HLS Support for Polymorphic Parallel Memories

L. Stornaiuolo*, M. Rabozzi*, D. Sciuto*, M. D. Santambrogio*, G. Stramondo+, C. Ciobanu+, A. L. Varbanescu+
*Politecnico di Milano, Milan, Italy; +Universiteit van Amsterdam, Amsterdam, Netherlands
{luca.stornaiuolo, marco.rabozzi, donatella.sciuto, marco.santambrogio}@polimi.it
{g.stramondo, c.b.ciobanu, a.l.varbanescu}@uva.nl

***Abstract*—The importance of High-Level Languages in abstracting machine language to enhance productivity has been proved in many sectors, and has recently encouraged the spread of reconfigurable hardware for general purpose computing. At the same time, Field Programmable Gate Arrays (FPGAs) become popular for data-intensive applications, because they promise customized hardware accelerators and achieve high-performance with low power consumption. However, taking advantage of parallel accesses to the local memories of FPGAs remains difficult, as it currently requires application re-engineering. A solution to this challenge is PolyMem, an easy-to-use parallel memory. In this work, we investigate the implementation, integration, and performance of PolyMem for HLS applications. To this end, we present a novel open-source implementation of PolyMem, optimized for the Xilinx Design Suite. We further demonstrate the use of PolyMem for three different case studies, implemented using both the Vivado workflow with a Virtex-7 VC707, and the SDx workflow with a Kintex Ultrascale 3 ADM-PCIE. Finally, we provide a thorough empirical analysis of these three cases studies in terms of latency, hardware resources, and productivity. Our results demonstrate that PolyMem delivers the expected performance, while enhancing productivity at the cost of a small increase in resources.**

I. Introduction

The success of High-Level Languages (HLLs) for non-traditional computing systems, like Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs), have accelerated the adoption of these platforms for general purpose computing. In particular, the main hardware vendors released tools and frameworks to support their products by allowing the design of optimized kernels using HLLs. This is the case, for example, for Xilinx, which allows using C++ or OpenCL within the Vivado Design Suite [1] to target FPGAs.

Moreover, FPGAs are increasingly used for data-intensive applications, because they enable users to create custom hardware accelerators, and achieve high-performance implementations with low power consumption. However, one aspect still lagging behind is the efficient use of BRAMs, the FPGA distributed, high-bandwidth, on-chip memories [2]. BRAMs can provide memory-system parallelism, but their use remains challenging due to the many different ways in which data can be partitioned in order to achieve efficient parallel data accesses. Changing data access patterns on the application side is the current state-of-the-art approach, which does parallelize operations and reduces the kernel execution time, but also requires extensive modification of the application code.

To address the challenges related to the design and practical use of parallel memory systems for FPGA-based applications, PolyMem, a Polymorphic Parallel Memory, was proposed [3]. PolyMem is envisioned as a high-bandwidth, two-dimensional (2D) memory *used to cache performance-critical data on the FPGA chip*, making use of the existing distributed memory banks (the BRAMs). PolyMem is inspired by the Polymorphic Register File (PRF) [4], a runtime customizable register file for Single Instruction, Multiple Data (SIMD) co-processors. PolyMem is tailored for FPGA accelerators which require high bandwidth, even if they do not implement full-blown SIMD co-processors on the reconfigurable fabric.

The first hardware implementation of the Polymorphic Register File was designed in System Verilog [5]. MAX-PolyMem is the first prototype of PolyMem written entirely in MaxJ, and targeted at Maxeler DFEs [6], [7]. Our new HLS PolyMemis an alternative HLL solution, proven to be easily integrated with the Xilinx toolchains.

Figure 1 depicts the envisioned system architecture. The FPGA board (with a high-capacity DRAM memory), is connected to the host CPU through a PCI Express link. PolyMem acts as a high-bandwidth, 2D parallel software cache, able to feed an on-chip application kernel with multiple data elements every clock cycle. The focus of this work is to provide an efficient implementation of PolyMem in Vivado HLS, and employ it to maximize memory-accesses parallelism by exploiting BRAMs; we empirically demonstrate the gains we get from PolyMem by comparison against the partitioning of BRAMs, as provided by Xilinx tools, for three case-studies.

