Area Implications of Memory Partitioning for High-Level Synthesis on FPGAs

Luca Gallo[1,2], Alessandro Cilardo[1], David Thomas[2], Samuel Bayliss[2] and George A. Constantinides[2]

[1]University of Naples Federico II, Via Claudio 21, Napoli, 80125 , *luca.gallo@unina.it*
[2]Imperial College London, London SW7 2BT, U.K.

***Abstract*—FPGAs normally have numerous independent memory banks that can be accessed simultaneously, potentially offering a very large memory bandwidth. Adopting a suitable application-based memory partitioning strategy is thus vital to take full advantage of the memory architecture. In addition to improving the potential memory bandwidth, partitioning also affects the area complexity of the generated system because the required steering logic depends on the partitioning scheme. This work describes the area implications of a lattice-based memory partitioning technique in the context of high-level synthesis for FPGAs. Experimental results with a commercial HLS tool show that the proposed partitioning technique improves area efficiency compared to alternative approaches.**

I. Introduction

Complex FPGA-based systems are increasingly being designed relying on high-level software-like paradigms, e.g. parallel programming, in conjunction with high-level synthesis (HLS) [1], [2]. In this scenario, the quality of results, e.g. the area cost resulting from the automated translation [3], is still an issue. While several literature works focus on optimizing the mapping of computation cores on the FPGA, or even defining the architecture of the on-chip interconnect based on the application characteristics [4], [5], few contributions are concerned with the customization of the memory architecture. In fact, FPGAs contain distributed fine-grained memory elements which, when exploited effectively, can offer a considerable bandwidth. To actually take advantage of this potential, HLS needs suitable memory partitioning techniques to parallelize memory accesses while minimizing the resulting area overhead. A few works focus on high-level source transformations enabling the code-level parallelism to match the physical parallelism offered by the FPGA memory architecture [6], [7], [8]. In [6] and[7], memory accesses in different loop iterations are partitioned across different memory banks and scheduled in the same cycle to minimize the number of required banks. However, both approaches are designed for one-dimensional arrays and cannot be directly used for FPGA-accelerated algorithms using loop nests. An approach that models memory ports as n-dimensional hyperplanes is proposed in [8], introducing a new mathematical abstraction for the partitioning problem. However, hyperplanes cannot describe all valid partitioning solutions, as shown in this paper. Moreover, there is limited understanding of the area overhead caused by the steering logic used to drive data from memory banks to the synthesized processing blocks. There are other works concerned with memory optimization in high-level synthesis, although not strictly related to the partitioning problem. Geometric programming is used in [9] to combine data reuse and data level parallelism under a fixed memory area constraint. Each processing element is assumed to be connected with a single memory block and data is assumed to be replicated when needed, which simplifies the problem but possibly incurs memory waste. Another related problem is the minimization of on-chip storage. Lattices are used in [10] to find a memory mapping function that can reduce the amount of physical memory needed for the execution of an algorithm by using some information on the liveness of variables.

This paper focuses on the effects of memory partitioning on the steering logic. The work exploits a lattice-based approach, presented in Section II, able to capture partitioning solutions that cannot be expressed as hyperplanes. A discussion of the different partitioning solutions is presented in Section III along with their impact on the area consumed by the steering logic. The experimental results presented in Section IV show that the lattice-based partitioning technique results in reduced area overhead because of its spatial regularity and improves the balance between area and performance. The conclusions of our works are drawn in Section V.

II. Memory partitioning solution

This section describes the memory partitioning technique. The approach is based on the polyhedral model, meaning the transformed loops can be arbitrarily nested and are required to be affine Static Control Parts (SCoPs).

A. Background

The polyhedral model compactly represents the statement iterations executed within a loop nest as integer points contained inside a *polyhedron*.

Definition 1: (**Polyhedron**). A Polyhedron P is a subset of $\mathbb{R}^n$ satisfying a finite number of affine inequalities: $P = \left\{\vec{x} : A\vec{x} + \vec{b} \geq 0\right\}$

Consider the loop nest given in Figure 1(a). The bounds of the loop nest can be represented as a polyhedron in $\mathbb{R}^2$, indicated by the gray area in Figure 1(c). The iterations of the loop nest coincide with the points of an *affine integer lattice* overlaid on that polyhedron.

Definition 2: (**Affine Integer Lattice**). An affine integer lattice is a subset of $\mathbb{Z}^n$ spanned by a linear combination of integer vectors with integer coefficients offset by an integer vector. The set of vectors is said to be the basis of the lattice. If B is the matrix having the integer vectors as columns, the lattice is said to be full-rank iff B is square and non-singular. A Z-Polyhedron is the intersection between a polyhedron and an affine integer lattice.

