Designing and building application-centric parallel memories

Giulio Stramondo | Cătălin Bogdan Ciobanu | Cees de Laat | Ana Lucia Varbanescu

Institute of Informatics, University of Amsterdam, Amsterdam, The Netherlands

Correspondence
Giulio Stramondo, Institute of Informatics, University of Amsterdam, Amsterdam, The Netherlands.
Email: g.stramondo@uva.nl

Funding information
EU Horizon 2020 Framework Programme under project EXTRA, Grant/Award Number: 671653

Summary
Memory bandwidth is a critical performance factor for many applications and architectures. Intuitively, a parallel memory could be a good solution for any bandwidth-limited application, yet building application-centric custom parallel memories remains a challenge. In this work, we present a comprehensive approach to tackle this challenge and demonstrate how to systematically design and implement application-centric parallel memories. Specifically, our approach (1) analyzes the application memory access traces to extract parallel accesses, (2) configures our parallel memory for maximum performance, and (3) builds the actual application-centric memory system. We further provide a simple performance prediction model for the constructed memory system. We evaluate our approach with two sets of experiments. First, we demonstrate how our parallel memories provide performance benefits for a broad range of memory access patterns. Second, we prove the feasibility of our approach and validate our performance model by implementing and benchmarking the designed parallel memories using FPGA hardware and a sparse version of the STREAM benchmark.

KEYWORDS
FPGA, high-bandwidth parallel memories, memory access patterns, STREAM benchmark

1 INTRODUCTION

State-of-the-art accelerators featuring in modern heterogeneous systems typically focus on massive parallelism (eg, GPUs), addressing the ever increasing need for computational throughput. Although these accelerators offer a significant increase in memory bandwidth when compared with CPUs, many kernels remain bandwidth-bound. Our work focuses on alleviating this limitation efficiently, reading the data with a minimum number of accesses and maximum parallelism requires both an accurate analysis of the application’s memory access pattern and a flexible parallel memory, able to handle different access patterns efficiently. Therefore, our system approach relies on PolyMem, a polymorphic parallel-memory model with a given set of conflict-free parallel-access shapes. Our approach aims to optimally map the application memory access pattern to the PolyMem shapes (see Figure 1), thus ensuring maximum efficiency and speedup for our application-specific parallel memory.

Our framework for building application-centric parallel memories has four stages: (1) analyze the memory access trace of the given application to extract parallel-memory accesses (Section 2), (2) configure PolyMem to maximize the performance of the memory system for the given application (Sections 3.1-3.2), (3) compute the (close-to-) optimal mapping and scheduling of application concurrent memory accesses to PolyMem accesses (Section 3.3), and (4) implement the actual accelerator (using MAX-PolyMem), also embedding its management into the host code (Section 5.1).

The performance of our parallel memory is assessed using two metrics: speedup against an equivalent sequential memory, and efficiency. Using the achieved efficiency, we provide a simple performance model, calibrated with PolyMem parameters (like maximum bandwidth), that accurately estimates the resulting throughput of the parallel memory. In turn, using this estimate and benchmarking data of the original application, we can further estimate the overall performance gain of the application using the newly built heterogeneous system.

We test the feasibility of our approach with two different experiments. First, we present a feasibility study, which evaluates the performance gain that our parallel memory can provide (Section 4) for different types of applications, characterized by their memory access patterns. We used 512 synthetically generated memory access patterns, with different sparsity levels, and measured the speedup and efficiency for each pattern. Our results demonstrate that, for more than 90% of these synthetic applications, performance gain of at least 10% or more is guaranteed. Second,
we take the validation one step further and benchmark the implementation of the parallel memories designed and built for 19 Sparse STREAM instances: the original (dense) and 18 variants with various sparsity levels (Section 6). We demonstrate how our approach enables a seamless analysis and implementation of these 19 accelerators in hardware (using a Maxeler FPGA board). Our performance measurements (in terms of bandwidth) confirm (1) the expected performance gain in terms of speedup and (2) the accuracy of the model-based estimates for the achieved bandwidth.

In summary, our contribution in this paper, which extends our previous works, is five-fold.

- We present a methodology to analyze and transform application access traces into a (close-to-) optimal sequence of parallel-memory accesses.
- We provide a systematic approach to optimally configure a polymorphic parallel memory (eg, PolyMem) and schedule the set of memory accesses to maximize the performance of the resulting memory system.
- We provide a statistical analysis to determine the performance potential that our parallel memory offers to applications with various memory access patterns.
- We define and validate a model that predicts the performance of our parallel-memory system.
- We present empirical evidence that the designs generated using our approach can be implemented in hardware as parallel-memory accelerators, delivering the predicted performance.

## 2 | PRELIMINARIES AND TERMINOLOGY

### 2.1 | Parallel memories

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

A parallel memory can be implemented using a set of independent memories—referred to as sequential memories or memory banks. The width of the parallel memory is equal to the number of banks used and determines the number of elements that can be accessed in parallel. The capacity of the PM indicates the amount of data that it can store.

A location in a PM is a combination of a memory bank identifier and an in-memory address, which specifies where within a bank a data element is to be found/stored. Formally, the **parallel-memory location** of a data element is $locA[I] = (m_k, addr), k = [0, \ldots, M]$, where $A[I]$ represents an element of the application, see Section 2.2. $m_k$ is the memory bank identifier, $M$ is the width of the PM, and $addr$ is the in-memory address.

