POLITECNICO DI TORINO
Repository ISTITUZIONALE

Array-specific dataflow caches for high-level synthesis of memory-intensive algorithms on FPGAs

Original
Array-specific dataflow caches for high-level synthesis of memory-intensive algorithms on FPGAs / Brignone, G., Jamal, M.U., Lazarescu, M.T., Lavagno, L.. - In: IEEE ACCESS. - ISSN 2169-3536. - 10:(2022), pp. 118858-118877. [10.1109/ACCESS.2022.3219868]

Availability:
This version is available at: 11583/3011731 since: 2026-06-05T12:53:21Z

Publisher:
IEEE

Published
DOI:10.1109/ACCESS.2022.3219868

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

15 October 2026

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 11 October 2022, accepted 26 October 2022, date of publication 4 November 2022, date of current version 16 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3219868

RESEARCH ARTICLE

Array-Specific Dataflow Caches for High-Level Synthesis of Memory-Intensive Algorithms on FPGAs

GIOVANNI BRIGNONE, (Graduate Student Member, IEEE), M. USMAN JAMAL, (Graduate Student Member, IEEE), MIHAI T. LAZARESCU, (Senior Member, IEEE), AND LUCIANO LAVAGNO, (Senior Member, IEEE)

Department of Electronics and Telecommunications, Politecnico di Torino, 10129 Turin, Italy

Corresponding author: Giovanni Brignone (giovanni.brignone@polito.it)

ABSTRACT Designs implemented on field-programmable gate arrays (FPGAs) via high-level synthesis (HLS) suffer from off-chip memory latency and bandwidth bottlenecks. FPGAs can access both large but slow off-chip memories (DRAM), and fast but small on-chip memories (block RAMs and registers). HLS tools allow exploiting the memory hierarchy in a scratchpad-like fashion, requring a significant manual effort. We propose an automation of the FPGA memory management in Xilinx *Vitis HLS* through a fully-configurable C++ source-level cache. Each DRAM-mapped array can be associated with a private level 2 (L2) cache with one or more ports, and each port can optionally provide level 1 cache. The L2 cache runs in a separate dataflow task with respect to the application accessing it. This solution isolates off-chip memory accesses and data buffering into dedicated dataflow tasks, resembling the load, compute, store design paradigm, but without the drawback of manual algorithm refactoring. Experimental results collected from FPGA board show that our cache speeds up the execution of a variety of benchmarks by up to 60 times compared to the out-of-the-box solution provided by HLS, requiring very limited optimization effort. Our caches are not meant to compete with manually optimized implementations quality of results (QoR), but rather to significantly save design effort, in exchange for some QoR, to make the HLS flow a bit more software-like, allowing the designer to focus on algorithmic optimizations, rather than on explicit memory management. Moreover, caching could be the only feasible memory optimization for algorithms with data-dependent or irregular memory access patterns, but with good data locality.

INDEX TERMS Cache, FPGA, high-level synthesis, memory management.

I. INTRODUCTION

In the Post-Moore Era simultaneous performance and energy improvements can be obtained only from specialized hardware (HW) architectures [1], [2], [3]. While specialized HW is efficient, it also increases the design effort and the deployment cost. However, high-level synthesis (HLS) can significantly reduce the design effort, enabling convenient software (SW)-like tools and development flows. At the same time, field-programmable gate arrays (FPGAs) can reduce deployment cost allowing the designer to implement special-purpose HW modules on general-purpose reconfigurable architectures. Our work focuses on applications where the time to market, application lifetime, requirements to frequently update the implementation and so on make FPGAs the best solution at hand, and we strive to bring the development of FPGA-accelerated applications a bit closer to SW development experience.

A HLS open issue is the off-chip memory latency and bandwidth bottleneck, which limits performance and is especially critical for memory-bound algorithms. The FPGA memory system is composed of two main kinds of resources:

The associate editor coordinating the review of this manuscript and approving it for publication was Ilaria De Munari.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

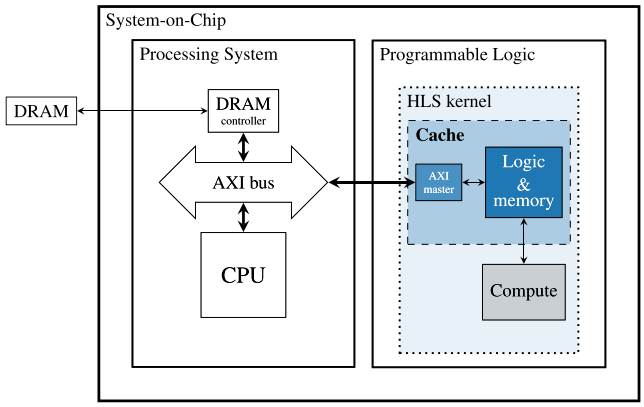


FIGURE 1. Our cache embedded in a HW setup.

fast small on-chip memories (registers and block RAMs (BRAMs)), and slow large off-chip memories (dynamic RAMs (DRAMs)) interfaced through DDR4 or HBM protocols (the latter characterized by even larger latency [4]). Current HLS tools, in particular those from the leading producer Xilinx, allow the designer to exploit this memory hierarchy only manually, in a scratchpad-like way, which often requires significant design and verification effort. *This makes harder to achieve the deployment of accelerated applications using FPGAs for a large number of applications*. Our work aims directly at filling this gap, thus *making HLS design more software-like* for use cases in which the ultimate performance need not be achieved, but *design time and effort are paramount*.

According to the best design practice from Xilinx [5], efficient HLS kernels should comply with the load, compute, store (LCS) paradigm to mitigate the off-chip memory bottleneck, i.e., access external DRAM only by load and store dataflow tasks, which are then responsible for buffering on the on-chip memory the data consumed and produced by the compute task(s). The main drawback of the LCS approach is the significant design effort needed for converting a generic algorithm into LCS form, which often requires full rewriting and redesigning of the source code.

The aim of our work is to automate efficient off-chip memory accesses through an easy to use and fully customizable cache system[1] *for HLS, which works as an interface with the off-chip DRAM, accessible through a advanced extensible interface (AXI) bus, and stores its data to on-chip BRAMs and registers.* Figure 1 shows the resulting system when our cache is used to accelerate a HLS kernel. Our cache is placed within the HLS kernel. The computation logic of the kernel accesses the cache, rather than the AXI bus directly.

A cache is in general helpful to implement well-performing designs in a short time. Moreover, techniques such as manual buffering or polyhedral transformations [6], [7] cannot be applied to programs with irregular or input-dependent memory access patterns, and are only partially implemented in commercial design tools such as *Vitis HLS*. Therefore, a cache could be the only solution for quickly optimizing the performance of such designs using commercial tool flows.

From a high-level point of view, the cache has the objective of isolating the off-chip memory accesses into a dataflow task, in accordance with the LCS pattern.

From a low-level point of view, the cache has the dual purpose of (a) reducing the number of DRAM accesses, i.e., the data stored in the cache is reused as long as it hits, and only the misses need to access the DRAM, and (b) optimizing DRAM accesses, i.e., the DRAM is accessed in lines (aligned groups of consecutive words), which allows taking advantage of AXI bursts and interface widening, even with hard to analyze or totally irregular access patterns.

HLS allows assigning each array to a different AXI master adapter. This enables implementing array-specific caches, each using its dedicated AXI adapter. Array-specific caches can be easily tuned to achieve high hit ratios, since the access patterns of a single array are typically characterized by good locality, and there is no interference with the accesses to other arrays, unlike when all arrays share a single cache.

To adhere to the HLS high-productivity philosophy, we paid a special attention to the HLS user-friendliness in terms of (a) *configurability* (the cache characteristics can be set through parameters), (b) *ease of use* (the cache can be inserted into existing designs with just a few lines of boilerplate code), (c) *observability* (cache information critical for parameter tuning, e.g., hit ratio, can be profiled during SW simulation).

This cache architecture can be implemented as a standard dataflow design in case of write-only (WO) accesses because the request (the address and the data to be written) flows from the application accessing the memory and the cache module. Read accesses include instead both the address request from the application to the cache and the data response moving in the opposite direction. This requires a feedback channel between dataflow tasks, making the dataflow graph cyclic. Cyclic dataflow is not natively supported by current HLS tools, and might impose severe functional and performance limitations. Therefore, we designed a throughput-oriented *Cyclic dataflow protocol* for *Vitis HLS*.

Since we do not have access to the *Vitis HLS* back-end, we implemented the cache module at the C++ source level to make the HLS tool aware of the cache instead of, e.g., a register-transfer level (RTL) module to be inserted during system integration, so that the HLS tool can apply its optimizations accordingly. This required us to almost mimic a low-level RTL design style in the C++ code to maximize the throughput (e.g., set explicitly the delay between the inter-tasks communication operations), and to minimize the resources (e.g., set explicitly the bitwidth of the cache data structures). The module is designed to be used with Xilinx *Vitis HLS* and was tested with the version *2021.2*.

The cache provides the best effectiveness either (1) when applied to algorithms with unpredictable access patterns, but good temporal and spatial locality properties, or (2) when used with a tool, like *Vitis HLS*, which only partially

[1] Available as open source at https://github.com/brigio345/DaCH

exploits powerful memory access optimization capabilities (e.g., state-of-the-art polyhedral analysis), and hence the tool is not able to understand the access pattern and infer an efficient memory burst.

Our main contributions are:

- Design of a high-throughput *Cyclic dataflow protocol* that adds support in HLS for dataflow networks with feedback between tasks, taking care of correct functionality and performance of the generated HW (unlike previous works [8], [9], which addressed only the functionality of SW simulation) (Section III-A).
- Design of a fully configurable two-level multi-port *Cache module* for HLS, which exploits the *Cyclic dataflow protocol* to automatically generate an LCS-like architecture, that is more scalable and can achieve higher performance than the cache by Ma et al. [10] (Sections III to V).
- Evaluation of the *Cache module* power, performance, and area (PPA) effects on different algorithms against (1) the HLS tool default implementations, (2) optimizations using a RTL cache module, integrated after the HLS and (3) manual optimizations compliant with Xilinx LCS best practice recommendations (Section VI).

II. PREVIOUS WORK

The need for automated memory management for FPGAs is attested by the multiple works on this topic.

Matthews et al. [11] and Choi et al. [12] designed FPGA-based caches. These works differ from ours as they are aimed to accelerate specific soft-processors implementations instead of generic HLS designs.

Jo et al. [13] developed an OpenCL framework whose memory subsystem inserts direct-mapped, single-level, and single-port caches in between the kernel accessing the memory, and the external memory. They implemented at RTL both the kernels (which consist of a predefined set of intellectual property (IP) blocks) and the cache. Our work therefore differs both in terms of cache architecture complexity (our caches provide set associativity, two levels, and multiple ports), and in terms of technology (our cache is compatible with any HLS design).

Several works focused on optimizing the memory accesses through RTL cache modules inserted between the kernel accessing the off-chip memory and the off-chip memory interface. These modules can be either inserted manually or through a dedicated framework, such as the one proposed by Adler et al. [14] which virtualizes the FPGA memory hierarchy and includes some caching capabilities. Winterstein et al. [15] improved this framework specifically for HLS by allocating the unused BRAMs to maximize the cache sizes. However, a RTL cache module fails to provide significant speedup when coupled with a HLS kernel. For example, the *Vitis HLS* scheduler, unaware of the external cache module, inserts a minimum latency between a memory request operation and its corresponding response based on the architecture of the memory adapter, thus preventing the exploitation of the cache acceleration. Our cache is instead implemented at the source level, and it is specifically designed to avoid scheduling based on the worst case (cache miss). This allows the HLS tool to optimize the circuit accordingly.

We ran some experiments adding a RTL cache module (specifically the Xilinx System Cache [16]) to the interface of a HLS kernel. The results show that the RTL cache did not provide any advantage. It simply introduced an overhead, as discussed in Section VI.

Cong et al. [6] and Pouchet et al. [7] designed a workflow for improving data locality of HLS programs through compiler-level loop transformations, taking advantage of the polyhedral representation. Moreover, they exploited this locality by automatically inserting on-chip buffers. These techniques are limited to programs with affine loop bounds and memory accesses, while a cache can be used with any program, including those with irregular or data-dependent memory accesses. A cache could benefit from their improved locality by achieving higher hit ratios with simpler cache configurations.

The Intel HLS [17] tool provides load-store units (LSUs) that can cache DRAM data in BRAM in case of read-only (RO) memories. Our experiments described in Section VI suggest that the tool fails to determine the optimal cache configuration and the user has limited control to improve it.

