# Towards Application-Centric Parallel Memories

Giulio Stramondo(✉), Cătălin Bogdan Ciobanu, Ana Lucia Varbanescu, and Cees de Laat

University of Amsterdam, Amsterdam, The Netherlands
{g.stramondo,c.b.ciobanu,a.l.varbanescu,delaat}@uva.nl

**Abstract.** Many applications running on parallel processors and accelerators are bandwidth bound. In this work, we explore the benefits of parallel (scratch-pad) memories to further accelerate such applications. To this end, we propose a comprehensive approach to designing and implementing application-centric parallel memories based on the polymorphic memory-model called PolyMem. Our approach enables the acceleration of a memory-bound region of an application by (1) analyzing the memory access to extract parallel accesses, (2) configuring PolyMem to deliver maximum speed-up for the detected accesses, and (3) building an actual FPGA-based parallel-memory accelerator for this region, with predictable performance. We validate our approach on 10 instances of Sparse-STREAM (a STREAM benchmark adaptation with sparse memory accesses), for which we design and benchmark the corresponding parallel-memory accelerators in hardware. Our results demonstrate that building parallel-memory accelerators is feasible and leads to performance gain, but their efficient integration in heterogeneous platforms remains a challenge.

**Keywords:** Polymorphic parallel memory
Memory bandwidth improvement · Parallel-memory accelerator

## 1 Introduction

Many heterogeneous systems are currently based on massively parallel accelerators (e.g., GPUs), built for compute-heavy applications. Although these accelerators offer significantly larger memory bandwidth than regular CPUs, many kernels using them are bandwidth-bound. New technologies hold promise for further bandwidth gain, but their adoption depends on the processor vendors, and can therefore be slow. Instead, our work addresses the need for increased bandwidth by enabling more parallelism in the memory system. In other words, for bandwidth-bound applications, this work demonstrates how to build heterogeneous platforms using parallel-memory accelerators.

Designing and/or implementing application-specific parallel memories is non-trivial [3]. Writing the data efficiently, reading the data with a minimum number of accesses and maximum parallelism, and using such memories in real applications are significant challenges. In this paper, we describe our comprehensive approach to designing, building, and using parallel-memory application-specific accelerators. Our parallel memory is designed based on PolyMem [5], a polymorphic parallel memory model with a given set of predefined parallel access patterns. Our approach follows four stages: (1) analyze the memory access trace of the given application to extract parallel memory accesses (Sect. 2), (2) configure PolyMem to maximize the performance of the memory system for the given application (Sects. 3.1, 3.2 and 3.3), (3) compute the (close-to-)optimal mapping and scheduling of application concurrent memory accesses to PolyMem accesses (Fig. 1, Sect. 3.4), and (4) implement the actual accelerator (using MAX-PolyMem), also embedding its management into the host code (Sect. 4.1).

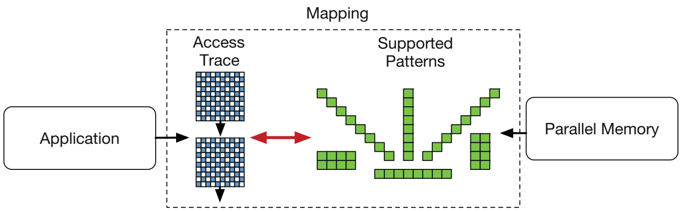

**Fig. 1.** Customizing parallel memories. Our research focuses on the mapping of the access trace from the application to the parallel access patterns of the parallel memory.

The performance of our accelerators is assessed using two metrics: speed-up against an equivalent accelerator with a sequential memory, and efficiency. Blased on a simple, yet accurate model that estimates the bandwidth of the resulting memory system. Using this estimate and benchmarking data, we could further estimate the overall performance gain of the application using the newly built heterogeneous system.

We validate our approach using 10 Sparse STREAM instances: the original (dense) and 9 variants with various sparsity levels (Sect. 4). We demonstrate how our method enables a seamless analysis and implementation of 10 accelerators in hardware (using a Maxeler FPGA board). Finally, using real benchmarking data from the PolyMem-based heterogeneous systems, we validate our performance model.

