Database Processing-in-Memory: An Experimental Study

Tiago R. Kepe
UFPR & IFPR, Brazil
trkepe@inf.ufpr.br

Eduardo C. de Almeida
UFPR, Brazil
eduardo@inf.ufpr.br

Marco A. Z. Alves
UFPR, Brazil
mazalves@inf.ufpr.br

ABSTRACT

The rapid growth of "big-data" intensified the problem of data movement when processing data analytics: Large amounts of data need to move through the memory up to the CPU before any computation takes place. To tackle this costly problem, Processing-in-Memory (PIM) inverts the traditional data processing by pushing computation to memory with an impact on performance and energy efficiency. In this paper, we present an experimental study on processing database SIMD operators in PIM compared to current x86 processor (i.e., using AVX512 instructions). We discuss the execution time gap between those architectures. However, this is the first experimental study, in the database community, to discuss the trade-offs of execution time and energy consumption between PIM and x86 in the main query execution systems: materialized, vectorized, and pipelined. We also discuss the results of a hybrid query scheduling when interleaving the execution of the SIMD operators between PIM and x86 processing hardware. In our results, the hybrid query plan reduced the execution time by 45%. It also drastically reduced energy consumption by more than 2× compared to hardware-specific query plans.

PVLDB Reference Format:
Tiago R. Kepe, Marco A. Z. Alves, and Eduardo C. de Almeida. Database Processing-in-Memory: An Experimental Study. *PVLDB*, 13(3): 334-347, 2019.
DOI: https://doi.org/10.14778/3368289.3368298

1. INTRODUCTION

Applications based on data analysis need to move large amounts of data between memory and processing units to look for patterns. Computers have relied on this traditional computing-centric execution since the introduction of the Von Neumann model. In this model, however, data movement severely affects performance and energy consumption. Recent studies show that data movement accounts for almost 63% of the total energy consumption and imposes high latencies [6,36].

Proceedings of the VLDB Endowment, Vol. 13, No. 3
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3368289.3368298

Traditional query execution systems have been operating only on computing-centric models [11]. The materialization query execution system generates lots of intermediate data that move along the memory hierarchy to process the operators of a query plan. The vectorized query execution system tries to exploit the caching mechanism and the CPU processing with a high interpretation overhead. The pipelined query execution system uses the Just-In-Time (JIT) compilation to fuse operators of the same pipeline into a monolithic code fragment. Although the authors of [28] call JIT as data-centric compilation, the query execution is still computing-centric by moving data to the CPU with many adaptations to make better use of the memory caches. In this paper, we study the data-centric execution model to tackle the data movement problem in query execution systems with logical units integrated closer to the data (inside memory devices), which is called Processing-in-Memory (PIM) [25,34,45].

Database engineers have been evaluating PIM approaches with processing components installed in magnetic disks [1, 12,26], RAM [38], and more recently in flash disks [10,13,46]. However, commercial products have not been adopting those approaches for three main reasons: 1) Limitations of the hardware technology; 2) The continuous growth in CPU performance complied to the Moore's Law and Dennard scaling; 3) The lack of a general programming interface that leads to low abstraction level when handling hardware errors.

Now, modern PIM hardware integrate traditional DRAM dies and logic cells in the same chip area with the Through-Silicon Via (TSV), forming a 3D-stacked memory with a high degree of parallelism. Therefore, modern PIM can leverage current memory protocols to handle hardware errors. Current GPUs already embed the emerging 3D-stacked memories, such as the Hybrid Memory Cube (HMC) [23] and the High Bandwidth Memory (HBM) [30]. However, there has not been any in-depth study of query processing on PIM with Single Instruction Multiple Data (SIMD) support.

In this experimental paper, we detail the implementation of five major query operators into a simulator of PIM hardware: selection, projection, aggregation, sorting, and join (hash join, sort-merge join, and nested loop join). In particular, we present a new SIMD sorting algorithm that requires fewer memory instructions compared to the state of the art [21]. For each operator, we gauge the latency and energy spend to process TPC-H and Zipf distribution datasets. We evaluate the high levels of parallelism and data access when using AVX512 extensions from x86 processors, compared to modern PIM architectures with SIMD support on registers of 256-bytes wide. To the best of our knowledge, we are the first experimental study to discuss the trade-offs of latency and energy efficiency between PIM and x86 in the main query execution systems: materialized, vectorized, and pipelined. Our major contributions are:

- We detail the implementation of query operators with support to SIMD instructions to execute in a PIM hardware. In particular, we present a new SIMD sorting algorithm.
- We present a comprehensive performance analysis of the query operators in modern PIM hardware regarding time and energy. We also distinguish the trade-offs to process each SIMD operator on top of the materialized, vectorized, and pipelined query execution systems.
- We discuss the potential of a hybrid scheduling of the query operators that interleaves their execution between PIM and x86 processing. We present a heuristic to build this hybrid scheduling and discuss the experimental results. Our scheduler reduced execution time by 35% and 45% when compared to PIM and x86 hardware-specific query plans, respectively. It also reduced energy consumption by more than 2× compared to the x86 processor.

Outline: In Section 2, we introduce current PIM architectures and motivate their adoption through a detailed execution of the selection operator. In Section 3, we present our experimental design and explain the reasons to choose a particular group of query operators. In Section 4, we detail our implementations of five SIMD query operators. In Section 5, we analyze the performance and energy consumption of the distinct query execution systems in each architecture. In Section 6, we discuss a hybrid query execution for future Database Management Systems (DBMS). In Section 7, we present related work, and Section 8 concludes.

2. PROCESSING-IN-MEMORY ARCHITECTURE

In this section, we describe how the PIM hardware works and how query operators can benefit from its parallelism and memory bandwidth.

A typical 3D-stacked memory consists of up to 8 layers of DRAM dies interconnected by the TSV to the logic die at the base. The 3D memory devices logically split the DRAM dies into 32 independent vaults. Each vault contains up to 8 independent DRAM banks, where each DRAM bank provides as much as 256-bytes of data per row access. This 3D design achieves 512 parallel requests and can deliver a maximum bandwidth of 320 GB/s, which is about 4x higher than a traditional DDR-3 design. Our work uses this 3D-stacked memory as the main memory for both architectures x86 and PIM, making a strong case for our comparisons.

The logic layer of 3D-stacked memories supports the implementation of traditional logic, similar to those present inside processors. In the case of the HMC proposal, it implements update operations performing arithmetic, logical, and bit-wise atomic instructions over scalars of up to 16 bytes size. A different logic layer design, called HIVE, proposes larger registers to provide SIMD parallelism inside the 3D memory [3]. Although HIVE foresees the feasibility of SIMD instructions operating over SIMD registers from 256 B up to 8 KB, we will use the modest size of 256 B per operation. Thus, our PIM-256B architecture works with registers of 256 bytes wide that shall store multiple operands. For simplicity, we call each operand position inside a SIMD register as a lane (e.g., when using 32-bit operands, a single 256 B register may contain up to 64 valid lanes). Similar to the HMC proposal, HIVE also relies on the CPU to trigger instructions to be executed inside the memory. We use HIVE in our experiments due to its simplicity and low energy consumption (not requiring a full processor inside the memory), its high performance, and its acceptance as it was used to implement derived architectures [44, 48].

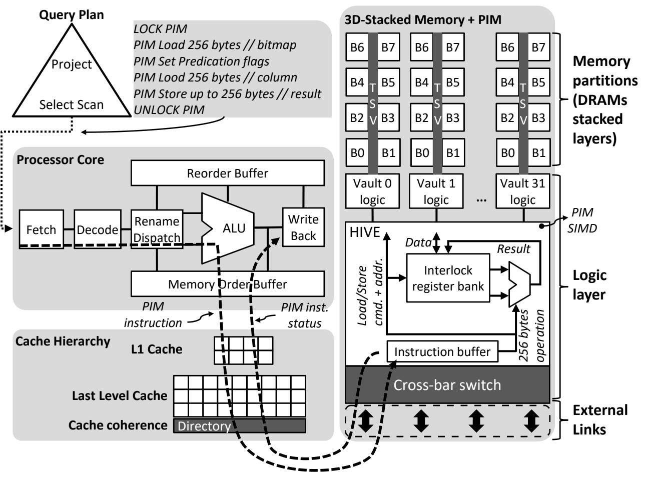


Figure 1: A query datapath movement in a traditional von Neumann architecture plus a modern 3D-stacked memory with PIM and SIMD support [3].

Figure 1 describes the PIM architecture with SIMD support from HIVE (to the right side) and the traditional x86 architecture inspired by the von Neumann model (to the left): formed by the processor core and a detached cache hierarchy. At the top of Figure 1, the processor dispatches PIM instructions (dashed line) directly to the PIM device bypassing the caches, while maintaining the coherence with the Last-Level Cache (LLC) directory. The instructions to access memory (load/store) might require accessing up to 256 bytes each. The 32 independent vaults allow 32 PIM SIMD-like load instructions of 256-bytes at a time. However, this high level of parallelism depends on the memory access pattern from the application. During a memory load from the logic layer, the data request goes to a specific DRAM bank inside a designated vault (using the load address to indicate the correct device). Once data is available, the Vault Controller transfers it to the PIM SIMD register bank in which every register implements a ready bit (interlock mechanism), and thus the operations only continue whenever case that bit is set. At the end of the execution of each instruction, the PIM device only returns the instruction status to the CPU. This data-centric design is the main advantage compared to current DBMS that filter data in hardware before passing to the CPU, like Netezza and Exasol. They have to deal with packing qualifying tuples into condensed pages to avoid unnecessary bufferpool pollution, which is expensive and error-prone. Therefore, the significant benefits to be explored in the DBMS-PIM co-design are the drastic reduction in energy consumption and the internal high memory bandwidth due to the high levels of data access parallelism and on-chip processing.

