Accelerating Large Table Scan using Processing-In-Memory Technology

Alexander Baumstark[1] Muhammad Attahir Jibril[2] Kai-Uwe Sattler[3]

Abstract: Today's systems are capable of storing large amounts of data in main memory. Particularly, In-memory DBMSs benefit from this development. However, the processing of data from the main memory necessarily has to run via the CPU. This creates a bottleneck, which affects the possible performance of the DBMS. Processing-In-Memory (PIM) is a paradigm to overcome this problem, which was not available in commercial systems for a long time. However, with the availability of UPMEM, a commercial product is finally available that provides PIM technology in hardware. In this work, we focus on the acceleration of the table scan, a fundamental and memory-bound operation. We show and investigate an approach that can be used to optimize this operation by using PIM. We evaluate the PIM scan in terms of parallelism and execution time in benchmarks with different table sizes and compare it to a traditional CPU-based table scan. The result is a PIM table scan that outperforms the CPU-based scan significantly.

Keywords: UPMEM; Processing-In-Memory; In-Memory Database

1 Introduction

In-memory databases aim at low latency and high throughput for queries and updates in order to support real-time data processing. By keeping (most of) the data in main memory, workloads on such databases are typically memory-bound, and accessing main memory becomes more and more a bottleneck – a phenomenon that is known as memory wall [WM95]. However, novel and emerging memory technologies open up new opportunities such as offloading computation to memory. One example is Processing-in-Memory (PIM), a rather new concept where (simpler) operations can be executed directly in memory (on the same die) without moving the data from DRAM. The basic idea of this approach is to equip memory chips with additional processing units. Data can be processed directly on the memory chips without involving the system's CPU. PIM offers great potential: CPU load could be reduced, and memory bandwidth could be increased by reducing the amount of data to be transferred to the CPU.

Though, many PIM architectures have been proposed in the past (see [Ng20] for a classification), the only publicly available commercial product is offered by the UPMEM company. In addition, Samsung has also announced a product, but it is not available on the market yet. The

[1] TU Ilmenau, DBIS, Helmholtzplatz 5, 98693 Ilmenau, alexander.baumstark@tu-ilmenau.de
[2] TU Ilmenau, DBIS, Helmholtzplatz 5, 98693 Ilmenau, muhammad-attahir.jibril@tu-ilmenau.de
[3] TU Ilmenau, DBIS, Helmholtzplatz 5, 98693 Ilmenau, kus@tu-ilmenau.de

doi:10.18420/BTW2023-51

UPMEM technology has only recently become available (first presented at HotChips 2019), therefore, only a few experimental studies of UPMEM have been published. In [Ni21], the authors evaluate UPMEM PIM using a few use cases such as data compression, encryption, JSON processing, and text search. [Gó22] presents PrIM – the Processing-In-Memory benchmarks – a benchmark suite from different application domains as well as several key observations and programming recommendations. Though PrIM contains also a database selection operator, it is not integrated into a database engine. Both papers discuss also the technical details of UPMEM.

Based on these works, we investigate in this paper the potential of offloading data management processing to memory using PIM. Based on the observation that the performance of typical data management operations often depends on memory bandwidth or latency, we try to answer the question: *Can we accelerate in-memory operations in a database using PIM?* For this purpose, we use a graph database engine Poseidon[4], which supports in addition to a persistent memory storage engine [Ji21] also an in-memory mode. However, because we focus in this paper on scan operations, the findings of our experiments are not limited to graph databases. Still, they can be generalized to scans on relational and other non-relational databases.

