $\underline{\mathrm{Out}}$, and $\overline{\mathrm{Out}}$.

It remains to prove that there are no redundant loads. By definition $\mathrm{Load}(T)$ is a subset of $(\overline{\mathrm{Out}}(T)\cup\overline{\mathrm{In}}'(T))\setminus(\overline{\mathrm{In}}'(t<T)\cup\overline{\mathrm{Out}}(t<T))$, thus a subset of $(\overline{\mathrm{Out}}(T)\cup\overline{\mathrm{In}}'(T))\setminus(\overline{\mathrm{In}}'(T')\cup\overline{\mathrm{Out}}(T'))$ if $T'<T$. Since $\mathrm{Load}(T')$ is a subset of $\overline{\mathrm{In}}'(T')\cup\overline{\mathrm{Out}}(T')$, $\mathrm{Load}(T)\cap\mathrm{Load}(T')=\emptyset$. Finally, note that loads are done as late as possible. Indeed, if x is loaded at T, either x is possibly read in T ($x\in\overline{\mathrm{In}}'(T)$) or x is possibly written ($x\in\overline{\mathrm{Out}}(T)$. It cannot be loaded later. □

If the Store operator is exact, i.e., $\mathrm{Store}(T)=\overline{\mathrm{Out}}(T)\setminus\overline{\mathrm{Out}}(t>T)$, then, in the equation of Theorem 6 that defines $\mathrm{Load}(T)$, one can simply subtract $\overline{\mathrm{In}}(t<T)\cup\overline{\mathrm{Out}}(t<T)$ instead of $\overline{\mathrm{In}}'(t<T)\cup\overline{\mathrm{Out}}(t<T)$. This leads to the same result since $\mathrm{Store}(t<T)\subseteq\overline{\mathrm{Out}}(t<T)$. We preferred to give the general equation to also cover the case where the Store operator is over-approximated. This may generate extra loads that are taken into account when defining the Load operator. Then, $\mathrm{Load}(T)$ can be over-approximated too, but $\overline{\mathrm{Out}}(t<T)$ must be removed from this over-approximated $\mathrm{Load}(T)$ to satisfy Condition (ii) of Theorem 3. Thus, if desired, we can restrict to the case where the sets $\overline{\mathrm{In}}(T)$

(and even the sets $\overline{\mathrm{In}}'(T)$ and $\overline{\mathrm{Ra}}(T)$), the sets $\underline{\mathrm{Out}}(T)$, $\overline{\mathrm{Out}}(T)$, and $\mathrm{Store}(T)$ are polyhedra. However, $\mathrm{Load}(T)$ may be a difference of two polyhedra.

All these theoretical results give opportunities for handling cases where program analysis cannot be performed exactly or when approximating Load and Store sets allows a better packing of data to be transferred. Abstract interpretation may be useful to compute the sets $\overline{\mathrm{In}}$, $\underline{\mathrm{Out}}$, and $\overline{\mathrm{Out}}$, as in [11]. Theorem 6 gives a specification of the sets $\mathrm{Store}(T)$ and $\mathrm{Load}(T)$ through set operations. Even if the equations look complicated, a tool such as the Omega calculator can be used to compute them. Indeed, all equations involve only parametric unions such as $\overline{\mathrm{Out}}(t < T)$ which amounts to add equations such as $t < T$, and intersections and differences of sets. We now give, as we did in Section 3.3 for the exact case, a direct solution based on parametric integer programming.

4.3 Using PIP to compute approximated loads and stores

For a memory cell $\vec{m}$, we define:

- $\mathrm{LastTileWriteMaybe}(\vec{m})$ the last tile T that may write $\vec{m}$ ($\vec{m} \in \overline{\mathrm{Out}}(T)$).
- $\mathrm{FirstTileRead}(\vec{m})$ the first tile T that may read $\vec{m}$ ($\vec{m} \in \overline{\mathrm{In}}(T)$).
- $\mathrm{FirstTileWriteSure}(\vec{m})$ the first tile T that write $\vec{m}$ for sure ($\vec{m} \in \underline{\mathrm{Out}}(T)$).
- $\mathrm{FirstTileWriteMaybe}(\vec{m})$ the first tile T that may write $\vec{m}$ ($\vec{m} \in \overline{\mathrm{Out}}(T)$).
- $\mathrm{FirstTileReadBeforeWriteSure}(\vec{m})$ the first tile T that accesses $\vec{m}$, if $\vec{m}$ is "read" in the tile strip before being written for sure. Thus, $\vec{m} \in \overline{\mathrm{In}}(T)$ or $\vec{m} \in \overline{\mathrm{Out}}(T)$. As explained before, a store of $\vec{m}$ is considered as a "read".