The work by Ma et al. [10] is closest to ours. They proposed an open-source array-specific HLS cache module as a set of C++ classes, compatible with *Vivado HLS 2016.2*. Different from our work, the cache logic is inlined in the application. While this helps keeping the hit latency low in simple cases, it violates the LCS pattern. Moreover, their architecture increases the pipelining complexity. To mitigate this problem, they mapped the whole cache data to registers. However, in the experiments discussed in Section VI, we verified that the pipelines embedding their cache require higher initiation intervals (IIs), or are not pipelineable at all. Moreover, mapping all the data to registers limits strongly the cache size scalability due to HW resources constraints. Instead, our architecture completely hides the cache logic and the memory interface from the main computations performed by the kernel. This allows the HLS to synthesize pipelines with low II while mapping cache data to cheaper BRAMs. Finally, their cache automatically handles only one access port thus providing only one read or write per clock cycle (CC). The only way to perform multiple accesses per CC is to guarantee that other accesses, beyond the first one in a given CC, will always be hits, and make it explicit through the `retrieve` and `modify` functions. This is both difficult and error-prone to analyze manually in complex cases.

III. DATAFLOW CACHE

The *Dataflow cache* architecture (Fig. 7a) is isolated into a dedicated dataflow process. A HLS kernel that is configured to use the cache for one of its top-level DRAM-mapped

(a) Standard mapping. (b) Swapped mapping.

FIGURE 2. Configurable address bit mapping.

```
+#include "cache.h"
+
+typedef cache<DATA_TYPE, RD_ENABLED, WR_ENABLED,
+   MAIN_SIZE, N_SETS, N_WAYS, N_WORDS_PER_LINE, LRU,
+   SWAP_TAG_SET, LATENCY> cache_type;

 template <typename T>
 void compute(T &a) {
    for (auto i = 0; i < (N - 1); i++) {
 #pragma HLS pipeline
        a[i] = a[i + 1];
    }
 }

 extern "C" void top(DATA_TYPE *a) {
 #pragma HLS interface m_axi port=a bundle=gmem0
+#pragma HLS dataflow
+   cache_type a_cache(a);
-   compute(a);
+   cache_wrapper(compute<cache_type>, a_cache);
 }
```

Listing 1. Source code modifications for accelerating the `compute` function with our cache.

arrays is split into two dataflow tasks: (i) the compute task, which includes all the application logic except for the external memory interface, which is replaced with the simpler cache interface, and (ii) the cache task (or, in general, one cache task per array that uses the cache), which buffers data and interfaces with the external DRAM. Thus, the kernel automatically complies with the LCS architecture without any manual code change.

This architecture is characterized by information flow from the compute task to the cache (the address to be accessed and the data to be written), and from the cache to the compute task (the read data). Therefore, the resulting dataflow graph is cyclic, which is not officially supported by *Vitis HLS*. For this reason, the *Dataflow cache* is implemented according to our *Cyclic dataflow protocol*.

Algorithm 1 describes the *Dataflow cache* functionality. The cache task waits for a request and executes the standard cache operations: it checks if the request is a hit or a miss, it updates the cache data structures (valid bits, dirty bits, tag bits, ...), and it performs the DRAM read or write operation. For reads, it also sends back the data. The compute task sends the read or write request to the cache. For reads, it waits for the response containing the read data.

The *Dataflow cache* uses the set associative mapping and the write-back consistency policy. It is configurable in terms of (a) word size, (b) number of words per line, sets, and ways, (c) replacement policy, least recently used (LRU) or first-in first-out (FIFO), (d) address bit mapping, standard (Fig. 2a) or swapped (Fig. 2b, convenient in use cases like the one discussed in Section VI-B, i.e., a matrix accessed by columns). It can implement a fully associative policy if the number of sets is one, or a direct mapped policy if the number of ways is one.

Algorithm 1 *Dataflow Cache* Functionality

Require: *Compute* needs to access an array associated with *Cache* at address *addr* in read mode ($op = R$) or write mode ($op = W$, $data$ = element to be written).

Ensure: The operation requested by *Compute* is fulfilled by *Cache*.

procedure Compute
  ...
  Send *op* to *Cache*
  Send *addr* to *Cache*
  if $op = W$ **then**
    Send *data* to *Cache*
  else
    Wait for *Cache* response
    Receive *data* from *Cache*
  end if
  ...
end procedure

procedure Cache
  Wait for *Compute* request
  Receive *op* from *Compute*
  Receive *addr* from *Compute*
  if $op = W$ **then**
    Receive *data* from *Compute*
  end if
  $line : addr \in line$
  if $line \Rightarrow MISS$ **then**
    if $line_{\text{old}} \Rightarrow DIRTY$ **then**
      $DRAM(line_{\text{old}}) \leftarrow BRAM(line_{\text{old}})$
    end if
    $BRAM(line) \leftarrow DRAM(line)$
  end if
  if $op = W$ **then**
    $BRAM(addr) \leftarrow data$
  else
    $data \leftarrow BRAM(addr)$
    Send *data* to *Compute*
  end if
end procedure

Listing 1 highlights the modifications needed for accelerating the `compute` function with our cache. Users simply need to (1) set the cache parameters through the `cache` class template arguments, and (2) instantiate the cache and call the `compute` function through the `cache_wrapper` function in a dataflow region. Complete examples can be found in our open source git repository.

It is worth noting that the `compute` function is unchanged, since we overloaded the `operator[]`, like Ma et al. [10], to allow using a cache object as if it were a traditional array.

A. CYCLIC DATAFLOW PROTOCOL

The dataflow optimization is crucial in HLS, since it enables (a) *parallelism*, i.e., multiple tasks are executed in parallel,

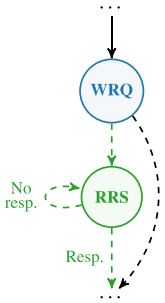
(b) *isolation*, i.e., the different tasks are organized in separate modules and share only the necessary data and synchronization, and (c) *dynamic behavior*, i.e., the tasks execute as soon as they are ready, and their input data is available, thus they are not statically bound to the worst case.

The current *Vitis HLS* tool directly and easily supports acyclic dataflow graphs only, while a kernel may contain feedback between tasks, making the dataflow graph cyclic. Our *Dataflow cache* is actually an example of cyclic dataflow graph. Fine Licht et al. [9] and Chi et al. [18] added support for SW simulation of cyclic dataflow designs by mapping each dataflow task to a separate thread or coroutine during simulation. However, they neglected the functional (deadlocks) and performance (stalls) penalties, due to the dataflow feedback, in the generated HW. We instead defined a *Master/Slave* communication protocol, compatible with both cyclic and acyclic dataflow graphs, which (i) adds support to *Vitis HLS* for SW simulation of cyclic dataflow designs *without the need for multi-threading or coroutines*, which complicate inter-process synchronization due to the need for mutexes or other inter-thread synchronization mechanism, and may increase the execution time of the simulation, due to the inter-process communication and context switch overheads. (ii) Moreover, it allows generating HW circuits that are deadlock-free and provides a high throughput. I.e., if the tasks are pipelined, the *Master* can send one request and receive one response from the *Slave* at each CC, with a II of 1 CC.

In our protocol, each dataflow task is either a *Master* or a *Slave*. A *Slave* executes the operations requested by its *Master*.

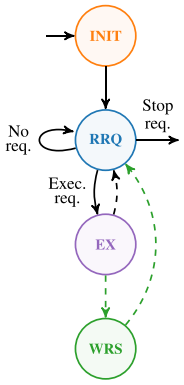
The tasks communicate and synchronize through FIFO queues. The request FIFO, which flows from *Master* to *Slave*, contains the inputs to the *Slave* operation (e.g., if the operation is a read access from an off-chip memory, it contains the address to be read). The response (feedback) FIFO, which flows from *Slave* to *Master*, contains the outputs from the *Slave* operation (e.g., if the operation is a read access from an off-chip memory, it contains the read data). This FIFO introduces a cycle in the dataflow graph. If the *Slave* does not send any response to the *Master* (e.g., the operation is a write access to an off-chip memory), the response FIFO is not allocated and the resulting dataflow graph is acyclic. Thus, our protocol supports both cyclic and acyclic dataflow graphs.

The *Master* structure, shown in Fig. 3, is not tightly constrained. It can start executing the sub-finite-state machine controlling its *Slave* at any point. Conversely, the *Slave* structure, shown in Fig. 4, is well-defined. It is implemented as an infinite loop that performs one iteration upon receiving an execution request from its *Master* and stops upon a stop request. If the *Slave* is pipelined, it must be flushable to avoid deadlocks. Once a request enters the pipeline, it must pass through all the pipeline stages till the completion even if no new request feeds the previous stages. For this, the *Slave* must read the requests using non-blocking stream reads.



State	Action	Next state
Write request (WRQ)	Write a request to the *Slave*	*RRS* if response exists (green arrow), exit otherwise (black arrow)
Read response (RRS)	Read pending response from the *Slave*, if any	Exit if response was read, *RRS* otherwise

FIGURE 3. Finite-state machine summarizing the behavior of a ***Master*** in our ***Cyclic dataflow protocol***.



State	Action	Next state
Initialize (INIT)	Initialization	*RRQ*
Read request (RRQ)	Read pending request from the *Master*, if any	*EX* if an execution request was read, exit if a stop request was read, *RRQ* otherwise
Execute (EX)	Execute the requested operation	*WRS* if response exists (green arrow), *RRQ* otherwise (black arrow)
Write response (WRS)	Write response to the *Master*	*RRQ*

FIGURE 4. Finite-state machine summarizing the behavior of a ***Slave*** in our ***Cyclic dataflow protocol***.

The protocol can be generalized in terms of both width (a single *Master* can have multiple *Slave*s or a single *Slave* can have multiple *Master*s), and depth (a *Slave* can be in turn a *Master* of another *Slave*).

1) HARDWARE FUNCTIONALITY AND PERFORMANCE

In the presence of feedback between the *Slave* and the *Master*, the HLS-generated HW circuit would deadlock. Moreover, we have to carefully specify cycle by cycle the scheduling of the *Slave* operations to avoid losing performance or causing unexpected deadlocks, as discussed next.

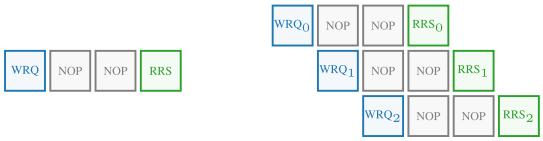
Whenever the *Master* writes a request, *WRQ*, it must wait for the *Slave* latency before being able to read the response, *RRS*. However, the HLS scheduler is not aware of that dependency and schedules both the *WRQ* and *RRS* into the same pipeline stage. This leads to a deadlock, because the *RRS* is blocked while reading from the empty response FIFO (the latency of the *Slave* has not elapsed, thus it cannot contain the response yet). This blocks the whole stage, including the *WRQ*: *RRS* is therefore waiting for the response to a request which has never been written.

To avoid the deadlock, the *WRQ* and *RRS* must be scheduled into separate pipeline stages by:

1) Explicitly declaring a dependency between *WRQ* and *RRS* using the `write_dep` and `read_dep` FIFO

(a) Static schedule. (b) Runtime behavior.

FIGURE 5. Stalling cyclic dataflow schedule of *Master*.



(a) Static schedule. (b) Runtime behavior.

FIGURE 6. Non-stalling cyclic dataflow schedule of *Master*.

access functions provided by *Vitis HLS* to define a partial ordering between accesses to different streams.

2) Setting the dependency distance to 1 CC by delaying it with the `reg` function, also provided by *Vitis HLS*.

While this solution guarantees the functionality of the generated HW, it fails to achieve high throughput. In fact, assuming that both the *Master* and the *Slave* are pipelined with a II of 1 CC (i.e., the most performance-critical case) and the *Slave* pipeline depth is $D > 1$, the HLS scheduler schedules the *RRS* in the CC following the *WRQ* because it is unaware of the latency between *WRQ* and *RRS*, as shown in Fig. 5a. At runtime, the RRS_0 scheduled in the cycle following the WRQ_0 stalls because the *Slave* takes D CCs before writing its response. Consequently, the writing of all the following requests stalls, i.e., WRQ_1 can be executed only when RRS_0 completes, after receiving the response from the *Slave* (Fig. 5b). Thus, the *Slave* never receives requests in consecutive cycles and its throughput is $1/D$, as if it were not pipelined.

However, if we set the dependency distance between *WRQ* and *RRS* to D CCs, the scheduler inserts $D-1$ pipeline stages between them, as shown in Fig. 6a. In each CC, the *Master* writes one request and receives one response, as shown in Fig. 6b. Therefore, our solution allows optimally exploiting the pipeline with a II of 1 CC, without incurring stalls.

