Processing in Storage Class Memory

Joel Nider
University of British Columbia

Craig Mustard
University of British Columbia

Andrada Zoltan
University of British Columbia

Alexandra Fedorova
University of British Columbia

Abstract

Storage and memory technologies are experiencing unprecedented transformation. Storage-class memory (SCM) delivers near-DRAM performance in non-volatile storage media and became commercially available in 2019. Unfortunately, software is not yet able to fully benefit from such high-performance storage. Processing-in-memory (PIM) aims to overcome the notorious memory wall; at the time of writing, hardware is close to being commercially available. This paper takes a position that PIM will become an integral part of future storage-class memories, so data processing can be performed *in-storage*, saving memory bandwidth and CPU cycles. Under that assumption, we identify typical data-processing tasks poised for in-storage processing, such as compression, encryption and format conversion. We present evidence supporting our assumption and present some feasibility experiments on new PIM hardware to show the potential.

1 Introduction

Problem Storage-class memory (SCM), also known as non-volatile memory (NVRAM) is starting to be adopted as a long-term storage media, as a complement, or even as an alternative to SSD and HDD. It is a class of memory technologies (including STT-MRAM [20], PCM [22], ReRAM [30] and FeRAM [4]) whose defining feature is that data persists across power cycles. It also features a high level of parallelism that is inherent in the design, due to its physical properties. Often, memories in this class are also byte addressable, although not always used in the manner.

Over the past two decades, performance of storage hardware has increased two orders of magnitude. First, with the introduction of SSDs (solid state drives) then with the transition from SATA to PCIe and most recently with the innovation in non-volatile memory technology. Last year, Intel released Optane DC Persistent Memory [8], which is built on PCM with 3D-XPoint technology and sits directly on the memory bus and further reduces I/O latency.

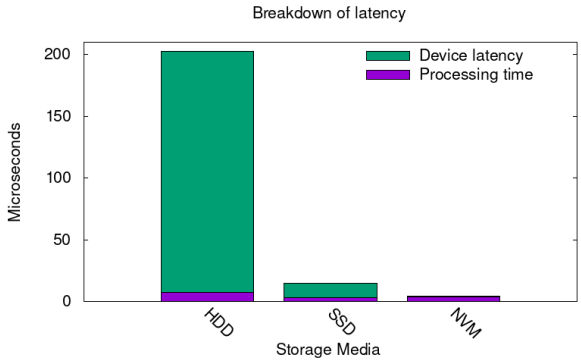

Figure 1: Latency breakdown of a 4KB block random read on various storage hardware technologies

At the same time, computation capabilities have not increased at the same rate due to the end of Moore's Law and Dennard scaling. This means that CPU cycles are too precious to waste on I/O processing, especially since I/O processing overhead is increasing at a faster rate than CPUs can cope with. Figure 1 illustrates that while device access time used to dominate I/O latency, the cost of navigating the software stack is becoming more crucial as device access time shrinks. As storage performance increases, the time spent on processing the data read from the storage media is becoming a more significant percentage of the overall access time. Similarly as storage density increases, the amount of CPU resources required to process data at this higher rate also increases, eventually becoming the bottleneck in an otherwise scalable system. To overcome the problem of scalability in the storage subsystem, compute resources must scale with storage capacity. We believe this will lead to PIM technologies being integrated with SCM.

Solution Accelerators have been proposed at all levels of the storage hierarchy; from smart caches to processing in storage. Processing-in-memory (PIM) architectures [27] have

been proposed to either overcome the limitations of memory bandwidth (i.e. "memory wall" problem) [11,19,33], or reduce energy consumption in data movement [2]. PIM processors are tightly coupled with the RAM, often being implemented on the same die, which has the unique property of scaling the processing capability with the amount of available memory.

