Solving Datapath Issues on Near-Data Accelerators

Omitted for blind review

Abstract. Leveraged by the advent of big data applications, many studies have been migrating accelerators closer and closer to the data in order to improve efficiency and performance. The most prominent types are classified as embedded Near-Data Accelerator (NDA) or Processor-in-Memory (PIM) accelerators. However, the adoption of these accelerators requires at least three novel mechanisms, since traditional approaches are not suitable to tackle these issues: how to offload instructions from the host to NDA, how to keep cache coherence, and how to deal with the internal communication between different NDA units. 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. 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 Advanced Vector Extensions (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: In-Memory Processing · Cache Coherence · Instruction Offloading · Vector Instructions · 3D-stacked memories.

1 Introduction

In the last years, the arising of big data workloads, machine learning algorithms, and data-intensive applications have demanded different flavors of accelerators. Meanwhile, trying to mitigate the memory wall effects widely increased by the numerous accelerators present on modern embedded systems, 3D-stacking technology has been evolving and enabling improvements on memory bandwidth and chip capacity. Both the requirements of data-intensive workloads and the emergence of 3D-stacked memories have been leveraging the resurgence of Processor-in-Memory (PIM) studies. Moreover, supported by the integration of logical and memory layers, such as found in the Hybrid Memory Cube (HMC) [13] and High Bandwidth Memory (HBM) [23], 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, thus

reducing energy 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 320GB/s [13, 23]. Moreover, the logic design must fit within the logic layer area and power constraints of 3D-stacked memories [9, 17]. Past studies, such as [21, 15, 22], 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 [9, 15], 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.

Although several works present solutions for the instruction-level offloading issue [4, 14, 16, 7], they lack solving two inherent challenges that comes with this approach: how data is kept coherent between host and PIM, and how data is kept coherent between multiple computing units within 3D-stacked memory. The solutions present in the literature demand specialized extra hardware while limit the generality of cache memories [4], isolate memory address between different processing elements and restrict the parallelism exploitation of 3D-stacked memories [11], and do not support Translation Look-aside Buffer (TLB) and virtual memory [7]. Moreover, the communication between different PIM units is neglected in previous studies available in the literature.

The present work focuses on solving the aforementioned main issues to allow a transparent offloading of PIM instructions directly from the host processor without incurring 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 application 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: Section 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 on Section 4. The results are present on Section 5, and the conclusions on Section 6.

2 The Road so Far

Although minimal changes in CPU may minimize the effort to programmers, it opens up new challenges, such as deciding between host and PIM instructions, how to keep coherence among PIM devices and host processor, and how to manage communication involving both PIM units (PIM-PIM communication) or CPUs (Host-PIM communication).

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, 14, 16]. 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, 11]. 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 [7, 11]). 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 [7, 11, 3, 4]. 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]. Another approach is to have split addressing spaces for each PIM unit [11], 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 [7], 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 others 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, 4, 12]. In other approaches [7], the coherence is kept within the first data cache level of each PIM core taking use of a *MESI* 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], a special module called PEI Managment 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). All PEI instructions access the 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 [11]. In this case, cache coherence is maintained by a three-step protocol: The Streaming Machine (SMT) that requested the instruction offloading pushes all memory update traffic from itself to memory before it sends the offloading request. Second, the memory stack SMT invalidates its private data cache. Third, memory stack SM sends all its modified data cache lines to the SMT 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 [11, 4, 7]. 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 [11], or they have fixed address ranges without shared memory locations [16]. In other works, where there is PIM-PIM communication, it is managed by MESI-based modified protocols in a similar way which MultiProcessor System-on-Chip (MPSoC) do [4].

3 A Near-Data Accelerator (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 [16] propose the use of the native PIM presented in the HMC [13] to accelerate graph algorithms. The HMC is the first 3D-stacked

