IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 8 February 2023, accepted 23 March 2023, date of publication 3 April 2023, date of current version 6 April 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3264265

APPLIED RESEARCH

IXIAM: ISA EXtension for Integrated Accelerator Management

BIAGIO PECCERILLO[1], **ELHAM CHESHMIKHANI**[2], **MIRCO MANNINO**[1], **ANDREA MONDELLI**[3], **AND SANDRO BARTOLINI**[1]

[1]Department of Information Engineering and Mathematics, University of Siena, 53100 Siena, Italy
[2]Department of Computer Engineering, Amirkabir University of Technology (Tehran Polytechnic), Tehran 15875-4413, Iran
[3]Huawei Technologies Research and Development (U.K.) Ltd., CB4 0WG Cambridge, U.K.

Corresponding author: Biagio Peccerillo (peccerillo@diism.unisi.it)

This work was supported by Huawei Technologies Research and Development (U.K.) Ltd.

ABSTRACT During the last few years, hardware accelerators have been gaining popularity thanks to their ability to achieve higher performance and efficiency than classic general-purpose solutions. They are fundamentally shaping the current generations of Systems-on-Chip (SoCs), which are becoming increasingly heterogeneous. However, despite their widespread use, a standard, general solution to manage them while providing speed and consistency has not yet been found. Common methodologies rely on OS mediation and a mix of user-space and kernel-space drivers, which can be inefficient, especially for fine-grained tasks. This paper addresses these sources of inefficiencies by proposing an *ISA eXtension for Integrated Accelerator Management* (IXIAM), a cost-effective HW-SW framework to control a wide variety of accelerators in a standard way, and directly from the cores. The proposed instructions include reservation, work offloading, data transfer, and synchronization. They can be wrapped in a high-level software API or even integrated into a compiler. IXIAM features also a user-space interrupt mechanism to signal events directly to the user process. We implement it as a RISC-V extension in the gem5 simulator and demonstrate detailed support for complex accelerators, as well as the ability to specify sequences of memory transfers and computations directly from the ISA and with significantly lower overhead than driver-based schemes. IXIAM provides a performance advantage that is more evident for small and medium workloads, reaching around 90× in the best case. This way, we enlarge the set of workloads that would benefit from hardware acceleration.

INDEX TERMS Hardware accelerators, domain-specific architectures, parallel architectures, heterogeneous systems, RISC-V.

I. INTRODUCTION

Today hardware accelerators are employed in a variety of contexts that span from wearable and embedded computing to supercomputers and data-centers. Hwu and Patel [1] define a hardware accelerator as "a separate architectural substructure [...] that is architected using a different set of objectives than the base processor, where these objectives are derived from the needs of a particular class of applications".[1] They are now regarded as the primary driving force of computer architecture [3], thanks to their ability to improve non-functional metrics such as throughput and energy efficiency. With respect to general-purpose resources, they can take advantage of efficient forms of parallelism, local/optimized memories, ad-hoc

The associate editor coordinating the review of this manuscript and approving it for publication was Thomas Canhao Xu.

[1] Some works refer to them as *loosely-coupled accelerators*, to distinguish them from *tightly-coupled accelerators* [2]. According to the definition, tightly-coupled accelerators are rather *specialized functional units* integrated into the processor.

datapaths, reduced fetch and decode overheads, and support for special data-types [3], [4].

In the last years, accelerators have greatly influenced the design of computing systems, with a profound impact on Systems-on-Chip (SoCs) in particular [5]. These have become increasingly heterogeneous, with various accelerators integrated with the central general-purpose cores on the same chip. Core-accelerator communication is achieved through a dedicated Network-on-Chip (NoC) or a bus with a standard interface such as ARM's Advanced Microcontroller Bus Architecture (AMBA) [6]. Usually, accelerators access the system's physical memory by connecting to a shared Last-Level Cache (LLC) [7], [8] or the DRAM [9], [10], relying on DMA techniques not to burden the CPU.

From a software perspective, functional units are usually targeted at the Instruction Set Architecture (ISA) level, but on-chip accelerators commonly require a mix of user-space and kernel-space drivers [5]. Programmers interact with a high-level accelerator-specific Application Programming Interface (API) that transparently invokes functions provided by the underlying driver layer. Drivers, in turn, may rely on system calls that ask for the Operating System (OS) mediation in a variety of tasks: management of accelerators' resources, scheduling of simultaneous accesses from different processes, data consistency maintenance across different privilege levels, virtual-to-physical address translation – whether as the main actor or as an accelerator assistant [11], [12]. The presence of many layers and different privilege levels to achieve CPU-accelerator communication can be a source of inefficiencies and limit the set of tasks eligible for acceleration [2], [13], [14]. In fact, the latency associated with CPU-accelerator interaction is significantly longer than the conventional core operations. This communication overhead can be amortized only when the accelerated task has a coarse-grain nature, as fine-grain ones would be dominated by communication latency and thus their acceleration would not be convenient.

Albeit the driver-oriented solution is the most common to achieve CPU-accelerator interaction, it is not the only one [5]. Some hardware accelerator proposals rely on special instructions added to the ISA of the processor [15], [16], [17], [18], [19], [20]. These ISA extensions, however, are always accelerator-specific, and as a result, they cannot be scaled. By adding new accelerators, the ISA would have to be further extended, leading to ISA bloating that would be particularly harmful to fixed-size instruction sets and negatively impact both the chip area and energy consumption of the decoding phase.

Overall, both accelerator-specific ISA extensions and user-space/kernel-space drivers have drawbacks – summarizing, lack of generality and low scalability in the first case, and high latency in the second. In this paper, we present an *ISA eXtension for Integrated Accelerator Management* (IXIAM), a low-latency, high-performance, general solution to orchestrate on-chip accelerators from CPU cores. It is based on a general ISA extension that supports a wide variety of accelerators and a limited hardware infrastructure. The proposed instructions facilitate managing accelerator reservation, work offloading, synchronization, and data transfers with fine-grained control of accelerator local memories. It is paired with a user-space interrupt mechanism to signal events such as execution completion and error occurrence from the accelerators directly to the user process. The supported hardware infrastructure implements communication and some tasks normally delegated to the OS, such as reservation. As a result, the entire proposal can be, in principle, implemented in user space while still guaranteeing security and isolation.

We extend and enrich our previous proposal presented in [21]. The main differences with the previous proposal are: a) the possibility to have fine-grained control of accelerator local memories, b) the user-space interrupt mechanism, c) the support of a far broader class of accelerators, since we do not require them to work according to a fixed finite state machine. Moreover, in this work, we discuss further aspects such as memory coherency and data consistency.

We implement our framework in the gem5 simulator [22] by extending the RISC-V ISA and adding all the necessary modules to simulate the proposed hardware infrastructure. We evaluate it performance-wise in comparison with conventional driver-based interfacing, on five different simulated accelerators. We show that our proposal dramatically improves performance for small workloads (around 90×) and medium workloads (more than 4×), surpassing in most cases even an optimized CPU-only implementation – thus, enlarging the set of workloads that benefit from hardware acceleration.

The main contributions of this paper can be summarized as follows:

- We propose IXIAM, an HW-SW framework articulated in an ISA extension and hardware infrastructure to manage on-chip accelerators, taking care of reservation, work offloading, data transfers, fine-grained control of accelerator local memories, and synchronization;
- We propose a user-space interrupt mechanism to signal execution completion and errors directly to the user process, with no OS mediation;
- We implement the whole proposal in the gem5 simulator by extending RISC-V ISA and providing the necessary hardware modules;
- We evaluate our proposal in the case of five widely-used accelerators and compare it against a conventional driver-based solution, showing that we improve performance in all the cases, but mostly for small and medium workloads;
- We show that our proposal enriches the set of workloads that are eligible for hardware acceleration, as we achieve better performance than a non-accelerated implementation in cases where a driver-based one would not be convenient.

The remainder of the paper is organized as follows. In Section II, we present the proposed framework in detail, and in Section III we analyze it critically and discuss some limitations and criticalities. Then, in Section IV we evaluate the proposal by simulating various accelerators in the gem5 simulator. Section V presents some related work. Finally, we conclude in Section VI.

II. PROPOSED FRAMEWORK

In this section, we present the details of IXIAM, our proposed general framework to implement instruction-based communication between accelerators and processors in SoCs. First, we present the reference architectural context. Then, we discuss the hardware infrastructure that must be added to it as part of our proposal. Finally, we present the additional instructions that must be added to the processor ISA.

A. ARCHITECTURAL REFERENCE

In this work, we assume a heterogeneous SoC as our architectural reference. The SoC includes a general-purpose processor with m cores and n accelerators.

With reference to the taxonomy presented in [5], we assume these accelerators are integrated on-chip, non-programmable, but may be reconfigurable and possibly endowed with heterogeneous memory resources. They accelerate tasks in any application domain. From an architecture point of view, they can have a register file (e.g., 32-bit, 64-bit registers), any level of cache memories (also *no caches*), and one or more addressable local memories. These can be used to store input data to be consumed by the accelerated task, or intermediate and output data produced during its execution. We do not impose any constraint on the nature of their computing engines. They have access to the physical memory of the system through a level of the memory hierarchy. Without loss of generality, we assume that they are connected to the LLC, which is located on the chip and shared between cores and accelerators – which is a common choice adopted by SoC designers [7], [8], [19], [23], [24]. They are equipped with a DMA controller to load/store data from/to the LLC without CPU intervention.

Cores and accelerators are connected through an interconnect, which can be a simple bus or a ring-/mesh-based NoC. We take as a reference a ring-based NoC, which is general and is used in different kinds of conventional accelerator-processor interconnections [25], [26]. The evaluation of different kinds of NoCs is beyond the scope of this paper and is left as future work. The network allows any core to send messages/packets to any accelerator and vice versa. We refer to these messages as Accelerator-Core Messages (ACMs).

B. IXIAM HARDWARE INFRASTRUCTURE

IXIAM requires some hardware additions to the SoC described above as a necessary infrastructure to allow the proposed instructions to control on-chip accelerators. We identify these additions as the *IXIAM Hardware Infrastructure*.

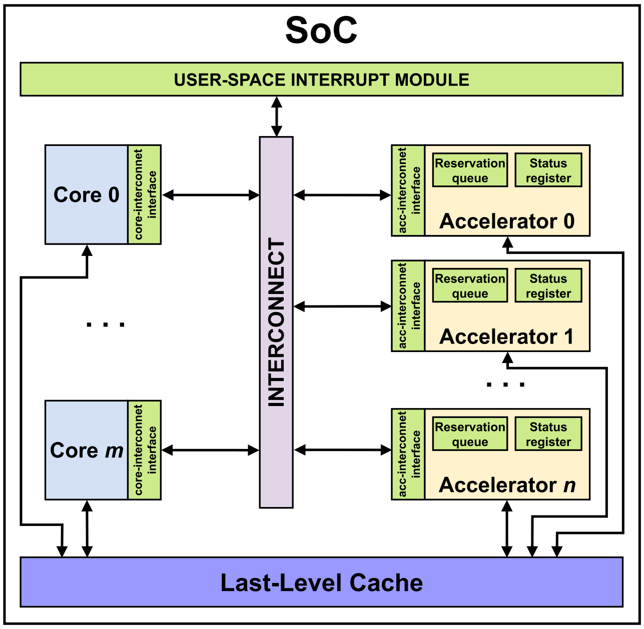


FIGURE 1. Reference SoC with IXIAM Hardware Infrastructure components highlighted in green.

Figure 1 shows the reference SoC architecture, with the additions highlighted in green.

Each accelerator must be identified by a unique id (in the following, *accId*). Without loss of generality, we assume an 8-bit id, to support up to 256 different accelerators on the same SoC. This id must be known to the NoC in order to deliver ACMs coming from the cores to the right target accelerator.

Accelerators must be reservable to processes running on the cores. To achieve this, each of them must be equipped with a FIFO *reservation queue* where reservation requests from different processes must be enqueued up until the queue is full – then, the requests are dropped. An accelerator is considered *reserved* to the process whose id (*procId* in the following) is at the front of the queue. We denote this process as the *owning process*. We adopt a process-based reservation scheme rather than a core-based or thread-based one for two main reasons:

- Processes are the *natural* entities to take advantage of task acceleration by offloading portions of work to dedicated accelerators;
- In a standard multi-core environment, a process can be made of various threads of execution, and the OS is free to schedule them on any core at any time according to a policy.

Some authors prefer a thread-based reservation scheme [2]. We could also adopt this solution with no substantial modifications to our proposal. However, reserving an accelerator to a single thread of execution would force the programmer to concentrate all the accelerator-related code in the same thread. With a process-based reservation scheme, conversely, the code can be structured more freely: for instance, one thread could be used to ask for reservation in a timer-like

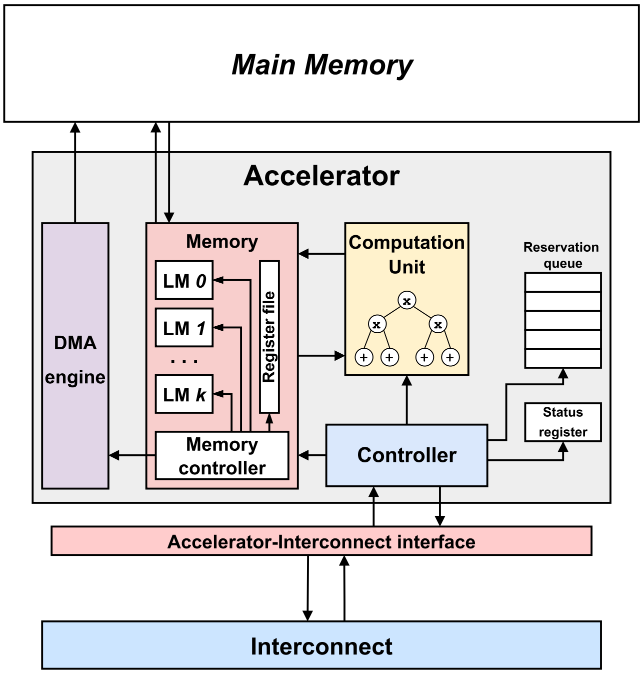


FIGURE 2. Sample internal structure of an accelerator.

fashion, signaling another thread to transfer data/launch accelerated tasks if reservation succeeds or execute non-accelerated fallback code otherwise, etc.

We consider an accelerator *busy* when it is executing an accelerated task. Each accelerator must be equipped with a *status register* to summarize the accelerator's status. Its values are interpreted as pertaining to three possible statuses:

- "0" if the accelerator is not busy and no errors occurred;
- "1" if the accelerator is busy and no errors occurred;
- error code if an error occurred.

The owning process can read the status register through a convenient instruction that will be discussed in Subsection II-D. If the status register holds an error state, it is cleared upon read. Figure 2 shows a block diagram of an accelerator with internal details.

In order to support a user-space interrupt mechanism, the SoC must be augmented with an ad-hoc module to let processes register their routines and hold information about occurred user-space interrupts, which include, for each interrupt: the process that should handle it, the accelerator where it originated, and some parameters to detail its reason. It will be discussed in-depth in Subsection II-E.

