Building High-Performance, Easy-to-Use Polymorphic Parallel Memories with HLS

L. Stornaiuolo[1(✉)], M. Rabozzi[1(✉)], M. D. Santambrogio[1(✉)], D. Sciuto[1(✉)], C. B. Ciobanu[2,3(✉)], G. Stramondo[3(✉)], and A. L. Varbanescu[3(✉)]

[1] Politecnico di Milano, Milan, Italy
{luca.stornaiuolo,marco.rabozzi,marco.santambrogio, donatella.sciuto}@polimi.it
[2] Technische Universiteit Delft, Delft, The Netherlands
c.b.ciobanu@tudelft.nl
[3] Universiteit van Amsterdam, Amsterdam, The Netherlands
{c.b.ciobanu,g.stramondo,a.l.varbanescu}@uva.nl

Abstract. With the increased interest in energy efficiency, a lot of application domains experiment with Field Programmable Gate Arrays (FPGAs), which promise customized hardware accelerators with high-performance and low power consumption. These experiments possible due to the development of High-Level Languages (HLLs) for FPGAs, which permit non-experts in hardware design languages (HDLs) to program reconfigurable hardware for general purpose computing.

However, some of the expert knowledge remains difficult to integrate in HLLs, eventually leading to performance loss for HLL-based applications. One example of such a missing feature is the efficient exploitation of the local memories on FPGAs. A solution to address this challenge is PolyMem, an easy-to-use polymorphic parallel memory that uses BRAMs. In this work, we present HLS-PolyMem, the first complete implementation and in-depth evaluation of PolyMem optimized for the Xilinx Design Suite. Our evaluation demonstrates that HLS-PolyMem is a viable alternative to HLS memory partitioning, the current approach for memory parallelism in Vivado HLS. Specifically, we show that PolyMem offers the same performance as HLS partitioning for simple access patterns, and outperforms partitioning as much as 13x when combining multiple access patterns for the same data structure. We further demonstrate the use of PolyMem for two different case studies, highlighting the superior capabilities of HLS-PolyMem in terms of performance, resource utilization, flexibility, and usability.

Based on all the evidence provided in this work, we conclude that HLS-PolyMem enables the efficient use of BRAMs as parallel memories, without compromising the HLS level or the achievable performance.

Keywords: Polymorphic Parallel Memory · High-Level Synthesis · FPGA

Published by Springer Nature Switzerland AG 2019
N. Bombieri et al. (Eds.): VLSI-SoC 2018, IFIP AICT 561, pp. 53–78, 2019.
https://doi.org/10.1007/978-3-030-23425-6_4

1 Introduction

The success of High-Level Languages (HLLs) for non-traditional computing systems, like Field Programmable Gate Arrays (FPGAs), has 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. Combining this trend with the fast-paced development of HLLs, more and more users and applications aim to experiment with FPGA accelerators.

In the effort of providing HLL tools for FPGA design, some of the features used by hardware design experts are difficult to transparently integrate. One such feature 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. Typical HLL solutions allow easy-to-use mechanisms for basic data partitioning. These mechanisms work well for simple data access patterns, but can significantly limit the patterns for which parallelism (and thus, increased performance) can be achieved. Changing data access patterns on the application side is the current state-of-the-art approach: by matching the application patterns with the simplistic partitioning models of the HLL, one can achieve parallel operations and reduce the kernel execution time. However, if at all possible, this transformation also requires extensive modification of the application code, which is cumbersome and error-prone to the point of canceling the productivity benefits of HLLs.

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 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 suitable for FPGA accelerators requiring 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 Data Flow Engines (DFEs) [3,6]. Our new HLS PolyMem is an alternative HLL solution, proven to be easily integrated with the Xilinx toolchains. The current work is an extension of our previous implementation presented in [7].

Figure 1 depicts the architecture of a system using (HLS-)PolyMem. 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 par-

allel 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.

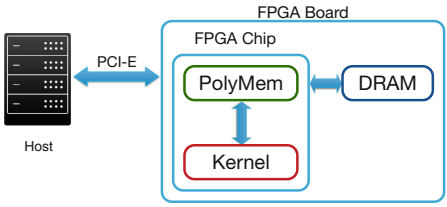


Fig. 1. System organization using PolyMem as a parallel cache.

In this work, we provide empirical evidence that HLS-PolyMem provides significant improvements in terms of both performance and usability when compared with the current memory partitioning approach present in Vivado HLS. To this end, we highlight the following novel aspects of this work:

- We provide a new, complete, open-source implementation [45] of PolyMem for Vivado HLS. This new implementation contains all the memory access schemes supported by the original PRF, as well as its multiview feature. Our implementation can be easily integrated within the Xilinx Hardware-Software Co-Design Workflow;
- We present a basic, high-level PolyMem interface (i.e., a rudimentary API for using PolyMem). The API includes basic parallel read and write operations. Furthermore, our API was further extended to support masked writes, avoiding overwrites and further reduce latency. For example, when PolyMem supports wide parallel access (e.g., 8 elements), but the user requires less data to be stored (e.g., 5 elements), and wants to avoid overwriting existing data (e.g., the remaining 3 elements). We demonstrate the use of the API in all the applications discussed in this paper (synthetic and real-life examples alike);
- We design and prototype a synthetic, parameterized microbenchmarking framework to thoroughly evaluate the performance of HLS-PolyMem. Our microbenchmarking strategy is based on chains of operations using one or several parallel access patterns, thus stressing both the performance and flexibility of the proposed parallel memory. The framework is extended to enable the comparison against existing HLS memory partitioning schemes. Finally, we show how to use these microbenchmarks to provide an extensive analysis of HLS-PolyMem's performance.

- We design, implement, and analyze in detail two case-study applications which demonstrate the ability of our HLS-PolyMem to cope with real applications and data, multiple memory access patterns, and tiling. Our experiments for these case-studies focus on performance, resource-utilization, and productivity, and contrast our HLS PolyMem with standard memory partitioning techniques.

Our results, collected for both synthetic and real-life case-studies, thoroughly demonstrate that HLS PolyMem outperforms traditional HLS partitioning schemes in performance and usability. We therefore conclude that our HLS-PolyMem is the first approach that enables HLS programmers to use BRAMs to construct flexible, multiview parallel memories, which can still be easily embedded in the traditional HLS modus operandi.

The remainder of this paper is organized as follows. Section 2 provides an introduction to parallel memories, and discusses the two alternative implementations presented in this work: the PRF-inspired PolyMem and the HLS partitioning schemes. Section 3 presents the HLS PolyMem class for Vivado, together with the proposed optimizations. In Sect. 4 we present our microbenchmarking framework, as well as the our in-depth evaluation using this synthetic workload. Section 5 describes our experience with designing, implementing, and evaluating the two case studies. Section 6 highlights relevant related work and, finally, our conclusion and future work directions are discussed in Sect. 7.