Other capabilities of PIM include memory protocols to support all the idiosyncrasies of PIM instructions, such as cache coherence, Memory Management Unit (MMU), Error-Correcting Code Memory (ECC) and Direct Memory Access (DMA). The execution flow works at instruction-granularity as the traditional CPU processing (e.g., AVX/SSE x86-extensions), i.e., programmers insert intrinsics PIM instructions into the code, like Intel Intrinsics, and the compiler flags them as special memory PIM instructions.

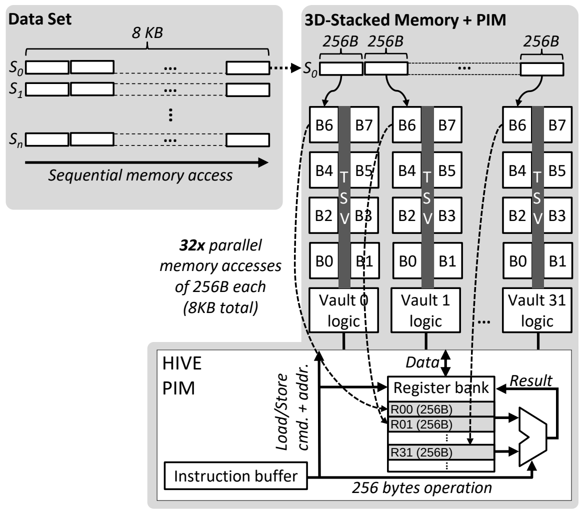
2.1 Understanding the PIM Selection

To understand the impact of PIM with SIMD on query processing and to simplify our analysis, we initially focus on the execution of the selection operator, instead of more complex operations (like join). In the traditional selection operator, the memory requests start from the CPU to the main memory reaching all levels of cache, moving data up and down through the memory hierarchy.

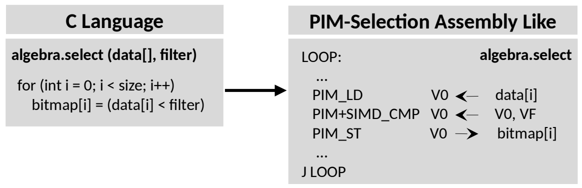
As the traditional von Neumann architecture detaches the processor from the main memory, the data movement is an inherited side effect that memory caches try to alleviate. The caching mechanism is particularly efficient for applications with good data reuse. However, this is not the case of the selection operator because it streams datasets polluting the hierarchy of memory caches with dead-on-arrival cache lines. Even the selection with an index (select-index) has the same streaming behavior. The processes are blocked within a corresponding indexed-data portion.

The selection operator appears as a good fit for PIM with the potential to exploit the high internal bandwidth of 3D stacked memories. Figure 2a depicts the execution of the selection using PIM with SIMD support. A dataset can be either an entire table, a column, or even a chunk of an indexed-data portion. The selection operator performs sequential memory access to process fragment-at-a-time of 256 B (S_0 to S_n in Figure 2a). Figure 2b presents the C language code of the selection operator and the respective translation to PIM Assembly-like code. As a simplistic use case, we generate the output of the selection as a bitmap, although it is also possible with few adaptations to design a PIM algorithm to emit a selection vector as output.

During the execution of the selection operator in HIVE, the CPU sends the PIM instructions for on-chip processing. Inside the logic layer, HIVE interprets and executes each instruction. During bursts of memory loads, up to 32 parallel reads can be performed by HIVE, using all the throughput of the memory vaults (up to 320 GB/s) [23, 30]. Although it is possible to issue multiple loads in parallel, the execution follows strict in-order fashion. We observe that all the registers can receive data from any memory vault during memory loads as the implementation of HIVE is coupled with the interconnection of the vaults. For the first instruction, HIVE loads 256 bytes of data ($data[i]$) from one specific memory vault into the SIMD register bank. Then, the **PIM+SIMD_CMP** instruction compares the PIM+SIMD loaded register and the SIMD register of filter (VF): a pre-load PIM+SIMD register that has replicas of the filtering value. In the end, the **PIM_ST** instruction writes the resulting bitmap into a given memory vault.



(a) Selecting data using PIM + SIMD support.



C Language

```
algebra.select (data[], filter)

for (int i = 0; i < size; i++)
    bitmap[i] = (data[i] < filter)
```

PIM-Selection Assembly Like

```
LOOP:                        algebra.select
   ...
   PIM_LD          V0 <--    data[i]
   PIM+SIMD_CMP    V0 <--    V0, VF
   PIM_ST          V0 -->    bitmap[i]
   ...
J LOOP
```

(b) The selection operator in C and PIM Assembly-like.

Figure 2: The selection operator in PIM.

2.2 The potential parallelism of PIM-256B vs. x86 AVX512-64B

In this section, we briefly highlight the potential parallelism of PIM compared to the x86 processor for processing selections (Section 3 presents the details of the experiments).

The x86 version of the selection operator uses AVX512 extensions with SIMD registers of 64 bytes (AVX512-64B), and the PIM version uses SIMD registers of 256 bytes (PIM-256B). Notice that AVX512-64B uses the largest SIMD registers available for the traditional x86 processor. We also applied the loop unrolling technique to push the architectures to the maximum degree of parallelism available, i.e., the AVX512[1] processing up to unroll depth of 8× and PIM up to 32× to take advantage of the 32 independent vaults.

This experiment measures the latency of the operator varying the size of the input dataset to fit into the L1/L2 caches. Figure 3 shows an appealing case for the x86 processing: A small dataset processing with a low ratio of cache misses to show the potential of PIM even in unfavorable cases. In datasets bigger than cache sizes, the cache misses degrades the performance of the x86 processing.

The three foremost benefits in here for PIM processing are: 1) Only a single load inside PIM-256B shall retrieve up to 256 B whereas the AVX512-64 requires 8 operations to access the same amount of data; 2) Considering 4 B operands (e.g., integer variable) the PIM-256B shall operate over 64 elements (lanes), while AVX512-64 operates over only 16 by the same time; 3) It is possible to considerably reduce

[1]Generally, 8× is the deepest unroll implemented by compilers due to the reduced number of general purpose registers.

the number of data transfers between CPU and main memory operating directly inside the memory for streaming data patterns. Based on those benefits, the PIM execution is 3× faster than AVX512 for both datasets when using all the memory vaults. In Section 5, we provide an in-depth analysis of the results for many query operators.

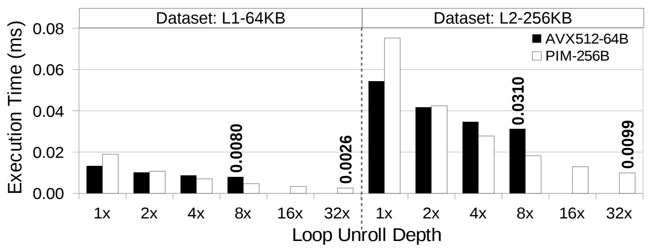


Figure 3: The execution time of the selection operator for AVX512-64B and PIM-256B. The dashed line separates the X-axis in two data sets: one fits in cache L1 and the other in cache L2. Also, we ranged the loop unroll depth from 1× up to 32×, which implies in varying the degree of parallelism.

Although x86 ISAs provide load instructions that bypass the cache memories, it is important to notice that off-chip communication is still present, consuming time and energy. Processors usually can only perform 10 parallel requests per processing cores (due to MSHR - miss status handler register - limitations), resulting in total parallelism of 10× 64 bytes (640B), which is smaller than the parallelism of PIM, i.e., 32× 256-bytes (8KB). In this paper, we are using only a single thread to execute the operators on both systems AVX512-64B and PIM-256B. Note that the x86 processor would require at least 13-cores/threads (13× 640B = 8.1KB) to achieve the same bandwidth present in PIM, requiring a much higher power budget.

3. EXPERIMENT DESIGN

In this section, we detail the design of our experiments. We describe how we choose the query operators for analysis. Then, we discuss the data distribution used in the rest of the paper. Finally, we describe the simulation environment.

3.1 Choosing the Group of Operators

In this section, we investigate the most time and memory consuming query operators to justify a relevant group of operators in our study. First, we investigate the response time breakdown of the TPC-H queries with 100 GB using the column-wise database MonetDB v11.33.11 [2]. For this section, we perform the experiments on a real machine using an Intel quad-core i7-5500U processor running at 2.40 GHz with 16 GB of RAM (DDR-3L 1333/1600) and 4 MB LLC running OpenSuse Leap 42.3 on Linux kernel 4.4.76-1-default. We added the TRACE statement modifier of MonetDB on each query to collect statistics and performance traces.

Figure 4 presents the query execution breakdown plotting the most time and memory consuming operators: projection, selection, join, aggregation, grouping, and the remnant ones grouped into the category "others". The last bar summarizes the entire benchmark ("All TPCH"). We set as the relevant group of operators the projection, selection, join, and aggregation, as they represent almost 90% of the 100 GB TPC-H benchmark for execution time and memory usage.