2 Related Work

PIM is a well-known technique to overcome the CPU-memory bottleneck for several decades. There have been a number of concepts and approaches to provide PIM on hardware since the 1990s [Pa97, Pa97, Dr02]. The high cost and lack of industrial support for this concept prevented the production and sale of real PIM hardware. Still, research was conducted based on prototypes. The PIM technology follows a similar approach to GPU processing. The design space of GPU-accelerated architectures transferred to PIM was investigated in the work of [Zh14]. Further, with LazyPIM, the authors of [Bo17] published a mechanism for reducing data exchange between CPU and PIM cores by means of caching. With the company around UPMEM, hardware providing real PIM-enabled DRAM DIMMs was published [UP22]. There are already a number of works concerning this architecture investigating its characteristics and applicability. Gomez-Luna et. al investigates the architecture for its limitations and performance as well as energy consumption [Gó22]. The result of the work shows that the UPMEM system achieves suitable performance as long as the individual components (DPUs) do not require communication (DPU-to-DPU). There is also available work concerning the applicability of PIM hardware on real use cases. [Gu] investigates the potential of PIM hardware for the acceleration of ML training. The results show that ML training using PIM hardware can improve the training process compared with GPU-based ML training. [Gi22] investigates the improvement of sparse matrix-vector multiplication using real PIM hardware. [Ka22] provided an efficient index data structure that leverages PIM.

[4] https://dbgit.prakinf.tu-ilmenau.de/code/poseidon_core/-/tree/upmem

However, due to the short availability of real PIM hardware at the time of this work, there is, to our knowledge, no DBMS that directly integrates PIM.

3 PIM Technology

The first publicly available real-world PIM technology is provided by the UPMEM company [UP22]. Because our work is based on this technology, in the following we give a brief overview of this architecture and the programming model.

3.1 UPMEM Architecture

The core of the UPMEM architecture is the UPMEM DIMMs, which are based on regular DDR4-2400 DIMM modules but equipped with additional PIM chips. A structural overview is given in Fig. 1. The UPMEM DIMMs are organized into ranks. A UPMEM DIMM consists of up to two ranks and each rank consists of up to 8 PIM-enabled chips. A PIM chip usually consists of 8 DRAM Processing Units (DPUs). Each DPU has exclusive access to 64 MB Main RAM (MRAM), 24 KB Instruction RAM (IRAM), and 64 KB Working RAM (WRAM) for processing. As DPUs have only access to their own MRAM there is no direct communication possible between different DPUs. Further, a DPU consists of a general-purpose 32-bit RISC core with a maximum achievable frequency of 400 MHz, which can execute a special instruction set in a multithreaded in-order pipeline. For multithreading, there are 24 hardware threads available. The context is switched on every cycle between the threads, which hides the memory latency [La16]. All threads share the same memory on the DPU which requires synchronization to guarantee consistency. This architecture allows the parallel execution of a program on different pieces of data directly on DRAM.

3.2 Programming Model

For the utilization of the 24 hardware threads of a DPU, up to 24 tasklets can be used. This follows the Single Program Multiple Data programming model. All threads are executed with the same code but on different pieces of data. The number of used tasklets must be defined by the programmer at compile-time. As the MRAM and WRAM are shared among all tasklets on a DPU, the model provides synchronization primitives like mutexes, semaphores, barriers, and handshakes. Critical sections in the execution of a DPU program can be protected by mutexes with `mutex_lock` and `mutex_unlock` methods. Their effect is the same as the mutexes in usual systems. The critical section is then only accessible by one tasklet at a time. The purpose of the barriers is to control the execution flow of all tasklets. This can be done by using the `barrier_wait` method. The tasklets of the DPU wait at this

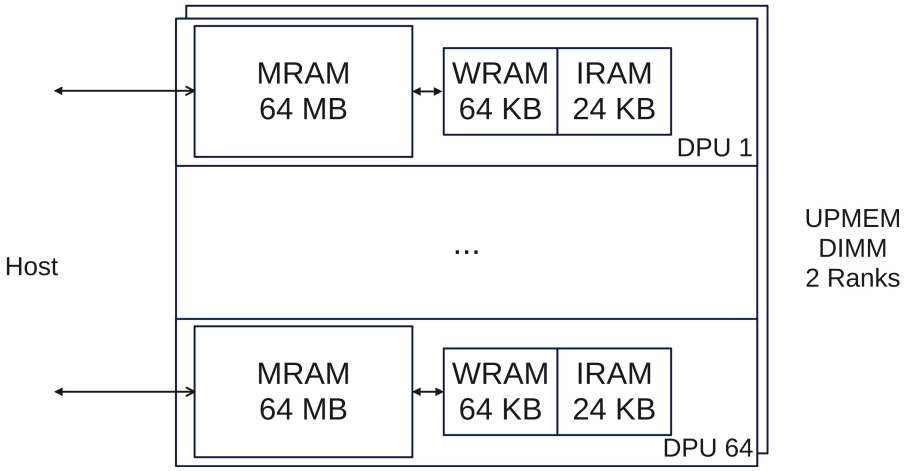

Fig. 1: Overview of a UPMEM DIMM with 2 Ranks of 64 DPUs.

