A Case Study of Processing-in-Memory in off-the-Shelf Systems

Joel Nider, Craig Mustard, Andrada Zoltan, John Ramsden, Larry Liu, Jacob Grossbard, and Mohammad Dashti, *University of British Columbia;* Romaric Jodin, Alexandre Ghiti, and Jordi Chauzi, *UPMEM SAS;* Alexandra Fedorova, *University of British Columbia*

https://www.usenix.org/conference/atc21/presentation/nider

This paper is included in the Proceedings of the 2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

A Case Study of Processing-in-Memory in off-the-Shelf Systems

Joel Nider[1], Craig Mustard[1], Andrada Zoltan[1], John Ramsden[1], Larry Liu[1], Jacob Grossbard[1], Mohammad Dashti[1], Romaric Jodin[2], Alexandre Ghiti[2], Jordi Chauzi[2], and Alexandra (Sasha) Fedorova[1]

[1]University of British Columbia
[2]UPMEM SAS

Abstract

We evaluate a new *processing-in-memory* (PIM) architecture from UPMEM that was built and deployed in an off-the-shelf server. Systems designed to perform computing *in* or *near* memory have been proposed for decades to overcome the proverbial memory wall, yet most never made it past blueprints or simulations. When the hardware is actually built and *integrated* into a fully functioning system, it *must* address realistic constraints that may be overlooked in a simulation. Evaluating a real implementation can reveal valuable insights. Our experiments on five commonly used applications highlight the main strength of this architecture: computing capability and the internal memory bandwidth *scale with memory size*. This property helps some applications defy the von-Neumann bottleneck, while for others, architectural limitations stand in the way of reaching the hardware potential. Our analysis explains why.

1 Introduction

The memory wall has plagued computer systems for decades. Also known as the *von Neumann bottleneck*, it occurs in systems where the CPU is connected to the main memory via a limited channel, constraining the bandwidth and stretching the latency of data accesses. For decades, computer scientists have pursued the ideas of *in-memory* and *near-memory* computing, aiming to bring computation closer to data. Yet most of the proposed hardware never made it past simulation or proprietary prototypes, so some questions about this technology could not be answered. When we obtained early access to soon-to-be commercially available DRAM with general-purpose processing capabilities, we recognized a unique opportunity to better understand its limitations when integrated into existing systems. Most solutions proposed in the past required specialized hardware that was incompatible in some way with currently deployed systems [2, 3, 9, 14, 16]. The hardware we evaluate was designed specifically to be used as a drop-in replacement for conventional DRAM, which has imposed some limitations that are not present in many of the simulated architectures.

UPMEM's DRAM DIMMs include general-purpose processors, called *DRAM Processing Units* (DPU) [5]. Each 64MB slice of DRAM has a dedicated DPU, so *computing resources scale with the size of memory*. Playing to the strengths of this hardware, we ported five applications that require high memory bandwidth and whose computational needs increase with the size of the data. We observed that, indeed, application throughput scaled with data size, but scaling was not always the best that this hardware could achieve. The main reasons were the difficulty of accessing data located inside DPU-equipped DRAM from a host CPU without making a copy, the limited processing power of the DPUs, and the granularity at which the DPUs are controlled.

The main contribution of our work is understanding the limitations of PIM when it is integrated into off-the-shelf systems. We approach it via a case study of a particular implementation. We are not aware of any similar hardware that we could access, so a comparison to other architectures was outside the scope of this work. Although a comparison to other accelerators was not our goal either, we did compare with GPUs where it helped deepen our analysis.

2 Architecture

Many PIM architectures were proposed in the past [2, 3, 6–8, 10, 12, 21, 27, 28, 30, 32], ranging from logic gates embedded in each memory cell up to massive arrays of parallel processors that are located close to the memory. They all share the goals of overcoming the limitations of memory bandwidth but differ vastly in design. Nguyen et al. [22] introduce a classification system for various designs based on the distance of the compute units from the memory. We study specific hardware from the *COMPUTE-OUT-MEMORY-NEAR* (COM-N) model. This model includes compute logic that is located outside of the memory arrays but inside the memory package (same or different silicon die). This is sometimes called CIM (compute-in-memory) but is more widely known

