Improving Memory Interfacing in HLS-Generated Accelerators with Custom Caches

Claudio Barone, Ankur Limaye,
Antonino Tumeo
{claudio.barone,ankur.limaye,antonino.tumeo}@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

Giovanni Gozzi, Michele Fiorito,
Fabrizio Ferrandi
{giovanni.gozzi,michele.fiorito,fabrizio.ferrandi}@polimi.it
Politecnico di Milano
Milano, Italy

1 INTRODUCTION

Accelerators based on reconfigurable devices (e.g., Field Programmable Gate Arrays, FPGAs) are becoming popular for data analytics in high performance computing or cloud computing systems. In fact, they allow instantiating custom accelerators tailored to specific, but ever evolving, computational patterns. However, designing efficient accelerators for these devices at the register transfer-level (RTL) is a difficult task. High Level Synthesis (HLS) tools can bridge this productivity gap, allowing to generate RTL designs starting from high-level languages (typically C/C++). The majority of HLS tools focus on optimizing the computational part of a kernel, often not considering data movement and memory access. For many applications, instead, memory operations take a significant part of the overall execution time and can be the actual bottleneck limiting accelerator performance. This is especially true for data intensive applications that require accessing large external, and often remote, memories. Commercial HLS tools only provide limited solutions to improve the access patterns to higher-latency memories external to the generated accelerators (e.g., DRAM). Xilinx, for instance, allows to reorganize data access operations in a kernel to enable AXI [1] bursts (i.e., load/store operations of multiple consecutive data following the AXI protocol, used to interface to memory controllers), but this approach requires the user to restructure the original code and adding pragma directives. We propose, instead, an approach based on the automatic generation and integration of accelerator caches. Similarly to how caches are used in general purpose processor and accelerators, our approach allows reducing the latency with which an HLS-generated accelerator accesses external memory through spatial and temporal locality. However, HLS-generated accelerators are highly specific to the original computational kernel, hence it is necessary to provide the flexibility to customize caches as each accelerator is designed. We show how we integrated our approach in Bambu [2], a state-of-the-art open-source HLS tool, and how it requires minimal user effort to be triggered. We show how our approach allows to easily explore tradeoffs between performance and resource utilization.

2 CACHES DESIGN

The architecture of our caches is inspired by the IObundle (IOb) caches [3]. However, we provide significant improvements and the integration within the HLS flow. Our caches can be customized in size, ways, and behaviors (e.g., write-through/write-back) as needed by the application through specific parameters. The cache module presents an AXI4 master [1] interface towards the external memory that can be used to read or write data in bursts. Burst transactions allow optimizing memory access by reducing traffic on

```
48  #pragma HLS_interface a m_axi direct bundle = gmem0
49  #pragma HLS_interface b m_axi direct bundle = gmem1
50  #pragma HLS_interface output m_axi direct bundle = gmem2
51
52  #pragma HLS_cache bundle = gmem0 way_size = 16 line_size = 16
53  #pragma HLS_cache bundle = gmem1 way_size = 16 line_size = 16
54  #pragma HLS_cache bundle = gmem2 way_size = 16 line_size = 16
55  void mmult(int* a, int* b, int* output)
```

Figure 1: Cache declarations

the memory interface and access latency, by transferring an entire cache line in a single transaction instead of an element at a time. Bursting also enables more efficient prefetching for applications that have dense data structures and high locality, such as dense linear algebra operations. In HLS-generated accelerators, a function transformed in accelerator can have a single channel to a memory or multiple separate channels to different memories (for example, one per argument of the original function, if they are pointers that do not alias and access separate data structures in memory). Our approach can generate a separate cache for each of these channels. This allows to customize each cache for the access patterns of each channel, and removes cache line contention for different memory areas accessed in parallel. Adding our caches to an accelerator only requires annotating the main kernel function with a few simple directives indicating their use and parameters for each of the desired AXI memory channel. We extended Bambu to parse the pragmas and configure the cache modules, before instantiating them as it synthesizes the accelerator. Figure 1 shows the pragmas that we pass to the extended version of Bambu. The code of the kernel does not need modifications. Our design provides several improvements with respect to the IOb caches. The most notable is a more efficient implementation of write transactions. Our solution can support outstanding write requests, i.e., it can initiate a new transaction before receiving the response to the previous one. This allows to further reduce channel latency, making our solution more efficient in case of frequent write transactions. Furthermore, our caches include a flushing mechanism, so that they can write back dirty cache lines to the external memory before the accelerator signals completion of the execution, assuring that they moved all data to the external memory. Our caches do not provide a coherency mechanism. As highlighted, the HLS tool user assures, when writing the HLS annotated code, that separate memory ports operate on data in different memory regions, hence there is no data sharing.