2) SOFTWARE SIMULATION

The SW simulation natively provided by *Vitis HLS* consists of compiling the top function of the kernel with a standard C++ compiler, which ignores the HLS pragmas, and executing it as SW. Thus, dataflow functions are executed sequentially and introduce a deadlock if there is feedback between the *Slave* and the *Master*. As discussed above, the *Master* blocks waiting for the *Slave* response, but the *Slave* cannot start until the *Master* has completed, resulting in a deadlock.

Even the SW simulation of an acyclic dataflow graph, which is officially supported by *Vitis HLS*, is affected by a severe limitation. All the requests are pending in the FIFO until the *Master* returns and its *Slave* can finally consume them. The functionality is preserved, but the memory usage for storing the pending requests may explode.

```
response_type slave_ex(request_type rq) {
    ...
}

void slave(hls::stream<request_type> &rq_stream,
        hls::stream<response_type> &rs_stream) {
    while (1) {
#pragma HLS pipeline II=1
        request_type rq;
RRQ:    if (rq_stream.read_nb(rq)) {
            if (rq.type == STOP_RQ)
                break;
EX:         response_type rs = slave_ex(rq);
WRS:        rs_stream.write(rs);
        }
    }
}

void master(hls::stream<request_type> &rq_stream,
        hls::stream<response_type> &rs_stream) {
#pragma HLS pipeline II=1
    request_type rq;
    response_type rs;
    ...
#ifdef __SYNTHESIS__
WRQ:bool dep = rq_stream.write_dep(rq, false);
    dep = delay<D>(dep);
RRS:rs_stream.read_dep(rs, dep);
#else
    rs = slave_ex(rq);
#endif /* __SYNTHESIS__ */
    ...
}

void top(...) {
#pragma HLS dataflow
    hls::stream<request_type> rq_stream;
    hls::stream<response_type> rs_stream;
    master(rq_stream, rs_stream);
#ifdef __SYNTHESIS__
    slave(rq_stream, rs_stream);
#endif /* __SYNTHESIS__ */
}
```

Listing 2. *Vitis HLS* source code implementation of the *Cyclic dataflow protocol*.

We solved the issues by automatically changing the SW simulation code with respect to what is used by HW synthesis. Each *Slave* is mapped to a function whose argument list is the *Master* request and whose return value is the response value. The top function only calls the *Master* function (which in turn calls the function of its *Slave*) whenever it would issue a request FIFO write in the HW model. This solution both ensures the absence of deadlocks in case of cyclic dataflow graphs and avoids the accumulation of pending requests in case of acyclic dataflow graphs.

3) PROTOCOL IMPLEMENTATION

Listing 2 contains our implementation of the protocol, compatible with *Vitis HLS*. The discrimination of the HLS synthesis code from the SW simulation code is automatically done by checking the definition of the `__SYNTHESIS__` preprocessor identifier, which is defined by *Vitis HLS* during synthesis.

The request and response FIFOs are implemented as `hls::streams`. The *Master* finite-state machine is contained in the `master` function. The *Slave* finite-state machine is implemented by the `slave` function, and its

EX state is isolated in the `slave_ex` function. The whole system is integrated within the `top` function. During the HLS synthesis, it instantiates the *Master* and *Slave* dataflow tasks, while for SW simulation it calls the `master` function, which calls the `slave` function.

Note that with our caches all this code is hidden from the designer, who only needs to instantiate the cache class as shown in Listing 1.

B. DATAFLOW CACHE IMPLEMENTATION

The *Dataflow cache* is implemented as a C++ class, compatible with *Vitis HLS*. All the configurable parameters are set using class template arguments.

It complies with our *Cyclic dataflow protocol*: the cache task is a *Slave*, whose *Master* is the compute task. The *Master* operations are hidden behind the cache application programming interfaces, therefore end users of our cache are not required to be aware of the underlying protocol.

The cache task provides high throughput in steady state (it serves a hitting request in one CC), since it is optimally pipelined with $II = 1$ CC.

1) CACHE PIPELINE

The most critical factor that may increase the cache II, and hence reduce performance, is the external DRAM access at a generic address (generally, the compute task can access any array element, in any order), which introduce long-distance data and structural dependencies.

Considering that the external DRAM is accessed only in case of a miss, and that we want to optimize the hitting accesses, we extracted the AXI interface, in charge of accessing the DRAM, into the *Memory interface Slave* task. The associated *Master* is the *Core* task, which includes all the remaining cache logic (Fig. 7a). This solution removes the AXI interface dependencies from the cache hit logic, which is fully contained in the *Core* task. For misses, the *Core* task sends the DRAM access request to the *Memory interface* task, and it dynamically stalls until it receives the response.

To avoid both data and structural dependencies, cache helper data (e.g., `tag`, `valid`, `dirty`, ...) are stored in completely partitioned arrays, bound to registers since they are typically much smaller than the cache data.

To limit the register usage and to enable cache size scaling, the cache data memory is bound to BRAM. However, the BRAM read after write (RAW) latency of 1 CC makes the *Core* task RAW dependency on data memory (which exists because a newly loaded line may hit in the following access) to have a distance of 2 CCs. This would require the *Core* task to be pipelined with a II of 2 CCs.

To lower the II to 1 CC, we removed the dependency using a small auxiliary cache (*RAW cache*). It is a two-line fully associative cache, implemented with registers, providing the functions `get_line` (for hits, it reads the *RAW cache* line; for misses, it reads the *Dataflow cache* line), and `set_line` (it writes both the *Dataflow cache* line and the *RAW cache* line, according to the FIFO replacement policy).

The data memory of the cache is always accessed through the *RAW cache*, thus *ensuring that the dependency with a distance* 2 CCs *is false*. This is because the `set_line` function is at most called once per pipeline iteration: if a cache line is written, it will not be read in the next two iterations (which would be the origin of the distance 2 CCs dependency), since the RAW cache would hit and return its data directly.

2) AXI INTERFACE

The *Memory interface* task accesses the AXI bus at every request from the *Core* task. To save resources, it is not pipelined. Pipelining would rarely help, because a well-configured cache should never get multiple sequential misses, especially considering that there is one dedicated cache per source code array.

All DRAM accesses handle whole cache lines, which are sequential and aligned to the line size. To enable the HLS tool to infer that accesses are aligned, we explicitly zeroed the least significant bits of the address. This enables automated port widening and burst inference. If the line size is at most the maximum AXI interface width, it is accessed in a single request, else (more commonly) it is accessed in a burst request.

By default, *Vitis HLS* assumes *AXI latency* 64 CCs. This is useful to send pipelined requests on the AXI interface. However, our *Memory interface* is not pipelined. Thus, we set the AXI latency to zero, which makes the *Memory interface* stall after issuing a AXI request and resume right after the response, saving resources without losing performance.

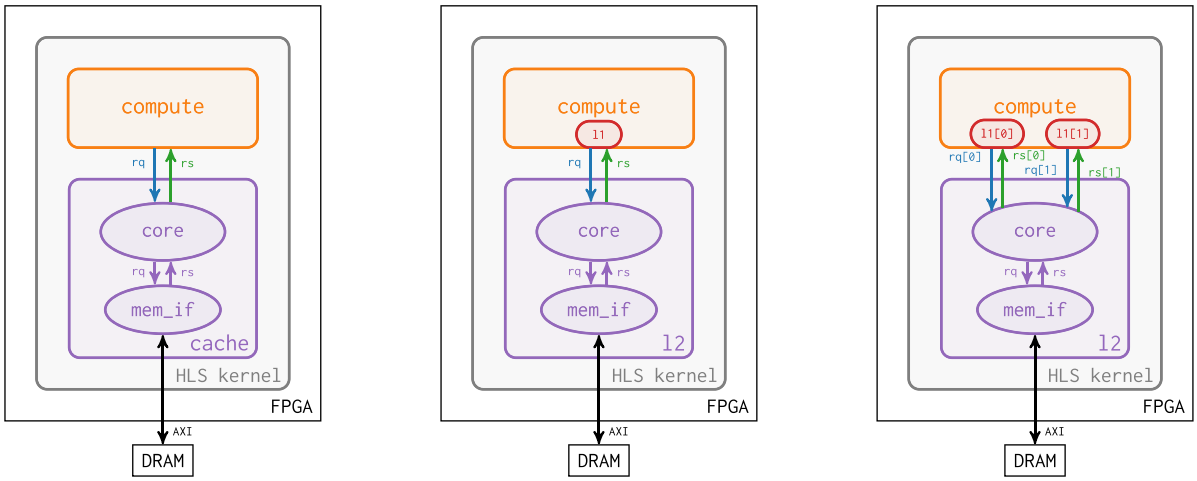
3) CACHE INTERFACE

To interface with the cache, we exposed the user-callable application programming interfaces (APIs) for managing requests and responses between the compute task and the cache.

- The `get` function accepts as input the address to read from cache and returns the read data. Internally it sends a read request (writing the address to the request FIFO), waits for the request-response latency (discussed later) in case of a hit or for longer in case of a miss, reads the data from the response FIFO, and returns the received data.
- The `set` function accepts as input the address and the data to write to the cache. Internally, it sends a write request (writing the address and the data to the request FIFO).

We overloaded the `operator[]` to automatically call the `get` and `set` functions, e.g., in Listing 1, `a[i] = a[i + 1]` is automatically compiled to `a.set(a.get(i + 1), i)`.

As discussed in Section III-A1, the request-response distance should match the cache latency. However, cache latency varies at runtime, as hits and misses (which have different latencies) are interleaved, depending on the access pattern and the cache configuration. Moreover, we need to distinguish between the different memory access types.



(a) *Dataflow cache*. (b) *Multi-level cache*. (c) *Multi-port cache*.

FIGURE 7. Baseline ***Dataflow cache*** architecture, and its extensions.

- For RO caches, the optimal distance value is typically around the average memory access latency $\overline{lat}$

$$\overline{lat} = lat_{\text{cache}} \cdot hit\ ratio + lat_{\text{DRAM}} \cdot miss\ ratio. \quad (1)$$

  lat_{cache} varies from 3 CCs to 5 CCs based on cache configuration and timing constraints, lat_{DRAM} depends on the target FPGA board, and *hit ratio* and *miss ratio* depend on the application and cache configuration.
- Read-write (RW) caches are affected by data dependencies with distances corresponding to the request-response distance. The latter should therefore balance cache performance and computation task performance (task II depends on the dependency distance). Experimental results (Section VI-D1) show that a 2 CCs distance typically gives the best overall performance.
- For WO caches, the request-response distance has no meaning because there is no response.

A template parameter is available to the users willing to fine-tune the distance of the caches in their designs.

IV. MULTI-LEVEL CACHE

The *Multi-level cache* extends the memory hierarchy of the cache by adding a level 1 (L1) cache on top of the *Dataflow cache*, i.e., the level 2 (L2) cache, as shown in Fig. 7b. This architecture is aimed at reducing the latency between a read access request and the corresponding response.

We are not interested in further accelerating the writes. Write latency has a negligible impact on performance, considering that they never stall the compute task (there is no response from the cache to the main computation), provided that the request FIFO is deep enough to accommodate all the pending writes. Moreover, write accesses are usually less frequent than reads.

Finally, the *Multi-level cache* is the starting point for enabling multiple concurrent accesses in the *Multi-port cache* described in Section V.

Similarly to the cache by Ma et al. [10], the L1 cache is inlined in the compute logic. This reduces the latency of the memory accesses by avoiding the inter-task communication. Even if the L1 cache is inlined, the compute task pipeline II is preserved, unlike the cache by Ma et al. [10]. This is because (i) in case of miss the L1 cache interacts with the L2 cache instead of with the external DRAM. Furthermore, (ii) the L1 cache complies with the write-through policy (the L1 cache aims at accelerating only the reads), introducing fewer dependencies compared with the write-back policy.

To implement the *Multi-level cache* architecture, we extended the *Dataflow cache* source code. In the *Dataflow cache*, the response flows from the L2 to the compute task and contains a single word. In the *Multi-level cache* architecture, the response flows from the L2 to the L1 cache, and holds a whole cache line.

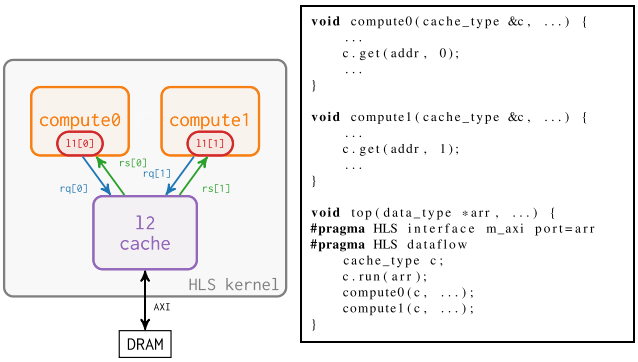
The *Dataflow cache* APIs were updated to support the L1 cache by adding the `get_line` function. Moreover, we upgraded the implementation of the `get` and `set` functions, while keeping their signature unchanged.