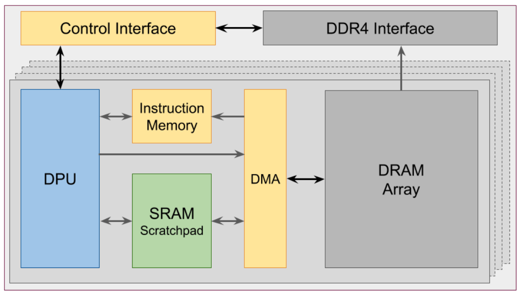

Figure 1: DPU Architecture

as PIM (processing-in-memory). After our study was concluded, Samsung announced the production of similar PIM hardware (called FIMDRAM) that also falls into the COM-N model [15]. While we could not perform an in-depth comparison, some similarities are apparent from the published design. These common design points are likely to appear in new designs as well and the lessons we learned from studying UPMEM-hardware will be generally applicable to this class of PIM.

Organization In the architecture we study, DRAM is organized into ranks (two ranks per DIMM), and each rank contains eight memory chips. Each DRAM chip includes eight DRAM Processing Units (DPUs). Each DPU is coupled with a 64MB *slice* of DRAM. There are 64 DPUs per rank and 128 DPUs per 8GB memory DIMM. Since the DPUs and DRAM are built on the same silicon die, introducing the DPUs comes at the cost of some DRAM capacity. With this design, the sum of DPUs and DRAM is limited by the silicon area on the die and the process density (number of transistors per unit area). Our experimental system has 36GB of DPU DRAM and 576 DPUs.

DPU Capabilities A DPU is a simple general-purpose processor. UPMEM's design only supports integer operations (no floating-point hardware). Each DPU has exclusive access to its slice of DRAM over a private bus. The more total DRAM there is, the more DPUs there are. In other words, we get an additional unit of computation, and an additional bus, with each additional unit of memory – *computing capacity and intra-DIMM bandwidth scale with the size of memory*. Figure 1 shows a schematic view of a single DPU and its periphery. Before a DPU can compute on the data, it must copy it from DRAM to a 64KB private SRAM buffer, called the *working memory*. The copy is performed with an explicit, blocking DMA instruction. The DMA engine can copy up to 2KB in a single operation but the copy time increases linearly with the size. To hide DMA latency, each DPU has 24 hardware threads, called *tasklets*, that can be scheduled for execution simultaneously. Because it is an interleaved multithreading (IMT) design [31], only one tasklet can advance at each cycle. When a tasklet is blocked on a DMA operation, other tasklets can still make progress.

DPU Speed Processing capabilities of DPUs are well below that of a common host CPU. DPUs have a simple, in-order design, and are clocked at anywhere from 266MHz (in our experimental system), to 500MHz (projected in commercial offerings). This was necessary to fit within the power envelope of commodity systems.

DPU communication In DRAM there are no communication channels between individual slices of memory dedicated to DPUs. Thus, DPUs cannot share data, other than by copying it via the host. Communication between DPUs on different chips would require additional pins in the package which would change the form factor from standard DRAM. Creating a communication bus between DPUs on the same chip is theoretically possible, but severely restricted by the number of available metal layers. Therefore, we cannot expect any PIM designs in the COM-N model to have any DPU-to-DPU communication until these problems can be solved. This constraint implies that programs running on DPUs must shard data across DPUs ahead of the execution and use algorithms that don't require data sharing.

Interleaving At the chip level, interleaving dictates the ordering of bytes as they are written into DRAM. In the DIMMs in our test system, each DRAM chip connects to the DDR4 bus by an 8-bit interface. 8 chips together form the 64-bit wide bus. When a 64-byte cache line is committed to memory, the data is interleaved across DRAM chips at byte granularity, the first byte going into *chip 0*, the second byte into *chip 1*, etc. So when the write is done, the first chip receives bytes 0, 8, 16, ..., 56. Interleaving at this level is used to hide read and write latency by activating multiple chips simultaneously. Interleaving is transparent to the host since the data is reordered appropriately in both directions (i.e., during reads and writes). But a DPU, which can only access data in its own slice, only sees what is written to that particular chip (i.e., every *n*th byte of the data). This property makes it difficult to access the data by the host CPU and the DPU in the same location. Unless a program running on the DPU can be modified to operate on non-contiguous data, we must make a separate copy, *transposing* it in the process, so that each DPU receives a contiguous portion of the data.

