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

Christophe Alias, Alain Darte, and Alexandru Plesco
Compsys, LIP, UMR 5668 CNRS, INRIA, ENS-Lyon, UCB-Lyon
Email: 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, not only intra-tile but also inter-tile data reuse 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 and should be kept locally. 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.

I. Introduction

HLS tools [1], e.g., Catapult-C, Impulse-C, Pico-Express, C2H, Gaut, 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 automatically 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. High-level transformations and optimizations are common in high-performance compilers, not yet in high-level synthesis. But their interest and their specificities for HLS have been demonstrated through handmade designs or restructuring methodologies [2], [3], [4], [5].

The goal of this paper is to show how the handmade restructuring proposed in [5] in the context of C2H, the Altera HLS tool, can be *fully automated*, thanks to advanced polyhedral techniques for code analysis and code generation, 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 not uniform: successive accesses to the same DDR row are pipelined an order of magnitude faster than when accessing different rows. Consequently, accessing data by blocks is a direct way of improving performances: if not, the hardware accelerator, even if it is 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 or when optimizing remote accesses for GPGPUs. More generally, this *offloading problem* occurs 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, thanks to the introduction of software-pipelined communicating processes. Data reuse within a strip, in particular inter-tile reuse, 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 the sets 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.

978-3-9815370-0-0/DATE13/©2013 EDAA

Code generation Driven 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 access functions, 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 II, we recall loop tiling and illustrate our technique through a synthetic example. Section III 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 IV, 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 [5], optimized by hand, and those optimized automatically thanks to our method.

II. Loop Tiling and Transfer Sets

Our method can be applied to offload a kernel on which *loop tiling* [7] 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, i.e., when loop bounds, `if` conditions, and array access functions are affine expressions of surrounding loops counters and structure parameters. Many compute-intensive kernels (e.g., from linear algebra, image processing) fit into this model.

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 within 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 an 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. Then, 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 $\mathcal{D}_S$ specifies an iteration domain $\mathcal{D}'_S$ of dimension $2n$. If the transformation θ leads to n permutable loops, then a valid sequential schedule of the tiled code is, for example: $\theta_{\text{tiled}}(S, I_1 \ldots I_n, \vec{i}) = (I_1, \ldots, I_n, \theta(S, \vec{i}))$.

[1]This restriction is enough for the kernels of Section IV and, more generally, when reads are approximated. When writes are approximated, the technique can be extended [6], but this goes out of the scope of this paper.

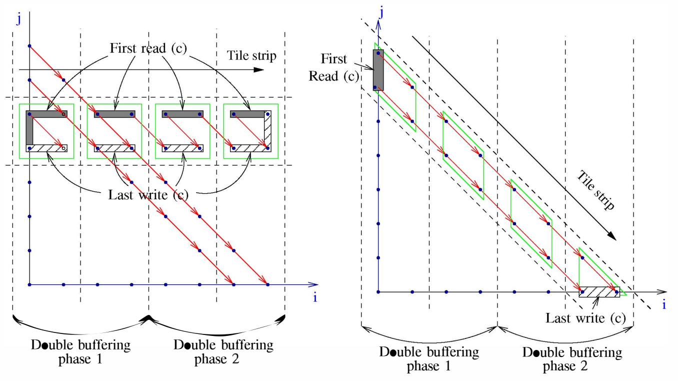

Figure 1. Different tilings and communications

Main example: We illustrate the different steps of our technique with the next code, which computes, in c, the product of 2 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 $(i, j) \mapsto (N - j, i)$, corresponding to a loop interchange and a loop reversal of the j loop, see the left of Fig. 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, if maximal data reuse is exploited, and in hatched white those that must be stored back by each tile.

With the tiling in the right of Fig. 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. As an illustration, 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 $J = \lfloor \frac{i}{b} \rfloor$. ■

The choice of the tiling is left to the user [2] and specified, thanks to a C pragma, as a function θ, such as $(N - j, i)$. Given S, $\mathcal{D}'_S$, and θ_{tiled}, standard polyhedral code generation could be used to generate a tiled code. However, applying such a preliminary rewriting step would complicate our subsequent optimizations. Instead, all analysis and code generation steps described hereafter are done with respect to this function θ.

[2]For the automatic selection of tile directions and size, see for ex. [7], [8].

