Design and Analysis of a Processing-in-DIMM Join Algorithm: A Case Study with UPMEM DIMMs

CHAEMIN LIM, Yonsei University, South Korea
SUHYUN LEE, Yonsei University, South Korea
JINWOO CHOI, Yonsei University, South Korea
JOUNGHOO LEE, Yonsei University, South Korea
SEONGYEON PARK, Seoul National University, South Korea
HANJUN KIM, Yonsei University, South Korea
JINHO LEE, Seoul National University, South Korea
YOUNGSOK KIM*, Yonsei University, South Korea

Modern dual in-line memory modules (DIMMs) support processing-in-memory (PIM) by implementing in-DIMM processors (IDPs) located near memory banks. PIM can greatly accelerate in-memory join, whose performance is frequently bounded by main-memory accesses, by offloading the operations of join from host central processing units (CPUs) to the IDPs. As real PIM hardware has not been available until very recently, the prior PIM-assisted join algorithms have relied on PIM hardware simulators which assume fast shared memory between the IDPs and fast inter-IDP communication; however, on commodity PIM-enabled DIMMs, the IDPs do not share memory and demand the CPUs to mediate inter-IDP communication. Such discrepancies in the architectural characteristics make the prior studies incompatible with the DIMMs. Thus, to exploit the high potential of PIM on commodity PIM-enabled DIMMs, we need a new join algorithm designed and optimized for the DIMMs and their architectural characteristics.

In this paper, we design and analyze *Processing-In-DIMM Join (PID-Join)*, a fast in-memory join algorithm which exploits UPMEM DIMMs, currently the only publicly-available PIM-enabled DIMMs. The DIMMs impose several key challenges on efficient acceleration of join including the shared-nothing nature and limited compute capabilities of the IDPs, the lack of hardware support for fast inter-IDP communication, and the slow IDP-wise data transfers between the IDPs and the main memory. PID-Join overcomes the challenges by prototyping and evaluating hash, sort-merge, and nested-loop algorithms optimized for the IDPs, enabling fast inter-IDP communication using host CPU cache streaming and vector instructions, and facilitating fast rank-wise data transfers between the IDPs and the main memory. Our evaluation using a real system equipped with eight UPMEM DIMMs and 1,024 IDPs shows that PID-Join greatly improves the performance of in-memory join over various CPU-based in-memory join algorithms.

CCS Concepts: • **Information systems → Join algorithms**; **Main memory engines**; • **Hardware → Dynamic memory**; *Memory and dense storage.*

*Corresponding author

Authors' addresses: Chaemin Lim, cmlim@yonsei.ac.kr, Yonsei University, Seoul, South Korea; Suhyun Lee, su_hyun@yonsei.ac.kr, Yonsei University, Seoul, South Korea; Jinwoo Choi, jinwoo1029@yonsei.ac.kr, Yonsei University, Seoul, South Korea; Jounghoo Lee, jounghoolee@yonsei.ac.kr, Yonsei University, Seoul, South Korea; Seongyeon Park, syeonp@snu.ac.kr, Seoul National University, Seoul, South Korea; Hanjun Kim, hanjun@yonsei.ac.kr, Yonsei University, Seoul, South Korea; Jinho Lee, leejinho@snu.ac.kr, Seoul National University, Seoul, South Korea; Youngsok Kim, youngsok@yonsei.ac.kr, Yonsei University, Seoul, South Korea.

2836-6573/2023/6-ART113 $15.00
https://doi.org/10.1145/3589258

Additional Key Words and Phrases: in-memory join, processing-in-memory, processing-in-DIMM

ACM Reference Format:
Chaemin Lim, Suhyun Lee, Jinwoo Choi, Jounghoo Lee, Seongyeon Park, Hanjun Kim, Jinho Lee, and Youngsok Kim. 2023. Design and Analysis of a Processing-in-DIMM Join Algorithm: A Case Study with UPMEM DIMMs. *Proc. ACM Manag. Data* 1, 2, Article 113 (June 2023), 27 pages. https://doi.org/10.1145/3589258

1 INTRODUCTION

Processing-in-memory (PIM) is a promising solution to mitigate the data movement bottleneck caused by the limited bandwidth between host central processing units (CPUs) and main memory [21, 38, 47]. Unlike conventional dual in-line memory modules (DIMMs) having no compute capabilities, modern commodity PIM-enabled DIMMs are equipped with in-DIMM processors (IDPs) located near memory banks. The DIMMs allow applications to offload their computation from host CPUs to the IDPs. The IDPs perform computation by directly accessing the data stored in the memory banks without involving the host-side resources. In this way, PIM can greatly help applications overcome the data movement bottleneck by reducing the amount of data transfer between the CPUs and main memory and by exploiting the high internal memory bandwidth and computational throughput of the DIMMs.

Aimed at exploiting PIM to improve the performance of relational database management systems (RDBMSs), prior studies have proposed to accelerate relational operations using PIM [16, 17, 35, 43, 49]. As real PIM hardware has not been available until very recently, the prior studies heavily rely on PIM hardware simulators and various architectural assumptions on the PIM hardware.

Unfortunately, we find that the architectural assumptions of the prior studies greatly differ with those of commodity PIM-enabled DIMMs (e.g., UPMEM DIMMs [12]), making the prior studies incompatible with the DIMMs. First, the IDPs are assumed to have single-instruction multiple-data (SIMD) units capable of executing vector operations (e.g., AVX-512); however, the DIMMs are equipped with multi-threaded, scalar IDPs providing limited native support only for integer addition/subtraction and bitwise operations. Second, the prior studies assume shared memory and fast communication paths between the IDPs, whereas the DIMMs rely on host CPUs for inter-IDP communication. Third, as the DIMMs follow an accelerator model and co-exist with the host-side main memory, the data transfer between the DIMMs and the main memory must be optimized. The prior studies, on the other hand, neglect the data transfers by assuming that all the data for join always reside in the DIMMs. Therefore, to realize the high potential of PIM on real systems, we need a new join algorithm designed and optimized for commodity PIM-enabled DIMMs.

In this paper, we design and analyze *Processing-In-DIMM Join (PID-Join)*, a fast in-memory join algorithm optimized to and evaluated with UPMEM DIMMs, currently the only publicly-available commodity PIM-enabled DIMMs. PID-Join employs the following key ideas to efficiently exploit the high internal memory bandwidth and computational throughput of the DIMMs. First, PID-Join employs a distributed and partitioned hash join algorithm which maximizes the join throughput on the IDPs. We prototype and evaluate IDP-friendly hash, sort-merge, and nested-loop join algorithms, and identify that the hash join algorithm is best-suited to the multithreaded, scalar IDPs having limited compute capabilities and the limited per-IDP scratchpad memory size. Second, PID-Join performs global partitioning of two tables on the DIMMs instead of the CPUs. Performing global partitioning on the DIMMs allows PID-Join to exploit the high internal memory bandwidth and computational throughput of the DIMMs, but incurs all-to-all inter-IDP communication involving the CPUs. PID-Join minimizes the performance overheads of the inter-IDP communication with host CPU cache streaming and vector instructions. Third, PID-Join minimizes the host-DIMM data transfer latency by exploiting the fact that performing join with global partitioning does not impose any strict ordering on the input and output tuples. PID-Join therefore accelerates the host-DIMM data transfers by transferring the tuples at the granularity of a rank, rather than an IDP. The rank-wise host-DIMM data transfers consume much lower host CPU bandwidth and achieve higher host-DIMM data transfer bandwidth.

We prototype and evaluate PID-Join on a real system equipped with eight UPMEM DIMMs having a total of 1,024 IDPs. Compared to parallel histogram based radix join algorithm [6] on the baseline system with 10 standard DDR4-2400 DIMMs, our evaluation shows that PID-Join achieves geometric mean speedups of 1.92× and 1.89× with uniform and skewed tables, respectively. The evaluation also shows that integrating PID-Join into MonetDB [9] can achieve a geometric mean speedup of 1.14× with single-join TPC-H queries. The results clearly show that PID-Join is not only fully compatible with commodity PIM-enabled DIMMs, but also a promising in-memory join algorithm which efficiently exploits the DIMMs.

To the best of our knowledge, PID-Join is the first in-memory join algorithm designed for and evaluated on real commodity PIM-enabled DIMMs. The key contributions of this paper are:

- We identify that the prior studies on accelerating in-memory join using PIM are incompatible with real commodity PIM-enabled DIMMs. The prior studies heavily rely on the simulators and architectural assumptions incompatible with the DIMMs.
- We present PID-Join, a fast in-memory join algorithm designed and optimized for commodity PIM-enabled DIMMs. PID-Join efficiently exploits the DIMMs by performing distributed and partitioned hash join and global partitioning on the DIMMs, optimizing the all-to-all inter-IDP communication, and employing fast rank-wise host-DIMM data transfers.
- We prototype and evaluate PID-Join on a real system equipped with eight UPMEM DIMMs. Our evaluation shows that PID-Join greatly outperforms modern CPU-based in-memory join algorithms with both uniform and skewed data distributions, and that integrating PID-Join into MonetDB, a full-fledged in-memory RDBMS, can accelerate single-join TPC-H queries.

1.1 Dual In-line Memory Modules

Dual in-line memory modules (DIMMs) are currently the de-facto standard for implementing the main memory of a system. Fig. 1 shows the internal architecture of standard DIMMs. A DIMM consists of several chips, and each of the chips consists of multiple banks. The data stored in a DIMM get interleaved across the chips of the DIMM, and the data stored in a chip get interleaved across the banks of the chip. Most of the DIMMs attached to modern systems operate with double data rate (DDR); DDR DIMMs can transfer data on both the rising and the falling edges of their

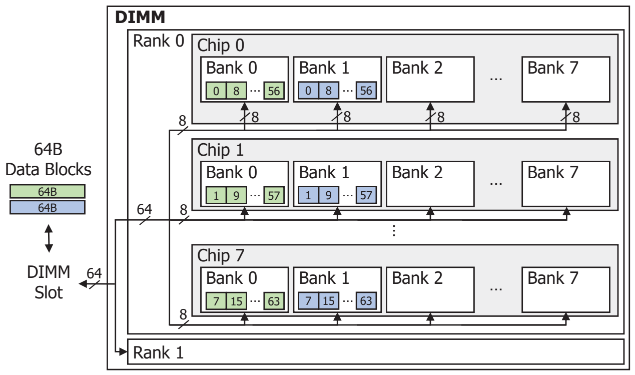


Fig. 1. DIMM architecture and data layout. The numbers denote the byte indices of two 64-byte data blocks.

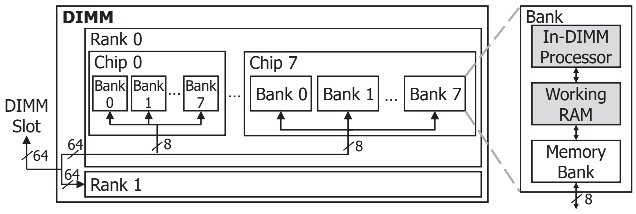


Fig. 2. UPMEM DIMM architecture

clock signals. The data stored in a DIMM can be accessed with a granularity of the burst length of the DIMM (e.g., 64 bytes for DDR4 DIMMs [4]).

One key characteristic of DIMMs is that a sequential piece of data does not get stored in a single chip. Instead, a DIMM groups its chips into ranks, and stores a sequential piece of data to one of the ranks. A rank refers to a set of chips sharing the same control signals [4]. This makes the data stored within a rank get interleaved across all the chips of the rank, typically with a granularity of one byte per chip. When a DIMM receives a data read/write request, the DIMM first identifies the target rank and then reads/writes the requested data from/to all the chips of the target rank. For example, reading a 64-bit word from a rank consisting of eight chips involves reading a single byte from each of the eight chips in a rank. Fig. 1 illustrates the rank-driven data layout of DIMMs.

To provide larger memory capacity and higher memory bandwidth, multiple DIMMs can be attached to a system. When attaching DIMMs to a system, organizing them into multiple memory channels is highly beneficial. As different memory channels can operate independently, increasing the number of memory channels linearly increases the total memory bandwidth of a system. Accordingly, a common practice for attaching multiple DIMMs to a system is to first increase the memory bandwidth by assigning one DIMM per memory channel, and then enlarge the memory capacity by attaching more DIMMs to the memory channels.

