Designing Intelligent Memory Interfaces for Improving Accelerator-based System Performance

Anonymous Author(s)

ABSTRACT

Domain-specific systems, typically consisting of one or more custom accelerators, improve the execution efficiency of a specific set of applications compared to general-purpose systems. These accelerator designs can be generated using high-level synthesis (HLS) flows to improve productivity and enable a comprehensive design space exploration to meet various energy and performance constraints. HLS tools often ignore the challenges of implementing a complex system of parallel custom accelerators. In particular, the way accelerators access memory is often constrained by the tool to either on-chip memory or external memory, with limited optimization opportunities. From the system-level perspective, efficient data movement between accelerators and memory while reducing the system bus contention is critical to maximizing system performance. This paper proposes the design of a buffering system that improves memory access for HLS-generated accelerators by efficiently employing burst transactions using the AXI4 protocol. We also discuss the improvements that our design brings to HLS methodologies for dealing with external memory accesses.

ACM Reference Format:
Anonymous Author(s). 2023. Designing Intelligent Memory Interfaces for Improving Accelerator-based System Performance. In *Proceedings of International Conference on Supercomputing (ICS 2023).* ACM, New York, NY, USA, 8 pages. https://doi.org/xxxxxxx.xxxxxxx

1 INTRODUCTION

With the end of Dennard scaling, new technology nodes keep increasing the transistor density but fail to improve energy efficiency. Domain-specific accelerators, designed to perform only a subset of recurring functionalities, have become the leading solution to increase performance in tight power constraints. Modern computing systems at all scales include several application-specific accelerators [1, 8, 10, 19, 20, 23] and, in some cases, field programmable gate arrays (FPGAs), which are devices configurable after their deployment, that allow adding specialized accelerators [4, 18, 26].

High-Level Synthesis (HLS) tools [9, 11, 25] allow creating hardware designs in hardware description languages (HDLs) starting from descriptions in high-level languages (e.g., C/C++), significantly reducing the time and effort required to develop custom accelerators, for both FPGAs and application-specific integrated circuits (ASICs) devices.

ICS 2023, June 21–23, 2023, Orlando, FL
© 2023 Association for Computing Machinery.
ACM ISBN xxx-x-xxxx-xxxx-x/xx/xx…$xx.xx
https://doi.org/xxxxxxx.xxxxxxx

Typically, accelerators generated through HLS employ a load-compute-store paradigm [7]. The accelerator loads the initial data required to start the computation from an external memory through a memory channel, performs computation, and stores back data through the same memory channel. Thus, at first, the accelerator is stalled, waiting for data. The memory channel only becomes available for further data movement after the accelerator starts computing.

From the system-level perspective, the ideal scenario is for accelerators to avoid stalling while waiting for data [13, 15] and no contention on the memory channels. When considering complex designs with multiple accelerators connected to the same external memory, their load/store operations may create contention on the memory channels and consequently increase the time the accelerators are stalled. Hence, optimizing management of memory access is critical to maximize overall system efficiency.

One widely used approach to address this issue is to leverage locality and burst memory transfers [12, 17]. Accelerators can, in fact, employ private local buffers to prefetch and store data in bursts (a set of consecutive data in memory). The accelerator can then load or store relevant data with low latency. A burst memory transaction moves more data with a single transaction and requires single handshaking before the actual transfer; thus, it could be larger and slower than performing a load/store operation of a single memory word, but can significantly reduce the number of transactions needed to transfer all the data, providing higher memory efficiency. As transactions grow larger, the overhead for headers and handshaking becomes smaller. HLS tools allow employing burst memory transactions to optimize data transfers but require significant code restructuring and tool-specific pragma annotations to map the code patterns to burst transactions. For example, the Xilinx Vitis HLS manual devotes an entire chapter to optimizations for burst transfers [2]. Additionally, HLS tools typically only optimize memory access for a single accelerator at a time, not considering effects due to multiple accelerators accessing, in parallel, the same shared pool of external memory.

Effectively implementing burst memory accesses can significantly improve the efficiency of accelerator-based systems. Methods to simplify and automate their implementation are highly sought after for hardware generators. The ideal scenario would be to perform a single transaction to move in all input data, perform the computation, and finally move out data in a single transaction. While this requires the accelerator to wait longer to start computing, it reduces transfer overheads and subsequent accelerator stalls by exploiting spatial locality. Crucially, it also frees the memory bus for other transactions, allowing it to overlap computation with communication in multi-accelerator systems or in single-accelerator solutions that can exploit multi-buffering and pipelining. However, this is not always a practical solution, especially for HLS, where accelerators might be complex designs coming from several large and mostly

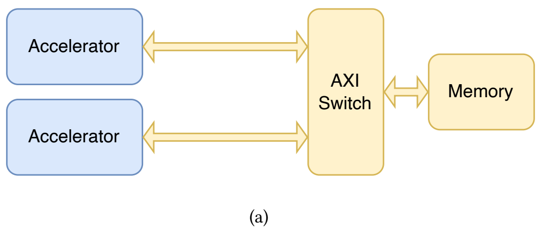

Figure 1: HLS-generated accelerator systems: (a) Baseline, and (b) Proposed design