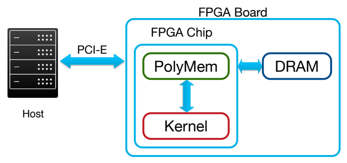

Fig. 1. System organization using PolyMem as a parallel cache.

Specifically, the main contributions of this paper are:

1) A novel, open-source implementation[1] of PolyMem for Vivado HLS, that allows its integration within the Xilinx Hardware-Software Co-Design Workflow;
2) Optimizations of the previously proposed PolyMem interface by adding masked methods to avoid overwrites and reduce latency;
3) Comparisons in terms of performance, resource-utilization, and productivity, of HLS PolyMem against standard memory partitioning techniques for three case-studies.

978-1-5386-4756-1/18/$31.00©2018 European Union

[1]https://github.com/storna/hls_prf

II. Background

A. The PRF and PolyMem

A PRF is a parameterizable register file, which can be logically reorganized by the programmer or a runtime system to support multiple register dimensions and sizes simultaneously [4]. The simultaneous support for multiple conflict-free access patterns, called *multiview*, is crucial, providing flexibility and improved performance for target applications. The *polymorphism* aspect refers to the support for adjusting the sizes and shapes of the registers at runtime. Table I presents the PRF *multiview* schemes (ReRo, ReCo, RoCo and ReTr), each supporting a combination of at least two conflict-free access patterns. A scheme is used to store data within the memory banks of the PRF, such that it allows different parallel *access types*. The different *access types* refer to the actual data elements that can be accessed in parallel.

TABLE I
The PRF memory access schemes

PRF Schemes	Available Access Types
ReO	Rectangle
ReRo	Rectangle, Row, Main/Secondary Diagonals
ReCo	Rectangle, Column, Main/Secondary Diagonals
RoCo	Row, Column, Rectangle
ReTr	Rectangle, Transposed Rectangle

PolyMem reuses the PRF conflict-free parallel storage techniques and patterns, as well as the polymorphism idea. Figure 2 illustrates the set of access patterns supported by the PRF and, ultimately, by PolyMem. In this example, a 2D logical address space of 8×9 elements contains 10 memory Regions (R), each with different size and location: matrix, transposed matrix, row, column, main and secondary diagonals. Assuming a hardware implementation with eight memory banks, each of these regions can be read using one (R1-R9) or several (R0) parallel accesses.

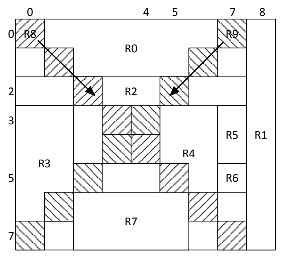

Fig. 2. PolyMem supported access patterns

By design, the PRF optimizes the memory throughput for a set of predefined memory access patterns. For PolyMem, we consider $p \times q$ memory modules and the five parallel access schemes presented in Table I. Each scheme supports dense, conflict-free access to $p \cdot q$ elements. When implemented in reconfigurable technology, PolyMem allows application-driven customization: its capacity, number of read/write ports, and the number of lanes to best support the application needs.

The block diagram in Figure 3 shows, at high level, the PEF architecture. The multi-bank memory is composed of a bi-dimensional matrix containing $p \times q$ memory modules. This enables parallel access to $p \cdot q$ elements in one memory

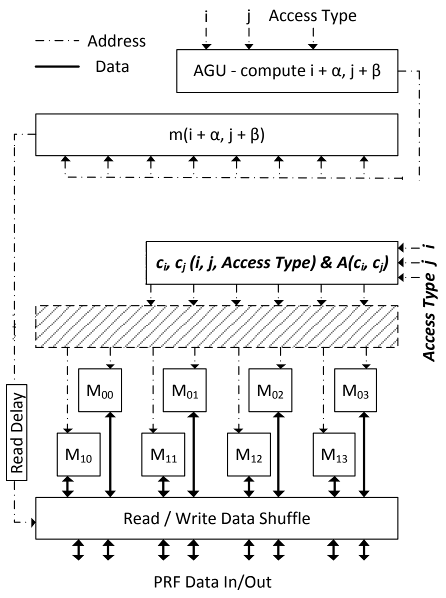

Fig. 3. Block diagram of the PRF [4]. The inputs are the matrix indexes (i, j) pointing to the first cell of the block of data the user wants to read/write in parallel, and the AccessType to compute the other addresses to point the right PRF banks of memory.