1.2 Processing-in-Memory

Modern commodity DIMMs now support processing-in-memory (PIM) by implementing in-DIMM processors (IDPs) located near the memory banks of a DIMM. Fig. 2 shows the internal architecture of UPMEM DIMMs, currently the only publicly-available commodity PIM-enabled DIMMs. An UPMEM DIMM implements PIM by associating a near-bank IDP to each of its 64-MB memory banks. The IDPs are in-order, scalar, and multithreaded; they support up to 24 threads and execute one instruction of a thread per cycle. An IDP cannot directly access the data stored in its memory bank. Instead, the IDP needs to load the data to its 64-KB Working RAM (WRAM), a static random-access memory (SRAM)-based scratchpad memory.

The procedure for performing computations on PIM-enabled DIMMs follows an accelerator model. An application first prepares a PIM kernel, a function which will be executed by the IDPs. Then, the application transfers the input data for the PIM kernel from the main memory (i.e., standard DIMMs) to a rank of a PIM-enabled DIMM. The application then invokes the execution of the PIM kernel to the IDPs of the rank, and the IDPs execute the PIM kernel using the data stored in their memory banks. After the IDPs complete executing the PIM kernel, the application retrieves the output data by copying them to the main memory.

PIM-enabled DIMMs are designed to be a drop-in replacement of standard DIMMs, so all the IDPs of the same rank of a PIM-enabled DIMM operate in a synchronous manner. All the IDPs of a rank either write data to their memory banks, execute the same PIM kernel using the data stored in their memory banks, or read data from their memory banks. The rank-driven nature of the DIMMs enables rank-level parallelism which allows different ranks to perform different operations. For example, the IDPs of one rank may write data to their banks while the IDPs of another rank execute

Table 1. Prior studies on PIM-assisted join algorithms

	Evaluation	**Join Algorithm(s)**
Mirzadeh et al. [43]	Simulation [32]	Hash, Sort-Merge
Drumond et al. [17]	Simulation [56]	Sort-Merge
Kepe et al. [35]	Simulation [2]	Hash, Sort-Merge
Santos et al. [16]	Simulation [2]	Bloom
Boroumand et al. [10]	Simulation [56]	Sort-Merge
PID-Join (this work)	**Real system** [12]	**Distributed & Partitioned Hash**, Sort-Merge, Nested-Loop

a PIM kernel. Applications can exploit the rank-level parallelism and utilize the IDPs of multiple ranks, and thus multiple DIMMs, by performing the procedure for the ranks in parallel.

The IDPs of UPMEM DIMMs implement a shared-nothing architecture. Each IDP can only access the data stored in its memory bank by transferring them to its WRAM, and thus cannot directly access the data stored in the other memory banks. As the DIMMs lack native hardware support for direct inter-IDP communication, host CPUs must transfer the data between the memory banks of different IDPs. Such CPU-driven inter-IDP communication consumes limited host CPU bandwidth and incurs notable performance overheads. The performance overheads can become increasingly significant without careful inter-IDP communication optimizations due to the rank-level parallelism of the DIMMs; when transferring data between IDPs, all the other IDPs belonging to the same rank of the source and destination IDPs must transfer data as well as the IDPs of a rank operate in a synchronous manner. Thus, one of the optimization guidelines for efficiently exploiting the DIMMs is to minimize the inter-IDP communication [21].

2 MOTIVATION

Although many prior studies on accelerating in-memory join using PIM exist, we make a key finding that *the prior studies on PIM-assisted join algorithms assume key architectural assumptions which greatly differ from real PIM hardware*. If the prior studies turn out to be incompatible with commodity PIM-enabled DIMMs, it is clear that we need a new PIM-assisted join algorithm not only compatible with the DIMMs, but also fully exploits the architectural characteristics of the DIMMs to maximize the performance of join.

We make another key finding that *the prior studies heavily rely on cycle-level simulators to evaluate their PIM-assisted join algorithms*. As real PIM hardware has not been available for a long time, the simulators have been a de-facto evaluation method for the prior studies. However, even with extensive validation, the simulators are known to frequently incur significant hardware modeling errors, often making the simulation-driven findings invalid on real hardware [24, 25, 45]. For in-memory RDBMSs to positively and aggressively employ PIM-assisted join algorithms in their designs, it is essential to prototype and examine the algorithms on real systems and commodity PIM-enabled DIMMs.

Aimed at verifying the two key findings, we conduct a survey on the existing PIM-assisted join algorithms. The key objective of the survey is to identify whether the prior studies can seamlessly be implemented on commodity PIM-enabled DIMMs. Table 1 summarizes the prior studies, their evaluation methods, and their join algorithms. The survey confirms that the prior studies heavily rely on simulators to prototype and evaluate their join algorithms. The survey also reveals that the following major architectural assumptions of the prior studies prevent them from being compatible with the DIMMs. First, the prior studies assume that in-memory processors are equipped with SIMD units and capable of executing vector and floating-point operations. However, due to the tight chip area and power budgets of the DIMMs, the IDPs of the DIMMs are highly limited in their compute capabilities; the IDPs of commodity UPMEM DIMMs lack native hardware support for vector and floating-point operations [21]. Second, the prior studies assume shared memory and

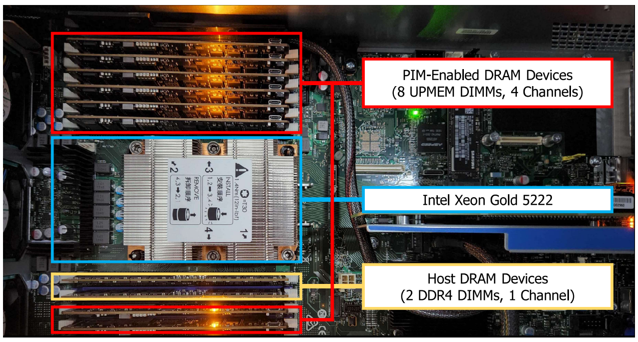


Fig. 3. Our PIM-enabled system equipped with eight UPMEM DIMMs

Table 2. Peak memory bandwidth of the CPU-based baseline and the PIM-enabled systems

	Sequential		Random	
	Read	Write	Read	Write
CPU-Based Baseline System				
(5 standard DIMM channels)				
CPU-to-Memory	32.0 GB/s	31.9 GB/s	9.7 GB/s	9.7 GB/s
PIM-Enabled System				
(4 UPMEM DIMM channels & 1 standard DIMM channel)				
IDP-to-WRAM	2,652.2 GB/s	2,670.6 GB/s	2,099.2 GB/s	2,109.4 GB/s
WRAM-to-Bank	585.7 GB/s	592.8 GB/s	64.4 GB/s	86.0 GB/s
CPU-to-UPMEM	23.3 GB/s	23.3 GB/s	9.2 GB/s	9.1 GB/s
CPU-to-Memory	7.3 GB/s	7.2 GB/s	5.9 GB/s	5.9 GB/s

fast direct communication paths between the in-memory processors. The DIMMs, on the other hand, involve host CPUs and the bandwidth-limited host-DIMM data transfers for performing inter-IDP communication. Third, the prior studies do not take the host-DIMM data transfer overheads into account by assuming that all the input data of a join are already available to the in-memory processors. Unfortunately, as the DIMMs employ an accelerator model and have an independent physical memory space, the CPUs must transfer input and output data between the DIMMs and the main memory. In summary, the prior studies cannot be directly applied to commodity PIM-enabled DIMMs which invalidate the key architectural assumptions of the prior studies.

3 PID-JOIN: A PROCESSING-IN-DIMM JOIN ALGORITHM FOR COMMODITY DIMMS

In this section, we present the design and analysis of *Processing-In-DIMM Join (PID-Join)*, a fast in-memory join algorithm which efficiently exploits PIM-enabled DIMMs. To exploit the architectural characteristics of the DIMMs, we employ a bottom-up approach to derive PID-Join on a real system equipped with UPMEM DIMMs.

3.1 System Configuration

Our PIM-enabled system for designing and evaluating PID-Join, shown in Fig. 3, consists of an Intel Xeon Gold 5222 CPU, one memory channel with two standard DDR4-2400 DIMMs, and four memory channels each of which has two UPMEM DIMMs. The system has eight UPMEM DIMMs, and thus 1,024 IDPs in total, providing a theoretical internal memory bandwidth of 716.8 GB/s and a theoretical computational throughput of 358.4 GOPS [20]. The CPU has been chosen to facilitate fair performance comparisons between CPU- and PID-based join algorithms; the maximum computational throughput of the CPU is 345.6 GOPS [30] which is similar to that of the DIMMs. Making the CPU and the DIMMs achieve similar computational throughput allows us to precisely evaluate the performance benefits of exploiting PIM for in-memory join. The two standard DIMMs serve as the main memory and are necessary as PIM-enabled DIMMs employ an accelerator model. The input data for PIM kernels are initially stored in the main memory, and the main memory retrieves the output data after the IDPs complete executing the kernels.

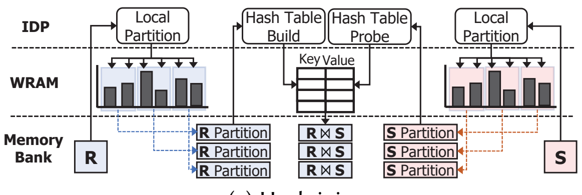
Aimed at demonstrating the high internal memory bandwidth of PIM-enabled DIMMs, we compare the peak memory bandwidth of the PIM-enabled system with that of the CPU-based baseline system. The baseline system is equipped with ten standard DDR4-2400 DIMMs by replacing the eight UPMEM DIMMs attached to the PIM-enabled system with the standard DIMMs. For measuring the peak memory bandwidth, we implement a microbenchmark which sequentially and randomly reads/writes data from/to a large sequentially-allocated memory region at the granularity of the CPU cache line size (64 bytes).

Table 2 shows the peak memory bandwidth of the two systems measured with the microbenchmark. The results clearly demonstrate the high potential of exploiting PIM-enabled DIMMs for accelerating in-memory join; the peak internal memory bandwidth of the UPMEM DIMMs is significantly higher than the peak memory bandwidth of the standard DIMMs. In addition, the results reveal the following performance optimization hints for PID-Join. First, PID-Join should fit its working set size to the limited WRAM capacity as the IDP-to-WRAM bandwidth is much higher than the WRAM-to-bank bandwidth. Second, as random WRAM-to-bank accesses achieve much lower bandwidth than sequential WRAM-to-bank accesses, PID-Join should filter its random accesses out using the WRAMs and perform sequential accesses between the WRAMs and the memory banks. Third, PID-Join should minimize and optimize the data transfer between the CPU and the DIMMs by offloading as many operations as possible to the IDPs as CPU-to-UPMEM and CPU-to-memory accesses are relatively slow.

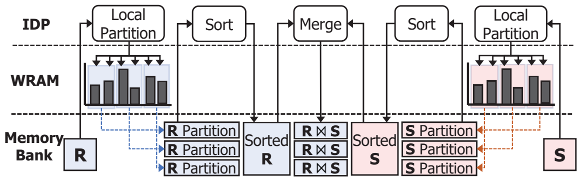
3.2 Single-IDP Join Algorithms

Our first step toward designing PID-Join is to address the question of how to design a join algorithm which fully exploits the high internal memory bandwidth and computational throughput of an IDP. We aim to develop a high-throughput single-IDP join algorithm which performs join on two tables R and S, where $|R| \leq |S|$ without loss of generality, stored in the memory bank. To develop the single-IDP join algorithm, we prototype and evaluate three representative types of equi-join algorithms: hash, sort-merge, and nested-loop joins. After designing the algorithms, we compare their performance with synthetic workloads and select the best-performing algorithm as the single-IDP join algorithm of PID-Join. For the performance comparison, we assume that an IDP processes an equi-join query `SELECT * FROM R, S WHERE R.key = S.key` where `R.key` and `S.key` are the 32-bit integer keys of R and S, respectively. All the keys are initially stored in the memory bank of an IDP, and the resulting tuples of the query get stored in the memory bank.