This work addresses **non-redundant** parallel memories. These PMs rely on bijective functions to map the coordinates of an application element to its memory location. Non-redundant PMs guarantee data consistency and the complete use of the capacity of their banks by avoiding data replication. However, they restrict the possible parallel accesses: only elements located in different banks can be accessed in parallel (see Section 2.2).

### 2.2 | The application

The term *application* refers to the entity using the PM to access data, eg, a hardware component connected to the PM or a software application interfaced with the PM.

Without loss of generality, we will consider the data of an application to be stored in an array $A$ of $N$ dimensions. Each data element can then be identified by a tuple containing $N$ coordinates $i = (i_0, i_1, \ldots, i_{N-1})$, which are said to be the coordinates of element $A[i] = A[i_0, i_1, \ldots, i_{N-1}]$ in the application space.

An application *memory access* is a read/write operation which accesses $A[I]$. A *concurrent access* is a set of memory accesses, $A[I], j = 1, \ldots, P$, where $P$ is the number of elements that the application can access concurrently at a given step of the computation. An application *memory access trace* is a temporal series of concurrent accesses. Finally, a parallel-memory access is an access to multiple data elements which actually happens in parallel.

Ideally, to maximize the performance of an application, any concurrent access should be performed as a parallel access, happening in one memory cycle. However, when the size of a concurrent access ($P$) is larger than the width of the PM ($M$), a scheduling step is required, to schedule all $P$ accesses on the $M$ memories. Our goal is to systematically minimize the number of parallel accesses for each concurrent access in the application trace. We do so by tweaking both the memory configuration and the scheduling itself.
Parallel-memory configuration

To specify a $M$-wide parallel access to array $A$, stored in the PM, one can explicitly enumerate $M$ addresses $[A[l_0], \ldots, A[l_{M-1}])$ or use an access pattern to describe the relative distances between the elements. The access pattern is expressed as a $M$-wide set of $N$-dimensional offsets, $\{0_{00}, 0_{01}, \ldots, 0_{NN-1}\} - (0_{10}, 0_{11}, \ldots, 0_{MN-1})$. Using a reference address, i.e., $A[l]$, and the access pattern makes it possible to derive all $M$ addresses to be accessed. For example, for a 4-wide access ($M = 4$) in a 2D array ($N = 2$), where the accesses are at the N, E, S, W elements, the access pattern is $\{(-1, 0), (0, -1), (1, 0), (0, 1)\}$. When combining the pattern with a reference address, e.g., $(4, 4)$, we obtain a set of $M$ element coordinates, e.g., $(13, 4), (4, 3), (5, 4), (4, 5))$. We call the operation of instantiating a memory access pattern into a set of addresses based on a reference address resolving the pattern. In Section 3.1, we will use the function $\text{resolve_pattern}(p, a)$, where $p$ is an access pattern and $a$ is a reference address, to indicate this operation.

Conflict-free parallel access. A set of $Q$ memory accesses $[A[l_0], \ldots, A[l_{Q-1}])$ form a parallel-memory access iff they constitutes a conflict-free access, namely, each memory access is done in a different memory bank.

To map the access to an element in application space to a parallel access in PM space, we need to define a mapping function that guarantees $M$-wide conflict-free accesses. Determining the function to use is a key challenge in defining a custom parallel memory.

Memory mapping function. The Memory Mapping Function (MMF) maps an application memory access to its parallel-memory location, i.e., its address on a given data bank: $\text{MMF} : [A[l], M, D[l]] \rightarrow (m, \text{addr}_k), k = [0, \ldots, M)$, where $l = (l_0, l_1, \ldots, l_{M-1})$ are the coordinates of the access in the application space, $M$ is the width of the parallel memory, and $D[l]$ are the sizes of each dimension of the application space array.

We note that due to the restriction that only conflict-free accesses can be parallel accesses, there is a limited set of access patterns that a parallel memory can support. These patterns are an immediate consequence of the MMF.

PM configuration. A PM configuration is the pair $(\text{MMF}, C)$, where $\text{MMF}$ is a mapping function and $C$ is the capacity of the PM.

Customizing a parallel memory entails finding, for a given application, the PM configuration that is able to retrieve the elements at the position specified in the application memory access trace using the minimum number of conflict-free parallel accesses, hence maximizing the bandwidth of the memory system. In the remainder of this paper, we focus on a methodology to customize parallel memories with the right $M$, $C$, and $\text{MMF}$ for a given application, thus building application-centric parallel memories (see Section 3 and further).

Scheduling concurrent accesses

Once the parallel-memory configuration is known, a transformation between the application concurrent accesses and the memory parallel accesses is necessary. We call this transformation scheduling, and note it can be static, i.e., computed pre-runtime, per concurrent access, or dynamic, i.e., computed at runtime. In this work, we assume static scheduling is possible, and the actual schedule is an outcome of our methodology (see Section 3 and further).

3 | METHODOLOGY

In this section, we describe two approaches for scheduling an application access trace onto a set of PM parallel access patterns: (1) a solution to determine the minimum number of PM accesses that cover the application access trace using an ILP-formulation, and (2) a faster alternative, in the form of a heuristic method which trades-off optimality for speed. We conclude this section by presenting our full methodology for building application-centric parallel memories, and a simple predictive model to calculate the performance of the resulting memory system.

3.1 | From concurrent accesses to set covering

We express the problem of scheduling an application access trace onto a set of PM accesses as a particular instance of the set covering NP-complete problem.⁴

Set covering. Given a universe $U$ of $n$ elements, a collection of sets $S = \{S_1, \ldots, S_n\}$, with $S_j \subseteq U$, and a cost function $c : S \rightarrow Q^+$, find a minimum-cost subset of $S$ that covers all elements of $U$.