Transposition UPMEM's SDK counteracts the interleaving by transposing the data. While the cost of the transposition is small (the current SDK provides an efficient implementation that uses SIMD extensions), the cost of data copy, which requires going over the DRAM bus, is not. For applications that can place their dataset into the DPU memory and compute on it using only DPUs, the one-time cost of the copy can be negligible, but for applications where the data is short-lived or both the DPU and host CPU must access it, frequent copying will stress the memory channel – the very bottleneck this architecture aims to avoid.

Control granularity Accessing multiple chips in each bus cycle also means it is natural to schedule DPUs across those chips as a group. With the current SDK implementation, a full rank must be scheduled as a single unit. We can only send a command to an entire rank of DPUs and cannot launch a DPU or read its memory while another DPU in the same rank is executing. Similarly, the host cannot access any memory in a rank if any DPU in that rank is executing. In practice, this results in treating DPU-equipped memory as an accelerator, rather than part of main RAM. The host typically keeps programs and data in the "normal" DRAM (i.e., *host DRAM*), copies data for processing into the *DPU DRAM*, and copies back the results. Finer-grained execution groups will likely improve the performance of algorithms that "stream" data such as *grep*.

3 Evaluation

3.1 Workloads

Considering DPU architecture properties we chose programs that need more computational resources with the increasing data size, that could be easily parallelized and where data sharing is minimal.

The SDK for the DPU architecture includes a C compiler and libraries with functions to load and launch DPUs, copy and transpose the data, etc. While we could write the code in a familiar language, the main challenges had to do with memory management (as the DPU needs to copy data from DRAM into the working memory prior to processing it), and with efficiently controlling ranks of DPUs.

Snappy Snappy is a light-weight compression algorithm designed to process data faster than other algorithms, such as gzip, but at the cost of a lower compression ratio [11]. Snappy divides the original file into blocks of a fixed size (typically 64KB) and compresses each block individually. The original implementation is designed for sequential (i.e., single-threaded) processing. The file format concatenates compressed blocks head to tail, without any separation marker. The result is that to decompress block n, the program must first decompress blocks 0 .. $n-1$. To enable parallelism, we changed the compressed file format by prepending each block with its compressed size. This small change enables the host program to quickly locate the start of each compressed block without having to decompress the previous blocks. We also modified the header by adding the decompressed block size. That enables us to experiment with different block sizes. Snappy is bandwidth-intensive because of the small amount of processing, especially during decompression. The majority of the time is spent copying literal strings or previously decoded strings.

Hyperdimensional computing Hyperdimensional Computing (HDC) [13] is used in artificial intelligence to model the behaviour of large numbers of neurons. It relies on vectors with high dimensionality (at least 10,000 dimensions), called *hypervectors*, to classify data and find patterns. Our HDC application performs classification of raw electromyography (EMG) signals into specific hand gestures [25]. We built upon a prior reference implementation [20]. The classification works by computing the Hamming distance between a previously encoded input hypervector and previously trained vectors. Our implementation distributes the raw signal data among DPUs, which then encode it into hypervectors and perform the classification.

AES Encryption The Advanced Encryption Standard (AES) is a symmetric-key algorithm designed to be highly secure and run efficiently on a variety of hardware. Our analysis focuses on AES-128, using the implementation by Rijmen, Bosselaers, and Barreto found in OpenSSL [26]. Encryption is an ideal problem; the data can be divided into small individual blocks, the algorithm can operate in parallel, and the data access pattern makes it simple to optimize DMA transfers. AES can be operated in a variety of modes. We use ECB (electronic code book) mode which ensures each block can be processed individually without dependencies on the other blocks.