Listing 1.1: 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
```

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, 20] show PIM designs to exploit data-level parallelism by providing vector processing units in memory, also relying on instruction offloading and requiring data coherence mechanisms.

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 [15]. As described in [15], the micro-architecture called Reconfigurable Vector Unit (RVU) [19] meets power and area constraints given by HMC while providing the maximum processing bandwidth when compared to other recent 3D-stacked PIM proposals [8, 21, 10, 3]. 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 Instruction Set Architecture (ISA) is a subset of Intel's Advanced Vector Extensions (AVX) that contains arithmetic, logic, data transfer, and data reordering operations. As described in [15], 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]. Moreover, as RVU augments the CPU data-path, native host instructions and PIM instructions are expected to happen in the same binary code [2] and use the same address space as illustrated on Listing 1. Each RVU instance can request a region of memory from another vault or even request an entire register from a distinct instance using the internal logic-layer communication path.

4 Mechanism

The goal of this paper is to provide an efficient implementation of host-NDA interface, thus allowing the transparent utilization of 3D-stacked memory bandwidth with the lowest possible performance overhead. Figure 1 illustrates the proposed flow 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. Thus, modules present in any host CPU, such as TLB, page walker and even host registers can be reused without incurring additional hardware. For instance, memory access instructions, such as *PIM_LOAD* and *PIM_STORE*, rely on the host address translation mechanism, 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 a NDA instruction in the host CPU. Part of the instruction fields in the NDA ISA can be used to read or write x86 registers, and to calculate the memory addresses using any x86 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. The *NDA Instruction Dispatcher* unit illustrated in Figure 1 presents the modifications made in the LSQ to support the instruction-level offloading. 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.

As NDA instructions are sent as a regular packets to the memory system, they are addressed by the destination NDA unit and an architecture-specific flag

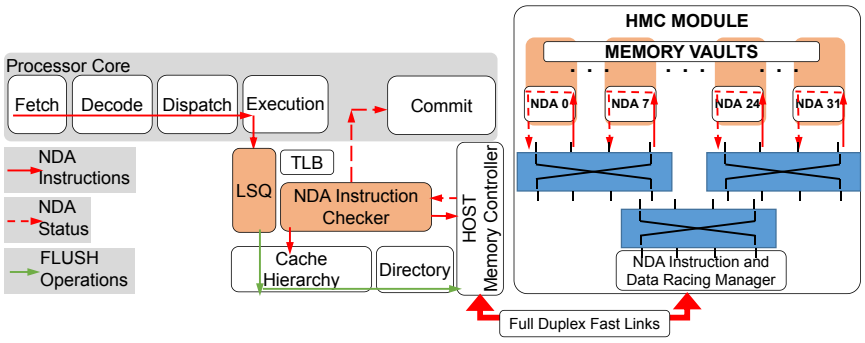

Fig. 1: Datapath for efficient utilization of NDA logic

is set in the packet to differ it from typical *read* 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.

4.2 Data coherence between NDA and host's cache hierarchy

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., MOESI, may not be enough 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 the 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, and coherence mechanism proposed here invalidates only conflicting memory region of the cache hierarchy.

The cache coherence data management mechanism proposed works as follows: Before sending a memory access instruction, the LSQ unit emits a *flush* request of the corresponding NDA memory operand size (ranging from 4 Bytes to 8 KBytes) to the data cache memory port. The flush request is sent to the first level data cache and then it is forwarded to the next cache level until it arrives 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 maintain coherence between a single-core host and NDA. However, for multi-core host CPUs emitting instructions to NDAs, we need an existing cache coherence, such as MOESI, to be in charge of keeping data shared coherent between the hosts.

4.3 Data coherence among NDA units within a memory device

Since host CPU and NDA instructions share the same address space, it is likely to occur a data race within the 3D-stacked memory. Even 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. Leaving it unhandled 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 *inter-vault* 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 protocol. To do so, we define three commands to use within the *inter-vault* communication subsystem: *memory-write* and *memory-read* and *register-read* requests. These requests can be used with either *acquire* or *release* flag and carry a sequence number related to the original NDA instruction.

Figure 1 illustrates the distribution of the *NDA Instruction and Data Racing Manager*, which allows the PIM architecture to synchronize and keep the order of memory requests as soon as the NDA instruction arrives in the NDA logic. In short, when a NDA instruction is dispatched from the LSQ unit, it crosses the HMC serial link and arrives at the Data Racing Manager. 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. 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. Thus, the NDA execution flow can continue without any data hazard.