- The `get_line` function receives as input the address to read from cache and returns the line to which the address belongs. In particular, if the address hits the L1 cache, the line is read from the L1 cache. Otherwise, the request is issued to the L2 cache, as with the `get` function of the *Dataflow cache*.
- The `get` function calls the `get_line` function and returns the requested word only.
- The `set` function marks the L1 cache line as dirty, if it hits, according to the write-through policy. Additionally it forwards the write request to the L2 cache as with the *Dataflow cache*.

The L1 cache supports the set-associative mapping policy. The number of sets and ways of the L1 cache are configurable through template parameters. Note that when the L1 cache parameters are set to zero, the resulting architecture is equivalent to the *Dataflow cache*.

Similarly to the L2 cache, the L1 cache memory is bound to BRAMs and the helper data is bound to registers. Both the L1 and the L2 caches use the same memory technologies, therefore the L1 cache could have comparable or even bigger size than the L2 cache.

According to our experimental results (Section VI-B3), when a L1 cache is included on top of the L2 cache a convenient default value for the L2 request-response distance is 3 CCs for RO accesses, and 2 CCs for RW accesses. Note that the default RW distance is lower than the RO one because higher distance values would make the RAW dependencies distance longer and reduce the overall performance, as discussed in Section III-B3.



(a) Dataflow graph. (b) Source code.

FIGURE 8. Multiple-reader DRAM-mapped array, associated with our cache.

V. MULTI-PORT CACHE

The *Dataflow* and *Multi-level* cache architectures provide a maximum throughput of one access per CC. This is efficient for pipelined algorithms, which access each cached array at most a single time per iteration. To efficiently implement algorithms which access the same array multiple times per iteration (either due to the user code or after a loop unrolling), we designed the *Multi-port cache* that enables multiple concurrent read accesses to the same array.

With the *Multi-port cache* architecture, a shared L2 cache exposes an arbitrary number of ports, each with a private L1 cache, as shown in Fig. 7c. The private L1 caches enable scheduling multiple memory accesses at the same time, without increasing the II of the compute task.

Hence, unlike Ma et al. [10], the L1 cache does not use directly the single DRAM interface, but goes through the shared L2 cache. Thus, we do not require users to manually mark explicitly some accesses as "always hit" (through the `retrieve` and `modify` functions), which would require extensive manual analysis and code changes and may lead to incorrect behavior.

To keep the cache logic simple and to avoid negatively affecting the compute task II, we did not implement any coherency mechanism. To guarantee the correct functionality, the *Multi-port cache* only supports read accesses. The extension to concurrent write or RW accesses is left to future work.

The *Multi-port cache* is implemented as an extension of the *Multi-level cache*. The number of ports P can be configured through a template parameter. When it is set to one, the architecture is equivalent to the *Multi-level cache*.

The *Core* task of the L2 cache was updated to cycle over each port, i.e., it sequentially serves the requests from the first to the last port, before restarting from the first one. Any port that did not send any request is skipped. This code pattern (hidden from the user behind the cache `operator[]`) can be optimized by the HLS tool to statically schedule P array accesses with $II = 1$ CC in most cases.

For each port, we allocate a private L1 cache, and the related pair of request and response FIFOs (to communicate with the shared L2 cache).

The access port can be selected either automatically or manually, when the user-friendly automatic port selection does not lead to the desired II for the algorithm pipeline.

- With the automatic selection, each call to `get_line` (which is in turn called by `get`) is automatically associated to a specific port by means of a member variable holding the port index, which is updated after each access. This is implemented directly in the `get` function, that keeps track of the last accessed port and uses this information to bind a specific request to a specific port.
- The manual port selection allows one to explicitly inform the tool that each access uses different address and data streams, and that the dependencies are false. It is implemented by adding the `port` parameter (which identifies the number of the port to be accessed) to the `get` function (in this case the `operator[]` cannot be used).

In addition to the performance advantage, our *Multi-port cache* allows overcoming the *Vitis HLS* limitation of a single reader per AXI interface. Indeed, each L2 cache (associated with a single AXI interface) can expose multiple ports in the form of pairs of request/response FIFOs. These ports can connect the L2 cache to one or more `compute` dataflow tasks. Since the L2 cache ignores the ports with no pending requests, the `compute` tasks can seamlessly issue requests to the L2 cache at different rates. Figure 8 shows the dataflow graph of a kernel with a DRAM-mapped array that is read from two `compute` dataflow tasks, through a single L2 cache. Additionally, each `compute` task has its own private L1 cache. In *Vitis HLS*, if designers need to access the same DRAM array from different dataflow tasks, they must instantiate multiple AXI bundles, associated to the same underlying buffer in DRAM. Note that, due to the loose synchronization between dataflow tasks in *Vitis HLS*, both a dual-ported cache and a pair of bundles can be used meaningfully only for read-only arrays. Otherwise enforcing cache coherency or preserving data dependencies in a shared array between two processes would be very difficult.

VI. EVALUATION

To evaluate the impact of the proposed cache architecture in terms of PPA, we used the cache in some memory-intensive benchmarks with very different access patterns. *We selected two "classical", frequently used algorithms (matrix multiplication and convolution), since they are widely known and provide good and easy to understand examples. In practice, our caches should be used either (i) for seldom used algorithms, for which a manual optimization effort would not be justified, or (ii) for those that do not exhibit regular access patterns, such as bitonic sorting, which is our third benchmark.*

We synthesized the benchmarks as *Vitis HLS* kernels and deployed them on a physical FPGA board to measure the resulting PPA. The experimental workflow consists of: (1) SW simulation, (2) HLS synthesis, (3) logic synthesis, place and route, and bitstream generation, and (4) execution and measurements.

Steps (1) and (2) were performed in *Vitis HLS* 2021.2 (using *Vitis* flow defaults), and step (3) in *Vivado* 2021.2 [19] (using *Vivado* defaults for synthesis and implementation). All steps targeted the *Avnet Ultra96v1* [20] board, hosting a *Xilinx Zynq UltraScale+* FPGA. Figure 9 shows the block design for implementing a HLS kernel with three DRAM-mapped arrays (such as the matrix multiplication and convolution test cases). Given an algorithm (which determines the number of inputs and outputs, and by consequence of the AXI interfaces), the HLS kernel exposes the same interface, even when it is optimized with our cache, since the cache is fully implemented with HLS inside the kernel itself.

The board runs the *PYNQ Linux 2.7.0* [21] operating system, whose *PYNQ* library is exploited in step (4).

We collected the data from different sources:

- *SW simulation reports*
  - *Hit ratio*: ratio between the number of requests that hit data in cache and the number of all requests for a specific cache memory.
- *Post place and route reports*
  - *Area*: number of lookup tables (LUTs), flip-flops (FFs), BRAMs and digital signal processing units (DSPs) required to implement the whole design.[2]
  - *Maximum clock frequency*: the maximum frequency at which timing was met by the implementation flow, achieved by gradually increasing the clock frequency constraint. The frequency higher bound is 333 MHz, that is the maximum supported frequency for the AXI adapter (330 MHz in practice, due to the clock generation logic limited precision).

[2]It is virtually impossible to accurately report only the resource usage of our caches, because our caches are not separate RTL modules which interface with the kernel to be accelerated, but they are synthesized together with the kernel, and are not separable from the kernel logic. To get a rough approximation of the cache resource overhead, we can only subtract the resource usage of the kernel without any cache (later referred as *Baseline*) from that of the kernel with our caches. In case of multiple ports, even this approximation cannot be applied, since the loop unrolling enabled by the cache increases both the resource usage and the performance of the application itself.

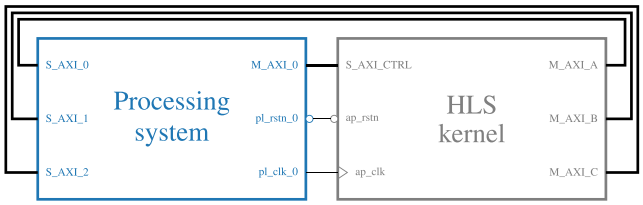


FIGURE 9. Block design with three DRAM arrays.

- *Runtime measurements*
  - *Performance* (t_{ex}): execution time, measured between the assertions of the start and the end signals of the kernel.
  - *Power* (P): average power, measured by the sensor on the system power rail during kernel execution. Note that the selected board does not allow measuring the power of the FPGA only, therefore P is the power consumed by the whole board, including the CPU.

  The measured quantities are not fully deterministic. The timer measuring t_{ex} may not be stopped at the exact time when the kernel asserts its end signal, since it checks this condition through polling and the CPU might be busy running other tasks of the operating system. Also, power consumption is affected by different factors, such as the CPU load or the temperature. Thus, each runtime measurement was taken five times and is collected as the average and the standard deviation of these measurements. The energy consumption (E) is computed as the average energy, $E := P\,t_{ex}$.

To limit the design space, in all the cache configurations we used a default L2 cache request-response latency. For single-level RO configurations, we computed the default distance value as 7 CCs, according to (1), where the $\text{lat}_{\text{cache}}$ was set to the worst-case, i.e., 5 CCs, lat_{DRAM} was set to 40 CCs according to the measurements by Marjanovic [22] of the read latency of the high-performance coherent (HPC) ports of the target board, and hit ratio was assumed to be 95 % (these values were only used to set the cache parameters, while the runtime results reflect the real latencies and hit ratios). The experiments show that these approximations achieve good pipeline performance. A significant performance degradation is observed only if one assumes very low (1 CC), or very high (more than lat_{DRAM}) distance values. We used a default distance of 3 CCs for multi-level, and 2 CCs for RW cache configurations. Write-only cache configurations are unaffected by the distance parameter.

In order to compare directly the cycle count performance of the various designs, we constrained the clock frequency to 100 MHz in all experiments, except for those that are related to the timing impact of the cache (Sections VI-B2, VI-C1 and VI-D2).

We manually chose the cache parameters, such as the line size, number of lines, and so on, based on the array access patterns. However, there are multiple methods to automate the selection of these parameters, as attested by a large amount

of past work, for example those analyzed by Upadhyay et al. [23]. Integration of those approaches with our cache is left to future work.

A. REFERENCE DESIGNS

We compared the collected results with:

1) *Baseline*: the kernel generated by default by the HLS tool, whose computational core directly accesses the external DRAM through the AXI interface.
2) *RTLCache*: the *Baseline* HLS kernel, with the Xilinx System Cache RTL module inserted in between the HLS kernel and the AXI DRAM interface (when the cache module configurability allows a setup with non-zero hit ratios).
3) *Manual*: the kernel manually optimized for buffering the data using the on-chip memories (when the memory access patterns allow it).

1) Ma et al. [10] CACHE REFERENCE

Ma et al. [10] reported results collected from unreliable sources. They collected the area figures from post-HLS-synthesis reports, which are estimations known to be affected by significant errors. Moreover, they estimated performance and power data using RTL simulation, which is based on simplified models (especially for the AXI model, the DRAM controller, and the DRAM itself), which are crucial in this context. Additionally, due to the long execution time of the RTL simulations, their input sizes were limited to small values.

Nevertheless, since their code is open source, we tried to generate results comparable to ours by applying our implementation flow to their cache. We first adapted their cache, (designed for *Vivado HLS*) to *Vitis HLS*. The changes involved only their APIs, not the HW. However, using *Vitis HLS* for the kernels embedding their cache generates very poorly performing HW, e.g., the matrix multiplication innermost loop achieved $II = 141$ CC instead of 1 CC in their tests using *Vivado HLS*, and the bitonic sorting loop was not pipelined at all in *Vitis HLS*. Therefore, we stopped the implementation flow at the HLS synthesis step, since their cache would perform even worse than the *Baseline* that achieves better pipelining, and we avoided any further comparison.

2) INTEL CACHE REFERENCE

To evaluate the caching capabilities of the Intel LSUs [24], we used the *Intel DevCloud* environment, which provides the Intel HLS tool and enables remote access to an *Intel programmable acceleration card* hosting an *Arria 10 GX* FPGA. The tool automatically allocates an LSU for each off-chip array, and each RO LSU can include a cache. The cache characteristics (number of words per line, number of sets, number of ways, ...) are determined automatically and are not reported to the user, who can optionally control only the total cache size.