2 Parallel Memories: Challenges and Solutions

2.1 Parallel Memories

Definition 1 (Parallel Memory). *A Parallel Memory (PM) is a memory that enables the access to multiple data elements in parallel.*

A parallel memory can be realized by combining a set of independent memories, referred to as *banks* or *lanes*. The *width of the parallel memory*, i.e., the number of banks used in the implementation, represents the maximum number of elements that can be read in parallel. The *capacity* of the parallel memory refers to the amount of data that it can store. A specific element contained in a PM is identified by its *location*, a combination of a *memory module identifier* (to specify which one of the sequential memories hosts the data) and an *in-memory address* (to specify where within that memory the element is stored).

Depending on how the information is stored and/or retrieved from the memory, we distinguish three types of parallel memories: redundant, non-redundant, and hybrid.

Redundant PMs. The simplest implementation of a PM is a fully redundant one, where all M sequential memory blocks contain fully replicated information. The benefit of such a memory is that it allows an application to access any combination of M data elements in parallel. However, such a solution has two

major drawbacks: first, the total capacity of a redundant PM is M times lower than the combined capacities of all its banks, and, second, parallel writes are very expensive in order to maintain information consistency.

To use such a memory, the application requires minimal changes, and the architecture is relatively simple to manage.

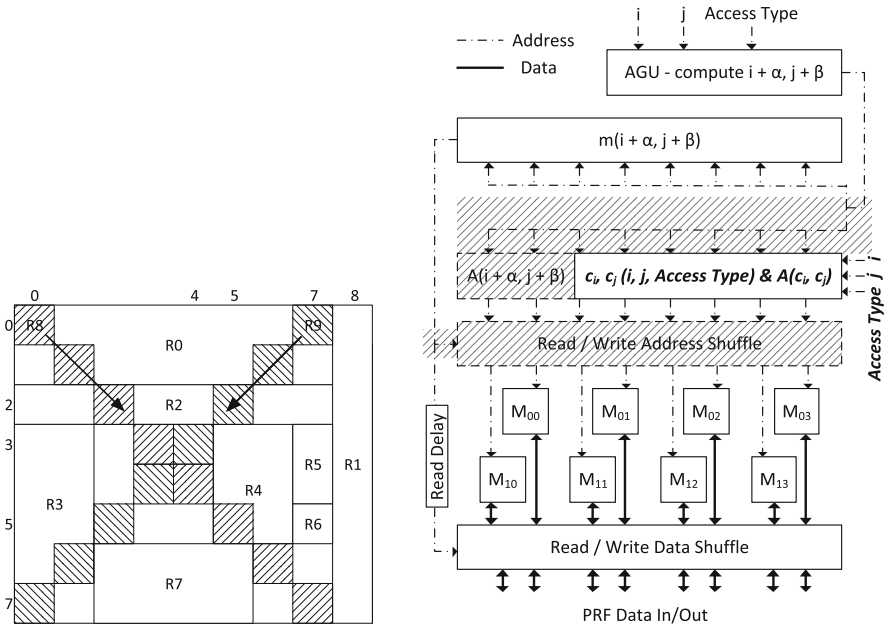
Non-redundant PMs. Non-redundant PMs completely avoid data duplication: each data item is stored in only one of the M banks. The one-to-one mapping between the coordinate of an element in the application space and a memory location is part of the memory configuration. These memories can use the full capacity of all the memory resources available, and data consistency is guaranteed by avoiding data replication, making parallel writes feasible as well. The main drawback of non-redundant parallel memories is that they require additional logic - compared to redundant memories - to perform the mapping, and they restrict the possible parallel accesses: if two elements are stored in the same bank, they cannot be accessed in parallel.

There are two major approaches used to implement non-redundant PM: (1) use a set of predefined mapping functions that enable parallel accesses in a set of predefined shapes [4,8–10], or, (2) derive an application-specific mapping function [11,12]. For the first approach, the application requires additional analysis and potential changes, while the architecture is relatively fixed. For the second approach, however, a new memory architecture needs to be implemented for every application, potentially a more challenging task when the parallel memory is to be implemented in hardware.

Hybrid PMs. Besides the two extremes discussed above, there are also hybrid implementations of parallel memories, which combine the advantages of the two previous approaches by using partial data redundancy [13]. Of course, in this case, the challenge is to determine which data should be replicated and where. In turn, this solution requires both application and architecture customization.

2.2 The Polymorphic Register File 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 1 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. PolyMem reuses the PRF conflict-free parallel storage techniques and patterns, as well as the polymorphism idea. Figure 2(a) illustrates the access patterns supported by the PRF and PolyMem.



(a) Supported AccessTypes (b) Block diagram

Fig. 2. PRF [4] design. 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 select the shape of the parallel access.

Table 1. 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

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. In a hardware implementation with eight memory banks, each of these regions can be read using one (R1–R9) or several (R0) parallel accesses.

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 1. Each scheme supports dense, conflict-free access to $p \cdot q$ elements. When implemented in reconfigurable tech-

nology, PolyMem allows application-driven customization: its capacity, number of read/write ports, and the number of lanes can be configured to best support the application needs.

The block diagram in Fig. 2(b) shows, at high level, the PRF 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 operation. The inputs of the PRF are shown at the top of the diagram. AccessType represents the parallel access pattern. (i, j) are the top-left coordinates of the parallel access. The list of elements to access is generated by the AGU module and is sent to the A and m modules: the A module generates one in-memory address for each memory bank in the PRF, while the m module applies the mapping function of the selected scheme and computes, for each accessed element, the memory bank where it is stored. The Data Shuffle block reorders the Data In/Out, ensuring the PEF user obtains the accessed data in their original order.