We make the supposition that hardware will emerge that uses PIM processors to improve SCM scalability. These embedded PIM processors are not a drop-in replacement for the host CPU, because they exist in a severely limited environment. Highly constrained die area and a limited power envelope dictate the capabilities of the PIM processors, which in turn affect the scope of the software which they can execute. PIM on SCM shares many of these constraints with PIM on DRAM, but since SCM is essentially storage, *the kind of operations performed by PIM on SCM will be different than that performed by PIM on DRAM*.

We see *use cases* as the main source of difference between PIM on SCM and PIM on DRAM. How the hardware is used should drive the design of both hardware and software. The simplest use case of PIM on SCM will be to perform data transformation between the format of stored data and its in-memory form. We propose to offload simple data processing and manipulation operations (e.g., compression) from the CPU to processors embedded within the SCM. This may encompass several scenarios such as a compressed cache for a block device, preparing data for long-term storage by adding checksums, filtering, or collecting statistics on the data. By offloading these tasks, the CPU will be freed to handle more latency-sensitive or computationally intensive operations. Performing these same tasks using PIM on DRAM would involve using the CPU or a DMA engine to transfer data to DRAM and then operate on it. PIM on SCM eliminates this data movement.

Benefits PIM on SCM makes sense because it offers many benefits. The following is a list of the most important ones.

- **Free the Bus** Since the SCM and in-storage processors are tightly coupled, the data does not have to move across a shared bus. Thus, we can avoid the von-Neumann bottleneck [18] on the memory bus between storage (SCM or disk) and RAM that current PIM on DRAM implementations suffer from. This is especially important with SCM, as many functions process more data than they return, transferring large amounts of data can be completely avoided.

- **Scalable Processing** As the size of the storage increases, processing capabilities should also increase in order to evenly scale the throughput of service requests in the system. PIM on SCM couples processors with storage, so as to guarantee linear scaling.

- **Post-processing of Stored Data** Many applications need to guarantee data durability by ensuring that the data has been written to persistent media. PIM on SCM can continue to transform the data in storage after it has become durable, possibly increasing the throughput of the application.

Contributions Our contribution is recognizing the opportunity to use PIM to solve scalability issues for low-latency persistent storage. We provide motivating experiments with two applications and present microbenchmarks on new PIM hardware to assess the feasibility of this idea. We believe that this approach warrants further investigation.

2 Background

PIM has been described in several hardware architecture papers [2, 3, 12, 14, 18, 19, 28, 32] which have explored various parameters in the design space. Our proposed hardware sits at the junction between *processing-in-memory* which generally refers to processors embedded in volatile memory, and *processing in storage* referring to processors embedded in block storage devices. We refer the reader to Siegl [27] for a more comprehensive survey and taxonomy of NDP (near data processing) including PIM and NMP (near memory processing). We view PIM as a collection of small processors embedded in the memory, which we refer to as *Data Processing Units* (DPU). Here we discuss some of the pertinent features of various related designs, and the associated trade-offs. We base our evaluations on the hardware built by UPMEM [10], which is likely to reflect their commercial offering. Even though the UPMEM hardware is implemented on DRAM, we would expect to see hardware very much like what UPMEM is producing in future persistent memory systems, perhaps with the additional features discussed below.

Inter-PIM Communication Several proposed designs feature a secondary bus (i.e. that does not connect to the host or main memory system) that is used to communicate data and control between the various DPUs, effectively making a distributed system. This has the advantage of being able to coordinate effort for dynamic load balancing at runtime, and sharing partial results to improve locality. The major drawback is the cost of integrating such a bus in silicon. Beyond the obvious costs of larger silicon area and energy consumption, such a bus also requires a more complicated front-end to handle loads and stores from multiple sources. This would also impact scalability because of the amount of communication required between processors.

Address Translation One of the requisite features for supporting virtual memory is an address translation function for converting virtual addresses to physical addresses. DPU support for virtual memory is very convenient because it means

pointers can be shared verbatim between the host CPU and the DPU. Memory translation can be expensive because it requires maintenance of the translation tables. Each process on the host CPU has its own set of tables, and it would become necessary to either share or duplicate these mappings for each DPU in order to support virtual memory. In addition, most DPU designs can only access a limited range of the physical address space. For these reasons, many PIM designs do not support address translation.