In summary, our contribution in this paper is four-fold:

- We present a methodology to analyze and transform application access traces into a sequence of parallel memory accesses.
- We provide a systematic approach to optimally configure a polymorphic parallel memory (e.g., PolyMem) and schedule the set of memory accesses to maximize the performance of the resulting memory system.
- 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

In this section we present the terminology and basic definitions necessary to understand the remainder of this work.

### 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 combining a set of independent memories - referred to as *sequential memories*. The *width of the parallel memory*, identified by the number of sequential memories 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 sequential memory hosts the data) and an *in-memory address* (to specify where within that memory the element is stored). We call this pair *the parallel memory location* of the data element. Formally, thus, $loc(A[I]) = (m_k, addr), k = [0..M)$, where $A[I]$ represents an element of the application - see Sect. 2.2, $m_k$ is the memory module identifier, $M$ is the width of the PM, and $addr$ is the in-memory address.

Our approach focuses on *non-redundant* parallel memories. These memories use a one-to-one mapping between the coordinate of an element in the application space and a memory location. Non-redundant parallel memories can use the full capacity of all the memory resources available, and data consistency is guaranteed by avoiding data replication. However, these parallel memories restrict the possible parallel accesses: only elements stored in different memories can be accessed in parallel (see Sect. 2.2).

### 2.2 The Application

We use the term *application* to refer to the entity using the PM to read/write data - e.g., a hardware element directly 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 $I = (i_0, i_1, ..., i_{N-1})$, which are said to be the coordinates of element $A[I] = A[i_0][i_1]...[i_{N-1}]$ 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, $A[I_j], j = 1..P$, which the application can perform concurrently. 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 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.

**Tweaking the Memory Configuration.** To specify a $M$-wide parallel access to array $A$ – stored in the PM –, one can explicitly enumerate $M$ addresses ($A[I_0]...A[I_{M-1}]$), or use an *access pattern*. The access pattern is expressed as a $M$-wide set of $N$-dimensional offsets - i.e.,$\{(o_{0,0}, o_{0,1}, ..., o_{0,N-1}) - (o_{M-1,0}, o_{M-1,1}, ..., o_{M-1,N-1})\}$. Using a reference address - i.e. $A[I]$ - 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, $\{(3, 4), (4, 3), (5, 4), (4, 5)\}$. We call the operation of instantiating a memory access pattern into a set of addresses based on a reference address *resolving the pattern*. In Sect. 3.2 we will use the function *resolve_pattern(p,a)* - where $p$ is an access pattern and $a$ is a reference address - to indicate this operation.

**Definition 2 (Conflict-Free Parallel Access).** *A set of $Q$ memory accesses $A[I_0]..A[I_{Q-1}]$ form a parallel memory access iff it constitutes a conflict-free parallel access, namely:*

$$\forall(A[I_i], A[I_j])$$
$$where\ i \neq j, 0 \leq i, j \leq Q-1, Q = M$$
$$loc(A[I_i]) = (m_i, addr_i), loc(A[I_j]) = (m_j, addr_j)$$
$$m_i \neq m_j.$$

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.

**Definition 3 (Memory Mapping Function).** *The Memory Mapping Function (MMF) maps an application memory access to its parallel memory location.*

$$MMF : (A[I], M, D[I]) \rightarrow (m_k, addr_k), k = [0..M)$$

*where $I = (i_0, i_1, ..., i_{N-1})$ are the coordinates of the access in the application space, $M$ is the width of the parallel memory, and $D[I]$ 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 $MMF$.

A PM configuration is the pair $(MMF, C)$, where $MMF$ is a mapping function and $C$ is the capacity of the PM. Customizing a parallel memory entails finding, for a given application, the configuration that minimizes the number of parallel accesses to the PM.

In the remainder of this paper we focus on a methodology to configure a custom parallel memory with the right $M$, $C$, and $MMF$ for a given application (see Sect. 3 and further).

**Scheduling Concurrent Accesses.** Once the parallel memory configuration is known, the 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 Sect. 3 and further).

## 3 Scheduling an Application Access Trace to a PM

In this section we describe two approaches for scheduling an application access trace using a set of PM parallel access patterns. The first one finds an optimal solution to this problem - the minimum number of PM accesses that cover the application access trace - using ILP. The second one proposes an alternative to ILP, in the form of a heuristic method which trades-off optimality for speed. Finally, we end this section with an overview of our full approach towards application-centric parallel memories and a simple predictive model to calculate the performance of the resulting memory system.