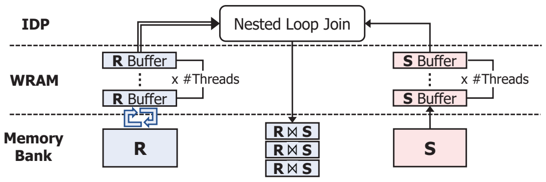
3.2.1 Design Considerations. To achieve high join performance, single-IDP join algorithms should be optimized for the following key architectural characteristics of the IDPs. First, as an IDP can only access the data stored in its WRAM, the algorithms must fit their working set within the 64-KB capacity of the WRAM. If the working set size exceeds the WRAM capacity, then the IDP can suffer from the performance degradation caused by frequent data transfers between the WRAM and the memory bank. Second, the algorithms should transfer large chunks of sequential data, at least 2 KB in size, to maximize the internal memory bandwidth of the IDP. As the DRAM-based memory bank provides higher bandwidth with larger sequential data accesses, the WRAM should be used to filter out small and/or random data accesses. Third, due to the limited compute capabilities of the IDP, the algorithms need to employ the operations which the IDP has native hardware support for (e.g., integer additions). The IDP supports the non-native operations such as integer multiplication/division and floating-point operations through emulation; however, it achieves significantly lower computational throughput with the non-native operations [20].



(a) Hash join



(b) Sort-merge join



(c) Nested-loop join

Fig. 4. Working models of PID-Join's single-IDP hash, sort-merge, and nested-loop join algorithms

3.2.2 Hash Join. Hash join algorithms utilize a hash table to perform join on two tables, namely R and S. The algorithms first build the hash table with the tuples of R. The hash table consists of (key, value) pairs; the key and the value of a pair correspond to the original join key and the tuple ID of a tuple of R, respectively. The algorithms then probe the hash table with a tuple of S, identify the tuple of R in the corresponding hash table entry, and examine whether the two tuples satisfy the given join predicate(s).

Our single-IDP hash join algorithm employs the following key ideas to fully exploit the architectural characteristics:

- **WRAM-pinned hash table:** The algorithm pins its hash table to the WRAM as the entries of the hash table are frequently accessed. Pinning the hash table to the WRAM prevents the algorithm from incurring the slow memory-bank accesses.
- **Local partitioning:** The algorithm performs local partitioning on R regarding the limited capacity of the 64-KB WRAM, and then processes each partition of R by building and probing a hash table for the partition. Building a single hash table containing all the tuples of R would make the hash table get spilled to the memory bank and incur slow random memory-bank accesses.
- **IDP-friendly hash function:** The algorithm employs an IDP-friendly hash function whose operations are natively supported by an IDP. Avoiding the use of slow non-native operations allows the algorithm to achieve high computational throughput.
- **Linear probing:** The algorithm uses linear probing as its collision handling method. There exist various alternative methods (e.g., cuckoo hashing); however, linear probing requires no additional storage, making it suit well to the limited WRAM capacity.

Based on the key ideas, the single-IDP hash join algorithm operates as follows (Fig. 4a). Initially, all the tuples of R and S are stored in the memory bank. The algorithm first performs local partitioning on R by applying an IDP-friendly hash function on the tuples of R, counting the number of tuples per hash value, and grouping the tuples whose hash values have the same radix bits into the same partition. The algorithm groups the tuples into the partitions in a way that each of the partitions

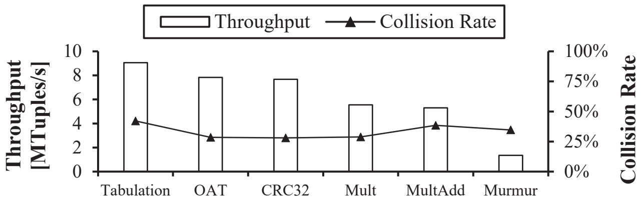


Fig. 5. Single-IDP performance of various hash functions

induces a 32-KB hash table having a fill rate of 50%. Targeting the fill rate of 50% is known to avoid excessive collisions and has been employed by prior studies on hash join [55]. The algorithm also partitions S in a similar manner. Then, the algorithm loads a partition of R and a partition of S, builds a WRAM-pinned hash table using the tuples of R, and probes the hash table with the tuples of S. If a collision occurs when building the hash table, the algorithm utilizes linear probing to handle the collision without enlarging the hash table. After that, the resulting tuples, the pairs of R and S tuples having the same key value, get stored to the memory bank. The algorithm repeats the hash table build and probe for the remaining partitions.

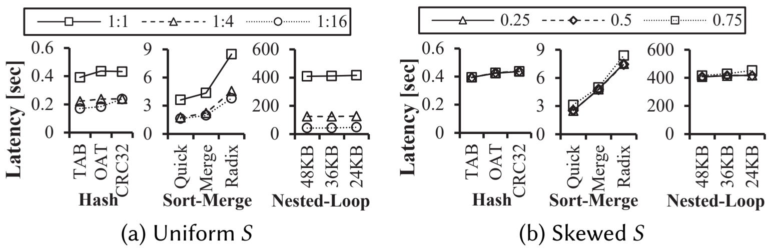
To achieve high computational throughput, the algorithm should use a hash function whose operations are natively supported by an IDP. Fig. 5 shows the hashing throughput and the collision rates of the evaluated hash functions. The hash functions utilizing integer multiplications, i.e., multiply-shift (Mult) [14], multiply-add-shift (MultAdd) [14], and Murmur [3], achieve low hashing throughput due to the IDP's lack of native hardware support for the multiplications. The other three hash functions, i.e., tabulation [52], once-at-a-time (OAT) [33], and CRC32 [64], utilize XOR-based operations, and thus achieve much higher hashing throughput. We also compare the collision rates of the hash functions by measuring the number of hash collisions occurred while building and probing a hash table with 128-K randomly-generated tuples. OAT and CRC32 achieve lower collision rates than tabulation; however, additional measurements reveal that the higher hashing throughput of tabulation overcomes its higher collision rate, making tabulation the fastest. We show a sensitivity study on how the algorithm's performance differs with different hash functions in Section 3.2.5.

3.2.3 *Sort-Merge Join.*

Sort-merge join algorithms first sort the tuples of R and S, and then merge the tuples by evaluating join predicate(s) [5]. Using sequential data accesses and fast tuple sorting algorithms, sort-merge join algorithms have been shown to outperform hash join algorithms under various scenarios [17, 37].

Our single-IDP sort-merge join algorithm uses the following key ideas regarding the architectural characteristics of an IDP:

- **Range partitioning:** To fit its working set within the WRAM, the algorithm employs range partitioning which groups the tuples having the same range of join key values. Then, the algorithm sorts the tuples of one partition at a time.
- **Parallel in-place sort:** As range partitioning makes different partitions have distinct join key value ranges, sorting the tuples of one partition is independent from the tuples of the other partitions. The algorithm exploits this to parallelize the sorting of the tuples using the multithreading support of an IDP. Different threads of the IDP are assigned different partitions, and then the threads concurrently sort their assigned partitions. The algorithm employs an in-place sorting algorithm to prevent its working set size from exceeding the WRAM capacity.
- **Parallel merge:** The algorithm exploits multithreading for merging the tuples of R and S. The algorithm loads multiple R and S partitions to the WRAM and merges the tuples of one S partition with all the R partitions by assigning the R partitions to different threads of the IDP.



(a) Uniform S (b) Skewed S

Fig. 6. Join execution latency of our three single-IDP join algorithms with varying design choices

Fig. 4b shows the working model of our single-IDP sort-merge join algorithm. First, the algorithm performs range partitioning on the tuples of R and S which are initially stored in the memory bank. Then, the algorithm sorts the tuples belonging to the same partition using an in-place sorting algorithm and stores the sorted tuples to the memory bank. After that, the algorithm loads multiple R and S partitions to the WRAM, performs parallel merging on the loaded partitions, and stores the resulting tuples to the memory bank. For each of the S partitions, the algorithms make different threads of an IDP merge the S partition with different R partitions. The algorithm employs quick sort;we present a sensitivity study on the in-place sorting algorithm in Section 3.2.5.

3.2.4 *Nested-Loop Join.*

Despite their high time complexity, nested-loop join algorithms are employed by many modern RDBMSs [23, 29, 46]. The algorithms iterate and evaluate join predicate(s) on all the pairs of the R and S tuples using a nested loop.

To maximize the memory-bank bandwidth, our single-IDP nested-loop join algorithm builds on top of the following key idea:

- **Adaptive WRAM Management:** Depending on whether all the tuples of R can fit within the WRAM, the algorithm allocates different amounts of WRAM to the buffers which temporarily store the tuples of R. The key objective is to minimize the amount of data transfers between the WRAM and the memory bank. If all the tuples of R fit within the WRAM, the algorithm allocates a WRAM buffer whose capacity matches that of the tuples, loads all the tuples of R to the buffer, and utilizes the remaining WRAM buffer for streaming the tuples of S. Otherwise, the algorithm allocates most of the WRAM for storing the tuples of S, and uses the remaining WRAM as a circular buffer for the tuples of R.

Fig. 4c depicts how the single-IDP nested-loop join algorithm works. The outer loop scans the tuples of S, and the inner loop scans the tuples of R. The algorithm first inspects the amount of WRAM buffer needed for storing all the tuples of R, and allocates one WRAM-pinned circular buffer for storing the tuples to each thread of an IDP. The threads then load a series of the S tuples to the WRAM, populate their circular buffers with the R tuples, and evaluate join predicates on the R and S tuples. After scanning all the R tuples, the algorithm moves on to the next series of the S tuples and repeats the process. All the accesses to the tuples of R and S are sequential, allowing the algorithm to achieve fast data transfers between the WRAM and the memory bank on the IDP.

3.2.5 *Comparison of the Three Single-IDP Join Algorithms.*

To identify the best-performing single-IDP join algorithm for PID-Join, we compare the join execution latency of the three single-IDP join algorithms on a single IDP. For the comparison, we configure S to have 256 K tuples, vary the $|R| : |S|$ ratio from 1:1 to 1:16 (for uniform S), and adjust the Zipf factor of S from 0.25 to 0.75 (for skewed S). R is configured to have a uniform data distribution for all the experiments. We also evaluate the impact of the per-algorithm design choices: the hash function for the hash join, the in-place sorting algorithm for the sort-merge join, and the size of the WRAM buffer for storing the S tuples for the nested-loop join.

Fig. 6 shows the measured join execution latency of the three single-IDP join algorithms. Overall, the hash join algorithm is the fastest across all the experiments. It achieves the highest performance due to its efficient use of the WRAM and minimal data transfers between the WRAM and the memory bank. The other join algorithms, on the other hand, suffer from frequent memory-bank accesses.In terms of the hash function, employing tabulation (TAB) achieves faster join processing than using OAT or CRC32. Although tabulation achieves a higher collision rate than OAT and CRC32, its higher hashing throughput mitigates the penalty of the higher collision rate. Therefore, PID-Join employs the single-IDP hash join algorithm and tabulation.

Due to the limited WRAM capacity, our single-IDP join algorithms employ in-place algorithms which do not dynamically increase the working set sizes. Out-of-place algorithms may be promising alternatives to the in-place algorithms; however, they involve dynamic memory allocation and are likely to increase the working set sizes to exceed the WRAM capacity. Due to the limited WRAM capacity and its associated overheads, we envision the in-place algorithms surpass the out-of-place algorithms, and thus leave the study of employing the out-of-place algorithms as future work.