Core Density We refer to the ratio of DPUs to memory as the *core density* of the system. Higher core density means each DPU is responsible for less memory (i.e. more cores per given memory capacity). It is a tradeoff between silicon area and power consumption vs. parallelism. Not enough cores means limiting parallelism, while too many means needlessly consuming power and die area that could have been used for other purposes. One question that we hope to be able to answer is how to determine the "correct" core density for a given application.

Instruction Set DPUs must decode and execute instructions, just like any other processor. Some designs have reused an existing ISA from some embedded processor, while others have opted to design their own custom ISAs. The main advantage to an existing ISA is a mature ecosystem, meaning the software development toolchain (compiler, linker, debugger, etc) already exists and is familiar to developers. It is plausible that there may be binary compatibility between two processors that share an ISA. On the other hand, designing a new ISA means only the necessary instructions must be implemented, which can lead to a leaner design. This comes at a cost of having to develop new tools and provide training for developers.

3 Architecture and Limitations

The first step in evaluating feasibility and potential of our idea is to experiment with existing PIM hardware. Even though the UPMEM hardware is implemented on DRAM, SCM and DRAM have enough similarities so we may experiment freely without depending on simulation or emulation, which both have drawbacks. UPMEM DRAM is DDR4-compatible: it can be used as a drop-in replacement for DDR4 DIMMs. Their DPUs are general-purpose processors, which we believe to be a crucial feature for our problem. In the rest of this section we describe the architecture of the UPMEM PIM hardware and discuss its advantages and limitations. We conclude with the discussion of features that we believe would be beneficial to future PIM on SCM.

UPMEM augments DRAM chips by including a specialized implementation of a general-purpose processor (DPU) inside each chip, which has direct access to the DRAM memory. This architecture was designed so that it can be readily integrated into existing DRAM designs. The memory is divided into ranks, and each rank has a set of dedicated, multithreaded DPUs. That way, the number of DPUs is proportional to the size of the memory of the machine.

Before a DPU can compute on the data, the data must be copied by a DMA engine from DRAM to a 64KB private SRAM buffer. Each DPU has a large number of hardware threads that can be scheduled for execution, but because it is an interleaved multithreading (IMT) design [31] , only one thread can advance at each cycle. The large number of threads helps hide latency while moving data, since several DMA operations can progress concurrently. Since all of the memory chips are decoupled, all of the DPUs can process data in parallel, completely independent of one another. However, since there is no direct communication channel between DPUs, the host must carefully control the dataset and plan execution before processing begins because of the high cost of synchronization, which must be performed by the host.

Because the capabilities of the PIM processors are well below that of a common host CPU, it is necessary to ensure that the offloaded functions are simple enough to be implemented and executed efficiently on these embedded processors. Despite the simple design of the PIM processors, they are general purpose, and are applicable to a wide range of problems. They are easy to program since standard development tools (compiler, linker, debugger) are used. In many cases, code snippets can be ported directly from existing code. This is an important feature to encouraging adoption of the new technology in future memory designs. The fact that these processors use a C compiler means that engineers can feel comfortable with the programming environment, and get up to speed quickly.