Table 1: Memory operations and execution cycles distribution of the PolyBench kernels

Kernel	Memory operations			Execution Cycles		
	Read	Write	Total	Total	Computation	Communication
2mm	2,302	200	2,502	31,138	18,828 (60.47%)	12,310 (39.53%)
3mm	3,300	300	3,600	44,433	26,733 (60.16%)	17,700 (38.84%)
atax	112	22	134	2,119	1,471 (69.42%)	648 (30.58%)
bicg	120	20	140	2,083	1,403 (67.35%)	680 (32.65%)
doitgen	12,000	2,000	14,000	161,213	93,213 (57.82%)	68,000 (42.18%)
mvt	240	20	260	3,104	1,824 (58.76%)	1,280 (41.24%)

sequential code structures (often replicated through classical loop optimization techniques to increase instruction-level parallelism). Generating accelerators for such codes would need large buffers to fit all the data required by the computation. As system designs integrate more custom processing elements, FPGAs may need more resources and/or ASICs would need to devote even more area for the necessary on-chip memory.

This paper introduces the design of a novel, parametric buffering system for HLS-generated accelerators. The buffering system is transparent with respect to HLS tools, attaching to a conventional interface generated along the Finite-State Machine with Datapath (FSMD) model. HLS tools, however, can derive parameters for the buffer system through appropriate analysis passes. Specifically, this paper discusses several design tradeoffs for the buffer system to achieve an optimal balance between performance in single and multiple accelerator systems.

In summary, the contributions of this paper are:

- the design of a buffer system to transparently improve memory interfacing in HLS-generated accelerators;
- the evaluation of the design tradeoffs in single and multiple accelerator configurations;
- the considerations for the integration of such buffering mechanism in HLS tools.

The paper proceeds as follows: Section 2 discusses the motivation for our design. Section 3 presents the details of the design. Section 4 presents the experimental evaluation and discusses the various design tradeoffs. Section 5 is about the related work. Finally, Section 6 concludes the paper.

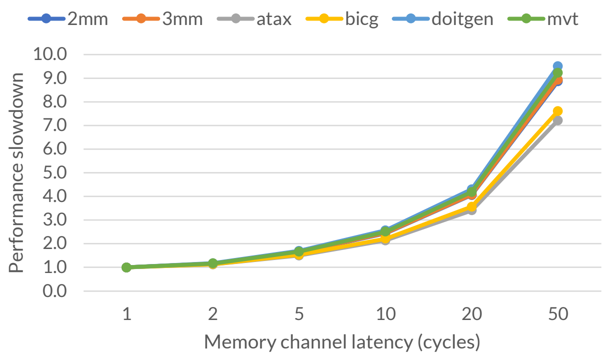

Figure 2: Performance degradation of PolyBench kernels' accelerators with increasing memory channel latency

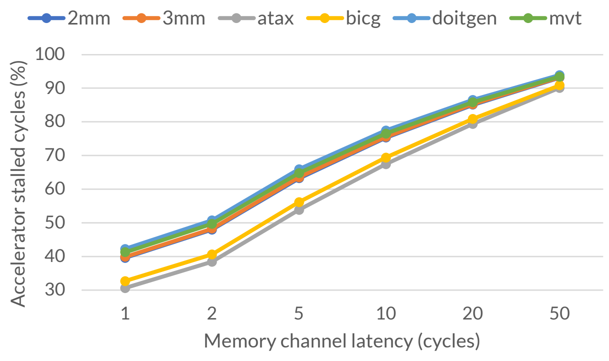

Figure 3: Stalled cycles analysis for PolyBench kernels' accelerators with increasing memory channel latency

2 MOTIVATION

We present a case study highlighting the need for optimizing memory interfaces to improve the performance of HLS-generated accelerators. We synthesized the linear algebra kernels from the PolyBench [21] benchmark suite for the case study. Table 1 gives a brief overview of the synthesized kernels. The *2mm* and *3mm* represent two and three consecutive matrix multiplications, with data dependencies between the operations. The *atax* kernel represents matrix transpose and vector multiplication, while the *mvt* kernel performs matrix-vector product and transpose operation. The *bicg* kernel is the biconjugate gradient sub-kernel of the BiCGStab linear solver. *doitgen* is the multi-resolution analysis kernel. These kernels represent many algorithmic operations found in scientific computing or high-level data science programming frameworks.

We synthesized these kernels using the Bambu [9] HLS tool from the PandA framework. Bambu is an open-source state-of-the-art HLS tool that generates register-transfer level (RTL) designs starting from high-level C/C++ codes or LLVM intermediate representations (IRs). The RTL designs generated by Bambu follow the FSMD model. For the case study, we obtain separate RTL designs (represented in synthesizable Verilog code) for each PolyBench kernel.