3.2 Workload's Data Distribution

Our goal in the design of the data distributions is to evaluate the impact of different memory accesses. We study this impact in two cases: 1) The case when the input datasets fit into the cache hierarchy; 2) When they do not. In theory, the first case is the best one for the x86 processing because the operators can take advantage of the caching mechanism for data reuse. We assume three particular queries. TPC-H Query 01 is a low-cardinality group query without joins (fitting inside the cache memory). Most of its execution time is spent projecting columns and computing the aggregation. TPC-H Query 03 is a high-cardinality group query with joins (does not fit inside the cache memory). Most of its execution time is spent filtering and projecting columns. We run the query operators varying the size of the input columns to fit in the L1, L2, LLC caches, and in DRAM with at least 1 GB. In the third query, we evaluated the query aggregation with the Zipf distribution in the caches (L1, L2, and LLC) and the DRAM of 1 GB:

```
SELECT sum(col_zipf) FROM table GROUP BY col_zipf
```

The Zipf distribution presents a bias based on the frequency of the values, which we use to simulate random memory access to the groups in the hash table of the aggregation operator. As a result, some groups are more accessed than others generating data reuse in the memory caches.

3.3 SiNUCA: A Validated Simulator

We implemented the PIM architecture on top of the SiNUCA[3] cycle-accurate simulator [4]. Notice that current PIM hardware do not yet implement all the extensions depicted in Figure 1. Therefore, we rely on architectural simulators to implement the required hardware extensions for our study, which is the standard approach adopted by processor industries and hardware research [49]. Using SiNUCA, it is possible to execute the database operators in the simulated environment obtaining performance results for x86 and PIM executions. SiNUCA was validated against two real machines [4] and it implements a realistic out-of-order processor, advanced multi-banked and non-blocking caches together with the PIM hardware. Furthermore, SiNUCA was adopted by studies that extend PIM hardware in computer architecture [44, 48], and database [27, 47] contexts.

The baseline architecture was inspired by the Intel Sandy-Bridge microarchitecture that we extended with the AVX-512 instruction set capabilities referred to as AVX512-64B. Although this microarchitecture does not represent the state-of-the-art, the memory bottleneck is still an unsolved problem for newer architectures that rely on the computing-centric execution. In all the cases, the traditional main memory is a high-bandwidth 3D-stacked memory [23, 30]. Moreover, by adding 3D-stacked memory to this architecture, we are virtually providing up to 32 channels to the x86 processor, which is more than 5× higher than the 2019's Cascade-Lake processor will offer. Table 1 presents the parameters of the target architectures with the same setup used by related work [3, 48]. The PIM architecture has 32

[2]MonetDB available at: `https://www.monetdb.org/`

[3]`https://bitbucket.org/mazalves/sinuca/src`

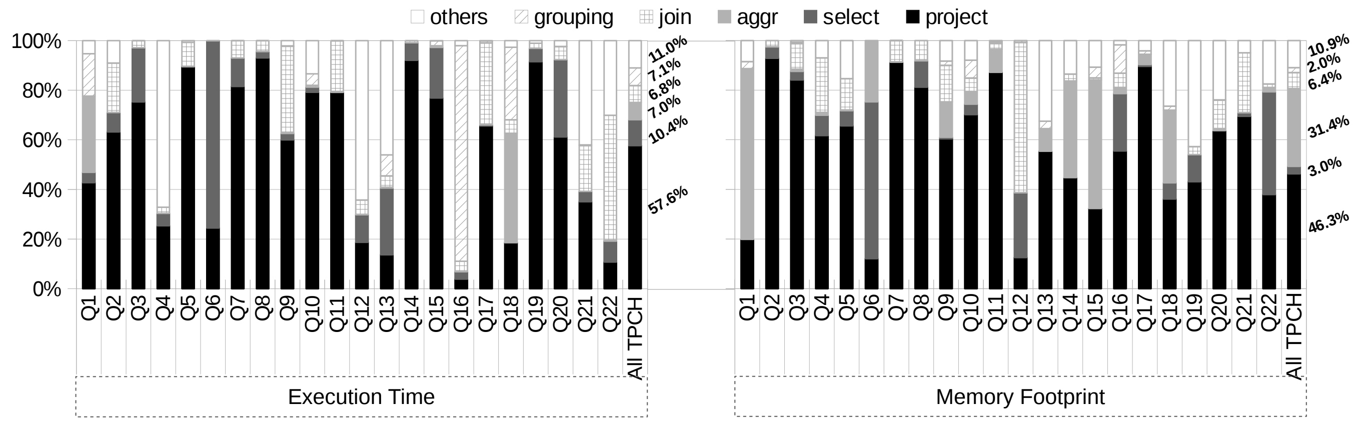


Figure 4: The 100 GB TPC-H benchmark breakdown in the top time and memory consuming operators with MonetDB [19].

vaults with 8 DRAM banks each, and the total memory capacity is 8 GB. Also, this architecture has 36 SIMD registers of 256 bytes that operates with operands from 4 to 256 bytes.

Our evaluation metrics take into consideration the operator execution time and energy consumption. In the micro-benchmark analysis, each operator is evaluated in isolation, with no interactions among them, except in the pipelined execution that requires such interaction. For the operator latency, we record the execution time. For energy consumption, we measure the memory read, write, and data transfer operations. We compute the memory energy estimation of the DRAM values considering the architecture of the current 3D-stacked memories [23,30]. In the macro-benchmark analysis, we evaluate the whole query execution.

Table 1: Parameters of the target architectures taken into account to design the experiments [48].

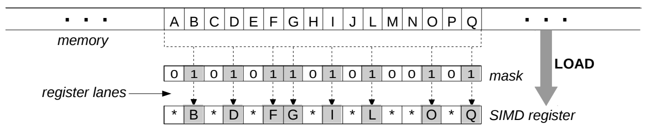
OoO Execution Cores	16 cores @ 2.0 GHz, 32 nm; 6-wide issue; 16 B fetch; Buffers: 18-entry fetch, 28-entry decode; 168-entry ROB; MOB entries: 64-read, 36-write; 1-load, 1-store units (1-1 cycle); 3-alu, 1-mul. and 1-div. int. units (1-3-32 cycle); 1-alu, 1-mul. and 1-div. fp. units (3-5-10 cycle); 1 branch per fetch; Branch pred.: Two-level GAs. 4,096 entry BTB;
L1 Data + Inst. Cache	32 KB, 8-way, 2-cycle; Stride prefetch; 64 B line; MSHR size: 10-request, 10-write, 10-eviction; LRU policy;
L2 Cache	Private 256 KB, 8-way, 4-cycle; Stream prefetch; 64 B line; MSHR size: 20-request, 20-write, 10-eviction; LRU policy;
L3 Cache	Shared 40 MB (16-banks), 2.5 MB per bank; LRU policy; 16-way, 6-cycle; 64 B line; Bi-directional ring; Inclusive; MOESI protocol; MSHR size: 64-request, 64-write, 64-eviction;
PIM device	32 vaults, 8 DRAM banks/vault; DRAM@166 MHz; 8 GB total size; 256 B Row buffer; Closed-page policy; 8 B burst width at 2:1 core-to-bus freq. ratio; 4-links@8 GHz; DRAM: CAS, RP, RCD, RAS, CWD cycles@166 MHz (9-9-9-24-7);
SIMD units	Unified func. units (integer + floating-point) @1 GHz; Latency (cpu-cycles): 2-alu, 6-mul. and 40-div. int. units; Latency (cpu-cycles): 10-alu, 10-mul. and 40-div. fp. units; Op. sizes (bytes): 4, 8, 16, 32, 64, 128, 256; Register bank: 36x 256 B (Originally 16x 8192 B in HIVE proposal);

4. IMPLEMENTATION DETAILS OF THE SIMD QUERY OPERATORS

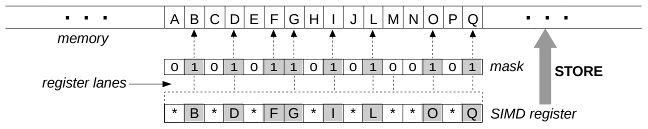
In this section, we describe the implementation details of the query operators with SIMD. We also describe the relevant SIMD vectorization features applied in the operators.

In a nutshell, the implementations of the selection and projection operators require SIMD *load* and *store* memory instructions. However, each operator requires a different strategy to better use SIMD instructions. The hash join and aggregation require the *gather* and *scatter* SIMD memory instructions to load and store multiple entries of hash tables. Finally, the sorting operation and sort-merge join require the *min/max* and *shuffle* SIMD instructions.

Selection. The selection operator filters data and generates a bitmap with bits set to 1 for qualified data. We discuss our two selection implementations with an example of a chain of selections. Figure 5a depicts the *selective load* SIMD instruction using a bitmap as a bitmask to filter the next selection column from a contiguous memory location. In a chain of selections, the output bitmap of an operator is the input to the next one. Another common implementation of this operator generates a selection vector as output. The output is a SIMD register with the values arranged according to the input selection vector (e.g., the index register in Figure 6a). In a chain of selections, the *gather* instruction reads the next column from a non-contiguous memory location based on the selection vector from the previous selection.



