International Journal of Parallel Programming (2021) 49:237–252
https://doi.org/10.1007/s10766-020-00674-y

Enabling Near-Data Accelerators Adoption by Through Investigation of Datapath Solutions

Paulo C. Santos[1] · João P. C. de Lima[1] · Rafael F. de Moura[1] · Marco A. Z. Alves[2] · Antonio C. S. Beck[1] · Luigi Carro[1]

Received: 29 October 2019 / Accepted: 15 July 2020 / Published online: 28 January 2021

Abstract

Processing-in-Memory (PIM) or Near-Data Accelerator (NDA) has been recently revisited to mitigate the issues of memory and power wall, mainly supported by the maturity of 3D-staking manufacturing technology, and the increasing demand for bandwidth and parallel data access in emerging processing-hungry applications. However, as these designs are naturally decoupled from main processors, at least three open issues must be tackled to allow the adoption of PIM: how to offload instructions from the host to NDAs, since many can be placed along memory; how to keep cache coherence between host and NDAs, and how to deal with the internal communication between different NDA units considering that NDAs can communicate to each other to better exploit their adoptions. In this work, we present an efficient design to solve these challenges. Based on the hybrid Host-Accelerator code, to provide fine-grain control, our design allows transparent offloading of NDA instructions directly from a host processor. Moreover, our design proposes a data coherence protocol, which includes an inclusion-policy agnostic cache coherence mechanism to share data between the host processor and the NDA units, transparently, and a protocol to allow communication between different NDA units. The proposed mechanism allows full exploitation of the experimented state-of-the-art design, achieving a speedup of up to 14.6× compared to a AVX architecture on PolyBench Suite, using, on average, 82% of the total time for processing and only 18% for the cache coherence and communication protocols.

Keywords Processing-in-memory · Nead-data processing · Datapath · More

This study was financed in part by the CAPES (Finance Code 001), CNPq, FAPERGS, and Serrapilheira (Serra-1709-16621).

✉ Paulo C. Santos
pcssjunior@inf.ufrgs.br

Extended author information available on the last page of the article

1 Introduction

In the last years, mainly due to the arising of big data workloads, machine learning and data-intensive algorithms, modern applications have demanded different flavors of accelerators. Meanwhile, supported by 3D-stacking technologies, 3D-stacked memories have emerged enabling improvements on memory bandwidth, energy efficiency, and chip capacity, aiming to mitigate the memory wall effects widely increased by the numerous accelerators present on modern embedded systems. In this context, Near-Data Accelerator (NDA) or Processing-in-Memory (PIM) studies have been leveraged by both the requirements of data-intensive workloads and the emergence of 3D-stacked memories. Moreover, supported by the integration of logical and memory layers, such as found in the Hybrid Memory Cube (HMC) [14] and High Bandwidth Memory (HBM) [28], more sophisticated and efficient PIM designs have emerged.

PIM techniques intend to mitigate the memory bottleneck by processing data near to, or directly on main memory, which mainly avoids data movements via off-chip buses and trough complex cache hierarchies, hence avoiding unnecessary cache pollution, reducing energy consumption and improving performance. Ideally, an efficient PIM architecture must be able to take advantage of the internal bandwidth available on 3D-stacked memories, which can provide at least 320 GB/s [14, 28]. Moreover, the logic design must fit within the logic layer area and power constraints of 3D-stacked memories [10, 21]. Past studies, such as [16, 26, 27], have analyzed how different PIM logic designs exploit the huge bandwidth provided by 3D-stacked memories, as well as how much area and power they require.

From those studies, it is depicted that the most suitable PIM logic designs require a massive amount of simple Functional Units (FUs) to efficiently benefit from the available bandwidth. Consequently, to achieve high performance (TFLOPs), while still matching power and area budgets [10, 16], the PIM design needs to move from typical processor's front-end and sophisticated Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP) hardware techniques, leaving room for FUs and register files. Therefore, the adoption of FU-centric and reduced logic approaches relies on fine-grain instruction offloading [2].

Although several works present solutions for the instruction-level offloading issue [4, 8, 15, 17], they lack solving three inherent challenges that comes with this approach: how data is kept coherent between host and PIM and between multiple computing units within 3D-stacked memory, and how to allow communication between many PIM units that are necessary to take advantage of the available resources [16]. The solutions present in the literature demand specialized extra hardware while they limit the generality of cache memories [4], isolate memory address between different processing elements and restrict the parallelism exploitation of 3D-stacked memories [12], and do not support Translation Look-aside Buffer (TLB) and virtual memory [8]. Moreover, the communication between different PIM units is neglected in previous studies available in the literature, which prevents the generality of applications, high-performance exploitation, and efficient utilization of available resources.

The present work aims to provide resources for the adoption of FU-centric PIM architectures, such as those presented in [16, 19, 24, 25]. Therefore, this work focuses on solving the aforementioned main issues to allow a transparent offloading of PIM instructions directly from the host processor without incurring performance overhead to the Central Processing Unit (CPU). Furthermore, our PIM architecture design proposes a data coherence protocol that includes not only a mechanism to share data between the host processor and PIM units transparently, but also among several in-memory functional units-based processors. Thus, our design addresses programmability and cache coherence issues to reduce programmers' effort in designing applications to be executed in PIM architectures. Moreover, a complete explanation of how our mechanism handles both the Host-PIM and PIM–PIM communication is shown. We compare the present approach with the optimal oracle-based case, showing that our PIM approaches to solving instruction offloading, data coherence, and communication can get close to the optimal scenario.