point until all other tasklets have reached the barrier. After this, the execution continues. The handshake primitives are used for direct synchronization between the tasklets and the semaphores primitives for counters, similar to the counters known in operating systems. Using these primitives enables the effective utilization of multithreading provided by the DPUs.

The execution and control of the DPU program are handled by the host application. The host application allocates the set of desired DPUs and selects the appropriate DPU program. It is possible to allocate a specific rank or a specific number of DPUs. Further, the host application manages the execution of the DPU program and the data transfer to and from MRAM. The actual execution and data transfer can be handled synchronously and asynchronously by the host application. When executing the DPU program launch or the data transfer synchronously, the host application waits for the complete execution of the launch or data transfer. When transferring data, it is often desired to prepare the next batch for data transfer, while transferring the old batch. For this purpose, the UPMEM host library provides the possibility to execute the data transfer and DPU launch asynchronously. The asynchronous execution executes the instructions in the background using another thread and gives the control back to the host application. It allows the host application to proceed with the next batch for data transfer, or launch the same of another program on another set of DPUs.

The workflow of a host program running a DPU program with UPMEM technology can be summarized in the following steps:

1. Environment allocation (DPU, Ranks, DPU Program),
2. Buffer population from the host's main memory to MRAM of DPUs.

3. Execution of the DPU Program.
4. Retrieving of the processed results from the MRAM of the DPUs to the host's main memory.

The DPU program can be executed several times. The data is retained in the MRAM of the DPUs and does not have to be reinitialized. This is useful for tasks where a solution has to be calculated in several iterations. The DPU programs are written in the programming language C and compiled by a special compiler, which is based on LLVM and Clang.

3.3 Memory Management

As already mentioned, various memory types are available to a DPU. These differ in size and also in connection to the DPU. The largest memory available to a DPU is the MRAM. It has a capacity of 64 MB and has the purpose of exchanging data with the host. The host system can copy data from its main memory to the MRAM and also transfer data from the MRAM to the main memory of the host.

The WRAM of a DPU is a working memory in which a DPU stores the stack and global variables. Access to this memory is restricted to the DPU itself. Direct access from the host is not possible. Further, the DPU can access the WRAM only through 8-64 bit DMA instructions. The UPMEM runtime library provides for the transfer between MRAM and WRAM the methods `mram_read` for WRAM-MRAM and `mram_write` for MRAM-WRAM transfer. Each DMA instruction can copy up to 2 KB of data.

Communication with the host is done through data transfers between the main memory of the host and the MRAM of the DPU. The UPMEM runtime library provides different instructions for this purpose like `dpu_copy_to/dpu_copy_from` for copying a buffer from and to MRAM of specific DPUs. For parallel data transfer, the library provides the method `dpu_prepare_xfer` which assigns a buffer to a specific MRAM of a DPU. The actual data transfer is then performed in parallel using the `dpu_push_xfer` method but requires the same buffer size for all DPUs.

4 The Poseidon Graph Database

The present work is mainly developed for the graph database Poseidon. Although Poseidon was originally optimized for the characteristics of persistent memory, these characteristics can also be transferred to the exploitation of in-memory processing in DRAM. For this purpose, Poseidon already provides the necessary optimized data structures. In the following, the general architecture of Poseidon is described.