Algorithm 2 Standard Matrix Multiplication

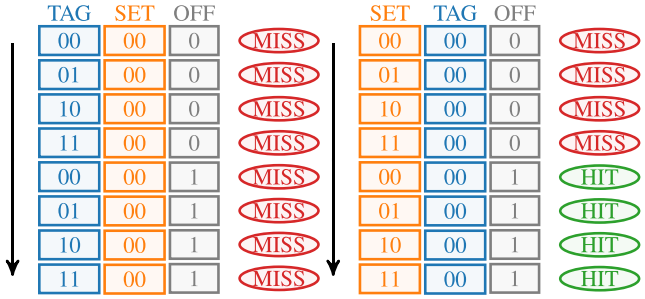
Require: $A \in \mathbb{R}^{N\times M}, B \in \mathbb{R}^{M\times P}, C \in \mathbb{R}^{N\times P}$
Ensure: $C = A \times B$
procedure StdMatMult(A, B, C)
  LOOP_I: **for** ($i \leftarrow 0; i < N; i \leftarrow i+1$) **do**
    for ($j \leftarrow 0; j < P; j \leftarrow j+1$) **do**
      $acc \leftarrow 0$
      LOOP_K: **for** ($k \leftarrow 0; k < M; k \leftarrow k+1$) **do**
        $acc \leftarrow acc + A[i][k] \cdot B[k][j]$
      end for
      $C[i][j] \leftarrow acc$
    end for
  end for
end procedure

We analyzed the PPA impact of the LSUs by running some experiments using a standard matrix multiplication (Algorithm 2). The tested configurations include (a) the automatic test case, in which we did not set the cache sizes, (b) the lower-bound test case, in which we set all the cache sizes to 0, and (c) the upper-bound test case, in which we set the caches to fit the whole matrices. Compared with the lower-bound test case, the automatic case is 8 % faster and the upper-bound is 80 % faster. The automatic cache parameters selection is therefore suboptimal. Most probably because one matrix is accessed by columns, hence with limited locality. Moreover, the performance advantages are quite limited even in the upper-bound case, when the matrices are entirely stored to cache. This is because the *Intel Arria 10* has a low off-chip memory latency, from 3 CCs to 23 CCs [25]. We did not have access to an Intel FPGA with a higher off-chip memory latency, which would make the cache impact more significant. Thus, the low-latency of off-chip memory coupled with the limited control over the LSU cache parameters prevented us from performing a more thorough comparison with our cache.

B. MATRIX MULTIPLICATION

The *Matrix Multiplication* (*MatMult*) standard implementation (*StdMatMult*) is shown in Algorithm 2. It accesses each matrix according to a specific pattern:

- A is accessed by rows, and each row is accessed P times, for a total of $N \times M \times P$ memory accesses. Its cache should fit a matrix row at a time.
- B is accessed by columns, and each column is accessed P times, for a total of $N \times M \times P$ memory accesses. Since the matrix is stored in row-major order, the spatial locality is very poor. To get a non-zero hit ratio, we need either an expensive M-way fully associative cache, or a more efficient M-set direct-mapped cache exploiting the swapped address bit mapping (Fig. 2b).
  With an M-set direct-mapped cache, the standard address bit mapping (Fig. 10a) results in subsequent accesses to the same set with new tags leading to



(a) Standard address bit map. (b) Swapped address bit map.

FIGURE 10. ***MatMult*: sequence of addresses of *B* accessed during the first 8 iterations, where $B \in \mathbb{R}^{4\times8}$ has a 4-set direct-mapped cache.**

Algorithm 3 Block Matrix Multiplication

Require: $A \in \mathbb{R}^{N\times M}, B \in \mathbb{R}^{M\times P}, C \in \mathbb{R}^{N\times P}, BLK \in \mathbb{N}$
Ensure: $C = A \times B$
procedure BlkMatMult(A, B, C)
  for $(jj \leftarrow 0; jj < P; jj \leftarrow jj + BLK)$ **do**
    for $(kk \leftarrow 0; kk < M; kk \leftarrow kk + BLK)$ **do**
      LOOP_I: **for** $(i \leftarrow 0; i < N; i \leftarrow i + 1)$ **do**
        for $(j \leftarrow jj; j < jj + BLK; j \leftarrow j + 1)$ **do**
          $acc \leftarrow 0$
          LOOP_K: **for** $(k \leftarrow kk; k < kk + BLK; k \leftarrow k + 1)$ **do**
            $acc \leftarrow acc + A[i][k] \cdot B[k][j]$
          end for
          $C[i][j] \leftarrow C[i][j] + acc$
        end for
      end for
    end for
  end for
end procedure

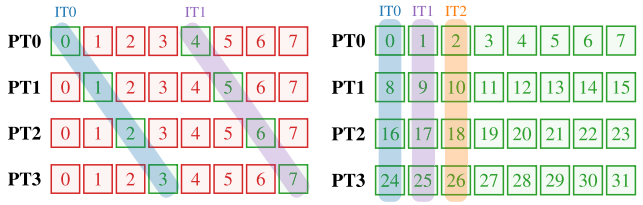
continuous cache line overwriting and misses. Our custom address bit mapping (Fig. 10b) enables instead subsequent reads to access distinct sets with the same tag and yields a high hit ratio.

- C is accessed sequentially, once. A single-line n-word cache provides $n-1$ hits every n accesses, making write burst inference easier.

The *StdMatMult* algorithm requires the B cache to have M lines. While this is feasible with relatively small matrices, it cannot scale up with matrix sizes.

To make the cache configuration independent of M, and ensure scalability, we also implemented a blocked matrix multiplication (*BlkMatMult*) algorithm (Algorithm 3). It is a commonly used efficient implementation of *MatMult*, which accesses all matrices by blocks, instead of columns, to improve the spatial locality of accesses to the B matrix:

- A is accessed by sub-rows, within a block. Each sub-row, of BLK elements, is accessed BLK times. Therefore, the A cache should fit a block row at a time.
- B is accessed by sub-columns, within blocks. Each block is accessed N times, therefore its cache should fit one block at a time.
- C has the same access pattern as A, but its cache requires up to BLK ways to provide non-zero hit ratio when the partial unrolling (discussed later) is applied to the innermost loop.



(a) *Horizontal* unrolling. (b) *Tiled* unrolling.

FIGURE 11. ***MatMult*: content of L1 caches of *A* during the first iterations, where $A \in \mathbb{R}^{4\times8}$ is associated with a four-port single-line cache with eight words. *PTn* identifies the *n*-th port. The green boxes represent elements that read during execution, red boxes are elements loaded in cache but never accessed. The numbers inside the boxes are the addresses of the elements of the *A* matrix. *ITi* highlight the elements accessed in parallel at the *i*-th iteration.**

In all implementations, the algorithm innermost loop (`LOOP_K`) was pipelined with $II = 1$. The implementation was further optimized through loop unrolling by a factor UF.

For *StdMatMult*, we considered two kinds of unrolling:

- *Horizontal*: unrolls the innermost loop (`LOOP_K`). To keep $\text{II} = 1$ for `LOOP_K`, each iteration of the unrolled loop is assigned to one of the UF A and B cache ports.
  Figure 11a highlights a fundamental limitation of this unrolling approach when combined with multi-port caches. The data is replicated in each cache, but only one every UF elements is actually used, thus leading to significant resource and performance waste.
- *Tiled*: divides the `LOOP_I` iteration count by UF and adds a fully unrolled inner loop [26]. All iterations of that new loop use the same element of B and a different one of A. Therefore, the B cache is single-port, while the A cache has UF ports. With this approach, each A port contains different data (Fig. 11b). The hit ratio is preserved as the unrolling factor scales up and no resources are wasted. All the elements loaded into the cache are actually used, allowing the algorithm to run at full speed for as many iterations as the words per cache line, significantly improving the performance with the same resource usage as *Horizontal*.

In *BlkMatMult* we exploited the *Tiled* unrolling only, for similar reasons to *Tiled StdMatMult*. To maximize the performance, we doubled UF until we used all the resources of our (small) FPGA.

All the *MatMult* tests use the same matrix sizes, $N = P = 1024$, $M = 128$, and data type of 32-bit integers.

Table 1 shows the cache configurations tested with *StdMatMult*, while Table 2 summarizes the *BlkMatMult* ones. We tested block sizes of 16, 32, and 64.

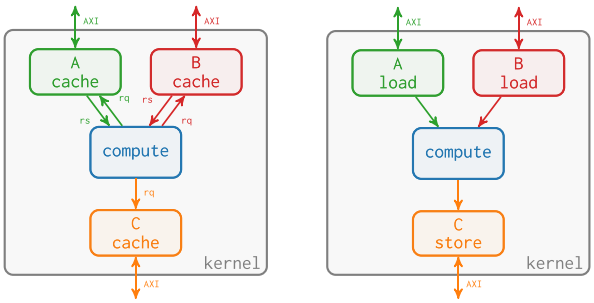
As a reference, we implemented the *Baseline* test case. The unrolling, applied to the *Baseline* test case, would be detrimental, since the II of `LOOP_K` would dramatically increase due to structural dependencies on the AXI interface (which exposes one port only), resulting in performance degradation. Therefore, our cache enabled us to conveniently unroll the algorithm loop, without any change to the algorithm itself.

TABLE 1. ***StdMatMult*: tested cache configurations.**

Implementation	Array	Words	L2 sets	L2 ways	L1 sets	L1 ways
Single-level (L2)	A	$M/2$	2	1	0	0
	B	32	M	1	0	0
		64	M	1	0	0
	C	32	1	1	0	0
		64	1	1	0	0
Horizontal (L1)	A	$M/2$	1	1	2	1
	B	32	1	1	M/UF	1
		64	1	1	M/UF	1
	C	32	1	1	0	0
		64	1	1	0	0
Tiled (L1tld)	A	$M/2$	1	1	2	1
	B	32	1	1	M	1
		64	1	1	M	1
	C	32	1	UF	0	0
		64	1	UF	0	0

TABLE 2. ***BlkMatMult*: tested cache configurations.**

Implementation	Array	Words	L2 sets	L2 ways	L1 sets	L1 ways
Single-level (L2blk)	A	BLK	1	1	0	0
	B	BLK	1	BLK	0	0
	C	BLK	1	BLK	0	0
Multi-level (L1blk)	A	BLK	1	1	1	1
	B	BLK	1	1	1	BLK
	C	BLK	1	BLK	0	0



(a) Our cache. (b) LCS.

FIGURE 12. ***MatMult*: tested dataflow architectures.**

The *Manual* test case optimizes the design according to the LCS pattern. All the off-chip memory accesses use the maximum AXI interface bitwidth of the board (128 bits, or four 32-bit elements per transaction). The B load task reads the B matrix once, four columns at a time. The A load task reads the A matrix multiple times, in bursts. The compute task computes 16 multiply-acccumulate (MAC) operations per CC. The store task stores four elements of C at a time.

Figure 12 compares the dataflow architecture generated with our caches with the LCS one. The similarity between the two architectures is very strong: the only major difference is the absence of the request FIFO from the compute to the load tasks, in case of the LCS architecture. This is because the input data address computation must be factored out of the compute task and moved into the load and store tasks to implement the LCS paradigm. This refactoring is the major design cost that our cache alleviates.

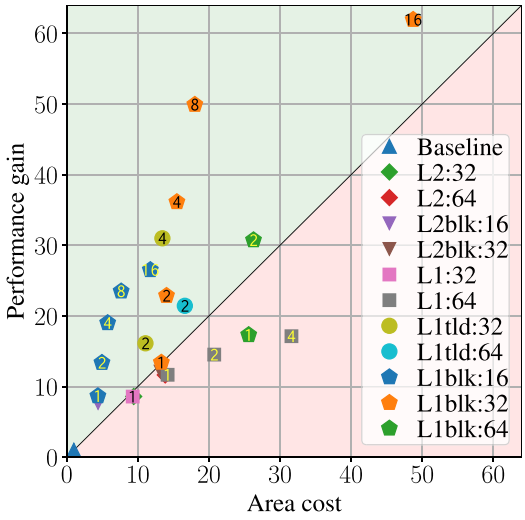


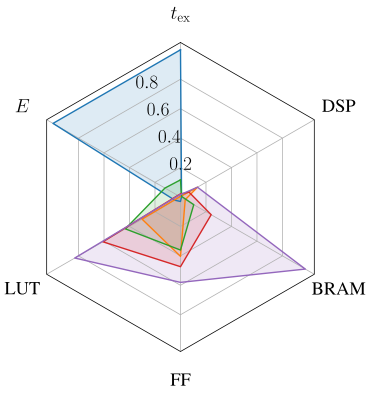
FIGURE 13. ***MatMult*: performance gain (t_{ex} relative to *Baseline*) with respect to area cost (average of LUTs, FFs, BRAMs, and DSPs usage relative to *Baseline*). *StdMatMult Single-level* is labelled *L2:WORDS*, *Horizontal* is *L1:WORDS*, and *Tiled* is *L1tld:WORDS* (*WORDS* are the number of words per line of *B* and *C* caches). *BlkMatMult Single-level* is labelled *L2blk:BLK*, and *Multi-level* is *L1blk:BLK* (*BLK* are the block sizes). The numbers over the markers are the unrolling factors.**

For the *RTLcache*, due to the limited configuration options of the Xilinx System Cache (it provides only two or four ways, and it does not support our custom address bit mapping), the best performing configuration in that case is the *BlkMatMult* algorithm, with block size equal to four.

