Mack-PolyMem: High-Bandwidth Polymorphic Parallel Memories for DFEs

Cătălin Bogdan Ciobanu, Giulio Stramondo, Cees de Laat, Ana Lucia Varbanescu
Informatics Institute, University of Amsterdam, The Netherlands,
{c.b.ciobanu,g.stramondo,de_laat,a.l.varbanescu}@uva.nl

Abstract—Memory bandwidth is critical for many modern applications and architectures. Parallel memories should alleviate this problem, but they are difficult to design and implement for non-trivial access patterns. Our work addresses these challenges through PolyMem, a Polymorphic Parallel Memory which acts as a software cache and enables parallel memory accesses for different combinations of access patterns. PolyMem’s design is based on the Polymorphic Register File (PRF) design, which guarantees conflict-free parallel access for memory access patterns widely used in scientific and multimedia applications.

We present in detail the design and implementation of MAX-PolyMem, the first native implementation of PolyMem using Maxeler’s toolchain. Our design supports multiple lanes, multiple read ports, and concurrent read and write operations.

We further provide a detailed empirical analysis of the performance of MAX-PolyMem, including two sets of results. First, we conduct a thorough design space exploration to determine the best configurations and/or the performance bounds of MAX-PolyMem. For example, the design with the maximum read bandwidth is a 512KB memory, with 4 read ports, running at 137MHz, which can reach a peak read bandwidth of around 32GB/s. Second, to determine whether the MAX-PolyMem can reach and sustain these peaks, we implement the STREAM-Copy benchmark. The benchmarking results demonstrate that, in practice, MAX-PolyMem reaches over 99% of the theoretical peak performance.

I. INTRODUCTION

In this era of parallel processing and data-intensive applications, memory bandwidth is a hot commodity. For many applications, ranging from graph processing to machine learning, and from scientific simulations to financial analysis, more bandwidth is likely to amount to better performance. One approach to address this demand for increased bandwidth is to re-think existing memory systems. Newly-emerging technologies [1], [2] hold promise, but their large-scale integration depends on the processor vendors and is, therefore, rather slow. A more viable solution is to develop parallel memories, which could provide an immediate memory bandwidth increase as large as the number of parallel lanes. While this proposal sounds straightforward in theory, many challenges emerge when designing and/or implementing such memories in practice [3]. Efficient data writes, reading the data with a minimum number of accesses and maximum parallelism, and actually using such memories in real applications are only three of these challenges.

To address these challenges, we propose PolyMem, a Polymorphic Parallel Memory. We envision PolyMem as a high-bandwidth, two-dimensional (2D) memory which is used to cache performance-critical data right on the FPGA chip, making use of the existing distributed memory banks (the BRAMs). We chose a 2D address space for PolyMem to allow the programmers to easily place data structures such as vectors and matrices in this smart buffer, thus decreasing the need for complex index computation typically needed for a traditional, linear access memory. Furthermore, using polymorphism, PolyMem not only delivers high performance for the most common two-dimensional access patterns (such as rows, columns, rectangles, or diagonals), but it also enables combining several such patterns in the same application. Finally, by supporting customization of capacity, bandwidth, number of read/write ports, and different parallel access patterns, PolyMem allows the user to configure the parallel memory to fit his/her application.

Figure 1 depicts the envisioned system architecture. The FPGA board, featuring its own high-capacity DRAM memory, is connected to the host CPU through a PCI Express link. PolyMem acts like a high-bandwidth, 2D parallel software cache, able to feed an on-chip application kernel with multiple data elements every clock cycle.

PolyMem is inspired by existing research on the Polymorphic Register File (PRF) [4], [5]. While the PRF was designed as a runtime customizable register file for Single Instruction, Multiple Data (SIMD) co-processors, PolyMem is tailored for FPGA accelerators for High Performance Computing (HPC), which require high bandwidth but do not necessarily implement full-blown SIMD co-processors and their corresponding instruction sets on the reconfigurable fabric. We have selected FPGAs as our target for three reasons: (1) FPGAs are increasingly used for HPC acceleration due to their high energy efficiency and large amount of on-chip computational resources, (2) FPGAs enable PolyMem to be reconfigured depending on the current workload, and (3) current FPGAs feature relatively large amounts of on-chip, distributed, independent memories – i.e., the BRAM blocks – that can be used as parallel memory banks.