operation. The inputs of the PRF are shown at the top of the diagram. AccessType represents the parallel access pattern. The indexes (i, j) are the top-left coordinate of the parallel accesses. The list of elements to access is generated by the AGU module and is sent to the A module and to the m module. The A module generates one in-memory address for each memory bank in the PRF; the m module, applies the mapping function relative to the implemented scheme and computes for each accessed element the respective memory bank where it is stored. The Data Shuffle block reorders the addresses, generated by the m module, to the respective memory banks, then reorder the Data In/Out ensuring that the user of the PRF obtains the accessed data in their original order.

B. Matrix storage in a parallel memory

Figure 4 compares two ways for a 6×6 matrix to be mapped in BRAMs to enable parallel accesses. Thus, the default Vivado HLS partitioning techniques with a factor of 3 is compared against a PolyMem with 3 memory banks, organized exploiting the PolyMem RoCo scheme. The memory banks, in this case, are organized in a 1×3 structure, allowing parallel access to rows and columns of three, eventually unaligned, elements. The left side of the Figure shows an example of a matrix that the user wishes to store on partitioned BRAMs to achieve hardware parallelism in data reads/writes. The right side illustrates the techniques used to partition the matrix. Taking two random, unaligned, parallel accesses of 3 elements and using a RoCo scheme, starting respectively from the cells contain elements 8 and 23, it is possible to see that each element of each access type is mapped within PolyMem on a different memory bank. Hence, with one memory operation performed in parallel on each different memory bank, it is possible to read/write 3 elements in parallel. This small-scale example is included for visualization purposes only. Real-applications are like to use more memory banks, allowing parallel accesses to larger data blocks.

III. Implementation details

This section describes the main components of our Poly-Mem implementation for Vivado HLS. The goal of integrating PolyMemin the Xilinx workflow is to provide users with an easy-to-use solution to exploit parallelism when accessing data stored on the on-chip memory with different patterns.

Our Vivado HLS PolyMem implementation exploits one of the five schemes, the RoCo, to store on the BRAMs of the FPGA the data required to perform the application operations. Compared to the default Vivado memory partitioning techniques, which allow hardware parallelism in one dimension without consuming too many hardware resources, a Poly-Mem configured with the RoCo scheme can manage two types of access patterns simultaneously.

We implemented a template-based class prf that exploits loop unrolling to parallelize memory accesses. When the HLS PolyMem is instantiated within the user application code, it is possible to specify PRF_DATA_T, i.e., the type of data to be stored, the $(p \times q)$ number of internal banks of memory (which also represents the level of parallelism), the $(N \times M)$ dimension of the matrix to be stored (also used to compute the depth of each bank of data), and the $scheme$ to organize data within the different banks of memory.

In Listing 1 the interfaces of methods that allow accesses to data stored within PolyMem are presented. Simple **read** and **write** methods use the m and A functions to compute, respectively, the address and the depth of the bank of memory in which the required data is stored or needs to be saved. On the other hand, the **read_block** and the **write_block** exploit optimized versions of m and A to read/write $(q \cdot p)$ elements in parallel, while limiting the hardware resources used to reorder data. Finally, we optimized the memory access operations by implementing a **write_block_masked** method to specify which data in the block has to be overwritten within PolyMem. As an example, this method is useful when PolyMem supports a wide parallel access (e.g., 8 elements), but the user has less data to be stored (e.g., 5 elements), and wants avoid overwriting existing data (e.g., the remaining 3 elements).