The cache configurations selected for the test cases reach high hit ratios, above 96 % for *StdMatMult* and 99 % for *BlkMatMult*. Figure 13 shows the performance gain, i.e., $t_{ex,Baseline}/t_{ex}$, with respect to the area cost, i.e., $(\text{LUT}/\text{LUT}_{Baseline}+\text{FF}/\text{FF}_{Baseline}+\text{BRAM}/\text{BRAM}_{Baseline}+\text{DSP}/\text{DSP}_{Baseline})/4$, of the test cases embedding our caches. Most of the points are in the "green" area, where t_{ex} speedup is larger than the resource overhead.

Figure 14 shows the detailed data for some significant test cases, including (a) the reference test cases, i.e., *Baseline* and *Manual*, (b) the least resource-demanding cache configuration with the *StdMatMult* algorithm, i.e., *L2:32*, (c) the most convenient cache configuration in terms of performance gain and area cost ratio, i.e., *L1blk:32 (8 ports)*, and (d) the fastest cache configuration, i.e., *L1blk:32 (16 ports)*. Compared with the test cases with caches, the *Manual* design provides better overall quality of results. However, the aim of our work is not to achieve better PPA than manual optimizations, but rather to get significantly better quality of results (with respect to the *Baseline*), while greatly reducing the design effort.

Note that increasing the number of ports of the caches, and hence their resources, uniformly increases performance. Figure 15 shows the results of using regression to predict the resource usage to achieve a given execution time with our cache. According to this model, to achieve performance on par with the *Manual* reference design, our caches would



	Baseline	Manual	L2:32	L1blk:32 (8 ports)	L1blk:32 (16 ports)
t_{ex} (s)	31.98	0.13	3.72	0.64	0.52
P (W)	4.35	4.83	4.56	4.67	4.68
E (J)	139.3	0.62	16.96	2.99	2.41
LUT	3104	21259	30534	42954	58515
FF	4292	56905	50866	66810	81863
BRAM	1.5	8	22.5	52	211.5
DSP	3	48	3	24	48
Perf. gain	1.0	246.0	8.6	50.0	62.7
Area cost	1.0	10.4	9.4	18.0	48.7

FIGURE 14. *MatMult*: PPA of some significant test cases. t_{ex} and E are relative to the *Baseline*. The resource usages are relative to the total resources provided by the target FPGA.

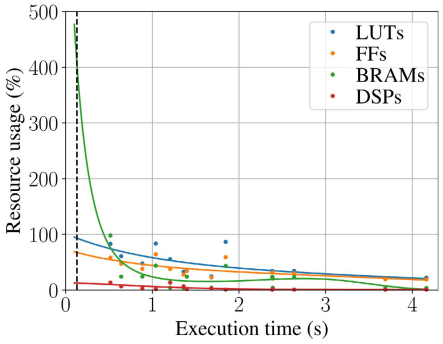


FIGURE 15. *BlkMatMult*: regression estimating the resource usage with respect to the execution time of the test cases with our caches. The dashed vertical line highlights the execution time of the *Manual* test case. The dots are the real data, the lines are the regression predictions.

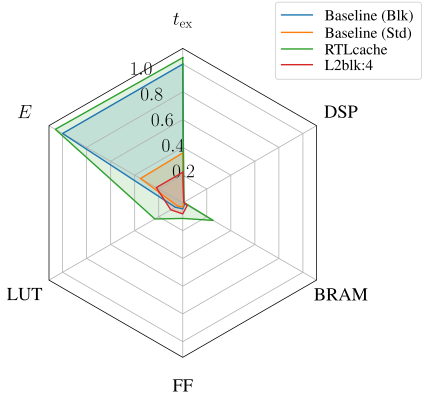


FIGURE 16. *MatMult*: PPA of some test cases related to the *RTLcache* case. t_{ex} and E are relative to the *Baseline (Blk)*.

require 4 times the available BRAMs, while the other kinds of resources would be sufficient.

1) MATRIX MULTIPLICATION RTL CACHE TEST CASE

The Xilinx System Cache supports only two or four ways. Therefore, the theoretically most performant setup is with *BlkMatMult* with block size four (which is still too small to provide large performance gains). The caches associated with *A* and *C* should be single-line, while the cache associated with *B* should provide four ways, each of four words. However, the Xilinx System Cache minimum size is 32 kB, with at least two ways and 64 B per line, therefore the caches of the *RTLcache* test case are dramatically oversized. On the contrary, the test case with our cache (*L2blk:4*), thanks to its fine-grained configurability, was set up to allocate only the resources that are actually needed. Figure 16 summarizes the results of these tests. For reference, besides the usual *Baseline (Std)* test case, that implements an unoptimized version of *StdMatMult* algorithm, we also included the *Baseline (Blk)* test case, which implements the unoptimized *BlkMatMult* algorithm with block size four. We included it to quantify the impact of the Xilinx System Cache on the very same kernel, directly connected to the AXI interface.

Both the *Baseline (Blk)* and the *RTLcache* designs are significantly slower than the *Baseline (Std)*. For the *Baseline (Blk)*, this is because the *BlkMatMult* is not meant for running without a cache. For the *RTLcache*, this is because the RTL cache module is inserted a posteriori (after HLS), thus the kernel is synthesized assuming that all the memory references access the off-chip memory. Therefore, it is scheduled to wait for the expected latency of the AXI master controller that is used to access DRAM, which has a minimum latency, hardcoded into the HLS scheduler, of *at least* 7 CCs. Thus for cache hits it waits for much longer than needed, while for misses it waits for shorter than needed (the cache introduces an additional latency when missing), and then it stalls until the memory request is fulfilled. On the other hand, our dataflow protocol hides from the computation process schedule the fact that it is accessing DRAM, thus allowing it to achieve the best throughput in case of cache hits.

The result is that the RTL cache is not only unable to provide any advantage, but it also slightly worsens the performance and the energy consumption. Moreover, it also introduces a large area overhead, due to the oversized caches.

The *L2blk:4* test case is significantly faster than the *Baseline (Blk)*, proving the effectiveness of our HLS cache implementation with respect to the System Cache. However, it is not much faster than the *Baseline (Std)*, since the small block size limits the performance advantage.

TABLE 3. ***MatMult*: maximum achievable clock frequency of some test cases. The relative maximum clock frequency is normalized over the maximum clock frequency of the AXI adapter (330 MHz).**

Test case	Maximum clock frequency (MHz)	Relative maximum clock frequency (%)
Baseline	330	100
Manual	330	100
L2:32	330	100
L1:32	330	100
L2blk:32	260	79
L1blk:32 (1 p)	250	76
L1blk:32 (16 p)	150	45

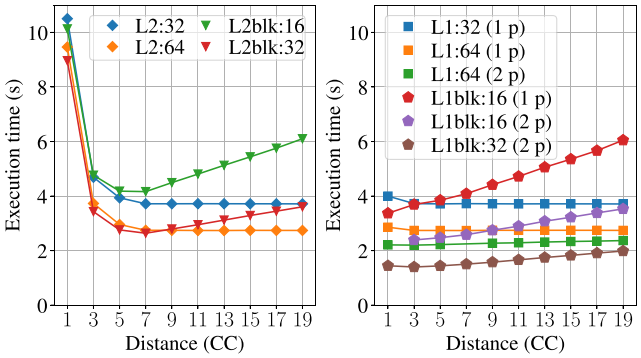
2) MATRIX MULTIPLICATION TIMING ANALYSIS

To evaluate the impact of our cache on the critical path, we measured the maximum clock frequency of some test cases. Table 3 reports the results of the experiments. The *Baseline* design is very simple: it consists of a loop that computes a multiply-acccumulate operation per iteration using a DSP (which is one of the fastest resources on the FPGA). Therefore, it is able to achieve the maximum clock frequency of 330 MHz. With the *StdMatMult*, all the instantiated caches are direct-mapped (including the *B* one, thanks to our custom address bit mapping). The resulting design can still run at 330 MHz, even in the *Multi-level* configuration. The *BlkMatMult* test cases require 32-way fully-associative caches. The high number of ways makes these caches inherently more complex than the direct-mapped ones, therefore they introduce a critical path which limits the maximum clock frequency. The *Single-level* configuration can run at a clock frequency up to 260 MHz. For the *Multi-level* configurations, the single-port test case can reach a clock frequency of 250 MHz. The extreme case with 16 ports can only reach a maximum clock frequency of 150 MHz. This is not only due to the more complex cache architecture, but also because of the algorithm unrolling, and the high resource utilization.

3) MATRIX MULTIPLICATION REQUEST-RESPONSE DISTANCE

To check the efficiency of the approximations for the default L2 cache request-response distance of RO cache configurations, we characterized the t_{ex} with respect to the distance in some test cases. For the *Single-level* configurations, Fig. 17a shows that in all test cases a distance of 1 CC results in a very high t_{ex} since it prevents exploiting the cache pipelining, as discussed in Section III-A1. The t_{ex} significantly decreases with the distance up to 5 CCs to 7 CCs. For higher distances, the t_{ex} of the *StdMatMult* test cases is approximately constant, while the one for *BlkMatMult* increases again. These results suggest that our choice of a default distance of 7 CCs is effective.

For the *Multi-level* configurations, Fig. 17b shows that the t_{ex} of *StdMatMult* is roughly constant with the distance, apart from the distance of 1 CC which is slightly slower. The *BlkMatMult* t_{ex} is instead directly proportional (by a small factor) to the distance. Any distance value between 1 3CCs should be a balanced choice. Our default value of 3 CCs is therefore well suited.



(a) *Single-level MatMult*. (b) *Multi-level MatMult*.

FIGURE 17. ***MatMult*: execution time with respect to L2 cache request-response distance.**

Algorithm 4 2D Convolution

Require: $A \in \mathbb{R}^{N \times M}$, $ker \in \mathbb{R}^{P \times Q}$
Ensure: $B \in \mathbb{R}^{N \times M} : B = A * ker$
procedure Conv(ker, A, B)
  for $(i \leftarrow 0; i < N; i \leftarrow i+1)$ **do**
    for $(j \leftarrow 0; j < M; j \leftarrow j+1)$ **do**
      $tmp \leftarrow 0$
      LOOP_M: **for** $(m \leftarrow 0; m < P; m \leftarrow m+1)$ **do**
        LOOP_N: **for** $(n \leftarrow 0; n < Q; n \leftarrow n+1)$ **do**
          $ii \leftarrow i + m - Q/2$
          $jj \leftarrow j + n - P/2$
          if $(ii \geq 0 \ \& \ ii < N \ \& \ jj \geq 0 \ \& \ jj < M)$ **then**
            $tmp \leftarrow tmp + A[ii][jj] \cdot ker[m][n]$
          end if
        end for
      end for
      $B[i][j] \leftarrow tmp$
    end for
  end for
end procedure

C. 2D CONVOLUTION

Algorithm 4 implements the *2D Convolution* (*Conv2D*). Each matrix is characterized by a specific memory access pattern.

- A is accessed according to a window pattern with size $P \times Q$ and stride one.
  A cache associated with A requires P ways to achieve a high hit ratio, since all the lines belonging to a window can be stored in the cache, *effectively implementing a line buffer without source code changes*.
  Cache lines sizes of $n \times Q$ enable prefetching n windows. To keep in cache windows which are not aligned to the cache line size, the cache should have two sets.
- ker is accessed $N \times M$ times, by rows. Since its size is typically small, its cache can be configured to fit the whole ker in the L1 cache.
- B is sequentially accessed once per element. B has a low impact on performance, since it is accessed only once

TABLE 4. *Conv2D*: tested cache configurations.

Implementation	Array	Words	L2 sets	L2 ways	L1 sets	L1 ways
Single-level (L2)	A	$n \cdot Q$	2	P	0	0
	ker	Q	1	1	P	1
	B	32	1	1	0	0
Multi-level (L1)	A	$n \cdot Q$	1	1	2	P
	ker	Q	1	1	P	1
	B	32	1	1	0	0

every $P \times Q$ accesses to A and ker. A single-line cache helps HLS to efficiently infer bursts.

All test cases use the same 8-bit integer data type and matrix sizes: $N = 1080$, $M = 1920$, $P = Q = 15$. In all implementations, the innermost loop (`LOOP_N`) was pipelined with $II = 1$ CC.

In the tests including our cache, each matrix was associated with a cache configured according to the previous considerations. Table 4 summarizes the tested cache configurations, where n is 1, 2, 4, 8, and 16. Since our cache only supports power-of-2 words, ways, and sets, all the parameters were rounded to the next power of 2.

