Interplay of loop unrolling and multidimensional memory partitioning in HLS

Alessandro Cilardo and Luca Gallo
Department of Electrical Engineering and Information Technologies, University of Naples Federico II
via Claudio 21, 80125 Napoli, Italy,
Email: acilardo@unina.it, luca.gallo@unina.it

***Abstract*—This paper deals with memory partitioning in the context of high-level synthesis for FPGA technologies. In particular, the work focuses on the area overhead caused by partitioning and sheds light on the interplay with a technique commonly used in HLS, i.e., loop unrolling. As a practical outcome, the study proposes a solution to reduce the area overhead by appropriately controlling the degree of loop unrolling. The experimental results confirm the significance of the analysis as well as the effectiveness of the proposed optimization technique.**

I. Introduction

The ever growing complexity of electronic system design is increasingly requiring high-level software-like approaches [1], [2], [3] and methods for application-driven customization of the processing element architecture and even the on-chip interconnect [4], [5]. High-Level Synthesis (HLS) [3], in particular, has today emerged as an effective link between software programming and digital design. Current HLS tools can generate high-quality RTL code by enabling key optimization techniques, such as loop pipelining and unrolling, which expose the underlying parallelism by overlapping the execution of consecutive iterations of the original loop. In this scenario, memory partitioning [6], [7] is of fundamental importance because data accesses may cause potentially parallel iterations to be serialized due to structural conflicts on memory ports, resulting in significantly degraded performance and area costs. For platforms offering numerous fine-grained parallel memory blocks, such as FPGAs, partitioning is even more crucial as it may deeply impact the actual degree of utilization of the available memory. This paper focuses on the area overhead involved in memory partitioning. We analyze the area implications of partitioning, particularly the bank switching effect, which results in increased steering logic to route the data from the different memory banks to the processing blocks. As a practical outcome, the study proposes a solution to reduce the area costs related to the steering logic by appropriately controlling the degree of loop unrolling. The experimental results presented at the end of the paper confirm the impact of the presented area optimization technique.

The paper is organized as follows. Section II presents the relevant background in memory partitioning for HLS. Section III summarizes the formal model adopted for multidimensional partitioning throughout the paper. Section IV introduces the notion of bank switching and proposes a solution to minimize its impact. Section V discusses a few experimental results demonstrating the proposed technique. Section VI concludes the paper with some final remarks.

II. Related Work

Memory partitioning is today recognized to be a central issue in HLS. While few current commercial HLS tools, such as Xilinx VivadoHLS [8], offer the possibility of automating the partitioning of a multidimensional array, the implemented strategies are mostly suboptimal. In fact, they treat memory accesses on a statement-level basis rather than on an instance-level basis. As a consequence, they miss substantial opportunities for exploiting parallelism. For instance, in the case two memory accesses cannot be proved to involve two different physical banks *for all* instances of the statement, they are not parallelized. Furthermore, current tools only implement monodimensional partitioning schemes, i.e., they first linearize the array and then partition it according to a specific strategy, e.g. block or cyclic. On the other hand, academic research is actively investigating new methodologies and tools for improving automated partitioning in HLS. In [6] and [9] memory partitioning strategies are presented that improve the throughput and power efficiency of perfect loop nests. In addition to providing a partitioning solution, the techniques also determine a scheduling function for the memory accesses in the loop nest. The joint partitioning and scheduling reduces both area and power costs. The authors of [10] extend partitioning to support memory references containing modulo operations with limited memory padding. These partitioning techniques are however designed for one-dimensional arrays and require a linearization step in case of multidimensional array accesses. Hence, [11] introduces a new mathematical abstraction for representing a partitioning solution, i.e., hyperplanes. In [7] hyperplanes are shown not to capture potentially good partitioning solutions. [7] adopts lattice-based partitioning instead, which generalizes hyperplanes. The work in [12] adopts geometric programming to combine data reuse and data level parallelism under a fixed memory area constraint. There is a one-to-one correspondence between processing elements and memory banks and data is replicated when needed, which simplifies the problem but may potentially incur memory waste. Although the research on memory partitioning for HLS has been quite active during the last years, there are very few attempts aimed at understanding the effect of partitioning on area occupation and improving area results through code transformations. The authors of [13] recognize the presence of an area overhead due to address generation and data assignment. However, the two effects are not modeled mathematically and, importantly, the overhead estimation does not take into account bank switching, which may have a considerable impact on area efficiency, even more than the number of memory banks itself.