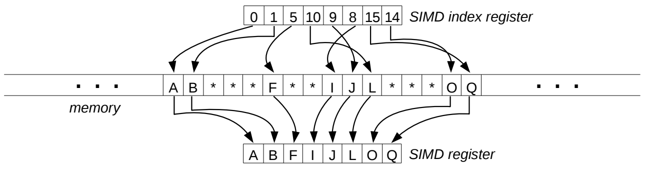
(a) Selective Load (Selection).



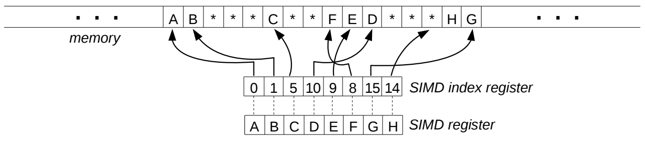
(b) Selective Store (Projection).

Figure 5: SIMD memory instructions based on bitmask.

Projection. We present two implementations of the projection operator: 1) Projection with an index register for high selectivity queries, like the selection vector of MonetDB [19], and 2) Projection without an index register for low selectivity queries reducing the memory footprint. Our first implementation uses the *selective store* to project the target column without an index register. Figure 5b shows an example of this execution. The projection writes data from a subset of register lanes to a contiguous memory location. In the example, the output bitmap generated by the selection is the input of the projection, where the bits set to '1' indicate the values to project. In our second implementation using an index register as input, the projection uses the *scatter* memory instruction to write the non-contiguous values of the target column (see Figure 6b).



(a) *Gather* Instruction.



(b) *Scatter* Instruction.

Figure 6: SIMD instructions based on index register.

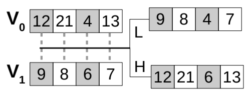
Hash Table and Hash Join. Our implementation of the hash join is based on a vectorized SIMD-friendly linear building and probing algorithm [40]. We use the *gather* and *scatter* memory instructions to implement hash tables for the join and aggregation operations. The *gather* instruction loads multiple entries of the hash table (non-contiguous memory locations). The *scatter* is the symmetric instruction that writes data to multiple memory locations based on an index register. For the x86 implementation, those instructions iterate (loop iteration) over the index register, identify the register lanes pointing to the same cache line, then read/write one or two cache lines per iteration until there are no more indexes to process. For the PIM implementation, the instructions iterate over an index register, group the register lanes pointing to the same DRAM banks and generate up to 32 load/store instructions of 256-bytes per iteration until there are no more indexes to process.

Aggregation. The aggregation operator updates the aggregated values into the hash table using *gather* and *scatter* memory instructions. It *gathers* multiple entries from a hash table and applies the conflict-free [17] to update the aggregation values. Then it *scatters* them back to the hash table.

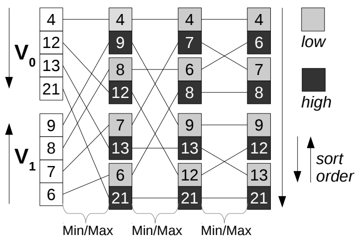
Sorting. Now, we discuss the implementation of the Sort-Merge algorithm that outperforms other sorting algorithms when exploiting the SIMD instructions [21]. The implementation of the Sort-Merge algorithm in SIMD is more intricate than the previous query operators. Both sort and merge phases of the algorithm rely on SIMD *min/max* and *shuffle* instructions available on current SIMD processors [22].

The *min/max* instructions process two SIMD registers V_0 and V_1 of length k (where k is the number of register lanes). These instructions compare the corresponding lanes of the registers and emit as output a new SIMD register that contains the lowest/highest values between V_0 and V_1, respectively. Figure 7a exemplifies those instructions that receive as input the SIMD registers V_0={12,21,4,13} and V_1={9,8,6,7}. The *min* instruction emits as output the SIMD register L={9,8,4,7} with the lowest values of each lane (dashed gray lines) between V_0 and V_1, and the *max* instruction emits as output the SIMD register H={12,21,6,13} with the highest values.

The *bitonic merge* is a networked merge algorithm that compares every element of two SIMD registers. The execution requires one register sorted in ascending order and the other in descending order. Figure 7b shows a *bitonic merge* network with two 4-wide SIMD registers (k=4). The network has log_2^k levels applied in parallel. Therefore, the execution of the whole sort-merge requires 2 log_2^{2k} *min/max* and $1 + 2\ log_2^{2k}$ *shuffle* instructions.



(a) SIMD Min/Max instructions.



(b) Bitonic Merge.

Figure 7: *Min/Max* and *bitonic merge* examples.

Our SIMD sort instruction consists of two operations: the in-register SIMD Sort and in-block-register Merge. The former sorts a SIMD register with k lanes using an odd-even sorting network. First, it sorts the register lanes by applying successive *min/max* instructions. Then, each lane is sorted (shown as the gray and white lanes in Figure 8a). Finally, it applies a series of *shuffle* instructions to transpose the k sorted lanes (vertical order) to form k sorted registers (horizontal order).

Figure 8b brings a general overview of our in-register SIMD Sort with eight registers. The process has two steps: 1) It compares all registers to distinguish the overall lowest and highest values of each lane, resulting in two registers (V_0 and V_7), which requires $k\ log_2^k$ *min/max* instructions; 2) The next step compares the remaining registers (i.e., $k-2$ registers) as a full binary tree data structure, where the result of each level is the lowest and highest registers. Then the number of registers to compare is reduced by a factor of 2 at every level until remains two registers to process at the last level. As a result, the number of instructions is $\sum_{i=1}^{2^{(log_2^k-1)}-1} 2i$, where i is the number of levels. This process can be generalized and extended to an arbitrary number of k registers since k is a multiple of two. The general formula to calculate the total number of *min/max* instructions to perform our in-register SIMD Sort is:

$$k\ log_2^k + \sum_{i=1}^{2^{(log_2^k-1)}-1} 2i \quad | \quad i \in \mathbb{Z} \tag{1}$$

On the other hand, the related work [9] requires:

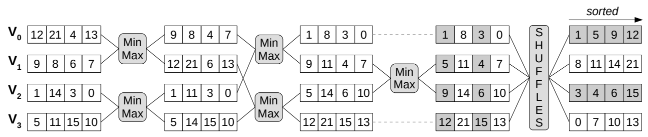
$$2(k-1+(k(log_2^k)(log_2^k-1))/4) \tag{2}$$

Table 2 compares the number of *min/max* instructions for both approaches in the target architectures. Notice that our SIMD sort algorithm requires less *min/max* instructions in both architectures.

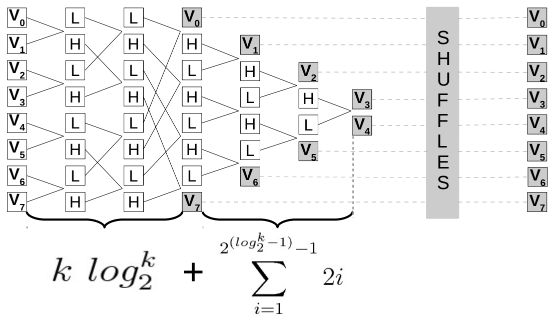
Table 2: Number of *min/max* instructions of the in-register SIMD sort computed by Equations 1 and 2.

Architecture	Register Length	Number of min/max inst.	
		In-register SIMD Sort	Related Work [9]
AVX512-64B	16 lanes of 4B	120	126
PIM-256B	64 lanes of 4B	880	1918

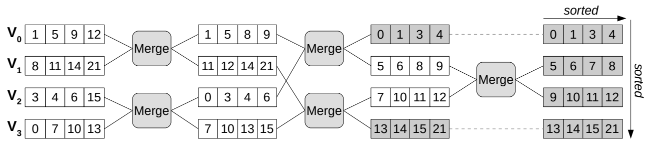
After the in-register SIMD Sort, our in-block-register Merge combines sorted registers to produce an overall sorted block of k registers. In Figure 8c, the resulting sorted registers (four registers, $k = 4$) of Figure 8a are the input to the in-block-register Merge. The execution uses an odd-even network to shuffle the registers and applies the *bitonic merge* to compare two individual sorted ones, producing, after all, a sorted block of k registers, i.e., $V_0 \leq V_1 \leq .. \leq V_{n-1} \leq V_n$. The execution of the merge phase uses the multiway merge from related work [9, 21] to boost parallelism.



(a) In-register SIMD Sort example.



(b) The number of instructions of the in-register SIMD Sort.



(c) In-block-register Merge for individual sorted registers.

Figure 8: SIMD Sort example and number of instructions, and the in-block-register Merge to combine sorted registers.

Sort-Merge Join. Now we are ready to discuss the implementation of the Sort-Merge Join algorithm with the SIMD Sort-Merge Operation. The data structure of the join column is of the form "key and object-id" in all of our join implementations. In the particular case of the Sort-Merge-Join, we sort the column by the key using our SIMD sort algorithm, with the addition of one SIMD permutation instruction after the comparison operation to reflect the sort in the object-id, as implemented by related work [20]. With the two relations sorted, we apply the multiway merge to conclude the operation [21, 29].