Optimally solving the set covering⁴ problem requires it to be formulated as an Integer Linear Program (ILP), where variable $x_j = \{0, 1\}$ indicates if set $S_j$ is part of the solution, $c(S_j)$ is the cost of set $S_j$, and the solution is constrained to have for each element $e \in U$ at least one set $S_j : e \in S_j$.

$$\begin{align*}
\text{minimize} & \quad \sum_{S \in S} c(S_j) \cdot x_j \\
\text{subject to} & \quad \sum_{S_j \in S} x_j \geq 1, \forall e \in U.
\end{align*}$$
An optimal schedule of an application access trace on a set of PM parallel accesses can be found by reducing this problem to a set covering one, and leveraging the ILP formulation. Although an application access trace contains a list of application concurrent accesses, we schedule each of those separately. For every application concurrent access, the universe \( U \) is formed by all accesses. From the PM predefined parallel access patterns, we define \( S \) as the collection of all possible parallel accesses in PM (see Algorithm 1). Finally, the solution obtained using an ILP solver, \( S_{\text{min}} \), is a list of sets which optimally cover the concurrent accesses and will be converted back into a sequence of parallel-memory accesses.

Algorithm 1 shows how to generate \( S \), from which the minimal coverage will be extracted. Set \( P \) contains the list of PM conflict-free accesses patterns, and it is obtained from the PM configuration. Set \( A \) contains the coordinates of the application data. Each pair of an application element and an access pattern (i.e., elements from \( A \) and \( P \), respectively) is resolved into a set of coordinates of application elements, \( p_a \), by \( \text{resolve_pattern} \) (see Section 2.1). To map our problem to the ILP formulation above, we need to guarantee that the union of the collection of subsets in \( S \) is equal to the universe \( U \). This is done by removing the elements that are not being accessed in the concurrent access, i.e., the elements in \( A \) but not in \( U \), from the parallel access \( p_a \). The elements of \( S \) will be all these \( S_{p_a} \) sets, for which it holds that \( \bigcup_{S_{p_a} \in S} S_{p_a} = U \).

To solve our original problem, we are interested in finding the minimum collection of sets \( S_{\text{min}} \) such that \( \bigcup_{S_{p_a} \in S_{\text{min}}} S_{p_a} = U \) and \( S_{\text{min}} \subseteq S \), so the cost function will be defined as \( c(S_{p_a}) = 1 \), \( \forall S_{p_a} \in S \). Once \( S, U, c \) are defined, an ILP solver can be used to compute \( S_{\text{min}} \), the minimum collection of sets that covers the universe \( U \).

### 3.2 A heuristic approach

Because ILP is a major bottleneck in our framework, speedwise,\(^*\) we have designed and implemented an alternative, heuristic approach, which trades-off optimality for speed. Our heuristic is based on the work of Vazirani,\(^4\) and the solution is guaranteed to be within an harmonic factor from the optimal solution (extracted with the ILP approach).

The Greedy algorithm we have used to implement the heuristic solution is presented in Algorithm 2. \( E \) is a set used to keep track of the elements still to be covered with a parallel access, and it is initialized with \( U \), the set containing all the elements in the concurrent access. \( S \) contains all parallel accesses from \( A \) for a given PM configuration (Algorithm 1, Section 3.1). In each iteration, the parallel access \( S_{p_a} \in S \), which contains the maximum number of elements that still needs to be covered, is added to the solution, and the elements covered by \( S_{p_a} \) are removed from \( E \). Once all the elements in the application concurrent access have been covered, the algorithm returns the set of parallel access \( S_{h} \) containing the solution.

---

\(^*\) For very small problem sizes, the runtimes of our Heuristic and ILP approaches are comparable. However, even for a problem size as small as a 32 × 32 matrix, the Heuristic becomes 26x faster than ILP, making the use of the ILP prohibitive.
3.3 | The complete approach

Our complete approach is presented in Figure 2. We start from the Application Access Trace, a description of the concurrent accesses in the application, discussed in detail in Section 2.2. The Application Access Trace can be collected from a given source code performing static analysis or polyhedral analysis.\(^5,6\) We test different parallel-memory configurations by providing different Configuration Files to our Memory Simulator.\(^7\) Each Configuration File contains details regarding mapping scheme, number of parallel lanes, and capacity of the parallel memory. The Memory Simulator produces all the available parallel accesses, compatible with the given parallel-memory Configuration File, that cover elements contained in the Application Access Trace. The set of parallel accesses is then given as input to our ILP or Heuristic solver—implemented as described in Sections 3.1 and 3.2. The Solver selects the minimum number of parallel accesses that fully cover the elements in the Application Access Trace, thus producing a Schedule of parallel-memory accesses. The Schedule can then directly be used in the hardware implementation of the application parallel memory.

An important side-effect of our approach is that the information contained in the schedule can further be used to accurately estimate the performance of the generated memory system. Thus, to calculate the achievable average bandwidth of the memory system for the given access trace, we can "penalize" the theoretical bandwidth (ie, assuming that all lanes are fully used) by our efficiency metric:

\[
BW_{\text{real}} = BW_{\text{peak}} \times \text{Efficiency} = \left( \frac{\text{Frequency} \times \text{Bitwidth} \times \text{Lanes}}{\text{Nseq}} \right) \times \frac{\text{Nseq}}{\text{Nelements}}.
\]

Frequency is the frequency the PM is operating at, Bitwidth is the size of each element stored in the PM, and Lanes represents the amount of elements that can be accessed in parallel; Nseq is the number of required sequential accesses and Nelements is the total number of elements accessed by the PM using a Schedule.