This work is organized as follows: Sect. 2 highlights previous and state-of-the-art PIMs approaches while summarizes their mechanisms to support the adoption of each design. Section 3 gives a brief overview of the case study. Our approach is present in Sect. 4. The results are present on Sect. 5, and the conclusions on Sect. 6.

2 State-of-the-Art Solutions

Although allowing interference in CPU's designs may minimize the effort to programmers, i.e. by allowing Instruction Set Architecture (ISA) extensions in traditional processors [2, 25], it opens up new challenges, such as deciding between host and PIM instructions at compiling time or at running time, how to keep coherence among PIM devices and host processor, and how to manage communication between different PIM units while keeping data coherence among them.

2.1 Offloading PIM Instructions

Two main ways of performing code offloading are highlighted in the literature: fine-grain offloading and coarse-grain offloading. In the former way, PIM instructions are seen as individual operations and issued one by one to fixed-function PIM logic from CPU [4, 6, 15, 17, 19, 25]. In the coarse-grain instruction offloading approach, an application can be seen as having an entire or partial PIM instruction kernel as presented in [5, 12, 24, 29]. Coarse-grain approaches often have portions of code that should execute as PIM instructions surrounded with macros (like PIM-begin and PIM-end as seen in [8, 12]). From the CPU side, when it fetches a PIM instruction, it sends the instruction's Program Counter (PC) to a free PIM core, and the assigned core begins to execute starting from this given PC. Later, when the PIM unit finishes its execution, the CPU is notified about its completion [3, 4, 8, 12]. Thus, these ways of performing PIM instruction offloading provide the illusion that PIM operations are executed as if they were host processor instructions [4]. Considering that PIM instructions also perform load and store operations, these instructions require some mechanism to perform address translation. There are three common ways to treat this in state-of-art PIM architectures. The first one is to keep the same virtual address mapping scheme used by the CPU and Operating System (OS) [4]. In this way, some works may also adopt traditional multi-threading techniques (MPI, OpenMP, CUDA) [18, 23, 29]. However, this approach demands full-core implementation, which can present a critical overhead in terms of area and power [16]. Another approach is to have split addressing spaces for each PIM unit [12], although it demands each PIM instance to have its virtual address mapping components. The last way is to utilize only physical addresses on PIM instructions [8], but it has some critical drawbacks such as memory protection, software compatibility, and address mapping management schemes.

2.2 Keeping Coherence

After the offloading handler addresses a given PIM instruction, it may have to perform load/store operations and consequently have memory addresses shared along with other PIM instances or even CPUs. To cope with this data coherence problem, some designs opt for not offer a solution in hardware, requiring the programmer to explicit manage coherence or mark PIM data as non-cacheable [3–5, 13]. In other approaches [8], the coherence is kept within the first data cache level of each PIM core taking use of a *MESI* [20] protocol directory inside the Dynamic Random Access Memory (DRAM) logic layer. In this solution, coherence stats are updated only after the PIM kernel's execution: PIM cores send a message to the main processor informing it all the accessed data addresses. The main memory directory is checked, and if there is a conflict, the PIM kernel rolls back its changes, all cache lines used by the kernel are written back into the main memory, and the PIM device restarts its execution. Other methodologies use protocols based on single-block-cache restriction policy [4], which utilizes last level cache tags. To guarantee coherence, special PIM memory fence instructions (*pfence*) must surround shared memory regions code. On [4], to allow PEI (PIM Enable Instructions) operations, a special module called PEI Management Unit (PMU) maps the read and written addresses by all PIM elements using a read-write lock mechanism and monitors the cache blocks accesses issuing requests for back-invalidation (for writing PEIs) or back-writeback (for reading PEIs). Although this design modifies the last level cache and limits its access, the advantage of this design is that PEI instructions access the Translation Look-aside Buffer (TLB) as normal load/store instructions. Alternatively, some PIM designs put caches, TLB and Memory Management Unit (MMU) within each memory vault to perform addressing translations [12]. In this case, cache coherence is maintained by a three-step protocol: The Streaming Machine (SM) that requested the instruction offloading pushes all memory update traffic from itself to memory before it sends the offloading request. Second, the memory stack SM invalidates its private data cache. Third, memory stack SM sends all its modified data cache lines to the SM Graphic Processing Unit (GPU) that subsequently gets the latest version of data from memory.

2.3 Managing Communication

Each PIM exposes an interface to the host CPU and, due to the absence of a standard model or even a protocol for this interface, most current works in the field of PIM research adopt their models to implement and handle CPU-PIM and PIM-PIM communication. Since PIM units are commonly seen by host CPU as co-processors, Host-PIM communication is treated in the literature by taking use the memory channel to pass instructions/commands from CPU to PIM units [4, 8, 12]. Some PIM approaches do not have communication among units. In these cases, there is not PIM-PIM communication because they either have separated memory chips and do not perform computation over external memory addresses [12], or they have fixed address ranges without shared memory locations [17]. In other works, where there is PIM-PIM communication, it is managed by MESI-based modified protocols in a similar way which Multi-Processor System-on-Chip (MPSoC) do [4]. Similarly, by adopting traditional protocols (e.g. OpenMP), it is possible to share memory, and therefore communicate through it. However, a performance penalty must be considered since the shared memory, in this case, is the main memory (typically DRAM).