Listing 1. List of the methods interfaces to allow user read/write data by used sequential or parallel accesses

```
PRF_DATA_T read(int i, int j);
void write(PRF_DATA_T data, int i, int j);
void read_block(int i, int j, PRF_DATA_T out[p * q],
                int PRF_ACCESS_TYPE);
void write_block(PRF_DATA_T in[p * q], int i, int j,
                 int PRF_ACCESS_TYPE);
void write_block_masked(PRF_DATA_T in[p * q],
                        ap_uint<p * q> mask,
                        int i, int j,
                        int PRF_ACCESS_TYPE);
```

IV. Experimental evaluation

In this section, we propose three applications (i.e., matrix multiplication, Markov chain, and LU decomposition) that exploit our HLS PolyMem to parallelize accesses to matrix data by using the RoCo scheme. Each application demonstrates different features of our HLS PolyMem. In the matrix multiplication case-study, we show how our approach outperforms implementations that use the default partitioning of Vivado HLS. For the Markov Chain application, we show how HLS PolyMem enables performance gain with minimal changes to the original software code. Finally, we present the use of the masked methods in the LU decomposition implementation.

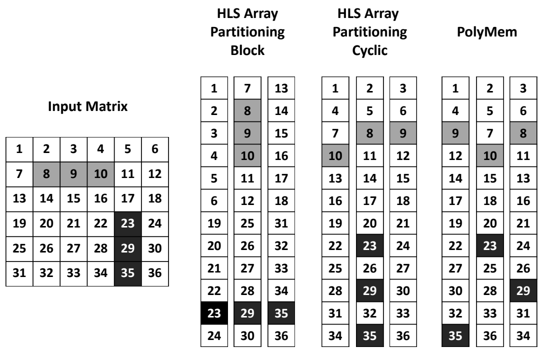

Fig. 4. Comparison between different partitioning techniques offered by Vivado HLS (factor = 3) and the RoCo scheme of PolyMem, with 3 memory banks, for data stored in a 6x6 matrix. PolyMem allows 3 parallel data reads/writes, from the rows and the columns of the original matrix. Unaligned blocks are also supported.

A. Matrix multiplication

In this case study, we analyze the multiplication between two square matrices, B and C, of size DIM, that are stored by using either the default HLS array partitioning techniques or the HLS PolyMem implementation. Since the multiplication $B \times C$ is performed by accessing the rows of B and multiply-accumulating the data with the columns of C, it is convenient, when using HLS default partitioning, to partition B on the second dimension and C on the first one. Indeed, this allows to achieve parallel accesses to the rows of B and columns of C in the innermost loop of the computation. On the other hand, for the HLS PolyMem implementation, we store both B and C in the HLS PolyMem , configured with a RoCo scheme, because it allows parallel accesses to both rows and columns.

Listings 2 and 3 show the declaration of the matrices and their partitioning using the HLS default partitioning and the HLS PolyMem, respectively. In both listings, a parallel factor of 16 has been used. The B and C HLS PolyMem instances are initialized with $p = 4$ and $q = 4$, which results in partitioning the data onto 16 memory banks.

Listing 2. Declaration and partitioning of matrices to parallelize accesses to rows (dim=2) of B and to columns (dim=1) of C with a parallel factor of 16.

```
float B[DIM][DIM];
#pragma HLS array_partition variable=B
                block factor=16 dim=2
float C[DIM][DIM];
#pragma HLS array_partition variable=C
                  block factor=16 dim=1
```

Listing 3. Declaration of the matrices stored by using the HLS PolyMem with the RoCo scheme with a parallel factor of $4 \cdot 4 = 16$.

```
#include "hls_prf.h"
hls::prf<float, 4, 4, DIM, DIM, SCHEME_RoCo> B;
hls::prf<float, 4, 4, DIM, DIM, SCHEME_RoCo> C;
```

Listings 4 and 5 show the matrix multiplication code when using the HLS default partitioning and the HLS PolyMem, respectively.

Listing 4. Matrix multiplication code that leverages default HLS partitioning to perform parallel accesses.

```
// B*C matrix multiplication
for (int i = 0; i < DIM; ++i)
  for (int j = 0; j < DIM; ++j) {
#pragma HLS PIPELINE II=1
    float sum = 0;
    for (int k = 0; k < DIM; ++k)
      sum += B[i][k] * C[k][j];
    OUT[i][j] = sum;
  }
```