To enable a quick design and benefit from a high-level programming abstraction, our first prototype of PolyMem,
called MAX-PolyMem\(^1\), is implemented using Maxeler’s platform and their MaxJ programming model [6]. This choice further enables us to easily integrate this parallel memory into Maxeler applications\(^2\). To thoroughly test the properties and limitations of MAX-PolyMem, we further propose a multi-dimensional Design Space Exploration (DSE) approach, where the capacity, number of lanes, and number of read ports for each PolyMem scheme are empirically evaluated. Our results show that (1) MAX-PolyMem can utilize the entire capacity of on-chip BRAMs, allowing the instantiation of a 4MB parallel memory on the Maxeler Vectis Data Flow Engine (DFE); (2) the maximum bandwidth delivered by the MaxJ design exceeds 32GB/s at a clock frequency of up to 202MHz, and (3) we are able to utilize all the available BRAMs with reasonable logic utilization.

Finally, to determine whether any unexpected bandwidth limitations occur when using MAX-PolyMem in practice, we have designed and implemented the STREAM benchmark [7], [8], which measures the bandwidth of different in-memory array operations. Using the COPY component of STREAM, we measured the bandwidth of a polymorphic memory with 1 read and 1 write port, and found that we achieve over 99% of the calculated peak performance.

In summary, the contributions of this work are the following:

- We introduce PolyMem, a Polymorphic Parallel Memory built using BRAMs as a high-throughput software-cache for FPGAs;
- We present MAX-PolyMem, the first prototype implementation of PolyMem for Maxeler’s Data Flow Engines. We further analyze the productivity of MaxJ for our implementation: we quantify it through a combined metric (lines of code and development time), and qualify it through a set of lessons learned;
- We perform a DSE analysis to show how MAX-PolyMem scales with the number of lanes (up to 32), capacity (up to 4MB), clock frequency (up to 202MHz), and peak bandwidth (above 32GB/s);
- We design a MaxJ framework for the STREAM benchmark; we further implement and synthesize the STREAM-Copy application, and use it to benchmark the actual, achievable MAX-PolyMem bandwidth in practice.

II. BACKGROUND

A. The Polymorphic Register File

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 [5]. 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. In Table I, each multiview scheme (ReRo, ReCo, RoCo and ReTr) supports a combination of at least two conflict-free access patterns.

In this work, we reuse the PRF conflict-free parallel storage techniques and patterns, as well as the polymorphism idea to design PolyMem. 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 \(8 \times 9\) 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. 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 \(p \times q\) elements\(^3\). When implemented in reconfigurable technology, PolyMem allows application-driven customization: its capacity, number of read/write ports, and the number of lanes can be set pre-runtime (or even at runtime using partial reconfiguration), to best support the application needs.

In summary, PolyMem uses the technology developed for the PRF to build a parallel memory (Figure 2) for three reasons: (1) it provides a generic, out-of-the-box solution to implement a parallel memory, thus avoiding error-prone, time-consuming custom memory design; (2) it can be customized for the application at hand; (3) its multiview property allows 2D arrays to be distributed across several BRAMs, enabling runtime parallel data access using multiple, different “shapes” without the need for hardware

\(^{1}\) We use PolyMem to denote the Polymorphic Memory design, and MAX-PolyMem as the Maxeler-based implementation.

\(^{2}\) This integration work is beyond the scope of this paper.

\(^{3}\) In this work, we will use “\(\times\)” to refer to a 2D matrix, and “\(\cdot\)” to denote multiplication.

---

**Table I**

<table>
<thead>
<tr>
<th>PRF Access Scheme</th>
<th>Available Access Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReRo (Rectangle Only)</td>
<td>Rectangle</td>
</tr>
<tr>
<td>ReCo (Rectangle, Column)</td>
<td>Rectangle, Column, Main and secondary Diagonals</td>
</tr>
<tr>
<td>RoCo (Row, Column)</td>
<td>Row, Column, Rectangle</td>
</tr>
<tr>
<td>ReTr (Rectangle, Transposed Rectangle)</td>
<td>Rectangle, Transposed Rectangle</td>
</tr>
</tbody>
</table>

---

**Fig. 2.** PolyMem supported access patterns.
reconfiguration (see Table I). Effectively, with the PRF-based PolyMem, programmers can assume a parallel memory and focus on algorithm optimizations rather than complex data transformations or low-level details.