The execution of each tile T is then decomposed, thanks to *communication coalescing* (see Section III), into three pipelined processes, for loading data, storing data, and performing the computations, as required for T. The function θ is then also used to express the relative schedules of these processes and to help us synthesize the adequate local buffers in a double-buffering fashion. Actually, "double-buffering" is a language simplification: we will not use two buffers, but one single (larger) buffer. However, two successive blocks of computation in a tile strip are indeed pipelined with two blocks of communications, which results in an overlapping between communications and computations (see Section IV).

To define the *transfer sets* $\text{Load}(T)$ and $\text{Store}(T)$ expressing the data to be loaded/stored before/after the execution of a tile indexed by T, we make the following assumptions:

- Elements in $\text{Load}(T)$ are loaded from external memory before the tile T starts, but in any order for a given T.
- Elements in $\text{Store}(T)$ are stored to external memory after the tile T ends, but in any order for a given T.
- Tiles are executed in sequence, following the sequential order specified by θ_{tiled}, in particular with increasing T.
- Similarly, transfers in $\text{Load}(T)$ (resp. $\text{Store}(T)$) are initiated before those in $\text{Load}(T')$ (resp. $\text{Store}(T')$) if $T < T'$.

In addition, we will make sure that a data is never loaded from the external memory if it has already been loaded earlier. Instead, it will be kept in local memory until its last use.

Back to the example: Consider Fig. 1 with the left tiling corresponding to $(i,j) \to (I,J,ii,jj)$ where $ii = N-j$ and $jj = i$. Our technique determines that the elements of c to be loaded are those depicted in grey boxes, indicated as "First read (c)". For that, given 3 parameters, the loop bound N, the outer tile index I, and the memory index m of array c, we derive (here for $b = 10$) an expression of the initial loop indices (i,j) that perform the first reads of c in the tile strip:

- $(i,j) = (0,m)$ if $0 \le -10I + N - m \le 9$ (this case corresponds to a vertical portion of c);
- $(i,j) = (10I-N+m, N-10I)$ if $1 \le 10I-N+m \le N$ (this corresponds to an horizontal portion).

Then, we derive $\text{Load}(T)$ (here $T = J$ with previous notations) as the set of data m read in T if this is the first access in the tile strip indexed by I:

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

How to get these load/store sets (basically, first read/last writes) in general, how the corresponding transfers are orchestrated, and how the necessary local memories are dimensioned and addressed, is detailed in the rest of this paper. ■

III. 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 can then 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, for optimizing communications and scratch-pad memories [9], [10], [11], [12], [8], [13], [14], [15]. 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 is the approach implemented in the RStream compiler, as described in [13]. 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. Another important difference is that our technique performs loads from the external memory even when this memory is not fully up-to-date.

Our strategy consists of 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 is that, unlike for previous approaches where the resulting 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 [13], [10], [8] 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 IV-B.

Our technique relies on the following definitions and theorems. For a tile T, let $\text{In}(T)$ 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 $\text{Out}(T)$ be the data written in T. We assume $\text{In}(T)$ and $\text{Out}(T)$ to be exact. The following theorem gives a 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 1: The functions Load and Store defined by

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

avoid useless transfers and reduce lifetimes in local memory.

Theorem 1 specifies optimized transfers based on set operations. Instead of relying on such operations, which could be done with the libraries Omega or ISL [3], our implementation uses an alternative approach based on PIP, a tool for parametric linear programming [16]. Assuming that the analyzed kernel fits in the polytope model, i.e., has affine loop bounds and access functions, we define:

- $\text{FirstOpReadBeforeWrite}(\vec{m})$, first operation that accesses an array cell indexed by $\vec{m}$, if it is a read.
- $\text{LastOpWrite}(\vec{m})$, last operation accessing $\vec{m}$ as a write.

Theorem 1 can then be reformulated as follows:

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

- $\text{Load}(T) = \{\vec{m} \mid \text{FirstOpReadBeforeWrite}(\vec{m}) \in T\}$
- $\text{Store}(T) = \{\vec{m} \mid \text{LastOpWrite}(\vec{m}) \in T\}$

$\text{Load}(T)$ gives the data accessed for the first time in T if this is a read, $\text{Store}(T)$ the data written for the last time in T.