Listing 5. Matrix multiplication code that exploits the HLS PolyMem with RoCo scheme to perform parallel accesses.

```
// B*C matrix multiplication
for (int i = 0; i < DIM; ++i)
  for (int j = 0; j < DIM; ++j) {
#pragma HLS PIPELINE II=1
    float sum = 0;
    for (int k = 0; k < DIM; k += 16) {
      B.read_block(i, k, temp_row, ACCESS_Ro);
      C.read_block(k, j, temp_col, ACCESS_Co);
      for (int t = 0; t < 16; t++)
        sum += temp_row[t] * temp_col[t];
    }
    OUT[i][j] = sum;
  }
```

Even though both approaches achieve the goal of computing the matrix multiplication by accessing 16 matrix elements in parallel, the HLS PolyMem solution provides more flexibility when additional data access patterns are required, which is often the case for larger kernels. In order to highlight this aspect, we also consider a kernel function in which both the $B \times C$ and the $C \times B$ products need to be computed.

Table II reports the latency and resources utilization estimated by Vivado HLS when computing the matrix multiplication $B \times C$ (rows 1,2), and when computing $B \times C$ followed by $C \times B$ (rows 3,4 and 5,6) for the two approaches. By using the default Vivado HLS partitioning techniques, the second multiplication $B \times C$ cannot be computed efficiently due to the way in which the matrix data is partitioned into the memory banks, as described in Section II. Indeed, C can only be accessed in parallel by rows and B by columns. On the other hand, the implementation based on HLS PolyMem is also capable of performing the matrix product $C \times B$ efficiently. This is also reflected in the estimated latency reported in Table II, which is the same for both products.

It is also worth noting that for matrix size of 32, the two approaches have similar resource consumption, while for matrices with larger dimensions and a parallel factor of 16, the HLS PolyMem has a resource consumption overhead in terms of FF and LUT of at most 8.5% compared to the HLS default partitioning schemes. Finally, in order to empirically validate the designs, we implemented the kernel module performing both $B \times C$ and $C \times B$ with matrix size of 96 and a parallel factor of 16 on a Xilinx Virtex-7 VC707 with a target frequency of 100MHz. The HLS PolyMem achieved a read and write throughput of 0.4 GB/s and a speedup of 5x compared to the implementation based on default HLS memory partitioning.

TABLE II
LATENCY AND HARDWARE RESOURCES FOR MATRIX MULTIPLICATION WITH DIFFERENT MEMORY CONFIGURATIONS AND MATRIX DIMENSIONS

Memory	Matrix size	Parallel factor	Latency		Hardware resources			
			$B \times C$	$C \times B$	BRAM	DSP	FF	LUT
HLS	32	4	4227	n.a.	18	40	6162	6485
PolyMem	32	$2 \cdot 2$	4227	n.a.	18	40	6153	6018
HLS	32	4	4227	16503	18	40	7444	9197
PolyMem	32	$2 \cdot 2$	4227	4227	18	40	7367	7364
HLS	96	16	28033	442722	96	164	28554	40474
PolyMem	96	$4 \cdot 4$	28033	28033	96	160	30969	43636

B. Markov Chain and the Matrix power operation

A Markov Chain is a stochastic model used to describe real-world processes. Some of the most relevant applications are found in queuing theory and study of population growths [8], while they are also used in stochastic simulation methods such as Gibbs sampling [9] and Markov Chain Monte Carlo [10]. Moreover, Page Rank [11], an algorithm used to rank websites by search engines, leverages a time-continuous variant of this model. A Markov Chain can also describe a system composed of multiple discrete states, where the probability of being in a state depends only on the previous state of the system. A Markov Transition Matrix A, which is a variant of an adjacency matrix, can be used to represent a Markov Chain. In this matrix, each row contains the probability to move from the current state to any other state of the system. More specifically, given two states i and j, the probability to transition from i to j is $a_{i,j}$, where $a_{i,j}$ is the element at row i and column j of the transition matrix A.