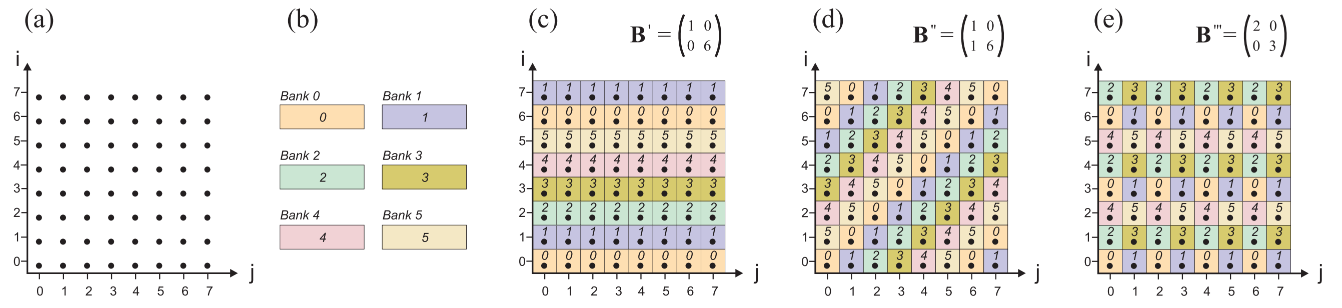

(a)
(b)
(c)
(d)
(e)
$\mathbf{B}' = \begin{pmatrix} 1 & 0 \\ 0 & 6 \end{pmatrix}$
$\mathbf{B}'' = \begin{pmatrix} 1 & 0 \\ 1 & 6 \end{pmatrix}$
$\mathbf{B}''' = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$
Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
i
j

Fig. 1. A few examples of lattice-based partitioning. a) Original two-dimensional array to be partitioned. b) Representation of the available memory banks. c)-e) Three partitioning solutions corresponding to three different bases.

III. Multidimensional Memory Partitioning

Generally speaking, partitioning an array A over NB physical banks consists of associating each separate portion of the array to a certain position in a different bank, possibly with a location-level granularity. Efficient partitioning must satisfy a number of conditions, particularly when the high-level code is synthesized to a parallel datapath: the mathematical transformation associating a bank/position to each original location must be easily computable, the least possible number of conflicts must arise as the datapath input ports access the physical banks, the steering circuitry connecting the banks to the datapath must be as simple as possible. A few examples of partitioning approaches include cyclic and block partitioning strategies [6]. Both first linearize the array. Then, cyclic partitioning assigns a memory location m to memory bank $m \bmod NB$, where NB is the number of banks, whereas block partitioning maps m to memory bank $\lfloor m/NB \rfloor$. An improved approach is proposed in [11]. They model memory ports as n-dimensional hyperplanes. However, by limiting their solutions to a single family of hyperplanes, they might ignore some potential solutions, as shown in [7], [14] Below we briefly review the lattice-based technique [7], which encompasses the others as particular cases.

Definition 1. (**Affine Lattice**) *An affine integer lattice $\mathcal{L}$ is a subset of $\mathbb{Z}^n$ spanned by an integer linear combination of independent vectors $\vec{b_i} \in \mathbb{Z}^n$, $i = 0 \dots m-1$, plus an integer vector offset $\vec{t}$:*

$$\mathcal{L} = \{z_0\vec{b_0} + z_1\vec{b_1} + \ldots + z_{m-1}\vec{b}_{m-1} + \vec{t},\ \ z_0, z_1 \ldots z_{m-1} \in \mathbb{Z}\}$$

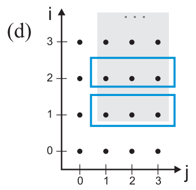
The ordered set of vectors $\{\vec{b_i}\}$ is said to be a *basis* of the lattice. If $\mathbf{B}$ is the matrix having such integer vectors as its columns, the lattice is said to be *full-rank* if and only if $\mathbf{B}$ is a square (nonsingular) matrix. The *determinant* of a lattice is the absolute value of the determinant of one of its bases. All the bases that can be obtained from each other through a unimodular transformation generate the same lattice. Geometrically, a lattice is a subset of $\mathbb{Z}^n$ of regularly spaced points. The pattern that such points follow in space is dictated by the basis $\mathbf{B}$. Essentially, lattice-based partitioning [7] regards an n-dimensional array in memory as an n-dimensional integer space $\mathbb{Z}^n$. It can be shown that $\mathbb{Z}^n$ can always be partitioned in disjoint lattices. Each lattice coincides with a memory bank, i.e., all the integer points belonging to a specific lattice are mapped to the same bank. All the lattices are translates of each other, and the one that contains the origin is called the *fundamental lattice*. The number of such translates (including the fundamental lattice) is equal to the determinant of one of its bases, which is hence also equal to the number of banks.