2.3 Matrix Storage in a Parallel Memory

Figure 3 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 Fig. 3 shows an example of a matrix to be stored in the partitioned

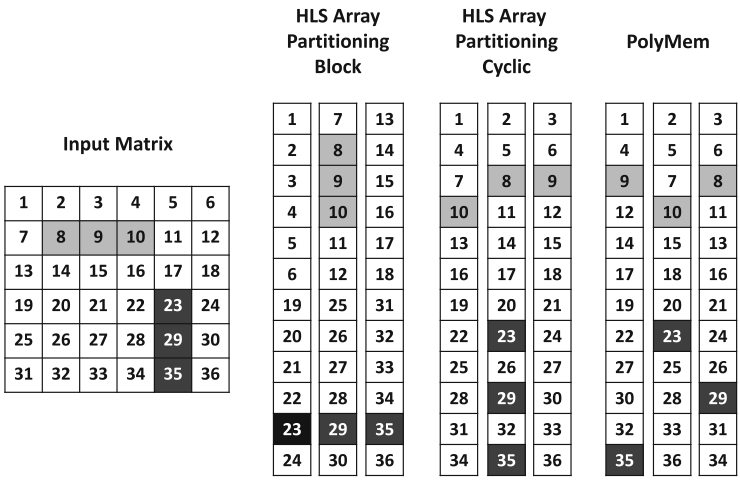


Fig. 3. Comparison between different partitioning techniques offered by Vivado HLS (facto = 3) and the RoCo scheme of PolyMem, with 3 memory banks, for data stored in a 6×6 matrix. PolyMem allows 3 parallel data reads/writes, from the rows and the columns of the original matrix. Unaligned blocks are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

BRAMs, aiming to achieve read/write parallelism. The right side illustrates three techniques used to partition the matrix, using two unaligned, parallel accesses of 3 elements (gray and black in the figure), starting respectively from the cells containing elements 8 and 23. The HLS Array Partitioning techniques enable either the black or the gray access to be performed in parallel (for Block and Cyclic, respectively). Using PolyMem with a RoCo scheme, each element of each access is mapped on a different memory bank; in turn, this organization enables *both* the gray and the black access to happen in a single (parallel) operation[1].

3 Implementation Details

This section describes the main components of our PolyMem implementation for Vivado HLS. The goal of integrating PolyMem in 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 access patterns.

Our Vivado HLS PolyMem implementation exploits all the presented five schemes (ReO, ReRo, ReCo, RoCo, ReTr) to store on the FPGA BRAMs the data required to perform the application operations. Compared to the default Vivado memory partitioning techniques, which allow hardware parallelism with a single access pattern, a PolyMem employing a multiview scheme allows multiple types of access simultaneously for unaligned data with conservative hardware resources usage.

We implemented a template-based class *polymem* that exploits loop unrolling to parallelize memory accesses. When HLS PolyMem is instantiated within the user application code, it is possible to specify *DATA_T*, i.e., the type of data to be stored, the $(p \times q)$ number of internal banks of memory (i.e., 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. Listing 1.1 presents the interfaces of methods that allow accesses to data stored within PolyMem. Simple **read** and **write** methods use the m and A modules (described in Sect. 2.2) 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 requires less data to be stored (e.g., 5 elements), and wants avoid overwriting existing data (e.g., the remaining 3 elements).

[1] This small-scale example is included for visualization purposes only. Real-applications are likely to use more memory banks, allowing parallel accesses to larger data blocks.

Listing 1.1. List of the methods interfaces to allow user read/write data by used sequential or parallel accesses

```
DATA_T read(int i, int j);

void write(DATA_T data, int i, int j);

void read_block(int i, int j, DATA_T out[p * q],
                int PRF_ACCESS_TYPE);

void write_block(DATA_T in[p * q], int i, int j,
                 int PRF_ACCESS_TYPE);

void write_block_masked(DATA_T in[p * q],
                        ap_uint<p * q> mask,
                        int i, int j,
                        int PRF_ACCESS_TYPE);
```

4 Evaluation and Results

In this Section, we focus on the evaluation of HLS PolyMem. The evaluation is based on a synthetic benchmark, where we demonstrate that PolyMem offers a high-performance, high-productivity alternative to partitioned memories in HLS.

4.1 Experimental Setup

We present the design and implementation of our microbenchmarking suite, and detail the way all our measurements are performed. All the experiments in this section are validated and executed on a Xilinx Virtex-7 VC707 board (part xc7vx485tffg1761-2), with the following hardware resources: 303600 LUTs, 607200 FFs, 1030 BRAMs, and 2800 DSPs. We instantiate a Microblaze processor on the FPGA to control the DMA that transfers data between the FPGA board DRAM memory and the on-chip BRAMs where the computational kernel performs memory accesses. The Microblaze also starts and stops an AXI Timer to measure the execution time of each experiment. The data transfers to and from the computational kernel employ the AXI Stream technology.

Microbenchmark Design. To provide an in-depth evaluation of our Polymorphic memory, we designed a specific microbenchmark which tests the performance of PolyMem together with its flexibility - i.e., its ability to cope with applications that require different parallel access types *to the same data structure*. Moreover, we compare the results of the Polymem-augmented design with the ones achievable by partitioning the memory with the default techniques available in Vivado HLS. To ensure a fair comparison, we utilize a Vivado HLS Cyclic array partition with a factor equal to the number of PolyMem lanes (both designs can access at most $p \cdot q$ data elements in parallel from the BRAMs).

The requirements we state for such a microbenchmark are:

1. Focus on the reading performance of the memory, in terms of bandwidth;
2. Support all access types presented in Sect. 2.2;
3. Test a combination of more access types, to further demonstrate the flexibility of polymorphism;
4. Measure the overall bandwidth achieved by these memory transfers.

To achieve these requirements, we designed different computational kernels (IP Cores) that perform a (configurable) number of parallel memory reads, from various locations inside the memory, using different parallel access patterns. Each combination of parallel reads is performed in two different scenarios. The accessed on-chip FPGA memory (BRAMs) M, where the input data are stored, is partitioned by using (1) the default techniques available in Vivado HLS, and (2) the HLS PolyMem technology.

A high-level description of the operations executed by the computational kernels and measured by the timer is presented in Listing 1.2. Memory M is used to store the input data and it is physically implemented in partitioned BRAMs. The kernel receives the data to fill the memory M and N_READS matrix coordinates to perform parallel accesses with different access types - i.e., *given an access type and the matrix coordinates (i, j), the computational kernel reads a block of data starting from (i, j) and following the access type.* When the memory reads are done, the kernel sends sampled results on the output stream.

Listing 1.2. The structure of the proposed microbenchmark

```
stream in data to fill memory M
stream in N_READS read_coordinates

synchronize // wait for streaming to complete

// process reads
foreach ACCESS_TYPE in POLYMEM_SCHEME_SUPPORTED_ACCESS_TYPES:
  chunk_size = N_READS / N_SUPPORTED_ACCESS_TYPES
  foreach (i,j) in chunk_of_read_coordinates:
      current_results_block = M.read_block(i, j, ACCESS_TYPE)
// done processing reads

foreach k in range(N_RESULTS_BLOCKS):
  stream out the k^th results_block

synchronize // wait for streaming to complete
```

By comparing the performance results of HLS-partitioning and PolyMem, we are able to assess which scheme provides both performance and flexibility, and, moreover, provide a quantitative analysis of the performance gap between the two. We provide more details on how the measurements are performed in the following paragraphs.