B. The Maxeler platform

Maxeler builds FPGA boards for High Performance Computing using chips from Xilinx and Intel/Altera. It uses an High Level Synthesis (HLS) language, MaxJ, to describe the hardware. MaxJ adopts the dataflow programming paradigm, where an application is described as a directed graph: each node represents an operation on the data, while the edges represent the flow of data. During the computation, the data is streamed through the FPGA, and the operations are directly applied on the stream. The FPGA board features its own high capacity DRAM which can be used to store application data. However, the latency of this memory is relatively high (typical for off-chip DRAM) and even with multi-channel implementations, the off-chip DRAM bandwidth is limited. MAX-PolyMem is designed as an on-chip cache, aiming to maximize data reuse and minimize access to off-chip DRAM.

As a programming language, MaxJ is based on Java. It contains datatypes and operations useful to describe the dataflow graph of an application. From a MaxJ description, the Maxeler framework generates a dataflow graph that is then translated to a hardware description language (HDL). Finally, using third party tools, the HDL is synthesized and the bitstream required to program the FPGA is generated.

III. DESIGN AND IMPLEMENTATION

In this section, we briefly present our approach for designing PolyMem to fit a given application, and further dive into the implementation of MAX-PolyMem. This implementation is open source, and is available online at [9].

A. End-to-end design

A great advantage of PolyMem is its ability to be configured to fit the needs of given applications. A configuration consists of a storage capacity C (e.g., 512KB), distributed in $p \times q$ memory lanes, a PRF access scheme, and the number of read ports. The access scheme supports up to four parallel-access patterns (out of the six supported - see II-A), each of which is a dense access to $p$ parallel-access patterns (out of the six supported - see II-read ports. The access scheme enables support for up to four memory lanes, a PRF access scheme, and the number of a storage capacity $C$ to fit the needs of given applications. A configuration consists of datatypes and operations useful to describe the chip DRAM.

A. End-to-end design

A great advantage of PolyMem is its ability to be configured to fit the needs of given applications. A configuration consists of a storage capacity C (e.g., 512KB), distributed in $p \times q$ memory lanes, a PRF access scheme, and the number of read ports. The access scheme supports up to four parallel-access patterns (out of the six supported - see II-A), each of which is a dense access to $p \cdot q$ elements. To customize PolyMem for a given application, we start from the application memory access pattern, for which we find the optimal parallel access schedule - i.e., the best sequence of parallel accesses to the application data - for each potential configuration (scheme, capacity, lanes). To determine the optimal schedule we formulate the problem as a set covering problem [10], using Integer Linear Programming (ILP) for the search itself. We finally select the best configuration based on two metrics: speedup and efficiency. More details on this process are presented in [11].

B. From PolyMem to MAX-PolyMem

Figure 3 shows a diagram describing MAX-PolyMem, our MaxJ PolyMem implementation; we further refer to blocks in this Figure in bold and to signals with a spaced-out font.
The Module Assignment Function (MAF) is a mathematical function that maps each element in the 2D address space to one of the Memory Banks. The MAF guarantees conflict-free access to the supported access patterns. In this work, we use the five MAFs listed in Table I and described in detail in [5]. M implements all the MAFs supported by our design and outputs the select signal in the three types of Shuffles: Read Data Shuffle, Address Shuffle, and Write Data Shuffle. The Shuffles are implemented using full crosstabs and are used to reorder input and output data according to the MAF used. The Addressing Function A computes, for each accessed element, the intra-memory bank address. The Read/Write Data Shuffle and Address Shuffle reorder the data elements and their corresponding intra-memory bank addresses (generated by the A) so that each memory bank receives the correct address and input data.

For each access to PolyMem, the input signals and data flow through all the blocks of the design in Figure 3, top to bottom. Both the DataIn and DataOut are arranged in our predefined order (left to right, top to bottom) to ensure consistency between reads and writes. When writing to PolyMem, the Memory Banks store each input element into the assigned memory module at the corresponding intra-memory module address. More specifically, the input data - DataIn - is written in the memory locations identified by A and M, after they have been reordered by the Write Data Shuffle. During a read access, the output of the Memory Banks, containing the accessed data, is reordered by the Read Data Shuffle. If the WriteEnable signal is low the DataIn elements are ignored. Simultaneous reads and writes are supported because of the independent read and write ports, and our design supports multiple read ports.