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

(b)
```
for(i=1; i<N; i++){
    Y[i][1]=A[i-1][0]+A[i][1];
    Y[i][2]=A[i-1][1]+A[i][2];
    Y[i][3]=A[i-1][2]+A[i][3];
}
```

(c)

$$\Theta = (0, 1) \cdot (j, i)^{\mathrm{T}}$$

(d)


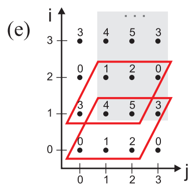
(e)


Fig. 2. a) A simple example of a parallel loop. b) The corresponding unrolled code. c) The schedule function of the statement in the loop body. d) A representation of the iteration domain. The j, i pairs corresponding to the iteration domain of the loop are emphasized by the gray backgroud. The blue rectangles represent parallel sets of iterations. e) The red parallelograms represent the memory locations accessed by the parallel iterations in part (d).

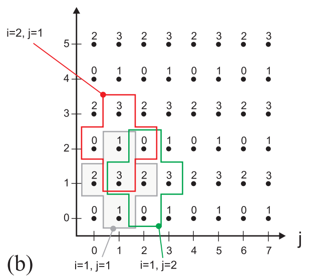
Figure 1 shows a few examples of lattice-based memory partitioning in a two-dimensional case, highlighting the choices corresponding to three different bases with six physical memory banks available. For instance, in Figure 1.e we choose basis $\mathbf{B}''' = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$. The set of points marked with zero forms the fundamental lattice of basis $\mathbf{B}'''$, as each point having coordinates $(2j, 3i)$ belongs to that lattice. Similarly, the sets of points marked with other numbers are affine lattices having the same basis and differing from the fundamental lattice by an offset vector $\vec{t}$. For example, the points corresponding to translate 5 are obtained as $\mathbf{B}''' \cdot (j, i)^T + (1, 2) = (2j+1, 3i+2)$.

A. Representing memory access patterns

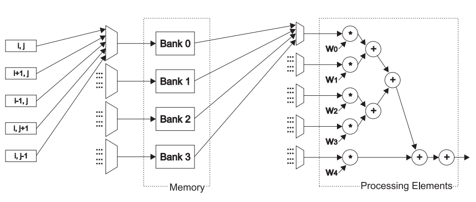
Most of the above cited techniques for partitioning rely on the polyhedral model [15], [16], [17], a mathematical abstraction that can be used to model compactly static control loop nests where loop bounds and memory references are affine functions of the loop iterators. The code in Figure 2.a provides an example. In this section, we only recall a few fundamental concepts that are used in the presented method-

```
void function(int A[N][N], Aout[N][N]){
  int i,j;
  for (i=1; i<=N-1; i++)
    for (j=1; j<=N-1; j++)
      Aout[i][j] = w0 * A[i][j]   +
                   w1 * A[i-1][j] + w2 * A[i][j-1] +
                   w3 * A[i][j+1] + w4 * A[i+1][j];
}
```

(a)



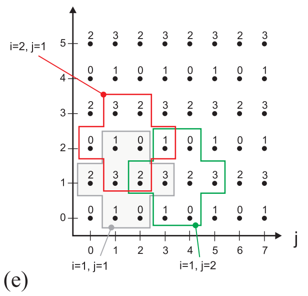
(b)



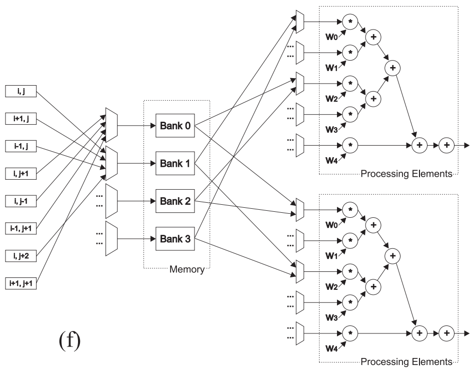
(c)

```
void functionUnrolled(int A[N][N], Aout[N][N]){
  int i,j;
  for (i=1; i<=N-1; i++)
    for (j=1; j<=N/2-1; j++)
      Aout[i][2*j-1]= w0 * A[i][2*j-1]   +
                      w1 * A[i-1][2*j-1] + w2 * A[i][2*j-2]  +
                      w3 * A[i][2*j]     + w4 * A[i+1][2*j-1];
      Aout[i][2*j]  = w0 * A[i][2*j]     +
                      w1 * A[i-1][2*j]   + w2 * A[i][2*j-1] +
                      w3 * A[i][2*j+1]   + w4 * A[i+1][2*j];
}
```

(d)



(e)



(f)

Fig. 3. Motivating example: a) Code. b) Data sets for a few iterations. c) Synthesized datapath. Unrolled version of the same example: d) Code. b) Data sets for a few iterations. c) Synthesized datapath. Notice that N is assumed even.