The complete code used for all the experiments described in this section is available in our code repository [45].

Measurement Setup. In order to measure the performance of the two different parallel memory microbenchmaks, we propose a setup as presented in Fig. 4. Specifically, in this diagram, "Memory" is either an HLS-partitioned memory, or an instance of PolyMem (as described in the previous paragraph).

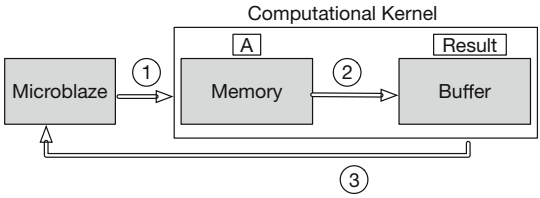


Fig. 4. The measurement setup used for microbenchmarks. The measured bandwidth corresponds to phase 2, where reading the data from the parallel memory happens.

In order to measure the performance of the two memories, we propose an approach based on the following steps.

1. Measure, on the Microblaze processor, the overhead of phases 1 and 3. We note that phases 1 and 3 are implemented using the basic streaming constructs provided by the AXI Stream technology. We achieve this by a one-time measurement where no memory reads or only one memory read are performed.
2. Measure, on the Microblaze processor, the execution time of the complete setup, from the beginning of phase 1 to the end of phase 3. Due to the explicit synchronization between phases 1, 2, and 3, we can guarantee that no overlap happens between them.
3. Determine, by subtraction, the absolute execution time of phase 2 alone, which is a measure of the parallel memory's reading performance.
4. Present the absolute performance of the two memories in terms of achieved bandwidth. For the case of PolyMem, we can also assess the efficiency of the implementation by comparing the achieved bandwidth with that predicted by the theoretical performance model [14].
5. Present the relative performance of the two parallel memories as **speedup**. We calculate speedup as the ratio of the execution time of HLS-based partitioning solution over the execution of the PolyMem-based solution. We chose to use the entire execution time, including the copy overhead, as an estimate of a realistic benchmark when the same architecture is used for real-life applications. We note that this choice is pessimistic, as the overhead of phases 1 and 3 can be quite large.

4.2 Results

All the results presented in this section are from experiments performed using the settings in Table 2.

The input data stream employs double precision (64-bit) numbers, and the computational kernel receives an amount of data (equal for all the experiments), that includes the input matrix and the list of coordinates (i, j):

Table 2. Microbenchmark settings

Clock frequency (ClkFr)	200 MHz
Data type (DType)	64-bit double
Input matrix size ($DIM \times DIM$)	96×96
HLS partitioning factor (FACTOR)	16
PolyMem lanes ($p \times q$)	16 (2×8)
Number of passed coordinates (i, j) (N_READS)	3072
Size of each read block (BLOCK_SIZE)	16
Number of output blocks (N_RESULTS_BLOCKS)	50

$$\text{N\_IN\_DATA} = (\text{DIM} \cdot \text{DIM}) + (\text{N\_READS} \cdot 2) = 15360 \text{ 64-bit elements}$$

The number of data that the computational kernel reads from the memory is computed as follow:

$$\text{N\_READ\_DATA} = (\text{N\_READS} \cdot \text{BLOCK\_SIZE}) = 49152 \text{ 64-bit elements}$$

The output data stream employs double precision 64-bit numbers, and the computational kernel sends back to the microblaze a sample of the results (data read), equal for all the experiments, amounting to:

$$\text{N\_OUT\_DATA} = (\text{N\_RESULTS\_BLOCKS} \cdot \text{BLOCK\_SIZE}) = 80064\text{-bit elements}$$

To measure the overheads introduced by the data transfers in terms of hardware resources utilization and execution time, we implemented two computational kernels: the first one does not perform any memory accesses (the BRAMs are not even partitioned) and the second one performs only one memory access (the added execution time of this one access is negligible). The second kernel was executed for both memory configurations (HLS Cyclic and PolyMem). The results are shown in Table 3. The consistent execution time indicates that the overhead is systematic and constant.

Table 3. Hardware resources utilization and execution time spent in phases 1 and 3 of the proposed architecture

Memory	Access	LUT	FF	BRAM	DSP	Runtime [µs]
-	-	41400	34064	21	0	265
HLS Cyclic	Row	43194	35302	172	0	265
PolyMem	Row	46375	36444	172	0	265

Table 4. Hardware resources utilizations, execution times and bandwidths for microbenchmark experiments with different memory configurations and access schemes

Memory	Scheme	LUT	FF	BRAM	DSP	Runtime [µs]	BW [GB/s]
HLS Cyclic	ReO	45800	36121	172	0	503	1.54
PolyMem	ReO	45590	36364	172	0	283	20.35
HLS Cyclic	ReRo	90197	65993	174	224	503	1.54
PolyMem	ReRo	59082	40661	172	0	283	20.35
HLS Cyclic	ReCo	85055	64679	174	164	503	1.54
PolyMem	ReCo	62549	40434	172	0	283	20.35
HLS Cyclic	RoCo	67066	54217	174	100	503	1.54
PolyMem	RoCo	55025	38944	172	0	283	20.35
HLS Cyclic	ReTr	62259	54244	174	40	503	1.54
PolyMem	ReTr	51282	37744	172	0	283	20.35

Given the data transfers execution time overhead equal to 265 ns, we can compute the bandwidth (BW) in GB/s for each new experiment with the following formula:

$$BW[B/s] = \frac{\text{N\_READS} * \text{BLOCK\_SIZE} * 8}{(\text{Exec. time} - \text{overhead})}$$

In Table 4, we report the detailed results of our microbenchmarking experiments, in terms of hardware resource utilization, execution time, and bandwidth. We provide results for the two different memory configurations and all PolyMem access schemes. As shown in Listing 1.2, the memory accesses are equally divided among the access patterns supported by the selected scheme. We further note that, for all the schemes, the speedup of the end-to-end computation (i.e., phases 1, 2 and 3 from Fig. 4) is 1.78x. For the actual computation, using the parallel memory (i.e., without the data transfer overhead), the PolyMem outperforms HLS partitioning by as much as 13.22x times. Moreover, in terms of hardware resources, (1) the BRAM utilization is similar for both parallel memories, which indicates no overhead for PolyMem, (2) PolyMem is more economical in terms of "consumed" LUT and FF (up to 20% less), and (3) HLS partitioning makes use of DSPs, while PolyMem does not. The following paragraph contains an evaluation of these results.