4.1 Data Model and Storage

The data layout of the Poseidon Graph Database is based on the labeled property graph model wherein labels and property values can be assigned to nodes and relationships. A complete graph in the Poseidon Graph Database consists of different tables in which the nodes, relationships, and respective property entries are stored. The tables are directly stored on DRAM. The Poseidon Graph Database provides also support for the storage of data directly on disk or Persistent Memory using similar data structures but optimized for utilization of the underlying storage. However, for the scope of this paper, we focus on the implementation of the storage on DRAM. For the underlying data structure, a linked list of fixed-size arrays (*chunks*) is used, which is referred to as a chunked vector. Furthermore, the nodes, relationships, and properties are stored in fixed-size records within the appropriate chunked vector. For entries with variable sizes, such as string values, an entry is created in a dictionary, and the corresponding dictionary code is stored in the respective record. To achieve the connection between nodes and relationships, the offsets of the respective entries are used. A node record stores the offsets of the first incoming and outgoing relationships. This offset points to an entry in the relationship table. A relationship record contains the offsets of the source and destination nodes. Moreover, the corresponding relationships are also linked to each other. A relationship record, therefore, contains the next offset of the relationship list of the source and destination node. The traversing through a graph can be achieved by alternately searching the node and relationship table for the offsets of the respective records.

4.2 Graph Queries

Processing graph database queries consists mainly of discovering a path between nodes in a graph. Besides the usual operators known from relational DBMSs like selections, projections, or joins, the Poseidon Graph Database provides an additional set of operators, especially for the processing of graph queries. These operators are based on graph algebra which is an extension of relational algebra [HG16]. For the data flow between operators, we implemented a push-based query processing approach. Here, the operators are organized into a pipeline and push their results toward the consuming operator until a pipeline breaker occurs [NL14]. For various reasons regarding the simplicity of a graph query language, we implemented an easy and manageable query language oriented to graph algebra.

```
Project({{0,"name"},{2,"name"}},
 Expand(OUT, "Person",
  ForeachRelationship(FROM, ":friendOf",
   NodeScan("Person"))))
```

Fig. 2: Example Graph Query in Poseidon

An example graph query in our language is given in 2. The aim of this query is to find all *Person* nodes in the graph, which are connected to another *Person* node with a *:friendOf* relationship, resulting in an overview of all friends in a graph.

The entry point of every query in Poseidon is the `NodeScan` operator. As the name suggests, it scans the underlying table for nodes, compares optionally each node with a given label, and pushes the appropriate nodes to the next operator. A scan as shown in the example is actually a scan combined with a filter for finding nodes with the given label. Because strings are dictionary encoded, this kind of filter is a simple integer comparison representing an appropriate candidate for offloading to UPMEM.

The nodes can be then processed with the `ForeachRelationship` operator, in order to find an ingoing or outgoing relationship of the previous node. Optionally, it compares the relationship label with a given label. The found relationship is then passed to the next operator. A relationship tuple can then be processed using the `Expand` operator. This operator extracts the source or destination node of the handed relationship. With these operators, it is possible to traverse a graph to find paths between two nodes.

4.3 Query Processing

For the processing of graph queries, Poseidon's query engine relies on push-based query processing and Morsel-driven parallelism [Le14]. The data flow at query processing is organized in a pipeline, and the resulting tuples are pushed from one operator toward their consuming operator. This flow continues until a pipeline breaker is reached. For parallelism, the engine exploits Morsel-driven parallelism using the underlying chunked vector data structure. Before the execution of the query, the query and each individual chunk of the chunked vector will be assigned to the task and pushed into a task pool. When executing the query, the engine spawns several threads which pull a task from the pool and executes the given query on this task until all tasks are processed.

The query engine provides three different execution modes for the actual processing: executing ahead-of-time (AOT) compiled code, just-in-time (JIT) compilation, and an adaptive approach. The AOT-compiled mode processes the given query using pre-compiled C++ methods, which execute the given operators. The JIT-compilation mode transforms the given graph query into highly optimized machine code and executes it directly. For this, we use the LLVM compilation framework. The graph query will be transformed into a single function in LLVM IR. Then, it will be optimized using several optimization passes like dead-code elimination or instruction combining. The resulting optimized LLVM IR code will then be transformed into machine code and executed by the engine. To hide compilation time, the engine can execute queries in the adaptive mode. Here, the engine starts the query processing using the AOT-compiled mode and compiles the query in the background. As soon as the compilation is complete, it switches to the new compiled code. Additionally, this

mode is useful to hide access latencies of the underlying storage type like disk or persistent memory [BJS21].