ology. The *iteration vector* associated with a certain statement appearing in the body of an n-level nested loop is a vector containing n components, each representative of an index of the loop nest. The code in Figure 2.a contains two nested `for` loops. The statement in the loop body has iteration vector $\vec{v} = (j,i)^T$. The *iteration domain* of a statement can be represented as a finite polyhedron containing as many integer points as the number of statement instances. The coordinates of each integer point in the polyhedron represent a specific value of the loop iterators. If the statement is surrounded by n loops, such polyhedron is n-dimensional. The statement in Figure 2.a has an iteration domain coinciding with the polyhedron $P = \{\vec{v} : 1 \leq i \leq N-1, 1 \leq j \leq 3\}$. A *memory access function* $F(\vec{v})$ of a memory reference to an array A in a specific statement S is a correspondence between the iteration domain of S and the set of memory locations of A. If the loop nest is amenable to be described by the polyhedral model, each $F(\vec{v})$ is necessarily affine, hence of type $F(\vec{v}) = \mathbf{F} \cdot \vec{v} + \vec{t_0}$. Consider the access $A[i-1][j-1]$ in the statement of Figure 2.a. The corresponding memory access function is $F(j,i) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} j \\ i \end{pmatrix} + \begin{pmatrix} -1 \\ -1 \end{pmatrix} = \begin{pmatrix} j-1 \\ i-1 \end{pmatrix}$. Each instance $\vec{v}$ of a statement S in a loop nest can be associated to a point in time based on a *schedule function* $\Theta_S(\vec{v})$, which in fact establishes an ordering of the instances of S, possibly based on a parallelizing transformation. The schedule function has the form $\Theta_S(\vec{v}) = \mathbf{\Theta} \cdot \vec{v}$. In Figure 2 the schedule has two columns, shown in Figure 2.c, corresponding to the loop iterators j and i, respectively. The parallelism in the code can be exposed by transforming the iteration domain and making the schedule matrix $\mathbf{\Theta}$ have a zero-column corresponding to each parallel `for` loop. In Figure 2.a, the outer loop was kept serial, while the inner loop was parallelized (denoted with `parfor`). Consequently, the schedule is one-dimensional, i.e., the matrix $\mathbf{\Theta}$ has one row. The parallelism can be easily recognized by the fact that the schedule has a zero term 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 taken by j. Commonly, this parallelization can be expressed in a HLS tool flow through *unrolling*. Figure 2.b shows the parallelized `for` nest as obtained through full unrolling of the innermost loop. Figure 2.d depicts the subsets of iterations in the iteration domain that are executed concurrently. The concurrent iterations corresponding to the same subset are embraced by a blue rectangle. They share the same value of the schedule function. Last, we define *data set* the subset of memory locations that are accessed by a subset of concurrent iterations. In Figure 2.e data sets are represented as red parallelograms: all memory locations in the same parallelogram can potentially be accessed simultaneously based on the given schedule Θ. Please notice that, for the sake of simplicity, the chart in Figure 2.e represents the points of both the iteration domain (spanned by the components of the iteration vector, j and i) and the memory locations (spanned by the reference subscripts, which here are bidimensional and depend again on j and i). In general, the dimensionalities of the iteration domain and the memory space need not necessarily be the same. Figure 2.e also displays the bank index associated with each memory location for a specific partitioning solution.

IV. Bank Switching

The main aim of this paper is to understand the impact of a side-effect of memory partitioning, that we call *bank switching*. Normally, in a HLS tool flow the statements in a loop body are synthesized to a parallel physical datapath, which processes the locations embraced by the data set concurrently. The input/output ports of the datapath correspond to the read/write memory references in the loop body. When the memory is partitioned over multiple banks, the locations in the data set can be accessed in parallel, ideally in a single access cycle when a one-to-one correspondence between data set locations and memory banks is achieved [7]. Furthermore, in case each *specific* reference in the data set always corresponds to the same bank, the connection between the respective datapath port and the physical bank can be direct, otherwise access must be multiplexed. Bank switching essentially refers to different instances of the same memory reference accessing different banks. We say that reference $A[F(\vec{v})]$ has an amount of bank switching equal to bs, if it accesses bs different memory banks over the whole execution. To clarify the above definition, refer to the example in Figure 3. The code contains a two-level nested loop representing a cross-shaped window sliding over a bidimensional array. Figure 3.a shows the code while Figure 3.b illustrates the array and the data sets accessed by a few iterations of the loop (coinciding with the cross-shaped sliding window). At each iteration, the window picks five different memory locations and multiplies the values by five different weights. Figure 3.b also shows a possible partitioning solution using four memory banks: the number associated with each memory location represents the memory bank to which it is allocated. Notice that the weight associated with each memory location depends on the position within the sliding window. For example, the point located in the middle of the window is always multiplied by w_0, whereas the point on the left by w_2. As the window slides through the array, each w_i is multiplied by a value from a different memory bank, as Figure 3.b also highlights, causing the bank switching effect. Take weight w_0 as an example. In iteration $(i = 1, j = 1)$, the bank containing the data to be multiplied by w_0 is Bank 3, whereas in the consecutive iteration $(i = 1, j = 2)$ it is located in Bank 2.