A data $\vec{m}$ is "read" before being written for sure in two cases:

- there exists t such that $\vec{m} \in \overline{\mathrm{In}}(t) \setminus \underline{\mathrm{Out}}(t' < t)$.
- there exists t such that $\vec{m} \in \mathrm{Store}(t) \setminus \underline{\mathrm{Out}}(t' \leq t)$.

As a data $\vec{m}$ is stored after the last tile that may writes it, the second case occurs only if $\vec{m}$ is never written for sure, i.e., $\vec{m} \notin \underline{\mathrm{Out}}(t)$ for all t. Therefore, we can compute $\mathrm{FirstTileReadBeforeWriteSure}(\vec{m})$ almost as $\mathrm{FirstTileAccess}(\vec{m}) = \min(\mathrm{FirstTileRead}(\vec{m}), \mathrm{FirstTileWriteSure}(\vec{m}), \mathrm{FirstTileWriteMaybe}(\vec{m}))$. For that, we use two special symbols $\bot$, which, as before, means "undefined" and is larger than any other value, and $\top$, which is smaller than any other value.

1. Compute $Q_1(\vec{m}) = \min(\mathrm{FirstTileRead}(\vec{m}), \mathrm{FirstTileWriteSure}(\vec{m}))$. When combining the QUASTs, if $\mathrm{FirstTileWriteSure}(\vec{m}) < \mathrm{FirstTileRead}(\vec{m})$, we replace the corresponding leaf by the symbol $\top$.
2. Compute $Q_2(\vec{m}) = \min(Q_1(\vec{m}), \mathrm{FirstTileWriteMaybe}(\vec{m}))$.

Then, $\mathrm{FirstTileReadBeforeWriteSure}(\vec{m})$ is obtained by replacing each $\top$ by $\bot$:

- if $Q_1(\vec{m}) = \bot$, $\vec{m} \notin \overline{\mathrm{In}}(t)$ and $\vec{m} \notin \underline{\mathrm{Out}}(t)$, for all t. Then $Q_2(\vec{m}) \neq \bot$ if and only if $\vec{m} \in \overline{\mathrm{Out}}(t)$ for some t. This is the case where $\vec{m}$ is not read in the tile strip and is possibly written (thus stored), but never for sure.

- if $Q_1(\vec{m}) \notin \{\bot, \top\}$, $\vec{m}$ is accessed as a read and not written for sure before. It may be written earlier, but not for sure. In any case, $Q_2(\vec{m}) \notin \{\bot, \top\}$.
- if $Q_1(\vec{m}) = \top$, $\vec{m}$ is written for sure before any possible read or store. Then, by definition of $\top$, $Q_2(\vec{m}) = \top$, which is then replaced by $\bot$.

This proves that FirstTileReadBeforeWriteSure($\vec{m}$) is correctly computed. As for the exact case, we can now reformulate Theorem 6 as follows:

Theorem 7. *The operators of Theorem 6 can be defined as:*

$$\begin{aligned} \text{Load}(T) &= \{\vec{m} \mid \text{FirstTileReadBeforeWriteSure}(\vec{m}) = T\} \\ \text{Store}(T) &= \{\vec{m} \mid \text{LastTileWriteMaybe}(\vec{m}) = T\} \end{aligned}$$

Proof. Following Theorem 6, Store(T) contains all data possibly written in T, except those possibly written later, thus all data $\vec{m}$ whose last possible write is in T, i.e., such that LastTileWriteMaybe($\vec{m}$) $= T$.

