Optimizing Remote Accesses for Offloaded Kernels: Application to High-Level Synthesis for FPGA

Christophe Alias Alain Darte Alexandru Plesco
Compsys, LIP, UMR 5668 CNRS, INRIA, ENS-Lyon, UCB-Lyon
firstname.lastname@ens-lyon.fr

ABSTRACT

Some data- and compute-intensive applications can be accelerated by offloading portions of codes to platforms such as GPGPUs or FPGAs. However, to get high performance for these kernels, it is mandatory to restructure the application, to generate adequate communication mechanisms for the transfer of remote data, and to make good usage of the memory bandwidth. In the context of the high-level synthesis (HLS), from a C program, of hardware accelerators on FPGA, we show how to automatically generate optimized remote accesses 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.

Our first contribution is to show how to generate the sets of data to be read from (resp. written to) the external memory just before (resp. after) each tile so as to reduce communications and reuse data as much as possible in the accelerator. The main difficulty arises when some data may be (re)defined in the accelerator. Our second contribution is an optimized code generation scheme, entirely at source-level, i.e., in C, that allows us to compile all the necessary glue (the communication processes) with the same HLS tool as for the computation kernel. Both contributions use advanced polyhedral techniques for program analysis and transformation. Experiments with Altera HLS tools demonstrate how to use our techniques to efficiently map C kernels to FPGA.

1. INTRODUCTION

HLS tools [9], e.g., Catapult-C, C2H, Gaut, Impulse-C, Pico-Express, Spark, Ugh, provide a convenient level of abstraction (in C-like languages) to implement complex designs. Most of these tools integrate state-of-the-art back-end compilation techniques and are thus able to derive an optimized internal structure, thanks to efficient techniques for scheduling, resource sharing, and finite-state machines generation. However, integrating the automatically-generated hardware accelerators within the complete design, with optimized communications, synchronizations, and local buffers, remains a hard task, reserved to expert designers. In addition to the VHDL glue that must sometimes be added, the input program must often be rewritten, in a proper way that is not obvious to guess. For HLS tools to be viable, these issues need to be addressed: a) the interface should be part of the specification and/or generated by the HLS tool; b) HLS-specific optimizing program restructuring should be available, either in the tool or accessible to the designer, so that high performances (mainly throughput) can be achieved. Such high-level transformations and optimizations are standard in high-performance compilers, not yet in high-level synthesis, even if their interest has been demonstrated through hand-made designs or restructuring methodologies [19, 11, 6, 2].

The goal of this paper is to show how the handmade restructuring demonstrated in [2] in the context of C2H, the Altera HLS tool, can be *fully automated*, thanks to advanced polyhedral code analysis and code generation techniques, entirely at source level (i.e., in C). We focus on the optimization of hardware accelerators that work 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 temporarily stored 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 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. A similar situation occurs when accessing a bus for which burst communications are more efficient, when optimizing remote accesses for GPGPUs or, more generally, when transfers, 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 our optimization techniques, although developed for HLS and specialized to Altera C2H, may 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,

IMPACT 2012
Second International Workshop on Polyhedral Compilation Techniques
Jan 23, 2012, Paris, France
In conjunction with HiPEAC 2012.

http://impact.gforge.inria.fr/impact2012

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 to store the communicated data and exploit data reuse. Our main contributions are the following:

Program analysis We show how to compute Load(T) and Store(T) of data to be loaded/stored before/after the execution of a tile T, thanks to parametric linear programming, so that the lifetime of each individual data in the local memory is minimized, which tends to reduce its size. Unlike previous approaches, ours can pipeline communications and exploit reuse among tiles even for data redefined in the tile strip. It can also be extended to the case where data accesses are approximated, i.e., when reads/writes are not known for sure.

Code generation Parameterized by a "scheduling function" that expresses the tiling of loops and the pipelining of tiles, our technique generates automatically the size of local buffers, the scanning of data sets to access the DDR row-wise, and the generation of communicating processes, thanks to the integration of several polyhedral techniques.

HLS integration A unique feature of our scheme is that the original computation kernel and all generated communicating processes are expressed in C and compiled into hardware with the same HLS tool (C2H), used as a back-end compiler.

In Section 2, we recall loop tiling and introduce some new features related to parametric polyhedral optimizations. Section 3 explains how to optimize remote accesses for an offloaded kernel, when the sets of data read and written in a tile are known exactly. [1] In Section 4, we apply our technique to the special case of HLS with Altera C2H. We present the different steps of the code generation and some experimental results comparing the performances of the hardware accelerators of [2], optimized by hand, and those optimized automatically thanks to our method.

2. PREREQUISITES

Our method can be applied to offload a kernel on which *loop tiling* [22] and polyhedral transformations 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 when access functions are not fully analyzable or when the iteration domain is restricted by some complex `if` conditions.