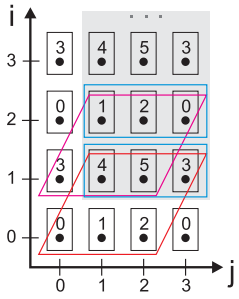
The (partial) ordering of the statement iterations can be captured using a linear schedule function which maps each statement iteration to a specific m-dimensional time-stamp. For

```
for(i=1; i<N; i++){
   parfor(j=1; j<4; j++){
      a[i][j]=a[i-1][j-1]+a[i][j];
   }
}
```

(a) Example Code

$$\Theta_S = \begin{pmatrix} 1 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i & j & N & 1 \end{pmatrix}^T$$

(b) Schedule Function



(c) Diagram

Fig. 1: A simple example representing a nested loop

a given statement S and the n-dimensional Z-Polyhedron representing the statement iterations, an m-dimensional schedule is $\Theta_S(\vec{x}) = T\vec{x}$ where T is a $n \times m$ matrix. Figure 1(b) gives a schedule function corresponding to the statement S in the example code. The schedule has four columns: two for the loop iterators, one for the parameter N, and one for the constant term. The parallelism is expressed by the fact that the schedule has a zero in the position corresponding to the parallel loop iterator: all instances with the same value of i can be executed in parallel, independent of the value of j.

B. Lattice-based partitioning

First of all, we need to model data-level parallelism by using suitable mathematical structures representing the memory locations that may be accessed in parallel. We denote $P_{S_{\vec{k}}}$ a parametric slice of the iteration domain of a statement S. It is defined as follows:

$$P_{S_{\vec{k}}} = \left\{ \vec{x} : \begin{pmatrix} A & \mathbf{0} \\ T & -I \end{pmatrix} \begin{pmatrix} \vec{x} \\ \vec{k} \end{pmatrix} \frac{\geq}{=} \begin{pmatrix} -\vec{b} \\ \mathbf{0} \end{pmatrix} \right\}$$

$\Theta_s(\vec{x})$ has a number of null columns equal to the number of parallel dimensions. $\vec{k}$ is a timestamp in the program schedule and, as a consequence, $P_{S_{\vec{k}}}$ captures the specific set of operations executed atomically when the iterators of sequential loops take the values specified in that timestamp. Consider the very simple example in Figure 1(c) where the inner loop is parallelizable. The blue boxes are the $P_{S_{\vec{k}}}$. Only two of them are shown corresponding to the first two values of the outer sequential loop iterator, namely $i = 1$ and $i = 2$.

We can use this to determine the memory cells accessed concurrently. Let $M_R(P)$ be the set of all memory cells in array R that are read or written by statement instances in the polyhedron P. $M_R(P)$ is calculated by taking the union of the images of P by all the access functions contained in the statement. As a consequence of this definition, $M(P_{S_{\vec{k}}})$ represents the set of memory cells accessed by the parallel iterations described by $P_{S_{\vec{k}}}$. The aim of the lattice-based methodology is to partition arrays into multiple banks so that parallel operations do not simultaneously access the same bank; in other words, we should obtain an allocation of memory cells such that all the integer points contained in each union $M(P_{S_{\vec{k}}})$ are bound to different memory ports. The red boxes in Figure 1(c) represent $M(P_{S_{\vec{k}}})$ for different values of $\vec{k}$. The numbered boxes represent the number of the on-chip memory ports to which we allocate the reference to array a with those coordinates. For instance, $a[3][1]$ is mapped to memory port 3. As shown in the figure, six different ports are used and the mapping is such that there never are two equal numbered boxes in each red box, i.e. full parallelization is achieved.

The lattice-based technique consists in partitioning the integer points, i.e. the memory locations, in different integer lattices. Hence, given an integer lattice L, each point belonging to L must be mapped onto the same on-chip memory port. In the case of Figure 1(c), the set of points allocated to memory port 0 can be represented by the lattice L having $B = \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}$ as a basis. All the lattices differ only by a translation factor and the one containing the origin is called **Fundamental Lattice**. The number of different lattices needed to cover the entire space is equal to the determinant of one of the translates. As a consequence, the determinant of the lattices must be equal to the number of available memory banks NB. The best Fundamental Lattice must be chosen so that the memory location in the sets $M(P_{S_{\vec{k}}})$ mapped to the same bank is minimum.