### 3.1 The Set Covering Problem

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 [12].

**Definition 4 (Set Covering [12]).** *Given a universe* $\mathbb{U}$ *of* $n$ *elements, a collection of sets* $\mathbb{S} = \{S_1, ..., S_k\}$*, with* $S_i \subseteq \mathbb{U}$*, and a cost function* $c : \mathbb{S} \rightarrow Q+$*, find a minimum-cost subset of* $\mathbb{S}$ *that covers all elements of* $\mathbb{U}$*.*

The set cover can be formulated as an integer program:

$$\begin{aligned} \text{minimize} \quad & \sum_{S_i \in \mathbb{S}} \mathrm{c}(S_i) \cdot x_{S_i} \\ \text{subject to} \quad & \sum_{S_i : e \in S_i} x_{S_i} \geq 1, \quad e \in \mathbb{U}. \end{aligned}$$

In this formulation, $x_{S_i} = \{0, 1\}$ is a variable indicating if set $S_i$ is part of the solution, $\mathrm{c}(S_i)$ is the cost of set $S_i$, and the solution is constrained to have for each element $e \in \mathbb{U}$ at least one set $S_i : e \in S_i$.

## 3.2 From Concurrent Accesses to Set Covering

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 discussed in the previous section. 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 $\mathbb{U}$ is formed by all accesses. From the PM predefined parallel access patterns, we define $\mathbb{S}$ as the collection of all possible parallel accesses in PM (see Algorithm 1). Finally, the solution obtained using an ILP solver, $\mathbb{S}_{min}, \mathbb{S}_{min} \subseteq \mathbb{S}$, 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.** Generation of the Collection of Sets

```
 1: $\mathbb{S} \leftarrow \varnothing$
 2: $\mathbb{A} \leftarrow$ {all application elements}
 3: $\mathbb{U} \leftarrow$ {all accessed elements}
 4: $\mathbb{P} \leftarrow$ {PM parallel access patterns}
 5: for $p \in \mathbb{P}$ do
 6:     for $a \in \mathbb{A}$ do
 7:         $pa \leftarrow$ resolve_pattern(p, a).
 8:         $S_{pa} \leftarrow pa \cap \mathbb{U}$.
 9:         $\mathbb{S} \leftarrow \mathbb{S} \cup S_{pa}$
10:     end for
11: end for
12: return $\mathbb{S}$.
```

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

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

## 3.3 An Heuristic Approach

As our preliminary results show that ILP is a major bottleneck in our system, speed-wise, we also investigate the possibility to offer an alternative to the ILP formulation for solving the scheduling problem. Therefore, we have designed and implemented a heuristic approach, based on a greedy algorithm (see Algorithm 2). Our heuristic is based on [12], and the solution is guaranteed to be within an harmonic factor from the optimal solution (extracted with the ILP approach).

**Algorithm 2.** Heuristic Application Trace Scheduling

```
 1: $\mathbb{U} \leftarrow$ {all accessed elements}
 2: $\mathbb{S} \leftarrow$ {possible parallel accesses}
 3: $\mathbb{S}_h \leftarrow \varnothing$
 4: $E \leftarrow \mathbb{U}$
 5: while $E \neq \varnothing$ do
 6:     Find $S_{pa} \in \mathbb{S}$ s.t. $|E \backslash S_{pa}|$ is minimum.
 7:     $\mathbb{S}_h \leftarrow \mathbb{S}_h \cup S_{pa}$.
 8:     $E \leftarrow E \backslash S_{pa}$
 9: end while
10: return $\mathbb{S}_h$.
```

Algorithm 2 shows our heuristic approach. $E$ is a set used to keep track of the elements still to be covered with a parallel access, and it is initialized with $\mathbb{U}$, the set containing all the elements in the concurrent access. $\mathbb{S}$ contains all parallel accesses from $\mathbb{A}$ for a given PM configuration (Algorithm 1, Sect. 3.2). In each iteration, the parallel access $S_{pa} \in \mathbb{S}$, which contains the maximum number of elements that still needs to be covered, is added to the solution, and the elements covered by $S_{pa}$ are removed from $E$. Once all the elements in the application concurrent access have been covered, the algorithm returns the set of parallel access $\mathbb{S}_h$ containing the solution.