5 PIM-based Table Scans

The starting points of most queries are table scans. Often there is no other way than to traverse the entire table for tuples that match a given predicate. Especially in the case of predicates with particularly low selectivity, tuples that do not correspond to the predicate must be transferred unnecessarily via the CPU of the system. This procedure in today's usual systems leads to a bottleneck and reduces the possible performance. In this section, we will show the possibility of implementing a table scan operator by exploiting the PIM technology.

5.1 Memory layout

For the execution of table scans on the DPU, the memory of the DPUs must be taken into account according to their characteristics. The tables of nodes, relationships, and properties of the Poseidon Graph database are based on the chunked vector data structure. The chunking of the table can also be exploited for the design of the memory layout on the DPUs.

```
struct mram_node {
 uint8_t tx_pad[40];
 uint64_t id;
 uint64_t from_rship_list;
 uint64_t to_rship_list;
 uint64_t property_list;
 uint32_t node_label;
};
```

List. 1: Structure of node in MRAM

```
struct mram_chunk {
 struct mram_node data[C_ELEMENTS];
 struct mram_chunk* next;
 char bitset[BS_SIZE];
 uint32_t first;
 char padding[PAD_SIZE];
};
```

List. 2: Structure of chunk in MRAM

Listing 1 shows the structure of the nodes and Listing 2 is the structure of the chunks which are stored in MRAM. The required size of the nodes for storing the necessary data is 80 bytes. We leave the parts that are used for transactional processing out of the scope of this paper for the moment and label them as `tx_pad`. A similar structural layout is used to represent the relationships and properties in the MRAM of the DPUs. Basically, the representations are equivalent to those used for storing the data in the main memory of the host. In addition, care was taken that the size is a multiple of 8 bytes to allow transfer to MRAM and between MRAM-WRAM without additional transformation. The appropriate alignment of the data allows the direct transfer to the DPU but also buffering a part of the data in the DPU WRAM for faster access. With large tables, it may happen that more chunks exist than available DPUs. To use the memory of a DPU efficiently and to process as much

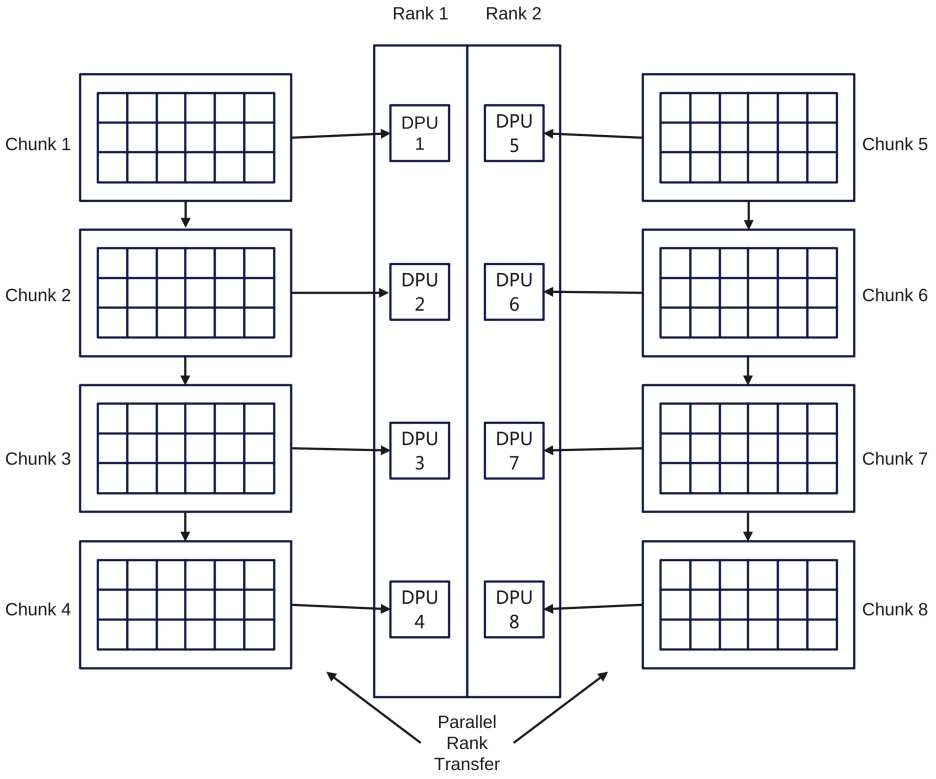

Fig. 3: Rank Parallel Chunk to DPU Assigment.

data as possible on it, it is more beneficial to transfer several chunks to a DPU. Considering the chunk size of 65536 bytes, we reserve in each DPU space that is able to hold up to 1000 chunks. The remaining MRAM space can be used for parameters such as the number of chunks passed, filter arguments, and storing the results.