III. Impact of Partitioning on Area

This section analyzes the impact that memory partitioning techniques have on area in terms of LUTs (Look-Up Tables) and FFs (Flip-Flops). There are two main phenomena affecting steering logic: bank access conflicts and what we call bank switching. We will use the example shown in Figure 2 in this section, where part (a) shows the code, and parts (b), (c), and (d) illustrate three partitioning solutions. Of the three solutions, (b) can only be generated using our Memory Lattice technique, while (c) and (d) can be generated using both lattices and hyperplanes. (d) can be also regarded as a cyclic partitioning. The color of the integer points indicates the memory area accessed by a specific combination of the iterators. The datapaths corresponding to the three partitioning solutions are shown in Figure 3. The computational hardware operators remains the same but the steering logic varies according to the partitioning solution.

A. Conflicts

Conflicts are a first source of logic overhead incurred by memory partitioning. A HLS compiler must serialize the accesses by adding some extra logic to drive the address port of each memory bank and at the output of the data ports in order to store the addressed value temporarily. Figure 3.(c) shows the hardware resulting from Figure 2.(d) which features two conflicts per iteration. The multiplexers on the address port are necessary since each bank is addressed twice in each statement execution. The flip-flops are compulsory since the data must be buffered in order for the data-path to compute the right output value once all data has been loaded.

B. Bank Switching

In the above example, although memory conflicts are present, the output of each memory bank is always steered to the same multiplier. For example, the elements to be multiplied by w_0 are always allocated to memory bank 0. If every memory reference is always mapped to the same memory bank across all the instances of the statement, a direct connection of banks to operators is possible. On the other hand, consider the alternative partitioning choice in Figure 2.(c), which is a

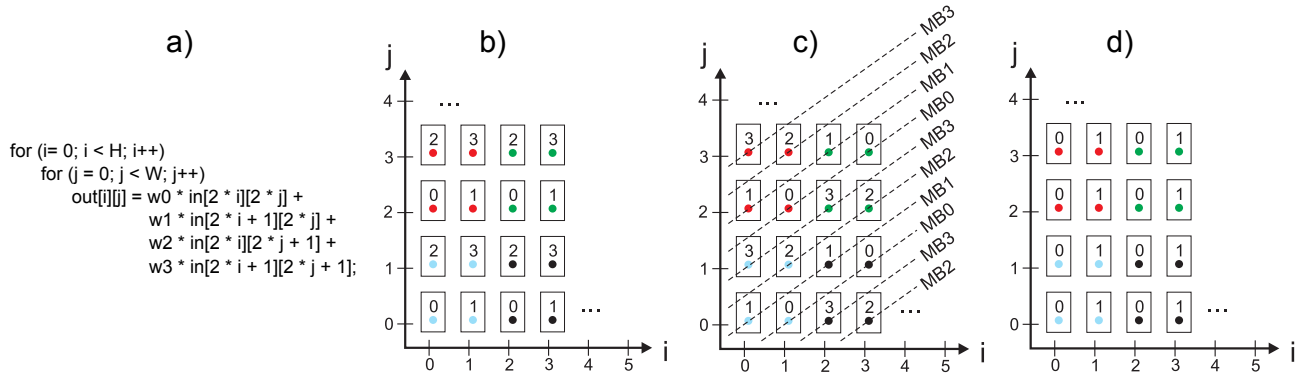


Fig. 2: a) Image Resizing Algorithm b) Lattice-based Solution (4 Banks), c) Hyperplane-based Solution (4 Banks), d) Cyclic solution (2 Banks)

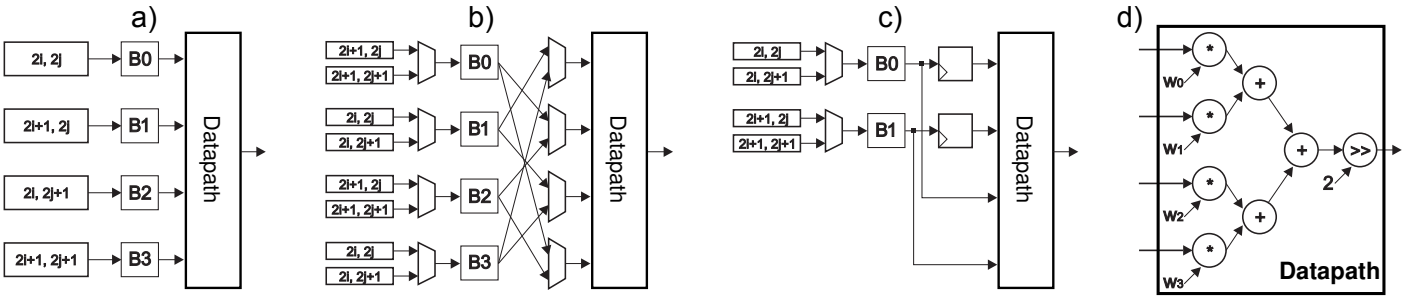


Fig. 3: Steering corresponding to several partitioning choices: a) Ideal; b) Bank switching; c) Conflicts. The Datapath is shown in d)