## 3.4 The Complete Approach

Our complete approach is presented in Fig. 2. We start from the *Application Access Trace*, a description of the concurrent accesses in the application, discussed in detail in Sect. 2.2. We test different parallel memory configuration by providing different *Configuration Files* to our *Memory Simulator*. 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 Sects. 3.2 and 3.3. 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 (i.e., assuming that all lanes are fully used) by our efficiency metric: $BW_{real} = BW_{peak} \times Efficiency = (Frequency * Bitwidth * Lanes) \times \frac{N_{seq}}{N_{elements}}$. $Frequency$ 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; $N_{seq}$ is the number of required sequential accesses and $N_{elements}$ is the total number of elements accessed by the PM using a *Schedule*.

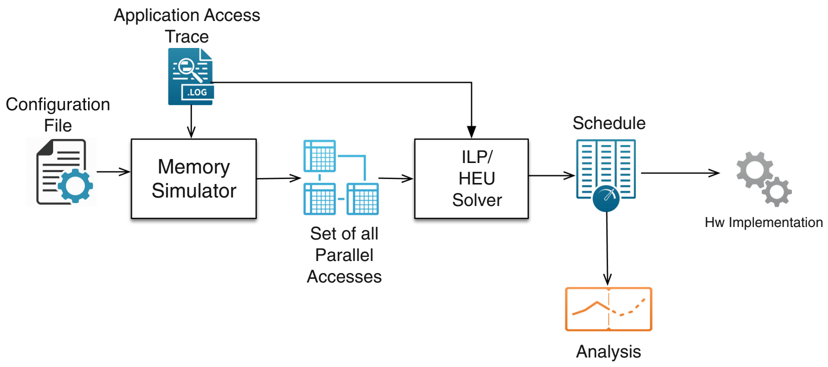

**Fig. 2.** An overview of our complete approach.

## 4 Experiments and Results

We evaluate the feasibility and performance of our approach by designing and implementing 10 parallel-memory accelerators on an FPGA-based system. We use a Maxeler Vectis board, equipped with a Xilinx Virtex-6 SX475T FPGA[1] featuring 475k logic cells and 4 MB of on-chip BRAMs.

### 4.1 MAX-PolyMem

Our parallel memory is based on PolyMem, a design inspired by the polymorphic register file [6]. The hardware implementations and performance analysis presented in this section are all based on the Maxeler version of PolyMem, MAX-PolyMem [5].

[1] Xilinx Virtex-6 Family Overview: http://xilinx.com/support/documentation/data_sheets/ds150.pdf.

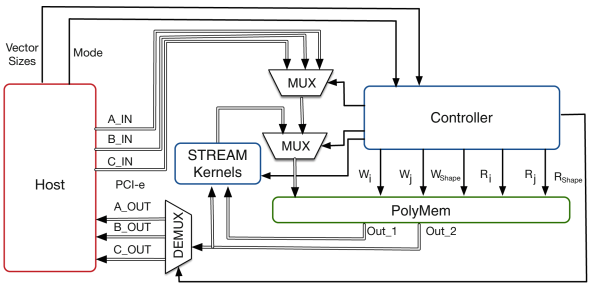

**Fig. 3.** The implementation of the STREAM benchmark for MAX-PolyMem (figure updated from [5]). All transfers between host (the CPU) and PolyMem (on the FPGA) are done via the PCIe link.

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 patterns. 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 [6], PolyMem supports several *access schemes*, i.e, it can perform memory operations with different access patterns without reconfiguration:

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

## 4.2 Sparse STREAM

To prove the feasibility of our approach, from application access traces to hardware, we adapt the STREAM benchmark [2,10], 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 ($A = q \cdot B$), Sum ($A = B + C$), and Triad ($A = B + q \cdot C$).

We have designed a version of STREAM for MAX-PolyMem [5]. A high-level view of our design[2], is presented in Fig. 3.

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 1 presents 10 possible variants of Sparse STREAM, labeled based on their

[2] STREAM for MAX-PolyMem is open-source and available online [1].