Unaligned Accesses and Final Evaluation. The results suggest that the Vivado HLS default partitioning techniques are not able to exploit parallel reads for the described access patterns. This is due to the fact that, even if the data are correctly distributed among the BRAMs to perform at least one access type, parallel accesses unaligned with respect to the partitioning factor are not supported. To prove that, we perform experiments where *the memory reads are forced to be aligned with respect to the partitioning factor*, for one of the access

type - e.g. having a cyclic partitioned factor of 4 on the Ro access, it is possible to read 4 data in parallel at the coordinates {(i, j), (i, j+1), (i, j+2), (i, j+3)}, only if j is a multiple of 4. This is possible, at compile time, by using the *integer division* on the reading coordinates (i, j) as follows:

$$\text{aligned\_j} = \left\lfloor \frac{j}{\text{BLOCK\_SIZE}} \right\rfloor * \text{BLOCK\_SIZE}$$

This ensures that aligned_j is a multiple of the number of memory banks - i.e. BLOCK_SIZE. Using aligned_j for the data access allows the HLS compiler to perform more aggressive optimizations parallelizing the access to the partitioned memory. Table 5 shows the results for the RoCo scheme with different combinations of access types, where forced aligned accesses are performed or not. The cases where the memory reads are aligned with respect to the partitioning factor are the only ones where the default Vivado HLS partitioning is able to achieve the same performance of PolyMem, while using fewer hardware resources. However, even in this cases, the default Vivado HLS partitioning is not able to perform all the memory accesses with the right amount of parallelism if the application requires multiple access patterns. Practical examples showing the advantages of using PolyMem are provided in the following section.

5 Application Case-Studies

In this Section, we analyze two case-study applications, i.e., matrix multiplication and Markov chain, that exploit our HLS PolyMem to parallelize accesses to matrix data.

Each application demonstrates different HLS PolyMem features. 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 gains with minimal changes to the original software code.

Table 5. Hardware resources utilizations, execution times and bandwidths for the RoCo scheme with different combinations of access types with and without forced aligned accesses (**FA**)

Memory	Access types	LUT	FF	BRAM	DSP	Runtime [µs]	BW [GB/s]
HLS Cyclic	Ro	60127	52552	174	64	503	1.54
PolyMem	Ro	45641	36432	172	0	283	20.35
HLS Cyclic	**FA Ro**	43048	35316	173	0	283	20.35
PolyMem	**FA Ro**	45175	36391	173	0	283	20.35
HLS Cyclic	Ro, Co, Re	67066	54217	174	100	503	1.54
PolyMem	Ro, Co, Re	55025	38944	172	0	283	20.35
HLS Cyclic	Ro, Co, **FA Re**	47812	38003	173	0	429	2.23
PolyMem	Ro, Co, **FA Re**	55328	38975	173	0	283	20.35

5.1 Matrix Multiplication (MM)

With this case study, we aim to demonstrate the usefulness of the multiview property of HLS-PolyMem. Specifically, we investigate, in the context of a real application, two aspects: (1) if there is any performance loss or overhead between the two parallel memories for a single matrix multiplication, and (2) what is the performance gap between the two types of parallel memories in the case where multiple parallel access shapes are needed, on the same data structure, in the same application.

Single Matrix Multiplication. For our first experiment, the application performs one multiplication of 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 1.3 and 1.4 show the declaration of the matrices and their partitioning using the HLS default partitioning and the HLS PolyMem, respectively. Both parallel memories use 16 lanes (i.e., data is partitioned onto 16 memory banks): the HLS partitioned scheme uses a parallel factor of 16, while the B and C HLS PolyMem instances are initialized with $p = 4$ and $q = 4$.

Listing 1.3. 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 1.4. 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 1.5 and 1.6 show the matrix multiplication code when using the HLS default partitioning and the HLS PolyMem, respectively.

Listing 1.5. 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 1.6. 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;
  }
```

Double (Mirrored) Matrix Multiplication. 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 second kernel function, in which both the $B \times C$ and the $C \times B$ products need to be computed. This effectively means that the new kernel can only enable 16 parallel accesses for both multiplications if the matrices allow parallel reads in using both row- and column-patterns.

Results and Analysis. Table 6 reports the latency and resource utilization estimated by Vivado HLS when computing the single matrix multiplication kernel (1MM), $B \times C$ (rows $1, 2$), and when computing the double multiplication (2MM's), $B \times C$ followed by $C \times B$ (rows $3, 4$ and $5, 6$) for the two parallel memories under consideration.

As expected, when using the default Vivado HLS partitioning techniques, the second multiplication ($C \times B$) cannot be computed efficiently due to the way in which the matrix data is partitioned into the memory banks, as described in Sect. 2. Indeed, C can only be accessed in parallel by rows and B by columns.

Table 6. Latency and hardware resources for matrix multiplication with different memory configurations and matrix dimensions

Memory	Matrix size	Parallel factor	Latency		Hardware resources			
			1 MM	2 MM's	BRAM	DSP	FF	LUT
HLS	32	4	4227	n.a	18	40	6162	6485
PolyMem	32	4 (2×2)	4227	n.a	18	40	6153	6018
HLS	32	4	4227	16503	18	40	7444	9197
PolyMem	32	4 (2×2)	4227	4227	18	40	7367	7364
HLS	96	16	28033	442722	96	164	28554	40474
PolyMem	96	16 (4×4)	28033	28033	96	160	30969	43636

On the other hand, the implementation based on HLS PolyMem is perfectly capable of performing both matrix products ($B \times C$ and $C \times B$) efficiently. The performance data reflects this very well: the estimated latency reported in Table 6 is the same for both products in the PolyMem case, and drastically different in the case of HLS partitioning.

It is also worth noting that for a matrix size of 32×32, 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 100 MHz. The benchmarking system is similar to that presented in Sect. 4: a soft Microblaze core is used to manage the experiment, the input/output data (matrices B and C, and the result) are streamed into parallel memory, and the actual multiplication operations are performed using the parallel memory. For the kernel with a single multiplication, the performance of the two solutions is the same. However, for the kernel with the double multiplication, the HLS PolyMem version achieves an overall speedup of 5x compared to the implementation based on HLS memory partitioning.

5.2 Markov Chain and the Matrix Power Operation

With this case study, which has at its core the matrix power operation, we aim to reinforce the need for multiview accesses to the same data structure, and further demonstrate how tiling can be easily achieved and used in conjuction with HLS-PolyMem, to further alleviate its resource overhead.

A Markov Chain is a stochastic model used to describe real-world processes. Some of its most relevant applications are found in queuing theory, the study of population growths [15], and in stochastic simulation methods such as Gibbs sampling [16] and Markov Chain Monte Carlo [17]. Moreover, Page Rank [18], 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. We then 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.

Listing 1.7 shows an HLS PolyMem-based algorithm that can be used to compute A^{2^h}. The implementation consists of an outermost loop repeated h times in which we compute the product $A \times A$ whose result is stored in A_temp and copied back to the PolyMem for A before the next iteration.

Implementing the same algorithm by using the HLS partitioning techniques, as presented in the previous case study, results in poor exploitation of the available parallelism, or in duplicated data, since A needs to be accessed both by rows and columns.

Listing 1.7. HLS PolyMem implementation of A^{2^h}