With the multi-level test cases, we further improved the performance exploiting the multi-port feature to enable partial loop unrolling while keeping the II of `LOOP_K` at one (by setting the number of ports of A and ker as the unrolling factor). We unrolled `LOOP_M`, instead of the innermost `LOOP_N`, for reasons similar to those explained in Section VI-B, and shown in Fig. 11. We tested unrolling factors of 3, 5, 8, and 15 (complete unroll).

The *Manual* reference design was implemented by Xilinx Inc. [27], according to the LCS pattern. It is not possible to implement a meaningful *RTLcache* test case, since the Xilinx System Cache can only provide up to 4 ways, but the A cache requires at least 15 ways to achieve a sufficiently high hit ratio.

All tested cache configurations had hit ratios higher than 99 %. Figure 18 shows the trade-offs between performance and area, in different test cases. Figure 19 shows the details of some relevant test cases, including (a) the reference designs, i.e., *Baseline* and *Manual*, (b) the least resource-demanding cache configuration, i.e., *L2:16*, (c) a cache configuration balanced between performance and resources, i.e., *L1:64 (5 ports)*, and (d) the fastest cache configuration, i.e., *L1:64, (15 ports)*.

Our caches introduce multiple trade-offs in the PPA space, which perform better than the *Baseline* case, in exchange for higher resource usage. The *Manual* design is significantly faster than all the tested cache configurations, since it is able to process a whole window per CC (255 multiply-acccumulate operations per CC), while our cache configurations process at most one window column per CC (15 multiply-acccumulate operations). Figure 20 again shows the results of using regression to predict the resource usage to

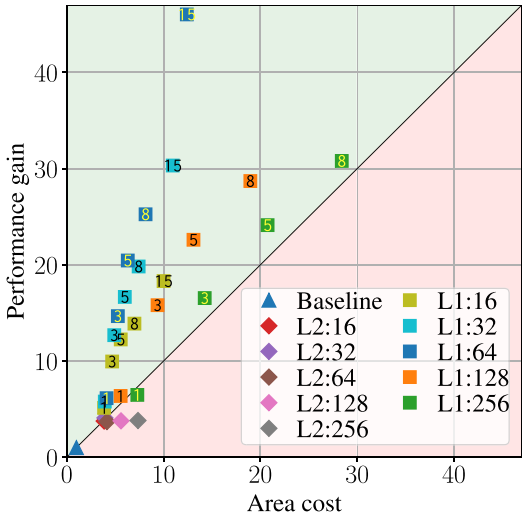


FIGURE 18. *Conv2D*: performance gain with respect to area cost. *Single-level Cache* is labelled as *L2:WORDS*, and *Multi-level* as *L1:WORDS*. The *WORDS* suffix stands for the number of words per line of the *A* cache.

TABLE 5. *Conv2D*: maximum achievable clock frequency of some test cases.

Test case	Maximum clock frequency (MHz)	Relative maximum clock frequency (%)
Baseline	330	100
Manual	330	100
L2:16	330	100
L1:16 (1 p)	330	100
L1:64 (15 p)	200	61

achieve a given execution time with our cache. According to the regression prediction, to achieve performance on par with the *Manual* reference design, our caches would require roughly 50 % more LUTs than those available on the target FPGA, while the other kinds of resources should suffice.

Note that the objective of our cache is not to compete with manually optimized designs, but rather to introduce new trade-offs between PPA and design effort. Our caches provided suboptimal results in terms of PPA, but required very low design effort, while being much more efficient than the designs automatically generated by the HLS tool, both in terms of execution time, reduced by up to 46 times, and in terms of energy consumption, reduced by up to 44 times, at the cost of an area overhead up to 12 times.

1) 2D CONVOLUTION TIMING ANALYSIS

Table 5 reports the maximum clock frequency achieved by some test cases. Similarly to the *MatMult* case, the *Baseline* design is very simple: it consists of a loop that computes multiply-acccumulate operation per iteration using a DSP. Therefore, it can run at the higher-bound clock frequency of 330 MHz. Even the single-port test cases (*L2:16* and *L1:16 (1 p)*), despite being characterized by a large amount of cache ways (16), do not introduce any critical path limiting the frequency below the 100 %. Only with the multi-port test case (*L1:64 (15 p)*), which also involves an application loop

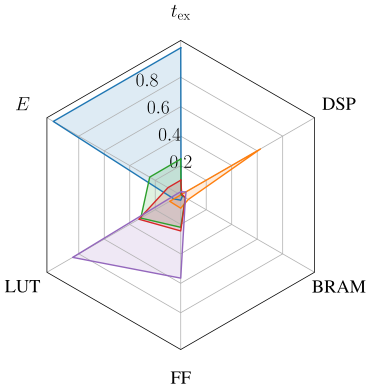


FIGURE 19. ***Conv2D*: PPA of some significant test cases.**

	Baseline	Manual	L2:16	L1:64 (5 ports)	L1:64 (15 ports)
t_{ex} (s)	32.69	0.03	8.69	1.60	0.71
P (W)	4.59	4.54	4.55	4.64	4.82
E (J)	150.0	0.1	39.5	7.4	3.4
LUT	3766	6082	21602	30880	59828
FF	4962	12670	30068	46458	79717
BRAM	3	13	8	8	8
DSP	1	225	1	5	15
Perf. gain	1.0	1089.7	3.8	20.4	46.0
Area cost	1.0	65.9	3.9	6.3	12.4

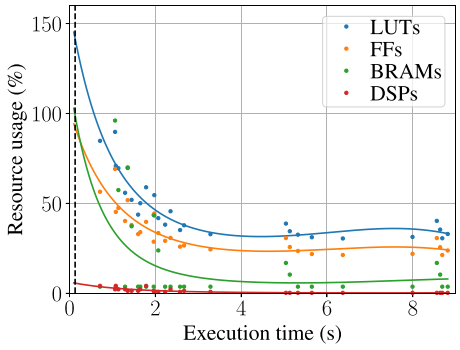


FIGURE 20. ***Conv2D*: regression of resource usage with respect to the execution time of the test cases with our caches.**

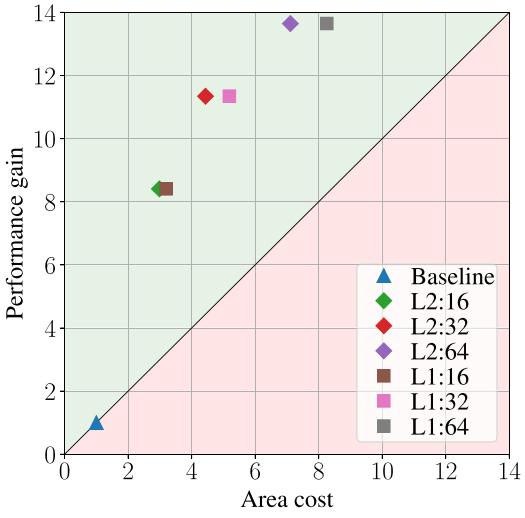


FIGURE 21. ***BitSort*: performance gain with respect to area cost. *Single-level Cache* is labelled as *L2:WORDS*, and *Multi-level* as *L1:WORDS*. The *WORDS* suffix stands for the number of words per line of the *a* cache.**

unrolling by a factor of 15, we face a frequency degradation of 39 %.

D. BITONIC SORTING

Bitonic sorting (*BitSort*) is a sorting algorithm, whose implementation is shown in Algorithm 5. From the memory access point of view, at each inner loop (`LOOP_I`) iteration:

Algorithm 5 Bitonic Sorting

Require: $a \in \mathbb{R}^N : N = 2^n$
Ensure: $a[i] \leq a[j], \forall i \geq j$
  procedure Sort(a)
    for ($b \leftarrow 1; b < n; b \leftarrow b + 1$) **do**
      for ($s \leftarrow b - 1; s \geq 0; s \leftarrow s - 1$) **do**
        LOOP_I: **for** ($i \leftarrow 0; i < N/2; i \leftarrow i + 1$) **do**
          $dir \leftarrow (i/2^{b-1})\&1$
          $dir \leftarrow dir \wedge 1$
          $step \leftarrow 2^s$
          $pos \leftarrow 2\,i - (i\&(s-1))$
          $a_0 \leftarrow a[pos]$
          $a_1 \leftarrow a[pos + step]$
          if ($a_0 > a_1 \neq dir$) **then**
            $tmp \leftarrow a_0$
            $a_0 \leftarrow a_1$
            $a_1 \leftarrow tmp$
          end if
          $a[pos] \leftarrow a_0$
          $a[pos + step] \leftarrow a_1$
        end for
      end for
    end for
  end procedure

(1) $a[pos]$ is read, (2) $a[pos + step]$ is read, (3) $a[pos]$ is written, and (4) $a[pos + step]$ is written. Therefore, the cache associated with the a array should be set-associative with at least two sets, so that the interleaved accesses to pos and $pos + step$ do not overwrite the related cache lines.

In the designs under test, the inner loop was pipelined, but due to the data dependencies on the a array the pipeline performance is limited. The pipeline of the *Baseline* test case (accessing a directly from DRAM) requires a very high $II = 142$ CCs because it must guarantee the dependency on the slow AXI interface. Our cache allows shortening the dependency distance and building a more performant pipeline, with an $II = 6$ CCs. All the tests use the same data type (32-bit

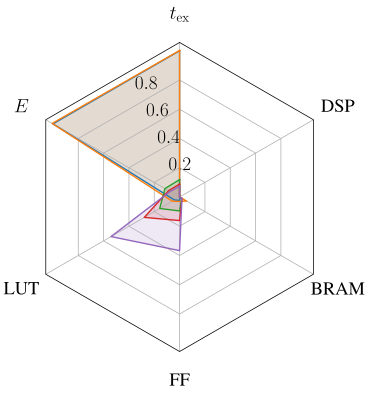


FIGURE 22. ***BitSort*: PPA of some significant test cases.**

	Baseline	RTLcache	L2:16	L2:32	L1:64
t_{ex} (s)	156.6	156.9	18.6	13.8	11.5
P (W)	4.52	4.54	4.39	4.45	4.64
E (J)	707.8	712.3	81.7	61.5	53.3
LUT	2896	4077	11089	19667	38206
FF	3270	4270	13454	22751	51758
BRAM	1	12	4	4	4
DSP	0	0	0	0	0
Perf. gain	1.0	1.0	8.4	11.3	13.6
Area cost	1.0	3.7	3.0	4.4	8.3

TABLE 6. ***BitSort*: tested cache configurations.**

Implementation	Words	L2 sets	L2 ways	L1 sets	L1 ways
Single-level (L2)	16	1	2	0	0
	32	1	2	0	0
	64	1	2	0	0
Multi-level (L1)	16	1	2	1	1
	32	1	2	1	1
	64	1	2	1	1

integers) and sizes ($N = 2^{20}$). Table 6 shows the tested cache configurations.

We were unable to implement a *Manual* design for an optimized reference, since the irregular access pattern, makes the on-chip data buffering challenging, especially considering that the array is accessed both in read and in write mode, introducing data dependencies. We believe that caching is the most convenient solution for optimizing this algorithm.

The *RTLcache* test case inserts the Xilinx System cache between the HLS kernel and the AXI interface. We set the total cache size to 32 kB (the minimum possible), with 2 ways, 64 words per line, and, by consequence, 128 sets.

The selected cache configurations achieve high L2 hit ratios, above 96 %. The L1 hit ratios are instead very low, from 8 % to 24 %, since our L1 caches use the write-through consistency policy.

Figure 21 plots the performance gain with respect to the area overhead of each test case with our cache. All the test cases provide significantly more performance gains than area cost. The L1 caches introduce a very limited performance advantage, because of their low hit ratio.

Figure 22 reports the full information on (a) the reference designs (*Baseline* and *RTLcache*), (b) the least resource-demanding cache configuration, i.e., *L2:16*, (c) the best cache configuration in terms of performance gain and area cost ratio, i.e., *L2:32*, and (d) the fastest cache configuration, i.e., *L1:64*.

The *RTLcache* is worse than the *Baseline* in all dimensions in the PPA space. This is because the cache module is inserted after HLS, therefore HLS optimizes the circuit as if all memory accesses were off-chip. In particular, the loop

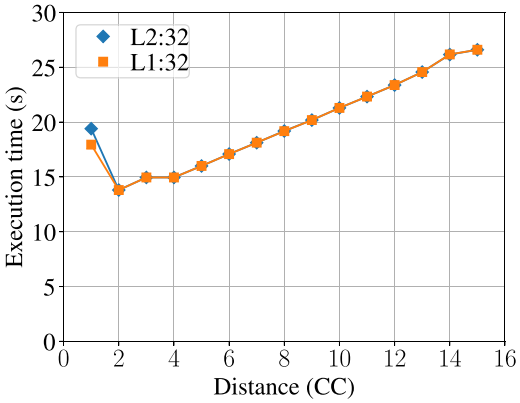


FIGURE 23. ***BitSort*: execution time with respect to L2 cache request-response distance.**

pipeline is still characterized by a very high II. This is another example showing that it is counterproductive to insert a RTL cache module a posteriori, after HLS. It is only introducing overhead, not only in terms of area, but also in terms of t_{ex} and power.