Figure 1(a) shows the high-level block diagram of the generic HLS-generated accelerators. Typically, these accelerators employ a load-compute-store paradigm. The accelerator requests the data required for current computation from an external memory using a memory channel. After the computation finishes, it stores the data back into the external memory through the same memory channel. Thus, for the typical system, the memory channel is free when the accelerator is actively computing. Meanwhile, when the memory channel is servicing the data requests, the accelerator is stalled for data. Table 1 shows the memory operations for PolyBench kernels, as well as the distribution of execution cycles. As observed, the system is compute-bound, with accelerators computing for 57.82% (for *doitgen*) – 69.42% (for *atax*) of the total execution cycles. The memory channel is active for only 30.58% – 42.18% of the total execution cycles.

However, these results represent the best-case scenario with single-cycle memory channel latency. The total latency includes both the latency of the interconnect and of the actual memory access. The various components in the memory channel (e.g., AXI switches and network-on-chip routers) can introduce a multi-cycle latency in the memory channel. Thus, in the typical system, the accelerators are stalled longer, waiting for the data as the memory channel latency increases. In our case study, we experimented with a range of memory channel latency to understand the impact on overall system performance.

Figure 2 presents the normalized accelerator performance for a range of memory channel latency. The results in Table 1 (i.e., single-cycle memory channel latency) are used as the baseline in the case study. As expected, we observe performance degradation as the channel latency increases. For example, on average, the accelerators required 2.39× more execution cycles for 10 cycles of latency, while for 50 cycles of latency, the accelerators required 8.56× more execution cycles.

The performance degradation can be attributed to additional stalled cycles for accelerators. Figure 3 presents the results for stalled cycles. For a memory channel latency of 50 cycles, the accelerators are stalled for 90.07% (for *atax*) – 93.92% (for *doitgen*) of the total execution cycles. Figure 3 also presents an additional observation: channel latency of as low as five cycles can convert the compute-bound system to a memory-bound system (i.e., the accelerators spend more time stalled for data than computation cycles). Moreover, in the case of multiple accelerators sharing the memory channel, the multiple load/store operations and the channel latency can drastically reduce the overall system performance.

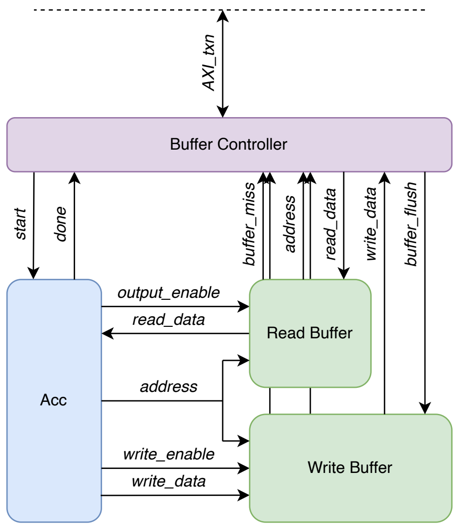

Figure 4: Buffer Design Overview

Thus, optimizing memory interfaces is essential to improve system performance. One widely used solution to address this issue is to decouple the accelerators from the shared memory channel, as shown in Figure 1(b). We can leverage burst memory transfers to prefetch and store the data in private local buffers. The accelerators can access the buffered data with low latency while allowing the shared memory channel to be available for load/store requests from other accelerators. In the following section, we describe the implementation of our efficient memory interface in detail.

3 METHODOLOGY

The main objective of our design is to localize data via dynamic burst prefetching. Our solution aims to be completely transparent to the accelerator, i.e., the proposed buffering system can be attached as a plug-in component to an existing design, regardless of, for example, the granularity or pattern with which the accelerator accesses the external memory. This feature provides several benefits, facilitating:

- integration in existing intellectual property (IP) designs;
- automatic generation/integration with HLS flows;
- design space exploration of different configurations.

This solution is also transparent to the user of the HLS flow. In fact, there is no need to restructure or annotate the code feed to the HLS tool.

Figure 1(a) depicts a generic HLS-generated system, where the custom accelerator interfaces with external memory with a given communication protocol (e.g., AMBA AXI4 [3]). Our buffering components interpose between the accelerator and the memory interface, as in Figure 1(b). In the proposed design, the accelerator keeps issuing memory requests as if directly accessing the external memory. The proposed buffering system intercepts such requests, and either serves the data immediately if the accessed data has been localized (due to an earlier memory request) or triggers burst read/write operations on the external memory.

Figure 4 provides a schematic overview of our buffering system, which consists of two major components: a *buffer controller* and the *buffers* (read-only, write-only, and read-write). All the buffers use on-chip memory (e.g., block RAMs for FPGAs). Each buffer configuration (i.e., capacity and burst size) can be adjusted individually. We present a design space exploration analysis for optimizing the buffer configurations in Section 4.

We analyze the call graph of the specification to generate the buffering system configuration. We allocate one buffer per each function argument bound to external memory. We also differentiate between read-only, write-only, and read-write buffers based on the direction of the data movements. To enrich the specification call graph with data-flow information, we leverage SODA-OPT [4], a compiler that extends the MLIR framework [14]. MLIR enables a compilation flow that builds on progressive lowering and optimizations at the correct abstraction. In this work, we leverage the automatic partitioning and the state-of-the-art optimization pipeline for HLS kernels presented in [4], and implement buffer analysis to capture the direction of the memory operation (i.e., in, out, inout) for every individual function argument, always ensuring that the memory spaces of different arguments are disjoint in the absence of any data dependencies.