```
hls::prf<float, p, q, DIM, DIM, SCHEME_RoCo> A;

for(int iter=0; iter<h; iter ++){
  // A*A 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 += p*q) {
        A.read_block(i, k, temp_row, ACCESS_Ro);
        A.read_block(k, j, temp_col, ACCESS_Co);
        for (int t = 0; t < p*q; t++)
          sum += temp_row[t] * temp_col[t];
      }
      A_temp[i][j] = sum;
    }
  }
```

```
  // Copy back results to PolyMem
  for (int i = 0; i < DIM; ++i){
    for (int t = 0; t < DIM; t += p*q) {
#pragma HLS PIPELINE II=1
      A.write_block(&A_temp[i][t], i, t, ACCESS_Ro);
    }
  }
}
```

The HLS PolyMem enables parallel accesses to matrix A for both rows and columns, but adds some 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 a simple form of tiling, were we reduce the dimension of PolyMem by dividing the input matrix A and storing its values in a grid of multiple PolyMem s. If A has $DIM \times DIM$ elements, it is possible to organize the on-chip memory to store data in a grid of $b \times b$ square 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 PolyMem reduces 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.

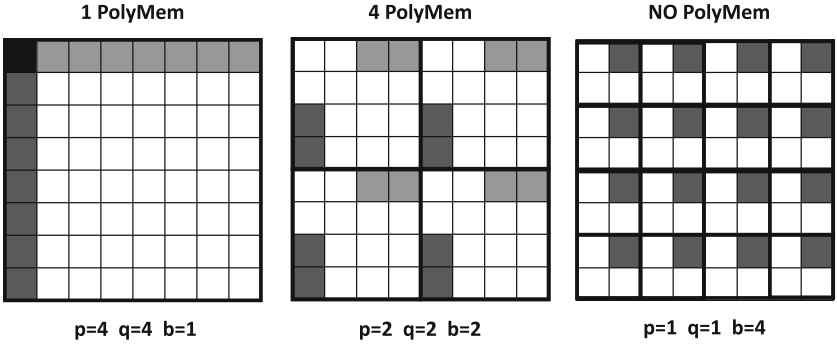


Fig. 5. Comparison between different partitioning of the input matrix in a grid of b^2 components implemented by PolyMem with a level of parallelism of $p \times q$. When both p and q are set to 1, it is possible to remove the HLS PolyMem logic.

In Table 7 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 384×384 matrix. The numbers demonstrate that 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 as long, in terms of lines of code, as 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. In this case, the benchmarking follows a similar method as the one presented in Sect. 4.1 and Fig. 4, with two amendments: (1) instead of using the Microblaze softcore, we manage the experiment directly from the CPU of the host system where the FPGA board acts as an accelerator, and (2) the transfers from stages (1) and (3) are performed in blocks over the PCIe bus. We synthesized a design for a matrix size of 256 and parameters $p = q = b = 2$ at 200 MHz, and we benchmarked its performance on the Xilinx Kintex Ultrascale ADM-PCIE-KU3 platform. The obtained throughput is 1.6 GB/s. We note that this number is significantly lower than the expected performance of the HLS-PolyMem itself because it also includes the PCIe overhead. Without this overhead, the performance of the computation using the parallel memory alone is expected to be similar to the performance of a single PolyMem block with $p \times q$ lanes, running

Table 7. 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	1557835871	1036	14	9936	11071	98
PolyMem	2	4	1	840333407	1044	17	19678	28855	98
PolyMem	4	4	1	488632423	1060	31	36138	53621	98
multi PolyMem	1	1	2	758085955	1036	14	6967	5572	188
multi PolyMem	1	2	2	394149976	1044	28	14709	12934	188
multi PolyMem	2	2	2	214032480	1060	45	24845	22418	188
NO PolyMem	1	1	4	101848419	1124	76	32852	13706	188

at 200 MHz, which should be in the same order as that presented in Table 1 (i.e., 21 GB/s for a 16-lane HLS-PolyMem).

6 Related Work

The concept of parallel memory is fairly old, and has been widely discussed in scientific literature. As early as 1971, Kuck et al. discuss the advantages and disadvantages of using memory systems with power of two memory banks [19], based on results collected from a study on performed on the Illiac IV machine.

One of the earliest design methodologies and general designs of a parallel memory system suitable, dedicated to image processing applications, are presented in [20]. The memory space is already organized as a 2D structure, while the parameters p, q are the parameters of the parallel region to be accessed; the authors discuss three different mapping functions, and ultimately demonstrate the benefits parallel accesses bring to image processing.

In the 90s, more work has been devoted to investigating various addressing schemes and their implementation. For example, [9] investigates schemes based on linear addressing transformation (i.e., XOR schemes), and the use of these schemes for accessing memory in conflict-free manner using multiple strides, blocks, and FFT access patterns. In [21], another memory system design, enabling different parallel accesses to a 2D parallel memory is presented; their design is different in that it focuses on 2D memories to be accessed by arrays of 2D processing units, and thus their mapping and addressing functions are specialized.

SIMD processors have fueled more research in building and using parallel memories efficiently. For example, the use of memory systems that leverage a prime number of memory modules to provide parallel accesses for rectangles, rows, columns, and diagonals is investigated in [22]; the authors prove the advantages in building fast mapping/addressing functions for such particular memories, an idea already envisioned and analyzed in [23]. In the same work [22], Park also introduces a Multi Access Memory System, which provides access to multiple sub-array types, although it uses memory modules in a redundant manner. Research proposing an addressing function for 2D rectangular accesses, suitable for multimedia applications, is presented in [10]; the aim of this work is to minimize the number of required memory modules for efficient (i.e., full utilization) parallel accesses. The work in [24] also aims at the full utilization of the memory modules, introducing a memory system based on linear skewing (the same idea from 1971 [19]) that support accesses to block and diagonal conflict-free accesses in a 2D space. [25] proposes a memory system with power of 2 memory modules able to perform strided access with a power of two interval in horizontal and vertical directions. The analysis of parallel memories is also refined - for example, the effect of using a parallel memory to the dynamic instruction count of an application is explored in [8].

The PRF multiview access schemes - which are fundamental for this work - are explained in detail in [4], together with the hardware design and implementation requirements. This work introduces an efficient HLS implementation of