$\text{FirstOpReadBeforeWrite}(\vec{m})$ is obtained by first extracting the set of operations accessing $\vec{m}$. Then, we compute the access 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 data-flow analysis [17]. More precisely, in the polytope model, reads (this is similar for writes) 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 $\vec{i} \in D$:

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

(If c occurs more than once in S, each access is distinguished.) Now, remember that S is given an affine schedule θ_S, see Section II. 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:

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

Then, we use PIP to compute the lexicographic minimum of $\text{Read}(\vec{m}, S)$. The result depends on the parameters (in our example, the loop bound N, the outer tile index I, the memory cell $\vec{m}$) and is presented as a discussion on their values (and possibly some additional parameters expressing integer division), more precisely as a tree of affine conditions (a quasi-affine selection tree or Quast in PIP's terminology), where each result (leaf of the tree) is expressed as an affine function. When several references exist (reads and writes) to the array c, the previous process is applied to each reference and the resulting piece-wise affine functions are combined, with standard Quasts combinations and simplifications, to get the global minimum. Of course, for computing $\text{FirstOpReadBeforeWrite}(\vec{m})$,

[3]Omega: http://chunchen.info/omega/, ISL: http://freecode.com/projects/isl

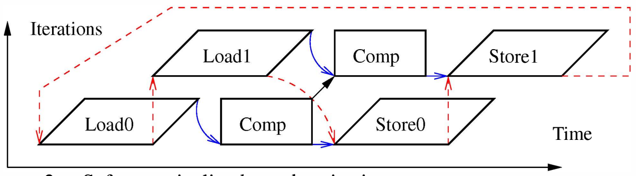

Figure 2. Software-pipelined synchronizations.

only the cases where the minimum is a read (and not a write) are kept. Then, following Theorem 2, it remains to add the constraints that express the fact that an operation $\text{FirstOpReadBeforeWrite}(\vec{m})$ belongs to a given tile T. The final result is described as a relation between $\vec{m}$ and T that can be read as a set of value $\vec{m}$, parameterized by T to get the sets $\text{Load}(T)$ and $\text{Store}(T)$. Similarly, $\text{Store}(T)$ is obtained by maximization, through the computation of $\text{LastOpWrite}(\vec{m})$

Back to the example: For the left tiling of Fig. 1, computing $\text{FirstOpReadBeforeWrite}(\vec{m})$, for the tile strip indexed by I, amounts to finding (i, j) such that (J, ii, jj, i, j) is lexicographically minimum, 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 the tile size b is fixed. After simplifications, we obtain the expression given in Section II for $b = 10$. Finally, given the schedule $\theta(i, j) = (N - j, i)$, we put back the constraint $\lfloor \frac{i}{b} \rfloor = T$, where T is a parameter indexing tiles, and, from this relation between T and $\vec{m}$, we derive the set $\text{Load}(T)$ given in Section II. ■

IV. Application to HLS for FPGA

We now use the theory developed in Section III 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 [5]. 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 w.r.t. the schedule θ. This makes the whole technique transparent, without even generating an initial loop tiling.

A. Synchronization of computation/communication processes

Following the methodology of [5], we generate 5 functions (called *drivers*) to be translated by C2H into separate hardware accelerators. For each tile strip, the Compute driver executes all computations of tiles in sequence, whereas, for communications, the tiles are processed by pairs, by 2 load and 2 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. 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 Fig. 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) 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 C2H specificities were detailed in [5].

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. Several software pipelines are possible, the one implemented in our tool is as depicted in Fig. 2. It is captured as follows. 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):

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

B. 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 rely on hardware-inexpensive array contraction based on modular mappings [19], [20]: 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. When the array index functions are translations w.r.t. the loop indices, as in `a[i][j-1]`, the set of live array cells is a window sliding during a tiled program execution, allowing efficient memory optimizations. The framework presented in [20] generalizes this particular situation, given an analysis of live array cells.

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, although correct, this produces inefficient code, mostly due to C2H constraints. It is better to generate each kernel as a single "linearized" loop executing one instruction per iteration, using the Boulet-Feautrier algorithm [21]. This avoids the penalty due to the pipeline of inner loops that must be drained (see Sect. IV-A). Also, as recalled in Sect. III, 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.

C. 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. This C source code is generic and cannot be immediately compiled C2H. At this point, a few modifications by hand are needed, 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 generate codes with the right initiation intervals, etc. These changes are minor and systematic, but they are not integrated yet in our code generator and take time when performed by hand.