Read Operations. The first read on an address space associated with a particular buffer triggers a burst read on external memory. For each subsequent read, the buffer control logic checks if the target address falls in the space of the fetched data. In case of a *hit* (i.e., data has been already fetched), data is returned at on-chip memory latency. In case of a miss, a new batch of data is fetched, starting at the address which caused the miss. Accesses on a miss have the latency cost of a burst transaction to external memory.

Write Operations. Write operations always occur locally, as long as the target address is in the buffer address space boundaries (*hit*). Every time a *miss* occurs, the locally written data must be flushed to external memory. Similarly, local data is flushed to external memory once the accelerator completes execution.

Proposed HLS Flow. Our proposed HLS flow starts with the analisys and optimization of the initial code specification at the frontend. The process generates individual LLVM-IR representations of the accelerated kernels, and a call graph with data-flow information, characterized by both dependencies and direction of data movements with respect to memory. We synthesize each kernel individually with conventional HLS, each with its own memory ports. From the information encoded in the call graph, we allocate read, write, and read/write buffers, which have different control logic and ports. Finally, we assemble the system connecting the accelerators' memory ports to the buffers, and the buffers to the shared bus. We also introduce control logic to trigger the execution of the accelerators based on the dependency information in the call graph, as detailed in [7]. It is important to note that this compositional approach is only possible since our buffering system is transparent to the accelerators, and does not require any modification of the individual kernel accelerators, including their internal control logic.

Table 2: Dot product: Execution latency (clock cycles) with varying buffers sizes and channel latency

Channel Latency	Buffer size (# data elements)								
(clock cycles)	0	1	2	4	8	16	32	64	128
1	2,109	2,525	1,925	1,625	1,489	1,433	1,381	1,357	1,345
2	2,512	2,930	2,130	1,730	1,546	1,466	1,402	1,370	1,354
5	3,721	4,145	2,745	2,045	1,717	1,565	1,465	1,409	1,381
10	5,736	6,170	3,770	2,570	2,002	1,730	1,570	1,474	1,426
20	9,766	10,220	5,820	3,620	2,572	2,060	1,780	1,604	1,516
50	21,856	22,370	11,970	6,770	4,282	3,050	2,410	1,994	1,786

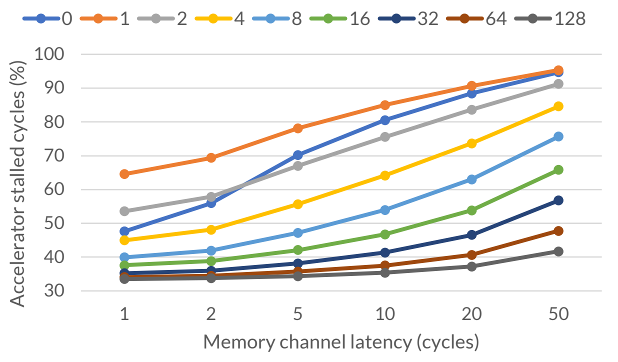

Figure 5: Dot product: Stalled cycles with varying buffer sizes and channel latency

4 EXPERIMENTAL EVALUATION

The proposed solution provides high flexibility for buffer configurations, depending on the design tradeoffs, to improve system performance. We present a design space exploration (DSE) analysis for the buffer configuration options for system characteristics (buffer sizes, memory channel latency) and the memory access patterns. We analyze the system performance on two features: performance speedup and memory channel occupancy required to achieve the speedup. In an ideal scenario, we want to achieve maximum performance speedup while keeping the memory channel occupancy at a minimum to mitigate any possible memory channel congestion.

We chose to synthesize a dot-product kernel for the DSE analysis concerning the buffer sizes. The dot-product kernel has a highly regular memory access pattern, so any variation in the buffer sizes directly affects the performance. Due to its importance, we also synthesized a matrix multiplication kernel to present the DSE analysis for the memory access patterns. The memory access patterns of the individual function arguments are different from each other. In the typical matrix multiplication, the elements of the first matrix are accessed row-wise, and the elements of the second matrix are accessed by column (i.e., a strided access pattern), while resulting elements need to be temporarily stored for subsequent computations. We synthesized both kernels using the open-source Bambu HLS tool (discussed in Section 2).

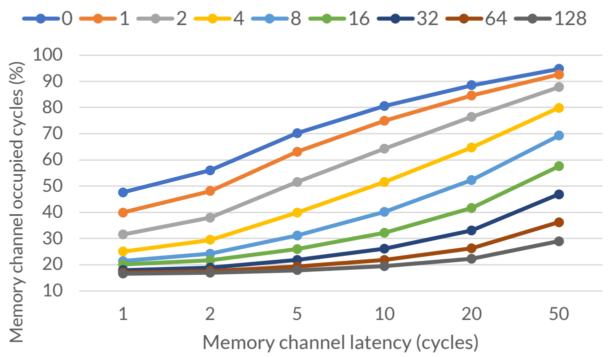

Figure 6: Dot product: Memory channel occupancy with varying buffer sizes and channel latency

4.1 DSE for buffer sizes

We first analyze how the buffer sizes and channel delay directly affect performance. For this study, we synthesized the dot-product kernel with an input size of 100 elements. We vary the size of the buffers from 0 (i.e., no buffer) to 128, which corresponds to the best-case scenario (i.e., full prefetching/localization). We also vary the memory channel delay from 1 to 50 clock cycles.