5.2 Chunk-DPU Assignment

In order to transfer the data efficiently, we make use of asynchronous and parallel host-to-DPU data transfer. To implement the data transfer as efficiently as possible the data must be transferred as parallel as possible. This is achieved with DPU and Rank parallel data transfer. The underlying data structure in Poseidon, which is used for the storage of nodes and relationships, is perfectly suited to achieve this with the least possible implementation effort. Fig. 3 shows the rank parallel chunk-to-DPU assignment. Each chunk is assigned a DPU on a rank in a round-robin way. After the assignment is done, the data is transferred

in parallel per rank. This ensures that the workload on all DPUs is similar. Furthermore, multiple chunks can be assigned to a DPU to make efficient use of the available MRAM memory. If the table does not fit completely into the MRAM, the program must be executed with the already transferred part. Then the remaining part must be transferred back to the DPU. The following listing shows the algorithm for the chunk to DPU assignment.

```
foreach(chunk) {
  DPU_RANK_FOREACH(set, rank) {
    DPU_FOREACH(rank, dpu) {
      dpu_prepare_xfer(dpu, chunk);
      dpu_push_xfer(rank, DPU_XFER_DEFAULT, "mram_chunks",
                offset, CHUNKS_SIZE, DPU_XFER_ASYNC);
      calc_offset(); }}}
```

List. 3: Host to DPU chunk transfer algorithm

The algorithm iterates over the available chunks of nodes, relationships, or properties. Then it iterates over the DPU of a rank. This is advantageous to allow efficient parallel data transfer, as the buffer for the data transfer must be the same size and write to the same offset address in the MRAM. Otherwise, the transfer would be serial.

5.3 DPU Scan

To enable an efficient multithreading scan of the chunks, we divide the workload among all available tasklets. For this, we distribute the elements to all available tasklets per chunk. Each tasklet thus works on an allocated area in each chunk assigned to the DPU. Then each tasklet iterates over the allocated area of the chunks. In each iteration, a record is checked for a given filter predicate. As soon as a record matches the predicate, the result is saved by setting the corresponding position of the record in the chunk to 1 in a bit vector. Per DPU there is a single bit-vector for each passed chunk. This tasklet design also has the advantage that no further synchronization mechanisms are necessary since each tasklet writes the result to its own memory area.

6 Evaluation

We use the Social Network Benchmark (SNB) dataset from the Linked Data Benchmark Council (LDBC) for the following benchmarks. This is an applicable benchmark to evaluate the performance of this approach in a graph DBMS. The used scale factor of the dataset is 1. We further restrict ourselves to the nodes of the dataset which we store in a nodes table in Poseidon. In total, the table contains 1.180.565 entries, which are stored in 1445 chunks in DRAM with 817 entries per chunk. For the scan query, we scan the node entries for nodes labeled as `Post` with a selectivity of 10%.