3.1 Virtual Memory

DRAM [17] is organized into banks. In UPMEM memory the core density is one DPU for each bank of 64MB. The DPU can only access data in its bank. This creates a challenge, because when a host CPU writes a cache line to DRAM, that data is striped across multiple banks at the granularity of one byte – this is a common design in DRAM DIMMs to reduce latency. This striping is transparent to the host – as long as the data written is identical to the data read back. But when the host writes data that is meant for the DPU to read, the striping causes each DPU to see only every 8th byte of the data. In order to present the DPU with a contiguous dataset, the API functions must reorder the bytes before copying them into the memory attached to the DPUs. To be able to share the memory efficiently between the host and the DPUs, we need to be able to map virtual memory pages directly into the address space of the application, and access them as such. Currently this is not possible, due to the limitations incurred by striping. Mixing SCM and DRAM together in a single rank

would further exacerbate the situation, due to the differences in physical layout. Since the DPUs are permanently tied to a specific memory range, we are unable to use the usual method of migrating sectors used to implement wear levelling on SCM [25], and would be forced to use in-place methods [5].

3.2 Wishlist for PIM architectures

Based on our understanding of PIM on DRAM, we believe PIM on SCM would benefit from some additional features that would make the data flow easier to manage, and software easier to write.

Data Triggered Functions Similar to the concept of Data Triggered Threads [29] and EDGE [16], we propose the idea of *data triggered functions*. The idea is that the host CPU can trigger a function execution on the DPU by reading from a memory address owned by that DPU. The address is first registered on the DPU as a function target. When the CPU issues a memory load, it will trigger the DPU to execute a function, and stall until the function completes, at which time the memory will return the result. This will keep the program flow simple, as no expensive API calls would be needed in order to execute a function on the DPU, and no additional mechanism (polling, interrupt, etc.) would be required to know when the results are available. This would require a memory controller that can support the longer latency of executing a function. The NVDIMM-P specification [13] is expected to add support for non-deterministic access times to handle longer latencies associated with persistent memory, which may be able to also handle such an extreme case as function execution.

Concurrent Memory Access The PIM architecture must arbitrate between concurrent accesses to the same memory rank from the host CPU and the DPU. Currently, this is controlled by a register written by the host, to select the permitted bus master. This is a very coarse granularity, which means PIM cannot arbitrate between individual accesses from multiple sources. Effectively, it means that the DPU and CPU cannot operate on the same memory rank concurrently. In order to pipeline requests from software, it would be necessary to support a producer-consumer model while the DPU is processing buffer *N* the CPU could read the results of *N-1* and write the buffer *N+1*. This is used for double-buffering, or the more general case of a queue of buffers. This would require support from the memory bus, which is not possible with the commonly used DDR4. Memory fabrics such as OpenCAPI [7], GenZ [21] or CCIX [6] may provide the necessary support.

Mix of Memory Types Each DPU can access a single region of physical memory (64MB on UPMEM hardware). It would be advantageous to allow the DPU to access a mix of DRAM and SCM. The SCM is used for persistent storage of application data, while the DRAM is used for holding temporary results, such as uncompressed or decrypted blocks of data that is stored in the SCM portion. Ultimately, we would like to dynamically select the ratio of SCM to DRAM in each region. Even if that were not possible, there is still an advantage to having a static partitioning of memory types at a fixed ratio.

Tuning For Performance A system is only as good as its weakest point. Therefore, it is important to balance the throughput of the components to get maximum performance while minimizing cost. While we believe current PIM designs used with DRAM would be essentially the same on SCM, the system characteristics will likely be different. For example, SCM has higher access latency than DRAM for both reads and writes. Therefore, a PIM design may require modifications such as increasing the number of threads to have more memory accesses in-flight in order to compensate for higher latency. Alternatively, increasing the core density may also provide the same result. These details are dependent on many factors, including the cost of additional silicon area, power consumption, bus frequency and memory latency. Further study is required to draw any meaningful conclusions.

4 Feasibility Experiments

In this section, we identify some promising applications that PIM on SCM can accelerate. We show that these use cases can have high software overhead on SCM, and that software becomes the main performance bottleneck.