Table 2 summarizes the performance of the different design space configurations. The dot-product regularity provides us with information on both worst-case and best-case scenarios. The worst-case scenario corresponds to a buffer size of 1 element. This scenario is analogous to having a 100% buffer miss rate. In this case, as expected, the buffering system leads to performance degradation, since it causes the data to be read/written twice: once from/to external memory and once from/to the local buffer memory. The performance penalty becomes less significant as we increase the memory channel delay.

On the other side, we achieve the maximum performance with a buffer size of 128, which exceeds the kernel data size requirements. This best-case scenario corresponds to prefetching the entire input data to the local buffer memory, and finally writing it all at once to external memory when execution completes. We observe a performance speedup when increasing the buffer sizes, especially at higher channel delays. For example, using buffers with 128 elements, we observe a speedup of 4.02× for a channel latency of 10 cycles, while the speedup improves to 12.24× for a channel latency of 50 cycles.

The relative speedup diminishes as we approach the maximum buffer size. We observe this because increasing the buffer size reduces the miss rate, especially for regular computation. The lower miss rates signify that the memory channel is accessed fewer times, minimizing any potential impact of memory channel latency on the performance. However, it is important to note that even if bigger buffer sizes offer smaller relative gains, they have very low resource overheads, which are mostly limited to additional on-chip memory. From a system-level perspective, an automated design exploration engine can adjust the size of the buffers to achieve performance speedup closer to the best-case scenario while respecting the target device resource constraints (e.g., on-chip memory availability).

Figure 5 shows how the percentage of the accelerator idle time (i.e., time *stalled* waiting for memory requests to be serviced) varies with the buffer sizes and channel latency. The idle time follows the same trends of the execution latency, diminishing with increasing buffer sizes. Both latency and idle time are good metrics for evaluating resource utilization efficiency, since the original accelerator design remains unchanged. Figure 5 also presents an interesting observation: the accelerator is stalled for more time for a buffer size of 1 compared to no buffers. This observation confirms the reason for the performance degradation observed in Table 2. For a smaller memory channel latency, the buffer of size 2 results in more stalled cycles. This observation presents the design tradeoff scenario for a larger buffer size for a lower channel delay.

Finally, we evaluate the variations in channel occupancy time (percentage with respect to execution time), as reported in Figure 6. Memory channel occupancy, in addition to absolute performance, is an important characteristic from the system-level perspective. When multiple concurrent accelerators are attached to the same memory channel, keeping the channel occupancy low is critical to reducing congestion and the response time of memory operations, improving overall system performance. The relative time when the memory channel is occupied significantly decreases as the buffer sizes increase, since larger buffer sizes reduce memory channel accesses. It is important to note that, for the case of no buffer and buffer size of 1, the total number of cycles when the bus is occupied is identical, as the number of memory channel accesses is the same in both conditions. However, the occupancy percentage (calculated with respect to execution time) observed in Figure 6 is lower for size 1 because of the performance degradation seen in Table 2.

4.2 DSE for memory access patterns

We also analyze how the buffering system parameters depend on memory access patterns. We synthesized a matrix multiplication kernel with input matrices of size 6×6 (i.e., 36 elements). Matrix multiplication is an essential operation in almost all the scientific computing and machine learning applications. It also presents the use case where the memory access patterns for all the matrices are diverse. The first matrix (*matrix A*) requires row-wise element access, while the second matrix (*matrix B*) requires column-wise element access. In an external memory, the data elements are stored in a contiguous pattern. Thus, a strided memory access pattern emerges (with the stride being equal to the row size) when the consecutive elements of the same column are accessed. While the input matrices are read-only, the matrix multiplication results (and the intermediate products) must be stored for frequent read/write operations. We vary the size of buffers from 0 (i.e., no buffers) to 36 (i.e., entire data can be stored in the buffers). To limit the design space for brevity, we only vary the buffer sizes of a single function argument at a time while keeping the buffer size constant for other arguments.