We note that our design is implemented using two types of Shuffles. Given a reordering signal, the regular Shuffle reorders the elements, while the Inverse Shuffle, with the same reordering signal, restores the initial order. In this design, therefore, the Write Data Shuffle is implemented using an inverse Shuffle, while the Read Data Shuffle is implemented using a regular Shuffle.

C. Productivity Analysis

One of the reasons for using Maxeler’s platform for this work was the alleged ease-of-use of the MaxJ toolchain. We reflect here on our development process and analyze, qualitatively and quantitatively, the productivity of MaxJ. The development process started by implementing each module in Figure 3 in isolation. Table II illustrates the implementation effort (in days) taken by each module, as well as the required LOC (lines of code). In our experience, Maxeler’s toolchain does enable fast prototyping: it takes little effort to have a simple kernel running on a Maxeler board, due to the Java-like language and the integrated behavioural simulator.

Once all kernels were available, we created a modular multikernel design, using a custom manager to connect the different modules. This approach helped testing and debugging. We found that the lack of a graphical representation of the blocks in a design forces the developer to programmatically link the modules, a time-consuming and error-prone process; furthermore, some of the toolchain errors are not documented: we had problems with the PCI-express interface, the file management in the IDE, and several simulator crashing/hanging instances. With all these issues, the integration took 5 days.

We further explored the trade-off between modularity and performance: we implemented a fused, single-kernel implementation (which took 7 days) and compared the two versions. We found that the modular version consumes twice as many resources, mainly due to the additional inter-kernel communication infrastructure.

Aiming to further optimize the code, we ran into the real challenge of most HLS approaches: low-level details of the implementation are hidden within layers of abstractions and tools, and low-level optimizations are difficult to integrate.

Overall, we find that Maxeler’s toolchain is an asset during the first development stages of an application because (1) MaxJ’s HLS approach hides most of the complexity of hardware design, (2) the behavioral simulator from the MaxIDE saves time during implementation and debugging, and (3) the design can be written with very few lines of code and it is easily readable. On the downside, more documentation and tool support are needed to fine-tune and optimize non-trivial applications. Moreover, the integration of multiple kernels into a single design is complex due to the lack of visualization tools, and MaxJ makes it difficult to fine-tune low-level behavior.

IV. DESIGN SPACE EXPLORATION AND RESULTS

We analyze the performance of MAX-PolyMem through DSE, reporting memory bandwidth (see IV-B) and resource utilization (see IV-C).

A. Design Space Exploration setup

For this study, we have selected three relevant parameters for the design space exploration, listed and explained in Table III. For all experiments in this paper we use a Maxeler Vectis board that uses a Xilinx Virtex-6 SX475T FPGA featuring 475k logic cells and 4MB of on-chip BRAMs. All our experiments configure PolyMem for a data width of 64 bits. Our design is easily configurable: a simple configuration file sets, at compile time, the required DSE parameters. We collected information regarding the FPGA resource usage and the clock frequency for each configuration. We have further computed the maximum read and write bandwidth that can be achieved. We validate each design with a simple read/write cycle: the host fills MAX-PolyMem with unique numerical values, and then reads them back using parallel accesses. The remainder of this section focuses on the detailed analysis of these results.

**TABLE III**

<table>
<thead>
<tr>
<th>DSE Parameter</th>
<th>Values</th>
<th>Explanation / Affected blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Size [KB]</td>
<td>512, 1024, 2048, 4096</td>
<td>The number and capacity of each Memory Bank</td>
</tr>
<tr>
<td>Number of lanes ((p \times q))</td>
<td>8 (2 × 4), 16 (2 × 8)</td>
<td>Number of data elements delivered for each port per clock cycle. Affects each block of the design.</td>
</tr>
<tr>
<td>Number of Read Ports</td>
<td>1, 2, 3, 4</td>
<td>Number of independent data blocks, (p \cdot q) elements each, which can be read in each clock cycle. Affects the aggregate PolyMem bandwidth and the number and capacity of each Memory Bank</td>
</tr>
</tbody>
</table>