2.1 Loop tiling and transformation function

Loop tiling is a standard loop transformation, known 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 to innermost positions. In some cases, a first loop transformation, e.g., loop skewing, is needed to make the loops tilable (i.e., fully permutable). "Rectangular" has to be understood w.r.t. this preliminary change of basis.

We call *tile strip* the set of tiles described by the innermost tile loop, for a given iteration of the outer tile loops. This notion is widely used in our approach, as our optimizations are performed within such a one-dimensional tile strip, parameterized by the counters of the outer tile loops.

A loop tiling for a statement S, within n nested loops with iteration domain $\mathcal{D}_S$, can be defined thanks to a n-dimensional affine function $\vec{i} \mapsto \theta(S,\vec{i})$ (the permutable dimensions), where $\vec{i}$ is the *iteration vector* scanning $\mathcal{D}_S$, and a (single, to make things simpler) tile size b. 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. The next code computes, in c, the product of two polynomials of degree N, stored in arrays p and q.

```
    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] + p[i]*q[j];
```

From now on, we suppose that the offloaded kernel is the set of nested loops containing S_2. If commutativity and associativity are not exploited, some preliminary loop transformation is needed to make the loops permutable.

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 the left of Figure 1. For such a tiling, there is maximal inter-tile reuse of q within a tile strip (along the j axis), maximal intra-tile reuse of p within a tile (along the i axis), and some intra- and inter-tile reuse for c between two successive tiles. In grey are shown the elements of c that must be loaded by each tile and in blue those that must be stored back by each tile.

With the tiling in the right of Figure 1, defined by the schedule $(i,j) \mapsto (i+j,i)$, the data dependences on c are always 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 p are the same in both cases. However, the number of transfers for array q now increases compared to the first tiling. For 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 $J = \lfloor \frac{i}{b} \rfloor$. □

Given S, $\mathcal{D}'_S$, and θ_{tiled}, polyhedral code generation can be used to generate the tiled code. However, we do not apply such a rewriting as a preliminary step as this would complicate our subsequent optimizations. Instead, all analysis and code generation steps described hereafter are done with respect to the function θ. This function is also used to express the relative schedules of the pipelined load, store, 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. But two successive blocks of computation in a tile strip are indeed pipelined with two blocks of communications, so as to overlap commu-

[1] This restriction is enough for the kernels of Section 4 and, more generally, when reads are approximated. However, it needs to be extended when writes are approximated, as explained in [3]. We sketch this extension in Section 3.3.

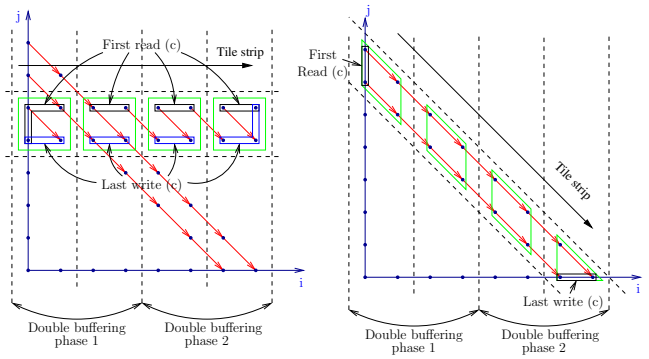

Figure 1: Different tilings and communications

nications and computations. The choice of the tiling is left to the user and is specified by means of a function θ, such as $(N-j, i)$, with a C pragma. Before explaining how the sets Load(T) and Store(T) are generated for a given tile indexed by T and how the local buffers are organized, we first summarize the assumptions that characterize our scheme:

- Elements in Load(T) are loaded from external memory before the tile T starts, but in any order for a given T.
- Elements in Store(T) are stored to external memory after the tile T ends, but in any order for a given T.
- Tiles are executed in sequence, following a sequential order specified from θ (the sequential order θ_{tiled} 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, 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 are performed before all tiles are executed. This is explained in more detail in Section 4, following the scheme proposed in [2], which is specific to the HLS tool C2H.

2.2 Parametric integer programming

We make an extensive use of parametric integer programming as defined by Feautrier [12], thanks to the software tool PIP (`http://www.piplib.org`). Following [15], a parametric polyhedron is a set $P(\vec{z}) = \{\vec{x} \mid A\vec{x} + B\vec{z} + \vec{c} \geq 0\}$ 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*. Without loss of generality, 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 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, illustrated in Section 2.2.2, is that the minimum (or maximum) of several QUASTs can also be represented as a QUAST, by merging and simplification rules.