Table 3 summarizes the performance of the matrix multiplication kernel for various buffer configurations and memory channel latency values. The second and last columns in Table 3 present the worst-case and best-case scenarios, respectively. Varying the buffer sizes for matrix A (i.e., matrix with row-wise element access) has a higher impact on performance improvement compared to

Table 3: Matrix multiplication: Execution latency (clock cycles) with varying buffers sizes and channel latency

Channel latency		Bufer sizes for matrices B and C = 36 Buffer size for matrix A						Bufer sizes for matrices A and C = 36 Buffer size for matrix B						Bufer sizes for matrices A and B = 36 Buffer size for matrix C						
(cycles)	0	1	2	4	8	16	32	1	2	4	8	16	32	1	2	4	8	16	32	36
1	5,678	4,818	4,170	3,976	3,460	3,412	3,374	4,818	5,034	5,466	4,818	4,890	3,912	3,775	3,560	3,454	3,418	3,390	3,368	3,348
2	6,578	5,256	4,392	4,116	3,488	3,428	3,386	5,256	5,472	5,904	5,040	5,040	3,966	3,925	3,638	3,496	3,444	3,408	3,380	3,356
5	8,630	6,570	5,058	4,536	3,572	3,476	3,422	6,570	6,786	7,218	5,706	5,490	4,128	4,375	3,872	3,622	3,522	3,465	3,420	3,380
10	11,470	8,760	6,168	5,236	3,712	3,556	3,482	8,760	8,976	9,408	6,816	6,240	4,398	5,125	4,262	3,832	3,652	3,560	3,486	3,420
20	17,205	13,140	8,388	6,636	3,992	3,716	3,602	13,140	13,356	13,788	9,036	7,740	4,938	6,625	5,042	4,252	3,912	3,750	3,616	3,500
50	34,410	26,280	15,048	10,836	4,832	4,196	3,962	26,280	26,496	26,928	15,696	12,240	6,558	11,125	7,382	5,512	4,692	4,312	4,006	3,740

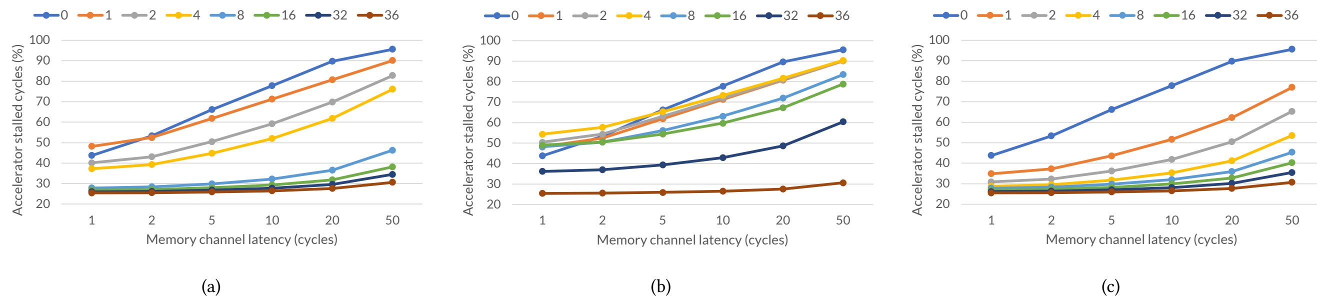

Figure 7: Matrix multiplication: Stalled cycles with varying channel latency and buffer sizes for matrices: (a) matrix A, (b) matrix B, and (c) matrix C, where [C] = [A]×[B]

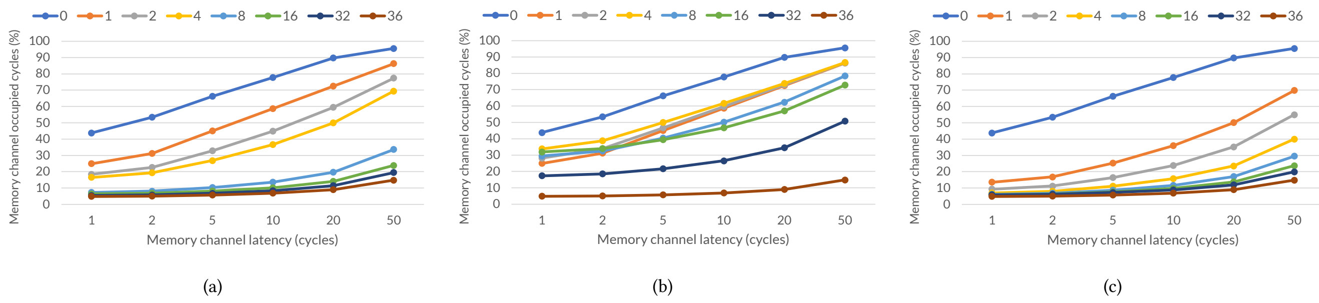

Figure 8: Matrix multiplication: Memory channel occupancy with varying channel latency and buffer sizes for matrices: (a) matrix A, (b) matrix B, and (c) matrix C, where [C] = [A]×[B]

changing the buffer sizes for matrix B (i.e., matrix with column-wise elements access). When varying the buffer sizes for matrix B, we observe a performance degradation with increasing buffer sizes up to size 4, and also degradation from 8 to 16 for smallest channel latency. There is marginal performance improvement for buffer of size 8. This observation implies that the current buffering system implementation does not optimize well for strided memory accesses. One possible optimization for strided access is to have multiple smaller buffers instead of one larger buffer. On the other hand, larger buffers are shown to improve the performance for larger channel delay values. Thus, for strided memory access patterns, it is essential to consider both the stride amount and channel delays to optimize the buffering system. When the buffer sizes for matrix C are varied, there is not much performance improvement observed. In this scenario, the input matrices A and B are stored in the local buffers, thus all the input load operations are serviced by the buffers.