JSON filtering JSON is a flexible and human-readable format for storing and exchanging data. Due to its ASCII representation, parsing is notoriously slow [17, 19, 29]. When analytics systems filter records in JSON, they parse each one and check the filter condition. Parsing is performed for *all* records, even for those not meeting the filter criteria. Instead, Sparser [24] showed that it is better to filter records *before* parsing them by running string comparisons over raw (unparsed) JSON. Raw filtering is highly parallel and memory-intensive [9], and hence promising for our study. We modified Sparser [23] to offload raw filtering to DPUs, while the host performs the rest of the parsing.

Grep Grep [1] is a command-line utility for searching plain-text for lines that match a regular expression. We implement a subset of grep, which searches only for exact text matches. Our design uses the DPUs in a work pool by preparing files in small batches, and starting the search as soon as enough DPUs are ready to perform the work.

To address the challenge of DPUs needing to be controlled in large groups we must balance the work across all DPUs in the rank. By balancing the work, we minimize the difference in running time between the fastest DPU and slowest DPU, which minimizes the idle time in the rank while waiting for the slower DPUs. Preparing the work on the host consumes a significant portion of the time so it is more important to prepare the files quickly rather than efficiently packing them into DPUs. To fill the rank as evenly as possible within the time constraints, files are assigned to DPUs in a round-robin fashion in the order they appear in the input. We do not spend the time to sort the files by size, and even limit the number of files that can be processed by a single DPU to save time during preparation. A maximum of 256 files was determined empirically to have the best results.

Our baseline for performance comparison is a single host

CPU[1] since our focus is to understand when the DPU architecture overcomes the von Neumann bottleneck and when it does not.

3.2 Memory bandwidth scaling

In the DPU architecture, each additional slice of DRAM gets its own DPU along with the working memory and the DMA channel. Herein lies its main strength: ***scaling computing capacity and intra-DIMM bandwidth with the size of memory***. To demonstrate, we ran a simple program where all DPUs iterate over the data in DRAM, copying every byte from their respective DRAM slices into the working memory. As shown in Fig. 2, the data processing bandwidth increases with the size of the data, reaching roughly 200GB/s for 36GB worth of DRAM (approximately 350MB/s per DPU). For a system with 128GB of DRAM and DPUs clocked at 500MHz (which is the target for commercial release) the aggregate bandwidth attained by all DPUs would reach 2TB/s.

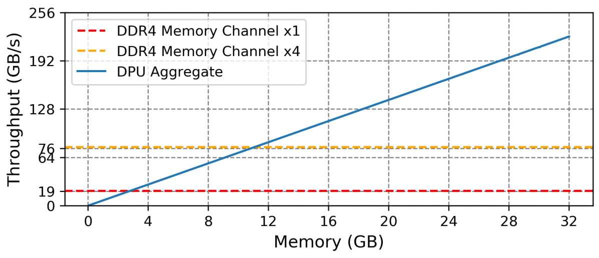

Figure 2: Memory bandwidth scales with memory capacity.

For applications whose compute and bandwidth needs grow with the size of the data, scaling of compute and bandwidth resources with memory has a direct and positive impact on performance. E.g., Fig. 3 shows the throughput of *Snappy compression* as the file size (and the number of DPUs sharing the work) increase. We use the optimal number of DPUs for each file size – i.e., using more DPUs yields no additional speedup. The input data resides in host DRAM; the experiment copies the data to DPU DRAM, allocates and launches the DPUs to run compression, and copies the compressed data back to host DRAM. As the file size grows *so does the throughput* – a direct result of compute capacity and internal bandwidth scaling with additional memory.

Breaking down the runtime, we observe that the execution time on DPUs (*Run* in Fig. 4) remains roughly the same even as we increase the file size, because more DPUs are doing the work. Again, this is the effect of scaling compute resources with data size. The DPU execution time does increase for 512MB and 1GB file sizes, because as we reach the limits of our experimental hardware, each DPU gets more data to

[1]Our host machine is a commodity server with an Intel(R)Xeon(R) Silver 4110 CPU @ 2.10GHz with 64GB of conventional DDR4 DRAM.