4 | EVALUATION

This section describes a statistical analysis, based on simulation results, of the potential benefits of PMs for different types of applications, characterized by their memory access patterns. It further compares the solutions obtained by our heuristic against the optimal solutions produced by the ILP algorithm (see Section 3).

4.1 | Experiment setup

To empirically demonstrate the potential of parallel memories to improve bandwidth and, ultimately, provide speedup over non-parallel solutions even for non-dense memory access patterns, we propose an experiment where we test the PM for a large number of synthetic memory access patterns. We assume that the capacity of the PM is sufficient to contain the application data. For each pattern, we measure both the performance gain and the efficiency of using PMs. This experiment also enables us to compare the two algorithms for scheduling a memory access trace (see Sections 3.2 and 3.1).

Synthetic application concurrent accesses

The set of concurrent accesses—strided—is generated assuming an 8 × 8 data structure and using three parameters: offset, number of reads, and number of skips. The pattern is generated alternating series of reads and series of skips. The offset defines the number of elements to skip from the element [0][0]. The entire set of synthetic concurrent accesses has been generated using 8-by-8 patterns with offset varying from 0 to 7, number of reads varying from 1 to 8, number of skips from 1 to 8. This resulted in a total of 512 application access traces.

PM configurations

The Memory Mapping Functions (MMFs) used in the PM configurations guarantee conflict-free access to the following 2D patterns (see Figure 1): Rectangle, Diagonal, Secondary Diagonal, Row, Column, and Transposed Rectangle. We assumed a memory capacity sufficient to store all...
application data and experimented with a PM width, M, from 2 to 8 (M = 8 is sufficient to allow full rows/columns/diagonals to be read from our synthetic concurrent accesses), and all combinations of the PRF access patterns. In total, we tested 448 different PM configurations.

Evaluation metrics
We introduce two metrics to evaluate how an application benefits from a parallel memory: speedup and efficiency.

Speedup is a measure of the performance gain from using a custom parallel memory, defined as:

\[ \text{Speedup} = \frac{N_{\text{seq}}}{N_{\text{par}}} \]

where \( N_{\text{seq}} \) refers to the number of accesses required using a sequential memory, i.e., equal to the number of elements in the application concurrent access, and \( N_{\text{par}} \) is the number of parallel-memory accesses, obtained using the algorithms in Section 3.

Efficiency is a measure of the “wasted access” when using a custom parallel memory, defined as:

\[ \text{Efficiency} = \frac{N_{\text{seq}}}{N_{\text{ele}}}, \]

where \( N_{\text{ele}} \) is the total number of elements accesses by the PM and it is equal to \( N_{\text{par}} \times M \), where \( M \) is width of PM. We note that efficiency is an indirect measure of the overhead of a parallel memory for a sparse access and can be correlated with the power efficiency of the memory system.

4.2 Results
We have scheduled all 512 synthetic concurrent accesses (Section 4.1) on all 448 memory configurations (Section 4.1) using both the algorithms proposed in Section 3, i.e., ILP and heuristic. To determine whether the custom parallel memories are successful in improving the performance of different applications, we analyze speedup; to determine whether the heuristic algorithm can be used as a replacement of the ILP-based solution, we analyze the observed trade-off between the optimality (by the ILP method) and speed (by the heuristic method).

Figure 3A shows the speedup results, grouped per PM-width. We make the following observations.

- The bottom parts of the plots, indicating low speedups, are very narrow, showing that only very few concurrent accesses did not benefit from using PM. This is correlated to the sparsity of the memory accesses in the concurrent access and the fact that the parallel access patterns we used only allow dense parallel accesses.
- The top parts of the violins, corresponding to high speedups, are also narrow, indicating that only few concurrent accesses can gain maximum speedup. Moreover, the figure also shows that, for odd numbers of memories (3, 5, 7), the occurrence of close-to-maximum and maximum speedup is very rare: in fact, 1-5 patterns, at most, reach the maximum.
- The majority of the concurrent accesses lies in between those two extremes, showing that they gain significant speedup by using PM, but that it is not possible to fully utilize the all memory banks.

We note that our efficiency results (not included due to space limitations) show a similar picture: few (concurrent access, PM configuration) pairs gain maximum or minimum efficiency, while the average efficiency varies between 0.8 for 2 memories and 0.58 for 8 memories.

ILP versus heuristic
Figure 3B presents the speedup results for all concurrent accesses and memory configurations, using the heuristic algorithm instead of the ILP. From the figure, there is little difference in the distribution of the speedups: few configurations at the bottom and at the top, and most configurations in the middle. On average, the heuristic approach gives a speedup of 0.05% below the optimum computed with ILP.

To see in how many cases the difference between the ILP and heuristic approaches is significant, we compute the ratio \( \frac{\text{Speedup}_{\text{HEU}}}{\text{Speedup}_{\text{ILP}}} \) and plot the density distribution of this ratio in Figure 3C. In the large majority of cases, the speedups are similar, with a loss is less than 15%; the worst result obtained by the heuristic is 53% of the optimal speedup for one single configuration. These results indicate that the heuristic algorithm is acceptable as a replacement of the ILP when quick estimation is required.