Figure 7 shows the percentage of time the accelerator is stalled waiting for the requested data, for various buffer sizes and channel latency. In Figure 7(a), the accelerator is stalled significantly longer for the buffer sizes up to 4. These buffer sizes are smaller than the row size of the input matrix (in this case, 6). Hence, smaller buffer sizes cause additional memory load operations to fetch a part of the row, resulting in multiple buffer overwrites. Any buffer size larger than the row size shows similar stalled cycle trends. Figure 7(b) shows that the accelerator is stalled for a long time in almost all cases. The strided memory access pattern causes significant overwrites in the buffer. The accelerator is least stalled for data in Figure 7(c), since all the input data is present in the buffers. In this case, the accelerator is stalled only when it writes the results to the external memory. We report the variation in memory channel occupancy time in Figure 8. As expected, since all the input data is stored in buffers in the case of Figure 8(c), the memory channel is least occupied compared to Figures 8(a) and 8(b). The additional memory accesses because of inadequate buffer sizes (for buffer sizes less than 6, in the case of Figure 8(a)) and strided memory access pattern (for Figure 8(b)) increase the memory channel occupancy.

5 RELATED WORK

Burst transfers are a feature of the AMBA AXI protocol [3] that improves the throughput of the load-store functions by reading/writing chunks of data to or from external memory with a single transaction. Vitis HLS automatically generates a design that can perform burst transactions if the compiler can infer the burst lengths from the induction variable [2]. This optimization aggregates memory accesses inside loops/functions from user code in larger fixed-sized read/write global memory requests.

The AMBA AXI Data Prefetch Buffer IP block described in [24] provides a mechanism to prefetch contiguous data during read operations over the AMBA AXI bus. Unlike our work, this IP block performs asynchronous memory copies into a FIFO instead of a RAM, and is not designed for direct integration in HLS tools.

Shah *et al.* [22] propose Cache-accel, an FPGA accelerated cache simulator with a prefetcher positioned between the L1 and the L2 cache. The prefetcher module is parametric and can be partially reconfigured at runtime. The possible configurations are *next-line prefetching*, *stride prefetching*, and *best-offset prefetching* (BOP). Originally described in [16], BOP implements a learning system to determine the offset for the next data to be prefetched. The BOP hardware design inspired some of the ideas in our work.

The adaptive Memory Interface Controller (MIC) [5] is a multi-ported memory interface that manages concurrent memory access from arrays of parallel accelerators to multiple memory channels or banks. MIC is a templated module that HLS tools can specialize for methodologies that generate parallel accelerator designs [6]. The design proposed in our work could provide accelerator side buffers with prefetch and burst transfer functionalities to such an interface when the memory controller uses the AXI protocol, as in Xilinx hard and soft IP blocks.

6 CONCLUSION

Increasing the efficiency of memory accesses is critical in domain-specific systems composed of several custom accelerators when these designs need to access and share external memories directly. HLS tools that generate specialized accelerators from high-level languages, such as C or C++, typically require specific code annotations and restructuring to optimize their memory interfacing, often leading to additional significant development efforts. Moreover, these tools cannot efficiently deal with effects due to the presence of multiple accelerators. In this paper, we presented a buffering system for accelerators generated with HLS tools that access external memory. We discussed how our solution transparently improves the performance of accelerators generated through HLS tools through prefetching and allows assembling burst transactions (using, for example, the AXI4 protocol). We also discussed the design tradeoffs of the buffering system in single and multiple accelerator designs, and demonstrated how it improves accelerator performance while reducing the occupancy of the memory interfaces. We finally provided a path to integrating such a parametric buffering system in HLS tools.

REFERENCES

[1] AMD. 2020. Versal: The First Adaptive Compute Acceleration Platform (ACAP). Retrieved Dev 11, 2022 from https://docs.xilinx.com/v/u/en-US/wp505-versal-acap

[2] AMD. 2022. Vitis High-Level Synthesis User Guide (UG1399): Overview of Burst Transfers. Retrieved Nov 20, 2022 from https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/AXI-Burst-Transfers

[3] ARM. 2021. *AMBA AXI and ACE Protocol Specification.* Technical Report. ARM.

[4] Nicolas Bohm Agostini, Serena Curzel, Vinay Amatya, Cheng Tan, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Kaeli, and Antonino Tumeo. 2022. An MLIR-based Compiler Flow for System-Level Design and Hardware Acceleration. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD'22).* IEEE, San Diego, CA. https://doi.org/10.1145/3508352.3549424

[5] Vito Giovanni Castellana, Antonino Tumeo, and Fabrizio Ferrandi. 2014. An adaptive Memory Interface Controller for improving bandwidth utilization of hybrid and reconfigurable systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE'14).* 1–4. https://doi.org/10.7873/DATE.2014.192

[6] Vito Giovanni Castellana, Antonino Tumeo, and Fabrizio Ferrandi. 2021. High-Level Synthesis of Parallel Specifications Coupling Static and Dynamic Controllers. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'21).* IEEE, Portland, OR, USA, 192–202. https://doi.org/10.1109/IPDPS49936.2021.00028