Back to the main example. Consider Figure 1 with the left tiling. 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: the initial loop bound N, the outer tile index I, and the memory index m of array c. This amounts to finding the lexicographic minimum of (J, ii, jj, i, j) with the constraints:

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

This system can be solved with PIP if b is fixed. Here, for $b = 10$, with the context $10I \leq N$ and $0 \leq m \leq 2N$ that can be pre-computed, PIP returns the QUAST:

```
if (−10I + N − m ≥ 0)
  if (10I − N + m + 9 ≥ 0) /* vertical band, first tile */
    (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, 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 */
```

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 be simplified into:

```
if (−10I + N − m ≥ 0)
  if (10I − N + m + 9 ≥ 0)
    (i, j) = (0, m) /* vertical portion of c */
  else ⊥
else
  if (−10I + 2N − m ≥ 0)
    (i, j) = (10I − N + m, N − 10I) /* horizontal portion */
  else ⊥ /* means undefined */
```

For that, we also used the simplification rule proposed in [13]:

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

Here, thanks to the elimination of useless variables, the simplified expression of FirstOpRead(m) has no additional parameter (such as k). This may not be true in general. □

2.2.2 Simplification and inversion of QUASTs

The minimum of two QUASTs is a QUAST. It can be obtained by applying the following simple combination rule [13]:

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

where Q, Q_1, Q_2 are QUASTs, and its symmetric variant.

Particular cases can be exploited to combine QUASTs as:

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

The previous rules combined internal nodes of the QUASTs. To compare leaves, in addition to the simplification rule $\min(\bot, Q) = \min(Q, \bot) = Q$, the following rule is used:

$$\min(\vec{i}, \vec{j}) \equiv \text{if } (\vec{i} \ll \vec{j}) \text{ then } \vec{i} \text{ else } \vec{j}$$

where $\vec{i}$ 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 unsatisfiable constraints, can be replaced by the symbol $\bot$, then possibly simplified with the rule of Equation 1. More advanced mechanisms can reduce the redundancy in the parameter conditions [16].

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 complement for which there is no read to 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 $\text{FirstOpRead}(m)$ performing the first read is given by an affine function, as seen before.

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. Here, if tiles are defined along the axis that define operations, $\text{FirstReadInTile}(T)$ is equal to $\{m \mid \text{FirstTileRead}(m) = T\} = \{m \mid \left\lfloor \frac{\text{FirstOpRead}(m)}{b} \right\rfloor = T\}$, thus $\{m \mid bT \leq \text{FirstOpRead}(m) \leq bT + b - 1\}$.

Back to the main example. Consider $\text{FirstOpRead}(m)$ as given in Section 2.2.1, which specifies pairs of indices (i, j). Given the schedule $\theta(i, j) = (N - j, j)$, we now incorporate $J = T = \lfloor \frac{i}{b} \rfloor$ as a parameter, we consider m as an unknown, and, after simplifications, we get $\text{FirstReadInTile}(T)$ as:

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

which gives, for each tile T, the set of data m accessed as a read for the first time in the tile strip indexed by I. □

In Section 3.2, we will use such an inversion mechanism to go from an expression such as $\text{FirstTileRead}(m)$, which maps memory cells m to tiles T, to an expression such as $\text{FirstReadInTile}(T)$, which maps tiles T to memory cells m. The same mechanism can be used to compute the inverse of any QUAST, see details in [3]. 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 never been exploited so far.

3. COMMUNICATION COALESCING

We now show how to select the array regions to be loaded from and stored to the external DDR memory. This step impacts the amount of communications, the lifetime of array elements in the local memory, and the size of this memory.

To perform data transfers, the most naive solution is to access the DDR for each remote data access 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 thus be pipelined (a feature available in Altera C2H) so that the accelerator throughput depends not on the DDR latency, but on its throughput. The accelerator is then able to receive 32 bits every 80 ns, if successively accessed data are not in the same DDR row. 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 hoisting transfers out of a tile and to regrouping the same accesses to eliminate redundancy.

Communication coalescing is a common optimization in compilers of parallel languages and for exploiting scratch-pad memories [7, 8, 21, 20, 4]. The form of communication coalescing we develop here is different as it exploits not only intra-tile reuse but also inter-tile reuse, even if data dependences exist between tiles, *at the granularity of individual array elements*. Usually, the approach 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 does not exploit inter-tile data reuse and, unless no data-flow dependence exists between successive tiles, forbids to overlap computations and communications. This seems to be the approach implemented in the RStream compiler, as described in [21]. The other extreme solution is to first load all data needed in a tile strip, then to execute all tiles in the strip, and finally to store to the DDR all data produced by the tile strip, in other words, to hoist some communications outside the innermost tile loop. This exploits data reuse but requires a large local memory to store all needed data. Also, computations cannot start before all data have arrived.

The strategy we formalize here consists in sending load and store requests to the DDR only at the time they are needed. Furthermore, we load from (resp. store to) the DDR any data read (resp. written) in the current tile strip *only once*. Between the first and the last accesses, the data is kept and used (read and written) in the local memory, exploiting data reuse. As a bonus, this method handles naturally the case where dependences exist between tiles of a tile strip. Indeed, as data concerned by inter-tile dependences are kept in local memory, the sequential execution of tiles guarantees the program correctness. Another consequence of our scheme is that, unlike previous approaches for which

the lifetimes of array elements are all the same (either from the first tile to the last tile, or just within a tile), memory allocation based on bounding box as in [21, 20, 4] is not enough: to exploit the different lifetimes of individual array elements, we need to use a more general allocation scheme, based on modular mappings, as explained in Section 4.2.