3.3 Processing-in-DIMM Partitioning

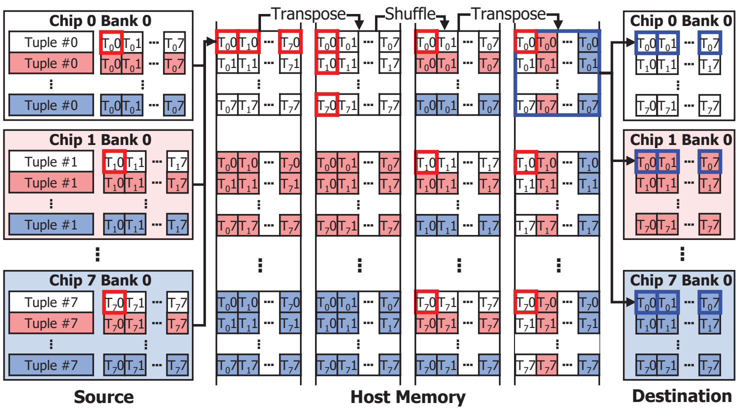
To fully exploit the high potential of PIM-enabled DIMMs, it is necessary to scale the single-IDP hash join algorithm to at least an entire rank consisting of 64 IDPs. The rank-level scaling inevitably leads to a partitioning problem. The single-IDP join algorithms assume that the entire R and S are stored within a memory bank. To expand to an entire rank, however, the tables have to be partitioned first in a way that each IDP processes only a single pair of R and S partitions and no dependency exists between the partitions.

Such a partitioning phase is commonly found in join algorithms [1, 6], but is especially crucial to PID-Join due to the shared-nothing architecture of the IDPs.The partitioning phase, often referred to as global partitioning, distributes the tuples of two tables to the IDPs such that all the tuples which fall into the same hash key range belong to the same IDP. Although efficient partitioning itself is a research topic of its own [5, 7, 60, 68], it is widely known that global partitioning is highly memory-bound [6, 7, 11, 61], which makes itself another great application for PIM-enabled DIMMs.

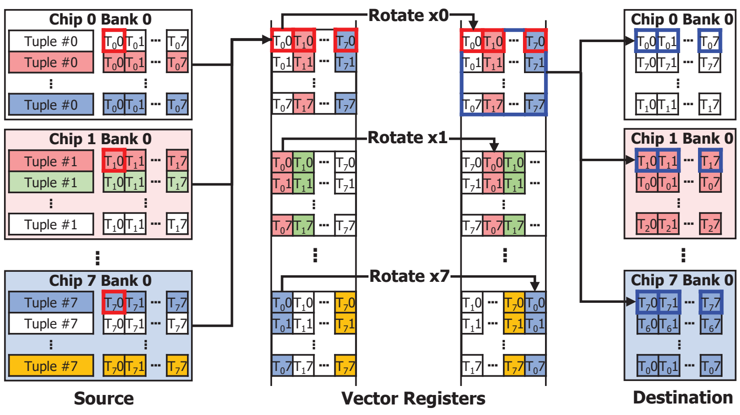
3.3.1 Design Considerations. Due to the shared-nothing architecture of the IDPs, the intervention of host CPUs is inevitable for inter-IDP communication. This characteristic incurs a tremendous amount of overhead and becomes a performance bottleneck. Thus, extra care has to be placed to reduce the amount of the communication, and to fully utilize the bandwidth of the DRAM channels.

3.3.2 Conventional All-to-All Inter-IDP Communication. The key challenge toward fast inter-IDP communication is the data layout of DIMMs as described in Sec. 1.1. Due to the byte-wise interleaving among eight chips in a rank, host CPUs are unable to interpret the raw data fetched from PIM-enabled DIMMs. To solve the problem, the existing kernel modules for the DIMMs perform a byte-level transpose over every received cache line on the CPUs. Fig. 7a depicts the working model of the conventional inter-IDP communication (i.e., UPMEM SDK) involving the byte-level transpose. First, the CPUs transfer the data from the individual banks of a PIM-enabled DIMM to the host memory and perform the byte-level transpose. Second, the CPUs shuffle the data in the host memory, which essentially place the data to the locations corresponding to the destination memory banks. Lastly, the CPUs transfer back the data to the DIMM, where the byte-level transpose is applied again, and the data get stored in their the destination memory banks.

Unfortunately, the conventional inter-IDP communication suffers from low bandwidth. First, the byte-level transpose places a lot of computational burden on the CPUs and becomes a performance bottleneck. Second, the data unnecessarily get written to the host memory; the data make an additional round-trip through the host memory, increasing the latency and reducing the bandwidth.



(a) Conventional inter-IDP communication



(b) PID-Join's Rotate-and-Stream (RnS)

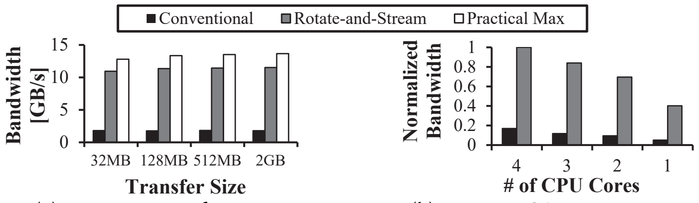
Fig. 7. Working models of the conventional and PID-Join's inter-IDP communication methods. The colors of the blocks indicate the tuples' destination banks.

3.3.3 *Rotate-and-Stream.*

We make two key observations to tackle the high latency and low bandwidth of the conventional inter-IDP communication. First, the data transferred by the inter-IDP communication do not need to be fully loaded into the host memory. For any inter-IDP communication, the source and destination are both the memory banks of PIM-enabled DIMMs. The host memory merely acts as a buffer for the CPUs to perform shuffling, and the shuffling can occur over gigabytes of data. Second, the data do not need to be interpretable by the CPUs. Unlike arithmetic operations (e.g., addition) which definitely require the interpretation by the CPUs, the inter-IDP communication only requires data relocation which can be replaced with bitwise operations. This implies a potential for reducing the CPUs' burden for the byte-level transpose.

Based on our observations, we present *Rotate-and-Stream (RnS)* which eliminates the need for the host memory and replaces the complex shuffling operations with a single bit-rotation operation. Slightly tweaking the partitioning behavior becomes the trick for dramatically reducing the CPUs' burden to a single rotation. In addition, the rotation occurs within a single cache line. This allows keeping the data for inter-IDP communication only in a register and not sending it to the host memory, and thus eliminates the need for host memory accesses and enables vector streaming.

Fig. 7b illustrates our RnS with an example of eight IDPs each sending and receiving eight 8-byte tuples; a single tuple is sent and received between two IDPs. During the partitioning phase, each IDP constructs the partitioning bins in a twisted order. In contrast to the conventional inter-IDP



(a) Varying transfer sizes (b) Varying CPU core counts

Fig. 8. All-to-all inter-IDP communication bandwidth of the conventional inter-IDP communication and RnS

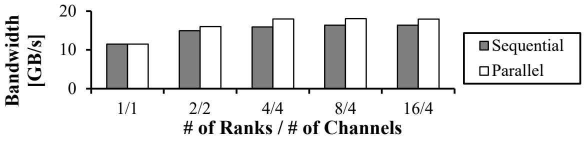


Fig. 9. RnS bandwidth with varying rank counts

communication where the partitioning bins are always ordered by the destination chip ID starting from 0 (Fig. 7a), for RnS, it is done by placing the bins destined to itself first and then in the increasing order of destinations. For example, the partitions in chip 0 are stored in the order of 0, 1, 2, ..., 7, and the partitions in chip 7 are stored in the order of 7, 0, 1, ..., 6. Then, a single 64-byte cache line-sized data block is read from a PIM-enabled DIMM without the byte-level transpose. The cache line contains the first eight bytes from each of the eight chips, but in a byte-interleaved manner which is depicted as same colored tuples being vertically aligned. After that, performing bitwise-rotation(s) on each of the 64-byte blocks (N bytes for N-th chip) completes the data layout identical to that of the conventional inter-IDP communication after the transpose-shuffle-transpose sequence in Fig. 7a. Lastly, simply transferring the data to the PIM-enabled DIMM achieves the inter-IDP communication.

As the rotation happens within 64-byte cache lines, it can fit into a 512-bit SIMD register. This allows for an efficient rotation using vector instructions, and more importantly, eliminates the need for the host memory. In addition, the cache access overhead can be minimized by streaming to the cache with non-temporal streaming operations [31]. We modify and extend the existing UPMEM SDK [65] to implement RnS. Since UPMEM SDK follows an accelerator model, it provides only DIMM-to-host and host-to-DIMM data transfers which perform in-flight vector transpose. Instead, our implementation of RnS loads 64-byte data from a PIM-enabled DIMM using the non-temporal stream load instruction `_mm512_stream_load_si512()` which does not pollute the host CPU caches. The rotates are performed in-place using the `_mm512_rol_epi64()` instruction. Then, when the data get written back to the DIMM, we use the stream store operation `_mm512_stream_si512()` which directly accesses the DIMM without writing the data into the cache.

Fig. 8 shows the all-to-all inter-IDP communication bandwidth of RnS in comparison with the conventional inter-IDP communication. Fig. 8a shows the bandwidth by varying the aggregate amount of data transferred between the 64 IDPs of a single rank from 32 MB to 2 GB. In terms of bandwidth, RnS outperforms the conventional method by 6.23× in geometric mean as the amount of computation for host CPUs is reduced and the data are not spilled to the host memory. Fig. 8b shows how the performance degrades when the number of the available CPU cores is reduced, normalized to the bandwidth with four cores of each method. RnS is less sensitive to the available computational resources than the conventional inter-IDP communication, again as RnS relies less on the CPUs.

To perform RnS, each IDP must prepare the data to transfer in the units of packets. As eight banks are interleaved in a 64-byte cache line, the minimum packet size becomes 8 bytes. While we empirically use 8-byte packets, we observe that the performance of RnS is insensitive to the packet size. We present a detailed sensitivity study on the packet size of RnS in Section 4.3.1.

3.3.4 Scaling Out to Multiple Ranks & DIMMs. We previously proposed RnS for performing inter-IDP communication within a single rank. However, to fully utilize PIM-enabled DIMMs, the single-rank inter-IDP communication has to be extended to multiple ranks and DIMMs. We observe that a simple extension of the rank-wise working model of RnS to multiple ranks and DIMMs works greatly for multi-rank/DIMM scaling: the whole all-to-all inter-IDP communication is split into multiple rank-to-rank jobs, where each of the jobs can be independently performed to complete the whole communication. There is a design choice of host CPU job allocation, where the jobs can be executed sequentially utilizing all host CPU threads for each job, or the jobs can be executed in parallel where a single thread is allocated for each job. We measure the bandwidth of the two methods in Fig. 9, with multiple ranks in the system. Throughout various settings, there is small but clear advantage to the parallel approach, and we take the parallel RnS approach as PID-Join's all-to-all inter-IDP communication method.

3.4 Optimizing Host-DIMM Data Transfers

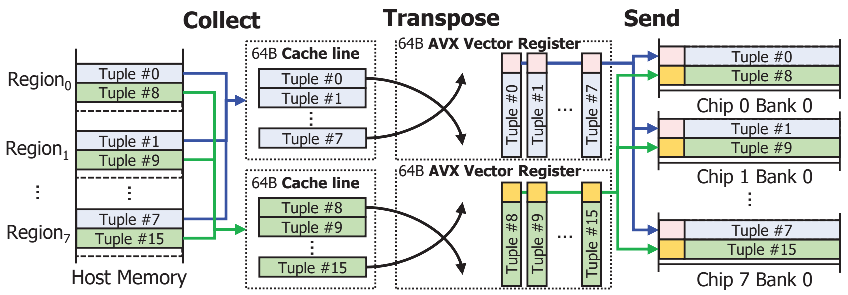
As PIM-enabled DIMMs follow an accelerator model, there are two instances of host-DIMM data transfers: sending the input (i.e., R and S tuples) to the DIMMs and retrieving the output (i.e., $R \bowtie S$) from the DIMMs. Such host-DIMM data transfers become a significant performance bottleneck, especially after RnS has been applied. Unfortunately, the key ideas from RnS are both infeasible to be applied to the host-DIMM data transfers; the input and output should be interpretable by host CPUs, and there is no intermediate buffer such as host CPU caches that can be streamed or bypassed. Thus, we design an optimization for the host-DIMM data transfers.