Computing the h-th power of the Markov Transition Matrix is a way to determine what is the probability to transition from an initial state to a final state in h steps. Furthermore, when the number of steps h tends to infinity, the result of A^h can be used to recover the stationary distribution of the Markov Chain, if it exists. From a computational perspective, an approximate value for the result of $lim_{x \to \infty} A^x$ is obtained for large enough values of x. In our implementation, matrix A is stored in a HLS PolyMem, so that both rows and columns can be accessed in parallel, then, we compute A^2 and save the result into a support matrix A_temp, partitioned on the second dimension. After A^2 is computed, we can easily compute A^{2^h} by copying back results to the HLS PolyMem and iterating the overall computation h times. Implementing the same algorithm by using the HLS partitioning techniques, as presented in the previous case study, results in poor exploitation of the available

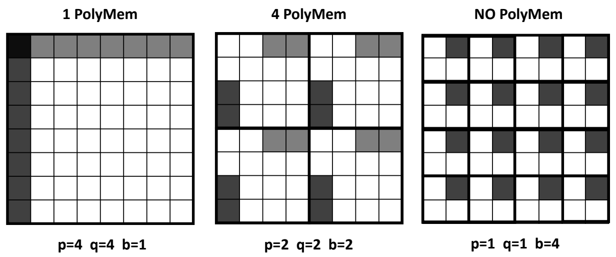

Fig. 5. Comparison between different partitioning of the input matrix in a grid of b^2 components implemented by PolyMem with a level of parallelism of $p \times q$. When both p and q are set to 1, it is possible to remove the HLS PolyMem logic.

TABLE III
LATENCY, HARDWARE RESOURCES AND LINES OF CODE, FOR 8 ITERATIONS OF THE MATRIX POWER OPERATION WITH DIFFERENT MEMORY CONFIGURATIONS AND A MATRIX SIZE OF 384

Memory	p	q	b	Latency	Hardware resources				LOC
					BRAM	DSP	FF	LUT	
PolyMem	2	2	1	1,557,835,871	1,036	14	9,936	11,071	98
PolyMem	2	4	1	840,333,407	1,044	17	19,678	28,855	98
PolyMem	4	4	1	488,632,423	1,060	31	36,138	53,621	98
multi PolyMem	1	1	2	758,085,955	1,036	14	6,967	5,572	188
multi PolyMem	1	2	2	394,149,976	1,044	28	14,709	12,934	188
multi PolyMem	2	2	2	214,032,480	1,060	45	24,845	22,418	188
NO PolyMem	1	1	4	101,848,419	1,124	76	32,852	13,706	188

parallelism, or in duplicated data, since A needs to be accessed both by rows and columns.

The HLS PolyMem enables paralell accesses to matrix A for both rows and columns, but adds to the design an overhead in terms of hardware resources and complexity of the logic to shuffle data within the right memory banks. The resources overhead has a quadratic growth with respect to the number $p \cdot q$ of parallel memories used to store data [4].

A possible solution to this problem, is to reduce the dimension of PolyMem by dividing the A input matrix and store the values in a grid of multiple PolyMems. If A has dimension $DIM \times DIM$, it is possible to organize the on-chip memory to store data in a grid of $b \times b$ squared blocks each having size $\frac{DIM}{b} \times \frac{DIM}{b}$. In order to preserve the same level of parallelism, we can re-engineer the original computation to work in parallel on the data stored in each memory within the grid. Instead of computing a single vectorized row-column product, it is possible to perform the computation on multiple row-column products in parallel and reduce the final results.

Figure 5 shows how the input matrix is divided in multiple memories according to the choice of the parameters p, q and b. Moreover, the figure also shows which is the data accessed concurrently at each step of the computation. As an example, for the case $p = q = b = 2$ there are 4 row-column products performed in parallel (b^2) and for each of them 4 values are processed in parallel ($p \cdot q$).

It is important to notice that when $p = q = 1$ the PolyMems reduce to memories in which a single element is accessed in parallel. In this case, each PolyMem can be removed and substituted by a single memory bank.

In Table III we report the latency and the resource utilization estimated by Vivado HLS together with the number of lines of code (LOC) for different configurations of the parameters p, q and b on 8 iterations of the power operation for a 384x384 matrix. As can be seen, by re-engineering the code and the access patterns ($b > 1$), it is possible to achieve a smaller overall latency. However, this comes at the cost of a more convoluted code which is approximately twice the lines of code of the original version. On the contrary, by using a single PolyMem ($b = 1$) we can still obtain higher performance than using the default HLS array partitioning techniques, with a much smaller and simpler code base. Indeed, PolyMem allows to reduce the time to develop an optimized FPGA-based implementation of the algorithm with minor modifications to the original software code. Thanks to HLS PolyMem we raise the level of abstraction of parallel memory accesses, thus enhancing the overall design experience and productivity.