3 A NDA Architecture for a Case Study

State-of-the-art NDA designs have been presented in the literature. The work presented in [1] allows cache memory to compute binary logic operations, which requires the offloading of instruction from the processor-host side, and also data coherence treatments as the proposed mechanism can be implemented in any cache level. The authors of [17] propose the use of the native PIM presented in the HMC [14] to accelerate graph algorithms. The HMC is the first 3D-stacked memory to specify atomic commands to perform read-update-write in-memory operations on data using 16 byte operands. However, the HMC design does not consider any solution for the instruction offloading and cache coherence issues. Similarly, [19, 24, 25] show PIM designs to exploit data-level parallelism by providing vector processing units in memory, also relying on instruction offloading. Additionally, the PIM presented in [19, 24, 25] share memory space with host, which require data coherence mechanisms.

Listing 1: Hybrid Code Host-PIM Example

```
mov    r10, rdx
xor    ecx, ecx
PIM_256B_LOAD_DWORD PIM_3_R256B_1, pimword ptr [rsp + 1024]
PIM_256B_VPERM_DWORD PIM_3_R256B_1, PIM_3_R256B_1, PIM_3_R256B_0
PIM_256B_VADD_DWORD PIM_3_R256B_0, PIM_3_R256B_0, PIM_3_R256B_1
PIM_256B_STORE_DWORD pimword ptr [rsp + 1536], PIM_3_R256B_0
mov     eax, dword ptr [rsi + 4*rcx + 16640]
imul    eax, r9d
add     eax, dword ptr [rsp + 1536]
mov     dword ptr [r10], eax
inc     rcx
```

In a sequence of proposals, this work focuses on a design that provides high compute bandwidth by augmenting existing CPU data-path with FUs placed in the logic layer of the HMC. As described in [16], the micro-architecture called Reconfigurable Vector Unit (RVU) [25] meets power and area constraints given by HMC while providing the maximum processing bandwidth when compared to other recent 3D-stacked PIM proposals [3, 9, 11, 26]. The RVU architecture consist of FU sets distributed along the 32 vaults within an HMC module, each vault containing a register bank with 8x256 bytes, 32 multi-precision floating-point/integer FU and a Finite State Machine (FSM). The RVU ISA is a subset of Intel's Advanced Vector Extensions (AVX) that contains arithmetic, logic, data transfer, and *in-vector* data reordering operations. As described in [16], most of the computing unit area is occupied by multi-precision floating point FUs which make possible to achieve a peak compute power of 8 TFLOPS.

The RVU ISA is based on operand size reconfiguration to create instructions with variable vector width within a RVU instance, that is operands varying from 4 Bytes to 256 Bytes. However, it is also capable of inferring large vectors by aggregating neighbor instances and create instructions ranging from 256 Bytes to 8 kBytes [19, 24, 25]. Each RVU instance can request a region of memory from another memory *vault* or even request an entire register from a distinct instance using the internal logic-layer communication path. Hence accessing all memory space without restrictions. Moreover, as RVU augments the CPU data-path, native host instructions and PIM instructions are expected to happen in the same binary and use the same address space. Listing 1 illustrates this behavior, where it is possible to notice that both host (x86) and PIM (RVU) instructions are explicitly accommodated in the same code. In this example, the assembly code shown highlights RVU code using the *PIM* prefix, followed by vector operation size (i.e., 256 Bytes), operation (e.g., LOAD, VADD), element-operand size (e.g. BYTE, WORD, DWORD, QWORD), and finally the RVU internal registers. This type of code generation can be automated, and the compiler must be aware of the PIM architecture, as presented in [2].

4 Proposed Mechanism

The goal of this paper is to provide an efficient implementation of host-Near-Data Accelerator (NDA) interface, thus allowing the transparent utilization of 3D-stacked memory bandwidth with the lowest possible performance overhead. Figure 1 illustrates the proposed datapath to overcome the challenges concerning instruction-level offloading, data coherence, and the intercommunication model used inside the 3D-stacked device, and the next sections will detail its purpose.

4.1 Instruction Offloading

The instruction-level or fine-grain offloading is convenient for FU-centric accelerators, since the application execution flow can remain in the host CPU by only including a dedicated ISA for accelerator's operations. Here, we consider an arbitrary ISA-extension for triggering NDA operations, and also for allowing binaries to be composed of both host CPU and NDA instructions. Hence, the host CPU has to fetch, decode and issue instructions transparently to the near-memory logic without or with minimal timing overhead.

In our modeling, we built the case study ISA upon the x86 ISA, and we use a two-step decoding mechanism to perform fine-grain instruction offloading to NDA: host CPU and NDA sides. The modules present in any host CPU, such as TLB, page walker, and even host registers can be reused by CPU and NDA instructions without incurring any limitation or additional hardware. For instance, memory access instructions, such as *PIM_LOAD* and *PIM_STORE*, rely on the host address translation mechanism (from Address Generation Unit (AGU) to TLB), which prevents hardware duplication in NDA logic, keeping software compatibility and memory protection. Thus, the host-side interface seamlessly supports register-to-register and register-to-memory instructions in the near-memory logic, and also register-to-register instructions between host CPU and NDA logic.

The first step to perform instruction offloading consists of decoding an NDA instruction in the host CPU. Part of the instruction fields in the NDA ISA can be used to read from or write to host registers, and to calculate the memory addresses using any host addressing method. In the meantime, other instruction fields are used to NDA-specific features, such as physical registers of *NDA* logic, operand size, vector width and so on, which are encapsulated into the *NDA* machine format in the execution stage.