3.4.1 Design Considerations. There are two architectural characteristics to consider for accelerating the host-DIMM data transfers. First, as the host memory and PIM-enabled DIMMs both follow the DDR protocol, it is the best to have sequential access patterns, often at least in a 64-byte granularity. While sequentially accessing larger chunks of data would further increase the bandwidth, accessing data chunks smaller than 64 bytes suffer from severe bandwidth degradation due to the minimum burst length of the DIMMs [4]. Second, due to the data layout of the DIMMs, a consecutive 64-byte block of data in the host memory will be transferred to eight different IDPs. The new host-DIMM data transfer method should embrace this pattern to maximize the bandwidth.

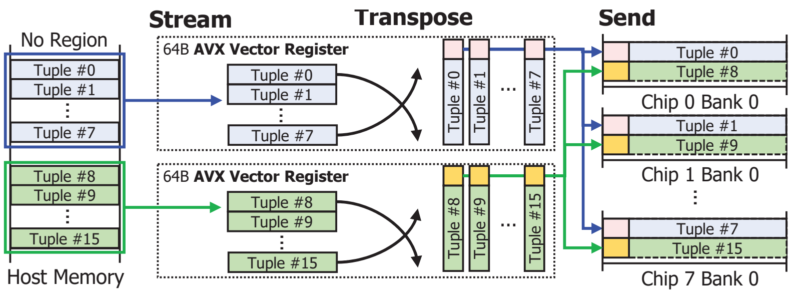
3.4.2 Conventional Host-DIMM Data Transfers. Conventional IDP-wise host-DIMM data transfers allow precise mapping of the data to their destination IDPs; however, they suffer from low bandwidth due to the small, random host-memory accesses necessary for correctly transferring a tuple to its destination IDP. Fig. 10a depicts the conventional host-to-DIMM data transfers. First, the tuples stored in the host memory get partitioned into different contiguous memory regions depending on their target IDPs. Then, host CPUs gather an eight-byte tuple from each of eight different regions to form a 64-byte block. After that, the CPUs perform byte-level transpose and send them to the memory banks, where each of the eight-byte tuples arrive at individual destination banks.

Unfortunately, the gather phase consumes a huge amount of host CPU bandwidth due to the eight-byte random reads from the host memory. While there is a library code that allows sending a contiguous block of data to a single bank in a DIMM, it is known to achieve low bandwidth due to the data layout of DIMMs [20].

3.4.3 Rank-Wise Unordered Scatter-Gather. Our key observation is that no strict ordering is required to the input and output tuples in the memory banks of PIM-enabled DIMMs as PID-Join's global



(a) Conventional IDP-wise host-DIMM data transfer



(b) PID-Join's rank-wise Unordered Scatter-Gather (USG)

Fig. 10. Working models of the conventional IDP-wise and PID-Join's rank-wise host-DIMM data transfers

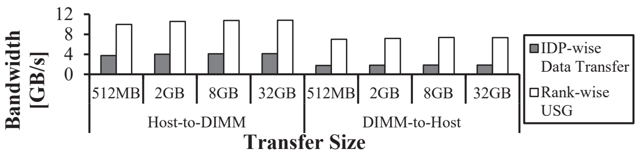


Fig. 11. Host-DIMM data transfer bandwidth of the IDP-wise data transfers and PID-Join's rank-wise USG

partitioning will place them to the right IDPs. To PID-Join, the only concern for the initial host-to-DIMM data transfer is the even distribution of the R and S tuples across all the IDPs.

Based on the key observation, PID-Join implements *rank-wise Unordered Scatter-Gather (USG)*, a mechanism that evenly distributes the tuples of R and S to the IDPs without requiring random accesses to the host memory. Fig. 10b shows the working model of USG (only the scatter phase shown). USG removes unnecessary small loads from multiple regions of the host memory by directly loading 64 bytes of tuples in a streamed manner into a 512-bit vector register. Then, simply performing byte-level transpose and then sending them to a PIM-enabled DIMM completes a transfer. The conventional IDP-wise data transfers suffer from their restriction that they must transfer the tuples to their exact destination IDPs. On the other hand, USG does not involve any region splitting. As a result, USG performs sequential accesses to both the host memory and the PIM-enabled DIMM, while the tuples get evenly distributed to the IDPs.

Similar to RnS, we extend UPMEM SDK to implement USG. Instead of targeting 64 memory banks at once, we split them into eight groups by reverse-engineering the physical address mapping of UPMEM DIMM to identify which of the eight memory banks share the same DRAM bank identifier but reside on different chips. USG gets a table as input, and transfers the tuples using the non-temporal load/store instructions of Intel AVX-512 [31].

Fig. 11 shows the host-DIMM data transfer bandwidth of the conventional IDP-wise data transfer and rank-wise USG. The results show that USG greatly outperforms the conventional IDP-wise data transfer in terms of host-DIMM data transfer bandwidth. The bandwidth is slightly higher

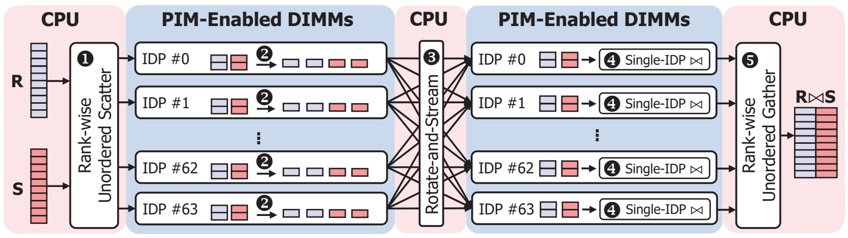


Fig. 12. End-to-end working model of PID-Join

than the DIMM-to-host bandwidth. It is due to the fact that the host-to-DIMM data transfer utilizes asynchronous AVX write instructions, but the DIMM-to-host data transfer involves synchronous AVX read instructions [31]. The synchronous read instructions prevent the DIMM-to-host data transfer from sustaining as many memory bank accesses as the host-to-DIMM data transfer, resulting in lower DIMM-to-host bandwidth [20].

3.5 Putting It All Together

Fig. 12 illustrates an end-to-end working model of PID-Join with 64 IDPs (a single rank). Using its single-IDP hash join algorithm, RnS, and rank-wise USG, PID-Join performs join on two tables R and S using PIM-enabled DIMMs as follows. First, ❶ PID-Join exploits USG to evenly distribute the tuples of R and S to the DIMMs. Second, ❷ the IDPs perform global partitioning on their R and S tuples and group the tuples of each partition to form the packets for RnS. Third, ❸ PID-Join performs all-to-all inter-IDP communication using RnS, making the R and S tuples having the same hash key values get transferred to the same IDP. Fourth, ❹ the IDPs execute the single-IDP hash join algorithm on their R and S tuples and produce the resulting tuples of the join. Lastly, ❺ PID-Join transfers the resulting tuples to the host memory using USG.

4 EVALUATION

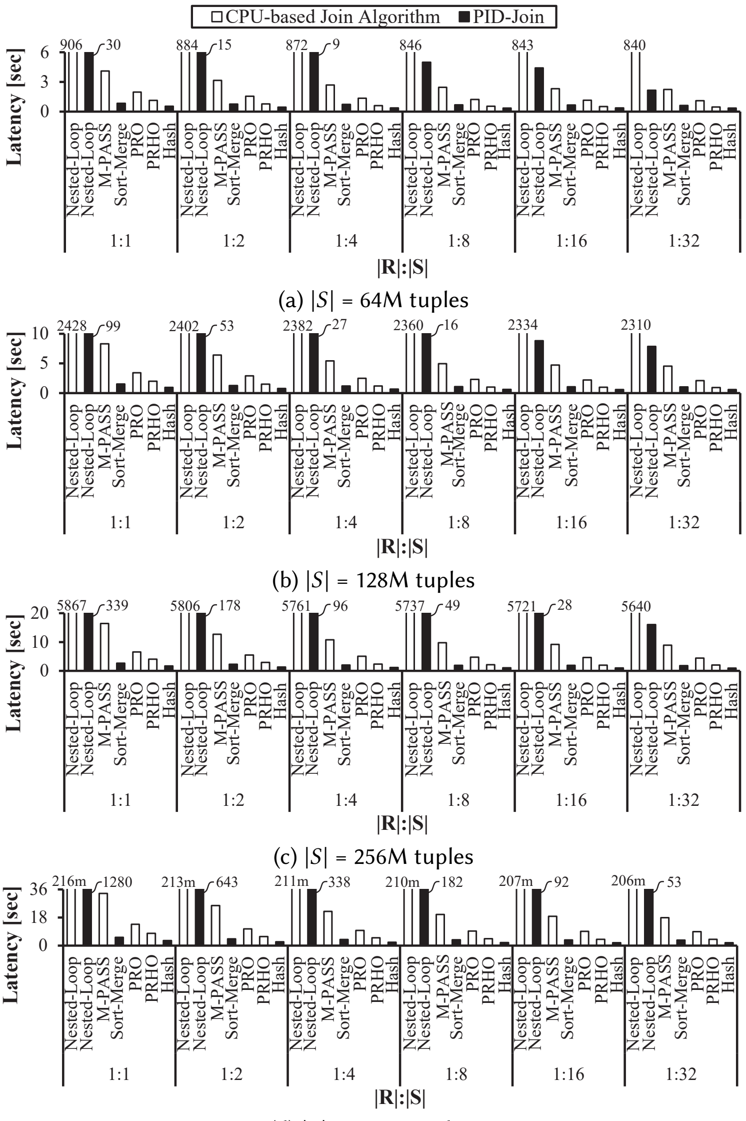
4.1 Experimental Setup

We evaluate PID-Join by comparing its join execution latency on our PIM-enabled system against that of the existing CPU-based join algorithms [5, 6, 29] on the CPU-based baseline system. For fair comparisons, we configure the two systems to have the same CPU and the DDR4 channel count; the baseline system has five channels, each with two standard DDR4-2400 DIMMs, and the PIM-enabled system has four UPMEM DIMM channels and one standard DIMM channel.

For the workloads, we use a set of workloads derived from various prior studies on accelerating in-memory join [41, 48, 57, 63, 66]. We measure the execution latency of join with two tables R and S having varying tuple counts and Zipf factors. Each of the tuples consists of a 32-bit integer join key and a 32-bit tuple ID. The workloads consist of six $|R| : |S|$ ratios (1:1, 1:2, 1:4, 1:8, 1:16, and 1:32) and four sizes of S (64-M, 128-M, 256-M, and 512-M tuples). We also evaluate the impact of skewed tables by configuring the Zipf factor of S to be 0.00, 0.25, 0.50, and 0.75, which are typical Zipf factors used for evaluating the join performance [5, 48, 58, 59].

4.2 Fast In-Memory Join Executions

4.2.1 Join Execution Latency with Uniform Tables. We first evaluate the execution latency of PID-Join compared to CPU-based join algorithms using uniform R and S. Each As the CPU-based join algorithms, we select SQLite [29] for nested-loop join, M-PASS [5] for sort-merge join, and PRO [6] and PRHO [5] for hash join. For PID-Join, we include the single-IDP sort-merge and nested-loop join algorithms in addition to the single-IDP hash join algorithm. RnS and USG are applied to all the three single-IDP join algorithms.



(a) $|S|$ = 64M tuples

(b) $|S|$ = 128M tuples

(c) $|S|$ = 256M tuples

(d) $|S|$ = 512M tuples

Fig. 13. Join execution latency of CPU-based join algorithms and PID-Join with uniform R and S

Fig. 13 shows the execution latency of the CPU-based join algorithms and PID-Join with uniform R and S and varying $|R| : |S|$ ratios. The results show that PID-Join outperforms the CPU-based join algorithms in all the workloads by efficiently exploiting the high internal memory bandwidth and computational throughput of the IDPs. On the other hand, the CPU-based join algorithms suffer from the limited memory bandwidth in the course of transferring data between host CPUs and the main memory. In the case of PID-Join employing the single-IDP hash join algorithm, which is the fastest among all the evaluated join algorithms, it achieves a geometric mean speedup of 1.92× over PRHO. As the hash join algorithm performs the best for the PIM-enabled DIMMs, further experiments use the hash join algorithm as PID-Join's join algorithm. We also choose PRHO, the fastest algorithm among the CPU-based join algorithms, for the further experiments.