The last components of the IXIAM Hardware Infrastructure are the modules that allow cores and accelerators to deliver messages through the NoC. We denote them as the *core-NoC interface* and the *accelerator-NoC interface*. Both have sender and receiver complementary roles, as they take command from their attached module (core in one case, accelerator in the other) and send/receive messages through the NoC.

Summarizing, the IXIAM Hardware Interface includes:

- a unique identifier `accId` for each accelerator;
- a *reservation queue* for each accelerator;
- a *status register* for each accelerator;
- a user-space interrupt module on the SoC;
- a core-NoC interface for each core;
- an accelerator-NoC interface for each accelerator.

C. ACCELERATOR-CORE MESSAGES

Every accelerator-oriented instruction causes an ACM sending from the core where it is executed to the target accelerator. We call it a *request ACM*, as opposed to the *response ACM* that may be sent back from the accelerator to the core. The majority of instructions do not need a response ACM. In those cases, the instructions are committed immediately right after a request ACM has been sent. Conversely, some instructions need a response ACM to carry a return value that will be written in an output register, typically specified as instruction operand. Thus, they will be committed when the response ACM is received.

Based on this, we define *synchronous* and *asynchronous* instructions as follows:

- Asynchronous instructions are committed immediately after sending a request ACM;
- Synchronous instructions are committed when the response ACM is received on the core.

Although synchronous instructions need a response ACM reception to be committed, they do not need to stall the pipeline till such event. For instance, in a core with out-of-order execution, no special care is needed apart from the ordinary Read after Write (RAW) dependency management among instructions: since these instructions have an output register, successive dependent instructions (i.e., having such register as input) are blocked until the synchronous instruction is committed.

Figure 3 shows the ACMs associated to the proposed instructions, that will be discussed in-depth in the next subsection. Without loss of generality, we adopt a sample format organized in indexed 64-bit packets.

Every ACM contains the parameters needed by the NoC and the accelerator to execute the intended operations. The first byte, `Inst`, identifies the executed instruction, and the others are usually provided as instruction operands. `coreId` and `procId` make an exception.

`coreId` is present only in CHECK, ISBUSY, and TRS, which are the only synchronous instructions in our proposal. It identifies the id of the core where the instruction has been executed. This is necessary because, this way, the accelerator can enrich the response ACM with the same `coreId` it received in the request, so the NoC can deliver it to the right core, the output register can be written and the instruction can be committed.

`procId` is necessary to identify the process that want to communicate with the accelerator. `procId` is used to reserve the accelerator to the calling process, and to check the legitimacy of subsequent requests, e.g., data transfers and operation execution, that should be granted only to the

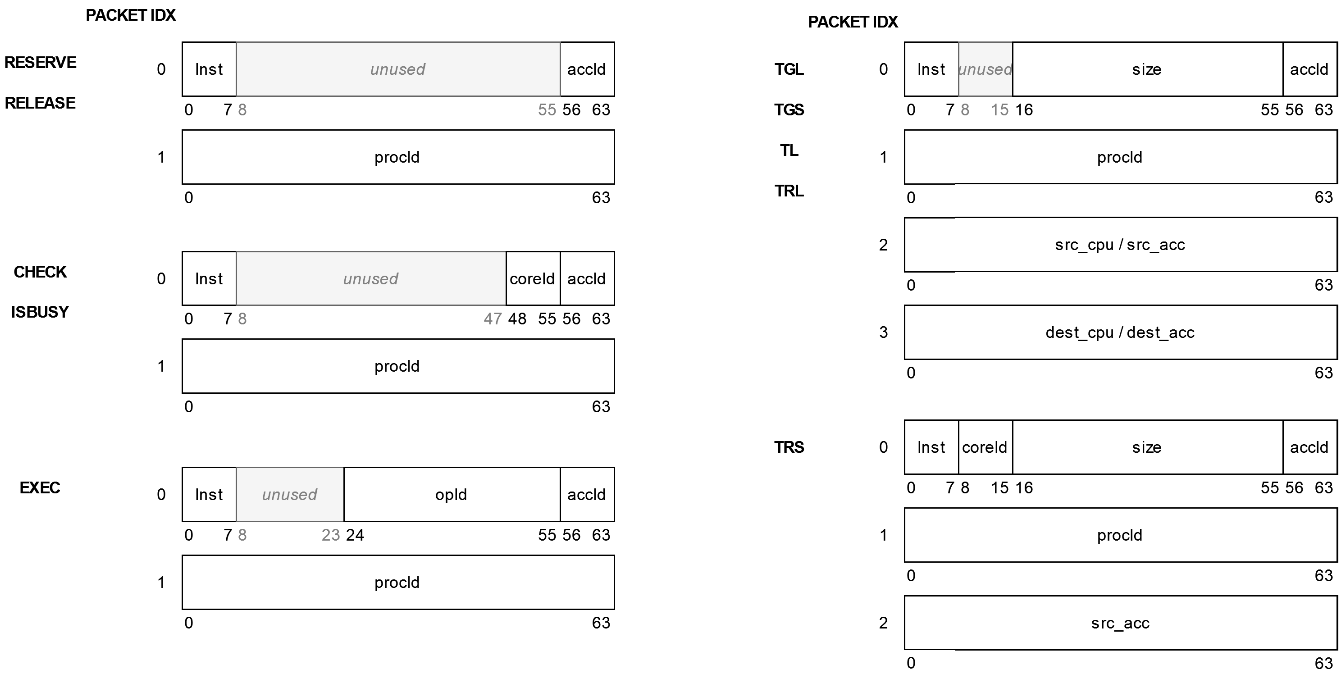


FIGURE 3. Possible request ACMs generated by different instructions, organized in 64-bit packets.

owning process. Since this value determines uniquely what can be done (or not) with an accelerator, it cannot be a user-provided parameter for security reasons. A malevolent process could easily disguise itself as another one by simply using another `procId` value. For this reason, this value is read automatically by the IXIAM Hardware Infrastructure from the Control and Status Registers (CSRs). An example of process identifier that would work in our proposal could be the pointer to the page table, which uniquely identifies each process.

The target accelerator is always determined with a user-provided `accId` parameter, which is needed by all the instructions and is present in all the related request ACMs. In this example, 8 bits are dedicated to it, so 256 different accelerators are supported. The NoC reads it to deliver the ACM to the right accelerator, and generates an exception like "Illegal Instruction" if this does not correspond to any accelerator in the SoC.

The remaining parameters will be discussed in the next subsection as operands of the proposed instructions.

D. ISA EXTENSION

In the following, we present our accelerator-oriented ISA extension. In principle, it could be implemented as an extension to any existing ISA, as it is independent on existing instructions. In this work, we design it as a RISC-V extension. More precisely, we extend the RISC-V RV64I instruction set with Zicsr instructions [27], and implement our instructions using the R-type format. All the operands are *register operands*: they indicate the register in which the needed value is stored. Our instructions have the minimum privilege level, so they can be executed by user-space code. Table 1 summarizes them.

TABLE 1. Details of the proposed instructions. CPU-reg operands are read from a conventional CPU register (e.g., *t0*). CSR operands are read from Control and Status Registers.

Instruction	Input	Output	CPU-reg	CSR
RESERVE	accId			procId
CHECK	accId	ret		procId
TGL	size, src_cpu, dest_acc		accId	procId
TGS	size, src_acc, dest_cpu		accId	procId
TL	size, src_acc, dest_acc		accId	procId
TRL	size, src_reg_cpu, dest_acc		accId	procId
TRS	size, src_acc	ret	accId	procId
EXEC	accId, opId			procId
ISBUSY	accId	ret		procId
RELEASE	accId			procId

1) RESERVE INSTRUCTION

Conventionally, accelerator-processor communication in SoCs needs to start with some handshaking protocol between the two entities involved. For instance, the processor could ask if the requested accelerator is ready to work or not. If so, it could *reserve* the accelerator and offload work to it until it is done. Then, it can *release* it.

In this work, we adopt this scheme on a per-process basis, allowing the process to reserve the accelerator for itself and offload work to it from any thread of execution, independently of the core where it is scheduled. We propose the RESERVE instruction to achieve this, with format `RESERVE <accId>`.

RESERVE execution generates an ACM containing `accId`, which is the only operand of the instruction, and `procId`, which is provided by the IXIAM Hardware Infrastructure as specified. The instruction is asynchronous: it is committed immediately, and the ACM is sent through the NoC to the target accelerator identified by `accId`.

As discussed in Subsection II-B, if the accelerator is free, the received `procId` is enqueued and the calling process becomes the owning process. Conversely, if the queue is not empty neither full and `procId` has not been previously enqueued, it is enqueued to wait for its turn to own the accelerator. Finally, if the queue is full the reservation request is just dropped. Listing 1 shows the RESERVE algorithm.

This way, the accelerator reservation is compatible with a multi-process environment, which is the most common use case nowadays. However, the choice of having a reservation queue in hardware permits implementing the whole logic in a secure manner without asking for OS intervention, which is beneficial from a latency point of view.

Algorithm 1 Micro-Operations of RESERVE Instruction

```
RESERVE(accId, procId);
acc = accelerators[accId];
if acc.reservationQueue.empty() then
    acc.reservationQueue.enqueue(procId);
    acc.status = RESERVED;
else if not acc.reservationqueue.full() then
    acc.reservationQueue.enqueue(procId);
else
    /* NOP */
end
```

2) CHECK INSTRUCTION

As discussed, RESERVE instruction is asynchronous and no feedback is sent back to the user. In order to query the accelerator about the reservation outcome, we provide the CHECK instruction, with format `CHECK <accId> <ret>`.

Same as RESERVE, also CHECK causes an ACM sending where `accId` is used to select the target accelerator and `procId` is added by the IXIAM Hardware Infrastructure to identify the calling process. Moreover, `coreId` is added to the request to identify the core and send back the response ACM.

The accelerator that receives the CHECK ACM accesses the reservation queue and checks whether it contains `procId` or not. Depending on this check, the outcome can be threefold: a) it can be 0 (`MISSING`), if `procId` is not contained in the queue; b) it can be 1 (`ENQUEUED`), if `procId` is in the queue but it is not at its head; and c) it can be 2 (`RESERVED`), if `procId` is found at the head of the queue.

The outcome is written into a response ACM together with `coreId` and sent through the NoC. When it is received by the core identified by `coreId`, it is written in the `ret` register so it can be read by the program logic.

In Subsection II-E, we discuss user-space interrupts and the possibility to employ that mechanism to avoid synchronous instructions altogether. Listing 2 summarizes the CHECK algorithm.

Algorithm 2 Micro-Operations of CHECK Instruction

```
CHECK(accId, procId, ret);
acc = accelerators[accId];
if not acc.reservationQueue.contains(procId) then
    ret = 0;                              /* MISSING */
else if acc.reservationqueue.head() $\neq$ procId then
    ret = 1;                              /* ENQUEUED */
else
    ret = 2;                              /* RESERVED */
end
```

3) TRANSFER INSTRUCTIONS

Accelerators can have various memory resources: a register file, cache memories, and one or more local memories. These can serve different purposes and be optimized for different tasks. NVIDIA GPUs' local memories are an example of such variability: *shared memory* is optimized for coalesced accesses from neighbor CUDA threads, *constant memory* is optimized to broadcast the same value to numerous threads, and *texture memory* is optimized for spatially local accesses [28], [29].

Local memories can be further grouped into *non-addressable* and *addressable* memories. While the former are managed by the accelerator transparently, e.g., to store intermediate results, the latter can be explicitly managed by the programmer. We propose five instructions to load data from/store data into *addressable* local memories or registers in a register file.

The operands of these instructions can refer to local memory locations or accelerator register files. To implement them, we adopt the *IXIAM Location Format*, exemplified in Figure 4. Bit 40 specifies whether a location indicates an address in local memory (0) or a register 1. Bits 61-63 indicate the local memory number in the accelerator: since there can be multiple local memories in each accelerator, we dedicate 3 bits to specify the one referred by the memory location – thus, we support 8 possible local memories for each accelerator. If bit 40 is set, this field is meaningless and is just ignored. Finally, bits 0-39 indicate the address in local memory, if bit 40 is 0, or the register number, if bit 40 is 1.

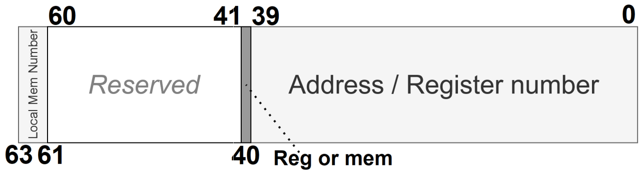


FIGURE 4. IXIAM Location Format. Bits 0-39 are dedicated to the *location*, which can be a memory address or a register number, depending on bit 40. Bits 61-63 indicate the local memory number in the accelerator.

Thus, we admit local memories with up to 2^{40} bytes (1 TiB) or, equivalently, register files with 2^{40} registers.

All the transfer instructions have four operands. Since the RISC-V ISA is limited to 3 operands, we choose to pass the fourth in a *convenience* register - e.g., `t0`.

All the transfer instructions generate a request ACM that contains a `procId` value read by the IXIAM Hardware Infrastructure to identify the calling process. Its role, in this case, is to let the target accelerator establish whether the transfer operation can be performed or not: the operation is performed only if the accelerator is reserved to the process identified by `procId`, and the request is ignored otherwise.

a: TGL, TGS

Transfer Global Load (TGL) is used to load data from the main memory into a location in the accelerator, and Transfer Global Store (TGS) stores data from a location in the accelerator into the main memory. Thus, *load* and *store* are expressed from the accelerator point of view.

Their formats are: `TGL <size, src_cpu, dest_acc, accId>` and `TGS <size, src_acc, dest_cpu, accId>`. `accId`, the fourth operand, is read by a convenience CPU register.

`size` indicates the data size in bytes to load/store. `dest_acc` in TGL and `src_acc` in TGS are locations on the accelerator (local memory address or register) expressed in the IXIAM Location Format described above. Finally, `src_cpu` in TGL and `dest_cpu` in TGS are pointers to locations in the main memory.

In this proposal, we suppose the virtual-to-physical address translation is performed CPU-side. The programmer specifies a virtual address in `src_cpu` or `dest_cpu` operands, which is translated into a physical address `phys_addr`. This is put in the request ACM that is sent to the accelerator, so it can load/store data from/to `phys_addr` with DMA techniques.

It is possible that the address range involved (i.e., [`phys_addr`, `phys_addr` + `size`)) spans more than one physical memory page. If that is the case, a single TGL/TGS instruction generates multiple ACMs, each with a physical address and a partial size that refers to a piece of the buffer located in the same memory page.

At this point, a page fault can occur on the accelerator if the accessed physical page has been swapped out. In this case, the accelerator can send an interrupt to the OS that would be served by loading the page and letting the accelerator retry the memory transfer. In general, pinning the memory involved in transfers is sufficient to avoid page faults.

b: TL

The Transfer Local (TL) instruction is used to move data between different locations on the accelerator. Its format is `TL <size, src_acc, dest_acc, accId>`, with both `src_acc` and `dest_acc` expressed in the IXIAM Location Format. What said about `size` and `accId` operands in TGL/TGS description applies also here. In this case, no memory translation is involved, since both locations are on the accelerator.

c: TRL, TRS

Transfer Register-wise Load (TRL) and Transfer Register-wise Store (TRS) are two convenient instructions to load/store data into/from the accelerator without accessing the main memory. The source in TRL and the destination in TRS are CPU registers. TRL reads a value from a CPU register and loads it into an accelerator location, while TRS reads a value from an accelerator location and stores it into a CPU register. Since TRS has an output register, it is a synchronous instruction – as such, it is committed when a response ACM is received. Thus, `coreId` is added to its request ACM with the usual role.