5 Experimental Setup and Results

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

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;

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 [20], which comprises a GEM5-based simulator [6], and an automation compiler tool for NDA software development [2, 20]. Further, we use a subset of the PolyBench benchmark suite to evaluate the impact of the proposed architecture in most of scientific kernel applications [18]. Table 1 summarizes the setup simulated.

5.2 Performance Results

Figure 2 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 possible to notice in the *vecsum* kernel 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 2, 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 a NDA instruction, the overall latency will depend on each cache-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

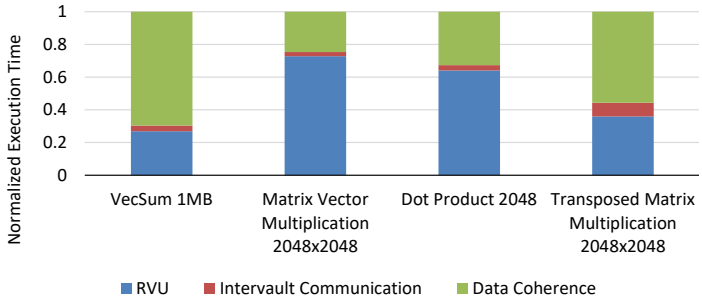

Fig. 2: Execution time of common kernels to illustrate the costs of Cache Coherence and *Inter-vault communication*

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.

Regarding *flush* operations and *inter-vault* communication as costs that could be avoided, in Figure 3 it is shown the overall performance improvement of an ideal NDA (no flush and no *inter-vault* communication), and the performance penalty employing the work's proposal in some benchmarks of Polybench Suite. In general, the present mechanism can achieve speed-up improvements between 2.5× and 14.6×, while using 82% of the execution time for computation, and hence only 18% for cache coherence and *inter-vault* communications. 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 [16] 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 [7] 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

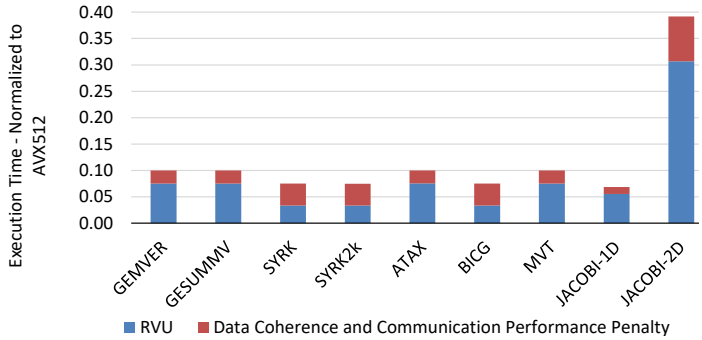

Fig. 3: Execution time of PolyBench normalized to AVX-512

the 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 works, 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 Int. Symp. on High Performance Computer Architecture (HPCA). pp. 481–492 (Feb 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: Int. Symp. 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: Int. Symp. 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. 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)
7. 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 Computer Architecture Letters (2016)

8. Drumond, M., Daglis, A., Mirzadeh, N., Ustiugov, D., Picorel, J., Falsafi, B., Grot, B., Pnevmatikatos, D.: The mondrian data engine. In: Int. Symp. on Computer Architecture. ACM (2017)
9. Eckert, Y., Jayasena, N., Loh, G.H.: Thermal feasibility of die-stacked processing in memory. In: 2nd Workshop on Near-Data Processing (WoNDP) (2014)
10. Gao, M., Kozyrakis, C.: Hrl: efficient and flexible reconfigurable logic for near-data processing. In: Int. Symp. High Performance Computer Architecture (HPCA) (2016)
11. 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 Computer Architecture News **44**(3), 204–216 (2016)
12. Hsieh, K., Khan, S., Vijaykumar, N., et al.: Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In: Int. Conf. on Computer Design (ICCD) (2016)
13. Hybrid Memory Cube Consortium: Hybrid memory cube specification rev. 2.0 (2013), http://www.hybridmemorycube.org/
14. Lee, J.H., Sim, J., Kim, H.: Bssync: Processing near memory for machine learning workloads with bounded staleness consistency models. In: Int. Conf. on Parallel Architecture and Compilation (PACT). pp. 241–252. IEEE (2015)
15. 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)
16. Nai, L., Hadidi, R., Sim, J., Kim, H., Kumar, P., Kim, H.: Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In: Int. Symp. High Performance Computer Architecture (HPCA). pp. 457–468. IEEE (2017)
17. Pawlowski, J.T.: Hybrid memory cube (hmc). In: Hot Chips 23 Symposium (HCS). IEEE (2011)
18. Pouchet, L.N.: Polybench: The polyhedral benchmark suite. URL: http://www. cs. ucla. edu/pouchet/software/polybench (2012)
19. 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)
20. 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)
21. Scrbak, M., Islam, M., Kavi, K.M., Ignatowski, M., Jayasena, N.: Exploring the processing-in-memory design space. Journal of Systems Architecture **75**, 59–67 (2017)
22. 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 Conf. on Digital System Design (DSD) (2018)
23. Standard JEDEC: High Bandwidth Memory (HBM) DRAM. JESD235 (2013)