Finally, to validate the flexibility the HLS PolyMem library, we implemented and tested the application by using Xilinx SDx tool, that enables OpenCL integration and automatically generates the PCIe drivers for communication. We synthesized a design for a matrix size of 256 and parameters $p = q = b = 2$ at 200MHz, and we benchmarked its performance on the Xilinx Kintex Ultrascale ADM-PCIE-KU3 platform, obtaining a read and write throughput of 1.6 GB/s.

C. LU decomposition

The last case study we present is the LU decomposition algorithm. This algorithm allows to decompose an input matrix A into a product of a lower triangular matrix L and an upper triangular matrix U:

$$A = LU$$

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ l_{10} & 1 & 0 \\ l_{20} & l_{21} & 1 \end{bmatrix} \begin{bmatrix} u_{00} & u_{01} & u_{02} \\ 0 & u_{11} & u_{12} \\ 0 & 0 & u_{22} \end{bmatrix}$$

This factorization is used in many applications, such as solving linear equations or compute the inverse of a matrix. Furthermore, such application has been proved to be suitable for hardware acceleration [12]–[14].

Listing 6 shows the LU decomposition algorithm. Matrix A is given as input, and matrices L and U are computed while matrix A is zeroed.

Listing 6. LU decomposition algorithm

```
for(k=0; k<DIM; k++){
  for(i=k; i<DIM; i++){
    L[i][k] = A[i][k] / A[k][k];
    U[k][i] = A[k][i];
  }
  for(i=k; i<DIM; i++)
    for(j=k; j<DIM; j++)
      A[i][j] = A[i][j] - L[i][k] * U[k][j];
}
```

Analyzing the loops, it is possible to see that the algorithm works on successive sub-matrices, identified by the iterator of the outermost loop k. In the first nested loop, matrix A is accessed both column-wise and row-wise. The results of those statements are respectively stored column-wise in matrix L and row-wise in matrix U. Finally, the second nested loop updates matrix A before starting the new iteration.

This brief analysis shows that this implementation of the LU decomposition algorithm uses interleaved row-wise and column-wise accesses to the same matrix; moreover, due to the offset introduced by iterator k, those accesses could be unaligned. Even in this case, HLS PolyMem represents a valid solution to parallelize the read and write operations.

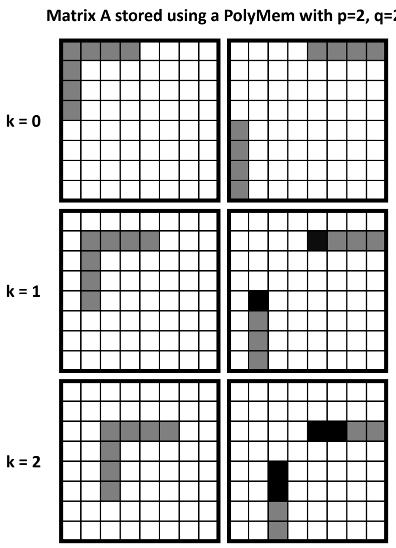

Fig. 6. Three iterations of loop k (rows of the figure) and the iterations of the nested loop (columns of the figure) to update in parallel blocks of values of the HLS PolyMem where input and output are stored. The writes are performed by using the **write_block_masked** method: the black cells of the matrices correspond to the 0 values of the passed mask and they are not written in the HLS PolyMem memory.

Unaligned accesses to the matrix represent an interesting case study and they are supported by the HLS PolyMem implementation. Furthermore, by exploiting the fact that the values of the matrix A are iteratively zeroed and the structure of the L and U matrices, we can store the entire computation on a single HLS PolyMem memory. The final L and U matrices are stored as follows:

$$\begin{bmatrix} u_{00} & u_{01} & u_{02} \\ l_{10} & u_{11} & u_{12} \\ l_{20} & l_{21} & u_{22} \end{bmatrix}$$

Depending on the value of the iterator k, the amount of data to be processed might not be a multiple of the parallel factor being used. For this reason, special care must be taken when dealing with the last block being processed as shown in figure 6. To avoid to overflow the matrix dimensions, the last block is computed out of the loop, and always starts at an offset of $DIM - (p \cdot q)$. Then, an appropriate write mask is applied to ensure that only the needed data is written to the HLS PolyMem memory. In order to enable this solution, we implemented the method **write_block_masked** that allows to pass a mask of bits that represent the positions of values within the block that need to be written. Since this method adds some logic to solve the mask, we use it only while updating the last block, out of the nested loop. Thanks to the introduction of this class of methods, we simplify the adoption of the HLS PolyMem for a broader set of applications.