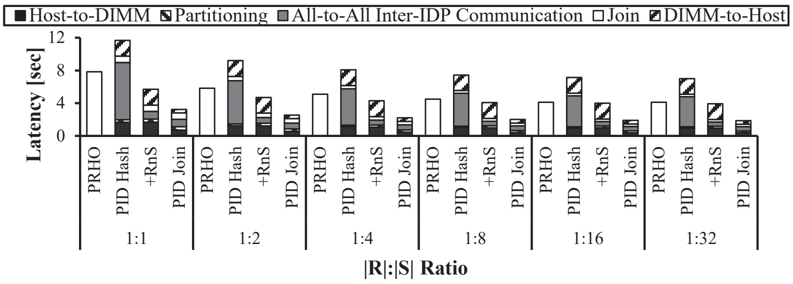


Fig. 14. Breakdown of PID-Join's join execution latency

As $|S|$ increases, the speedup of PID-Join over the CPU-based join algorithms increases. For example, when $|S|$ is 512 M tuples (Fig. 13d), PID-Join achieves a geometric mean speedup of 2.27× over PRHO, whereas the speedup is 1.6× with 64 M tuples (Fig. 13a). The reason is that as $|S|$ becomes larger, the PIM kernel invocation overhead gets amortized. In addition, the host-DIMM bandwidth slightly improves due to larger sequential accesses and the load between the chips of a rank gets more balanced.

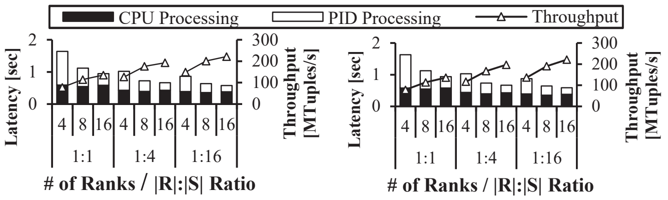
The large speedups of PID-Join are mainly due to its fast hash table builds and probes using the PIM-enabled DIMMs. As shown in Table 2, the CPU-to-memory bandwidth offered by the standard DIMMs (63.9 GB/s) are higher than the CPU-to-UPMEM bandwidth (46.6 GB/s); however, the large performance benefits of parallel hash table build and probe by exploiting the 1,024 IDPs and high internal memory bandwidth (592.8 GB/s) of the PIM-enabled DIMMs offset the lower CPU-to-UPMEM bandwidth. PRHO, on the other hand, gets bottlenecked by the CPU-to-memory bandwidth and the slow random main-memory accesses (19.4 GB/s).

4.2.2 The Effects of PID-Join's Key Ideas. Fig. 14 shows how each of PID-Join's key ideas contributes to the performance improvement along with the latency breakdown. We set $|S|$ to 512 M tuples and use $|R| : |S|$ ratios from 1:1 to 1:32. When we apply only the single-IDP hash join algorithm (PID Hash), a geometric mean slowdown of 1.62× is observed due to the high inter-IDP and host-DIMM data transfer overheads which account for 92.4% of the total latency on average. The inter-IDP communication and the host-DIMM communication achieve only 1.79 GB/s and 1.77 GB/s, respectively, incurring significant latency. Applying RnS (+RnS) optimizes the all-to-all inter-IDP communication and achieves a geometric mean speedup of 1.15× over PRHO. Applying rank-wise USG (PID-Join) then optimizes the host-DIMM data transfers, achieving a geometric mean speedup of 2.27× over PRHO.

As the $|R| : |S|$ ratio becomes higher, both PRHO and PID-Join incur lower latency due to smaller R. Nonetheless, the speedup remains relatively constant: a geometric mean speedup of 2.43× for 1:1 and a geometric mean speedup of 2.21× for 1:32. The small difference comes from the PIM kernel invocation overhead which remains constant due to invoking the same number of PIM kernels.

4.2.3 Scalability Analysis. We evaluate the scalability of PID-Join by examining how its execution latency changes with respect to the number of UPMEM ranks attached to the PIM-enabled system. PID-Join should achieve lower latency with more ranks by efficiently utilizing all the available PIM resources. The number of PIM-enabled channels is fixed at four, and thus the number of ranks for each channel increases from one to four. We set $|S|$ as 128 M tuples and use six combinations of $|R| : |S|$ ratios and Zipf factors for S.

Fig. 15 shows the execution latency of PID-Join with rank counts of 4, 8, and 16. The results show that PID-Join scales well in general, which is expected as each IDP is independent and the single-IDP join algorithm gets executed in parallel by the IDPs. The CPU processing latency, which refers to



(a) Zipf factor = 0.0 (b) Zipf factor = 0.5

Fig. 15. Join execution latency of PID-Join with varying rank counts and $|R| : |S|$ ratios

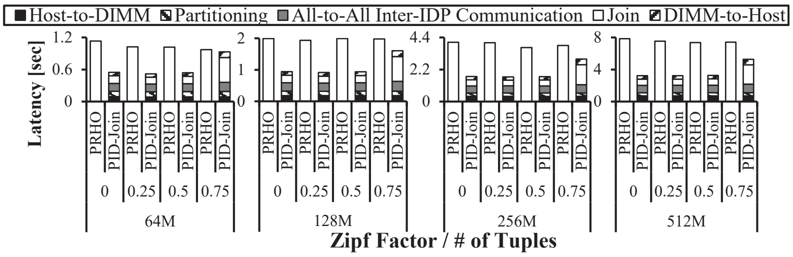


Fig. 16. Join execution latency of PRHO and PID-Join with varying Zipf factors of S. $|R| = |S|$.

the latency incurred by the host CPU for performing inter-IDP communication and host-DIMM data transfers, becomes a performance bottleneck. As the number of ranks increases, the amount of work performed by the CPU increases; however, the amount of CPU bandwidth remains constant. With skewed S, PID-Join achieves 1.63× higher speedup in geometric mean with 16 ranks compared to utilizing only 4 ranks.

4.2.4 Robustness to Skewed Tables. We evaluate the robustness of PID-Join with skewed tables by adjusting the Zipf factor of S from 0 to 0.75. PID-Join performs global partitioning with hashing, so high skewness may cause load imbalance between the IDPs. Due to a constraint of UPMEM DIMMs that all IDPs must be synchronized, PID-Join's performance gets bounded by the slowest IDP.

Fig. 16 shows the execution latency of PRHO and PID-Join with $|R| = |S|$ and varying sizes and Zipf factors of S. The results show that PID-Join is robust to skewed tables except for the Zipf factor of 0.75. However, with the Zipf factor of 0.75, the execution latency of PID-Join remains lower than that of PRHO. On the other hand, the execution latency of PRHO slightly decreases with larger Zipf factors. This is due to the increase in the redundant tuples within S; the redundant tuples will produce the same hash value and refer to the same entry in a hash table. To PID-Join, the redundant tuples cause imbalance between the IDPs as global partitioning assigns the redundant tuples to the same IDP. We expect that such load imbalance between the IDPs can be mitigated with skew handling methods (e.g., salting [67]) which alter redundant tuples to produce different hash values.

4.2.5 System Cost. We now compare the system cost of PID-Join against that of CPU-based PRHO. For the comparison, we first collect and add up the manufacturer's suggested retail prices (MSRPs) of our PIM-enabled and CPU-based baseline systems. After that, we obtain the system cost per performance of the two join algorithms by dividing their corresponding system costs with their peak join processing throughput (i.e., tuples/s) achieved in our experiments. Our approach is similar to that used in a prior study [44] which compares the costs of CPU-based and PIM-enabled systems.

Our comparison shows that PID-Join achieves 26.9% lower system cost per performance than PRHO. PID-Join achieves up to 158.4 M tuples/s on the PIM-enabled system which costs $4,020; the PIM-enabled system consists of eight UPMEM DIMMs ($300 each), two standard DIMMs ($60 each), and the CPU ($1,500). The baseline system costs $2,100 as it is equipped with the CPU ($1,500) and

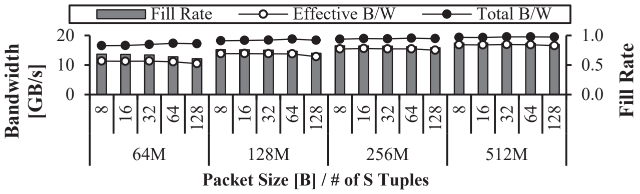


Fig. 17. All-to-all inter-IDP communication bandwidth of RnS with varying packet sizes

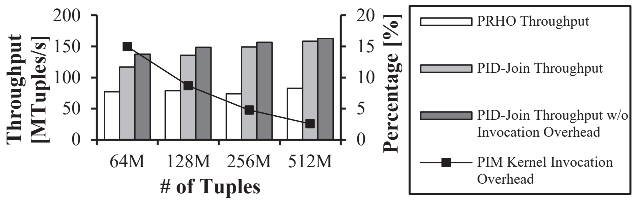


Fig. 18. Join throughput of PRHO and PID-Join with uniform R and S ($|R| = |S|$) and varying R and S sizes

ten standard DIMMs ($600); however, PRHO on the baseline system achieves up to 65.2 M tuples/s. The result shows that, despite the additional cost incurred by UPMEM DIMMs, PID-Join can be an attractive choice to RDBMSs over the CPU-based join algorithms.

4.3 Sensitivity Studies

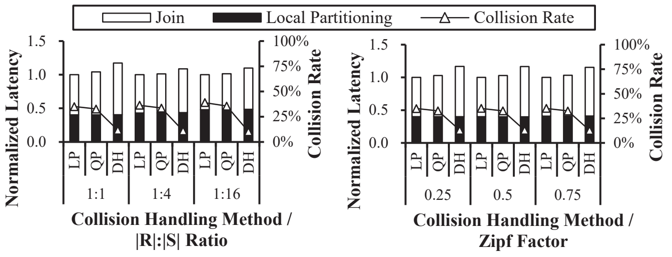
4.3.1 Impact of Rotate-and-Stream's Packet Size. To study the impact of the packet size of RnS on the all-to-all inter-IDP communication bandwidth, we measure the total bandwidth, the effective bandwidth, and the fill rate with five packet sizes (8, 16, 32, 64, and 128 bytes) on various sizes of S (64 M, 128 M, 256 M, and 512 M tuples). The total bandwidth is measured as the bandwidth including the null packets, whereas the effective bandwidth excludes the null packets; the fill rate is the ratio between the two bandwidth values.

Fig. 17 shows how the bandwidth of RnS changes with respect to the packet size. As mentioned in Section 3.3.3, RnS achieves fast inter-IDP communication using eight-byte packets (64 bytes for a rank containing eight IDPs). Ideally, the total bandwidth should remain steady with any packet size larger than eight bytes as larger packet sizes benefit from sequential memory-bank accesses; however, with larger packet sizes, the effective bandwidth slightly decreases due to the decreased fill rates. Based on the results, we employed eight-byte packets for RnS in all the experiments.

4.3.2 PIM Kernel Invocation Overheads. We analyze the influence of PIM kernel invocation overheads on the throughput of PID-Join. Every PIM kernel invocation incurs a latency overhead of about 12 ms on our PIM-enabled system. As PID-Join invokes different PIM kernels for partitioning, hash table build, and hash table probing, the overall PIM kernel invocation overheads of PID-Join remain steady. To prevent the overheads from offsetting the performance benefits of PID-Join, the PIM kernels should have sufficient amounts of work which increase with respect to the table sizes.

Fig. 18 shows the impact of the PIM kernel invocation overheads with varying sizes of R and S. The contribution of the invocation overheads to the join execution latency decreases as the table sizes increase; the invocation overheads become negligible with 256 M tuples. However, the invocation overheads account for 14.9% with 64 M tuples. Still, despite the overheads, PID-Join consistently outperforms PRHO for all the evaluated workloads.

4.3.3 Collision Handling Method. PID-Join's single-IDP hash join algorithm employs linear probing (LP) for handling collisions (Section 3.2.2); however, LP may incur significant performance overheads as the number of unsuccessful probes increases [50, 53]. To analyze the implications of employing LP, we conduct a sensitivity study on the collision handling method of the hash join algorithm.