As for the Load operator, it is defined from the sets $\overline{\text{In}}'(t)$. As shown in the proof of Theorem 6, a data x is loaded if and only if $x \in \overline{\text{In}}'(t) \setminus \underline{\text{Out}}(t' < T)$ for some tile index t. In other words, it corresponds to a "read" that is never written before for sure. Furthermore, it is loaded in Load(t_0) where t_0 is the smallest tile index such that $x \in \overline{\text{Out}}(t_0)$ or $x \in \overline{\text{In}}'(t_0)$, i.e., the first tile that accesses it, as defined by FirstTileReadBeforeWriteSure($\vec{m}$). □

As for the exact case, the same computation can be performed at the granularity of operation by computing:

- Compute $Q_1(\vec{m}) = \min_{\ll}(\text{FirstOpRead}(\vec{m}), \text{FirstOpWriteSure}(\vec{m}))$. When combining the QUASTs, if FirstOpWriteSure($\vec{m}$) $\ll_{\neq}$ FirstOpRead($\vec{m}$), replace the corresponding leaf by $\top$.
- Compute $Q_2(\vec{m}) = \min_{\ll}(Q_1(\vec{m}), \text{FirstOpWriteMaybe}(\vec{m}))$.
- Replace all $\top$ by $\bot$ to get FirstOpReadBeforeWriteSure($\vec{m}$).
- Define FirstTileReadBeforeWriteSure($\vec{m}$) by incorporating tile inequalities derived from $\vec{I} = \lfloor \theta(\vec{i}) \rfloor$, as explained in Section 3.3.
- Following Theorem 6, inverse the QUAST to get Load(T) as explained in Section 2.2.2.

Although it uses exact integer linear programming, this technique is a way to go beyond the polytope model, thanks to approximation. Indeed, as for fuzzy data-flow analysis [10], codes whose access functions cannot be fully analyzed or codes whose iteration domains are uncertain (for example, a data-dependent condition surrounding a statement) can still be captured.

So far, we only considered one type of parallelism, as expressed in Figure 2, resulting from the pipelining of successive tiles, executed sequentially. The next step will be to generalize the computations of the Load and Store operators for the more general situation of *parallel* kernels, communicating with an external DDR memory and, possibly, with data reuse between them.

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, USA, 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 International Conference on Application-specific Systems, Architectures, and Processors (ASAP'10)*. IEEE Computer Society, July 2010.

[3] Christophe Alias, Alain Darte, and Alexandru Plesco. Program analysis and source-level communication optimizations for high-level synthesis. Technical Report 7648, Inria, June 2011.

[4] 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, Arizona, June 2008.

[5] Pierre Boulet and Paul Feautrier. Scanning polyhedra without Do-loops. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 4–9, 1998.

[6] Altera C2H: Nios II C-to-hardware acceleration compiler. `http://www.altera.com/products/ip/processors/nios2/tools/c2h/ni2-c2h.html`.

[7] Daniel Chavarría-Miranda and John Mellor-Crummey. Effective communication coalescing for data-parallel applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, pages 14–25, Chicago, IL, USA, 2005. ACM.

[8] Wei-Yu Chen, Costin Iancu, and Katherine Yelick. Communication optimizations for fine-grained UPC applications. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 267–278. IEEE Computer, 2005.

[9] CLooG code generator in the polyhedral model. `http://www.cloog.org/`.

[10] Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy array dataflow analysis. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*, pages 92–101. ACM, 1995.

[11] Béatrice Creusillet. *Analyses de régions de tableaux et applications.* PhD thesis, École des mines de Paris, December 1996.

[12] Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, October 2005.

[13] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[14] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, February 1991.

[15] Paul Feautrier. *The Data Parallel Programming Model*, volume 1132 of *LNCS Tutorial*, chapter Automatic Parallelization in the Polytope Model, pages 79–103. Springer Verlag, 1996.

[16] Paul Feautrier. Solving systems of affine (in)equalities: PIP's user guide. Technical report, ENS-Lyon, 2001. Fourth version: `http://perso.ens-lyon.fr/paul.feautrier/pip.ps`.

[17] Paul Feautrier. Simplification of Boolean affine formulas. Technical Report RR-7689, Inria, July 2011.

[18] The Omega project: Frameworks and algorithms for the analysis and transformation of scientific programs. `https://github.com/davewathaverford/the-omega-project`.

[19] Parametric integer programming. `http://www.piplib.org/`.

[20] Alexandru Plesco. *Program Transformations and Memory Architecture Optimizations for High-Level Synthesis of Hardware Accelerators.* PhD thesis, Ecole normale supérieure de Lyon, France, September 2010. `http://tel.archives-ouvertes.fr/docs/00/54/43/49/PDF/alexandru.plesco_these.pdf`.

[21] Jingling Xue. *Loop Tiling for Parallelism.* Kluwer Academic Publishers, 2000.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399