Format Parsing Decoding stored data formats (JSON, XML, ORC, etc.) into in-memory representations is an expensive task. The best known CPU implementation of SIMDJSON today can only operate at 2GB/s per-core [23]. With the restricted core counts of modern processors, we envision using the numerous PIM cores to perform some or all of the parsing. SIMDJSON identifies all structural bytes of a record in parallel before traversing it. Performing the identification on PIM hardware would alleviate CPUs of this bandwidth-intensive task. Another complimentary approach could be to use PIM to filter records before using a CPU to parse them, as proposed by Sparser [26].

To motivate this use case, we performed an experiment to compare the overhead of format conversion when the data is stored on Intel Optane NVRAM and on HDD. We parse 14GB of JSON *store_sales* from the TPC-DS benchmark suite using SIMDJSON's multi-line parsing benchmark running on a single CPU core. To ensure the data is taken from disk, we flush the block cache before the test. On NVRAM, the test takes 26s with 11s (42%) of parse time. On HDD it takes

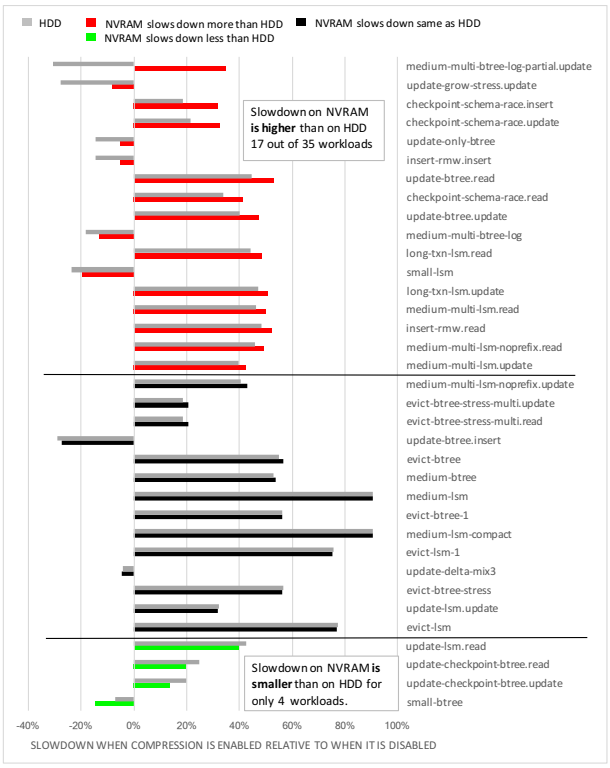

Figure 2: The X-axis shows the names of benchmarks and their respective operations in the format *benchmark.operation*, where the operation type can be *read*, *insert*, *modify* or *update*; the operation type is omitted when the benchmark performs only reads.

80s with 11s (13%) of parse time. SIMDJSON operates at 1.3 GB/s in these tests, so it can easily keep up with the HDD throughput at 200MB/s, whereas NVRAM at 1.6GB/s[1] it is able to sustain a higher throughput than the CPU can manage. By using multiple DPUs, we expect to parallelize the parsing, and see an increase in throughput.

Storage Compression Compression is used to improve storage density, but requires that data be decompressed before it can be used [24]. When this happens, CPU time that could be spent servicing other requests is spent on decompression.

MongoDB is the most popular NoSQL database [1] and its storage engine, *WiredTiger*, is responsible for managing the data storage on each database node. Like many other key-values stores, WiredTiger converts the data between on-disk format (for persistent data) and in-memory format (when the data is live), optionally including encryption or compression. These operations must be performed each time a block is transferred between persistent storage and main memory.

[1]Optane NVRAM here was used as a block device with a file system on top (no `dax` option) and data was read via system calls. If accessed via a `dax` file system and using `mmap`, the throughput would be even higher.