(a) Uniform R and S (b) Uniform R and skewed S

Fig. 19. Single-IDP performance of PID-Join with varying collision handling methods

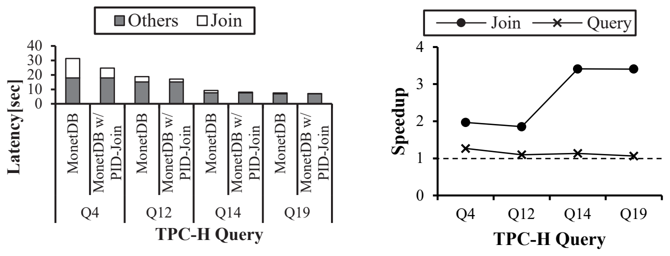
We consider two alternatives: quadratic probing (QP) [42] and double hashing (DH) [15]. LP, QP, and DH differ in choosing which entry to probe next when a collision occurs. LP probes the next entry, QP probes the i^2-th entry in the i-th iteration, and DH employs two hash functions $hash1$ and $hash2$: $hash1$ for locating the initial entry and $hash2$ for computing the step size. For LP and QP, we use tabulation as their hash functions. For DH, we use $hash1(x) = x \bmod 4096$ and $hash2(x) = q - (x \bmod q)$, where q is a prime number smaller than the hash table size [22]; there are 4,096 entries in the 32-KB hash table, and we set q as 7. To compare how the three collision handling methods affect the performance of PID-Join, we measure the execution latency and the collision rate of PID-Join with the collision handling methods on the PIM-enabled system by setting S to have 256-K tuples and with $|R| : |S|$ ratios from 1:1 to 1:16 for uniform S, and Zipf factors from 0.25 to 0.75 for skewed S. R is set to be uniform in all the experiments.

Fig. 19 shows the execution latency breakdowns and collision rates of PID-Join with LP, QP, and DH. Overall, LP achieves the lowest execution latency, although it incurs higher collision rates than QP and DH. LP's lower latency is due to its low computational overheads. Upon a collision, LP probes the next entry in the hash table by simply adding one to the previous key; however, QP requires integer multiplications not natively supported by the IDPs, and DH involves an additional hash function. The additional computational overheads make QP and DH incur higher join execution latency, making LP the fastest collision handling method. A similar tendency can be observed with skewed S as well.

The results also reveal that local partitioning accounts for up to 48.0% of the join execution latency of PID-Join. Local partitioning ensures the working set of a partition to fit within the WRAM by limiting the maximum per-partition tuple count, which is calculated by assuming the tuples of the partition to fill a 32-KB hash table with a fill rate of 50%. In case of extreme skewness, local partitioning may fail to fit the working set of a partition within the WRAM due to excessive redundant tuples. PID-Join can employ skew handling methods to tolerate high skewness (Section 4.2.4), yet we have not observed such extreme skewness in our experiments.

4.4 Potential Speedup with TPC-H Queries

We now examine the potential speedup of PID-Join, when integrated into a full-fledged RDBMS, with four single-join TPC-H queries (Q4, Q12, Q14, Q19) with a scale factor of 400. For the experiment, we first measure the end-to-end query execution latency of MonetDB [9], a prominent in-memory RDBMS, on the CPU-based baseline system. We then identify the execution latency of the join operator, replace the join execution latency with that of PID-Join, and calculate the estimated speedups for the queries. For each query, we first extract the tables which serve as the inputs to the join operator according to MonetDB's query plan. We then execute PID-Join with the extracted tables on the PIM-enabled system. When measuring the join execution latency of PID-Join for the queries, we take into account the host-DIMM data transfer overheads.



(a) Execution latency (b) Potential speedup

Fig. 20. Breakdown of the TPC-H query execution latency of MonetDB and the potential speedup of PID-Join

Fig. 20 shows the breakdown of the query execution latency on MonetDB and the estimated speedups of the queries when integrating PID-Join into MonetDB. PID-Join achieves a geometric mean speedup of 2.55× with the joins of the queries. The estimated speedup of PID-Join-augmented MonetDB is 1.14× in geometric mean. The results show that employing PID-Join is a promising choice to in-memory RDBMSs for accelerating joins.

5 DISCUSSIONS

5.1 PID-Join on Other PIM Hardware

We designed and optimized PID-Join for UPMEM DIMMs. Still, the key ideas of PID-Join are not confined to a specific product as PID-Join builds on top of common characteristics of modern DIMM and DRAM architectures. PID-Join assumes two key characteristics on its target PIM-enabled DIMMs: per-bank IDPs and data interleaving. We claim that, as long as the target DIMMs are based on DDR-compatible DIMMs, the two key characteristics are not specific to UPMEM DIMMs. For example, Samsung's HBM-PIM [40] and SK Hynix's AiM [27] also associate an IDP with per-IDP DRAM bank(s) and SRAM/registers. Future PIM-enabled DIMMs are expected to have similar designs as assumed by many recent studies (e.g., BLIMP [13], Newton [27], McDRAM [62]. Data interleaving is an inevitable characteristic of DIMMs caused by having multiple chips, and will be valid on future DIMMs as well. The byte-level transpose overhead is caused by the data interleaving of the DDR standards [4], originally designed for low critical-word latency. Some PIM hardware solves the problem with a custom layout [18, 39]; however, such a workaround is infeasible to be employed as a drop-in replacement for standard DIMMs. The other characteristics of UPMEM DIMMs such as the limited computational resources or bank/rank organizations come from the existing DRAM technology, such as the low thermal budget and the capacity-oriented design. Thus, as long as PIM-enabled DIMMs are DDR-compatible, it is highly likely for them to expose the similar challenges PID-Join faced with UPMEM DIMMs. In such circumstances, the key ideas of PID-Join will still be applicable.

Aside from PIM-enabled DIMMs, DRAM-based PIM hardware can take various forms. One popular approach is to utilize on-package memory such as high-bandwidth memory (HBM) as exemplified in HBM-PIM [40] or using the buffer chip as in AxDIMM [34]. However, as they belong to the DDR family, they still exhibit the shared-nothing architecture, where the difference is in the granularity (no notion of ranks in HBM, and one processing unit per rank in buffer chip). Because of this, the processing units will require similar all-to-all communication patterns, where all the communication data still travel via the host CPUs and memory. In addition, the processing units are likely to suffer from low thermal budget and limited silicon area. In such circumstances, the challenges PID-Join tackles still hold, with slightly different sweet spots as the data interleaving granularity is larger and there will be fewer independent processing units. Nonetheless, we posit that the solutions suggested in this work will generally be applicable with minor modifications.

5.2 Achieving Further Speedups

PID-Join has been shown to achieve higher join performance over the CPU-based join algorithms; however, we claim that the reported speedups do not reflect the upper-bound performance of PID-Join. First, although the PIM-enabled system supports higher DIMM counts, we evaluate PID-Join using only eight UPMEM DIMMs to facilitate fair comparisons against CPU-based join algorithms (Section 3.1). As modern servers provide tens of DIMM slots, we expect PID-Join to achieve much higher speedups on the servers by attaching more UPMEM DIMMs. Second, UPMEM DIMMs are a very early implementation of PIM-enabled DIMMs, so we expect the upcoming generations to provide higher IDP performance through process and manufacturing optimizations (e.g., higher clock frequency, floating-point units). PID-Join can take advantage of the optimizations and achieve high speedups. Third, PID-Join can achieve higher speedups with TPC-H queries by employing the prior studies on achieving 629.5× speedup for select and unique operators [20]. Note that PID-Join tackles the challenges of optimizing not only the join operator itself but also the high overheads of inter-IDP and CPU-to-UPMEM communications. The join operator requires these communications, whereas the select and unique operators do not.

5.3 Supporting Complex Join Queries

This work has evaluated PID-Join with the equi-join queries involving 32-bit integer join keys. Although the queries are one of the most representative types of join queries, more complex join queries can involve variable-length tuples [51] and non-trivial string equality rules [28, 36, 54, 69]. To PID-Join, variable-length tuples may degrade its performance as they affect its hashing throughput; however, we expect PID-Join to provide similar performance benefits over the CPU-based join algorithms with the tuples. PID-Join's inter-IDP communication bandwidth is not sensitive to its granularity (Section 4.3.1), the hashing throughput of the CPU-based join algorithm also decreases with the tuples, and the CPU-based join algorithms require extra main-memory accesses for processing the tuples. For non-trivial equality rules, PID-Join may use its single-IDP sort-merge join algorithm instead, similar to the prior studies on employing sort-merge join for the rules [1]. Using the single-IDP sort-merge join algorithm, PID-Join achieves a geometric mean speedup of 4.79× over M-PASS, the CPU-based sort-merge join algorithm (Fig. 13).

6 RELATED WORK

6.1 PIM-Assisted Join Algorithms

Although not directly applicable to the DIMM-based PIM architectures, several prior studies seek to improve join with 3D-stacked PIM architectures and simulation-based evaluations. Kepe et al. [35] show that hash join and sort-merge join are slower on PIM than the corresponding AVX-512 implementations, and suggest CPU-PIM hybrid query scheduling to exploit both PIM and CPUs. Boroumand et al. [10] design both hardware and software for in-memory hybrid transactional and analytical processing with specialized islands, respectively. Unlike the transactional island, the analytical island exploits PIM to mitigate the large data traffic. Santos et al. [16] implement database operations such as selection, projection and bloom join on the Vector-In-Memory architecture.

There exist some prior studies presenting contradicting conclusions compared to PID-Join. We briefly summarize the prior studies and give an explanation on why this is the case. Mirzadeh et al. [43] observe superiority of P-MPSM compared to radix hash join for near-memory execution. Similarly, Drumond et al. [17] also favor sort-merge join for near-memory processing. These conclusions stem from not the algorithm itself, but the difference in the underlying PIM hardware between the 3D-stacked and the DIMM-based PIM architectures. The in-memory processors of the 3D-stacked PIM architectures share memory and can rapidly transfer data between them, making the additional data movement in the merge phase less of a bottleneck. Therefore, the performance gains from the sequential memory accesses of sort-merge join stand out.

6.2 Mitigating the Data Movement Bottleneck

Diverse solutions have been proposed to optimize the performance of join by overcoming the memory-intensive nature of join. The memory-intensive nature has made in-memory join greatly suffer from the memory wall with the ever-increasing table sizes. Aimed at overcoming the memory wall, MonetDB [9] uses a columnar storage called decomposed storage model (DSM) to enable better use of host CPU caches. DSM stores individual columns in binary association tables and applies low-level relational algebra for fast computation. Additionally, MonetDB employs cache-conscious algorithms including its radix cluster algorithm to minimize memory accesses. Data compression has been investigated as it can reduce the amount of memory accesses. Begley et al. [8] compress hash keys by stripping off redundant bits and payloads by storing offset vectors to their parent tuples. Gao et al. [19] compress tables using run-length encoding, delta encoding, and bit-packing.

Some prior studies employ the ideas which resemble PID-Join's inter-IDP and host-DIMM communication optimizations. SIMDRAM [26] shares similarities with our work in that data have to be transposed between the co-processor and the host CPU due to different data views. Unlike our approach that utilizes the vector instructions on the CPUs, SIMDRAM has a specialized hardware designed for transposing data. The Mondrian data engine [17] utilizes data permutability; when data are partitioned and transferred, the exact order and location of the data objects within each partition do not affect the correctness of the join. While these parts from different works might look similar to our work, the innate nature and reasoning behind are completely different, and most importantly these works are all based on simulations and assume hardware configuration far from what is available in real hardware.

7 CONCLUSION

We proposed PID-Join, a fast in-memory join algorithm designed and optimized for commodity PIM-enabled DIMMs. PID-Join targets UPMEM DIMMs, currently the only publicly-available PIM-enabled DIMMs, and tackles their limited compute capabilities of the IDPs, slow inter-IDP communication, and slow host-DIMM data transfers. We first identified the single-IDP hash join algorithm as the best-performing join algorithm on an IDP. Then, we developed RnS, a fast all-to-all inter-IDP communication mechanism for the DIMMs, which exploits host CPU cache streaming and vector instructions. After that, we exploited the fact that the input and output tuples of a join do not need to be strictly ordered to implement fast rank-wise USG. Our evaluation using a real PIM-enabled system equipped with eight UPMEM DIMMs and a total of 1,024 IDPs shows that PID-Join greatly outperforms the existing in-memory join algorithms.