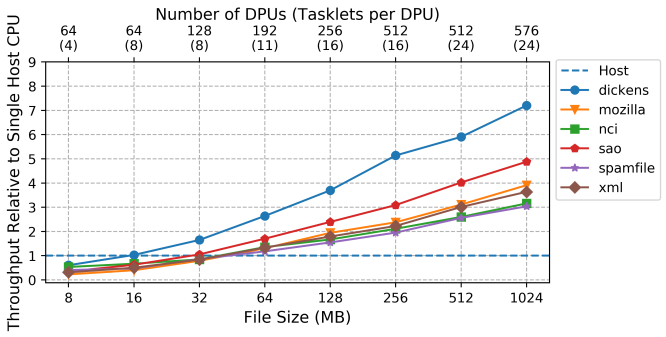

Figure 3: Snappy compression throughput

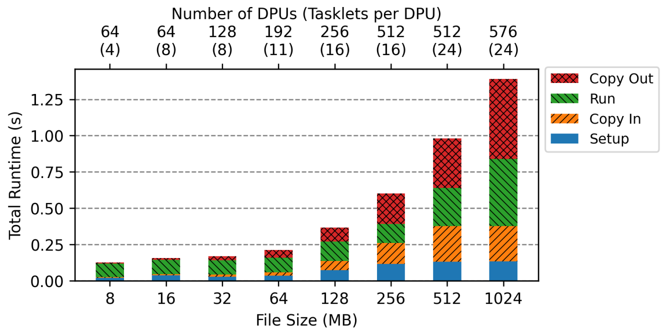

Figure 4: Runtime breakdown for compression

process. The non-scalable components of runtime in Fig. 4, *Copy In*, *Copy Out* and *Setup*, are discussed in §3.3.

A final experiment illustrating the strength of scaling involves a comparison with the GPU[2], an accelerator that shares similarities with DPUs (massive parallelism, high memory bandwidth), but does not have the property of scaling computational resources with memory. Fig. 5 shows the throughput relative to a host CPU (same as in the DPU experiments) of *snappy compression* as the file size grows. Contrary to the DPU architecture, the GPU scaling is flat and the throughput does not surpass that of the host CPU by more than 5×. This coincides with the internal memory bandwidth of the GPU, which is approximately 5× higher than that of the CPU.

Where each DPU has a dedicated working memory and a DMA channel, all GPU cores within an SM (streaming multiprocessor) share an L1 cache. This data-intensive workload with a random access pattern caused thrashing in the small 64KB L1 cache. Memory fetching became the bottleneck and as a result, we obtained the best performance by launching kernels consisting of a single thread per compute block (and hence one thread per SM) to reduce contention for the cache. The input file fit in the GPU memory, so there were no fetches

[2]We used an RTX 2070 GPU with 8GB of GDDR6 and 448GB/s global memory bandwidth. There are 36 SMs, with 64 CUDA cores per SM, clocked at 1.4GHz, There is a 64KB L1 cache per SM, and 4MB L2 cache shared by all SMs.

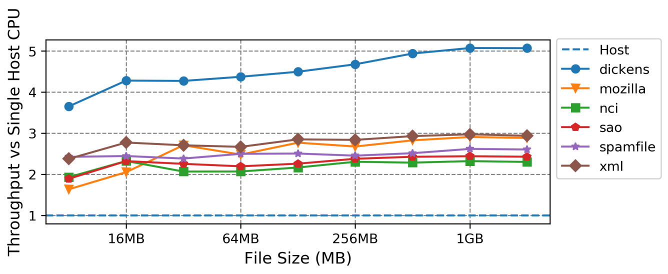

Figure 5: Snappy compression on GPUs

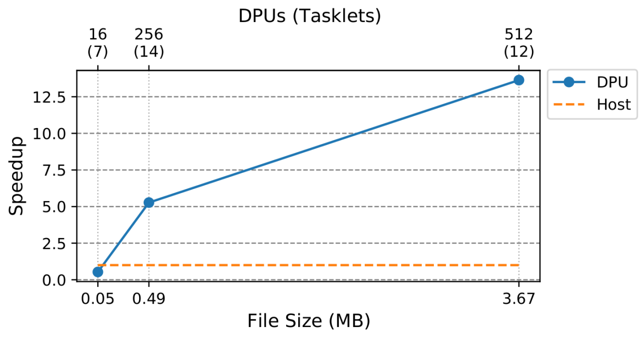

Figure 6: HDC speedup over the host as file size increases

over the PCIe bus once processing started. In other words, the GPU fell victim to a variant of the von Neumann bottleneck. In workloads where memory bandwidth is not the main limitation, DPUs will not reach the throughput of a GPU due to their less powerful execution engine – a limitation we discuss in §3.4.