**TABLE IV**

<table>
<thead>
<tr>
<th>Size</th>
<th>MAX-PolyMem Maximum Clock Frequencies [MHz].</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ReO</td>
</tr>
<tr>
<td></td>
<td>512KB</td>
</tr>
<tr>
<td>Read Ports</td>
<td>123</td>
</tr>
<tr>
<td>Capacity (KB), Number of Lanes, Number of Read Ports</td>
<td>ReO</td>
</tr>
<tr>
<td>Bandwidth per port (GB/s)</td>
<td>0</td>
</tr>
</tbody>
</table>

**B. Memory Performance**

In its role as a parallel memory, the most important performance metric for MAX-PolyMem is memory bandwidth. We compute the maximum bandwidth assuming all accesses use the full width of the memory. The main parameters influencing the bandwidth are: design clock frequency, which varies depending on the MAX-PolyMem parameters (see Table IV), the number of lanes, and the number of read ports.

Table IV lists the maximum clock frequencies achieved by our designs. The highest frequency, 202MHz, is achieved by the 512KB, 8-lane, single read port ReO design. For the multi-view schemes, the highest clock frequency is 196MHz for the 512KB, 8-lane, single read port ReCo configuration. The minimum clock frequency is 77MHz.

Figure 4 presents the maximum achievable bandwidth per single port, which is also the write bandwidth of our designs. The peak write bandwidth for the 16-lane configurations exceeds 22GB/s for the 512KB, 16-lane, ReO configuration. For the multiview schemes, the maximum achieved bandwidth is 20GB/s for the ReRo configuration. Moreover, we note that single-port bandwidth scales linearly when doubling number of memory banks from 8 to 16.

Figure 5 illustrates the maximum read bandwidth when increasing the number of read ports. The peak bandwidth is 32GB/s achieved by the 512KB, 8-lane, 4-port ReTr scheme. For the 8-lane configurations, we observe good bandwidth scaling when doubling the number of ports from 1 to 2 ports, and diminishing returns for the 3- and 4-port configurations. If the number of lanes is increased to 16, having 2 read ports does not significantly increase the bandwidth. We also note that bandwidth is reduced if the number of lanes and ports is kept constant, but the capacity of PolyMem is increased. This is most likely due to the additional pressure put on the synthesis tools to place and route all the additional BRAMs.

Please note that for the applications that utilize the read and write ports simultaneously, the total bandwidth delivered PolyMem data rate is the sum of the bandwidth delivered by all individual read and write ports.

**C. Resource utilization**

We continue by analyzing the Maxeler Vectis DFE synthesis results in terms of resource utilization. Specifically, we investigated logic, LUT, and BRAM utilization (Figures 6, 7, and 8, respectively).

The results indicate that when increasing the PolyMem capacity but keeping the number of lanes and ports constant, there is little to no increase in logic utilization for any of the target memory schemes. For example, MAX-PolyMem with 8 lanes and a single read port, the logic utilization varies...
The expected one since increasing the number of read ports
polyMem uses 19.31% and the 8-lane, dual read port
8-lane configuration utilizes 16.07% of the BRAMs, the 16-
example, for the single read port ReRo, 512KB design, the
and read ports leads to an increased BRAM utilization. For
Mem capacity and increasing the number of PolyMem lanes
fluence on the amount of BRAMs used. Increasing the Poly-
ports PolyMem. As expected, the memory scheme has no in-
varies from around 16% for a 512KB, 8-lane, 1-read
the LUTs utilization varying between 7% and 28%.
similar trends to the logic utilization described above, with
When doubling the lanes count from 8 to 16, we observe
a supra-linear logic utilization increase. For example, for
the 512KB, single read port, ReRo PolyMem, the logic
utilization increases from 10.78% to 23.73%. This can be
attributed to the quadratic increase in resource used by
the full crossbar in relation to the number of lanes [5].
Figure 7 illustrates the LUTs utilization. We observe here
similar trends to the logic utilization described above, with
the LUTs utilization varying between 7% and 28%.
Finally, Figure 8 illustrates the BRAM utilization, which
varies from around 16% for a 512KB, 8-lane, 1-read port
PolyMem up to 97% for a 2MB, 16-lane, 2-read ports PolyMem. As expected, the memory scheme has no in-
fluence on the amount of BRAMs used. Increasing the Poly-
Mem capacity and increasing the number of PolyMem lanes
and read ports leads to an increased BRAM utilization. For example, for the single read port ReRo, 512KB design, the
8-lane configuration utilizes 16.07% of the BRAMs, the 16-
lane PolyMem uses 19.31% and the 8-lane, dual read port
configuration uses 29.04% of the BRAMs. This behavior is
the expected one since increasing the number of read ports
involved duplicating data in BRAMs.
In summary, we make the following observations:
- MAX-PolyMem is able to utilize the entire capacity of
on-chip BRAMs, allowing the instantiation of a 4MB parallel memory on the Maxeler Vectis DFE while
keeping the logic utilization under 38% and LUTs usage
under 28%;
- Supra-linear logic and LUTs increase when doubling
the number of lanes;
- MAX-PolyMem delivers up to 22GB/s write bandwidth
and up to 32GB/s aggregated read bandwidth using up
to 4 read ports, at a clock frequency of up to 202MHz.