3.1 Exact formulation with reduced lifetimes

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 discussed in Section 3.3.

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, the notation $\mathrm{Out}(t > T)$ 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. The next theorems specify *exact* solutions, those that avoid useless loads/stores.

THEOREM 1. *The function $t \mapsto Load(t)$ is exact iff the following conditions hold ($T_{\max}$ is the last tile of the strip):*
(i) $In(T) \setminus Out(t < T) \subseteq Load(t \leq T)$.
(ii) $\cup_{t \leq T_{\max}} \{In(t) \setminus Out(t' < t)\} = Load(t \leq T_{\max})$.
(iii) $Load(T) \cap Load(T') = \emptyset$ *for any tile* $T \neq T'$.

Condition (iii) forbids redundant loads, i.e., data loaded several times. With Condition (ii), no useless data is loaded. The three conditions imply $\mathrm{Out}(t < T) \cap \mathrm{Load}(T) = \emptyset$, which is needed to not overwrite a valid data, see details in [3].

THEOREM 2. *The function $t \mapsto Store(t)$ is exact iff:*
(i) $Out(t \leq T_{\max}) = Store(t \leq T_{\max})$.
(ii) $Store(T) \cap Out(t > T) = \emptyset$ *for any tile* T.
(iii) $Store(T) \cap Store(T') = \emptyset$ *for any tiles* $T \neq T'$.

Unlike for loads where the equality in Condition (ii) of Theorem 1 is an optimization choice, the equality in Condition (i) of Theorem 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 and possibly overwrite a meaningful value. Condition (ii) and (iii) mean that a value defined by the tile strip is stored only once and only after its last definition.

Theorems 1 and 2 do not specify when exactly the loads and stores occur. Several schemes remain possible. Also, the Load and Store operators are not given explicitly. The following theorem gives a constructive solution where loads are performed as late as possible and stores as soon as possible. This has the effect of minimizing the lifetime of data in the local memory, which tends to reduce its size.

THEOREM 3. *The functions Load and Store defined by*
- $Load(T) = In(T) \setminus \{In(t < T) \cup Out(t < T)\}$
- $Store(T) = Out(T) \setminus Out(t > T)$

are exact and reduce the lifetimes in the local memory.

Intuitively, $\mathrm{Load}(T)$ contains all the data read in the tile T, excluding the data already read $\mathrm{In}(t < T)$ and the data already defined $\mathrm{Out}(t < T)$. $\mathrm{Store}(T)$ contains the data written for the last time in T, i.e., $\mathrm{Out}(T)$, the data written T, excluding $\mathrm{Out}(t > T)$, those written later.

Back to the main example. Consider the array c again, for the left tiling of Fig. 1, i.e., with $\theta = (N - j, i)$. Remember that $\mathrm{In}(T)$ is the set of array elements read in tile T but not previously written in the same tile. Here, any array element accessed is read before being written. Also, $\mathrm{In}(T) = \mathrm{Out}(T)$. Thus, with parameters N, I, and $T = J$, we get:

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

The set $\mathrm{In}(t < T)$ is obtained by adding the inequalities $0 \leq t < T$. Applying Thm. 3 and simplifications, we retrieve the first reads of c, as depicted in grey, left of Fig. 1. □

In general, for a program in the polytope model [14], all these computations require to be able to compute the difference of parametric LBLs. Tools such as ISL (`http://freecode.com/projects/isl`) or Omega (`http://chunchen.info/omega/`) could be used, again when the block size is a constant. However, as Omega uses a more general framework (Presburger arithmetic), it is not clear whether it will always express the final sets as easy-to-scan unions of polyhedra (or LBLs). Instead, we designed an alternative approach based on parametric integer programming, following the principles of Section 2.2.2. This technique always builds exact Load and Store operators, with no need to compute differences of sets (they are actually replaced by minimizations). The outputs are simplified thanks to QUAST simplifications. Comparison with ISL is left to future work.

3.2 Using PIP to compute exact loads & stores

Theorem 3 specifies optimized transfers based on set operations. We now show how to compute the Load and Store sets with parametric integer linear programming, in particular when the kernel fits in the polytope model, i.e., with affine loop bounds and access functions. We define:
- $\mathrm{FirstTileAccess}(\vec{m})$, first tile that accesses an array cell indexed by $\vec{m}$, as a read or a write.
- $\mathrm{FirstTileReadBeforeWrite}(\vec{m})$, first tile that accesses an array cell indexed by $\vec{m}$, if it is a read.
- $\mathrm{LastTileWrite}(\vec{m})$, last tile that accesses $\vec{m}$ as a write.