FIGURE 3 Evaluation of the ILP and Heuristic (HEU) results. A, Violin plot showing the results obtained using the ILP algorithm; B, Violin plot showing the results obtained using the HEU algorithm; C, Density plot of the ratio between the ILP speedup and the HEU speedup.
5 | 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, the specific Maxeler-based version of our memory system. This implementation is open source and is available online at github.com/giuliostramondo/RAW2018.7

5.1 | PolyMem

Our parallel memory is based on PolyMem, a design inspired by the Polymorphic Register File.8 PolyMem is a non-redundant parallel memory, using multiple lanes to enable parallel data access to bi-dimensional data structures, and a specialized hardware module that enables parallelism for multiple access shapes. For example, an 8-lane PolyMem allows reading/writing 8 elements at a time from/to a 2D memory. The access shapes supported by PolyMem, defined as bi-dimensional shapes, are Row, Column, Rectangle, Transposed Rectangle, Main Diagonal, and Secondary Diagonal. Due to its multi-view design,8 PolyMem supports several access schemes. Each access scheme enables the memory to perform memory operations using with multiple, different access shapes without reconfiguration. The access schemes are as follows.

- ReO: Rectangle.
- ReRo: Rectangle, Row, Diagonal, Sec. Diagonal.
- ReCo: Rectangle, Column, Diagonal, Sec. Diagonal.
- RoCo: Row, Column, Rectangle.
- ReTr: Rectangle, Transposed Rectangle.

5.2 | MAX-PolyMem design and implementation

The hardware implementation and performance analysis presented in this work are all based on the Maxeler version of PolyMem, MAX-PolyMem.2 Figure 4 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. Because PolyMem behaves as a 2D memory, parallel application accesses are made using two coordinates, \((i, j)\), and the shape of a parallel access, AccType. DataIn and DataOut represent the data which is written to and read from MAX-PolyMem. The core of MAX-PolyMem's design consists of a 2D array of memories \((p \times q)\) BRAMs), where each bank is identified using two coordinates. These are used to store the data in a distributed manner. In Figure 4, eight such memories are illustrated (M0-M7); these are the Memory Banks, also called memory modules. The number of banks defines the number of data elements which are read/written in parallel per data port, referred to as lanes.

Based on the \((i, j)\) coordinates and the requested access type AccType, the AGU expands the parallel access in its individual components by computing the coordinates of all the accessed elements \((p \times q)\) addresses in total). This operation is performed for the write port and for each read port, so that one write access and one read access for each read port can happen independently at the same time.

The Memory Mapping Function (MMF), defined in Section 2.2, is implemented using two components: Module Assignment Function and Addressing Function. 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 access

![FIGURE 4 The block-diagram of our MAX-PolyMem implementation](image-url)
patterns—and respective MAFs—listed in Section 5.1 and described in detail in the work of Ciobanu. 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 crossbars 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 4, 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.

5.3 | MAX-PolyMem performance

Experimental setup
To analyze the performance of MAX-PolyMem, we propose an extensive experimental setup, by varying the capacity, the number of memory lanes, the schemes, and the number of read ports of the parallel memory (see Table 1). We focus our evaluation on performance, thus memory bandwidth is our metric of choice. For all experiments in this paper, we use a Maxeler Vectis board that uses a Xilinx Virtex-6 SX475T FPGA featuring 475 k logic cells and 4 MB 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 peak read and write bandwidth that can be achieved; this peak performance has been empirically confirmed by actual measurements with the resulting FPGA design.

Performance results
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 2), the number of lanes, and the number of read ports.

Table 2 lists the maximum clock frequencies achieved by the FPGA implementation of our designs. The highest frequency, 202 MHz, is achieved by the 512-KB, 8-lane, single read port ReO design. For the multi-view schemes, the highest clock frequency is 196 MHz for the 512-KB, 8-lane, single read port ReCo configuration. The minimum clock frequency is 77 MHz.
Figure 5A 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 22 GB/s for the 512-KB, 16-lane, ReO configuration. For the multiview schemes, the maximum achieved bandwidth is 20 GB/s for the ReRo configuration. Moreover, we note that single-port bandwidth scales linearly when doubling the number of memory banks from 8 to 16.

Figure 5B illustrates the maximum read bandwidth when increasing the number of read ports. The peak bandwidth is 32 GB/s achieved by the 512-KB, 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 delivered PolyMem data rate is the sum of the bandwidth delivered by all individual read and write ports.

Our results have led to two main observations. First, MAX-PolyMem is able to utilize the entire capacity of on-chip BRAMs, allowing the instantiation of a 4-MB parallel memory on the Maxeler Vectis DFE while keeping the logic utilization under 38% and LUTs usage under 28%. This effectively means that there still is space on the device to implement additional computation if needed. However, we also observe a supra-linear increase of resource utilization when doubling the number of lanes, which indicates that wide memories are less efficient in terms resource utilization. Second, and more relevant for this work, MAX-PolyMem delivers up to 22 GB/s write bandwidth and up to 32 GB/s aggregated read bandwidth using up to 4 read ports, at a clock frequency of up to 202 MHz.

6 | AN EXTENSIVE CASE-STUDY: SPARSE STREAM

We evaluate the feasibility and performance of our complete approach by designing and implementing 19 parallel-memory accelerators on our FPGA-based system (Maxeler Vectis). Our 19 accelerators belong to a suite we call Sparse STREAM, which is a sparse version of the well-known STREAM benchmark. Our evaluation focuses on the memory bandwidth achieved by our parallel memory.

6.1 | Sparse STREAM

To prove the feasibility of our approach, from application access traces to hardware, we adapt the STREAM benchmark, a well-known tool for memory bandwidth estimation in modern computing systems, to support sparse accesses.