Bank switching has a direct effect on the datapath synthesized from the high-level code. Figure 3.c shows the datapath corresponding to our example. Notice the additional steering logic required at the output data ports of the memory banks and at the address input ports. In fact, over the different iterations of the loop, each multiplier takes input values from all of the memory banks. In addition, bank switching results in a more complicated logic for the generation of the addresses to the memory banks, since each bank must be addressed by one of five different combinations of the (i, j) indices as the loop proceeds through the iterations. In other words, we also need additional multiplexers driving different addresses to each of the four memory banks, depending on the iteration. The steering logic caused by bank switching may cause a significant area overhead, which is not inherently required by the algorithm itself, but rather by the way it is coded and the particular use of the available memory banks.

A. Interplay between loop unrolling and bank switching

Loop unrolling is a common technique in HLS. Although unrolling usually results in replicated datapaths corresponding to the unrolled loop iterations, we show here that its interplay with bank switching may enable improved efficiency in the use of the hardware resources. Consider Figure 3.d, where the inner loop of the previous code has been unrolled by two iterations. Like the previous example, Figure 3.e and Figure 3.f depict, respectively, the partitioned memory and the synthesized datapath. Now, there is no bank switching affecting the inner loop because of the unrolling transformation. In fact, when sliding the window along the j axis, every location of the window always happen to touch the same memory bank (on the other hand, when sliding the window along the i axis, bank switching still occurs, with every location of the window touching two different banks). This results in significantly smaller multiplexers at both the address input port and the data output port of the memory banks. Of course, more resources are going to be used for the replicated processing elements in the datapath because of the unrolling, but they are purely used for computation rather than being steering overhead, making the resulting design more area and power efficient.

Our aim is to analyze the connection between unrolling and bank switching when an n-dimensional memory array A is partitioned lattice-wise as summarized in Section III. Notice that lattice-based partitioning provides to date the most general approach, encompassing other techniques like hyperplanes as special cases [7]. Let the $\mathbb{Z}^n$ set represent the whole n-dimensional memory space containing A, and $\mathcal{L} = \{\mathbf{B}\cdot\vec{z}, \vec{z} \in \mathbb{Z}^n\}$ be the integer lattice used for partitioning. $\mathcal{L}$ is a subset (and a subgroup) of $\mathbb{Z}^n$. Overall, there are $NB = \det(\mathbf{B})$ different translate lattices, constituting a partition of $\mathbb{Z}^n$, each corresponding to a physical memory bank. To associate a different location to the corresponding bank, we rely on the following property, proven in [18]. Let $\mathbf{S} = \mathbf{U}_1\mathbf{B}\mathbf{U}_2$ be the *Smith Normal Form* (SNF) of the lattice basis $\mathbf{B}$. $\mathbf{S} = \{s_{ij}\}$ is a diagonal matrix and its diagonal elements, denoted $\vec{s} = (s_{00}, s_{11} \dots s_{n-1,n-1})$, are such that s_{ii} divides $s_{i+1,i+1}$, while $\mathbf{U}_1$ and $\mathbf{U}_2$ are unimodular matrices, and hence $NB = \det(\mathbf{B}) = \det(\mathbf{S}) = \prod_{i=0}^{n-1} s_{ii}$. Then, the *modular mapping* defined by $\sigma(\vec{z}) = \mathbf{U}_1 \cdot \vec{z} \bmod \vec{s}$, $\vec{z} \in \mathbb{Z}^n$ has $\mathcal{L}$ as the *kernel*, i.e., $\vec{z} \in \mathcal{L} \Rightarrow \mathbf{U}_1 \cdot \vec{z} \bmod \vec{s} = \vec{0}$. As a consequence, the quotient group $\mathbb{Z}^n/\mathcal{L}$, containing the NB translates of $\mathcal{L}$, each corresponding to a memory bank, is in one-to-one correspondence with the different values taken on by the modular mapping $\sigma(\vec{z}) = \mathbf{U}_1 \cdot \vec{z} \bmod \vec{s}$. In other words, each bank can be denoted by an n-dimensional index $\vec{b_F} = (b_0, \dots, b_{n-1})$ with $b_i < s_{ii}$ $\forall i$ and, given a particular memory reference of indices $\vec{z} = (z_0, \dots, z_{n-1})$ (e.g., $A[3][5]$ in a bidimensional space), the corresponding bank can be simply obtained as $(b_0, \dots, b_{n-1}) = \sigma(\vec{z}) = \mathbf{U}_1 \cdot \vec{z} \bmod \vec{s}$. Consider now a memory access in the code, and the corresponding access function $\vec{z} = F(\vec{v}) = \mathbf{F} \cdot \vec{v} + \vec{t_0}$. The bank accessed by the memory reference in a given iteration $\vec{v}$ is $\vec{b_F}(\vec{v}) = \mathbf{U}_1(\mathbf{F} \cdot \vec{v} + \vec{t_0}) \bmod \vec{s}$. Quantifying bank switching is equivalent to computing how many different values are taken by the modular mapping $\vec{b_F}(\vec{v})$ as $\vec{v}$ spans $\mathbb{Z}^n$ (we assume that the memory array is large enough to take all the different values of $\vec{b_F}$, which is normally the case in practice). Although it is possible to derive a closed formula for deriving the amount of bank switching based on $\mathbf{B}$ and $\mathbf{F}$, we omit here the details due to the lack of space. However, we show how the previous formulation can drive the choice of the unrolling factors in order to reduce, or even eliminate, the bank switching effect. Notice that, as discussed later in the paper, we assume that the choice of the unrolling factors is not constrained here by possible data dependencies across iterations.