Theorem 3 can then be reformulated as follows:

THEOREM 4. *The operators of Thm. 3 can be defined as:*

$$Load(T) = \{\vec{m} \mid FirstTileReadBeforeWrite(\vec{m}) = T\}$$
$$Store(T) = \{\vec{m} \mid LastTileWrite(\vec{m}) = T\}$$

$\mathrm{Load}(T)$ gives the data accessed for the first time in T if this is a read, $\mathrm{Store}(T)$ the data written for the last time in T. If $\mathrm{In}(T)$ and $\mathrm{Out}(T)$ are available as (unions of) polyhedra or LBLs, the set $\mathrm{FirstTileAccess}(m)$ can be defined by a QUAST obtained as the minimum of two QUASTs:

$$\min(\min\{T \mid \vec{m} \in \mathrm{In}(T)\}, \min\{T \mid \vec{m} \in \mathrm{Out}(T)\})$$

When combining the two QUASTs (see 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., $\mathrm{FirstTileReadBeforeWrite}(\vec{m})$. Using the inversion mechanism of Section 2.2.2, we invert this mapping to get $\mathrm{Load}(T)$. We get $\mathrm{Store}(T)$ by inverting the mapping $\mathrm{LastTileWrite}(\vec{m})$, obtained by maximization.

Actually, in practice, $\mathrm{In}(T)$ is computed from read and write accesses. Also, $\mathrm{In}(T)$ is not just the set of data read

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

- $\mathrm{FirstOpRead}(\vec{m})$, first op. in the strip that reads $\vec{m}$.
- $\mathrm{FirstOpWrite}(\vec{m})$, first op. in the strip that writes $\vec{m}$.

$\mathrm{FirstOpRead}(m)$ is obtained by first extracting the set of operations reading $\vec{m}$. Then, we compute the read that 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 dataflow analysis [13]. Additional leaf modifications remove useless variables in the solution, as illustrated in Section 2.2.1, which enables more QUAST simplifications. More precisely, in the polytope model, reads to c are as follows:

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

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

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

(If c occurs more than once in S, each access is distinguished.) Now, remember that S is given an affine schedule θ_S, as discussed in Section 2.1. We extend the definition of Read by incorporating the execution date of $\vec{i}$, i.e., $(\vec{I}, \vec{ii}) = (\lfloor \theta_S(\vec{i}) \rfloor, \theta_S(\vec{i}))$ to get $\mathrm{Read}(\vec{m}, S)$ as:

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

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. Thus, if the assignments reading c are $S_1, \ldots, S_n$, $\mathrm{FirstOpRead}(\vec{m})$ is computed as:

$$\min(\min \mathrm{Read}(\vec{m}, S_1), \ldots, \min \mathrm{Read}(\vec{m}, S_n))$$

We compute $\mathrm{FirstOpWrite}(\vec{m})$ similarly. Finally, as we previously explained at the granularity of tiles, we compute:

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

If we replace all leaves that correspond to a write by $\perp$, we get an expression of $\mathrm{FirstOpReadBeforeWrite}(\vec{m})$, 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 $\mathrm{FirstOpReadBeforeWrite}(\vec{m})$ to $\mathrm{FirstTileReadBeforeWrite}(\vec{m})$ 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 invert the resulting QUAST to get $\mathrm{Load}(T)$. Similarly, $\mathrm{Store}(T)$ is obtained by maximization, through the computation of $\mathrm{LastOpWrite}(\vec{m})$, then of $\mathrm{LastTileWrite}(\vec{m})$, and finally the inversion of $\mathrm{LastTileWrite}(\vec{m})$.

3.3 Extensions for approximated reads/writes

When exact analysis is not possible, 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 in our current implementation and presented here, restricting to the polytope model.

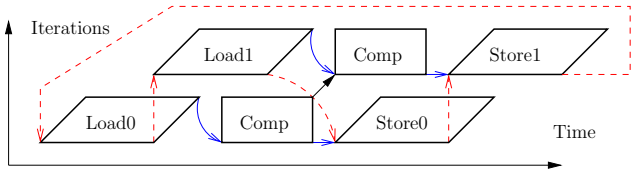

Figure 2: Software-pipelined synchronizations.

(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.

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, e.g., one may prefer to always manipulate simple polyhedra, if possible.

We now discuss briefly the second option. Some complications arise, but only due to an approximation of $\mathrm{Out}(T)$. Indeed, if $\mathrm{Out}(T)$ is exact, over-approximating $\mathrm{In}(T)$ into $\overline{\mathrm{In}}(T)$ only causes additional useless loads but does not affect the execution of the kernel. Thus, the procedure of Sect. 3.2 can 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 no value already written in the tile strip is loaded.