**Table 1.** The 10 variants of the STREAM benchmark and the predicted performance of the calculated schedules for two schemes (ReRo and RoCo). The other schemes are omitted because they are not competitive for these patterns. In the patterns, only the `R` elements need to be read.

| Pattern description | | | ReRo Scheme | | | | RoCo Scheme | | | | Selected |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Density | Pattern | $N_{seq}$ | $N_{par}$ | $N_{elements}$ | Speed-up | Efficiency | $N_{par}$ | $N_{elements}$ | Speed-up | Efficiency | Scheme |
| 20 | `RR________RR____` | 17408 | 4369 | 34952 | 3.98 | 49.81 | 4369 | 34952 | 3.98 | 49.81 | ReRo |
| 25 | `R___R___R___R___` | 21760 | 10880 | 87040 | 2.00 | 25.00 | 2816 | 22528 | 7.73 | 96.59 | RoCo |
| 33 | `R__R__R__R__R__R` | 29013 | 3724 | 29792 | 7.79 | 97.39 | 9671 | 77368 | 3.00 | 37.50 | ReRo |
| 40 | `RRRR____RRRR____` | 34816 | 8687 | 69496 | 4.01 | 50.10 | 8687 | 69496 | 4.01 | 50.10 | ReRo |
| 50 | `R_R_R_R_R_R_R_R_` | 43519 | 10880 | 87040 | 4.00 | 50.00 | 5504 | 44032 | 7.91 | 98.83 | RoCo |
| 60 | `RRRRRR____RRRRRR` | 52224 | 8821 | 70568 | 5.92 | 74.01 | 8821 | 70568 | 5.92 | 74.01 | ReRo |
| 66 | `RR_RR_RR_RR_RR_R` | 58026 | 7350 | 58800 | 7.89 | 98.68 | 9710 | 77680 | 5.98 | 74.70 | ReRo |
| 75 | `RRR_RRR_RRR_RRR_` | 65279 | 10880 | 87040 | 6.00 | 75.00 | 8192 | 65536 | 7.97 | 99.61 | RoCo |
| 80 | `RRRRRRRR__RRRRRR` | 69632 | 8806 | 70448 | 7.91 | 98.84 | 8806 | 70448 | 7.91 | 98.84 | ReRo |
| 100 | `RRRRRRRRRRRRRRRR` | 87040 | 10880 | 87040 | 8.00 | 100.00 | 10880 | 87040 | 8.00 | 100.00 | ReRo |

read access density. The main difference between these variants is its number of sequential accesses, $N_{seq}$.

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 $N_{elements}$ (Sect. 3), from which we calculate speed-up and efficiency per access scheme. We present these results for two schemes (namely, ReRo and RoCo) in Table 1. We select the best performing to test in hardware.

The final step in our approach is the translation from a schedule to a hardware implementation of our parallel-memory accelerator. The key challenge is to enable the controller (see Fig. 3) to orchestrate the parallel memory operations based on the given 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`.

## 4.3 Results

We have implemented all 10 STREAM variants in hardware by configuring MAX-PolyMem, for each test-case, with a memory of 261120 elements (i.e., the maximum capacity available fitting the arrays $A, B, C$ and the `schedule memory`), and the best scheme (see Table 1). We have measured the performance of each STREAM component and compared it against our bandwidth estimation.

We measure the bandwidth of our 10 Sparse STREAM kernels (average over 10000 runs)*[3]. The results - predicted vs. measured - are presented in Fig. 4. We make the following observations:

– Our performance model (see Sect. 3) accurately predicts the performance of the memory system (below 1% error in most cases).

[3] The overhead of uploading/downloading the arrays to PolyMem is not included in these results.

- For 6 out of the 9 sparse STREAM variants, we can achieve close to optimal speed-up due to our parallel memory being multi-view and polymorphic.
- For S-25, S-50, and S-75, the performance gain versus choosing the alternative scheme used in this experiment is, according to Table 1, of 70%, 50%, and 24%, respectively.
- Our STREAM PolyMem design uses only 25.98% of the logic available in the Vectis Maxeler board. More information regarding the resource usage is available in [5].

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.