5. RESULTS & ANALYSIS

In this section, we present the results of our SIMD query operators in PIM and AVX512. Our goal is to understand the trade-offs between these two highly parallel architectures considering the effect of data movement around memory. We evaluated response time and energy consumption metrics, but we normalized these metrics to resume the data sets in one graphic for each operator. The execution environment is the same described in Section 3.1. We implemented the operators in C++ language and recorded their memory access pattern as input to the Assembly-like memory traces of the simulator. Our query execution design assumes the column-wise storage. Initially, we assume the materialization query execution system (e.g., MonetDB, VoltDB, and Hyrise), but we also discuss the impact of PIM on the pipelined (e.g., PostgreSQL, DB2) and vectorized (e.g., Vectorwise, Peloton) systems in Sections 5.5 and 5.6.

5.1 Selection Operator

Now, we report the results of the selection operator when exploiting the data access parallelism of the PIM-SIMD units. The selection operator applies the predicates of the TPC-H Query 03. We adjusted the size of the columns in our memory traces to fit data into the LLC and the DRAM.

We observe that the selection with PIM outperforms the AVX512 execution with at least 4 active vaults. It reaches the best execution when all the 32 vaults are activated in parallel (see Figure 9). Therefore, regardless of the size of datasets, the selection operator processes at least 3× faster with PIM than AVX512. With more on-chip processing, PIM uses around 45% less energy than AVX512. This high reduction in energy consumption varies little with a different number of vaults or the size of the datasets.

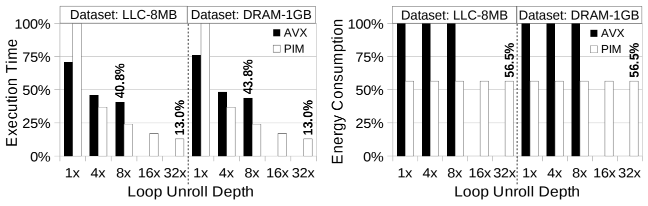


Figure 9: Normalized execution time and energy consumption of the selection operator varying the size of the datasets, and level of parallel processing.

5.2 Projection Operator

As discussed in Section 3.1, the projection operator is responsible for the materialization of intermediate data moving large amounts of data around the memory hierarchy. We observe the same results in Figure 10. The execution time of the projection on datasets of the same size as LLC-8MB or less is 7× on average faster in PIM than AVX512. For datasets that do not fit in the caches, e.g., the DRAM-1GB dataset, the execution time is one order (10×) of magnitude faster with PIM. Also, in all datasets, PIM reduces energy consumption in 3× compared to the AVX512. For instance, in the dataset DRAM-1GB, the execution of the AVX512 unrolled 8× spent 1, 913 Joules of energy (see Figure 10). On the other hand, the processing in PIM with all vaults, i.e., PIM-256B 32×, generated 0, 645 Joule of energy (an energy reduction of 3× compared to AVX512).

We conclude that pushing the selection and projection operators to PIM has a significant advantage over the x86 processor. In-memory parallel processing of PIM devices overcomes the processing power of the x86 due to the latency to move data around the caches.

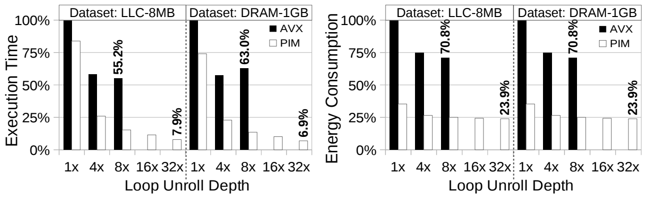


Figure 10: Projection operator in the datasets LLC-8MB and DRAM-1GB with the execution time and energy consumption normalized to AVX512 1×.

5.3 Join Operator

In the literature, the join operator has many algorithms, but most of them are variances of the nested loop, hash, and sort-merge join. Here, we analyze these SIMD implementations in PIM and AVX512.

5.3.1 Nested Loop Join

The Nested Loop Join (NLJ) algorithm traverses the join columns with two loops: the outer and the inner. In our implementations, the latter is unrolled up to 32× for PIM and 8× for the AVX512 execution. The goal is to exploit the highest levels of parallel processing and memory access to the devices. Figure 11 shows the results for datasets smaller or equal to the L2 cache (L2-256KB). The AVX512 execution unrolled 4× performs better than the PIM execution. The AVX512-style processing re-accesses data in caches for every inner loop iteration, while the inner column fits into the caches resulting in high data reuse (except by the first interaction that causes compulsory memory misses). In contrast, the PIM execution causes compulsory *load* / *store* for every inner loop iteration to access the memory banks at all times.

The PIM execution becomes appealing for datasets bigger than the L2 cache (e.g., LLC-8MB), which inhibit data reuse. The best AVX512 execution unrolled 8× spends 3.367 milliseconds to process the LLC-8MB dataset, whereas the PIM unrolled 32× requires 2.428 milliseconds: A reduction of 30% of the execution time. Moreover, the PIM processing saves around 50% of energy consumption in both datasets.

In practice, DBMSs choose the NLJ algorithm only to process small datasets, and for this reason, we suppressed the results for datasets bigger than the LLC cache. Moreover, we analyze the NLJ because it resembles the data access pattern of matrix multiplication that encompasses other applications, such as linear transformation, image processing, and machine learning algorithms. Our analysis on the NJL adds useful insights to that range of applications.

5.3.2 Hash Join

The hash join algorithm consists of the build and probe phases. These phases have two different memory access patterns: sequential memory access to read the join columns and random memory access to access the hash table entries. The build phase generates the hash table from the smallest relation. For instance, the TPC-H Query 03 generates two

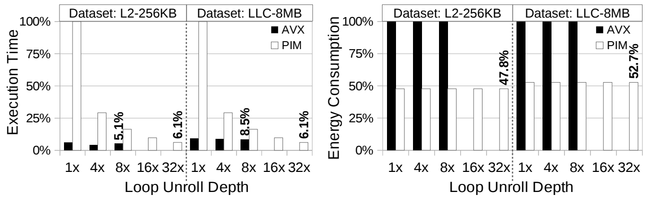
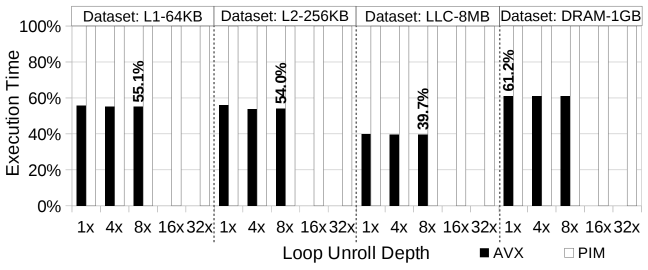


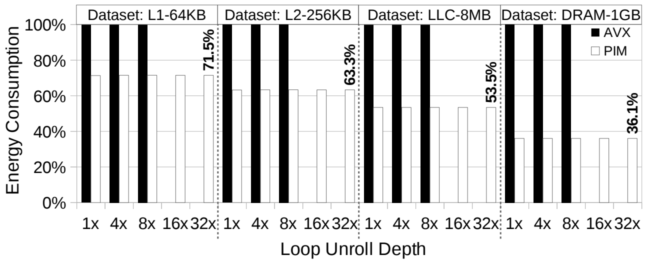
Figure 11: Normalized execution time and energy consumption of the NLJ algorithm, varying the size of the datasets and level of parallel processing.

hash tables for two join operations in the query plan. In the 1GB TPC-H, the hash table on "*c_custkey*" has 30,142 entries with a 173-KB memory footprint. The hash table based on "*o_orderkey*" has 147,126 entries with a 287-KB memory footprint. The probe phase searches the biggest relation to add the join values to the hash table.

Figure 12a presents the normalized execution time for the hash join. For all dataset sizes, the AVX512 execution is better than PIM. Two main effects impact the PIM execution: 1) Random access is sparse most of the execution, which means that only one register lane will be useful during PIM *load* operations. 2) Random access shall reuse some cache lines inside the x86 processor, although the reuse ratio may vary depending on the workload and cache size.



(a) Hash Join: execution time normalized to PIM-256B.



(b) Hash Join: energy consumption normalized to AVX512.

Figure 12: Normalized execution time and energy consumption of the Hash Join algorithm, varying the size of the datasets and level of parallel processing.

Figure 12a depicts the random access pattern problem. PIM unrolled 1× and 32× have the same performance, which means that regardless of the levels of parallelism used, the memory access serialization dictates the performance. However, the PIM execution reduces energy consumption in all datasets. Figure 12b shows that the energy savings by PIM increases as the hash table becomes bigger. The reasons behind those results are because AVX512 with bigger datasets generates more data movement than PIM to access the hash table entries from the main memory.

5.3.3 *Sort-Merge Join*

The execution of the Sort-Merge Join presents two different memory access patterns from its two phases. The first phase generates random memory access when sorting the join columns, while the second phase generates sequential memory access when merging the sorted columns. Figure 13 presents the execution results using as much parallelism as possible. We use an unroll depth of 8× in both PIM and AVX512, because the SIMD sort-merge algorithm reserves SIMD registers to hold intermediate values, such as lowest/highest values from *min/max* instructions and others from the *shuffle* instructions.

The execution of the AVX512 performs better while the datasets fit into the caches due to faster data access, as observed in both metrics: time and energy. The execution time remains smaller in the AVX512 execution. However, the energy consumption is higher with datasets bigger than the LLC due to the data movement. In those cases, the PIM uses around 40% less energy than AVX512.