We point out that it is possible to extend all previous results to handle the case where the set $\mathrm{Out}(T)$ is also under- and over-approximated, i.e., $\underline{\mathrm{Out}}(T) \subseteq \mathrm{Out}(T) \subseteq \overline{\mathrm{Out}}(T)$. The trick, developed in [3], is a) to pre-load any data that *may* be written, but not for sure, so as to avoid storing a dummy data in the DDR, b) to pre-load any data that is needed before it may be written. This leads to new set equations and an extension of the procedure of Section 3.2.

4. APPLICATION TO HLS FOR FPGA

We now use the theory developed in Section 3 to generate automatically a C specification of communicating processes that can be compiled into hardware by a HLS tool, namely Altera C2H, following the procedure proposed in [2]. It remains to show how the different communication and computation processes are scheduled and synchronized, in C using C2H, how the local memories (size and access function) are then defined with respect to this schedule, and how the load and store sets are finally scanned.

We point out that all these steps – computation of loads/stores, computation of a mapping for designing local buffers, scanning of sets for kernel generation – are done with respect to the schedule θ. This makes the whole technique transparent, without even generating an initial loop tiling.

4.1 Synchronization of comp/comm processes

Following the methodology of [2], we generate 5 functions (called *drivers*) to be translated by C2H into separate hardware accelerators. For each tile strip, the function Compute executes all tiles in sequence, whereas the tiles are processed by pairs by two load and two store drivers, e.g., if $T_{\min} = 0$, Load0 and Store0 deal with even tiles, Load1 and Store1 with odd tiles. Each driver contains a loop nest iterating over the tiles. For each tile, a piece of code (called *micro-kernel*) performs the required loads, computations, or stores (see Section 4.2). The drivers are run in parallel and software-pipelined as shown in Fig. 2, with synchronizations implemented as blocking reads and writes in FIFOs of size 1.

In C2H, nested loops are scheduled with a hierarchical finite-state machine (FSM) structure. Data fetches in loops

are pipelined to hide latency. Furthermore, a special state is added, after a precomputed constant number of cycles, that stalls the FSM until the data is received. We exploit this mechanism to guarantee the data-flow dependences induced by the remote data transfers (blue arrows in Figure 2) by placing the corresponding synchronizations outside the micro-kernels. On the contrary, the synchronizations used to sequentialize the accesses to the DDR (dotted arrows in Figure 2) are placed inside the micro-kernels, at the last iteration, i.e., as soon as the last DDR request within a tile is initiated. This avoids the important penalty due to the loop pipeline that must be drained. This way, computations and communications are pipelined and latencies are hidden. The subtleties of this implementation and the interaction with the specificities of C2H are detailed in [2].

For the design of local memories, we need to specify the software-pipelined schedule of the processes to know when buffer locations can be reused. As this double-buffering scheme operates on tiles, two by two, this schedule cannot be specified directly as an affine function. Instead, we emulate a loop unrolling by 2. If T is the innermost tile counter (i.e., iterating on the tile strip), we add the constraint $T = 2p$ (resp. $T = 2p + 1$) to the tile domains, where p is a fresh integer variable. Then, as far as memory reuse is concerned, it is enough to specify the pipelined schedule with the following 2D schedule θ_{db} (this is when $T_{\min}$ is even):

$\theta_{\text{db}}(\text{Load0}, 2p) = (p, 0)$	$--$
$\theta_{\text{db}}(\text{Comp}, 2p) = (p, 1)$	$\theta_{\text{db}}(\text{Load1}, 2p + 1) = (p, 1)$
$\theta_{\text{db}}(\text{Store0}, 2p) = (p, 2)$	$\theta_{\text{db}}(\text{Comp}, 2p + 1) = (p, 2)$
$--$	$\theta_{\text{db}}(\text{Store1}, 2p + 1) = (p, 3)$

4.2 Local memory management

With our method, all computations are done with variables from the local memory. The lifetime of such a variable starts at its first access (possibly resulting from a load operation) and ends at its last access (possibly resulting from a store operation). We now explain how variables are mapped in the local memory. It must be done so that (i) two data live at the same time are not mapped to the same local address, (ii) the local memory size is as small as possible.

Unlike the methods developed in [18], which try to pack data optimally (in size), possibly with complex and expensive mapping functions and reorganization, we prefer to rely on array contraction based on modular mappings [17, 10]. In this framework, an array cell $a(\vec{i})$ is mapped to a local array cell $a\_tmp(\sigma(\vec{i}))$ where $\sigma(\vec{i}) = A\vec{i} \bmod \vec{b}$, A is an integer matrix, and $\vec{b}$ is an integral vector defining a modulo operation component-wise. In many cases, the array index functions are translations with respect to the loop indices as in `a[i][j-1]` and the program reads and writes consecutive array cells. The set of live array cells is then a window sliding during a tiled program execution, allowing efficient memory optimizations. The framework presented in [10] generalizes this situation, given an analysis of live array cells.