3.3 Data copy

As we discussed in §2, the need for de-interleaving (transposing) data for DPUs means that it is difficult to avoid making copies when data needs to be accessed by both DPUs and host CPUs. For applications where DPUs can be used exclusively, the data can be copied to DPU memory just once and reside there for the rest of the runtime, making copy overhead negligible. In programs where data is ephemeral or must be accessed by both the host CPU and DPUs, frequent copying will be subject to memory channel limitations – the very von Neumann bottleneck PIM aims to address.

The compression runtime breakdown presented in the previous section (Fig. 4) illustrates this point. The DPU execution time remains roughly constant as data size increases, because each DPU has a dedicated DMA channel to its DRAM slice. In contrast, the time to copy input data to DPU memory and copy the output back increases with the file size and is limited by the DRAM channel.

Setup overhead in compression does not scale, because dividing up the work requires sequential parsing of the input file. This is a property of the workload and implementation, and not fundamental to the architecture. Copy overhead, on the other hand, is fundamental to the system and is the reason why the compression throughput in Fig 3 grows sublinearly with the increasing number of DPUs.

HDC is less affected by the copy-in overhead than Snappy and has negligible copy-out cost due to the small dataset. Overall, copying takes about 10% of the total runtime, and thus HDC enjoys better scaling, as shown in Figure 6. HDC and other AI applications, where DPUs can repeatedly perform inference on long-lived data that remains in DPU memory are therefore well-positioned to circumvent the memory channel bottleneck.

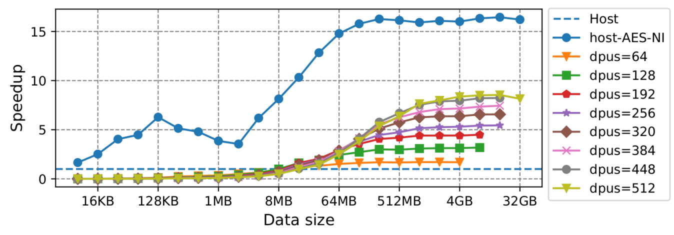

Figure 7: Encryption throughput relative to a single host CPU.

3.4 DPU Speed

Encryption provides an excellent illustration of a scenario when DPU processing power becomes a limitation. Figure 7 shows encryption throughput (bytes per second) relative to a single host CPU as the amount of data increases. We show a separate curve for each number of DPUs, to emphasize that processing power matters. Both *Snappy* and *HDC* required fewer DPUs for smaller datasets and more DPUs for larger data sets, but compute-hungry encryption relishes the highest number of DPUs we can muster, no matter the data size. Further, the far superior performance on the host with AES-NI [4] acceleration confirms that general-purpose DPUs cannot compete with specialized instruction sets.

Another example where DPU processing power was less impressive compared to other available accelerators was *snappy decompression* (figures omitted for brevity). On the DPU, this workload performed very similarly to compression. On the GPU, in contrast with compression, L1 cache was not the bottleneck: decompression works with smaller input block sizes and is largely write-intensive. As a result, the GPU showed similar scaling trends as on the DPU and outperformed the host CPU by up to 24×, while DPUs' advantage was at most 8×.

3.5 Communication and control granularity

Compliance with the DDR interface and software limitations in the current SDK makes it inefficient to control DPUs one at a time: as long as a single DPU in the rank is busy, all

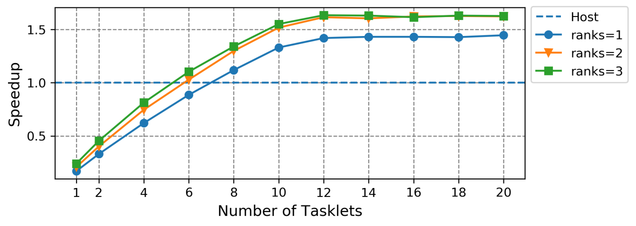

Figure 8: Speedup of grep over the host with varying ranks and tasklets.

others are inaccessible. This means that the program must decide how it will divide the data between DPUs in a rank and launch them all at once. While this is a trivial decision for embarrassingly parallel workloads, other workloads might under-utilize the hardware as a result. The fact that DPUs cannot communicate and steal work from one another makes matters worse. Our experience with *grep* demonstrates these challenges.