To formalize the impact of loop unrolling notice that, in terms of memory accesses, the essential effect of unrolling is to change the expressions of the memory access functions.

As an example, the first memory reference in the unrolled loop of Figure 3.d is $F'(j,i) = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} j \\ i \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix}$ instead of the original function $F(j,i) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} j \\ i \end{pmatrix}$. In general, let e_i and e_j be the unrolling factors of the two loops in the nest. Each memory access function of the form $F(j,i) = \begin{pmatrix} f_{00} & f_{01} \\ f_{10} & f_{11} \end{pmatrix} \begin{pmatrix} j \\ i \end{pmatrix} + \vec{k}$ is transformed after unrolling into:

$$F'(j,i) = \mathbf{F}' \cdot \vec{v} + \vec{k'} = \begin{pmatrix} e_j \cdot f_{00} & e_i \cdot f_{01} \\ e_j \cdot f_{10} & e_i \cdot f_{11} \end{pmatrix} \cdot \begin{pmatrix} j \\ i \end{pmatrix} + \vec{k'}$$

with $\vec{k'}$ being a suitable constant vector (not affecting bank switching). The following result connects the amount of bank switching with the transformation induced by the unrolling factors. Let $\mathbf{U}_1 = \begin{pmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{pmatrix}$ and $\vec{s}$ refer to the SNF of the lattice basis $\mathbf{B}$ used for partitioning. Then, the bank mapping function is

$$\vec{b_F}(\vec{v}) = \mathbf{U}_1 \cdot \left[\mathbf{F}' \cdot \vec{v} + \vec{k'}\right] \bmod \vec{s} = \left[\mathbf{T} \cdot \vec{v} + \vec{k''}\right] \bmod \vec{s}$$

where $\mathbf{T} = \mathbf{U}_1 \cdot \mathbf{F}' = \begin{pmatrix} t_{00} & t_{01} \\ t_{10} & t_{11} \end{pmatrix} = \begin{pmatrix} q_{00} \cdot e_j & q_{01} \cdot e_i \\ q_{10} \cdot e_j & q_{11} \cdot e_i \end{pmatrix} =$

$$\begin{pmatrix} (f_{00}u_{00} + f_{10}u_{01}) \cdot e_j & (f_{01}u_{00} + f_{11}u_{01}) \cdot e_i \\ (f_{00}u_{10} + f_{10}u_{11}) \cdot e_j & (f_{01}u_{10} + f_{11}u_{11}) \cdot e_i \end{pmatrix}$$

and $\vec{k''} = \mathbf{U}_1 \cdot \vec{k'}$ is a constant term.

Bank switching is thus given by the number of different values taken on by $[\mathbf{T} \cdot \vec{v}] \bmod \vec{s}$. In case the two rows of $\mathbf{T} \cdot \vec{v}$ are a multiple of s_{00} and s_{11}, respectively, the above modular expression takes on always one value. A sufficient condition is ensured by the following choice for the unrolling factors[1]:

$$e_j^* = \operatorname{lcm}\left(\frac{s_{00}}{\gcd(q_{00}, s_{00})}, \frac{s_{11}}{\gcd(q_{10}, s_{11})}\right)$$

$$e_i^* = \operatorname{lcm}\left(\frac{s_{00}}{\gcd(q_{01}, s_{00})}, \frac{s_{11}}{\gcd(q_{11}, s_{11})}\right)$$