V. STREAM-COPY: BANDWIDTH BENCHMARKING
This section focuses on the empirical evaluation of Poly-
Mem’s performance in practice. We aim to demonstrate
that our implementation has a measured throughput in line
with the estimated values presented in Section IV. For this
analysis, we have used the STREAM benchmark [7], [8],
a well-known tool for memory bandwidth estimation in
modern computing systems.
The STREAM benchmark contains four applications: Copy,
Scale, Sum, and Triad. The benchmark uses three vectors -
A, B and C - in all its applications. The Copy application
performs a vector copy operation \( c(i) = a(i) \), which involves
one read and one write for each element copied. The Scale
application performs the scaling of a vector and stores its
result in another vector \( a(i) = q \cdot b(i) \); thus performing two
memory accesses (a read and a write) and one floating point
multiplication per element processed. The Sum application
performs the sum of two vectors, \( a(i) = b(i) + c(i) \), featuring
two read, one write, and a floating point addition per element.
Finally, the Triad application is a combination of the Scale
and Sum, \( a(i) = b(i) + q \cdot c(i) \), thus featuring two reads, one
write, and the two floating point operations, a multiplication
and an addition.
To use STREAM for the assessment of MAX-PolyMem,
we must design the STREAM framework using Maxeler’s
toolchain and MAX-PolyMem. A high-level view of our
design, which is open-source and available online [12], is
presented in Figure 9. The host is connected through the
PCI-e to our STREAM design, and starts the computation
by sending the Vector Sizes and Mode parameters to
define the behavior of the Controller. The Controller
generates the write and read signals for MAX-PolyMem
selects the correct input for MAX-PolyMem’s write port by
driving the two MUXs. The signals \( Wi, Wj \) and \( Wshape \)
and \( Ri, Rj, Rshape \) signals identify the write/read locations
for the elements to be stored/retrieved from PolyMem.
Lastly, using the DEMUX, the controller selects the right
output stream (from A_OUT, B_OUT, C_OUT) to correctly retrieve the data from the PolyMem.

All the results we present further in this section are obtained using the STREAM-Copy application, which enables us to measure the achieved aggregated bandwidth for a design with 1 read and 1 write port, and report them using the standard reporting of the STREAM benchmark itself.

For measurements, we split the design in three separate stages: Load, Offload, and Copy. The current stage is specified by the host through the Mode signal. During the Load stage, the three vectors (A, B and C) are loaded into PolyMem, which is split in three (equally-sized) regions, to store each of these arrays. The Controller makes sure each array is written in its own space. In the second stage, Copy, the elements contained in vector A are copied in vector C. The parallel read and write operations can happen in simultaneously: the controller selects the feedback loop from the output port of PolyMem. The delay introduced by the read operation (i.e., its latency), is taken into account by our design, ensuring that the controller’s inputs to PolyMem are correctly aligned with the output of the parallel memory. The required delay applied on the output data is 14 clock cycles (estimated by Maxeler’s tools). Finally, in the last stage, Offload, the host retrieves the data from the PolyMem(A, B and C) using three separate streams.

Each of these stages is ran in isolation, orchestrated by the host. The use of blocking calls ensures the separation between stages, also enabling a clear separation between the stages’ execution times. Our focus is on the accurate measurement of the Copy stage, which represents the actual STREAM-Copy application, and is used to benchmark MAX-PolyMem’s bandwidth.