The instruction formats are `TRL <size, src_reg_cpu, dest_acc, accId>` and `TRS <size, src_acc, accId> <ret>`. `dest_acc` and `src_acc` are locations expressed in the IXIAM Location Format. Also in this case, `accId` is read from a convenience register in both TRL and TRS.

4) EXEC INSTRUCTION

The EXEC instruction permits executing an operation on the accelerator. It is intended to trigger the actual computation on the accelerator, usually manipulating data previously stored in registers and local memories.

The EXEC format is `EXEC <accId, opId>`. As usual, `accId` indicates the target accelerator. Since an accelerator can be capable of performing various operations, we dedicate the `opId` parameter to specify the operation to perform. As shown in the EXEC request ACM in Figure 3, 32 bits are dedicated to this value – so 2^{32} possible operations can be performed. This does not limit the number of possible operations: real accelerators can take advantage of registers to let programmers specify operation parameters (e.g., start address in local memory, size, operation *mode*, etc.) and use `opId` as a *coarse-grained* operation identifier. In principle, all the parameters needed could be written in registers and `opId` remain unused. We will discuss some possibilities in Section IV.

The EXEC instruction causes a request ACM to be sent with `procId` parameter with the usual meaning. Same as the transfer instructions, it is used by the accelerator to check if the request is legitimate or not. If it is, the accelerator sets the status register to "busy" 1 and the intended operation can start execution. Since the operation can last for several clock cycles, EXEC has been designed as an asynchronous instruction: it is committed immediately, and the process can continue executing different instructions without waiting for operation completion. In principle, its thread of execution could even be de-scheduled by the OS while the accelerator completes its task.

When the operation completes, the accelerator updates the status register with "non-busy" (0) or an error code if the execution was not successful.

5) ISBUSY INSTRUCTION

The ISBUSY instruction is used to query the status of the accelerator, which is stored in the status register. Its format is: `ISBUSY <accId> <ret>`.

As usual, `accId` denotes the target accelerator. `ret` denotes the register where the return value, which is read from a response ACM, is written. As explained in Subsection II-B, the possible values are 0 if the accelerator is not busy, 1 if it is busy, or an error code. The error code informs of the failure of a previous transfer or EXEC instruction, which are the only ones that can fail accelerator-side.[2] If that is the case, the status register is cleared and set to 0.

Since ISBUSY is a synchronous instruction, its request ACM contains `coreId` as well as `procId`. In principle, the accelerator status could be communicated also to processes that do not own it. However, revealing this information could constitute a security risk, as a malevolent process could gather information about the accelerator usage of another process. For this reason, the `procId` value is checked and, if it does not correspond to the owning process, an error is written in the `ret` register or a default value (whether 0 or 1) independently of the actual accelerator status.

6) RELEASE INSTRUCTION

Finally, the RELEASE instruction is intended to release the accelerator when the owning process finishes using it. Its format is: `RELEASE <accId>`, with `accId` denoting the target accelerator as usual.

The instruction execution causes a request ACM sending containing the `procId` of the calling process. When the instruction is received, it is eliminated from the reservation queue of the accelerator. If the calling process' `procId` occupies the head of the queue, the accelerator is assigned to the next `procId` in the queue or is set *idle* if the queue is empty.

E. IXIAM USER-SPACE INTERRUPT MECHANISM

In this subsection, we discuss our proposal for a user-space interrupt mechanism. Before presenting its details, we briefly examine instruction synchronicity.

1) INSTRUCTION SYNCHRONICITY

Among the proposed instructions, summarized in Table 1, only CHECK, TRS, and ISBUSY are synchronous. These need a return value from the accelerator, which they read from a response ACM and write it into an output register in the calling process.

For CHECK and ISBUSY instructions, the accelerator should immediately manage the request and send a value independently of its activity: even if it is busy executing a long operation, CHECK and ISBUSY requests should not be enqueued and should be managed immediately. This way, their latency would be known upfront and would amount to roughly the NoC round-trip time.

Conversely, the TRS case may be more critical, as the accelerator could be occupied in a long transfer operation that may involve the target local memory, and thus it could be forced to enqueue the transfer demanded by the TRS request and perform it later. So, the latency associated to a TRS instruction cannot be known in advance, and the core executing it could remain stalled for many clock cycles in the presence of RAW dependencies.

Another instruction that, in principle, could be designed as synchronous is EXEC, with the calling process waiting for the execution completion and getting a return value that informs it of the execution outcome. However, the execution of an operation on the accelerator can take up to thousands or even millions clock cycles, depending on the accelerator, and the possibility that a RAW dependency could stall the core for so much time makes this solution unfeasible: it could cause a huge performance degradation and would even have an impact on the whole system, since the OS could not dispose of the stalled core – e.g., to schedule other processes' threads of execution. Hence, the core should be able to continue its normal operation after delivering the EXEC instruction to the accelerator and should work *in parallel* with accelerator computations. For this reason, we keep EXEC asynchronous and rely on the ISBUSY synchronous instruction, that has a predictable and short latency, to allow the programmer to query the execution state. The intended use of ISBUSY is thus as part of a *polling* mechanism: after executing an EXEC, the process keeps calling ISBUSY in loop until it returns 0 or an error code, sleeping for a custom time interval or doing other work in-between loop iterations. In both cases, the process keeps a core busy-waiting.

2) IXIAM INTERRUPT MODULE

In all the aforementioned scenarios, an interrupt-based solution could prove beneficial. Synchronous instructions could be made asynchronous by eliminating the output registers and committing the instructions immediately upon sending their associated request ACM; the return value could be retrieved in the management of an interrupt originated on the accelerator. EXEC completion could be communicated through an interrupt as well, freeing the calling process from depending on multiple ISBUSY invocations.

[2] All instructions can fail before being delivered if `accId` does not refer to any existing accelerator.

Although it is possible to utilize the inherent interrupt handling mechanism in the core, its significant overhead associated with context switching and Interrupt Service Routine (ISR) invocation motivate us to propose a new *lightweight* interrupt mechanism. It could be used to make synchronous instructions asynchronous and also to inform the process of execution completion. In line with our ISA extension, we design the IXIAM User-space Interrupt Mechanism to live mostly in user-space.

As anticipated in Subsection II-B, the IXIAM User-space Interrupt Mechanism requires an ad-hoc module on the SoC – in the following, the IXIAM User-space Interrupt Module. This module is necessary to gather interrupt messages from the accelerators, keep information, and deliver messages to the cores. The whole mechanism is synthetically depicted in Figure 5 and explained in the following.

In order to handle user-space interrupts, each process should select a function that acts as a User-space Interrupt Service Routine (UISR). To achieve this, we propose the Register User-space Interrupt Service Routine (RUISR) instruction, with the following format: `RUISR <accId, func_addr>`, where `func_addr` indicates the address of the UISR, and `accId` specifies the accelerator whose interrupts should be handled by that routine. Without loss of generality, a possible UISR C signature is the following:

```
void sample_uisr(const uisr_params* params);
```

Where the only parameter is a pointer to a C struct containing the interrupt parameters, such as the `accId` where the interrupt originates, an enum to indicate the interrupt reason, and other possibly useful values to handle it. RUISR execution stores the UISR address in a convenient location of the program memory (e.g., in the Process Control Block), so it can be retrieved and invoked when a user-space interrupt occurs. It is sufficient that a process executes RUISR once for each accelerator it intends to use.

When an interrupt occurs on an accelerator, this sends a User-space Interrupt Message (UIM) to the IXIAM User-space Interrupt Module. The module stores the interrupt information into a private table that can be indexed per `accId`, since at most one process at once can own an accelerator – a fixed size of 256 entries would be sufficient in our proposed implementation. This information includes the aforementioned interrupt parameters, the `procId` of the owning process, and a `coreId`. The latter refers to the core that sent the request ACM carrying the command that originated this accelerator-side interrupt. As we specified, the interrupts can be associated to make synchronous instructions asynchronous or inform the core of the execution completion event. In the first case, the request ACMs already include the id of the originating core, as shown in Figure 3. In the second, it must be added to the EXEC request ACM for this purpose.

Then, the IXIAM User-space Interrupt Module sends the UIM to the core identified by `coreId` and erases its entry. At this point, that core may be executing a thread associated to `procId` (it was the case when the request ACM was sent), or not. In the first case, the core will handle the user-space interrupt by jumping into the stored UISR like a classic user-space function, adopting the same calling conventions dictated by the ISA. If the second case occurs, different policies can be adopted: the user-space interrupt could be elevated to a *classic* interrupt that the OS would be responsible to serve (or discard it if the owning process died), also accessing the owning process' UISR; conversely, it could be enqueued and served later.

III. DISCUSSION

In this Section, we do an in-depth discussion of some aspects that affect our proposal at various levels, as well as we introduce possible development and implementation variations of the proposed scheme.

A. MEMORY COHERENCY

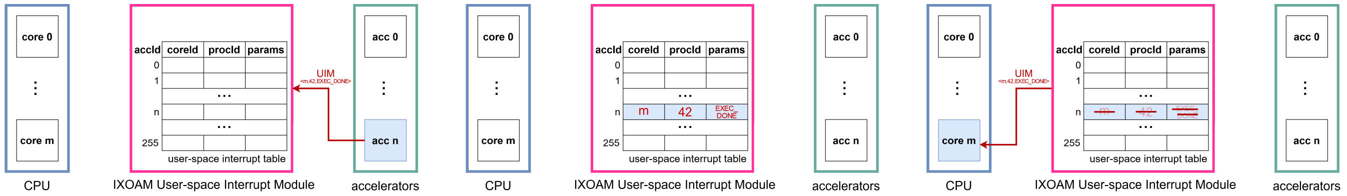
In the proposed configuration, accelerators and processor's cores share the memory space, as they have access to the same physical memory through the on-chip LLC. Through TGL and TGS instructions, they load data buffers from the main memory and store data buffers into it, respectively. Since these buffers are located in a memory space shared with the cores, data coherence issues may arise: a datum write from the accelerator, issued with a TGS, would update the value in the LLC (and DRAM, if LLC is a write-through cache), but an older copy could be cached also in the lower levels of one or more cores' cache hierarchy – L1 and L2. If no coherence mechanism is enforced, a subsequent read performed by such a core of its local copy in the L1 would read an outdated value. The opposite is true, as a core could modify a local datum and the updated value would not be read by a subsequent TGL, unless both L1 and L2 caches adopt write-through policies or some coherence mechanism is enforced.

Core-accelerator coherence is an issue that needs to be addressed in order to have correct programs. According to state-of-the-art, accelerator coherence (or lack of) in SoCs can be addressed in four different ways [24], [30]:

- Non-Coherent;
- LLC-Coherent;
- I/O-Coherent;
- Fully-Coherent.

Non-Coherent Accelerators access directly the DRAM through DMA bypassing the cores' cache hierarchy. In order to get coherent data, the processor caches must be flushed upon accelerator access, so to store updated values in the main memory. This protocol is affordable if large DMA bursts are supported, if few accelerators are connected to the system and each of them executes *long* operations – i.e., at least 1× or 2× DRAM access time. In order to use this protocol, data read/written by the accelerator must be locked.

LLC-Coherent Accelerator's memory requests are sent directly to the LLC. The accelerator is kept coherent with the LLC, but not with the private caches of the



(a) An interrupt occurs on accelerator n. It sends a UIM containing the interrupt information to the IXIAM User-space Interrupt Module.

(b) The module stores the interrupt information carried by the UIM at the n-th row of its user-space interrupt table.

(c) The module forwards the UIM to the m-th core, which delegated to the accelerator the command that generated this user-space interrupt.

FIGURE 5. **The IXIAM User-space Mechanism.**

processor cores, which must be flushed before accelerator's reads and invalidated after accelerator's writes. This model is efficient only if the LLC has high hit rates.

I/O-Coherent Similar to the LLC-coherent accelerator, since accelerator requests are sent directly to the LLC also in this case. The main difference is that the cache hierarchy itself maintains full hardware coherence, and the LLC is responsible for invalidating or recalling data in the lower-level caches.

Fully-Coherent Accelerators are included in the coherence protocol of the processor cores, usually MESI or MOESI. This is a common choice for accelerators that have their own private caches. This protocol permits achieving data coherence transparently, but can be harmful for the system performance if the accelerator has frequent memory accesses, since it would block the bus to the cores most of the time.

In the state-of-the-art, each of these protocols are used based on the requirements and system design. Often, coherence choices are done based on the nature of the accelerator. For instance, the coherence protocol used by GPUs is entirely software-based, with this usually assuming race-free data accesses, regular data storage, and coarse-grained synchronizations [31]. Cache invalidations and flushes are performed at synchronization points. An alternative is DeNovo [31], which does not require writer-initiated invalidation, because it assumes property of written data. Conversely, readers are responsible for invalidations and are tracked by the hardware.

Putting it all together, although our proposed instructions are compatible with any coherence protocol, in this work we assume LLC-coherent accelerators. From the architectural point of view, as shown in Figure 1, all the accelerators are connected to the LLC. Before a TGL instructions that would load data into the accelerator, private L1 and L2 caches of the cores must be flushed in order to update data values at least in the LLC. If supported by the architecture, the best option would be to flush only the dirty cache lines involved. Before a TGS instruction that would store data from the accelerator into the LLC, all private copies in L1 and L2 caches must be invalidated. Again, if supported, the best option would be to invalidate only the relevant lines – i.e., do a *selective* invalidation. This means that, in the LLC-coherent case, our instructions should be integrated with proper signals addressing the caches. It could be done by invoking cache-oriented instructions before ours, if provided by the ISA, or by sending the proper signals during TGL and TGS execution.

B. DATA CONSISTENCY

Another problem that may arise from different processing elements (i.e., cores and accelerators) loading data from / storing data to a shared memory space in parallel is the difficulty to maintain *data consistency*. This issue is relevant in our proposed framework.

To illustrate it in the context of our proposal, let's consider Listing 3. The registers in the example hold the following values: `t0` holds an accelerator id `accId`, `t1` holds a pointer to the main memory `cpu_ptr`, `t2` holds a location in the accelerator's local memory expressed in IXIAM Location Format `acc_ptr`, `t3` holds a buffer size in bytes `size`, and `t4` can hold any value. Therefore, the first instruction instructs the accelerator to store `size` bytes of data from the location `acc_ptr` to the main memory, at address `cpu_ptr`. The following instruction loads 64-bit from `cpu_ptr` into the register `t4`. So, it is a classic Read-After-Write (RAW) dependency: the LD instruction loads a value from a memory location that should have been just written by the previous TGS instruction – thus, it should read an *updated* value.