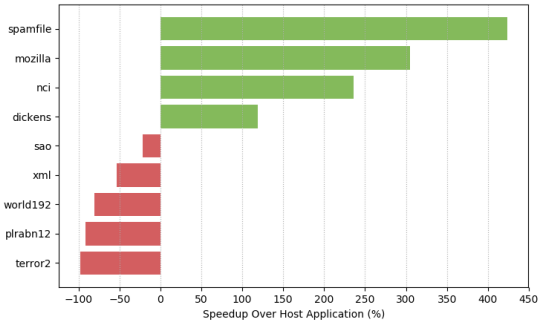

Figure 3: Snappy decompression benchmark results

Our hypothesis is that the relative overhead of compression is higher on NVRAM device than on HDD. In general terms, the compression overhead becomes higher (relative to the total execution time) as the speed of the storage device increases. To test the hypothesis, we used the WiredTiger performance suite `wtperf`. We ran each workload with compression on and with compression off, and compared the throughput. We repeated this experiment on the NVRAM block device[2] and on a conventional HDD. Our test system consists of an Intel Xeon 5218 CPU @ 2.3GHz, 256GB Intel Optane NVRAM (non-interleaved in app mode @2666 MHz), with a Toshiba 300 GB SAS HDD (@10K RPM).

Figure 2 shows the results. Compression overhead is higher on NVRAM than on HDD (by at least 3%) for 17 of the 35 benchmark/operations[3]. Only 4 benchmark/operations suffer higher compression overhead on HDD than on the NVRAM. Higher relative overhead is an indication that either the CPU cycles spent on compression or the increased cache miss rate due to increased cache pollution becomes a more significant factor in performance when I/O is fast. In both cases, offloading these tasks from the host CPU could reduce the overhead.

Snappy Decompression To test whether or not PIM hardware would be able to keep up with decompression offloaded from the CPU, we implemented a decompressor for the *Snappy* [15] algorithm. By using a set of test files that vary in size and compression ratio, we compiled a benchmark to measure the performance compared to a CPU (shown in Table 1). Figure 3 shows the speedup (slowdown) of our implementation over the same algorithm on the host CPU. For this experiment, we used a host with an AMD Ryzen CPU running at 2.2GHz, an Intel 660p 3D NAND SSD and 640 UPMEM DPUs. With the largest file size, we see a speedup

[2]Optane NVRAM was configured as a block device with the file system on top, so we could use WiredTiger without modifications.

[3]We did not run all of the workloads due to limited space on our NVRAM device and other configuration issues. We omitted from the Figure those where the differences between compressed and uncompressed configurations were not statistically significant

File	Size	# DPUs	# Tasklets	DPU cycles
terror2	100KB	1	4	7,274,144
plrabn12	500KB	2	8	8,822,944
world192	1MB	4	12	7,569,344
xml	5MB	15	12	7,391,632
sao	7MB	21	12	7,855,696
dickens	10MB	35	12	9,230,576
nci	30MB	64	18	7,235,184
mozilla	50MB	105	16	9,148,832
spamfile	84MB	172	16	9,800,320

Table 1: Snappy decompression benchmark parameters

of nearly 4.5x using only 172 DPUs. With the smallest file, we see a slowdown of 100% (i.e. 2x longer than the CPU). In both cases, the amount of available parallelism is the cause. The measurement does not take I/O time into account (i.e. reading the file from disk) in order to simulate the case in which the compressed data is stored in SCM, readily available to the PIM processors. While these are only initial results and our implementation has not yet been fully optimized, they are promising and indicate the need to extract sufficient parallelism.

4.1 Throughput

To expose the potential, we measured the maximum throughput between the DRAM and SRAM buffer of UPMEM PIM. Our test system is composed of nine single-rank DDR4-2400 UPMEM DIMMs, running at 267 MHz. That yields a total of 36GB of DRAM and 576 DPU cores. The host is one Intel Xeon Skylake SP (4110) socket.

The experiment copies the entire DRAM bank that is accessible by the DPU (64MB) in 2KB blocks from DRAM to the SRAM buffer using the DMA engine. We maximized parallelism by starting all DPUs simultaneously, and by using 16 threads concurrently in each DPU. The threads only copied data, and did not perform any additional processing.