The principles of lattice-based memory allocation are the following. First, a conflict relation $\bowtie$ is defined: $a(\vec{i}) \bowtie a(\vec{j})$ if the lifetimes of the array cells $a(\vec{i})$ and $a(\vec{j})$ overlap. From the relation $\bowtie$, the *conflict polyhedron* $DS = \{\vec{i} - \vec{j} \mid a(\vec{i}) \bowtie a(\vec{j})\}$ is derived, which represents the sliding window mentioned above. Then, an *admissible lattice* for DS is built, i.e., an integer lattice L such that $DS \cap L = \{0\}$. A mapping σ is finally derived from L so that $\ker \sigma = L$. In our implementation, we use the tool Clak described in [1]. It takes as input a polyhedron (the set DS) and produces an admissible lattice for DS and a corresponding mapping σ.

Back to the main example. As shown earlier, for the left tiling of Fig. 1, for every (complete) tile T of size $b \times b$, parameterized by the outer index I, the region $c[N - b(I + 1) + 1 : N - b(I - 1) - 1]$ is loaded in c_tmp if $T = 0$ and $c[N - bI + bT : N - bI + b(T + 1) - 1]$ if $T \geq 1$. The data are consumed in a tile while the loads for the next tile are processed. Therefore, the set DS is equal to $[-3b+2 : 3b-2]$ resulting in the mapping $\sigma_c(i) = i \bmod (3b - 1)$. If the first tile was handled differently, i.e., with no overlap with the next tile, the resulting memory for c would be of size $2b$. As for array p, each tile loads a new block of size b while the previous block is consumed, resulting in a memory of size $2b$. Finally, for array q, a single block of size b is loaded at $T = 0$ and kept in memory for the whole tile strip. □

For code generation, a direct approach is to feed ClooG (`http://www.cloog.org`) with the different data sets, together with a sequential schedule. In our context however, this gives a correct but inefficient code. It is better to generate each kernel as a single "linearized" loop executing one instruction per iteration, using the Boulet-Feautrier algorithm [5]. This avoids the penalty due to the pipeline of inner loops that must be drained (see Sect. 4.1). Also, as recalled in Sect. 3, accessing successively in different DDR rows degrades the throughput. With a single loop, we achieve spatial locality in the DDR accesses by scanning the different arrays one after the other, with no interleaving, and following rows, i.e., lexicographically with respect to the array indices. Furthermore, such a loop is nicely pipelined with C2H, with one DDR access per iteration.

4.3 Experimental results

We implemented our methods using the polyhedral tools PIP and Polylib. Our prototype takes as input the C source code of a small kernel to be optimized. The input parameters, such as the loop tiling, are specified with pragmas. Then, a C source code, which implements a double-bufferized version of the kernel, is automatically generated. It can be simulated using linux processes, FIFOs, and shared memories (with IPC linux library). The 5 driver codes are then synthesized using C2H, which integrates them automatically in the system instantiated using Altera SOPC builder. Before, we currently still need to do a few modifications by hand, such as inserting the adequate pragmas for C2H, transforming array accesses to linearized addresses with the right base addresses, instantiating memories in the SOPC builder, changing some arrays into non-aliasing pointers so that C2H, whose dependence analyzer and software pipeliner are weak, can compile the code with the right initiation intervals, etc. All these changes are systematic, but not integrated yet in our code generator and take some time.

The study provided in [2], for the HLS tool C2H, showed that, even for elementary kernels, generating adequate C codes that can be automatically synthesized with no additional handmade VHDL glue, while exploiting the maximal DDR bandwidth, is very tricky. But it is feasible if codes and synchronizations are written in a specific, though generic, way. Our techniques show that this process can be automated. We considered the 3 kernels studied in [2], DMA transfer, sum of vectors (VS), matrix multiply (MM), to check if we could achieve the same performance auto-

Kernel	ALUT	Reg.	T. reg.	IP	M. freq.	S-U
System alone	4406	3474	3606	8	205.85	
DMA original	4598	3612	3744	8	200.52	1
DMA manual	9853	10517	10649	8	162.55	6.01
DMA autom.	11052	12133	12265	48	167.87	5.99
VS original	5333	4607	4739	8	189.04	1
VS manual	10881	11361	11493	8	164	6.54
VS autom.	11632	13127	13259	48	159.8	6.51
MM original	6452	4557	4709	40	191.09	1
MM manual	15255	15630	15762	188	162.02	7.37
MM autom.	24669	32232	32364	336	146.25	7.32

Table 1: Synthesis: original, manual, automatic

matically. Matrix multiply, also the example demonstrated in [21], is the most involved kernel: the original code has a few lines but the hand-optimized version (a double-buffered matrix multiply by block) has more than 500 lines!