Algorithm 3 Sample Code Causing Memory Contention

```
/* t0=accId, t1=cpu_ptr,
   t2=acc_ptr, t3=size, t4=any    */
TGS t3, t2, t1;
LD t4, 0(t1);
```

TGS, as previously specified, is committed immediately on the core side as soon as the request ACM is sent. It is possible that, while the following load instruction LD enters the pipeline stages to be fetched, decoded, and executed, the data transfer is not complete, or did not even start if the request ACM has not reached the accelerator yet. Even worse, in principle, if the processor implements *Out-of-Order* (OoO) execution, the LD instruction may be executed *even before* the

TGS instruction. In both cases, it is possible that LD reads an outdated value and the correctness of the program is not guaranteed.

This is a very well-known and studied problem. Various solutions have been proposed during the years to address it, since its emergence that dates back to the diffusion of multitasking execution [32], [33].

Commonly, the processes use semaphores to access shared memory in a mutually exclusive manner [34]. They disallow shared data swapping and allow the OS kernel to avoid the internal page locking code. In this regard, some levels of concurrency control is applied to maintain data integrity in the shared memory.

Atomic Read-Modify-Write (RMW) instructions used in different ISAs, e.g., x86, IBM Power, ARMv8, and RISC-V are another solution to synchronize operations [35]. They are used either directly by programmers or OS libraries to provide higher-abstraction synchronization mechanisms. Locks, barriers, and other constructs are utilized to establish mutual exclusion among threads of parallel applications in those synchronization mechanisms [36]. The atomic RMWs in current Intel x86 processors serialize all outstanding load and store operations, and block subsequent memory operations until their commit [37]. Through the use of memory fences, i.e., fetch-and-increment, test-and-set, and compare-and-swap, this serialization can be easily implemented [38], [39].

Another way to guarantee consistency is Load Linked/-Store Conditional (LL/SC) pairs [40]. LL/SC are pairs of instructions that can be used to implement, in software, the same atomic operations as RMWs. Unlike the latter, however, LL and SC are distinct instructions, usually used to surround another operation. Thus, the whole primitive can be interrupted, e.g., by a context switch in-between. Moreover, an LL/SC pair will fail due to relevant external events such as coherence invalidations and cache evictions [35].

Another option are memory arbitration mechanisms, which have been proposed to decrease the impact of memory contention in contexts with multiple cores [32], [41]. This arbitration mechanism have been presented by the PREM execution model [42], [43], BWLOCK [44], and MEMGUARD [45].

In the rather common case of a CPU paired with an integrated GPU, both computing elements have access to the same physical memory – so, a reliable synchronization and memory consistency support is needed [31]. Two main memory consistency models widely used in GPUs are *Data-Race-Free* (DRF) [31], [46] and *Heterogeneous-Race-Free* (HRF) [47]. Common implementations of DRF and HRF enforce a program order requirement. DRF ensures sequential consistency to datarace-free programs, while HRF is defined similarly to DRF to handle scoped (i.e., a hardware-inspired mechanism that exposes the memory hierarchy to the programmer) synchronization problem of DFR. Each synchronization access has a scope attribute in the proposed HRF [31], [48]. Although HRF is a very well-defined model, it cannot hide the inherent complexity of using scopes.

The CUDA programming language for NVIDIA GPUs defines two memory fences, `__threadfence` and `__threadfence_block`, to enforce consistency between loads and stores [29], [49]. VectorPU expands CUDA with C++ functions that act as access specifiers for buffers [50]. This protocol realizes automatic creation of on-demand copies of to-be accessed elements in device memory, with all the copies kept coherent and data transfers performed only when needed. Global Memory for Accelerators (GMAC) [51] expands CUDA as well with device-independent allocation/deallocation primitives that enforce coherence.

Some works are based on data-containers, sometimes also referred to as *smart* containers: objects to perform coherent software caching of accessed elements in the different memory spaces [52], [53]. Therefore, they can be reused in order to avoid unnecessary data transfers, which are usually transferred lazily when needed.

As manually managing data to ensure programmers to have a consistent program state between the CPU and GPU memories is tedious and error-prone, some semi-automatic techniques have been proposed. These techniques are usually annotation-based, with programmers specifying the access modes of buffers in memory [52].

There are also completely automatic solutions to manage data and optimize communication. Some examples are Inspector-Executor (IE) [54], CPU-GPU Communication Management (CGCM) [55], and DyManD, an automatic system to manage complex and recursive data-structures without static analyses [56].

In general, relying on an explicit use of ad-hoc fences is the most straightforward solution to address consistency issues. In its most basic version, it puts the burden on the application programmer, but it could be easily wrapped in library constructs or be integrated in (semi-)automatic techniques.

In this work, we define our fence and show how to use it explicitly. We leave an in-depth analysis of alternative, more sophisticated techniques to future work.

We propose an ad-hoc fence instruction: Accelerator FENCE (AFENCE). We take inspiration from the simple RISC-V FENCE instruction, which blocks all the subsequent loads and stores. Its format is `AFENCE <accId>`, where `accId` identifies the target accelerator. Also in this case, `procId` is added to the associated request ACM with the usual role: to check if the process executing the instruction is the same as the owning process, so to ignore the request if that is not the case.

The AFENCE instruction should be selectively inserted in the code flow to enforce consistency. It should be placed between a read instruction and a write instruction involving overlapping memory locations. Listing 4 shows the three RAW hazards solved by using AFENCE this way, with the first one being the example from Listing 3 *corrected* with the fence.

Algorithm 4 Three Possible Core-Accelerator RAW Hazards Solved With AFENCE Instruction

```
/* t0=accId, t1=cpu_ptr,
   t2=acc_ptr, t3=size, t4=dummy */
/* acc->core                     */
TGS t3, t2, t1;
AFENCE t0;
LD t4, 0(t1);
/* core->acc                     */
SD t4, 0(t1);
AFENCE t0;
TGL t3, t1, t2;
/* acc->acc                      */
TGS t3, t2, t1;
AFENCE t0;
TGL t3, t1, t2;
```

We implement AFENCE as a synchronous instruction. It drains the Load-Store Queue (LSQ), and prevents any subsequent load or store instruction from surpassing it, being it core-oriented (e.g., LD, SD) or accelerator-oriented (e.g., TGS, TGL). It causes a request ACM to be sent to the accelerator and instruct it to do the same with its flying loads and stores: wait for their completion and prevent any subsequent memory operation to surpass the AFENCE. When these are complete, the accelerator sends the response ACM to the core to instruct it to commit the AFENCE and proceed. Should the accelerator be controlled by a single in-order instruction queue, the implementation of its logic would be trivial. It would be sufficient that the response ACM is sent when the AFENCE is popped from the head of the instruction queue. Moreover, in that case, the third RAW issue depicted in Listing 4 would not need any fences and would be solved *by design*.

C. WATCHDOG

It is possible that a process does not release the accelerator after the demanded job is completed. This can happen for various reasons: a code bug, malicious behaviour, the process died before executing the release instruction, etc. To prevent this scenario, we design a hardware *watchdog* mechanism that keeps track of the accelerator utilization.

The watchdog lays in the accelerator interface logic circuitry and interacts with it: when an accelerator completes an EXEC operation, the watchdog's stopwatch starts. If a RELEASE request is not received after a configurable number of clock cycles, the watchdog releases the accelerator automatically. The same mechanism can be applied for any request sent to the accelerator that is not followed, within a *reasonable* and configurable time-span, by a request that would come in a normal utilization flow. An example is a RELEASE that would normally be followed by a CHECK.

D. OS-BASED ARBITRATION

Our instruction extension is designed so all instructions can be executed from user-space, with the minimum privilege level. Tasks that would normally require higher privilege levels (e.g., reservation management), usually delegated to the OS, are implemented in hardware. This way, we are reducing the latency associated to many operations, as we will show in the next section.

These functionalities implemented in hardware, despite taking advantage of a reduced latency, necessarily require a higher chip area to host the transistors to implement them. Moreover, the flexibility that can be adopted in a hardware-based solution is generally significantly lower than that achievable in software.

Focusing on reservation queues and their management logic, the hardware reservation mechanism allows users to leverage accelerators with no OS intervention, pushing their requests in hardware queues that will extract them with a FIFO policy. Although this solution could be sufficient for various real use cases, it could be too limiting for others. For instance, in heavily virtualized environments, a finer grain distinction between what users (and their processes) can do or cannot do would certainly be needed. Configurable access policies based on subscription options could be needed as well, even preventing some accelerators from user-space programs altogether. Thus, a more strict control from the OS, despite not being the solution with the lowest latency, may be needed.

We could achieve this by elevating RESERVE and RELEASE instructions at kernel privilege level. This way, user processes would rely on apposite system calls to ask for an accelerator reservation/release, and the system call execution in kernel-space would implement custom logic to allow/prevent reservation/release and execute the high-privilege RESERVE/RELEASE instructions as part of its code.

Apart from allowing a finer-grained access control, this solution would simplify the management of other relevant scenarios. First, watchdog logic could be delegated to the OS, that could force release of inactive processes, or even monitor (and limit) the accelerator usage after a configurable time interval. Any scheduling policy could be implemented by the OS – e.g., round robin or priority-based scheduling. In the round robin case, the preemption implementation would require the OS to release the accelerator from the owning process, reserve it for itself, save its state to convenient locations, restore the state of the next process (if present), release the accelerator, and reserve it for the newly-scheduled process. The backed-up state would be restored the next time the preempted process is scheduled. This is doable by adding to the high-privilege instructions an explicit `procId` operand, which is a safe choice because it would be limited to kernel-space code.

One of the advantages of relying on user-space instructions is that the code would work also in virtual environments

TABLE 2. System configuration details.

CPU	Quad-core, 3.4GHz, RISC-V
L1 I/D Cache	32KB, 8-way, write-back, 64B block size, non-blocking, 2-cycles access time, private
L2 Cache	512KB, 8-way, write-back, 64B block size, non-blocking, 10-cycles access time, private
L3 Cache	8MB, 16-way, write-back, 64B block size, non-blocking, 36-cycles access time, shared
Interconnection	ring-based, 15-cycles average latency
Main Memory	DRAM-DDR3, 2GB, 300-cycles access time

TABLE 3. Latencies associated to the proposed instructions. L_{CPU} is the number of CPU cycles needed to fetch and decode instructions on the CPU. L_{acc} is the number of accelerator cycles needed to fetch and decode commands on the accelerator.

Command	L_{CPU}	L_{acc}	Command	L_{CPU}	L_{acc}
RESERVE	2	3	TRS	2	1
CHECK	2	3	EXEC	2	1
TGL	2	1	BUSY	2	1
TGS	2	1	RELEASE	2	3
TL	2	1	AFENCE	2	1
TRL	2	1			

with no substantial modifications. Elevating RESERVE and RELEASE privilege level would not change the virtualization-friendliness of the proposal, as it would require only a proper extension of the hypervisor to take care of the system calls associated to them. This hypervisor modification would allow an implementation of the subscription policies mentioned above, and would be possible with a limited effort from kernel programmers.

IV. EXPERIMENTAL RESULTS

In order to assess the value of our proposed interfacing for a variety of accelerators, we implement five different accelerators that meet the constraints expressed in Section II and pertain to different domains:

MatMul accelerates matrix-matrix multiplication;
Crypto accelerates data encryption/decryption;
DisparityMap accelerates disparity map calculation;
FFT accelerates Fast Fourier Transform;
CNN accelerates Convolutional Neural Network computation.

We model them as part of a simulated SoC in the gem5 simulator v21.2.1 [22]. The SoC includes a quad-core RISC-V processor, three levels of on-chip caches, and a dedicated NoC to connect the cores to the accelerators. We enrich the SoC with the additions dictated by the IXIAM Hardware Infrastructure, the IXIAM User-space Interrupt Module, and all the logic necessary to execute the proposed instructions and the user-space interrupt mechanism. The accelerators are connected to the L3 cache to access the main memory. The details of the system configuration are depicted in Table 2.

For each accelerator, we implement a benchmark in which the accelerator performs one offloaded operation. We execute each benchmark with two different interfacing and compare the obtained performance. In one case, we adopt our proposal, in the other, we use a conventional driver-based solution.

A. SIMULATED ACCELERATORS

We select five accelerators from different domains and with different characteristics, so to show the flexibility of our approach in terms of the variety of accelerators supported.

Table 3 lists the CPU-side and accelerator-side latencies associated to the fetch and decode of each instruction. Accelerators receive these commands in request ACMs together with their parameters, through the interconnect. The latencies in the table are expressed with respect to their clock cycles and are the same for all the simulated accelerators.

1) MatMul

We design a Matrix Multiplication accelerator, *MatMul* in this paper, that calculates the matrix $C \in \mathbb{R}^{M\times P}$ as the product of matrices $A \in \mathbb{R}^{M\times N}$ and $B \in \mathbb{R}^{N\times P}$, according to Eq. 1.

$$C_{ij} = \sum_{k=1}^{N} A_{ik} \cdot B_{kj} \quad \text{for } i = 1, \ldots, M;\ j = 1, \ldots, P \qquad (1)$$

Figure 6 shows the block diagram of the MatMul accelerator. Each element is represented as an IEEE-754 single precision floating-point value. It has three local memories to hold the three matrices: LM0, LM1, and LM2 for matrices A, B, and C, respectively. Its register file has three 32-bit registers to hold matrix dimensions M, N, and P.

The accelerator performs the matrix-matrix multiplication by multiplying fixed-side tiles together. Its computing core is a matrix of Processing Elements (PEs), each one capable of performing one tile-tile multiplication and one tile-tile addition. An internal controller fetches tiles from the input matrices, multiply them together, and stores the resulting tile into LM2 until all the output matrix has been covered.

We assume each PE is equivalent to an NVIDIA Tensor Core, and can perform a multiplication between two 4×4 tiles in one clock cycle [57]. Analyzing the die-shot of an NVIDIA TU102 chip [58], considering it features 72 Streaming Multiprocessors (SMs), each with 8 Tensor Cores [28], and considering the whole die area is 754 mm^2 [59], we calculate that each SM occupies around 1.98 mm^2 (0.263% of the total area), and thus each Tensor Core occupies at most 0.25 mm^2. We assume PEs in our MatMul accelerator have this size.

We design an accelerator with a 7 mm^2 die area, similar to a Kirin 990 5G NPU [60]. It has three 1 MiB local memories, that take 0.46 mm^2 with a 12 nm technology node (i.e., the same as the Tensor Core), as we estimate with CACTI [61]. This way, the MatMul accelerator can host up to 16 PEs arranged in a 4 × 4 matrix, as they would occupy 4 mm^2 and leave 2.54 mm^2 (36.28%) for the remaining *uncore* logic.

The fetching logic can fetch up to 4 tiles from A, each one broadcasting to a PE row, and 4 from B, each one broadcasting to a PE column.

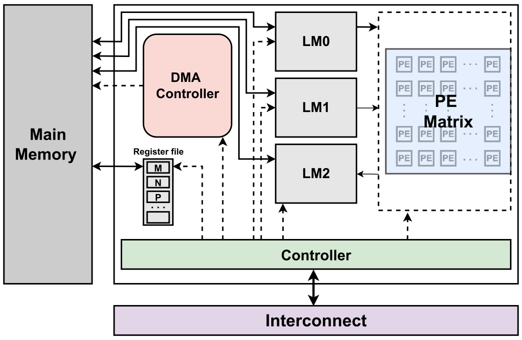


FIGURE 6. Block diagram of the MatMul accelerator.