Figure 8 shows the throughput relative to a single host CPU of *grep* executed on the 875MB Linux source tree for the varying number of ranks. The throughput gain relative to the host is modest, and there is no advantage from using more than two ranks (128 DPUs). The reason is that it is very difficult to distribute the work evenly between DPUs ahead of the launch. Input source files vary in size and even organizing them into equally-sized buckets per DPU does not guarantee that the processing time on all DPUs will remain similar. As a result, *grep* used the hardware inefficiently: the ratio of execution times on the fastest and slowest DPUs was 116×, indicating the presence of stragglers that held up the entire rank. In contrast, *JSON filtering*, a workload very similar to grep but where dividing the work evenly was easy, enjoyed the fastest/slowest execution ratio of 1 (no stragglers) and similar scaling properties as the rest of the workloads (figure omitted for brevity).

3.6 System cost

We conclude with the analysis of system cost, showing that for applications with high bandwidth requirements, a system that uses DPU memory is cheaper than the one without it.

With a memory-bound workload, increasing the number of processors without increasing the memory bandwidth does not improve performance by a significant amount [3, 18, 33]; we need a CPU with a high number of memory channels that can supply the cores with data and avoid starvation. CPUs are designed with a wide range of cores but do not include additional memory channels for a higher number of cores. Therefore, we must use more sockets and faster memory to scale up the memory bandwidth in a CPU-only system. Table 1 shows an estimated price comparison of system configurations with the maximum possible memory bandwidth with off-the-shelf Intel systems (1.6TB/s) assuming Snappy compression as the representative workload. We use the nominal price of $60 for an 8GB DIMM and $300 for an 8GB PIM module [5]. The costs shown include CPU, memory and PIM (where applicable). We assume DDR4-2400 (19.2 GB/s) with a Xeon 4110 ($500) for all configurations except the last, which uses DDR-3200 (25.6 GB/s – 33% more bandwidth) and a Xeon 8380 ($8100). With our conservative estimations, a CPU-only system is less expensive for applications below 500GB/s. However when scaling up, a CPU-only configuration is approximately 3.8× more expensive to attain the same Snappy compression throughput.

DRAM	PIM	Snappy throughput	Cost	Cost per MB/s
Low bandwidth requirements				
48GB	-	512MB/s	**$860**	$1.68
48GB	32GB	512MB/s	$2160	$4.21
High bandwidth requirements				
48GB	448GB	7168MB/s	**$17660**	$2.46
448GB	-	7263MB/s	$68640	$9.45

Table 1: System cost per MB/s for Snappy compression.

4 Conclusion

Memory-bound applications are limited by the conventional memory hierarchy, and adding more CPUs does not improve performance substantially. We have shown several real-world applications that can be accelerated by PIM by scaling memory bandwidth with compute resources. PIM shares many attributes (such as massive parallelism) with other accelerators but the distribution of processing units inside the memory gives it a unique advantage in certain cases. The architecture we evaluated has some limitations that prevent exploiting the full capabilities of the hardware, which remain as challenges for future designs. Costly data transfers which must be performed in order to share data between the host CPU and PIM memory adds significant overhead to processing time but can be mitigated by reusing data in-place. Despite the limitations, we find that PIM can be effective when applied to the correct class of problems by scaling the processing power with the size of the dataset.

Acknowledgements

Thank you to our shepherd, Heiner Litz, for his helpful comments and suggestions to improve this paper. A special thank you to Vincent Palatin for his patience and support explaining the details of this new hardware.

Availability

All source code for the projects described in the paper can be found at `https://github.com/UBC-ECE-Sasha`. The UPMEM SDK is publicly available at `https://www.upmem.com/developer/`.

References

[1] Tony Abou-Assaleh and Wei Ai. Survey of global regular expression print (grep) tools. In *Technical Report*, 2004.