We copied 36 GB in 0.14 seconds, for a total throughput of approximately 252 GB/sec. Compared to the maximum theoretical bandwidth of a single DDR4-2400 channel between the DRAM and CPU of 19 GB/sec [9], this is more than a 13x larger throughput. With the expected maximum frequency of the UPMEM DPU set to be 500 MHz in the future, the bandwidth advantage would increase up to 20×. This gives us the motivation to leverage this enormous bandwidth to accelerate our applications.

4.2 Other Use Cases

There are a few other use cases that we have conceived and investigated, but are not yet at the point that we can report results.

Encryption Databases often store their data in an encrypted form for security. When the data is to be read by the database, it must first be decrypted. Multiple client requests may require decrypting different blocks at the same time. Also, prefetching operations by the database may trigger decryption. Several data blocks can be decrypted in parallel, without intefering with the throughput of the main CPU which can continue to handle client requests.

Index Checks Checking if data is present in an index is a common task for databases, key/value stores, and analytics engines. Each index check is an independent operation, which means it can easily be parallelized. When a number of indexes need to be checked simultaneously, PIM can assist with the check, freeing up the CPU to deliver data to the client.

5 Conclusions

Implementing PIM on SCM is a promising direction in order to offload simple tasks and free up precious CPU cycles and memory bus cycles. Our motivating experiments have given some evidence supporting our belief. We plan to investigate how to best use this new architecture for software applications that are likely to gain the most benefit.

6 Discussion Topics

In keeping with the spirit of the workshop, we have composed the following list of items on which we hope to discuss with the other participants.

1. *Applicability* We would like to get feedback on the applicability of this architecture. We have come up with several use cases, but believe there are many more than we have not yet considered.

2. *Hardware Parameters* There are several hardware parameters that can be tweaked in a particular implementation that would affect the outcome of any software experiments we could run. How should we determine the correct mix of SCM and DRAM in each DPU rank? How to find the correct core density, given a particular PIM and memory technologies? This includes the items in our wishlist (section 3.2). Since many of these features are being discussed and defined in standards, we may have an opportunity to influence the community early enough to have a positive impact on technology we will be using for years to come.

3. *Benchmarking* One question that is not clear, is the most convincing way to benchmark this kind of architecture. Clearly, adding more CPU resources will increase overall system performance. However, we want to make a fair comparison to a system without these PIM processors, in order to show the cost vs. benefit.

References

[1] Db-engines ranking. DB-Engines, 2020. https://db-engines.com/en/ranking.

[2] Sandeep R Agrawal, Sam Idicula, Arun Raghavan, Evangelos Vlachos, Venkatraman Govindaraju, Venkatanathan Varadarajan, Cagri Balkesen, Georgios Giannikis, Charlie Roth, Nipun Agarwal, and et al. A many-core architecture for in-memory data processing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 245–258, New York, NY, USA, 2017. Association for Computing Machinery.

[3] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, June 2015.

[4] Dudley Allen Buck. Ferroelectrics for digital information storage and switching. In *Digital Computer Library*, 06 1952.

[5] Sangyeun Cho and Hyunjin Lee. Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, page 347–357, New York, NY, USA, 2009. Association for Computing Machinery.

[6] CCIX Consortium. Ccix cache coherency interface. 2019. https://www.ccixconsortium.com/.

[7] OpenCAPI consortium. Opencapi consortium. 2019. https://opencapi.org/.

[8] Intel Corporation. Intel® optane™ memory. 2019. https://www.intel.ca/content/www/ca/en/architecture-and-technology/optane-memory.html.

[9] Frank Denneman. Memory deep dive: Ddr4 memory. 2015. https://frankdenneman.nl/2015/02/25/memory-deep-dive-ddr4/.

[10] F. Devaux. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–24, Aug 2019.

[11] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. Computedram: In-memory compute using off-the-shelf drams. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 100–113, New York, NY, USA, 2019. Association for Computing Machinery.