Figure 7 shows the steps performed by the accelerator. For simplicity, we show a 2 × 2 PE matrix. Five distinct phases can be identified as follows:

Fetch (F) 4 tiles from matrix *A* are fetched from LM0 and broadcasted to each row of PEs. At the same time, 4 tiles from matrix *B* are fetched from LM1 and broadcasted to each column of PEs.

Multiply (M) in each PE, the two tiles are multiplied together.

Add (A) in each PE, the result tile extracted from the multiplication is added to an all-zero block at the first iteration, and to the partial result of the PE in the following iterations.

Accumulate (Ac) in each PE, the result tile extracted from the addition is stored in an accumulator large enough to hold one tile.

Store (S) in each PE, when the last tile from a row in *A* and the last tile from a column in *B* are multiplied together and added to the partial result in the accumulator, the resulting *C*'s tile is available and is stored to LM2.

As depicted in Figure 7, some phases are independent and can be safely overlapped in a pipeline. Considering the matrices *A*, *B*, and *C* in the figure, the resulting pipeline will be as follows:

F M A Ac
F M A Ac
F M A S

In order to operate the accelerator with our proposed interface, the programmer needs to write the matrix dimensions in the registers (TRL instructions), the matrix data in the local memories (TGL instructions), and trigger the matrix multiplication operation (EXEC instruction). The accelerator's controller will take care of performing all the steps needed to multiply the matrices together in the tiling fashion expressed above and write the resulting matrix *C* to LM2. When done, the programmer can read *C* from LM2 and copy it into the main memory (TGS instruction).

Table 4 summarizes the accelerator configuration parameters.

TABLE 4. Configuration parameters of the MatMul accelerator.

Frequency	1.0 GHz	**LM0 size**	1 MiB
Add latency	1 cycle	**LM1 size**	1 MiB
Mul latency	1 cycle	**LM2 size**	1 MiB
PEs	4×4	**LMs latency**	1 cycle
tile dims	4×4		

2) CRYPTO

Cryptography accelerator, which is called *Crypto* in this paper, encrypts/decrypts a given message using NIST Advanced Encryption Standard (AES) cipher [62]. The AES algorithm involves dividing the input into 16-byte blocks, represented as 4 × 4 byte matrices called *state* and applying the encryption/decryption algorithm to each block separately. According to the standard, there are three key lengths that can be used: 128, 192, and 256 bits [62]. Commonly, the algorithm is denoted as AES128, AES192, or AES256 depending on the key-length.

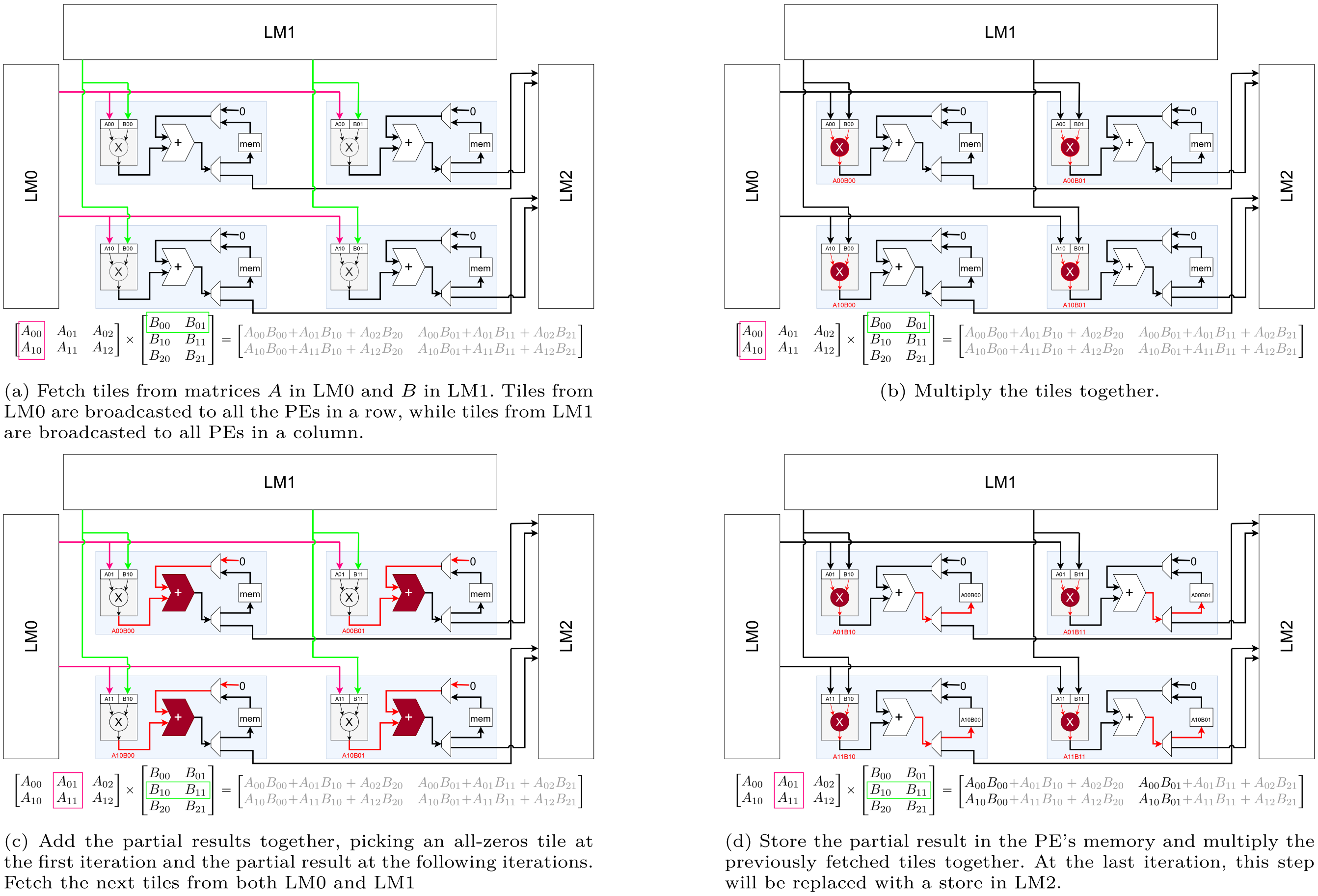
The classic AES algorithm requires the execution of the same sequence of operations for each *round*, with the number of rounds being 10, 12, or 14, according to the key length being 128, 192, or 256 bits, respectively. Since each round needs its own key, the initial key is expanded into an array of keys through the so-called *KeyExpansion* phase.

Then, for the encryption algorithm, the rounds are performed as fixed sequences of operations involving byte substitution (*SubBytes*), byte permutation (*ShiftRows, MixColumns*), and byte-key XORring (*AddRoundKey*). The decryption algorithm is similar, but the *plain* operations are replaced with their inverses. In both cases, the final round does not comprise *MixColumns* nor its inverse operation.

The accelerator implementation is based on the AES Engine IP core by Xilinx [63], which provides some variants of AES128, AES192, and AES256 ciphers. In our design, shown in Figure 8, we add a controller, a DMA controller, a small register file, and a global buffer alongside the AES Engine. These components are used to communicate with the AES Engine and orchestrate the cipher execution on it.

The global buffer is used to store both input (i.e., message and initial key) and output data locally. The input is retrieved from the main memory and the output is stored in the main memory, with the DMA controller managing the transfers. The register file provides three registers to store data addresses in the global buffer (i.e., message, initial key, and output) and one to store the message length. The controller is in charge of interpreting commands received as request ACMs from the interconnect and execute the necessary operations related to them. It also orchestrates execution on the AES engine by providing message blocks in a streaming fashion, taking advantage of the pipelining capabilities of the AES Engine [63].

In order to use the accelerator, a programmer needs to load the message length into the *size* local register (TRL instruction), the initial key into the global buffer (TGL instruction),


LM1
LM0
LM2
mem
0
+
X
A00 B00
A00 B01
A10 B00
A10 B01
A01 B10
A01 B11
A11 B10
A11 B11
A00B00
A00B01
A10B00
A10B01
A01B10
A01B11
A11B10
A11B11
$\begin{bmatrix}A_{00}&A_{01}&A_{02}\\A_{10}&A_{11}&A_{12}\end{bmatrix}\times\begin{bmatrix}B_{00}&B_{01}\\B_{10}&B_{11}\\B_{20}&B_{21}\end{bmatrix}=\begin{bmatrix}A_{00}B_{00}+A_{01}B_{10}+A_{02}B_{20}&A_{00}B_{01}+A_{01}B_{11}+A_{02}B_{21}\\A_{10}B_{00}+A_{11}B_{10}+A_{12}B_{20}&A_{10}B_{01}+A_{11}B_{11}+A_{12}B_{21}\end{bmatrix}$

(a) Fetch tiles from matrices A in LM0 and B in LM1. Tiles from LM0 are broadcasted to all the PEs in a row, while tiles from LM1 are broadcasted to all PEs in a column.

(b) Multiply the tiles together.

(c) Add the partial results together, picking an all-zeros tile at the first iteration and the partial result at the following iterations. Fetch the next tiles from both LM0 and LM1

(d) Store the partial result in the PE's memory and multiply the previously fetched tiles together. At the last iteration, this step will be replaced with a store in LM2.

FIGURE 7. Highlight of a MatMul accelerator with a 2 × 2 PE matrix and its functioning.

TABLE 5. Configuration parameters of the Crypto accelerator.

Frequency	250 MHz	**Global buffer size**	2 MiB
Encr AES128 block	12 cycles	**Global buffer latency**	2 cycles
Decr AES128 block	22 cycles		

and the message into the global buffer (TGL instruction). Then, execution can be triggered (EXEC instruction) and, when it completes, the result can be retrieved by reading from the global buffer (TGS instruction).

Table 5 summarizes the accelerator configuration parameters.

3) DisparityMap

A disparity map accelerator is used to compute the disparity map of two stereo images (i.e., left image and right image). Given a stereo image pair, a disparity map can be used to reconstruct depth information about the scene represented in the two images.

We simulate the disparity map accelerator described by Ibarra-Manzano et al. in [64], where the authors discuss an FPGA-based implementation. They use an algorithm based on the correlation of census transforms [65] of input images.

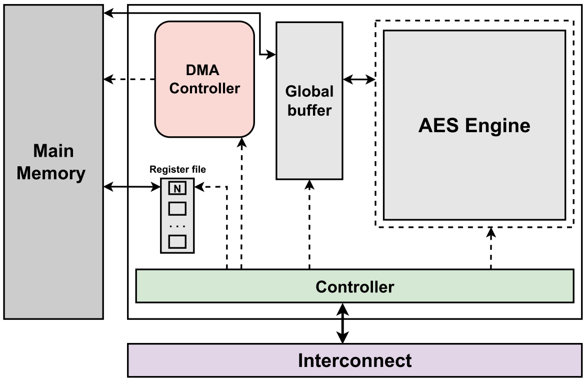


FIGURE 8. Block diagram of the Crypto accelerator.

Initially, through arithmetic mean filter, input images are smoothed to remove acquisition noise (*Image pre-processing*). This process is carried out using a 3 × 3 pixels window. In the next step, filtered images are processed by the census transform unit, producing images in which each pixel is represented as a binary string that encodes intensity with respect to neighbor pixels (*Census transformation*). In this case, a 7 × 7 window is used. Finally, census transforms are

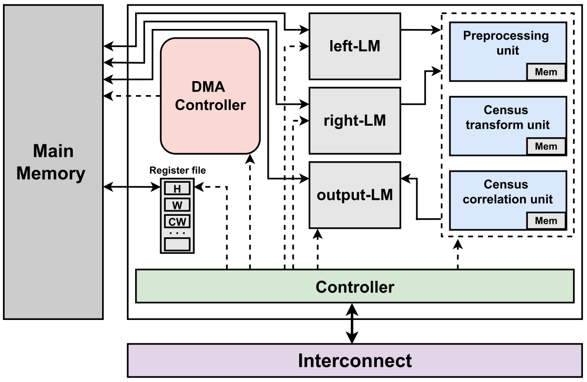


FIGURE 9. Block diagram of the Disparity accelerator.

TABLE 6. Configuration parameters of the DisparityMap accelerator.

Frequency	1.0 GHz	**left-LM size**	2 MiB
192×144 latency	34500 cycles	**right-LM size**	2 MiB
384×288 latency	138000 cycles	**out-LM size**	2 MiB
640×480 latency	384000 cycles	**LMs latency**	3 cycles

used as input of census correlation unit, that is able to find disparity measures (*Census correlation*). The maximal disparity measure selected is 64. Each algorithm step is implemented as a separate module in the accelerator.

The implemented accelerator has three 2 MiB local memories used to store input left image (left-LM), input right image (right-LM), and processing output (out-LM). In addition, each module is equipped with memories used to save temporary values during computation transparently. The total amount of these memories is 104 Kib.

In order to use the disparity map accelerator, the programmer needs to write configuration parameters, such as census transforms window dimension and input images size, in accelerator registers (TRL instructions). Then, input images must be loaded in the local memories (TGL intructions). Execution is triggered (EXEC instruction) and, after processing, the result is retrieved from accelerator local memory and copied into main memory (TGS instruction).

Table 6 summarizes the accelerator configuration parameters.

4) FFT

FFT accelerator is used to calculate the Fourier Transform of an N-point array of IEEE-754 single precision complex numbers and its inverse, transforming it from time to frequency domain (Eq. 2) and vice versa (Eq. 3). The transformed array is an N-point array as well.

$$X_k = \sum_{n=0}^{N-1} (x_n e^{\frac{-i2\pi kn}{N}}) \tag{2}$$

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} (X_k e^{\frac{-i2\pi kn}{N}}) \tag{3}$$

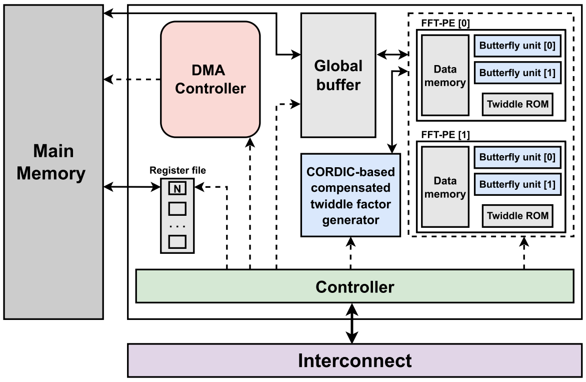


FIGURE 10. Block diagram of the FFT accelerator.

TABLE 7. Configuration parameters of the FFT accelerator.

Frequency	1.0 GHz	**256 elems latency**	277 cycles
LM size	4 MiB	**1 Kelems latency**	1380 cycles
LM latency	3 cycles	**4 Kelems latency**	21600 cycles
4 elems latency	2 cycles	**16 Kelems latency**	76900 cycles
16 elems latency	9 cycles	**64 Kelems latency**	306300 cycles
64 elems latency	52 cycles	**256 Kelems latency**	1285400 cycles

We implement an FFT accelerator based on the accelerator proposed in [66]. The algorithm used to address the transformation is a modified version of the well-known Cooley-Tukey algorithm [67], which uses matrix transposition in order to better exploit memory access.