the PRF addressing schemes, greatly simplifying the deployment of PolyMem on FPGAs. Alternative schemes also exist. For example, the Linear-Transformation-Based (LTB) algorithm for automatic generation of memory partitions of multi-dimensional arrays, which is suitable for being used during FPGA HLS loop pipelining, is described in [11]. The Local Binary Pattern (LBP) algorithm from [12] considers the case of multi-pattern and multi-array memory partitioning. [26] discusses the advantages of a hierarchical memory structures generated on tree-based network, as well as different methods for their automatic generation.

Building a memory hierarchy for FPGA kernels is recognized as a difficult, error-prone task [27,28]. For example, [28–32] focus on the design of generic, traditional caches. Moreover, the recently released High-Level Synthesis (HLS) tools for FPGAs [33] provide a simple set of parallel access patterns to on-chip memory starting from high-level languages implementations. More recently, work has been done on using the Polyhedral Model to automatically determine the module assignment and addressing functions [34]. By comparison, our work proposes a parallel, polymorphic memory which can be exploited from HLS tools and *acts as a caching mechanism* between the DRAM and the processing logic; instead of supporting placement and replacement policies, our memory is configured for the application at hand, and it is directly accessible for reading and writing. Moreover, PolyMem includes a *multiView feature*, enabling multiple conflict-free access types, a capability not present in other approaches [34].

Application-specific caches have also been investigated for FPGAs [26,29,35], though none of these are polymorphic or parallel. For example, in [36], the authors demonstrate why and how different caches can be instantiated for specific data structures with different access patterns. PolyMem starts from a similar idea, but, benefiting from its multi-view, polymorphic design, it improves on it by using a single large memory for all these data structures. Many of PolyMem's advantages arise from its PRF-based design [4], which is more flexible and performs better than alternative memory systems [37–40]; its high performance in scientific applications has also been proven for practical applications [41–43]. As stated before, the first hardware implementation of the Polymorphic Register File was designed in System Verilog [5]. MAX-PolyMem was the first prototype of PolyMem written entirely in MaxJ, and targeted at Maxeler DFEs [3,6]. Our new HLS PolyMem is an alternative HLL solution, proven to be easily integrated with the Xilinx toolchains.

In summary, compared to previous work on enabling easy-to-use memory hierarchies and/or caching mechanisms for FPGAs, PolyMem proposes a PRF-based design that supports polymorphic parallel accesses through a single, multi-view, application-specific software cache. The previous HLS implementation [3] has demonstrated good performance, but was specifically designed to be used on Maxeler-based systems. Our current HLS-PolyMem is the most generic implementation to date, it preserves the advantages of the previous incarnations of the system in terms of performance and flexibility, and adds the ease-of-use of an HLS library that can be easily integrated in the design flow of modern tools like Vivado HLx and Vivado SDx.

7 Conclusion and Future Work

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. Compared to the naive optimizations using HLS array partitioning techniques, the HLS PolyMem implementation is better in terms of performance, provides high flexibility in terms of supported parallel access patterns, and requires virtually zero implementation effort in terms of code re-engineering. Our design exposes an easy-to-use interface to enhance design productivity for FGPA-based applications. This interface provides methods for both the basic parallel read/write operations, and it is extended with to support masked on-chip parallel accesses. Furthermore, we provide a full, open-source implementation of HLS-PolyMem, supporting all the original PolyMem schemes [45]. Our evaluation, based on comprehensive microbenchmarking, demonstrates sustained high-performance for all these schemes. Our results demonstrate HLS-PolyMem achieves the same level of performance as HLS-partitioning for simple access patterns (i.e., rows and columns), and significant performance benefits compared with HLS-partitioning for more complex access patterns. We observe bandwidth improvement as high as 13x for complex access patterns combinations, which HLS partitioning simply cannot support.

We also 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 two case studies (Matrix multiplication and Markov Chains) 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 three different directions. First, we aim to provide the usability of HLS for more case-studies, and further develop the API to better support end-users. Second, we aim to further improve the implementation of the HLS-PolyMem backend. For example, we consider improving the HLS PolyMem shuffle module by exploiting a Butterfly Network [44] for the memory banks connections, and enhance our HLS implementation to support both standard and customized addressing. Third, we envision a wizard-like framework to automatically analyze the user application code, estimate the potential benefits of using HLS-PolyMem, and suggest how to actually embed the parallel memory in the code to reach the best possible performance.

References

1. White Paper: Vivado Design Suite: "Vivado Design Suite" (2012). https://www.xilinx.com/support/documentation/white_papers/wp416-Vivado-Design-Suite.pdf
2. Weinhardt, M., Luk, W.: Memory access optimisation for reconfigurable systems. IEE Proc. Comput. Digit. Tech. **148**(3), 105–112 (2001)
3. Ciobanu, C.B., Stramondo, G., de Laat, C., Varbanescu, A.L.: MAX-PolyMem: high-bandwidth polymorphic parallel memories for DFEs. In: IEEE IPDPSW - RAW 2018, pp. 107–114, May 2018

4. Ciobanu, C.: Customizable register files for multidimensional SIMD architectures. Ph.D. thesis, TU Delft, The Netherlands (2013)
5. Ciobanu, C., Kuzmanov, G.K., Gaydadjiev, G.N.: Scalability study of polymorphic register files. In: Proceedings of DSD, pp. 803–808 (2012)