[7] Serena Curzel, Nicolas Bohm Agostini, Vito Giovanni Castellana, Marco Minutoli, Ankur Limaye, Joseph Manzano, Jeff Zhang, David Brooks, Gu-Yeon Wei, Fabrizio Ferrandi, and Antonino Tumeo. 2022. End-to-End Synthesis of Dynamically Controlled Machine Learning Accelerators. *IEEE Trans. Comput.* 71, 12 (2022), 3074–3087. https://doi.org/10.1109/TC.2022.3211430

[8] Maico Cassel dos Santos, Tianyu Jia, Martin Cochet, Karthik Swaminathan, Joseph Zuckerman, Paolo Mantovani, Davide Giri, Jeff Jun Zhang, Erik Jens Loscalzo, Gabriele Tombesi, Kevin Tien, Nandhini Chandramoorthy, John-David Wellman, David Brooks, Gu-Yeon Wei, Kenneth Shepard, Luca Carloni, and Pradip Bose. 2022. A Scalable Methodology for Agile Chip Development with Open-Source Hardware Components. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD'22).* IEEE/ACM, San Diego, CA, USA.

[9] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. 2021. Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *ACM/IEEE Design Automation Conference (DAC'21).* IEEE, San Francisco, CA, USA, 1327–1330. https://doi.org/10.1109/DAC18074.2021.9586110

[10] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. 2020. RISQ-V: Tightly coupled RISC-V accelerators for post-quantum cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 20 (2020), 239–280. Issue 3. https://doi.org/10.13154/tches.v2020.i4.239-280

[11] Intel. 2020. Intel FPGA SDK for OpenCL. https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html Online accessed on 22-11-2022.

[12] Wooyoung Jang. 2019. Unaligned Burst-Aware Memory Subsystem. *IEEE Transactions on Very Large Scale Integration Systems* 27, 10 (2019), 2387–2400. https://doi.org/10.1109/TVLSI.2019.2922621

[13] Ozgur Kilic, Nathan Tallent, and Ryan Friese. 2020. Rapid Memory Footprint Access Diagnostics. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'20).* IEEE, Boston, MA, USA, 273–284. https://doi.org/10.1109/ISPASS48437.2020.00047

[14] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[15] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When Prefetching Works, When It Doesn't, and Why. *ACM Transactions on Architecture and Code Optimization* 9, 1, Article 2 (2012), 29 pages. https://doi.org/10.1145/2133382.2133384

[16] Pierre Michaud. 2016. Best-offset hardware prefetching. In *HPCA.* IEEE, 469–480.

[17] Matthew Naylor, Paul Fox, Theodore Markettos, and Simon Moore. 2013. Managing the FPGA memory wall: Custom computing or vector processing?. In *IEEE International Conference on Field programmable Logic and Applications (FPL'13).* IEEE, Porto, Portugal, 1–6. https://doi.org/10.1109/FPL.2013.6645538

[18] Jennifer Ngadiuba, Vladimir Loncar, Maurizio Pierini, Sioni Summers, Giuseppe Di Guglielmo, Javier Duarte, Philip Harris, Dylan Rankin, Sergo Jindariani, Mia Liu, et al. 2020. Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml. *ML: Science and Technology* 2, 1 (2020), 1–14. https://doi.org/10.1088/2632-2153/aba042

[19] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture, Unprecedented Acceleration at Every Scale. Retrieved Dev 11, 2022 from https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[20] Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, Tavio Guarino, Ajay Joshi, Mark Oskin, and Michael Bedford Taylor. 2020. BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs. *IEEE Micro* 40, 4 (2020), 93–102. https://doi.org/10.1109/MM.2020.2996145

[21] Louis-Noël Pouchet and Tomofumi Yuki. 2021. Polybench/C 4.2.1. Retrieved August 07, 2022 from https://web.cse.ohio-state.edu/~pouchet.2/software/polybench

[22] Shivani Shah, Vaibhavi Mathur, Sahithi Meenakshi Vutakuru, Kavya Borra, and Nanditha P. Rao. 2021. Cache-accel: FPGA Accelerated Cache Simulator with Partially Reconfigurable Prefetcher. In *2021 24th Euromicro Conference on Digital System Design (DSD)*. 97–100. https://doi.org/10.1109/DSD53832.2021.00024

[23] Axel Stjerngren, Perry Gibson, and José Cano. 2022. Bifrost: End-to-End Evaluation and optimization of Reconfigurable DNN Accelerators. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'22)*. IEEE, Singapore, 288–299. https://doi.org/10.1109/ISPASS55109.2022.00042

[24] Veriest. 2022. AMBA AXI Data Prefetch Buffer. Retrieved Nov 20, 2022 from https://www.design-reuse.com/sip/amba-axi-data-prefetch-buffer-ip-35567

[25] Xilinx. 2019. *Vivado Design Suite User Guide: High-Level Synthesis*. Technical Report UG902. Xilinx. 589 pages. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf

[26] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA'22)*. IEEE, Seoul, South Korea, 741–755. https://doi.org/10.1109/HPCA53966.2022.00060