The original STREAM benchmark uses three dense vectors—A, B, and C—and proposes four kernels: Copy (C = A), Scale (A = q · B), Sum (A = B + C), and Triad (A = B + q · C). However, the original STREAM does not challenge our approach because it uses dense, regular accesses. We therefore propose Sparse STREAM, an adaptation of STREAM which allows 2D arrays and configurable sparse accesses. Table 3 presents 19 possible variants of Sparse STREAM, labeled based on their read access density. The main difference between these variants is the number of sequential accesses, $N_{seq}$. We also distinguish between regularly strided patterns, generated as explained in Section 4.1, and random patterns generated using a pseudo-random function which can guarantee an average access density of the pattern.

We apply our methodology for each variant. Thus, for each variant, we obtain the (close-to-) optimal schedule per access scheme. The schedule is characterized by the number of parallel accesses $N_{par}$, and the total number of accessed elements $N_{elements}$ (Section 3), from which we calculate speedup and efficiency per access scheme. Because regular memory systems often optimize row-wise accesses, we define a Row Only scheme as our baseline. We further present results for two PolyMem schemes (namely, ReRo and RoCo) in Table 3. We select the best performing PolyMem scheme to test in hardware.

The final step in our approach is implementing the schedule in the hardware of our parallel-memory accelerator.
TABLE 3 The 19 variants of the STREAM benchmark and the predicted performance of the calculated schedules for two schemes (ReRo and RoCo), and the Row Only baseline. Randomly generated patterns can be distinguished from the regular ones by the “r” in their density. The other schemes are omitted because they are not competitive for these patterns.

| Pattern description | Density | \( N_{\text{seq}} \) | \( N_{\text{par}} \) | Row Only | ReRo Scheme | RoCo Scheme | Selected Scheme |
|---------------------|---------|---------------------|---------------------|----------|-------------|-------------|-----------------|
|                     |         | Speedup | Efficiency | \( N_{\text{par}} \) | Speedup | Efficiency | Speedup | Efficiency | |
| 20                  | 17408   | 2.00    | 25.00     | 4369     | 3.98       | 49.81      | 4369            | 3.98       | 49.81 | ReRo |
| 20r                 | 18148   | 2.39    | 29.84     | 6252     | 2.90       | 36.28      | 6508            | 2.79       | 34.86 | ReCo |
| 25                  | 21760   | 2.00    | 25.00     | 10880    | 2.00       | 25.00      | 2816            | 7.73       | 96.59 | ReCo |
| 25r                 | 22205   | 2.67    | 33.38     | 7020     | 3.16       | 39.54      | 7244            | 3.07       | 38.32 | ReRo |
| 33                  | 29013   | 2.99    | 37.43     | 3724     | 5.92       | 74.01      | 9671            | 3.00       | 37.50 | ReRo |
| 33r                 | 29436   | 3.13    | 39.18     | 8181     | 3.60       | 44.98      | 8411            | 3.50       | 43.75 | ReRo |
| 40                  | 34816   | 3.98    | 49.81     | 8687     | 4.01       | 50.10      | 8687            | 4.01       | 50.10 | ReRo |
| 40r                 | 35234   | 3.53    | 44.13     | 8989     | 3.92       | 49.00      | 9183            | 3.84       | 47.96 | ReRo |
| 50                  | 43519   | 4.00    | 50.00     | 10880    | 4.00       | 50.00      | 5504            | 7.91       | 98.83 | RoCo |
| 50r                 | 44150   | 4.05    | 50.58     | 10073    | 4.38       | 54.79      | 10202           | 4.33       | 54.09 | ReRo |
| 60                  | 52224   | 5.95    | 74.42     | 8821     | 5.92       | 74.01      | 8821            | 5.92       | 74.01 | Row  |
| 60r                 | 52760   | 4.60    | 57.50     | 10880    | 4.85       | 60.68      | 10965           | 4.81       | 60.15 | ReRo |
| 66                  | 58026   | 5.95    | 74.42     | 7350     | 7.89       | 98.68      | 9710            | 5.98       | 74.70 | ReRo |
| 66r                 | 57431   | 4.86    | 60.72     | 11276    | 5.09       | 63.67      | 11343           | 5.06       | 63.29 | ReRo |
| 75                  | 65279   | 6.00    | 75.00     | 10880    | 6.00       | 75.00      | 8192            | 7.97       | 99.61 | RoCo |
| 75r                 | 65395   | 5.39    | 67.39     | 11680    | 5.60       | 69.99      | 11730           | 5.58       | 69.69 | ReRo |
| 80                  | 69632   | 7.91    | 98.84     | 8806     | 7.91       | 98.84      | 8806            | 7.91       | 98.84 | Row  |
| 80r                 | 69618   | 5.67    | 70.82     | 11879    | 5.86       | 73.26      | 11979           | 5.81       | 72.65 | ReRo |
| 100                 | 87040   | 8.00    | 100.00    | 10880    | 8.00       | 100.00     | 10880           | 8.00       | 100.00 | Row |

FIGURE 6 The implementation of the STREAM benchmark for MAX-PolyMem (figure updated from our previous work). All transfers between host (the CPU) and PolyMem (on the FPGA) are done via the PCIe link.