hyperplane-based solution. In this case, the reference $in[2i][2j]$ is mapped alternately to Bank 1 or Bank 3 depending on whether i is even or odd; the same happens for $in[2i][2j+1]$. This alternation requires more complicated steering logic, as the output of Bank 1 must be used by both multiplier w_0 and multiplier w_2 in alternate cycles. We call this phenomenon *bank switching*. Bank switching affects the area of a HLS-derived circuit in two ways, as illustrated in Figure 3.b). First, as with bank conflicts, extra multiplexers are needed before the memory bank address ports. Second, and more important, we need additional logic at the output of the data ports, since data channels may be wide and multiplexers with many wide inputs are very expensive in FPGAs. Moreover, if the HLS tool is not able to recognize hardware sharing properly, instead of multiplexers we would incur a replication of the data-path. The more different banks a certain reference is mapped to, the worse the consequences of bank switching are in terms of area. In the worst case, we may have many multiplexers, each of which has one input for every memory bank. The solution depicted in Figure 2.(b) is able to avoid both bank switching and conflicts and leads to the improved data-path shown in Figure 3.(a). It cannot be obtained by using a hyperplane-based method.

IV. Results and Validation

This section provides some quantitative results concerning the technique presented above with respect to both latency and area. The methodology is then compared with a hyperplane-based strategy and with ordinary cyclic partitioning. Cyclic partitioning is a special case of hyperplane partitioning [8], in the same way hyperplane partitioning is a special case of our lattice-based approach. Furthermore, the section highlights a few general conclusions on the impact of partitioning on the quality of results of HLS tools for FPGAs. We used two benchmarks for evaluations: a 2D-Image Resizing Kernel and a Gauss-Seidel kernel.

A. Area Overhead

Figure 4.(a) shows the area trends for both the experiments, when using the Lattice-based technique and varying the number of memory banks. As far as LUTs are concerned, we have in both cases a minimum with four banks. In particular, in the case of the image resize algorithm, four banks completely avoid conflicts and bank switching. Also for the Gauss-Seidel kernel, the bank switching is minimized using four banks. By decreasing the number of banks we introduce conflicts (but no switching), whereas by increasing them we cause bank switching (but no conflicts) and, as a consequence, a larger area overhead. One interesting conclusion is that, although handling more memory ports requires additional logic for address and control signals generation, the area may decrease due to conflicts and switching. One of the conclusions drawn from this work is thus that *memory partitioning may be used for area reduction, as well as performance improvement*.

Returning to Figure 4.(a) and looking at FFs, they generally decrease as the number of banks is increased, since we require them in case of conflicts to keep data stable at the input of operators while sequentially reading data from banks. More specifically, *under the same conditions in terms of bank switching (i.e. the same number of memory references switching to a different memory bank in each iteration) an increase in the number of memory banks turns into a decrease in the number of FFs*.