In the host CPU pipeline, all NDA instructions are seen as store instructions, which are issued to the Load-Store Queue (LSQ) unit. This characteristic allows each NDA to be addressed at compile time (memory mapped), and therefore properly dispatched to the correct NDA within a *vault* controller. The *NDA Instruction Dispatcher* unit illustrated in Fig. 1 represents the modifications made in the LSQ to support the instruction-level offloading. As NDA instructions are sent as regular packets to the memory system, they are addressed by the destination NDA unit and an architecture-specific flag is set in the packet to differ it from typical *read*

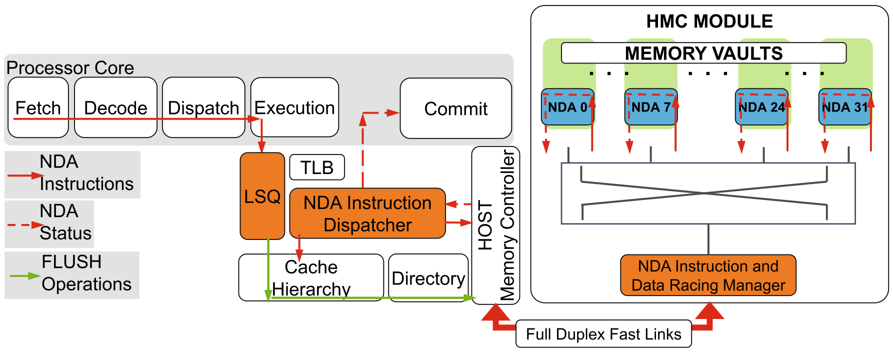

Fig. 1 Overview of the proposed datapath for efficient utilization of NDA logic. The orange boxes represent the additional modules required by the proposed mechanisms. The NDA units distributed along memory *vaults* is represented by the blue boxes. (Colour figure online)

and *write* requests. When the packet arrives at the *vault* controller, the second step of the offloading mechanism takes place on the memory side: the instruction fields are extracted from the packet and decoded by the NDA FSM, where data will be finally be transferred or modified. By using this methodology, we decouple NDA logic from the host interface and, thus, the NDA logic can be easily extended without implying in modifications either to the host CPU or the host-NDA interface.

Further, the *NDA Instruction Dispatcher* unit is also responsible for violation checking between native and NDA load/store requests. An exclusive offloading port connects the LSQ to the host memory controller directly. The NDA instructions are dispatched in a pipeline fashion at each CPU cycle, except for the memory instructions that need its data to be updated in the main memory and invalidated in the cache memories to keep data coherent, which is detailed in Subsection 4.2.

4.2 Cache Coherence Support for NDA

Cache coherence in near-data architectures must not only keep shared data between cache hierarchy and processors, but also between cache memories and main memory with processing logic, since both NDA and host instructions may have access to a shared memory, which is typically a DRAM. However, traditional coherence mechanisms (e.g., Modified, Owned, Exclusive, Shared, Invalid (MOESI) protocol), may not be sufficient to keep coherence in NDA because such protocols will require intense traffic of snooping messages in a narrower communication channel, which may be a source of bandwidth and time overhead. To minimize this overhead, and to maintain coherence with a fine-grain protocol, we delegate the offloading decision to the compiler so that it can minimize data sharing. Hence, the coherence mechanism proposed in this work invalidates only conflicting memory region of the cache hierarchy at running time.

The proposed cache coherence mechanism behavior is illustrated in Figure 2, and it works as follows: Before sending a memory access instruction (illustrated as *PIM LOAD* 256B from address 0x200), the LSQ unit emits a *flush* request of the corresponding NDA memory operand size (256 Bytes) targeting the same address of the memory operation (0x200). The *flush* request is divided according to the cache line size, and it is sent to the data cache memory port. The operation size can range from 4 Bytes to 8 KBytes [2, 25]), and considering a cache line size of 64 Bytes, 4 flush operations are required to flush 256 Bytes of data, as illustrated in Fig. 2. The flush request is sent to the first level data cache and then it is forwarded to the next cache level until it arrives at the last level cache, where the request is transmitted back to the LSQ unit. At each cache level, a specific hardware module interprets the *flush* request and triggers lookups for cache blocks within the address range of the NDA memory access instruction that originated the *flush* request. If there are cache blocks affected, they will either cause a *writeback* or an *invalidate* command that is enqueued in the write-buffer and finally, the *flush* request is enqueued. It is important to notice that the proposed mechanism maintains coherence between a single-core host and NDA. However, for multi-core host CPUs emitting instructions to NDAs, we need an existing cache coherence, (e.g. MOESI), to be in charge of keeping data shared coherent between the hosts. After the last

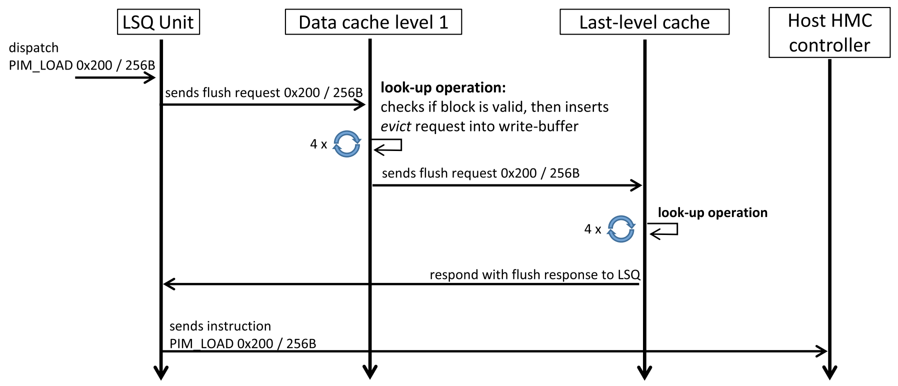
LSQ Unit
Data cache level 1
Last-level cache
Host HMC controller
dispatch PIM_LOAD 0x200 / 256B
sends flush request 0x200 / 256B
look-up operation:
checks if block is valid, then inserts evict request into write-buffer
4 x
sends flush request 0x200 / 256B
look-up operation
4 x
respond with flush response to LSQ
sends instruction PIM_LOAD 0x200 / 256B

Fig. 2 Cache Coherence Protocol - Example for a 256 Bytes PIM LOAD instruction

flush operation is flagged as done, the *NDA Instructions Dispatcher* allows the NDA instruction to follow its way towards the main memory. This way, the PIM instruction and the in-cache data will follow in-order towards the main memory, which ensures data coherence.