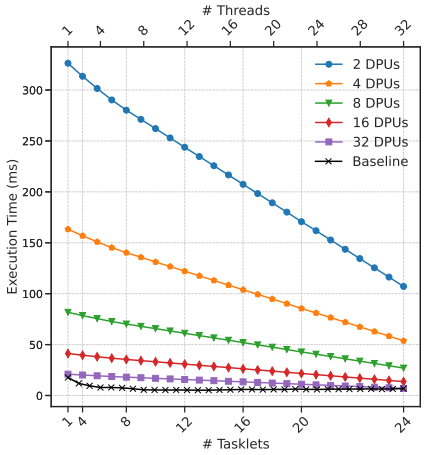

Fig. 4: Table Scan with 2 - 32 DPUs

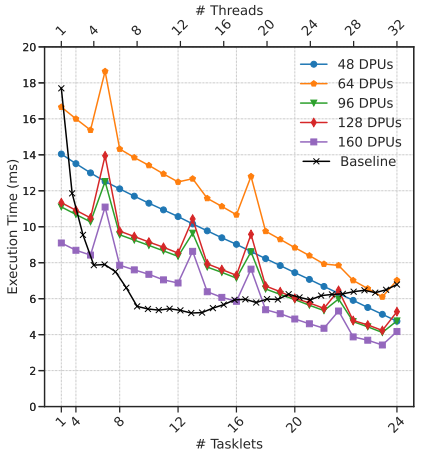

Fig. 5: Table Scan with 48 - 160 DPUs

6.1 System

The system used for the following benchmarks runs with two Intel Xeon Silver 4215R with a total of 16 cores with 2 threads each. A total of 32 threads can be executed on the system. Furthermore, the system has 512 GB of DRAM, which is made up of 8 x 64 GB DIMMs. In terms of PIM, the system has 4 UPMEM DIMMs with 16 GB each. Each UPMEM DIMM has 2 ranks with up to 64 DPUs each. The total number of DPUs is 510, divided into 8 ranks. The clock rates of the DPUs are between 200-400 MHz. The system runs under Ubuntu 20.04.1 with Linux kernel 5.4.0. The code of the host and DPU program was compiled with Clang at version 12 and full optimization at -O3.

The data layouts of the baseline and the DPU implementation are based on the same data structures and the same optimization to get a fair comparison.

6.2 DPU Parallelism

Fig. 4-Fig. 6 show the execution of table scans with different numbers of DPUs as well as with different numbers of tasklets. The baseline in these experiments is the usual CPU execution of the table scan with varying numbers of hardware threads (1-32). Furthermore, each thread performs the scan operation on the same number of chunks. The results in the baseline execution are saved in a result vector similar to the DPU program in order to obtain a workload as similar as possible. With a particularly large number of DPUs (164 or more), the execution runtimes change only slightly, since the size of the workload also changes only very slightly. The sweet spot for parallelism is around 12-24 task sets and a DPU count of around 160 for this table size.