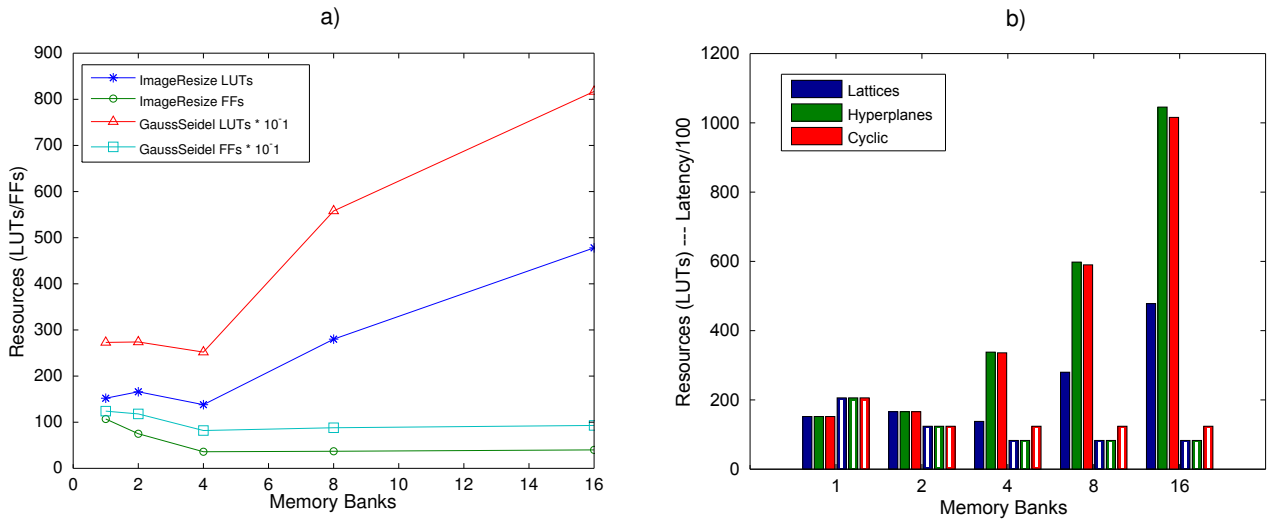


Fig. 4: Experimental results. a) Resources (FF/LUTs) vs. Number of banks; b) Comparison of the three techniques. The area is reported by solid bars, the latency by white filled bars.

B. *Comparisons of partitioning techniques*

The above conclusions refer to the lattice-based technique but similar considerations apply to the other techniques. However, the three techniques may perform very differently in terms of absolute area efficiency and performance. In particular, while there can be a significant advantage in terms of latency when comparing the lattice-based approach with the cyclic approach, the hyperplane-based solutions are general enough to achieve full parallelization in the memory accesses for the very regular benchmarks considered. The most important advantage of the lattice-based method over hyperplanes is related to the area overhead, as the bank switching effect occurs more frequently when only hyperplanes are used. In Figure 2.(b) and 2.(c) a clear example of the phenomenon is shown. A lattice-based solution helps keep regularity and ensures that each reference is always mapped to the same memory independent of the specific point of the iteration domain we are considering, unlike the hyperplane-based approach. Figure 4.(b) depicts latency and area obtained for the image resizing algorithm using the three techniques. As can be noticed, lattices and hyperplanes are comparable in latency but the area advantage increases with the number of banks up to more than two times.

V. Conclusions

This work was concerned with memory optimization in the context of high-level synthesis for FPGAs. In particular, we analyzed the main sources of area overhead related to memory partitioning, i.e. conflicts and bank switching. We described a memory partitioning technique based on integer lattices. Based on a set of real-world benchmarks, we demonstrated that the proposed technique is able to deal with conflicts and bank switching in a more effective way than other approaches because of the regularity of the underlying mathematical structure driving the partitioning.

Acknowledgment

This work was financially supported by the University of Naples Federico II and Compagnia San Paolo (STAR programme) and by the Engineering and Physical Sciences Research Council under grant numbers EP/I020357/1, EP/I012036/1, EP/K034448/1.

References

[1] A. Cilardo, L. Gallo, A. Mazzeo, and N. Mazzocca, "Efficient and scalable OpenMP-based system-level design," in *Procs. of Design, Automation Test in Europe Conference (DATE)*, 2013, pp. 988–991.

[2] A. Cilardo, L. Gallo, and N. Mazzocca, "Design space exploration for high-level synthesis of multi-threaded applications," *Journal of Systems Architecture*, vol. 59, no. 10, pp. 1171–1183, 2013.

[3] A. Cilardo, P. Durante, C. Lofiego, and A. Mazzeo, "Early prediction of hardware complexity in HLL-to-HDL translation," in *Procs. of the Int. Conference on Field Programmable Logic and Applications (FPL)*, 2010, pp. 483–488.

[4] A. Cilardo, E. Fusella, L. Gallo, and A. Mazzeo, "Automated synthesis of FPGA-based heterogeneous interconnect topologies," in *Procs. of the 23rd Int. Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2013, pp. 1–8.

[5] ——, "Joint communication scheduling and interconnect synthesis for FPGA-based many-core systems," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, March 2014, pp. 1–4.

[6] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, no. 2, pp. 15:1–15:25, Apr. 2011.

[7] P. Li, Y. Wang, P. Zhang, G. Luo, T. Wang, and J. Cong, "Memory partitioning and scheduling co-optimization in behavioral synthesis," ser. ICCAD '12. New York, NY, USA: ACM, 2012, pp. 488–495.

[8] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong, "Memory partitioning for multidimensional arrays in high-level synthesis," ser. DAC '13. New York, NY, USA: ACM, 2013, pp. 12:1–12:8.

[9] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Combining data reuse with data-level parallelization for FPGA-targeted hardware compilation: a geometric programming framework," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 28, no. 3, pp. 305–315, 2009.

[10] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," ser. CASES '03. New York, NY, USA: ACM, 2003, pp. 298–308.