The study provided in [5], 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 [5] (with the same tiling), DMA transfer, sum of vectors (VS), matrix multiply (MM), to check if we could achieve the same performance automatically. Matrix multiply, the main example demonstrated for the RStream compiler [13], is already, for circuit generation, very involved: 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 (i) because DDR requests are reorganized to get successive accesses on the same row as much as possible, (ii) because some communications overlap computations, and (iii) 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

Kernel	ALUT	Reg.	Tot. reg.	DSP	Max. freq.	Speed-up
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 I. SYNTHESIS: ORIGINAL, MANUAL, AUTOMATIC

memories involved to store the data locally. This is illustrated in Table I, which gives different parameters measuring the hardware usage: the number of look-up tables (column "ALUT"), of registers ("Reg."), of registers used by the whole system ("Total reg."), and of hard 9-bit DSP cores ("DSP"). Compared to the manually-optimized versions, the automatic ones use slightly more ALUT and registers, mostly because they use 2 separate FIFOs for synchronization between the drivers Load0 and Load1, and the driver Compute (we changed the design of [5] 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". Optimized versions have a slightly smaller *maximal* running frequency than the original ones (column "Max. 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, which induces additional synthesis constraints.

V. 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 performs as good as our previous fully-optimized handmade designs. To our knowledge, this is the first time, in the context of HLS, that such accelerators are automatically generated. Our method is also a generic form of kernel offloading to a distant platform and thus could be interesting in other contexts.

Our current implementation, exposed in this paper, is so far limited to the case where the transfer sets Load(T) and Store(T) can be built exactly. In theory, the case where the sets In(T) and Out(T) (data read/written in T) are approximated can be handled. This will give the opportunity to handle more irregular codes and to approximate the transfer sets if this is more efficient. Another interesting extension is to analyze and to generate codes in a parametric fashion w.r.t. tile sizes. These extensions have still to be implemented and validated.

REFERENCES

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

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

[3] 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 High-Performance Embedded Architectures and Compilers I*, ser. LNCS, P. Stenström, Ed. Springer, 2007, vol. 4050, pp. 159–178.

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

[5] C. Alias, A. Darte, and A. 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, Jul. 2010, pp. 329–332.

[6] ——, "Kernel offloading with optimized remote accesses," Inria, Tech. Rep. RR-7697, Jul. 2011.

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

[8] 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 *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08).* Salt Lake City, UT, USA: ACM, Feb. 2008, pp. 1–10.

[9] D. Chavarría-Miranda and J. Mellor-Crummey, "Effective communication coalescing for data-parallel applications," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05).* Chicago, IL, USA: ACM, 2005, pp. 14–25.

[10] M.Kandemir and A. Choudhary, "Compiler-directed scratch pad memory hierarchy design and management," in *Proceedings of the 39th annual Design Automation Conference (DAC'02)*, 2002, pp. 628–633.

[11] W.-Y. Chen, C. Iancu, and K. Yelick, "Communication optimizations for fine-grained UPC applications," in *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05).* IEEE Computer, 2005, pp. 267–278.

[12] I. Issenin, E. Borckmeyer, M. Miranda, and N. Dutt, "DRDU: A data reuse analysis technique for efficient scratch-pad memory management," *ACM Transactions on Design Automation of Electronics Systems (ACM TODAES)*, vol. 12, no. 2, Apr. 2007, article 15.

[13] A. Leung, N. Vasilache, B. Meister, M. M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin, "A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction," in *3rd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU'10)*, ser. Held with ASPLOS XVI. Newport Beach, CA: ACM, Mar. 2011, pp. 51–61.

[14] J. Cong, H. Huang, C. Liu, and Y. Zou, "A reuse-aware prefetching scheme for scratchpad memory," in *Proceedings of the 48th annual Design Automation Conference (DAC'11)*, 2011, pp. 960–965.

[15] S. Guelton, F. Irigoin, and R. Keryell, "Compilation for heterogeneous computing: Automating analysis, transformations, and decisions," Ecole des Mines de Paris, Tech. Rep. A-450, 2011.

[16] P. Feautrier, "Parametric integer programming," *RAIRO Recherche Opérationnelle*, vol. 22, no. 3, pp. 243–268, 1988, corresponding software tool PIP: http://www.piplib.org/.

[17] ——, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, Feb. 1991.

[18] A. Größlinger, "Precise management of scratchpad memories for localizing array accesses in scientific codes," in *International Conference on Compiler Construction (CC'09)*, ser. LNCS, O. de Moor and M. Schwartzbach, Eds., vol. 5501. Springer-Verlag, 2009, pp. 236–250.

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

[20] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1242–1257, Oct. 2005.

[21] P. Boulet and P. Feautrier, "Scanning polyhedra without Do-loops," in *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, 1998, pp. 4–9.