The accelerator includes two processing elements called FFT-PEs. Each FFT-PE is composed by two butterfly units and a 16 KiB dedicated dual port multi-banked memory.

FFT-PEs share a 4 MiB local memory used to store initial input points and final transformed output. During computation, each FFT-PE fetches and saves intermediate data in that memory.

The authors adopt a performance optimization technique called *hybrid twiddle factor generation*. This technique uses a ROM to store M twiddle factors, which are used by the butterfly units. The higher order factors are generated by a CORDIC-based [68] compensated twiddle factor generator, to accommodate higher size FFT. Figure 10 shows the block diagram of the implemented FFT accelerator.

Each FFT-PE can compute transform of at most 2^{10} points. It is possible to compute the FFT of up to 2^{20} points by combining the two FFT-PEs. In the latter case, the computation is split into two smaller batches that are assigned to each FFT-PE.

In order to use the accelerator, a programmer needs to write the input array size in the dedicated register (TRL instruction), send input data (TGL instruction), trigger the execution (EXEC instruction) and, finally, retrieve output data from accelerator local memory (TGS instruction).

Table 7 summarizes the accelerator configuration parameters.

5) CNN

CNN accelerator is used to accelerate the convolution between a three-dimensional tensor (input) and a

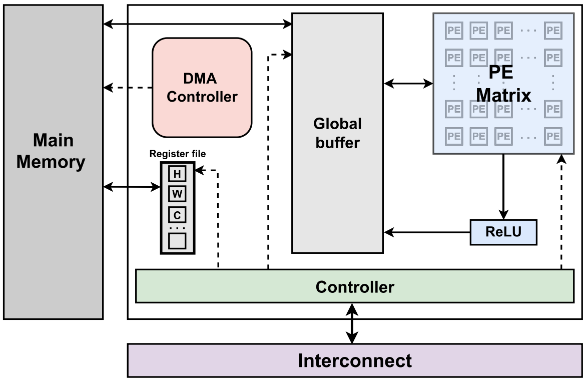


FIGURE 11. Block diagram of the CNN accelerator.

four-dimensional one (weight, filter or kernel), to produce a three-dimensional output tensor. This operation is the heart of Convolutional Neural Networks (CNN). The tensor dimensions are height (H), width (W), depth (C), and batch size (N).

The output is obtained multiplying elements of input and weight through a multiply-and-accumulate operation, according to Eq. 4. An activation function α (e.g., sigmoid, hyperbolic tangent, ReLU) is applied to each output element in order to introduce non-linearity.

$$O_{h_o,w_o,c_o} = \alpha\left(\sum_{n=0}^{H_f}\sum_{m=0}^{W_f}\sum_{i=0}^{C_i} I_{h_o+n,w_o+m,i} * F_{n,m,i,c_o}\right) \quad (4)$$

We implement CNN accelerator based on Eyeriss [69]. The computation is carried out by a 12 × 14 matrix of processing elements (PEs). Data is fetched from accelerator's local memory to PEs and orchestrated by the internal controller according to *row-stationary* dataflow (i.e., a particular dataflow that allows to efficiently reuse both output and weight elements during computation [69]). The accelerator has a 192 KiB local memory (*global buffer*), used to store each type of tensor involved in the convolution (input, weight, and output). Figure 11 shows the block diagram of the CNN accelerator.

Hyperparameters for convolution operation, such as tensor dimensions, are stored in the accelerator register file (TRL instruction). Input, weight and, eventually, partial sums are transferred from main memory to accelerator's global buffer (TGL instruction). After data loading, execution is triggered (EXEC instruction) and an internal controller orchestrates data movement according to a row-stationary dataflow. When execution finishes, output data are retrieved from a dedicated local memory and stored in main memory (TGS instruction).

It can happen that local memories are not big enough to store tensors involved in the whole computation. In this case, a tiled execution is mandatory. TRL, TGL, and TGS instructions can be used accordingly to a tiling strategy to move data from/to the accelerator.

Table 8 summarizes the accelerator configuration parameters.

TABLE 8. Configuration parameters of the CNN accelerator.

Frequency	200 MHz
LM size	192 KiB
LM latency	3 cycles
Throughput	40 GMACs

B. COMPARISON

For each accelerator, we implement a benchmark where the user code offloads one operation to the corresponding accelerator. The *nature* of the offloaded operations can be described as follows:

MatMul Multiplication between two square matrices;

Crypto Buffer ciphering through the AES128 algorithm;

DisparityMap Calculation of a single image disparity map;

FFT Fast-Fourier Transform of a vector of complex numbers;

CNN Convolution between a three-dimensional input tensor and a four-dimensional filter tensor.

In the following, we study the performance achieved by the benchmarks in correspondence of various workload sizes.

1) BASELINE LINUX DRIVER

We evaluate our proposal by comparing the performance obtained with the IXIAM interfacing and that obtained with a traditional reference interfacing. We implement the latter as a Linux driver designed according to high-performance principles adopted in modern kernel modules such as UACCE [70] and io_uring [71].

The driver is organized as a kernel module and a user-space driver. The user-space driver wraps the communication with the kernel module, giving a suitable interface to user-processes. In the kernel module, two single-consumer single-producer ring buffers are allocated for each user-process opening the device file associated. One, the *submission* ring-buffer, is used by a user-space process to enqueue commands (producer) and by the kernel to retrieve them (consumer). The other, the *completion* ring buffer, is used in a specular manner: the kernel enqueues the outputs from executed commands (producer) and the user-space process reads them (consumer). These buffers are made available to the user-process through memory mapping.

The *core* of the kernel module is a kernel thread that waits in a waiting queue for commands to arrive. The notification to wake the thread up comes from a *ioctl* system call and must be called from the user-space once one or more commands have been enqueued in the submission ring buffer. Once the thread wakes up, it cycles over the commands in the submission ring buffer, executes them in sequence, and enqueues their outputs in the completion ring buffer. The single-producer single-consumer nature of the ring buffers is such that no synchronization is necessary between user-process and kernel-thread: each of them can safely write data in the ring buffer where it acts as a producer and increment the index when done. Also, no synchronization between different user-processes is

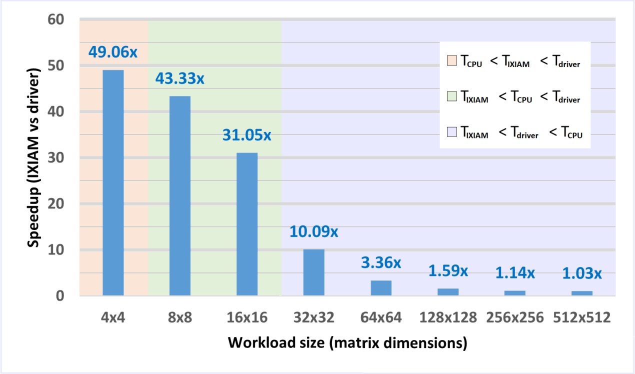


FIGURE 12. Speedup of IXIAM-based vs driver-based matrix-matrix multiplication on a MatMul accelerator, shown for eight workload sizes. In the legend, T_{CPU}, T_{IXIAM}, and T_{driver} refer to the execution time of a CPU-only, IXIAM-based, and driver-based implementation, respectively (lower is better).

necessary, since each user-process has its exclusive pair of ring buffers.

2) BENCHMARKS

Figure 12 shows the speedup given by our proposal with respect to a conventional driver-based interfacing (in the following, just "speedup") when accelerating a multiplication between two square matrices of the same size on a MatMul accelerator, in the case of eight different workloads. The speedup decreases as the workload size increases, with the maximum speedup amounting to 49.06× for a multiplication between two 4 × 4 matrices. Up to 64 × 64 matrices the speedup is still above 3× (3.36×), and it decreases down to around 3% with 512 × 512 matrices.

In Figure 12, we identify three areas: in the first area, in light orange, a CPU-based implementation achieves better performance than both accelerated solutions (i.e., IXIAM and driver-based); in the second area, in light green, an IXIAM implementation surpasses a CPU-only one, but this is still better performance-wise than a driver-based solution; in the third area, in light purple, both accelerated solutions achieve better performance than a CPU-only one, with IXIAM still being preferable to a driver-based interfacing. Looking at these areas, it is evident that our proposal enlarges the set of workloads where an accelerated implementation is palatable (second area, with 8 × 8 and 16 × 16 workloads). Even for those that already benefit from acceleration (third area), the speedup gain makes our proposal a more convenient solution with respect to drivers (e.g., 10.09× for 32 × 32 matrices), and even more with respect to CPU-only.

Figure 13 shows the speedup achieved by accelerating an AES block cipher on a Crypto accelerator in the case of eight workload sizes. Also in this case, the speedup decreases as the workload size increases: from 78.71× with 16 bytes, it is still more than 10× with 1 KiB workload, around 3.71× with 4 KiB, down to circa 19% with 64 KiB, and it reaches less than 4% from 256 KiB on.

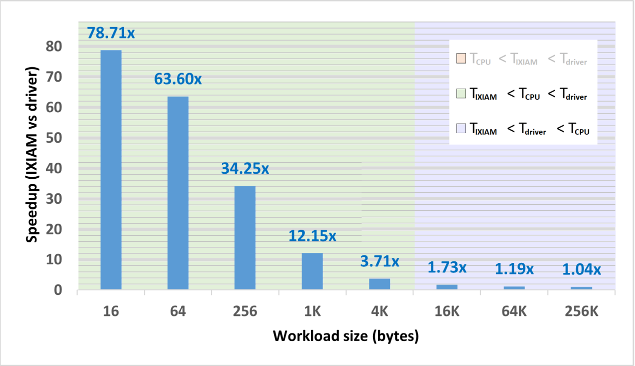


FIGURE 13. Speedup of IXIAM-based vs driver-based AES ciphering on a Crypto accelerator, shown for eight workload sizes. In the legend, T_{CPU}, T_{IXIAM}, and T_{driver} refer to the execution time of a CPU-only, IXIAM-based, and driver-based implementation, respectively (lower is better).

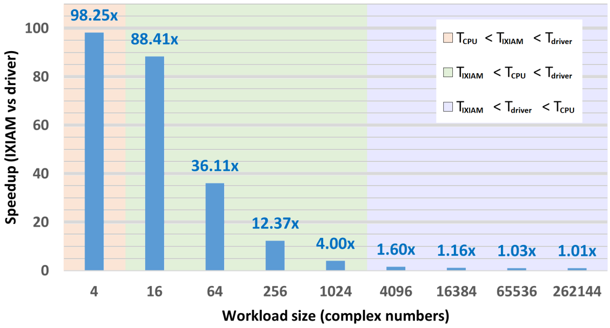


FIGURE 14. Speedup of IXIAM-based vs driver-based FFT calculation on an FFT accelerator, shown for nine workload sizes. In the legend, T_{CPU}, T_{IXIAM}, and T_{driver} refer to the execution time of a CPU-only, IXIAM-based, and driver-based implementation, respectively (lower is better).

As for the comparison against a CPU-only solution, we use the AES implementation provided by libcrypto, which is shipped with OpenSSL 3.2.0 and includes highly tuned RISC-V assembly source code [72], [73]. In this case, even the smallest workload (16B) benefits from IXIAM-based acceleration, as the absence of a light orange area testifies. The driver-based acceleration, conversely, is faster than a CPU-only implementation for workloads of 16 KiB and higher. Also in this case, our proposal enlarges the set of workloads that benefit from acceleration.

Figure 14 shows the achieved speedup of a Fast Fourier Transform performed on nine workloads on the FFT accelerator. Also in this case, we observe the speedup monotonically decreasing as the workload size increases. With 4 complex numbers, it amounts to 98.25×, and decreases down to 4× with 1024 numbers. It stays above 3% up to 65536 elements and drops below 1% for more than 262144 elements.

The CPU-only implementation is based on FFTW3 [74] compiled for RISC-V. Its performance surpasses that achievable with an IXIAM interfacing only in the case of the smallest workload considered (4 complex numbers). With 16 and up to 1024 numbers, the workloads are in the light

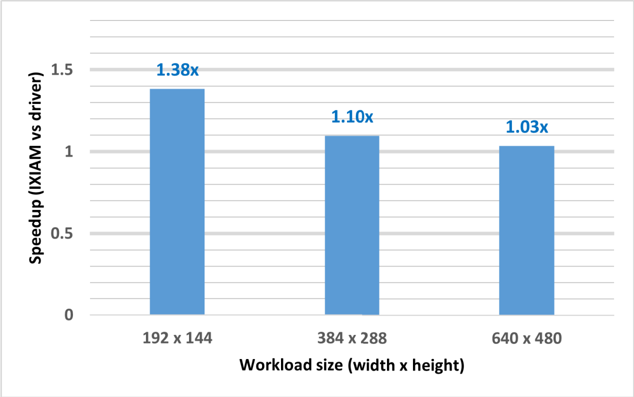


FIGURE 15. Speedup of IXIAM vs a driver-based disparity map calculation on a DisparityMap accelerator, shown for three workload sizes.

TABLE 9. Tensor dimensions of the convolutional layers of LeNet-5, AlexNet, and ResNet CNNs. We show stride (s), padding (p), height (H), width (W), number of channels (C), and batch size (N).

			Input			Filter			
	s	p	H	W	C	H	W	C	N
LeNet-5	1	0	32	32	1	5	5	1	6
	1	0	14	14	6	5	5	6	16
	1	0	5	5	16	5	5	16	120
AlexNet	4	0	227	227	3	11	11	3	96
	1	2	27	27	96	5	5	96	256
	1	1	13	13	256	3	3	256	384
	1	1	13	13	384	3	3	384	384
	1	1	13	13	384	3	3	384	256
ResNet	2	3	224	224	3	7	7	3	64
	1	1	56	56	64	3	3	64	64
	2	1	56	56	64	3	3	64	128
	2	1	28	28	128	3	3	128	256
	2	1	14	14	256	3	3	256	512

green area, when an IXIAM-based interfacing is preferable to a CPU-only solution, but a driver-based one is not. From 4096 complex numbers on, a CPU-only solution is not convenient anymore in terms of performance. Summing up, our interfacing makes the acceleration of workloads in the range 16–1024 numbers convenient performance-wise.

In Figure 15, we show the speedup of a disparity map calculation offloaded to the DisparityMap accelerator. Again, it decreases monotonically in the three workloads considered: 38% for 192 × 144 images, 10% for 384 × 288 images, and 3% for 640 × 480 images. In this case, we only test the three workloads for which performance results are available in the original paper describing the accelerator [64]. Although the achieved speedups seem less than those achieved in the two previous cases, it is due to the higher computation times required even by the smallest of the workloads, which is two and three orders of magnitude higher than the smallest workload executed on the Crypto and MatMul accelerator, respectively, as it emerges from Tables 4, 5, and 6.

We do not show the comparison against a CPU-only implementation in Figure 15, as the whole chart would be in a light purple area. In fact, even the smallest workload is demanding for a CPU, and thus even the driver-based implementation achieves higher performance.