[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] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 105–117, New York, NY, USA, 2015. Association for Computing Machinery.

[4] Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. Breakthrough AES performance with Intel® AES new instructions. In *Intel Whitepaper*. Intel, 2010.

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

[6] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. The architecture of the diva processing-in-memory chip. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, page 14–25, New York, NY, USA, 2002. Association for Computing Machinery.

[7] 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.

[8] 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.

[9] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development*, 63(6):3:1–3:19, 2019.

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

[11] Google. Snappy - a fast compressor/decompressor. *Goole Inc.*, 2020.

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

[13] Geethan Karunaratne, Manuel Le Gallo, Giovanni Cherubini, Luca Benini, Abbas Rahimi, and Abu Sebastian. In-memory hyperdimensional computing. In *Nature Electronics*, 2020.

[14] Liu Ke, Udit Gupta, Carole-Jean Wu, Benjamin Youngjae Cho, Mark Hempstead, Brandon Reagen, Xuan Zhang, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, and Xiaodong Wang. Recnmp: Accelerating personalized recommendation with near-memory processing, 2019.

[15] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, Youngmin Cho, Jin Guk Kim, Jongyoon Choi, Hyun-Sung Shin, Jin Kim, BengSeng Phuah, HyoungMin Kim, Myeong Jun Song, Ahn Choi, Daeho Kim, SooYoung Kim, Eun-Bong Kim, David Wang, Shinhaeng Kang, Yuhwan Ro, Seungwoo Seo, JoonHo Song, Jaeyoun Youn, Kyomin Sohn, and Nam Sung Kim. 25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2tflops programmable computing unit using bank-level parallelism, for machine learning applications. In *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 64, pages 350–352, 2021.

[16] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 740–753, New York, NY, USA, 2019. Association for Computing Machinery.

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

[18] Dominique Lavenier, Charles Deltel, David Furodet, and Jean-François Roy. BLAST on UPMEM. Research Report RR-8878, INRIA Rennes - Bretagne Atlantique, March 2016.

[19] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison: A fast json parser for data analytics. *Proc. VLDB Endow.*, 10(10):1118–1129, June 2017.

[20] F. Montagna, A. Rahimi, S. Benatti, D. Rossi, and L. Benini. Pulp-hd: Accelerating brain-inspired high-dimensional computing on a parallel ultra-low power platform. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[21] Amir Morad, Leonid Yavits, Shahar Kvatinsky, and Ran Ginosar. Resistive gp-simd processing-in-memory. *ACM Trans. Archit. Code Optim.*, 12(4), January 2016.

[22] Hoang Anh Du Nguyen, Jintao Yu, Muath Abu Lebdeh, Mottaqiallah Taouil, Said Hamdioui, and Francky Catthoor. A classification of memory-centric computing. *J. Emerg. Technol. Comput. Syst.*, 16(2), January 2020.

[23] Shoumik Palkar, Firas Abuzaid, and Justin Azoff. Sparser Source Code.

[24] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Filter before you parse: Faster analytics on raw data with sparser. *Proceedings of the VLDB Endowment*, 11(11), 2018.

[25] A. Rahimi, S. Benatti, P. Kanerva, L. Benini, and J. M. Rabaey. Hyperdimensional biosignal processing: A case study for emg-based hand gesture recognition. In *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–8, 2016.

[26] Vincent Rijmen, Antoon Bosselaers, and Paulo Barreto. Optimised ansi c code for the rijndael cipher (now aes), 2000. `https://github.com/openssl/openssl/blob/master/crypto/aes/aes_core.c`.

[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] Tencent. RapidJSON. `https://rapidjson.org/`.

[30] K. Wang, K. Angstadt, C. Bo, N. Brunelle, E. Sadredini, T. Tracy, J. Wadden, M. Stan, and K. Skadron. An overview of micron's automata processor. In *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–3, Oct 2016.

[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] Keira Zhou, Jack Wadden, Jeffrey J. Fox, Ke Wang, Donald E. Brown, and Kevin Skadron. Regular expression acceleration on the micron automata processor: Brill tagging as a case study. In *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data)*, BIG DATA '15, page 355–360, USA, 2015. IEEE Computer Society.