V. Conclusions

In this paper, we presented a C++ implementation of PolyMem optimized for Vivado HLS, ready-to-use as a library for applications requiring parallel memories. Our implementation exposes an easy-to-use interface to enhance design productivity for FGPA-based applications. Furthermore, we extended the original PolyMem implementation to support masked on-chip parallel accesses.

We proved the flexibility of the library among the Xilinx Design Tools, by implementing the kernels for *both* the Vivado workflow with a Virtex-7 VC707 and the SDx workflow with a Kintex Ultrascale 3 ADM-PCIE. Our empirical analysis of our library on three case studies (Matrix multiplication, Markov Chains, and LU decomposition) demonstrated competitive results in terms of latency, low code complexity, but also a small overhead in terms of hardware resource utilization.

Our future work focuses on (1) including support for additional PolyMem schemes optimized for Vivado HLS, (2) designing an automatic framework to analyze the user application code and suggest how to improve its performance with HLS PolyMem, and (3) improving the HLS PolyMem shuffle module by exploiting a Butterfly Network [15] for the memory banks connections.

References

[1] Xilinx, “Vivado high-level synthesis.” [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[2] M. Weinhardt and W. Luk, “Memory access optimisation for reconfigurable systems,” *IEE Proceedings-Computers and Digital Techniques*, vol. 148, no. 3, pp. 105–112, 2001.

[3] C. B. Ciobanu *et al.*, “MAX-PolyMem: High-Bandwidth Polymorphic Parallel Memories for DFEs,” in *RAW2018 (to appear)*, pp. 1–8.

[4] C. Ciobanu, “Customizable Register Files for Multidimensional SIMD Architectures,” Ph.D. dissertation, TUDelft, The Netherlands, 2013.

[5] C. Ciobanu *et al.*, “Scalability Study of Polymorphic Register Files,” in *Proc. of DSD*, 2012, pp. 803–808.

[6] C. B. Ciobanu *et al.*, “Max-polymem: High-bandwidth polymorphic parallel memories for dfes,” in *IEEE IPDPSW - RAW’18*, May 2018, pp. 107–114.

[7] ——, “EXTRA: An Open Platform for Reconfigurable Architectures,” in *Proceedings of SAMOS XVIII (to appear)*, July 2018, pp. 1–10.

[8] J. J. Arsanjani *et al.*, “Integration of logistic regression, markov chain and cellular automata models to simulate urban expansion,” *International Journal of Applied Earth Observation and Geoinformation*, vol. 21, pp. 265–275, 2013.

[9] A. F. Smith and G. O. Roberts, “Bayesian computation via the gibbs sampler and related markov chain monte carlo methods,” *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 3–23, 1993.

[10] W. R. Gilks *et al.*, *Markov chain Monte Carlo in practice*. CRC press, 1995.

[11] S. D. Kamvar *et al.*, “Extrapolation methods for accelerating pagerank computations,” in *Proceedings of the 12th international conference on World Wide Web*. ACM, 2003, pp. 261–270.

[12] G. Govindu *et al.*, “A high-performance and energy-efficient architecture for floating-point based lu decomposition on fpgas,” in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 149.

[13] V. Daga *et al.*, “Efficient floating-point based block lu decomposition on fpgas,” in *International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas*, 2004, pp. 21–24.

[14] M. K. Jaiswal and N. Chandrachoodan, “Fpga-based high-performance and scalable block lu decomposition architecture,” *IEEE Transactions on Computers*, vol. 61, no. 1, pp. 60–72, 2012.

[15] A. Avior *et al.*, “A tight layout of the butterfly network,” *Theory of Computing Systems*, vol. 31, no. 4, pp. 475–488, 1998.