For our experiments, we synthesized this design using a PolyMem with 8 lanes ($p \times q = 2 \times 4$). Because we access data in rows only, we have used the RoCo scheme. All arrays use 64-bit elements. The maximum allocated size for each array is $170 \times 512 \times 8$ bytes, which amounts to around 700KB. This limitation is due to the STREAM design, using 2 read ports, which translates to 2MB of storage effectively available. However, because STREAM-Copy only uses one read port, the design was optimized at synthesis - i.e., its complexity was reduced to that of a single read port design. Thus, we were able to synthesize this STREAM-Copy design with one read and one write ports at 120MHz, just 2 MHz lower than the maximum clock frequency for a 2048KB configuration with a single read port listed in Table IV.

Figure 10 shows the combined read/write throughput we measured with the Copy application, without the data transfers - which happen in separate stages and whose execution time does not contribute to our measurements. The reported data are obtained by measuring 1000 runs of the copy operation, to ensure sufficient measurement resolution and to limit the impact of the minimum overhead of the host-FPGA signal communication. This minimum overhead is, according to our dedicated measurements, around 300ns, and interferes with any measurements of applications with comparable runtimes. This effect is visible on the left side of the graph of Figure 10, before the memory reaches its sustained bandwidth.

As for the theoretical peak of this memory, we have 2 ports, each with 8 memory lanes, each lane being 64-bit wide. Thus, the aggregated (read + write) theoretical bandwidth of this copying operation is $2 \times 8 \times 8 \times 120 = 1360$ MB/s.

The maximum measured throughput we obtained from our STREAM Copy benchmark is 15301 MB/s, which represents more than 99% of the theoretical bandwidth. We conclude that our STREAM implementation validates the peak performance of MAX-PolyMem, demonstrating very little overhead when using the memory in practice.

VI. RELATED WORK

Building a memory hierarchy for FPGA kernels is recognized as a difficult, error-prone task [13], [14]. For example, [14], [15], [16], [17], [18] focus on the design of generic, traditional caches. By comparison, our work proposes a parallel, polymorphic memory which 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.

Application-specific caches have also been investigated [19], [20], [15], though none of these are polymorphic or parallel. Of special interest to this work is [21], where 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 [5], which is more flexible and performs better than alternative memory systems [22], [23], [24], [25]; its high performance in scientific applications has also been proven for practical applications [4], [26], [27].

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, to the best of our knowledge, is the first to support polymorphic parallel accesses through a single, multi-view, application-specific software cache. Moreover, MAX-PolyMem is the first prototype of a parallel software cache written entirely in MaxJ, and targeted at Maxeler DFEs.

VII. CONCLUSION AND FUTURE WORK

This work focuses on performance improvement of FPGA-accelerated applications through increased memory-system parallelism. In this context, PolyMem is an easily configurable, 2D multi-bank software caching mechanism which provides both performance - by combining BRAMs, multi-view parallel data accesses, and concurrent read and write operations - with the flexibility of read/write operations. Due to its multi-view parallel accesses, PolyMem enables applications with dense and/or sparse memory access patterns to benefit from memory-system parallelism.

We have implemented our prototype on the Maxeler platform, using MaxJ. Thus, MAX-PolyMem is a high bandwidth parallel caching mechanism, fully implemented in MaxJ, for Maxeler’s DFEs. As such, it can be directly integrated in other MaxJ designs which require a parallel memory, as proven by our STREAM implementation.

We have tested the limits of our prototype on the Maxeler Vectis DFE board. Our results show that the design can utilize the entire capacity of on-chip BRAMs, and parallel memories up to 4MB, featuring up to 16-lanes, and/or supporting up to 4 read ports are feasible. Assuming dense access patterns, MAX-PolyMem’s estimated peak bandwidth is up to 22GB/s for writes and above 32GB/s for reads. Finally, using a MaxJ implementation of the STREAM-Copy benchmark, we were able to confirm that MAX-PolyMem can achieve more than 99% of the estimated aggregated (read+write) peak bandwidth in practice.

In the near future, we plan to further improve our design. Additionally, we will finalize the implementation of STREAM and use it for more in-depth analysis of MAX-PolyMem’s performance. We also work to provide a proof-of-concept, systematic use of MAX-PolyMem for more complex applications. Our ultimate goal is to provide an HLS toolchain that can analyze applications, determine the requirements and configurations for the most suitable PolyMem based configurations, and enable the seamless integration of these high-bandwidth caching mechanisms with the target applications.

ACKNOWLEDGMENTS

This project has received funding from the EU Horizon 2020 research and innovation programme under grant No 671653.

REFERENCES