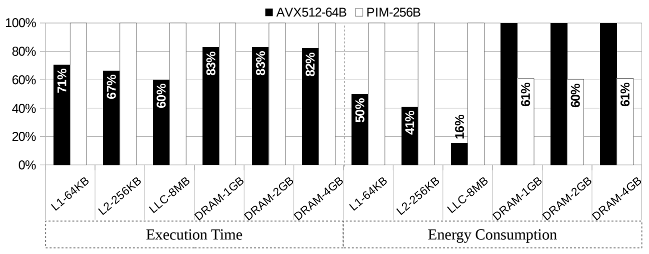


Figure 13: Sort-Merge Join: normalized execution time and energy consumption on the target architectures with unroll depth of 8×, varying the size of the datasets.

5.3.4 *Discussion*

In brief, the AVX512 overcomes PIM in terms of the execution time in the hash and sort-merge join. The PIM execution saves more energy avoiding off-chip data movement. Another significant join algorithm is the radix-join [33], which could be evaluated to reduce the energy consumption of the AVX512. Roughly, the radix-join has two distinct data access patterns: a random pattern while building radix-clusters for both join relations and a sequential one to probe the clusters with a nested-loop [33]. The random access pattern is also present in the hash join experiments, where our recommendation is to use PIM for energy saving. The sequential memory access is the same pattern evaluated in the NLJ experiments, in which PIM saves around 50% of the energy consumption, even in a dataset fitting in the L1 cache. The main reason for energy waste is the off-chip data movement. In our evaluation, such a factor shall not reduce with radix-join because its memory access patterns are already present in the experiments of the other algorithms. However, radix-join is a compelling case for future work.

All in all, we conclude that the performance of the join operator is very susceptible to the cache settings, the dataset size, and the target performance metric. The AVX512 execution benefits from the caching mechanism when the join columns fit into the caches or during random memory accesses, which enables data reuse inside the caches.

5.4 Aggregation Operator

The aggregation operator is based on a hash table to hold the aggregation values. It has two memory access patterns: 1) Data streaming while accessing the group columns to compute the hash addresses and the aggregation columns to accumulate the new values; 2) Random memory access while looking up the hash table. In this experiment, the limited number of PIM-SIMD registers restricts the data access parallelism to 16× to build the aggregation and groups.

5.4.1 *Query 01*

The aggregation operator in the TPC-H Query 01 has two columns for grouping, and eight aggregation functions based on five columns from the *Lineitem* table. With a small number of groups, i.e., hash table entries, the hash table also has a small memory footprint that fits into the L1 cache. Although the operator streams the five columns to compute the aggregation functions, Figure 14 shows that the memory access to the hash table dictates the performance regardless of the degree of parallelism used by the PIM device. With an unroll depth of 2×, the *gather* instruction of the AVX512 accesses two cache lines that are sufficient to load the entire hash table to SIMD registers outperforming PIM with all unroll depth versions.

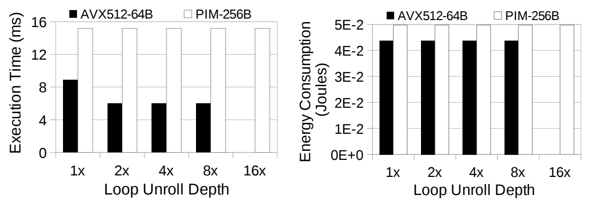


Figure 14: Aggregation: TPC-H Query 01.

5.4.2 *Query 03*

The aggregation operator in the TPC-H Query 03 has three columns of grouping and just one aggregation function based on two columns from the *Lineitem* table. The number of hash table entries is a few hundred, which still fit in the L2 cache. This fact leads the PIM to scale according to the degree of parallelism. The difference in performance decreases between PIM and AVX512 compared to the results of Query 01. However, the execution of the aggregation remains better in AVX512 (see Figure 15).

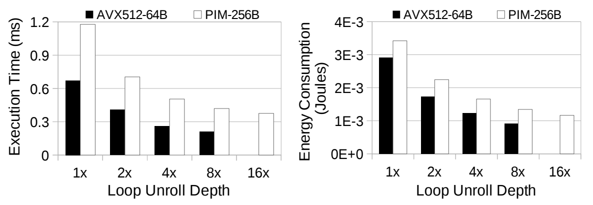


Figure 15: Aggregation: TPC-H Query 03.

5.4.3 *Zipf Distribution*

In the previous experiments with TPC-H queries 01 and 03, the hash table fit into the L1 and L2 caches. Now, we investigate the aggregation operator with the Zipf workload varying the size of the dataset using bigger sizes than the cache memories. The Zipf distribution was also used by the related work [41] to evaluate the aggregation operator.

Figure 16 shows that the execution time using AVX512 is still better than PIM and that the difference in energy consumption is quite marginal. The AVX512 performance results come from the high reuse of the hash table, especially for small hash tables that fit into the caches. The random access to the hash table restricts the data access parallelism of the PIM device, incurring in the same effects observed in the hash join (see Section 5.3.2), i.e., low usage of SIMD register lanes and x86 cache memory reuse.

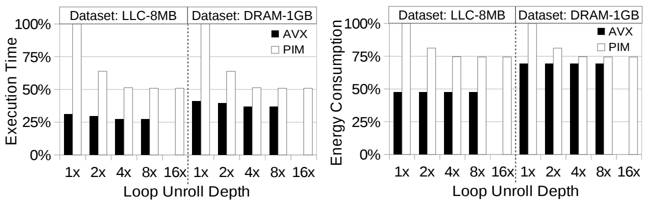


Figure 16: Normalized execution time and energy consumption of the aggregation operator with the Zipf distribution varying the size of the datasets and level of parallelism.

We conclude that the hash table access pattern dictates the performance of the aggregation operator regardless of the performance metric. The random access shows low data reuse as at most 32 memory addresses from the 64 possible addresses in the SIMD lanes can be accessed at once. In this case, hashing will require two loads to compute the hash keys if the unroll depth is set to 32×. As a remark, we did not consider, in this study, aggregations without grouping, i.e., no hash table, because it is a corner case in analytic workloads that we keep open for future work.

5.5 Pipelined vs. Vectorized Query Execution

In this section, we compare the pipelined and vectorized query execution systems. We implemented the selection vector and bitmap data structures to support the execution of both systems. In the pipelined execution, the selection operator uses those data structures to hold intermediate results in SIMD registers as long as possible, avoiding data re-access. These results are used by the next operators to filter columns along the pipeline. In the vectorized execution, the selection operates on vectors of 1024 elements[4] and stores the intermediate data structures into the memory to be loaded by the next operator in the query plan. Those *store/load* instructions are the main factor that differs between the implementation and performance of these query execution systems.

We analyze the selection operator that is followed by the build phase of a hash join. We noticed an opportunity to fuse these two operators in the TPC-H Query 03 query plan, the selection filter "c_mktsegment = 'BUILDING'", and the build of the hash table on *c_custkey* because there is no pipeline breaker [37] between them. Therefore, SIMD registers hold an intermediate selection vector that is used to filter the *c_custkey* column (*gather* instruction). Keeping the selection vector in SIMD registers precludes the exploitation of the maximum 32× data access parallelism of PIM. In our implementation, 16 SIMD registers hold the selection column (*c_mktsegment*), while the selection vector uses the remaining registers.

Figure 17 presents the execution time and energy consumption of the pipelined and vectorized execution. Results show that the pipelined execution performs better than the vectorized in both architectures due to the additional *store/load* instructions on the selection vector. PIM reduces execution time of the pipelined system when the 32 vaults are activated. In the AVX512 hardware, we observed almost 50% of energy saving due to the high selectivity of the selection vector that filters around 80% of the join column, and also the random access pattern to build the hash table.

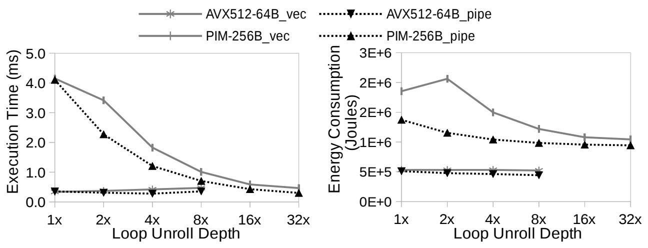


Figure 17: Pipelined vs Vectorized execution of the TPC-H Query 03 with a selection operator followed by building.

Now, we analyze the selection operator followed by aggregation in the query plan of the TPC-H Query 01. The selection predicate filters a small subset (around 1.5%) of the *Lineitem* table, and 98.5% remains to aggregate. The selection operator outputs a bitmap of bytes instead of a selection vector due to the low selectivity of the selection predicate. In this query plan, the pipelined execution with a bitmap as an intermediate structure achieves the maximum degree of parallelism of PIM overcoming the AVX512 processing. The selection operator reads data from all vaults to apply the selection predicate. This strategy compensates for the random memory access of the hash table. In the vectorized execution, SIMD registers hold the bitmap to build the grouping and aggregation columns using the selective load instruction. The aggregation operator applies the conflict-free updates technique [17] to mitigate the concurrence to the hash table. Figure 18 shows a marginal improvement to run a selection followed by aggregation on PIM. The vectorized and pipelined executions are worth when at least 4 or 16 vaults are activated, respectively.