6. Ciobanu, C.B., et al.: EXTRA: an open platform for reconfigurable architectures. In: SAMOS XVIII, pp. 220–229 (2018)
7. Stornaiuolo, L., et al.: HLS support for polymorphic parallel memories. In: 2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), pp. 143–148. IEEE (2018)
8. Gou, C., Kuzmanov, G., Gaydadjiev, G.N.: SAMS multi-layout memory: providing multiple views of data to boost SIMD performance. In: ICS, pp. 179–188. ACM (2010)
9. Harper, D.T.: Block, multistride vector, and FFT accesses in parallel memory systems. IEEE Trans. Parallel Distrib. Syst. **2**(1), 43–51 (1991)
10. Kuzmanov, G., Gaydadjiev, G., Vassiliadis, S.: Multimedia rectangularly addressable memory. IEEE Trans. Multimedia **8**, 315–322 (2006)
11. Wang, Y., Li, P., Zhang, P., Zhang, C., Cong, J.: Memory partitioning for multidimensional arrays in high-level synthesis. In: DAC, p. 12. ACM (2013)
12. Yin, S., Xie, Z., Meng, C., Liu, L., Wei, S.: Multibank memory optimization for parallel data access in multiple data arrays. In: Proceedings of ICCAD, pp. 1–8. IEEE (2016)
13. auf der Heide, F.M., Scheideler, C., Stemann, V.: Exploiting storage redundancy to speed up randomized shared memory simulations. Theor. Comput. Sci. **162**(2), 245–281 (1996)
14. Stramondo, G., Ciobanu, C.B., Varbanescu, A.L., de Laat, C.: Towards application-centric parallel memories. In: Mencagli, G., et al. (eds.) Euro-Par 2018. LNCS, vol. 11339, pp. 481–493. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-10549-5_38
15. Arsanjani, J.J., Helbich, M., Kainz, W., Boloorani, A.D.: Integration of logistic regression, Markov chain and cellular automata models to simulate urban expansion. Int. J. Appl. Earth Obs. Geoinformation **21**, 265–275 (2013)
16. Smith, A.F., Roberts, G.O.: Bayesian computation via the Gibbs sampler and related Markov chain Monte Carlo methods. J. R. Stat. Society. Ser. B (Methodol.) **55**, 3–23 (1993)
17. Gilks, W.R., Richardson, S., Spiegelhalter, D.: Markov Chain Monte Carlo in Practice. CRC Press, Boca Raton (1995)
18. Kamvar, S.D., Haveliwala, T.H., Manning, C.D., Golub, G.H.: Extrapolation methods for accelerating PageRank computations. In: Proceedings of the 12th International Conference on World Wide Web, pp. 261–270. ACM (2003)
19. Budnik, P., Kuck, D.: The organization and use of parallel memories. IEEE Trans. Comput. **C–20**(12), 1566–1569 (1971)
20. Van Voorhis, D.C., Morrin, T.: Memory systems for image processing. IEEE Trans. Comput. **C–27**(2), 113–125 (1978)
21. Kumagai, T., Sugai, N., Takakuwa, M.: Access methods of a two-dimensional access memory by two-dimensional inverse omega network. Syst. Comput. Jpn. **22**(7), 22–31 (1991)
22. Park, J.W.: Multiaccess memory system for attached SIMD computer. IEEE Trans. Comput. **53**(4), 439–452 (2004)
23. Lawrie, D.H., Vora, C.R.: The prime memory system for array access. IEEE Trans. Comput. **31**(5), 435–442 (1982)

24. Liu, C., Yan, X., Qin, X.: An optimized linear skewing interleave scheme for on-chip multi-access memory systems. In: Proceedings of the 17th ACM Great Lakes Symposium on VLSI, GLSVLSI 2007, pp. 8–13 (2007)
25. Peng, J.y., Yan, X.l., Li, D.x., Chen, L.z.: A parallel memory architecture for video coding. J. Zhejiang Univ. Sci. A **9**, 1644–1655 (2008). https://doi.org/10.1631/jzus.A0820052
26. Yang, H.J., Fleming, K., Winterstein, F., Chen, A.I., Adler, M., Emer, J.: Automatic construction of program-optimized FPGA memory networks. In: FPGA 2017, pp. 125–134 (2017)
27. Putnam, A., et al.: Performance and power of cache-based reconfigurable computing. In: ISCA 2009, pp. 395–405 (2009)
28. Adler, M., Fleming, K.E., Parashar, A., Pellauer, M., Emer, J.: Leap scratchpads: automatic memory and cache management for reconfigurable logic. In: FPGA 2011, pp. 25–28 (2011)
29. Chung, E.S., Hoe, J.C., Mai, K.: CoRAM: an in-fabric memory architecture for FPGA-based computing. In: FPGA 2011, pp. 97–106 (2011)
30. Yiannacouras, P., Rose, J.: A parameterized automatic cache generator for FPGAs. In: FPT 2003 (2003)
31. Gil, A.S., Benitez, J.B., Calvino, M.H., Gomez, E.H.: Reconfigurable cache implemented on an FPGA. In: ReConFig 2010 (2010)
32. Mirian, V., Chow, P.: FCache: a system for cache coherent processing on FPGAs. In: FPGA 2012, pp. 233–236 (2012)
33. Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., Zhang, Z.: High-level synthesis for FPGAs: from prototyping to deployment. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **30**(4), 473–491 (2011)
34. Wang, Y., Li, P., Cong, J.: 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, FPGA 2014, pp. 199–208. ACM, New York (2014)
35. Putnam, A.R., Bennett, D., Dellinger, E., Mason, J., Sundararajan, P.: CHiMPS: a high-level compilation flow for hybrid CPU-FPGA architectures. In: FPGA 2008, p. 261 (2008)
36. Nalabalapu, P., Sass, R.: Bandwidth management with a reconfigurable data cache. In: IPDPS 2005. IEEE (2005)
37. Kuck, D., Stokes, R.: The Burroughs scientific processor (BSP). IEEE Trans. Comput. **C–31**(5), 363–376 (1982)
38. Panda, D., Hwang, K.: Reconfigurable vector register windows for fast matrix computation on the orthogonal multiprocessor. In: Proceedings of ASAP, pp. 202–213, May–July 1990
39. Corbal, J., Espasa, R., Valero, M.: MOM: a matrix SIMD instruction set architecture for multimedia applications. In: Proceedings of the SC 1999 Conference, pp. 1–12 (1999)
40. Park, J., Park, S.B., Balfour, J.D., Black-Schaffer, D., Kozyrakis, C., Dally, W.J.: Register pointer architecture for efficient embedded processors. In: Proceedings of DATE, pp. 600–605 (2007)
41. Ramirez, A., et al.: The SARC architecture. IEEE Micro **30**(5), 16–29 (2010)
42. Ciobanu, C., Martorell, X., Kuzmanov, G.K., Ramirez, A., Gaydadjiev, G.N.: Scalability evaluation of a polymorphic register file: a CG case study. In: Proceedings of ARCS, pp. 13–25 (2011)
43. Ciobanu, C., Gaydadjiev, G., Pilato, C., Sciuto, D.: The case for polymorphic registers in dataflow computing. Int. J. Parallel Program. **46**, 1185–1219 (2018)

44. Avior, A., Calamoneri, T., Even, S., Litman, A., Rosenberg, A.L.: A tight layout of the butterfly network. Theory Comput. Syst. **31**(4), 475–488 (1998)
45. https://github.com/storna/hls_polymem