We used ModelSim to evaluate our designs, which were synthesized on the Altera Stratix II `EP2S180F1508C3` FPGA, running at 100 MHz, and connected to an outside DDR memory, of specification JEDEC DDR-400 128 Mb x8, CAS of 3.0, running at 200 MHz. The optimized versions can run 6x or more faster than the direct implementations (remember that the maximal speed-up is at most 8, if we start from a code where successive DDR accesses are in different rows). Note that these speed-ups are obtained not because computations are parallelized (tiles are run in sequential), not only because the communications are pipelined (this is also the case in the original versions), but because DDR requests are reorganized to get successive accesses on the same row as much as possible, because some communications overlap computations, and because some data reuse is exploited.

However, to achieve this, there is a (moderate) price to pay in terms of hardware resources, in addition to the local memories involved to store the data locally. This is illustrated in Table 1, which gives different parameters measuring the hardware usage: the number of look-up tables (column "ALUT"), of registers ("Reg."), of all registers including those used by the synthesis tool ("T. reg."), and of hard 9-bit multiplication IP cores ("IP"). Compared to the manually-optimized versions, the automatic ones use slightly more ALUT and registers, mostly because they use two separate FIFOs for synchronization between the drivers Load0 and Load1, and the driver Compute (we changed the design of [2] to make it more generic). They also use more multipliers to perform tile address calculations, which could be removed by strength reduction.

Speed-ups are given in the column "S-U". Note that the optimized versions have a slightly smaller *maximal* running frequency than the original designs (column "M. freq." in MHz). But, if the designs already saturate the memory bandwidth at 100 MHz, running the systems at a higher frequency will not speed them up anyway. This maximal frequency reduction could come from more complex codes, the Avalon interconnect routing, and the use of double-port memories available in the FPGA. The use of such memories induces additional synthesis constraints.

5. CONCLUSION

In the context of HLS for FPGA, we proposed an automatic translation method to optimize, at source level, a kernel linked to an external DDR memory. Our method relies on a code restructuring that combines loop tiling (specified by the user), advanced communication coalescing and data reuse, pipelining of communicating processes in a double-buffer fashion, buffer size optimization, and optimized loop linearization. It has been implemented as a prototype, and the first experimental results show that the method is effective and gives promising results compared to handmade design. To our knowledge, this is the first time, in the context of HLS, that such accelerators are automatically generated.

6. REFERENCES

[1] C. Alias, F. Baray, and A. Darte. An implementation of lattice-based array contraction in the source-to-source translator Rose. In LCTES'07, ACM, 2007.

[2] C. Alias, A. Darte, and A. Plesco. Optimizing DDR SDRAM communications at C-level for automatically generated hardware accelerators. In *ASAP'10*, pages 329–332. IEEE Computer, July 2010.

[3] C. Alias, A. Darte, and A. Plesco. Kernel offloading with optimized remote accesses. Tech. Rep. RR-7697, July 2011.

[4] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *PPoPP'08*, pages 1–10, ACM, Feb. 2008.

[5] P. Boulet and P. Feautrier. Scanning polyhedra without Do-loops. In *PACT'98*, pages 4–9, IEEE, 1998.

[6] J. Cardoso and P. Diniz. *Compilation Techniques for Reconfigurable Architectures*. Springer, 2009.

[7] D. Chavarría-Miranda and J. Mellor-Crummey. Effective communication coalescing for data-parallel applications. In *PPoPP'05*, pages 14–25, ACM, 2005.

[8] W.-Y. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grained UPC applications. In *PACT'05*, pages 267–278. IEEE, 2005.

[9] P. Coussy and A. Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.

[10] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, Oct. 2005.

[11] H. Devos, K. Beyls, M. Christiaens, J. Van Campenhout, E. D'Hollander, and D. Stroobandt. Finding and applying loop transformations for generating optimized FPGA implementations. In *Transactions on HIPEAC I*, volume 4050 of *LNCS*, pages 159–178. Springer, 2007.

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

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

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

[15] P. Feautrier. Solving systems of affine (in)equalities: PIP's user guide. Tech. rep., fourth version, ENS-Lyon, 2001.

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

[17] E. D. Greef, F. Catthoor, and H. D. Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parallel Computing*, 23:1811–1837, 1997.

[18] A. Größlinger. Precise management of scratchpad memories for localizing array accesses in scientific codes. In *CC'09*, volume 5501 of *LNCS*, pages 236–250. Springer, 2009.

[19] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: A high-level synthesis framework for applying parallelizing compiler transformations. *VLSI Design*, pp. 461–466, 2003.

[20] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *Design Automation Conference (DAC'02)*, pages 628–633, 2002.

[21] B. Meister, A. Leung, N. Vasilache, D. Wohlford, C. Bastoul, and R. Lethin. Productivity via automatic code generation for PGAS platforms with the R-Stream compiler. In *Workshop on Asynchrony in the PGAS Programming Model (APGAS'09)*, June 2009.

[22] J. Xue. *Loop Tiling for Parallelism*. Kluwer, 2000.