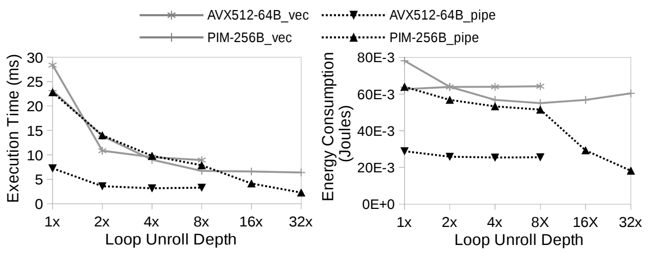


Figure 18: Pipelined vs Vectorized execution of the TPC-H Query 01 with a selection operator followed by aggregation.

We conclude that random memory patterns hamper the data access parallelism of PIM in both execution systems. This shows opportunities to re-design hash-based algorithms for PIM hardware.

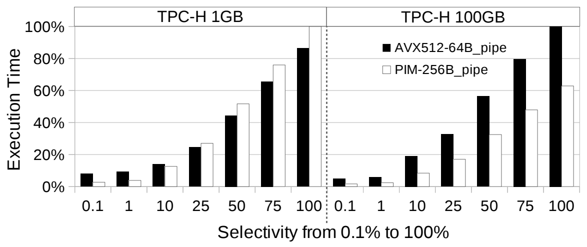
[4]The same quantity defined by related word [5].

5.6 The Effect of Selectivity

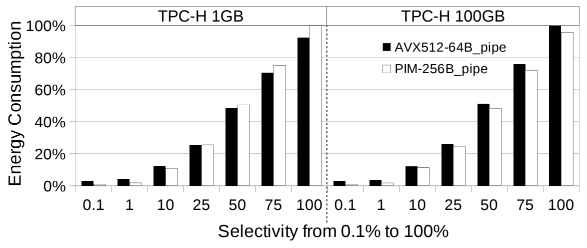
For a more holistic macro-benchmark examination, we evaluate the effect of the selectivity of the TPC-H Query 03 in the pipelined query system (the best performance of AVX512, as observed in Section 5.5). We randomly ranged the selectivity of the *c_mktsegment* between 0.1% and 100%. Varying the selectivity on the pipelined system implies to change the size of the selection vector and the projectivity on column *c_custkey*, and also the cardinality of the join, i.e., the number of entries in the hash table.

Figure 19 shows our findings. For selectivities between 0.1% and 10% on small datasets (e.g., TPC-H 1GB), PIM reaches a better performance in both metrics compared to the AVX512 because the selectivity reduces the hash table size alleviating the memory access serialization. For selectivities greater or equal to 25%, the AVX512 outperforms PIM due to two main reasons: 1) The hash table has more entries that imply a higher join cardinality and more memory access serialization; 2) The dataset fits into the caches leading to data reuse. However, for big datasets PIM is faster regardless of the selectivity because the input columns, selection vector, and hash table do not fit into the caches at the same time. In the TPC-H 100GB, the execution time of PIM ranges from 1.6× to 3× faster than the AVX512 varying the selectivity from 100% to 0.1%, respectively. Likewise, PIM uses 5% to 70% less energy than AVX512.

We conclude that the main factors to decide for PIM in the pipelined query execution system are the cache settings, the size of the dataset and the intermediate data structures.



(a) Varying Selectivity: normalized execution time.



(b) Varying Selectivity: normalized energy consumption.

Figure 19: Normalized time and energy of the 1GB/100GB pipelined system varying the selectivity from 0.1%∼ 100%.

6. HYBRID PIM-X86 SIMD PROCESSING

In this section, we discuss the potential of a Hybrid PIM-x86 SIMD query scheduler. We assume the materialization system and present our discussion with the execution of a macro-benchmark of TPC-H Query 03. Figure 20 shows the execution breakdown in the DRAM-1 GB dataset with the best execution of each operator in the target architectures. For instance, we choose the hash join that showed the best performance among the join algorithms. Processing the PIM-specific query plan improves the execution time by 12.5% and spends 66% less energy than the AVX512-specific query plan. This result matches the energy efficiency featured by commercial PIM architectures.

In Table 3, we correlate the results of the operators presented in Section 5 with their best processing architectures according to the dataset size and performance metric. We take into account these results to implement the heuristics that coordinates the execution of operators between PIM and x86. The hybrid query plan reduces the execution time by 35% and 45% compared to both hardware-specific PIM and AVX512 plan execution, respectively. For energy consumption, the hybrid query plan consumes less than half of the AVX512 consumption but presents a marginal result compared to the PIM plan. All in all, the hybrid query scheduler presents promising results to foster new developments of many-core DBMSs.

Table 3: Summary of the Query Operators.

Operator		Dataset **Fit in cache?**	Performance **Metrics**	Processing **Architectures**
Selection		no/yes	time/energy	**PIM**
Projection		no/yes	time/energy	**PIM**
Join	Nested Loop	L1/L2	time	**AVX512**
		LLC	time	**PIM**
		yes	energy	**PIM**
	Hash Join	no/yes	time	**AVX512**
		no/yes	energy	**PIM**
	Sort Merge	no/yes	time	**AVX512**
		yes	energy	**AVX512**
		no	energy	**PIM**
Aggregation		no/yes	time/energy	**AVX512**

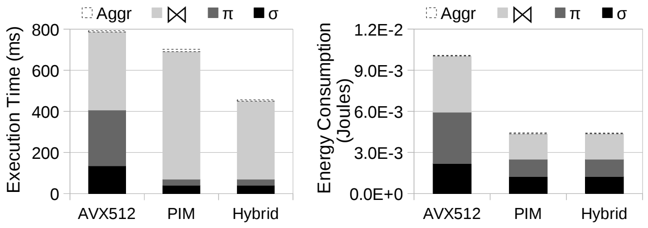


Figure 20: Execution breakdown when applying our findings to process a hybrid query plan for Query 3.

Regarding energy results, we observe several trade-offs whenever moving computation from the x86 to the PIM logic: 1) We expect that the functional units (ALU's) and the number of data accesses will consume the equivalent amount of energy. During data streaming, both x86 and PIM execution shall process an equal quantity of computing operations, spending the same amount of energy per operation no matter the hardware; 2) We can significantly save energy reducing off-chip data transfers, as they consume up to 71.5% of the total system energy budget [6]; 3) We also reduce the energy consumption of the cache subsystem with less data being stored and evicted from the cache memories. The energy consumption of the cache subsystem accounts for 25% to 50% of the full processor energy consumption [2]; 4) We expect that energy consumption increases to send the instructions inside the memory. On-chip processing requires extra hardware to handle the instructions and the messaging of their status to the CPU. However, the payload of instructions and messages to the CPU are much smaller than a cache line, resulting in a positive energy trade-off.

7. RELATED WORK

Data Analytics For PIM Architectures. Pioneer works have proposed PIM-based architecture for data analytics based on simulation. MapReduce (MR) applications with high spatial locality were adapted to PIM [42,43] leading memory cores to reduce the latency by 93.2%. This work is orthogonal to our results because the reported MR jobs have a resembling access pattern of the selection and projection operators. Another work [32] relies on an analytic model to estimate the latency of 3D-stacked memories through scan-aggregate queries. They presented improvements in latency and energy consumption against traditional CPU processing and big-memory servers. However, they only consider the dataset size variation in their analysis. In our work, we evaluate and argue more intricate factors, such as memory access patterns in caches. Mondrian [14, 15] implements an algorithm-hardware co-design for near-memory processing of data analytics operators. It is built upon a partitioning phase to turn random accesses to sequential ones, enabling thus a memory streaming hardware to exploit PIM capabilities. The presented results are complementary to ours. They ratify that sequential access favors PIM and show that random access is an obstacle to use the whole bandwidth. Mondrian considers algorithms with a PIM-tuned partitioning and probe phase. Instead, we evaluate pure query operators leading to a conclusion that they shall be optimized to benefit from PIM.

Flash Disks. Recently, flash disks also brought attention to accelerate [16] and save energy [31] of scan and join operators. However, current works have two downsides: 1) They rely on dedicated database hardware. Smart SSDs [13] use an embedded ARM processor into the SSD with a firmware for communication to evaluate the execution of database operators. Intelligent SSDs [10] add a reconfigurable stream processor to reach high processing performance with energy savings; 2) They are application-driven without a general interface to abstract hardware features. Active Flash [46] offloads particular functions of scientific workloads to run into the SSDs.

PIM As Query Accelerator. Recent works use PIM devices as isolated accelerators to boost query operators, such as selection [44, 48], projection [47], and join [35]. However, this one-sided approach is simplistic and neglects the potential of CPU-PIM co-processing with caching and energy-saving benefits.

Scheduling On Emerging Hardwares. Current intra-query scheduling focused on co-processing between GPU and CPU to improve execution time based on runtime learning model [7] and operator cost model [24]. A similar hybrid co-processing was tested in the Intel Xeon Phi co-processor [8]. However, this hybrid co-processing tackles compute-intensive applications and neglects the potential of PIM to run data-intensive ones.

Kernel Scheduling on PIM-Assisted GPU. Related work in GPU architectures proposed scheduling techniques with PIM devices installed as GPU main memory. GPU applications are split into independent GPU-kernels and interleave the processing of each kernel between the GPU cores and the PIM device [11,18,39]. Although GPUs are devices with high parallel processing degree, data still need to be transferred around the memory hierarchy before moving to the GPU-PIM device. In this paper, we focus on in-memory processing with data transfer only when needed.