In fact, taking the first row as an example, we have that $e_j^* = \alpha \cdot \frac{s_{00}}{\gcd(q_{00}, s_{00})}$ and $e_i^* = \beta \cdot \frac{s_{00}}{\gcd(q_{01}, s_{00})}$ by the above choices. Write $q_{00} = q'_{00} \gcd(q_{00}, s_{00})$ and $q_{01} = q'_{01} \gcd(q_{01}, s_{00})$. Then, the first row of $\mathbf{T} \cdot \vec{v}$ is

$$q_{00}e_j^* \cdot j + q_{01}e_i^* \cdot i = q'_{00} \gcd(q_{00}, s_{00}) \cdot \alpha \frac{s_{00}}{\gcd(q_{00}, s_{00})} \cdot j +$$

$$q'_{01} \gcd(q_{01}, s_{00}) \cdot \beta \frac{s_{00}}{\gcd(q_{01}, s_{00})} \cdot i = s_{00}\left[q'_{00}\alpha \cdot j + q'_{01}\beta \cdot i\right]$$

Notice that, although we considered two nested loops for simplicity, the treatment can be easily generalized to the case of an n-level loop nest. Furthermore, the technique was discussed for a single memory reference. When more memory accesses to the same array are found in the loop body, the technique can still be applied to each reference separately, then picking for each unrolling factor the least common multiple of the different solutions determined.

[1] $\gcd(a, b)$ and $\operatorname{lcm}(a, b)$ denote, respectively, the greatest common divisor and the least common multiple of two integers a, b.

V. Experimental Evaluation and Discussion

In order to assess the impact of bank switching, we define the *area efficiency* as the product of latency, i.e., the overall execution time of a kernel, and area cost. The area cost of FPGA designs is quantified in terms of elementary components such as Look-Up Tables (LUTs), Flip-Flops (FFs), and, in some cases, Digital Signal Processing (DSP) blocks. Here we only consider LUTs primarily because FPGA designs are often LUT-bounded. To validate our results, we used three benchmarks: an image resizing kernel using bilinear interpolation, a 2D Gauss-Seidel kernel, and a 2D-Jacobi kernel. The C code featuring partitioned memory accesses for each case study was processed by VivadoHLS [8] targeting a Virtex-7 device. Table I summarizes the experimental results for the three benchmarks. The results refer to a scenario where the processed array is partitioned lattice-wise over four or eight banks. For each case, the table shows the area efficiency corresponding to five different unrolling configurations, normalized to the case of a single memory bank. The table also indicates the e_j^*, e_i^* factors identified by the formulas above, providing the minimum values necessary for avoiding bank switching and, hence, for not degrading the area efficiency because of additional steering logic.

TABLE I. EXPERIMENTAL RESULTS. THE $A \cdot L$ PRODUCT IS NORMALIZED TO THE CASE OF A SINGLE MEMORY BANK.

Image Resize			**2D Gauss-Seidel**			**2D Jacobi**		
4 banks ($e_j^*, e_i^* = 1, 1$)			4 banks ($e_j^*, e_i^* = 2, 2$)			4 banks ($e_j^*, e_i^* = 2, 2$)		
e_j	e_i	$A \cdot L$	e_j	e_i	$A \cdot L$	e_j	e_i	$A \cdot L$
1	1	0.36	1	1	1.87	1	1	2.01
2	1	0.48	2	1	1.47	2	1	0.64
2	2	0.43	2	2	0.71	2	2	0.38
4	2	0.37	4	2	0.91	4	2	0.39
4	4	0.35	4	4	0.82	4	4	0.39
8 banks ($e_j^*, e_i^* = 2, 1$)			8 banks ($e_j^*, e_i^* = 4, 2$)			8 banks ($e_j^*, e_i^* = 4, 2$)		
e_j	e_i	$A \cdot L$	e_j	e_i	$A \cdot L$	e_j	e_i	$A \cdot L$
1	1	0.74	1	1	2.24	1	1	2.74
2	1	0.30	2	1	2.09	2	1	0.87
2	2	0.36	2	2	1.48	2	2	0.49
4	2	0.26	4	2	0.87	4	2	0.32
4	4	0.24	4	4	0.76	4	4	0.23

A. Discussion

The above results confirm that the model introduced in Section IV can effectively capture the bank switching effect. In both the 4-bank and 8-bank cases, we can notice that unrolling benefits efficiency considerably up to a certain point due to reduced amounts of bank switching. Beyond those points, there are diminishing or null returns. Those points are actually the minimum unrolling factors needed to avoid bank switching completely, as correctly predicted by our model. For example, looking at Table I for the Gauss-Seidel and the Jacobi kernels with four banks, the model predicts that the configuration $e_j^* = 2$, $e_i^* = 2$ represents the minimum amount of unrolling needed in order not to have switching, which is confirmed by the experiments. A more aggressive unrolling, beyond $2, 2$, is still unaffected by bank switching, but lower unrolling factors ($1, 1$ and $2, 1$) cannot avoid it. Notice also that the Gauss-Seidel and Jacobi kernels exhibit the same behavior in terms of unrolling choices for avoiding bank switching since they both process the input array with a unitary stride. In contrast, the image resizing kernel processes the input data block-wise, and this partly mitigates bank switching. In fact, it can be noticed that good unrolling factors tend to be lower in this case.