6.2 Sparse STREAM in hardware

We have designed a flexible template for implementing STREAM using MAX-PolyMem. A high-level view of our design is presented in Figure 6. We use this design as a template for our Sparse STREAM accelerators. Thus, the remaining challenge is to enable the controller to orchestrate the parallel-memory operations based on the calculated schedule. Our current prototype stores the schedule, which contains information regarding the required sequence of parallel accesses (coordinates, shape, and mask), in an on-chip Schedule memory. The controller reads, in every clock cycle, one entry from the schedule (coordinates, shape, and mask), and executes the required parallel-memory access. The host can (dynamically) load a schedule in this memory, as soon as such a schedule is available.

6.3 Performance results

We have implemented all 19 STREAM variants in hardware by configuring MAX-PolyMem, for each test-case, with a memory of 4 MB containing 261120 elements (ie, the maximum capacity available fitting the arrays \( A, B, C \) and the schedule memory), and the best scheme (see Table 3). We have measured the performance of each STREAM component and compared it against our bandwidth estimation.

1STREAM for MAX-PolyMem is open-source and available online.
The performance results (measured, predicted, and ideal) for the 10 different variants of the STREAM benchmark. The horizontal lines indicate the theoretical bandwidth of MAX-PolyMem, configured with 8-byte data, 8 lanes, and 2 (for Copy and Scale) or 3 (for Sum or Triad) parallel operations. Running at 100 MHz, MAX-PolyMem can reach up to 12.8 GB/s for 1-operand benchmarks and up to 19.6 GB/s for 2-operand benchmarks. A, 2 Read Ports STREAM Kernels; B, 3 Read Ports STREAM Kernels

We measure the bandwidth of our 19 Sparse STREAM kernels (average over 10000 runs).\(^7\) The results—predicted vs measured—are presented in Figure 7. We make the following observations.

- Our performance model (see Section 3) accurately predicts the performance of the memory system (below 1% error in most cases).
- For 6 out of the 9 regular sparse STREAM variants, we can achieve close to optimal speedup due to our parallel memory being multi-view and polymorphic.
- For almost all of the random sparse STREAM variants, except for 60, 80, and 100, the added flexibility given by the RoCo and ReRo scheme results in performance increase over the Row Only baseline.
- In the case of the random sparse STREAM variants, the performance gap given by using alternative sets of shapes is reduced. This is due to the randomicity of the accesses which prevents the generation of repeated structures, hence averaging the benefits of using one scheme over another.
- For S-25, S-50, and S-75, the scheme selected using our framework outperforms the scheme discarded by our framework, according to Table 3, by 70%, 50%, and 24%, respectively.
- Our STREAM PolyMem design uses only 25.98% of the logic available in the Vectis Maxeler board, leaving over 74% available for increasing the complexity of the application kernel or performing, concurrently, a different task. More information regarding the resource usage is available in our previous work.\(^2\)

Overall, our experiments are successful: we demonstrated that the schedule generated by our approach can be used in real-hardware, and we showed that the measured performance is practically the same with the predicted one.

### 7 RELATED WORK

To the best of our knowledge, the framework presented in this work is the first one that deals with configuration and usage of parallel memories for applications performing sparse concurrent accesses. However, we can identify three research areas related to our work: application access pattern analysis, the design of custom parallel memories, and the generation of memory systems on FPGAs.

Memory access patterns are extremely relevant for overall application performance, and they have been extensively studied in the past. For example, in polyhedral optimization, information regarding the application access patterns is used to reorder the computation and increase the application data locality. Many automated tools are available for this purpose, with Polly,\(^5\) Pluto,\(^12\) and Graphite\(^6\) among the most popular. However, these techniques are specifically designed and most often used to improve the software to match the memory architecture of a given system. We approach the same problem from a different angle, as we aim to optimize the memory system to match the computation.

Parallel memories designed to improve system memory bandwidth have been proposed in research since the 1970s and remain of interest today. Parallel memories which use a set of predefined memory mapping functions to enable parallel accesses in a set of predefined shapes\(^8,13,15\) have improved to better support more shapes, multiple views, and polymorphic access.\(^8\) Similarly, approaches which derive an application-specific mapping function\(^16,17\) have also recently emerged, constantly improving the efficiency and performance of the generated memory systems. The framework presented in this work gives the ability to compare these different memory mapping functions, predict their performance and optimize their use in the case of application performing sparse accesses. The current version uses a polymorphic parallel memory with fixed shapes, to which we add the novel analysis and configuration methodology, and an extensive evaluation. In the near future, we plan to explore a second back-end, based on generating application-specific mapping function, enabling an even finer-grain customization of the memory system.

\(^{7}\)The overhead of uploading/downloading the arrays to PolyMem is not included in these results.
Finally, with the huge increase in popularity of FPGAs through HLS, a lot of research has been invested in building application-specific caches for FPGAs. Although successful, such research does not (yet) address parallel and/or polymorphic memories. Our work is similar in its goal, ie, to make more efficient use of the FPGA BRAM memory by providing a more productive user interface, but we propose a solution based on a 2D scratch-pad memory, with automated, application-specific read/write operations.

In summary, our approach draws ideas from these topics, and it builds upon our previous work but it is the first work to describe in detail, evaluate, implement, and benchmark a complete methodology, from application to hardware, to build application-centric parallel memories on FPGAs.

8 CONCLUSION AND FUTURE WORK

Modern heterogeneous systems currently feature accelerators that offer massive parallelism for compute-intensive applications, but often suffer from memory bandwidth limitations. In this work, we have proposed a solution to tackle these limitation by instantiating and using parallel-memory accelerators. By basing our approach on a highly configurable parallel-memory system, we are able to instantiate application-specific accelerators, which provide both high bandwidth and high efficiency for the specific memory access patterns of the given application.