[12] M. Gao and C. Kozyrakis. Hrl: Efficient and flexible reconfigurable logic for near-data processing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 126–137, March 2016.

[13] Bill Gervasi and Jonathan Hinkle. Overcoming system memory challenges with persistent memory and nvdimm-p. In *JEDEC Server Forum*. JEDEC, June 2017.

[14] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: the terasys massively parallel pim array. *Computer*, 28(4):23–31, April 1995.

[15] Google. Snappy - a fast compressor/decompressor. 2020. http://google.github.io/snappy/.

[16] T. H. Hetherington, M. Lubeznov, D. Shah, and T. M. Aamodt. Edge: Event-driven gpu execution. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 337–353, 2019.

[17] Bruce Jacob, David Wang, and Spencer Ng. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[18] R. Kaplan, L. Yavits, and R. Ginosar. Prins: Processing-in-storage acceleration of machine learning. *IEEE Transactions on Nanotechnology*, 17(5):889–896, 2018.

[19] Roman Kaplan, Leonid Yavits, and Ran Ginosar. From processing-in-memory to processing-in-storage. Supercomput. Front. Innov. : Int. J. 4, 2017.

[20] Alexey Khvalkovskiy, Dmytro Apalkov, S Watts, Roman Chepulskii, R Beach, Adrian Ong, X Tang, A Driskill-Smith, W Butler, P. Visscher, Daniel Lottis, E Chen, Vladimir Nikitin, and M Krounbi. Basic principles of stt-mram cell operation in memory arrays. *Journal of Physics D: Applied Physics*, 46:074001, 01 2013.

[21] Michael Krause and Mike Witkowski. Gen-z dram and persistent memory theory of operation. 2019. https://genzconsortium.org/.

[22] S. Lai. Current status of the phase change memory and its future. In *IEEE International Electron Devices Meeting 2003*, pages 10.1.1–10.1.4, Dec 2003.

[23] Geoff Langdale and Daniel Lemire. Parsing gigabytes of json per second. *The VLDB Journal*, 28(6):941–960, 2019.

[24] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas. Zbd: Using transparent compression at the block level to increase storage space efficiency. In *2010 International Workshop on Storage Network Architecture and Parallel I/Os*, pages 61–70, May 2010.

[25] M. Nakanishi, Y. Adachi, C. Matsui, Y. Sugiyama, and K. Takeuchi. Application-oriented wear-leveling optimization of 3d tsv-integrated storage class memory-based solid state drives. In *2018 International Conference on Electronics Packaging and iMAPS All Asia Conference (ICEP-IAAC)*, pages 27–32, 2018.

[26] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Filter before you parse: Faster analytics on raw data with sparser. *Proc. VLDB Endow.*, 11(11):1576–1589, July 2018.

[27] Patrick Siegl, Rainer Buchty, and Mladen Berekovic. Data-centric computing frontiers: A survey on processing-in-memory. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, page 295–308, New York, NY, USA, 2016. Association for Computing Machinery.

[28] H. S. Stone. A logic-in-memory computer. *IEEE Transactions on Computers*, C-19(1):73–78, Jan 1970.

[29] Hung-Wei Tseng and Dean Michael Tullsen. Software data-triggered threads. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 703–716, New York, NY, USA, 2012. Association for Computing Machinery.

[30] Rainer Waser and Masakazu Aono. Nanoionics-based resistive switching memories. In *Nature Materials*, 11 2007.

[31] W. . Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *The 16th Annual International Symposium on Computer Architecture*, pages 273–280, May 1989.

[32] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. Top-pim: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14, page 85–98, New York, NY, USA, 2014. Association for Computing Machinery.

[33] Vasileios Zois, Divya Gupta, Vassilis J. Tsotras, Walid A. Najjar, and Jean-Francois Roy. Massively parallel skyline computation for processing-in-memory architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, New York, NY, USA, 2018. Association for Computing Machinery.