As a side remark, we highlight that the unrolling factors were assumed not constrained by data dependencies in this study. First of all, this choice may have a limited impact for loops with *uniform dependencies*, where one can always extract $n-1$ levels of parallelism if the loop hierarchy is made of n loops [19]. For such loops, it is possible to transform the nest, possibly automatically [20], [15], and unroll all the loops apart from one (e.g., the outermost, in case we have parallelized the inner loops of the hierarchy). In general, unrolling in presence of data dependencies may result in an underutilized parallel datapath. However, advanced HLS tools can detect such a situation and avoid allocating unnecessary resources, or still take advantage of the replicated hardware by exploiting the parallelism of accessory operations (e.g. arithmetic operations for the generation of addresses). Therefore, even with data dependencies, loop unrolling in presence of multidimensional memory partitioning can still potentially enable improved area efficiency.

VI. Conclusion

Memory partitioning is of paramount importance when high-level synthesis is adopted in the context of architectures provided with multiple independent memory banks. This paper analyzed the consequences of multidimensional partitioning on area efficiency and proposed a technique for improving efficiency by means of loop unrolling. The experimental results show that, because of the positive impact on bank switching, unrolling in presence of multidimensional memory partitioning *can significantly improve* the area efficiency of the synthesized circuit, provided that the unrolling factors are carefully chosen. As a future work, we plan to extend the formalization of the bank switching effect, aiming for a quantitative formula that expresses the amount of switching depending on the lattice basis. Such a formula may be useful in order to evaluate and quickly compare alternative partitioning choices.

Acknowledgment

This work was financially supported by the University of Naples Federico II and Compagnia San Paolo in the framework of the STAR programme.

References

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

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

[3] M. Fingeroff, *High-Level Synthesis Blue Book*. Xlibris, Corp., 2010.

[4] A. Cilardo, E. Fusella, L. Gallo, and A. Mazzeo, "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.

[5] A. Cilardo, E. Fusella, L. Gallo, A. Mazzeo, and N. Mazzocca, "Automated design space exploration for FPGA-based heterogeneous interconnects," *Design Automation for Embedded Systems*, pp. 1–14, 2014.

[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] A. Cilardo and L. Gallo, "Improving multi-bank memory access parallelism with lattice-based partitioning," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 4, 2014.

[8] Xilinx Inc., *Vivado Design Suite User Guide High-Level Synthesis*, 2012.

[9] P. Li, Y. Wang, P. Zhang, G. Luo, T. Wang, and J. Cong, "Memory partitioning and scheduling co-optimization in behavioral synthesis," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '12. New York, NY, USA: ACM, 2012, pp. 488–495.

[10] Y. Wang, P. Zhang, X. Cheng, and J. Cong, "An integrated and automated memory optimization flow for FPGA behavioral synthesis." in *ASP-DAC*. IEEE, 2012, pp. 257–262.

[11] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong, "Memory partitioning for multidimensional arrays in high-level synthesis," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: ACM, 2013, pp. 12:1–12:8.

[12] 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.

[13] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 199–208.

[14] L. Gallo, A. Cilardo, D. Thomas, S. Bayliss, and G. Constantinides, "Area implications of memory partitioning for high-level synthesis on FPGAs," in *Field Programmable Logic and Applications (FPL), 2014 24rd International Conference on*. IEEE, 2014.

[15] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, "Iterative optimization in the polyhedral model: Part II, multidimensional time," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. Tucson, Arizona: ACM Press, June 2008, pp. 90–100.

[16] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT'13 IEEE Int'l Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, France, September 2004, pp. 7–16.

[17] P. Feautrier, "Some efficient solutions to the affine scheduling problem. i. one-dimensional time," *International Journal of Parallel Programming*, vol. 21, no. 5, pp. 313–347, 1992.

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

[19] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 4, pp. 452–471, Oct. 1991.

[20] M. Griebl and C. Lengauer, "The loop parallelizer loopo-announcement." in *LCPC*, ser. Lecture Notes in Computer Science, D. C. Sehr, U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua, Eds., vol. 1239. Springer, 1996, pp. 603–604.