3 EXPERIMENTAL RESULTS

To evaluate the impact of our caches for HLS-generated accelerators, we synthesize with our modified version of Bambu 5 kernels from the PolyBench [4] suite: *2mm*, *atax*, *bicg*, *doitgen*, *mvt*. We simulate the generated accelerators and compare their execution delay with

Table 1: Resource utilization overhead (for registers, LUTs - R,L) and speed up (as execution delay in clock cycles - C) with 50 clock cycles of memory latency - 2mm and doitgen

	No cache			16, 256			32, 256			64, 256			128, 256			256, 256			256, 16			256, 32			256, 64			256, 128		
	L	R	C	L	R	C	L	R	C	L	R	C	L	R	C	L	R	C	L	R	C	L	R	C	L	R	C	L	R	C
2mm	6671	6629	136161	1.36	1.19	9.1	1.44	1.27	9.75	1.45	1.27	9.75	1.44	1.28	9.75	1.42	1.27	9.75	1.36	1.24	1.01	1.42	1.28	1.31	1.42	1.27	1.61	1.43	1.27	9.75
doitgen	4520	4901	739564	1.36	1.18	9.62	1.49	1.23	10.05	1.45	1.23	10.05	1.46	1.23	10.05	1.44	1.23	10.05	1.32	1.17	1.07	1.46	1.23	1.36	1.45	1.23	1.56	1.45	1.23	10.05

Table 2: Resource utilization overhead (for registers, LUTs - R,L) and speed up (as execution delay in clock cycles - C) with 50 clock cycles of memory latency - atax, bicg, mvt

	No cache			16			32			64			128			256		
	L	R	C	L	R	C	L	R	C	L	R	C	L	R	C	L	R	C
atax	7171	6393	7606	1.05	1.03	4.27	1.11	1.07	4.79	1.11	1.07	5.08	1.11	1.07	5.33	1.11	1.07	5.33
bicg	8094	6855	7332	1.06	1.04	2.97	1.11	1.07	3.32	1.11	1.07	3.94	1.24	1.07	4.09	1.11	1.07	6.37
mvt	4787	5513	14680	1.07	1.06	1.48	1.16	1.10	1.90	1.17	1.10	2.27	1.16	1.10	4.10	1.16	1.10	7.36

and without caches, swiping cache sizes from 16 to 256 words of 4 bytes each, and varying external memory latency from 5 to 50 clock cycles. We also compare the resource utilization by synthesizing the kernels for a Virtex7 FPGA device using Vivado 2020.2. We use an input of 10 and of 10 by 10 elements for every vector and matrix, respectively. We write the HLS code to instantiate a different AXI memory port and cache for each input matrix, unless there is no cache contention due to how a kernel operates. For instance, in the 2mm kernel, matrix C is only accessed after the computation on matrix B is over, and the two also have the same access pattern. Thus, the two inputs can share the same memory port and cache with no impact on performance, reducing resource utilization. We do not generate separate channels and caches for inputs/outputs corresponding to vectors, reusing the ones for accessing data of matrices. Since we use vectors much smaller than matrices, we expect only limited cache interference. In Tables 1 and 2 we show the area overhead (for look-up tables, L, and registers, R) and the execution delay in clock cycles (C) for each accelerator when accessing an external memory with a latency of 50 clock cycles. We express the cache sizes in number of memory words. We only show absolute values for the baseline (no caches), showing instead overhead factors (L and R columns) and speedup (C column) for cache configurations. *2mm* and *doitgen* perform matrix multiplications, thus access a first input matrix by row and a second by column. For these kernels, we explore size of the caches for the input matrices separately, reporting first the size of the cache pertaining to data accessed by row and then the size of the cache for data accessed by column. We show the analysis with the entire latency swipe only for *2mm* (Figure 2) and *atax* (Figure 3). The trends of the other benchmarks are similar. As expected, using caches provides higher speedup as the external memory latency increases. The best case configuration for *doitgen* reaches a speedup over 10× the baseline solution with no caches. With low external memory latency, caches become ineffective, to the point that handling misses and line replacements in some cases (e.g., small caches for matrices accessed by column in Figure 2) leads to a slowdown.

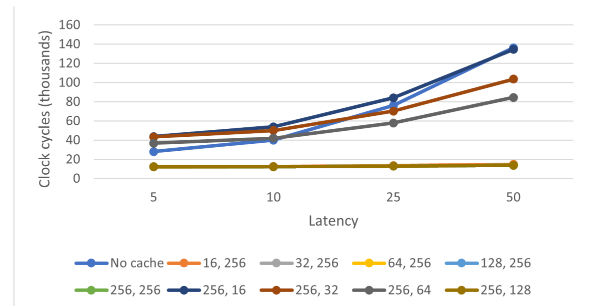

Figure 2: Execution delay in clock cycles - 2mm

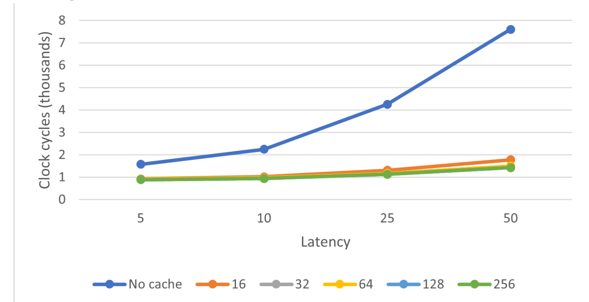

Figure 3: Execution delay in clock cycles - Atax

4 CONCLUSION

We presented an approach to integrate custom caches in HLS-generated accelerators that access external memory. Our approach allows to easily explore the cache trade-offs as users design accelerators within the HLS tool. We performed a trade-off analysis on the PolyBench benchmark suite, showing performance improvements over 10× a baseline solution with high external memory latency.

REFERENCES

[1] AMBA AXI and ACE Protocol Specification. Technical report, ARM, 2021.

[2] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *Proceedings of the ACM/IEEE Design Automation Conference*, DAC'21, pages 1327–1330, December 2021.

[3] Mário P. Véstias João V. Roque, João D. Lopes and José T. de Sousa. Iob-cache: A high-performance configurable open-source cache. *Algorithms*, July 2021.

[4] Louis-Noël Pouchet and Tomofumi Yuki. Polybench/c 4.2.1, 2021.