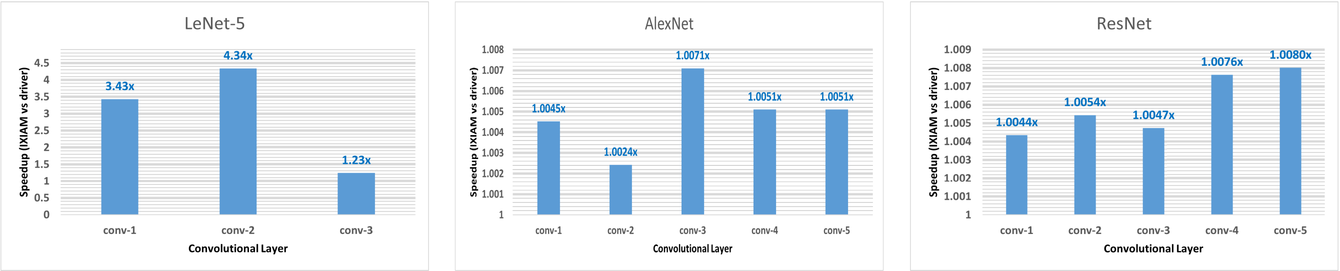
Figure 16 shows the speedup of a convolution calculation between two tensors (i.e., a three-dimensional input and a four-dimensional filter) performed on a CNN accelerator. The workloads are characterized by the dimensions of the tensors involved. These dimensions reflect those used in the convolutional layers of three popular CNNs: LeNet-5 [75], AlexNet [76], and ResNet [77]. The actual sizes of the tensors are displayed in Table 9.

The results on the convolutional layers are in continuity with those observed on other accelerators: the performance advantage of an IXIAM interfacing is higher when the workload is small. In particular, we achieve better results on LeNet-5 (at most, 4.34×), which is a smaller network with respect to AlexNet and ResNet. On these, the performance advantage is negligible (less than 1% on all the ten layers considered). Also in this case, even the smallest workloads are too onerous for a CPU-only implementation, and thus they would all be placed in the light purple area – i.e., with lower performance than both accelerated solutions.

Overall, the results shown so far on five different accelerators depict a similar scenario: the performance advantage introduced by an IXIAM interfacing is more evident for small and medium workloads than for bigger ones. By replacing drivers with CPU instructions backed by an ad-hoc hardware infrastructure, we allow faster communication between cores and accelerators, but do not intervene on the accelerator-side execution time. Its weight is higher for big workloads, and thus the performance advantage we introduce gets diluted accordingly, leading to smaller speedups that asymptotically tend to 1. Small and medium workloads, conversely, present very high speedups that are even close to two orders of magnitude, in some cases. Historically, these workloads have not been taken into consideration for acceleration because the time needed to communicate with the accelerators and move data dominates the total execution time, eliminating the possible performance advantage that an accelerated execution would have provided. With an IXIAM interfacing, the communication time is significantly reduced, to the point that the set of workloads eligible for acceleration is enlarged for each accelerator except CNN. In any case, CNN is no different: we suppose that there could be a network with a convolutional layer even smaller than those in LeNet-5 that would benefit from an IXIAM-based acceleration, but not a driver-based one. Another consequence of this is that, as accelerators improve and achieve higher and higher performance, our proposal becomes more valuable, as the weight of the accelerator-side execution reduces.

V. RELATED WORK

In this section, we discuss some related work in which the authors extend ISA for acceleration purposes.



(a) LeNet-5 (b) AlexNet (c) ResNet

FIGURE 16. CNN accelerator. Speedup of IXIAM vs a driver-based interface. A plain convolution between two tensors is executed. The tensor dimensions are those used in the convolutional layers of three popular CNNs: LeNet-5, AlexNet, and ResNet.

As we already specified in the Introduction, this work extends our previous proposal, presented in [21]. In this case, we define fine-grained transfer instructions that permit addressing local memories directly, and also transfer data from CPU registers to the accelerator, without the need to access the main memory. This enlarges the set of supported accelerators, as two instructions in the current proposal (TGL and TGS) can be easily interpreted as the TRANSFER instruction from the previous work on accelerators that work according to a fixed finite state machine – which were the target of the previous paper. Moreover, we design a user-space interrupt mechanism that may reduce the use cases for synchronous instructions.

The work that, most of all, is close to ours in intentions is a paper by Cong et al. [2], in which the authors aim at extending the ISA to control various on-chip hardware accelerators. They propose a central entity, which is called Global Accelerator Manager (GAM) to manage the communication between accelerators and processor cores. They propose 4 instructions to interact with the GAM and the accelerators, and 2 to implement a *lightweight* interrupt mechanism.

With respect to our work, there are some fundamental differences. The most important is the presence of the GAM, which acts as a central manager for all the accelerators and all the cores in the SoC – as such, it can be a bottleneck for the whole system. Conversely, we aim to augment the hardware support of the individual accelerators and avoid central modules. The GAM is designed to work in an "accelerator-rich" SoC, and thus supports a mechanism of *accelerator discovery* based on capabilities, while we address accelerators directly by id, which can be more appropriate for the current generation of SoCs.

Another fundamental difference is the fact that the core asking for accelerator availability needs to know *in advance* how long the accelerator should be reserved for it. We prefer to adopt a reserve-release mechanism with hardware (watchdog) or software (high-privilege reserve and release) support to manage violations. In [2], it is not clear whether there are system-wide policies to manage accelerator and processor states in case of wrong estimations.

In [2], the work is delegated to the accelerator by storing a *task description* in memory and passing the memory address to the accelerator. In this regard, we opt for a finer grain control of the accelerators' internal structures, like local memories, and also support small data transfers directly from/to host core registers to/from accelerator memory space. We admit the possibility of using low-latency, dedicated channels between accelerators and host cores (smaller latency than going through the system memory hierarchy), which is not mentioned [2].

As far as we know, [2] is the only work proposing an ISA extension to target generic accelerators. Other notable works propose ISA extensions as a way of letting the core communicate with their *particular* accelerator. In the following, we list some of the most recent solutions of this kind.

In [19], Nowatzki et al. propose Softbrain, a Coarse-Grain Reconfigurable Array (CGRA) accelerator for stream-dataflow applications. It can be integrated on an SoC, accessing the memory system either from the LLC or from the DRAM. They design an ad-hoc ISA to control Softbrain which includes 14 instructions, including instructions to read-/write from/to input and output ports or local memory, load the configuration from a given address, and some memory barriers.

PROMISE [18] is an in-SRAM mixed-signal accelerator for Machine Learning workloads. The authors identify the main operations in Machine Learning and propose a comprehensive ISA to target them. They propose 48-bit VLIW instructions that can specify up to 4 operations to be executed in sequence. There are 18 operations that include digital and analog reads/writes, *scalar distance* calculations, ADC, and *threshold* operations. They complete their proposal with a compiler that translates high-level Julia code into a PROMISE-oriented binary that leverages their ISA.

In [20], Jain et al. design an accelerator based on in-memory computing with Spin-Transfer Torque Magnetic RAM (STT-MRAM). In order to achieve integration with a general-purpose system, they propose architectural modifications to on-chip buses and an extension for the ISA. Their instructions act on pairs of address lines and address simple logical and arithmetic operations like XOR, NOT, AND, and ADD.

Fritzmann et al. propose *RISQ-V* to accelerate lattice post-quantum cryptography [15]. Since post-quantum cryptography requires mathematical elements and operations which are usually not easy to implement on standard processors, they propose an enhanced RISC-V architecture that integrates a set of powerful tightly coupled accelerators. They extend the RISC-V ISA with 29 new instructions to efficiently perform the needed operations.

In [78], Rao et al. design *IntersectX*, an accelerator for pattern enumeration with stream instruction set extension and architectural support. The proposed ISA extension is considered a natural extension to the traditional instructions that operate on scalar values. IntersectX architecture is composed of: a table to record the mapping between stream ID and stream register; a streaming cache that enables efficient stream data movements; a unit that implements sparse value computations; and the nested intersection translator that generates micro-operation sequences for implementing nested intersections. To employ all these steps in the proposed architecture for graph pattern mining by Intersect accelerator, 11 instructions in x86 are proposed as stream ISA extension.

Saarinen extend RISC-V ISA for *Advanced Encryption Standard* (AES) and SM4 block ciphers [79]. They propose sixteen instructions to implement an AES round, reducing the number of instructions down from 80. In their paper, RISC-V extensions are used to protect against cache timing side-channel attacks and eliminate slow, secret-dependent table lookups.

In [80], Amor et al. define a RISC-V ISA (RV32IM) extension for ultra-low power software-defined wireless IoT transceivers. The authors add 14 custom instructions tailored to the needs of 8/16/32-bit integer complex arithmetic typically required by quadrature modulations. The proposed extension instructions are designed to have a near-zero energy cost.

Arrow is a configurable hardware accelerator architecture that implements a subset of the RISC-V v0.9 vector ISA extension aimed at edge machine learning inference [81]. It is a kind of co-processor, which executes a suite of vector and matrix benchmarks fundamental to machine learning inference.

In [82], Albicocchi proposes a way to accelerate a post-quantum algorithm called Kyber and Dilithium. He chooses to embed lighter and more specific hardware accelerators directly in the core architecture to eliminate the overhead of communication and area. In order to use the hardware inside the core pipeline, he designs new RISC-V instructions to drive the accelerators. He extends the RISC-V ISA with three new instructions, which let the programmer use the hardware accelerators directly from the software layer.

In [83], Paulin et al. introduce RISC-V instruction extensions coupled with software optimizations for maximizing the throughput of a radio resource management that is critical in 5G mobile communications. Their radio resource management algorithm is based on models that use multi-layer perceptrons and recurrent neural networks. They extend the ISA by adding two new single-cycle HW instructions implemented as linear function approximations within certain intervals. They introduce another new instruction that combines the other two, which is capable of loading data and calculating the 16-bit packed SIMD sum-dot-product.

In [84], Manor et al. propose an ISA extension to support floating-point Coordinate Rotation Digital Computer (CORDIC), which is a dedicated low-power accelerator for mathematical functions used in Neural Network oriented non-linear equations. They propose a DMA-based ISA extension approach to perform repeated array calculations, offering speedup over software implementations.

All these works propose extensions deeply tailored to address a single accelerator. We think that such an approach has the impossibility of scaling as its biggest downside. The adoption of more of these solutions on one SoC would lead to ISA bloat, increasing power consumption and complexity of the decoding phase proportionally with the number of supported accelerators.

VI. CONCLUSION AND FUTURE WORK

In this paper, we presented IXIAM, a SoC-oriented HW-SW framework to control a wide variety of integrated accelerators directly from the CPU cores. It features a set of additional RISC-V instructions and a limited hardware infrastructure to support instruction execution, both on the core-side and accelerator-side. These instructions allow users to reserve/release accelerators, transfer data to/from their local memories, delegate execution, and query state. The instruction execution causes message exchange on the dedicated interconnect between cores and accelerators. We described also a user-space interrupt mechanism, which needs an ad-hoc module on the SoC to support it, to signal accelerator-side events to the cores without relying on the OS interrupt mechanism.

We analyzed some other aspects related to our proposal, showing that:

- memory coherency between cores and accelerators can be achieved in various ways, as IXIAM does not limit the choice;
- data consistency can be achieved by relying on a simple fence instruction on the core and adding another instruction (AFENCE) that constitutes a fence on the accelerator;
- a watchdog can be added to the system to protect against malevolent use and cleanup in case the accelerator's owning process dies without releasing;
- more sophisticated arbitration mechanisms can be achieved by elevating RESERVE and RELEASE privilege levels, implementing custom logic in ad-hoc system calls.

Then, we showed the performance achieved by an IXIAM-based interfacing with respect to a canonical driver-based one in the case of five different accelerators – i.e., MatMul, Crypto, DisparityMap, FFT, and CNN. We showed that IXIAM improves performance dramatically with respect to drivers in the case of small to medium workloads, thanks to the reduced interfacing costs. For those workloads, a CPU-only implementation is generally preferred to an accelerated one, as the communication dominates over computation. However, with an IXIAM interfacing, acceleration becomes a viable solution for many workloads, as one can have the advantages of acceleration (i.e., smaller computation time) without the disadvantages of a driver-based interfacing (i.e., high communication latencies). For bigger workloads, we showed that our proposal does not hamper performance, offering speedups that tend to 1× asymptotically.

As future work, we plan to investigate NoC design in order to better determine the topology and the characteristics that better suit IXIAM. Moreover, we plan to investigate more sophisticated mechanisms to achieve data consistency, like those listed in Subsection III-B.

Finally, we plan to extend the span of this work addressing also the additional requirements of *programmable* accelerators, which can execute binary *kernels* sent from the cores, like in GPUs.

ACKNOWLEDGMENT

The authors would like to express their deepest gratitude to Jonathan Cameron and Anthony Jebson for their fundamental teachings on the state of the art of Linux driver development.

REFERENCES

[1] W.-M. Hwu and S. Patel, "Accelerator architectures—A ten-year retrospective," *IEEE Micro*, vol. 38, no. 6, pp. 56–62, Nov. 2018.

[2] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture support for accelerator-rich CMPs," in *Proc. 49th Annu. Design Autom. Conf.*, Jun. 2012, pp. 843–849.

[3] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, pp. 48–60, Jan. 2019.

[4] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Commun. ACM*, vol. 63, no. 7, pp. 48–57, Jun. 2020.

[5] B. Peccerillo, M. Mannino, A. Mondelli, and S. Bartolini, "A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives," *J. Syst. Archit.*, vol. 129, Aug. 2022, Art. no. 102561. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762122001138

[6] ARM. *AMBA Overview*. Accessed: Feb. 8, 2023. [Online]. Available: https://developer.arm.com/architectures/system-architectures/amba

[7] Apple. (2020). *M1*. [Online]. Available: https://www.apple.com/uk/mac/m1/

[8] L. Codrescu. (2013). *Qualcomm Hexagon DSP: An Architecture Optimized for Mobile and Multimedia Communications*. [Online]. Available: https://developer.qualcomm.com/qfile/27696/qualcomm-hexagon-architecture.pdf

[9] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "GraphP: Reducing communication for PIM-based graph processing with efficient data partition," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2018, pp. 544–557.

[10] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel paterns," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Jun. 2017, pp. 389–402.

[11] E. G. Cota, P. Mantovani, G. D. Guglielmo, and L. P. Carloni, "An analysis of accelerator coupling in heterogeneous architectures," in *Proc. 52nd Annu. Design Autom. Conf.*, Jun. 2015, pp. 1–6.

[12] P. Vogel, A. Kurth, J. Weinbuch, A. Marongiu, and L. Benini, "Efficient virtual memory sharing via on-accelerator page table walking in heterogeneous embedded SoCs," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, pp. 1–19, Sep. 2017.

[13] Y. Hao, Z. Fang, G. Reinman, and J. Cong, "Supporting address translation for accelerator-centric architectures," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 37–48.

[14] Y.-K. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A quantitative analysis on microarchitectures of modern CPU-FPGA platforms," in *Proc. 53rd Annu. Design Autom. Conf.*, Jun. 2016, pp. 1–6.

[15] T. Fritzmann, G. Sigl, and J. Sepúlveda, "RISQ-V: Tightly coupled RISC-V accelerators for post-quantum cryptography," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2020, pp. 239–280, Aug. 2020.