4.3 NDA Instructions and Data Racing Management

Since host CPU and NDA instructions share the same address space, it is likely that data race within the 3D-stacked memory occurs. Also, the host may request data whose target *vault* is not the same as the *vault* where the data is being processed by a PIM instruction. Excluding the interference of requests from the host CPU, a code region that triggers multiple NDA instances is prone to have a data race between requests from distinct instances. Moreover, PIM instructions can use registers and memory addresses from distinct *vaults* to achieve high performance, however these behaviors also increase the chances of data racing. Additionally, a single instruction may trigger multiple PIM units at once, which requires efficient control over internal communication. Therefore, leaving data racing unmanaged can potentially cause data hazards, incorrect results, and unpredictable application behavior. For this reason, a data racing protocol is required to keep requests ordered and synchronized.

We consider a crossbar switching tree as an implementation of the interconnection network used in the logic layer of a 3D-stacked memory. This network is not only used for a request coming from the host CPU, but also requests made from one *vault* to another, which are called here as *intervault* requests. On top of that, we implemented a protocol for solving coherence and data racing of host-NDA and NDA-NDA communication using an *acquire-release* transaction approach. To enable the proposed communication, we define three commands to use within the *intervault* communication subsystem: *memory-write* and *memory-read*, and *register-read* requests. These requests can be used with either *acquire* or *release* flag and they carry a sequence number related to the original NDA instruction, which allows maintaining the ordering of the requests.

NDA Instruction and Data Racing Manager module is highlighted in Fig. 1, which allows the PIM architecture to synchronize and keep the order of memory requests as soon as the host CPU or NDA instructions arrive in the NDA logic.

4.3.1 Intervault Communication

As aforementioned, PIM instructions may require data from different memory *vaults* to better use the available resources. For instance, Fig. 3 illustrates the sequence of operations for allowing a *PIM_LOAD* instruction that arrives in the *vault 0*, to access data from the *vault 2*. In short, when an NDA instruction is dispatched from the LSQ unit, it crosses the HMC serial link and arrives at the Data Racing Manager (shown in Fig. 1). In this module, the *acquire memory-read* or *acquire memory-write* requests are generated for memory access instruction, or *acquire register-read* requests for modifying instructions involving registers from different vaults. In case of memory access the *acquire memory-read* is generated for *LOAD* instructions, and it is sent to the *source vault*. For *STORE* instructions, the *acquire memory-write* is generated, and this command is sent to the *destination vault*. Similarly, instructions that demand two *sources* require two *acquire* commands that are sent to the *source vaults*, while the *destination vault* is responsible for the release commands.

Finally, when the NDA instruction is decoded in the NDA FSM, its LSQ unit generates a *memory-write*, *memory-read* or *register-read* request with a *release* flag. In the target *vault* controller, the *release* request will either unlock the *register-read* instruction in the NDA's *Instruction Queue* or remove a blocking request from the memory buffer, according to the instruction type. Thus, the NDA execution flow can continue without any data hazard. Internally, the *NDA Instruction and Data Racing Manager* submodule called *Instruction Splitter* is responsible for identifying the source address/register according to instructions.

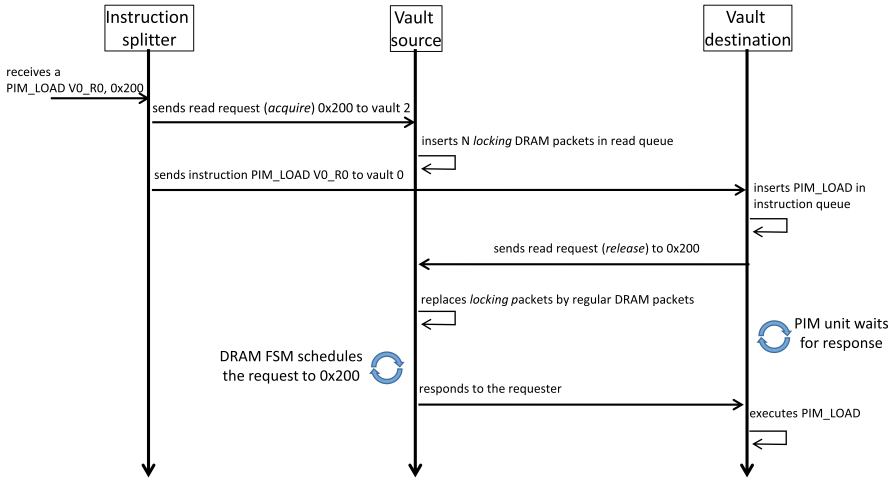
Instruction splitter
Vault source
Vault destination
receives a PIM_LOAD V0_R0, 0x200
sends read request (acquire) 0x200 to vault 2
inserts N locking DRAM packets in read queue
sends instruction PIM_LOAD V0_R0 to vault 0
inserts PIM_LOAD in instruction queue
sends read request (release) to 0x200
replaces locking packets by regular DRAM packets
PIM unit waits for response
DRAM FSM schedules the request to 0x200
responds to the requester
executes PIM_LOAD

Fig. 3 Intervault communication protocol example for a 256 bytes PIM LOAD

As another example, the instruction presented in Listing 2 is a vector *ADD* where the PIM placed within *vault 0* requires data from registers belonging to PIMs within *vault 10* and *vault 28*, respectively. In this case, two *acquire register-read* commands are dispatched from *NDA Instruction and Data Racing Manager* (to PIM_10 and PIM_28), and the original instruction is sent to the destination PIM.