8. CONCLUSION & FUTURE WORK

In this paper, we present results from extensive experiments on database SIMD operators over three distinguishable query execution systems: materialized, vectorized, and pipelined. Our experiments evaluated the SIMD operators on the widest SIMD architecture of modern x86 processor (i.e., AVX512), against a modern Processing-in-Memory (PIM) architecture that supports SIMD registers of 256-bytes wide. We gauged the execution time and energy consumption on more diverse datasets than previous studies [35, 44, 47, 48].

We have identified that the execution of the selection and projection query operators are more suitable to PIM, while the aggregation operator performs best in the x86 processing. However, the choice of the processing architecture for the join operator is notably intricate and depends on the join memory access pattern and the size of the dataset. All in all, the AVX512 processing presents the best result when the dataset fits in the L1 or L2 caches (due to the data reuse). The PIM processing presents the best result, regardless of the dataset size, considering the energy consumption to move data around the memory.

One valuable contribution of our experimental study appears when analyzing the hash join algorithm. We uncovered the effects of low usage of SIMD register lanes and also the data reuse that appears in the x86 processing. These problems inhibit the data access parallelism and processing capabilities of PIM, degrading the performance of applications that generates massive random memory access during the execution. Another contribution of this paper is our SIMD sorting algorithm that requires fewer SIMD instructions than the state of the art in both PIM and AVX512 architectures. We observed that our SIMD sorting algorithm presented the best results when executed by the AVX512.

Also, we evaluated the vectorized and pipelined query execution systems. The pipelined system is susceptible to varying the selectivity and, consequently, projectivity and join cardinality. However, the results showed that when the dataset and intermediate data do not fit into the caches at the same time; the PIM execution is up to 3× faster and spends 70% less energy than the AVX512 processing.

As future work, we are currently building a hybrid query plan scheduler between PIM and x86 processing. It showed promising results reducing the execution time between 35% and 45% compared to hardware-specific query plans, and saving 2× more energy than the x86 query plan.

9. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful feedback. This work was supported by CAPES, AWS Cloud Credits for Research program, NVIDIA Corporation, and Serrapilheira Institute (grant number Serra-1709-16621).

10. REFERENCES

[1] A. Acharya, M. Uysal, and J. H. Saltz. Active disks: Programming model, algorithms and evaluation. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 81–91, 1998.

[2] M. A. Z. Alves. *Increasing energy efficiency of processor caches via line usage predictors*. PhD thesis, Federal University of Rio Grande do Sul, Brazil, 2014.

[3] M. A. Z. Alves, M. Diener, P. C. Santos, and L. Carro. Large vector extensions inside the HMC. In *2016 Design, Automation & Test in Europe Conference & Exhibition*, DATE, pages 1249–1254, 2016.

[4] M. A. Z. Alves, C. Villavieja, M. Diener, F. B. Moreira, and P. O. A. Navaux. Sinuca: A validated micro-architecture simulator. In *IEEE International Conference on High Performance Computing and Communications*, HPCC, pages 605–610, 2015.

[5] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *Second Biennial Conference on Innovative Data Systems Research*, CIDR, pages 225–237, 2005.

[6] A. Boroumand, O. Mutlu, and et al. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pages 316–331, 2018.

[7] S. Breß, S. Mohammad, and E. Schallehn. Self-tuning distribution of db-operations on hybrid CPU/GPU platforms. In *GI-Workshop "Grundlagen von Datenbanken"*, pages 89–94, 2012.

[8] X. Cheng, B. He, M. Lu, and C. T. Lau. Many-core needs fine-grained scheduling: {A} case study of query processing on intel xeon phi processors. *J. Parallel Distrib. Comput.*, 120:395–404, 2018.

[9] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *PVLDB*, 1(2):1313–1324, 2008.

[10] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger. Active disk meets flash: a case for intelligent ssds. In *International Conference on Supercomputing*, ICS, pages 91–102, 2013.

[11] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hetexchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *PVLDB*, 12(5):544–556, 2019.

[12] D. J. DeWitt and P. B. Hawthorn. A performance evaluation of data base machine architectures (invited paper). PVLDB, pages 199–214, 1981.

[13] J. Do, Y. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart ssds: opportunities and challenges. In *International Conference on Management of Data*, SIGMOD, pages 1221–1230, 2013.

[14] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. N. Pnevmatikatos. The mondrian data engine. In *International Symposium on Computer Architecture*, ISCA, pages 639–651, 2017.

[15] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. N. Pnevmatikatos. Algorithm/architecture co-design for near-memory processing. *Operating Systems Review*, 52(1):109–122, 2018.

[16] G. Graefe, S. Harizopoulos, H. A. Kuno, M. A. Shah, D. Tsirogiannis, and J. L. Wiener. Designing database operators for flash-enabled memory hierarchies. *IEEE Data Eng. Bull.*, 33(4):21–27, 2010.

[17] T. Gubner and P. A. Boncz. Exploring query compilation strategies for jit, vectorization and SIMD. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures*, ADMS@VLDB, pages 9–17, 2017.

[18] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. Transparent offloading and mapping (TOM): enabling programmer-transparent near-data processing in GPU systems. In *International Symposium on Computer Architecture*, ISCA, pages 204–216, 2016.

[19] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[20] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In *nternational Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 189–198, 2007.

[21] H. Inoue and K. Taura. SIMD- and cache-friendly algorithm for sorting an array of structures. *PVLDB*, 8(11):1274–1285, 2015.

[22] Intel.Corporation. Intel Intrinsics Guide, 2019. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.

[23] J. Jeddeloh and B. Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *Symposium on VLSI Technology*, VLSIT, 2012.

[24] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner. Heterogeneity-aware operator placement in column-store DBMS. *Datenbank-Spektrum*, 14(3):211–221, 2014.

[25] W. H. Kautz. Cellular logic-in-memory arrays. *IEEE Trans. Computers*, 18(8):719–727, 1969.

[26] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Record*, 27(3):42–52, 1998.

[27] T. R. Kepe. Dynamic database operator scheduling for processing-in-memory. In *VLDB PhD Workshop*, volume 2175 of *CEUR Workshop Proceedings*, 2018.

[28] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. A. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *PVLDB*, 11(13):2209–2222, 2018.

[29] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *PVLDB*, 2(2):1378–1389, 2009.

[30] J. Kim and Y. Kim. HBM: memory solution for bandwidth-hungry processors. In *IEEE Hot Chips 26 Symposium*, HCS, pages 1–24, 2014.

[31] S. Kim, H. Oh, C. Park, S. Cho, and S. Lee. Fast, energy efficient scan inside flash memory. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures*, ADMS@VLDB, pages 36–43, 2011.

[32] J. Lowe-Power, M. D. Hill, and D. A. Wood. When to use 3d die-stacked memory for bandwidth-constrained big data workloads. *CoRR*, abs/1608.07485, 2016.

[33] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[34] R. C. Minnick, R. A. Short, J. G. H. S. Stone, and M. W. Green. Cellular arrays for logic and storage. Technical report, Apr 1966.

[35] N. S. Mirzadeh, O. Kocberber, B. Falsafi, and B. Grot. Sort vs. hash join revisited for near-memory execution. In *Workshop on Architectures and Systems*, ASBD@ISCA, 2015.

[36] O. Mutlu. Tutorial on memory systems and memory-centric computing systems: Challenges and opportunities. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, Tutorial@SAMOS, 2019.

[37] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[38] D. A. Patterson, T. E. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. E. Kozyrakis, R. Thomas, and K. A. Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, 1997.

[39] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das. Scheduling techniques for GPU architectures with processing-in-memory capabilities. In *International Conference on Parallel Architectures and Compilation*, PACT, pages 31–44, 2016.

[40] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD vectorization for in-memory databases. In *International Conference on Management of Data*, SIGMOD, pages 1493–1508, 2015.

[41] O. Polychroniou and K. A. Ross. High throughput heavy hitter aggregation for modern SIMD processors. In *International Workshop on Data Management on New Hardware*, DaMoN, page 6, 2013.

[42] S. H. Pugsley, J. Jestes, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. Comparing implementations of near-data computing with in-memory mapreduce workloads. *IEEE Micro*, 34(4):44–52, 2014.

[43] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. NDC: analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads. In *International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 190–200, 2014.

[44] P. C. Santos and et al. Operand size reconfiguration for big data processing in memory. In *Design, Automation & Test in Europe Conference & Exhibition*, DATE, pages 710–715, 2017.

[45] H. S. Stone. A logic-in-memory computer. *IEEE Trans. Computers*, 19(1):73–78, 1970.

[46] D. Tiwari, S. Boboila, and et al. Active flash: towards energy-efficient, in-situ data analytics on extreme-scale machines. In *USENIX conference on File and Storage Technologies*, FAST, pages 119–132, 2013.

[47] D. G. Tome, T. R. Kepe, M. A. Z. Alves, and E. C. de Almeida. Near-data filters: Taking another brick from the memory wall. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures*, ADMS@VLDB, pages 42–50, 2018.

[48] D. G. Tome, P. C. Santos, L. Carro, E. C. de Almeida, and M. A. Z. Alves. HIPE: HMC instruction predication extension applied on database processing. In *Design, Automation & Test in Europe Conference & Exhibition*, DATE, 2018.

[49] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 8(3):209–220, 2014.