[16] G. H. Eisenkraemer, F. G. Moraes, L. L. de Oliveira, and E. Carara, "Lightweight cryptographic instruction set extension on xtensa processor," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Oct. 2020, pp. 1–5.

[17] D. W. Todd, "Tightly coupling the PicoRV32 RISC-V processor with custom logic accelerators via a generic interface," Ph.D. dissertation, Dept. Comput. Eng., Clemson Univ., Clemson, SC, USA, 2021.

[18] P. Srivastava, M. Kang, S. K. Gonugondla, S. Lim, J. Choi, V. Adve, N. S. Kim, and N. Shanbhag, "PROMISE: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 43–56.

[19] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Jun. 2017, pp. 416–429.

[20] S. Jain, A. Ranjan, K. Roy, and A. Raghunathan, "Computing in memory with spin-transfer torque magnetic ram," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 3, pp. 470–483, Dec. 2017.

[21] E. Cheshmikhani, B. Peccerillo, A. Mondelli, and S. Bartolini, "A general framework for accelerator management based on ISA extension," *IEEE Access*, vol. 10, pp. 120702–120713, 2022.

[22] J. Lowe-Power, "The gem5 simulator: Version 20.0+," 2020, *arXiv:2007.03152*.

[23] Y. S. Shao, S. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Toward cache-friendly hardware accelerators," in *Proc. HPCA Sensors Cloud Archit. Workshop (SCAW)*, 2015, pp. 1–6.

[24] J. Zuckerman, D. Giri, J. Kwon, P. Mantovani, and L. P. Carloni, "Cohmeleon: Learning-based orchestration of accelerator coherence in heterogeneous SoCs," in *Proc. MICRO 54th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2021, pp. 350–365.

[25] P. Shantharama, A. S. Thyagaturu, and M. Reisslein, "Hardware-accelerated platforms and infrastructures for network functions: A survey of enabling technologies and research studies," *IEEE Access*, vol. 8, pp. 132021–132085, 2020.

[26] V. Gupta, P. Brett, D. Koufaty, D. Reddy, S. Hahn, K. Schwan, and G. Srinivasa, "The forgotten 'Uncore': On the energy-efficiency of heterogeneous cores," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2012, pp. 367–372.

[27] A. Waterman, K. Asanović, and J. Hauser, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, document Version 20211203, RISC-V International, 2015.

[28] NVIDIA. (2020). *NVIDIA Ampere GA102 GPU Architecture*. [Online]. Available: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf

[29] NVIDIA. (Nov. 2019). *CUDA C Programming Guide*. [Online]. Available: docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[30] D. Giri, P. Mantovani, and L. P. Carloni, "Accelerators and coherence: An SoC perspective," *IEEE Micro*, vol. 38, no. 6, pp. 36–45, Nov. 2018.

[31] M. D. Sinclair, J. Alsop, and S. V. Adve, "Efficient GPU synchronization without scopes: Saying no to complex consistency models," in *Proc. 48th Int. Symp. Microarchitecture*, Dec. 2015, pp. 647–659.

[32] R. Cavicchioli, N. Capodieci, and M. Bertogna, "Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms," in *Proc. 22nd IEEE Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2017, pp. 1–10.

[33] L. Chai, Q. Gao, and D. K. Panda, "Understanding the impact of multi-core architecture in cluster computing: A case study with Intel dual-core system," in *Proc. 7th IEEE Int. Symp. Cluster Comput. Grid (CCGrid)*, May 2007, pp. 471–478.

[34] D. Bovet and M. Cesati, *Understanding the Linux Kernel*. Sebastopol, CA, USA: O'Reilly & Associates Inc, 2005.

[35] A. Asgharzadeh, J. M. Cebrian, A. Perais, S. Kaxiras, and A. Ros, "Free atomics: Hardware atomic operations without fences," in *Proc. 49th Annu. Int. Symp. Comput. Archit.*, Jun. 2022, pp. 1–13.

[36] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," *ACM SIGPLAN Notices*, vol. 33, no. 11, pp. 58–69, Nov. 1998.

[37] Intel Corporation. (2011). *Intel 64 and IA-32 Architectures Software Developer's Manual—Volume 3B*. Accessed: Feb. 8, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[38] B. Rajaram, V. Nagarajan, S. Sarkar, and M. Elver, "Fast RMWs for TSO: Semantics and implementation," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2013, pp. 61–72.

[39] M. M. Michael and M. L. Scott, "Implementation of atomic primitives on distributed shared memory multiprocessors," in *Proc. 1st IEEE Symp. High Perform. Comput. Archit.*, Jan. 1995, pp. 222–231.

[40] E. H. Jensen, G. W. Hagensen, and J. M. Broughton, "A new approach to exclusive data access in shared memory multiprocessors," Lawrence Livermore Nat. Lab., Livermore, CA, USA, Tech. Rep. UCRL-97663, 1987.

[41] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo, "Global real-time memory-centric scheduling for multicore systems," *IEEE Trans. Comput.*, vol. 65, no. 9, pp. 2739–2751, Sep. 2016.

[42] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for COTS-based embedded systems," in *Proc. 17th IEEE Real-Time Embedded Technol. Appl. Symp.*, Apr. 2011, pp. 269–279.

[43] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *Proc. 24th Euromicro Conf. Real-Time Syst.*, Jul. 2012, pp. 299–308.

[44] H. Yun, S. Gondi, and S. Biswas, "Protecting memory-performance critical sections in soft real-time applications," 2015, *arXiv:1502.02287*.

[45] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Proc. IEEE 19th Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2013, pp. 55–64.

[46] S. V. Adve and M. D. Hill, "Weak ordering—A new definition," *ACM SIGARCH Comput. Archit. News*, vol. 18, no. 2SI, pp. 2–14, 1990.

[47] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-race-free memory models," in *Proc. 19th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Feb. 2014, pp. 427–440.

[48] L. Howes and A. Munshi, *The OpenCL Specification, Version 2.0*. Beaverton, OR, USAKhronos Group, 2015.

[49] T. Sorensen, G. Gopalakrishnan, and V. Grover, "Towards shared memory consistency models for GPUs," in *Proc. 27th Int. ACM Conf. Int. Conf. Supercomputing*, Jun. 2013, pp. 489–490.

[50] L. Li and C. Kessler, "VectorPU: A generic and efficient data-container and component model for transparent data transfer on GPU-based heterogeneous systems," in *Proc. 8th Workshop 6th Workshop Parallel Program. Run-Time Manage. Techn. Many-Core Archit. Design Tools Archit. Multicore Embedded Comput. Platforms*, Jan. 2017, pp. 7–12.

[51] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-M.-W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *Proc. 15th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2010, pp. 347–358.

[52] L. Henrio, C. Kessler, and L. Li, "Ensuring memory consistency in heterogeneous systems based on access mode declarations," in *Proc. Int. Conf. High Perform. Comput. Simul. (HPCS)*, Jul. 2018, pp. 716–723.

[53] B. Peccerillo and S. Bartolini, "PHAST—A portable high-level modern C++ programming library for GPUs and multi-cores," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 1, pp. 174–189, Jan. 2019.

[54] S.-J. Min and R. Eigenmann, "Optimizing irregular shared-memory applications for clusters," in *Proc. 22nd Annu. Int. Conf. Supercomputing*, Jun. 2008, pp. 256–265.

[55] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU communication management and optimization," in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2011, pp. 142–151.

[56] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, "Dynamically managed data for CPU-GPU architectures," in *Proc. 10th Int. Symp. Code Gener. Optim.*, Mar. 2012, pp. 165–174.

[57] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA tensor core programmability, performance precision," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2018, pp. 522–531.

[58] TechPowerUp. (2018). *NVIDIA TU102*. [Online]. Available: https://www.techpowerup.com/gpu-specs/nvidia-tu102.g813#gallery-6

[59] VideoCardz. (2018). *NVIDIA TU102 Graphics Processing Unit (GPU)*. [Online]. Available: https://videocardz.net/gpu/nvidia-tu102

[60] Andrei Frumusanu. (Nov. 2019), *The Huawei Mate 30 Pro Review: Top Hardware Without Google*. [Online]. Available: https://www.anandtech.com/show/15099/the-huawei-mate-30-pro-review-top-hardware-without-google

[61] S. J. E. Wilton and N. P. Jouppi, "CACTI: An enhanced cache access and cycle time model," *IEEE J. Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.

[62] M. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. Bassham, E. Roback, and J. Dray, "Advanced encryption standard (AES)," Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. 197, Nov. 2001.

[63] Xilinx. (Apr. 2022). *Advanced Encryption Standard (AES) Engine v1.1*. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/ip_documentation/aes/v1_1/pg383-aes.pdf

[64] M. A. Ibarra-Manzano, D.-L. Almanza-Ojeda, M. Devy, J.-L. Boizard, and J.-Y. Fourniols, "Stereo vision algorithm implementation in FPGA using census transform for effective resource optimization," in *Proc. 12th Euromicro Conf. Digit. Syst. Design, Archit., Methods Tools*, Aug. 2009, pp. 799–805.

[65] R. Zabih and J. Woodfill, "Non-parametric local transforms for computing visual correspondence," in *Proc. Eur. Conf. Comput. Vis.* Cham, Switzerland: Springer, 1994, pp. 151–158.

[66] X. Chen, Y. Lei, Z. Lu, and S. Chen, "A variable-size FFT hardware accelerator based on matrix transposition," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 10, pp. 1953–1966, Oct. 2018.

[67] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965.

[68] P. K. Meher, J. Valls, T.-B. Juang, K. Sridharan, and K. Maharatna, "50 years of CORDIC: Algorithms, architectures, and applications," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 56, no. 9, pp. 1893–1907, Sep. 2009.

[69] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, May 2016.

[70] Kernel Development Community. (2022). *Introduction of Uacce*. [Online]. Available: https://docs.kernel.org/misc-devices/uacce.html

[71] Shuveb Hussain. (2020). *Welcome to Lord of the IO_Uring*. [Online]. Available: https://unixism.net/loti/

[72] The OpenSSL Project. (1999). *OpenSSL Cryptography and SSL/TLS Toolkit*. [Online]. Available: https://www.openssl.org/

[73] The OpenSSL Project. (2022). *OpenSSL Cryptography and SSL/TLS Toolkit GitHub Repository*. [Online]. Available: https://github.com/openssl/openssl

[74] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. IEEE*, vol. 93, no. 2, pp. 216–231, Feb. 2005.

[75] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[76] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017.

[77] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[78] G. Rao, J. Chen, J. Yik, and X. Qian, "IntersectX: An efficient accelerator for graph mining," 2020, *arXiv:2012.10848*.

[79] M.-J. O. Saarinen, "A lightweight ISA extension for AES and SM4," 2020, *arXiv:2002.07041*.

[80] H. B. Amor, C. Bernier, and Z. Prikryl, "A RISC-V ISA extension for ultra-low power IoT wireless signal processing," *IEEE Trans. Comput.*, vol. 71, no. 4, pp. 766–778, Apr. 2022.

[81] I. Al Assir, M. El Iskandarani, H. R. Al Sandid, and M. A. R. Saghir, "Arrow: A RISC-V vector accelerator for machine learning inference," 2021, *arXiv:2107.07169*.

[82] F. Albicocchi, "A RISC-V ISA extension for speeding-up post quantum crystals algorithms through HW accelerators integrated in the ariane core pipeline," M.S. thesis, Dept. Electron. Eng., Dept. Inf. Eng., Univ. Pisa, Pisa, Italy, Apr. 2021.

[83] G. Paulin, R. Andri, F. Conti, and L. Benini, "RNN-based radio resource management on multicore RISC-V accelerator architectures," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 29, no. 9, pp. 1624–1637, Sep. 2021.

[84] E. Manor, A. Ben-David, and S. Greenberg, "CORDIC hardware acceleration using DMA-based ISA extension," *J. Low Power Electron. Appl.*, vol. 12, no. 1, p. 4, Jan. 2022.

BIAGIO PECCERILLO is currently a Postdoctoral Researcher with the Department of Information Engineering and Mathematical Sciences, University of Siena. He participated in various research and development projects involving high-productivity solutions to program heterogeneous architectures, hardware accelerators, haptic algorithms in virtual and augmented reality environments, and pharmaceutical supply chain simulation. His research interests include heterogeneous architectures, productivity-oriented high-level abstraction mechanisms, hardware accelerators, and parallel algorithms.

ELHAM CHESHMIKHANI received the B.Sc. degree in computer engineering from the Iran University of Science and Technology (IUST), in 2011, the M.Sc. degree in computer engineering from the Amirkabir University of Technology (Tehran Polytechnic), in 2013, and the Ph.D. degree in computer engineering from the Sharif University of Technology (SUT), Tehran, Iran, in 2020. She was a member of the Design and Analysis of Dependable Systems (DADS), AUT, from 2011 to 2015, the Dependable Systems Laboratory (DSL), SUT, from 2016 to 2018, and the Data Storage, Networks and Processing Laboratory (DSN), SUT, from 2018 to 2021. From 2021 to 2022, she was a Postdoctoral Researcher with the Department of Information Engineering and Mathematics, University of Siena, Siena, Italy. Since 2022, she has been with Tehran Polytechnic. She is currently an Assistant Professor with the Department of Computer Engineering, Tehran Polytechnic. Her research interests include hardware accelerators, SoC design, RISC-V ISA design, emerging nonvolatile memory technologies, processing-in-memory, dependable systems design, and storage systems. During her Ph.D. career, she received the Best Paper Award from IEEE/ACM Design, Automation, and Test in Europe (DATE), in 2019.

MIRCO MANNINO received the B.Sc. and M.Sc. degrees in computer engineering from the University of Siena, where he is currently pursuing the Ph.D. degree with the Department of Information Engineering and Mathematical Sciences. His research interests include the optimization of deep learning algorithms, parallel algorithms, high-performance computing, and hardware accelerators.

ANDREA MONDELLI received the Ph.D. degree in computer architecture. He was a Researcher and an Architect in various countries, such as Italy, the USA, France, China, and the U.K. He is currently the CPU Chief Architect with Huawei and a Principal Researcher of cybersecurity and architecture design. He is also a Technology Manager and is responsible for Huawei research projects and collaborations with European universities. He published multiple manuscripts and conference papers and a book on memory coherence protocols. His research interests include high-performance and low-power CPUs. He was part of RISC-V International as the Chair of the virtual memory area.

SANDRO BARTOLINI is currently an Associate Professor with the Department of Information Engineering and Mathematical Sciences, University of Siena. He has led and participated in various research and development projects. His main research interests include high-performance chip multiprocessors (CMPs), new approaches to productive programming of heterogeneous architectures (CPUs and GPUs), integrated photonics for CMPs, feedback-driven compiler optimizations for cache hierarchy performance and low power, and hardware accelerators. He is an active member of the HiPEAC NoE. He is an Associate Editor of *EURASIP Journal of Embedded Computing*. He has been a Co-Guest Editor of *Transactions on High-Performance Architectures and Compilation* (Springer) and *ACM SigArch Computer Architecture Newsletter*.

Open Access funding provided by 'Università degli Studi di Siena' within the CRUI CARE Agreement