Listing 2: Example of a PIM ADD Instruction

```
PIM_256B_VADD_DWORD PIM_0_R256B_0, PIM_10_R256B_3, PIM_28_R256B_2
```

4.3.2 Big Vector Instructions Support

Taking advantage of the *intervault* protocol, it is possible to provide resources for big vector instructions that can operate all PIM units at once.

Listing 3: Example of a Big Vector PIM ADD Instruction

```
PIM_8192B_VADD_DWORD PIM_0-31_R8192B_0, PIM_0-31_R8192B_3, PIM_0-31_R8192B_2
```

Considering the case study presented in Section 3, each PIM unit manages 256 Bytes, and within an environment of 32 *vaults* up to 8192 Bytes of data can be accessed in parallel fashion [2, 25]. As illustrated in Listing 3, it is possible to trigger 8192 Bytes at once with the same instruction. In this case, 32 PIM units need to be triggered, concurrently. For this, the *NDA Instruction and Data Racing Manager* splits the original instruction into 32 sub-instructions. Each sub-instruction must be delivered to the correct NDA unit (*source and destination*) to keep processing consistence and data coherence. Hence, it is possible to adopt the same protocol illustrated in Figure 3, and similarly to the case of the instruction in Listing 2, it is possible to trigger many sub-instructions as necessary to support big vector instructions. This means that several *acquire* and *release* commands may be generated to allow the allocation of the demanded resources.

5 Experimental Setup and Results

In this section, we present the methodology used to evaluate our mechanism and results.

5.1 Evaluation Setup

In order to accurate our simulation, we have implemented all the mechanisms mentioned in Section 4 on the NDA framework presented in [24], which comprises a GEM5-based simulator [7], and an automation compiler tool for NDA software development [2, 24]. Further, we use a subset of the PolyBench benchmark suite to evaluate the impact of the proposed architecture in most of scientific kernel applications [22]. Also, to show the impact on performance caused by the proposed mechanisms, we compare the case study against a General Purpose Processor (GPP) with the largest vector capacity available (AVX-512). This way, we can analyze how much performance is hindered due to cache coherence and *intervault* communication. Table 1 summarizes the setup simulated.

5.2 Performance Results

Figure 4 presents the execution time results for small kernels, which is decomposed into three regions. The bottom blue region represents the time spent only computing the kernel within the in-memory logic. The red region highlighted in the middle depicts the cost of *inter-vault* communication, while the top green region represents the cost to keep cache coherence. It is important to notice that the blue region also represents the lowest possible execution time, which occurs when no penalties are inferred by the *intervault* communication (red region) and cache coherence (green region) mechanisms.

In Figure 4, the *vecsum* kernel shows that more than 70% of the time is spent in cache coherence and internal communication, while only 30 % of the time is actually used for processing data. Although most of the *vecsum* kernel is executed in NDA, hence the data remains in the memory device during all execution time and no hits (writeback or clean eviction) should be seen in cache memories, there is a fixed cost for lookup operations to prevent data inconsistency. Regarding the matrix-multiplication and dot-product kernels in Figure 4, the impact of *flush* operations is amortized by the lower ratio of NDA memory access per NDA modification instructions.

Since the *flush* operation generally triggers lookups to more than one cache block addressed by an NDA instruction, the overall latency will depend on each cache-

Table 1 Baseline and design system configuration

Intel Skylake Microarchitecture
4GHz; AVX-512 Instruction Set Capable; L3 Cache 16MB
8GB HMC; 4 Memory Channels
HMC
HMC version 2.0 specification
Total DRAM Size 8GBytes - 8 Layers - 8Gbit per layer
32 Vaults - 16 Banks per Vault; 4 high speed Serial Links
RVU
1GHz; 32 Independent Functional Units; Integer and Floating-Point Capable
Vectorial Operations up to 256Bytes per Functional Units
32 Independent Register Bank of 8x256Bytes each
Latency (cycles): 1-alu, 3-mul. and 20-div. integer units
Latency (cycles): 5-alu, 5-mul. and 20-div. floating-point units
Interconnection between vaults: 5 cycles latency

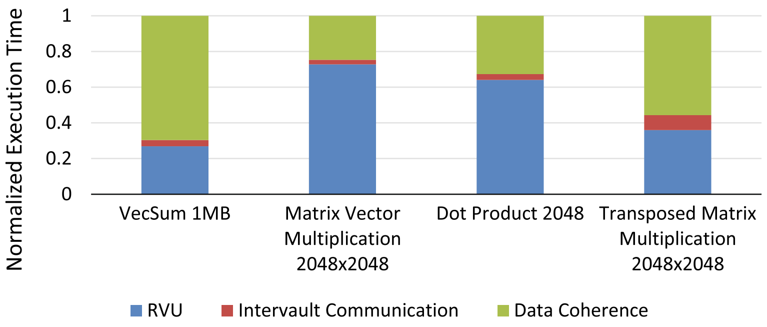

Fig. 4 Execution time of common kernels to illustrate the costs of Cache Coherence and *Inter-vault communication*

level lookup latency. Also, for each *flush* request dispatched from the LSQ, all cache levels will receive the request forward propagation, but it is executed sequentially from the first-level to last-level cache. Only improvements in lookup time or reduced cache hierarchy would impact in the performance of *flush* operations. On the other hand, *inter-vault* communication penalty generally has little impact on the overall performance. For the transposed matrix-multiplication kernel, it is possible to see the effect of a great number of *register-read* and *mem-read* to different *vaults* inherent to the application loop.