On the other hand, our cache improves the performance and the energy consumption by one order of magnitude compared to the *Baseline*. The *RTLcache*, despite having 128 sets instead of 1, consumes significantly less LUTs and FFs than our cache. It could be useful to combine the advantages of the source-level implementation with the resource efficiency of the RTL description to achieve the best performance at the lowest area cost. This could be achieved by exploiting the *Vitis HLS* capabilities to embed RTL code within HLS source code.

1) BITONIC SORTING REQUEST-RESPONSE DISTANCE

To evaluate the performance of the default L2 request-response distance for read-write (RW) cache configurations (2 CCs), we characterized the t_{ex} with respect to distance in a couple of test cases. As Fig. 23 shows, we chose the optimal value that balances the L2 cache pipeline exploitation (higher distance values better exploit it) and the algorithm loop II (the distance corresponds to the RAW dependency distance, and,

TABLE 7. ***BitSort*: maximum achievable clock frequency of some test cases. The relative maximum clock frequency is normalized over the maximum clock frequency of the AXI adapter.**

Test case	Maximum clock frequency (MHz)	Relative maximum clock frequency (%)
Baseline	330	100
L2:16	330	100
L1:32	300	91
L1:64	230	70

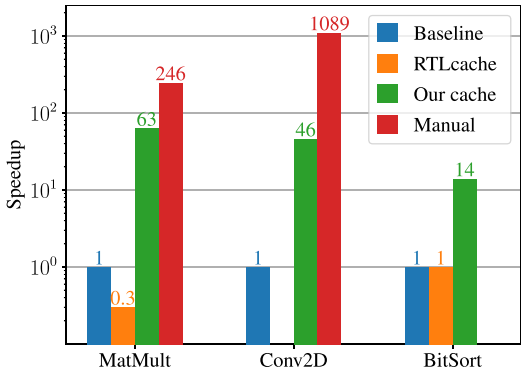


FIGURE 24. **Speedup of the tested benchmarks.**

by consequence, to the II). The data points of the multi-level configuration approximately overlap the single-level ones, because the L1 hit ratio is low. In a test case with high L1 hit ratio, the optimal distance value would probably be in 1 CC, since it would not need to exploit the L2 cache, and it could minimize the loop II.

2) BITONIC SORTING TIMING ANALYSIS

The maximum achieved clock frequencies for some test cases are shown in Table 7. Unlike the previous experiments, we encounter a slight maximum frequency degradation even with single-port cache configurations. This is due to the additional logic required for supporting both read and write operation within a single cache, differently from the read-only and write-only caches of *MatMult* and *Conv2D*.

VII. CONCLUSION

The experimental results, summarized in Fig. 24 show that our approach of semi-automatically generating a LCS-like architecture through dataflow caches is an effective solution for significantly improving performance and energy consumption, without requiring high design effort. Designers simply need to perform a design space exploration (DSE) of the cache configurations instead of extensively changing the algorithm for buffering data on-chip. Additionally, *for algorithms with irregular or data-dependent memory access patterns, caching would be the only way to actually improve memory access performance*.

To achieve performance comparable with the manually optimized designs of *MatMult* and *Conv2D*, our cache would require more resources than the ones provided by the small FPGA used in the tests. For *BitSort*, caching was the only feasible performance optimization we found, due to the irregular, but with good data locality, memory access pattern. Adding a RTL cache module post-HLS fails to provide any advantage, since the HLS-generated circuit is optimized for high-latency memory accesses, and cannot achieve any acceleration from an external cache.

It is worth noting that we collected the results from an embedded device, which provides a DDR4 memory. Modern datacenter-level devices are equipped with HBMs. HBMs, compared with DDR4 memories, are characterized by the availability of many more ports, thus dramatically increasing bandwidth, while paying a price in terms of access latency (roughly 2 times larger, as benchmarked by Wang et al. [4]). Thanks to these characteristics, a cache potentially provides even greater advantages than experienced with our setup, since caches are precisely designed for mitigating the performance penalties of high-latency memories. Moreover, irregular memory access patterns require word-sized accesses, since the HLS tool is unable to optimize the accesses through bursting and interface widening, underutilizing the high-bandwidth memory (HBM) ports bitwidth. On the other hand, caches always access the DRAM in lines, thus enabling the interface optimizations, resulting in better exploitation of the large interface bitwidth of HBM. We leave the evaluation of our caches on HBM-equipped HW, to quantitatively support these considerations, as future work.

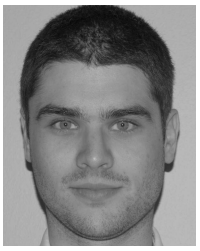
We plan to automate the design space exploration for optimal cache parameter selection, by extending one of the state-of-the-art cache parameter optimization methods [23] to support the configuration space of our cache architecture for some additional dimensions with respect to standard caches, such as the request-response distance, the number of ports, and the address bit mapping.

To further improve performance, we are considering to implement a prefetching mechanism to anticipate the memory requests by loading data in advance, before they are needed by the computation, thus fully emulating the LCS pattern.

REFERENCES

[1] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, no. 5, pp. 67–77, May 2011, doi: 10.1145/1941487.1941507.
[2] N. S. Kim, D. Chen, J. Xiong, and W.-M. W. Hwu, "Heterogeneous computing meets near-memory acceleration and high-level synthesis in the post-moore era," *IEEE Micro*, vol. 37, no. 4, pp. 10–18, Aug. 2017.
[3] J. S. Vetter, E. P. DeBenedictis, and T. M. Conte, "Architectures for the post-Moore era," *IEEE Micro*, vol. 37, no. 4, pp. 6–8, Aug. 2017.
[4] Z. Wang, H. Huang, J. Zhang, and G. Alonso, "Shuhai: Benchmarking high bandwidth memory on FPGAS," in *Proc. IEEE 28th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2020, pp. 111–119.
[5] Xilinx. *Vitis High-Level Synthesis User Guide*. Accessed: Dec. 2021. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_2/ug1399-vitis-hls.pdf
[6] J. Cong, P. Zhang, and Y. Zou, "Optimizing memory hierarchy allocation with loop transformations for high-level synthesis," in *Proc. 49th Annu. Design Autom. Conf. (DAC)*, San Francisco, CA, USA, 2012, pp. 1233–1238, doi: 10.1145/2228360.2228586.
[7] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, New York, NY, USA, 2013, pp. 29–38, doi: 10.1145/2435264.2435273.

[8] J. Choi, S. D. Brown, and J. H. Anderson, "From pthreads to multicore hardware systems in LegUp high-level synthesis for FPGAs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 10, pp. 2867–2880, Oct. 2017.
[9] J. de Fine Licht and T. Hoefler, "hlslib: Software engineering for hardware design," 2019, *arXiv:1910.04436*.
[10] L. Ma, L. Lavagno, M. T. Lazarescu, and A. Arif, "Acceleration by inline cache for memory-intensive algorithms on FPGA via high-level synthesis," *IEEE Access*, vol. 5, pp. 18953–18974, 2017.
[11] E. Matthews, N. C. Doyle, and L. Shannon, "Design space exploration of l1 data caches for FPGA-based multiprocessor systems," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, New York, NY, USA, Feb. 2015, pp. 156–159, doi: 10.1145/2684746.2689083.
[12] J. Choi, K. Nam, A. Canis, J. Anderson, S. Brown, and T. Czajkowski, "Impact of cache architecture and interface on performance and area of FPGA-based processor/parallel-accelerator systems," in *Proc. IEEE 20th Int. Symp. Field-Program. Custom Comput. Mach.*, Apr. 2012, pp. 17–24.
[13] G. Jo, H. Kim, J. Lee, and J. Lee, "SOFF: An OpenCL high-level synthesis framework for FPGAs," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, May 2020, pp. 295–308.
[14] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer, "Leap scratchpads: Automatic memory and cache management for reconfigurable logic," in *Proc. 19th ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, New York, NY, USA, 2011, pp. 25–28.
[15] F. Winterstein, K. Fleming, H.-J. Yang, J. Wickerson, and G. Constantinides, "Custom-sized caches in application-specific memory hierarchies," in *Proc. Int. Conf. Field Program. Technol. (FPT)*, Dec. 2015, pp. 144–151.
[16] Xilinx. *System Cache LogiCORE IP Product Guide (PG118)*. Accessed: Dec. 2021. [Online]. Available: https://docs.xilinx.com/r/en-U.S./pg118-system-cache
[17] Intel. *Intel High Level Synthesis Compiler Pro Edition Reference Manual*. Accessed: Dec. 2021. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683349/21-4/pro-edition-reference-manual.html
[18] Y. Chi, L. Guo, Y.-K. Choi, J. Wang, and J. Cong, "Extending high-level synthesis for task-parallel programs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2021, pp. 204–213.
[19] Xilinx. *Vivado Design Suite User Guide*. Accessed: Dec. 2021. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2021_2/ug973-vivado-release-notes-install-license.pdf
[20] Avnet. *Ultra96 Hardware User's Guide*. Accessed: Mar. 2018. [Online]. Available: https://www.avnet.com/opasdata/d120001/medias/docus/187/Ultra96-HW-User-Guide-rev-1-0-V0_9_preliminary.pdf
[21] Xilinx. (2021). *PYNQ: Python Productivity for Xilinx Platforms*. [Online]. Available: https://pynq.readthedocs.io/en/v2.7.0/
[22] J. Marjanovic. (Dec. 2021). *Exploring the PS-PL Axi Interfaces on ZYNQ Ultrascale+ Mpsoc*. [Online]. Available: https://j-marjanovic.io/exploring-the-ps-pl-axi-interfaces-on-zynq-ultrascale-mpsoc.html
[23] B. R. Upadhyay and T. S. B. Sudarshan, "Design space exploration of cache memory—A survey," in *Proc. Int. Conf. Elect., Electron., Optim. Techn. (ICEEOT)*, Mar. 2016, pp. 2294–2297.
[24] Intel. *Avalon Memory-Mapped Host Interfaces and Load-Store Units*. Accessed: Dec. 2021. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683349/22-2/memory-mapped-host-interfaces-and-load.html
[25] *Arria 10 EMIF Latency*. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683841/17-0/emif-latency-07619.html
[26] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 5, pp. 1014–1029, May 2021.
[27] Xilinx. (Dec. 2021). *Design and Analysis of Hardware Kernel Module for 2-D Video Convolution Filter*. [Online]. Available: https://xilinx.github.io/Vitis-Tutorials/2021-1/build/html/docs/Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial/lab2_conv_filter_kernel_design.html



GIOVANNI BRIGNONE (Graduate Student Member, IEEE) received the M.S. degree in computer engineering from the Politecnico di Torino, Italy, in 2021, where he is currently pursuing the Ph.D. degree with the Department of Electronics and Telecommunications under the supervision of Prof. L. Lavagno. His research interests include high-level synthesis, digital hardware design, and HW/SW co-design.

M. USMAN JAMAL (Graduate Student Member, IEEE) received the M.S. degree from the Politecnico di Torino, Italy, in 2018, where he is currently pursuing the Ph.D. degree with the Department of Electronics and Telecommunications under the supervision of Prof. L. Lavagno. His research interests include high-level synthesis, low-power high-performance computing, and machine learning for electronic design automation.

MIHAI T. LAZARESCU (Senior Member, IEEE) received the Ph.D. degree in electronics and communications engineering from the Politecnico di Torino, Italy, in 1998. He is currently an Assistant Professor with the Politecnico di Torino. He was a Senior Engineer at Cadence Design Systems and founded several startups. He has coauthored over 60 scientific publications, four books, and international patents. His research interests include design tools for WSN/IoT platforms, ubiquitous environmental sensing, efficient neural networks, indoor human localization, edge and leaf IoT data processing, and high-level HW/SW co-design and synthesis.

LUCIANO LAVAGNO (Senior Member, IEEE) received the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 1992. He was an Architect with the POLIS HW/SW co-design tool. From 2003 to 2014, he was an Architect with the Cadence Cto-Silicon high-level synthesis tool. Since 1993, he has been a Professor with the Politecnico di Torino, Italy. He has coauthored four books and over 200 scientific papers. His research interests include synthesis of asynchronous circuits, HW/SW co-design, high-level synthesis, and design tools for wireless sensor networks.

Open Access funding provided by 'Politecnico di Torino' within the CRUI CARE Agreement