Our methodology is an application-to-accelerator workflow, which performs the following actions: analyzes the application access trace, configures and builds a custom non-redundant parallel memory (eg, PolyMem), optimized for the kernel of interest, generates the parallel-memory accelerator in hardware, and embeds it in the original host code.

We validated our methodology by analyzing a large set of memory access patterns, with different sparsity levels. Our results demonstrate that parallel memories are not only useful for dense accesses, but that the true gain for PolyMem (due to its multi-view, polymorphic properties) is for sparser accesses, where we can still gain significant speedup at high efficiency. Overall, the results over 500 patterns show at least 10% gain for more than 90% of the cases. Given that the entire methodology is automated, this gain is virtually free of any intervention from the programmer.

We have further proven the feasibility of the approach by generating the hardware accelerators for 10 different instances of Sparse STREAM. We demonstrated that we can instantiate and benchmark all 10 designs in real hardware (ie, a Maxeler FPGA system and the MAX-PolyMem implementation of PolyMem), and our experimental results demonstrate clear bandwidth gains, which closely match our model’s predictions.

Our on-going work focuses on the analysis of more applications. In the near future, we aim to improve/automate the access traces extraction, to design more efficient ways to integrate the parallel-memory accelerator into the host application and to extend the model toward accurate full-application performance prediction.

ACKNOWLEDGMENT

This work was funded by the EU Horizon 2020 Framework Programme under project EXTRA (grant 671653).

ORCID

Giulio Stramondo https://orcid.org/0000-0002-3124-189X

REFERENCES

1. Che S, Boyer M, Meng J, et al. Rodinia: a benchmark suite for heterogeneous computing. In: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC); 2009; Austin, TX.
2. Ciobanu CB, Stramondo G, de Laat C, Varbanescu AL. MAX-PolyMem: high-bandwidth polymorphic parallel memories for DFEs. In: Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW); 2018; Vancouver, Canada.
3. Stramondo G, Ciobanu CB, Varbanescu AL, de Laat C. Towards application-centric parallel memories. In: Proceedings of the European Conference on Parallel Processing (Euro-Par 2018); 2018; Turin, Italy.
4. Vazirani VV. Approximation Algorithms. Berlin, Germany: Springer Science & Business Media; 2013.
5. Grosser T, Zheng H, Aloor R, Simbürger A, Größlinger A, Pouchet L-N. Polly-polyhedral optimization in LLVM. In: Proceedings of the 1st International Workshop on Polyhedral Compilation Techniques (IMPACT 2011); 2011; Chamonix, France.
6. Pop S, Cohen A, Bastoul C, Girbal S, Silber GA, Vasilache N. Graphite: polyhedral analyses and optimizations for GCC. In: Proceedings of the 2006 GCC Developers Summit; 2006; Ottawa, Canada.
7. PolyMem Code. github.com/giuliostramondo/RAW2018
8. Ciobanu C. Customizable Register Files for Multidimensional SIMD Architectures [PhD thesis]. Delft, The Netherlands: TU Delft; 2013.
9. McCalpin JD. A survey of memory bandwidth and machine balance in current high performance computers. IEEE TCCA News 1995;19:25.
10. The STREAM benchmark website. cs.virginia.edu/stream/
11. STREAM PolyMem MaxJ Code. github.com/giuliostramondo/PolyMemStream
12. Bondhugula U, Hartono A, Ramanujam J, Sadjayappan PA. Pluto: a practical and fully automatic polyhedral program optimization system. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’08); 2008; Tucson, AZ.

* This submission is an extended version of our Heteropar’18 paper.
13. Gou C, Kuzmanov G, Gaydadjiev GN. SAMS multi-layout memory: providing multiple views of data to boost SIMD performance. In: Proceedings of the 24th ACM International Conference on Supercomputing (ICS ’10); 2010; Tsukuba, Japan.
14. Harper DT. Block, multistride vector, and FFT accesses in parallel memory systems. IEEE Trans Parallel Distributed Syst. 1991;2(1):43-51.
15. Kuzmanov G, Gaydadjiev G, Vassiliadis S. Multimedia rectangularly addressable memory. IEEE Trans Multimed. 2006;8:315-322.
16. Wang Y, Li P, Zhang P, Zhang C, Cong J. Memory partitioning for multidimensional arrays in high-level synthesis. In: Proceedings of the 50th Annual Design Automation Conference (DAC ’13); 2013; Austin, TX.
17. Yin S, Xie Z, Meng C, Liu L, Wei S. Multibank memory optimization for parallel data access in multiple data arrays. In: Proceedings of the 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD); 2016; Austin, TX.
18. Yang H-J, Fleming K, Winterstein F, Chen Al, Adler M, Emer J. Automatic construction of program-optimized FPGA memory networks. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA ’17); 2017; Monterey, CA.
19. Putnam AR, Bennett D, Dellinger E, Mason J, Sundararajan P. CHIMPS: a high-level compilation flow for hybrid CPU-FPGA architectures. In: Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA ’08); 2008; Monterey, CA.
20. Chung ES, Hoe JC, Mai K. CoRAM: an in-fabric memory architecture for FPGA-based computing. In: Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA’11); 2011; Monterey, CA.

How to cite this article: Stramondo G, Ciobanu CB, de Laat C, Varbanescu AL. Designing and building application-centric parallel memories. Concurrency Computat Pract Exper. 2020;32:e5485. https://doi.org/10.1002/cpe.5485