Figure 5 shows the PolyBench runtime for the PIM case study over the GPP AVX-512 baseline. Regarding *flush* operations and *inter-vault* as costs that could be avoided, the blue region (bottom region in Figure 5) represents the time dispended by the PIM to compute logic/arithmetic operations, which is inherent to the PIM design. On the other hand, the red region (top region) represents the time consumed by the proposed mechanisms to keep cache coherence and allow communication between different memory *vaults* and PIM units.

In general, considering the presented mechanisms, the PIM can achieve performance improvements between 2.5× (*jacobi-2D*) and 14.6× (*jaboci-1D*) over the baseline (AVX-512). For *jacobi-2D*, 82% of the execution time is used for computation, and hence only 18% for cache coherence and *inter-vault* communications. As an opposite example, for applications *syrk*, *syrk2k* and *bicg*, it is possible to achieve speedup of 13×, 13.6× and 12.5×, respectively. However, int these cases, the proposed mechanisms use between 52% and 54% of the execution time to keep cache coherence and allow *inter-vault* communication.

Therefore, our proposal provides a competitive advantage in terms of speedup in comparison to other HMC-instruction-based NDA setups. For instance, the proposal presented in [17] relies on *uncacheable* data region, hence no hardware cost is introduced. However, it comes with a cost in how much performance can be extracted when deciding if a code portion must be executed in the host core or in NDA units. Besides, the speculative approach proposed in [8] has only 5% of performance penalty compared to an ideal NDA, but the performance can profoundly degrade if rollbacks are frequently made, which will depend on the

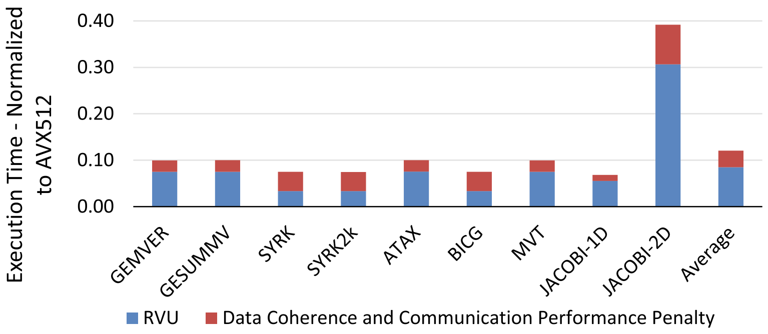

Fig. 5 Execution time of PolyBench normalized to AVX-512

application behavior. Also, another similar work [4] advocates locality-aware NDA execution to avoid *flush* operations and off-chip communication. However, they do not consider that large vectors in NDA can amortize the cost of cache coherence mechanism even if, eventually, the host CPU has to process scalar operands on the same data region.

6 Conclusions and Future Work

In this paper, we presented an efficient approach to solve both offloading of instructions, keep data coherence, and manage communication in PIM architectures. Based on the hybrid Host-PIM style, our mechanism transparently allows offloading of PIM instructions directly from a host processor without incurring overheads. The proposed data coherence protocol offers programmability and cache coherence resources to reduce programmers and compilers' effort in designing applications to be executed in PIM architectures. This work presents an acquire-release communication protocol to cope with distributed PIM units requirements. The experiments show that our mechanism can accelerate applications up to 14.6× compared to a AVX architecture, while the penalty due to cache coherence and communication represents an average percentage of 18% over the ideal PIM. In future work, we intend to improve the design presented, aiming at reducing the costs of offloading and cache coherence, allowing the adoption of PIM with zero-cost latency. Also, we expect to analyze a broader range of applications using our proposed data-path.

References

1. Aga, S., Jeloka, S., Subramaniyan, A., Narayanasamy, S., Blaauw, D., Das, R.: Compute caches. In: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 481–492 (2017)