ACKNOWLEDGMENTS

This work was partly supported by the National Research Foundation of Korea (NRF) grants (2022R1C1C1008131, 2022R1C1C1011307) and the Institute of Information & communications Technology Planning Evaluation (IITP) grants (2020-0-01361, 2021-0-00853) funded by the Korea government (MSIT), the Korea Institute for Advancement of Technology (KIAT) grant (P0016150) funded by the Korea Government (MOTIE), Samsung Advanced Institute of Technology (SAIT), and the Yonsei Signature Research Cluster Program (2022-22-0002).

REFERENCES

[1] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *Proceedings of the VLDB Endowment* 5 (2012).

[2] Marco Antonio Zanata Alves, Carlos Villavieja, Matthias Diener, Francis Birck Moreira, and Philippe Olivier Alexandre Navaux. 2015. SiNUCA: A Validated Micro-Architecture Simulator. In *Proc. 17th International Conference on High Performance Computing and Communications (HPCC).*

[3] Austin Appleby. 2011. MurmurHash. https://sites.google.com/site/murmurhash/

[4] JEDEC Solid State Technology Association. 2012. DDR4 SDRAM STANDARD. https://xdevs.com/doc/Standards/DDR4/JESD79-4%20DDR4%20SDRAM.pdf

[5] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment* 7 (2013).

[6] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proc. 29th International Conference on Data Engineering (ICDE).*

[7] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *Proc. International Conference on Management of Data (SIGMOD).*

[8] Steven Keith Begley, Zhen He, and Yi-Ping Phoebe Chen. 2012. Mcjoin: A memory-constrained join for column-store main-memory databases. In *Proc. International Conference on Management of Data (SIGMOD).*

[9] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the Memory Wall in MonetDB. *Commun. ACM* 51 (2008).

[10] Amirali Boroumand, Saugata Ghose, Geraldo F. Oliveira, and Onur Mutlu. 2022. Polynesia: Enabling High-Performance and Energy-Efficient Hybrid Transactional/Analytical Databases with Hardware/Software Co-Design. In *Proc. 38th IEEE International Conference on Data Engineering (ICDE).*

[11] Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. 2007. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)* 32 (2007).

[12] Fabrice Devaux. 2019. The true Processing In Memory accelerator. In *Proc. 2019 IEEE Hot Chips 31 Symposium (HCS).*

[13] Alexandar Devic, Siddhartha Balakrishna Rai, Anand Sivasubramaniam, Ameen Akel, Sean Eilert, and Justin Eno. 2022. To PIM or Not for Emerging General Purpose Processing in DDR Memory Systems. In *Proc. 49th Annual International Symposium on Computer Architecture (ISCA).*

[14] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. 1997. A Reliable Randomized Algorithm for the Closest-Pair Problem. *Journal of Algorithms* 25 (1997).

[15] E Knuth Donald et al. 1999. The art of computer programming. *Sorting and searching* 3 (1999).

[16] Sairo R. dos Santos, Francis B. Moreira, Tiago R. Kepe, and Marco A. Z. Alves. 2022. Advancing Database System Operators with Near-Data Processing. In *Proc. 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP).*

[17] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. 2017. The Mondrian Data Engine. In *Proc. 44th International Symposium on Computer Architecture (ISCA).*

[18] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. 2015. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *Proc. 21st International Symposium on High Performance Computer Architecture (HPCA).*

[19] Hao Gao and Nikolai Sakharnykh. 2021. Scaling Joins to a Thousand GPUs. In *Proc. 12th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS).*

[20] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F Oliveira, and Onur Mutlu. 2021. Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture. *arXiv preprint arXiv:2105.03814* (2021).

[21] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2022. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access* 10 (2022).

[22] Michael T Goodrich and Roberto Tamassia. 2015. *Algorithm design and applications.* Vol. 363. Wiley Hoboken.

[23] The PostgreSQL Global Development Group. 2022. Documentation: 7.2: Performance Tips - PostgreSQL. https://www.postgresql.org/docs/7.2/performance-tips.html

[24] Anthony Gutierrez, Bradford M. Beckmann, Alexandru Dutu, Joseph Gross, Michael LeBeane, John Kalamatianos, Onur Kayiran, Matthew Poremba, Brandon Potter, Sooraj Puthoor, Matthew D. Sinclair, Mark Wyse, Jieming Yin, Xianwei Zhang, Akshay Jain, and Timothy Rogers. 2018. Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level. In *Proc. IEEE International Symposium on High Performance Computer Architecture (HPCA).*

[25] Anthony Gutierrez, Joseph Pusdesris, Ronald G. Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D. Emmons, Mitchell Hayenga, and Nigel Paver. 2014. Sources of Error in Full-System Simulation. In *Proc. 2014 IEEE International*

Symposium on Performance Analysis of Systems and Software (ISPASS).

[26] Nastaran Hajinazar, Geraldo F. Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. 2021. SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM. In *Proc. 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[27] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and T. N. Vijaykumar. 2020. Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning. In *Proc. 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[28] Yeye He, Kris Ganjam, and Xu Chu. 2015. Sema-join: joining semantically-related tables using big table corpora. *Proceedings of the VLDB Endowment* 8 (2015).

[29] Richard D Hipp. 2022. SQLite. https://www.sqlite.org/index.html

[30] Intel. 2022. APP Metrics for Intel Microprocessors - Intel Xeon Processor. https://www.intel.com/content/dam/support/us/en/documents/processors/APP-for-Intel-Xeon-Processors.pdf

[31] Intel. 2022. Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 3A. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html

[32] Joe Jeddeloh and Brent Keeth. 2012. Hybrid Memory Cube New DRAM Architecture Increases Density and Performance. In *Proc. 2012 Symposium on VLSI Technology (VLSIT).*

[33] Bob Jenkins. 1997. A Hash Function for Hash Table Lookup. https://burtleburtle.net/bob/hash/doobs.html

[34] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shin-Haeng Kang, Sukhan Lee, Songyi Han, YeonGon Cho, Jin Hyun Kim, Yongsuk Kwon, et al. 2021. Near-memory processing in action: Accelerating personalized recommendation with AxDIMM. *IEEE Micro* 42 (2021).

[35] Tiago R. Kepe, Eduardo C. de Almeida, and Marco A. Z. Alves. 2019. Database Processing-in-Memory: An Experimental Study. *Proceedings of the VLDB Endowment* 13 (2019).

[36] Martin Kiefer, Max Heimel, Sebastian Breß, and Volker Markl. 2017. Estimating join selectivities using bandwidth-optimized kernel density models. *Proceedings of the VLDB Endowment* 10 (2017).

[37] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment* 2 (2009).

[38] Peter M. Kogge. 1994. EXECUBE - A New Architecture for Scaleable MPPs. In *Proc. 1994 International Conference on Parallel Processing (ICPP).*

[39] Jinho Lee, Jung Ho Ahn, and Kiyoung Choi. 2016. Buffered compares: Excavating the hidden parallelism inside DRAM architectures with lightweight logic. In *Proc. Design, Automation & Test in Europe Conference & Exhibition (DATE).*

[40] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, et al. 2021. Hardware architecture and software stack for PIM based on commercial DRAM technology: Industrial product. In *Proc. 48th Annual International Symposium on Computer Architecture (ISCA).*

[41] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects. In *Proc. International Conference on Management of Data (SIGMOD).*

[42] Ward Douglas Maurer and Ted G Lewis. 1975. Hash table methods. *ACM Computing Surveys (CSUR)* 7 (1975).

[43] Nooshin S. Mirzadeh, Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Sort vs. Hash Join Revisited for Near-Memory Execution. In *Proc. 5th Workshop on Architectures and Systems for Big Data (ASBD).*

[44] Joel Nider, Craig Mustard, Andrada Zoltan, John Ramsden, Larry Liu, Jacob Grossbard, Mohammad Dashti, Romaric Jodin, Alexandre Ghiti, Jordi Chauzi, and Alexandra (Sasha) Fedorova. 2021. A Case Study of Processing-in-Memory in off-the-Shelf Systems. In *Proc. USENIX Annual Technical Conference (USENIX ATC).*

[45] Tony Nowatzki, Jaikrishnan Menon, Chen-Han Ho, and Karthikeyan Sankaralingam. 2015. Architectural Simulators Considered Harmful. *IEEE Micro* 35 (2015).

[46] Oracle. 2022. MySQL 8.0 Reference Manual: Nested-Loop Join Algorithms. https://dev.mysql.com/doc/refman/8.0/en/nested-loop-joins.html

[47] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. 1997. A Case for Intelligent RAM. *IEEE Micro* 17 (1997).

[48] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. In *Proc. International Conference on Management of Data (SIGMOD).*

[49] Ben Perach, Ronny Ronen, Benny Kimelfeld, and Shahar Kvatinsky. 2022. PIMDB: Understanding Bulk-Bitwise Processing In-Memory Through Database Analytics. *arXiv preprint arXiv:2203.10486* (2022).

[50] Georg Ch Pflug and Hans W Kessler. 1987. Linear probing with a nonuniform address distribution. *Journal of the ACM (JACM)* 34 (1987).

[51] Orestis Polychroniou, Rajkumar Sen, and Kenneth A Ross. 2014. Track join: distributed joins with minimal network traffic. In *Proc. International Conference on Management of Data (SIGMOD).*

[52] Mihai Pătraşcu and Mikkel Thorup. 2012. The Power of Simple Tabulation Hashing. *J. ACM* 59 (2012).
[53] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11 (2018).
[54] Maximilian Reif and Thomas Neumann. 2022. A scalable and generic approach to range joins. *Proceedings of the VLDB Endowment* 15 (2022).
[55] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proceedings of the VLDB Endowment* 9 (2015).
[56] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* 10 (2011).
[57] Ran Rui, Hao Li, and Yi-Cheng Tu. 2015. Join algorithms on GPUs: A revisit after seven years. In *Proc. 2015 IEEE International Conference on Big Data (Big Data).*
[58] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient join algorithms for large database tables in a multi-GPU environment. *Proceedings of the VLDB Endowment* 14 (2020).
[59] Lukas Rupprecht, William Culhane, and Peter Pietzuch. 2017. SquirrelJoin: Network-Aware Distributed Join Processing with Lazy Partitioning. *Proceedings of the VLDB Endowment* 10 (2017).
[60] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. 2015. On the surprising difficulty of simple things: the case of radix partitioning. *Proceedings of the VLDB Endowment* 8 (2015).
[61] Ambuj Shatdal, Chander Kant, and Jeffrey F Naughton. 1994. Cache conscious algorithms for relational query processing.
[62] Hyunsung Shin, Dongyoung Kim, Eunhyeok Park, Sungho Park, Yongsik Park, and Sungjoo Yoo. 2018. McDRAM: Low Latency and Energy-Efficient Matrix Computations in DRAM. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 37, 11 (2018).
[63] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *Proc. 35th International Conference on Data Engineering (ICDE).*
[64] John S. Sobolewski. 2003. *Cyclic Redundancy Check.* John Wiley and Sons Ltd.
[65] UPMEM SAS. 2021. UPMEM SDK. https://sdk.upmem.com/2021.3.0/index.html
[66] Makoto Yabuta, Anh Nguyen, Shinpei Kato, Masato Edahiro, and Hideyuki Kawashima. 2017. Relational Joins on GPUs: A Closer Look. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 28 (2017).
[67] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59 (2016).
[68] Zuyu Zhang, Harshad Deshmukh, and Jignesh M Patel. 2019. Data Partitioning for In-Memory Systems: Myths, Challenges, and Opportunities.. In *CIDR.*
[69] Erkang Zhu, Yeye He, and Surajit Chaudhuri. 2017. Auto-join: Joining tables by leveraging transformations. *Proceedings of the VLDB Endowment* 10 (2017).

Received October 2022; revised January 2023; accepted February 2023