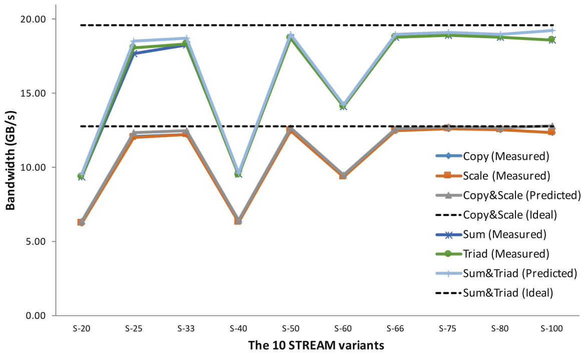

**Fig. 4.** The performance results (measured, predicted, and ideal) for the 10 different variants of the STREAM benchmark. The horizontal lines indicate the theoretical bandwidth of MAX-PolyMem, configured with 8-byte data, 8 lanes, and 2 (for Copy and Scale) or 3 (for Sum or Triad) parallel operations. 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.

## 5 Related Work

Research on using parallel memories to improve system memory bandwidth has started in the 70s, and remains of interest today. Parallel memories that use a set of predefined mapping functions to enable specifically shaped parallel accesses have improved to better support more shapes [7–9], multiple views, and polymorphic access [6]. Approaches that derive an application-specific mapping function [13,15] have also emerged, constantly improving the efficiency and performance of the generated memory systems. The current version of this work uses a polymorphic parallel memory with fixed shapes, to which we add the novel analysis and configuration methodology.

As for building such memories in hardware, a lot of research has been invested in building application-specific caches for FPGAs. Although successful, such research [4,11,14] does not (yet) address parallel and/or polymorphic memories. Our work fills this gap, by showing how to efficiently design a polymorphic, multi-view parallel memory embedded into an FPGA-based accelerator.

## 6 Conclusion and Future Work

Modern accelerators, currently embedded in heterogeneous systems, offer massive parallelism for compute-intensive applications, but often suffer from memory bandwidth limitations. Our work investigates the benefits of building accelerators with application-specific parallel memories as a solution to alleviate this bottleneck. Our approach is especially effective for applications with large sets of concurrent accesses.

To this end, we proposed an end-to-end workflow which analyzes the application access trace, configures and builds a custom non-redundant parallel memory (e.g., PolyMem), optimized for the data-intensive kernel of interest, generates our parallel-memory accelerator in hardware, and embeds it in the original host code.

We have empirically validated our approach using Sparse STREAM with 10 different access densities. We demonstrated that we can instantiate and benchmark all 10 designs in real hardware (i.e., a Maxeler system and the MAX-PolyMem version). Our experimental results demonstrate clear bandwidth gains, and 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, a more efficient integration of the parallel-memory accelerator into the host application, and an extension of the model towards accurate full-application performance prediction.

## References

1. STREAM PolyMem MaxJ Code. https://github.com/giuliostramondo/PolyMemStream
2. The STREAM benchmark website. https://cs.virginia.edu/stream/
3. Budnik, P., Kuck, D.J.: The organization and use of parallel memories. IEEE Trans. Comput. **100**(12), 1566–1569 (1971)
4. 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)
5. Ciobanu, C.B., Stramondo, G., de Laat, C., Varbanescu, A.L.: MAX-PolyMem: high-bandwidth polymorphic parallel memories for DFEs. In: IPDPSW 2018 (RAW 2018) (2018)
6. Ciobanu, C.: Customizable register files for multidimensional SIMD architectures. Ph.D. thesis, Delft University of Technology, Delft, Netherlands, March 2013
7. 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)
8. Harper, D.T.: Block, multistride vector, and FFT accesses in parallel memory systems. IEEE Trans. Parallel Distrib. Syst. **2**(1), 43–51 (1991)

9. Kuzmanov, G., Gaydadjiev, G., Vassiliadis, S.: Multimedia rectangularly addressable memory. IEEE Trans. Multimed. **8**, 315–322 (2006)
10. McCalpin, J.D.: A survey of memory bandwidth and machine balance in current high performance computers. IEEE TCCA Newslett. **19**, 25 (1995)
11. 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)
12. Vazirani, V.V.: Approximation Algorithms. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-662-04565-7
13. 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)
14. 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)
15. Yin, S., Xie, Z., Meng, C., Liu, L., Wei, S.: Multibank memory optimization for parallel data access in multiple data arrays. In: ICCAD 2016, pp. 1–8 (2016)