2. Ahmed, H., Santos, P.C., de Lima, J.P.C., de Moura, R.F., Alves, M.A., Beck, A., Carro, L.: A compiler for automatic selection of suitable processing-in-memory instructions. In: Design, Automation & Test in Europe Conference & Exhibition (DATE) (2019)
3. Ahn, J., Hong, S., Yoo, S., Mutlu, O., Choi, K.: A scalable processing-in-memory accelerator for parallel graph processing. In: International Symposium on Computer Architecture (2015)
4. Ahn, J., Yoo, S., Mutlu, O., Choi, K.: Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In: International Symposium on Computer Architecture (ISCA), pp. 336–348. IEEE (2015)
5. Akin, B., Franchetti, F., Hoe, J.C.: Data reorganization in memory using 3d-stacked dram. In: ACM SIGARCH Computer Architecture News, vol. 43, pp. 131–143. ACM (2015)
6. Alves, M.A., Diener, M., Santos, P.C., Carro, L.: Large vector extensions inside the hmc. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1249–1254. IEEE (2016)
7. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., et al.: The gem5 simulator. ACM SIGARCH Computer Architecture News **39**, (2011)
8. Boroumand, A., Ghose, S., Lucia, B., Hsieh, K., Malladi, K., Zheng, H., Mutlu, O.: LazyPIM: an efficient cache coherence mechanism for processing-in-memory. IEEE Comput. Architect. Lett. **16**(1), 46–50 (2016)
9. Drumond, M., Daglis, A., Mirzadeh, N., Ustiugov, D., Picorel, J., Falsafi, B., Grot, B., Pnevmatikatos, D.: The mondrian data engine. In: International Symposium on Computer Architecture. ACM (2017)
10. Eckert, Y., Jayasena, N., Loh, G.H.: Thermal feasibility of die-stacked processing in memory. In: 2nd Workshop on Near-Data Processing (WoNDP) (2014)
11. Gao, M., Kozyrakis, C.: HRL: efficient and flexible reconfigurable logic for near-data processing. In: International Symposium High Performance Computer Architecture (HPCA) (2016)
12. Hsieh, K., Ebrahimi, E., Kim, G., Chatterjee, N., O'Connor, M., Vijaykumar, N., Mutlu, O., Keckler, S.W.: Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems. ACM SIGARCH Comput. Architect. News **44**(3), 204–216 (2016)
13. Hsieh, K., Khan, S., Vijaykumar, N., et al.: Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In: International Conference on Computer Design (ICCD) (2016)
14. Hybrid Memory Cube Consortium: Hybrid memory cube specification rev. 2.0 (2013). https://www.micron.com/-/media/client/global/documents/products/data-sheet/hmc/gen2/hmc_gen2.pdf
15. Lee, J.H., Sim, J., Kim, H.: Bssync: Processing near memory for machine learning workloads with bounded staleness consistency models. In: International Conference on Parallel Architecture and Compilation (PACT), pp. 241–252. IEEE (2015)
16. de Lima, J.P.C., Santos, P.C., Alves, M.A., Beck, A., Carro, L.: Design space exploration for PIM architectures in 3d-stacked memories. In: Proceedings of the 15th ACM International Conference on Computing Frontiers, pp. 113–120. ACM (2018)
17. Nai, L., Hadidi, R., Sim, J., Kim, H., Kumar, P., Kim, H.: Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In: International Symposium High Performance Computer Architecture (HPCA), pp. 457–468. IEEE (2017)
18. Nair, R., Antao, S.F., Bertolli, C., Bose, P., et al.: Active memory cube: a processing-in-memory architecture for exascale systems. IBM J. Res. Develop. **59**, 2–3 (2015)
19. Oliveira, G.F., Santos, P.C., Alves, M.A., Carro, L.: Nim: An hmc-based machine for neuron computation. In: International Symposium on Applied Reconfigurable Computing (2017)
20. Papamarcos, M.S., Patel, J.H.: A low-overhead coherence solution for multiprocessors with private cache memories. SIGARCH Comput. Archit. News **12**(3), 348–354 (1984). https://doi.org/10.1145/773453.808204
21. Pawlowski, J.T.: Hybrid memory cube (HMC). In: Hot Chips 23 Symposium (HCS). IEEE (2011)
22. Pouchet, L.N.: Polybench: The polyhedral benchmark suite. http://www.cs.ucla.edu/pouchet/software/polybench (2012)
23. Pugsley, S., Jestes, J., Balasubramonian, R., et al.: Comparing Implementations of Near-Data Computing with In-Memory MapReduce Workloads. IEEE Micro **34**(4), 44–52 (2014)
24. Santos, P.C., de Lima, J.P.C., de Moura, R.F., Ahmed, H., Alves, M.A., Beck, A., Carro, L.: Exploring IoT platform with technologically agnostic processing-in-memory framework. In: Proceedings of the Workshop on INTelligent Embedded Systems Architectures and Applications, pp. 1–6. ACM (2018)

25. Santos, P.C., Oliveira, G.F., Tomé, D.G., Alves, M.A., Almeida, E.C., Carro, L.: Operand size reconfiguration for big data processing in memory. In: Design, Automation & Test in Europe Conference & Exhibition (DATE) (2017)
26. Scrbak, M., Islam, M., Kavi, K.M., Ignatowski, M., Jayasena, N.: Exploring the processing-in-memory design space. J. Syst. Architect. **75**, 59–67 (2017)
27. Singh, G., Chelini, L., Corda, S., Awan, A.J., Stuijk, S., Jordans, R., Corporaal, H., Boonstra, A.J.: A review of near-memory computing architectures: Opportunities and challenges. In: Euromicro Conference on Digital System Design (DSD) (2018)
28. Standard JEDEC: High Bandwidth Memory (HBM) DRAM. JESD235 (2013)
29. Zhang, D., Jayasena, N., Lyashevsky, A., et al.: TOP-PIM: throughput-oriented programmable processing in memory. In: International Symposium on High-performance Parallel and Distributed Computing (2014)

Affiliations

Paulo C. Santos[1] · **João P. C. de Lima**[1] · **Rafael F. de Moura**[1] · **Marco A. Z. Alves**[2] · **Antonio C. S. Beck**[1] · **Luigi Carro**[1]

João P. C. de Lima
jpclima@inf.ufrgs.br

Rafael F. de Moura
rfmoura@inf.ufrgs.br

Marco A. Z. Alves
mazalves@inf.ufpr.br

Antonio C. S. Beck
caco@inf.ufrgs.br

Luigi Carro
carro@inf.ufrgs.br

[1] Federal University of Rio Grande do Sul, Porto Alegre, Brazil

[2] Federal University of Paraná, Curitiba, Brazil