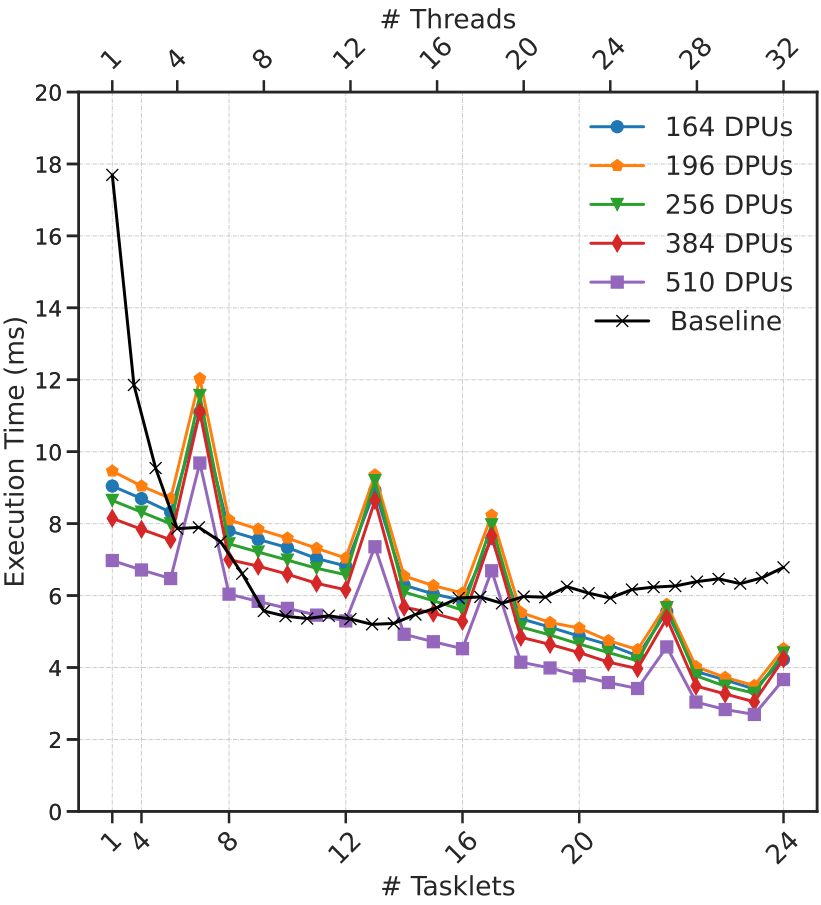

Fig. 6: Table Scan with 164 - 510 DPU

As the number of tasklets increases, the parallelism of the execution also increases. This can further improve the runtime. Furthermore, with an increasing number of DPUs the parallelism increases additionally. However, it can be seen that around 32 DPUs, which are used for the table scan, the runtime approaches the baseline of the CPU execution more and more. From 128 DPUs and the maximum number of 24 tasklets, the PIM execution is even faster than the CPU execution with all available hardware threads.

In summary, it can be concluded that the full utilization of the parallelism of the tasklets and a high number of DPUs can improve the runtimes of table scans by a considerable amount.

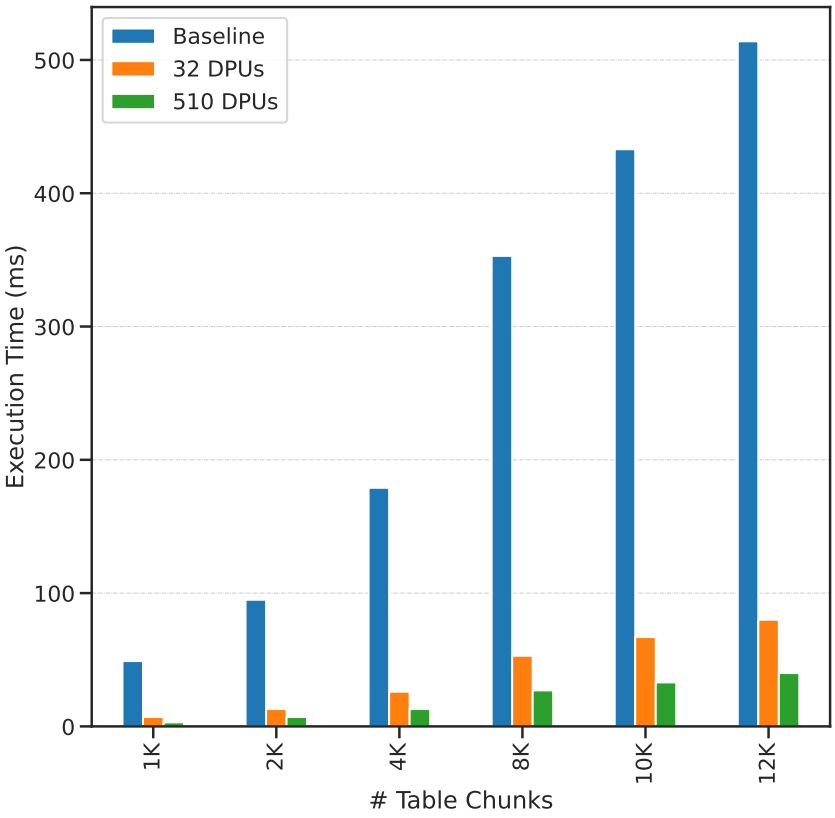

Fig. 7: Execution time of table scan on different table sizes.

6.3 Table Size

The execution times of the DPU scan with different table sizes are given in Fig. 7. The baseline of this experiment is again the execution of the table scan on the CPU with all available hardware threads. Each of these hardware threads executes the scan for several chunks. To achieve a similar workload, the baseline execution stores the result in a result vector, similar to the DPU program. The table size is represented by a different number of chunks. One chunk contains up to 817 records. A table that consists of 1000 chunks (1K) contains up to 817.000 entries. For this benchmark, we created an additional graph from using the LDBC SNB dataset containing 50% Post-nodes and 50% Person-nodes.

The linear increase in the execution time can be seen directly for all executions. Furthermore, the table scan on the DPU itself with a small number of 32 DPUs is much faster than the corresponding execution on the CPU with 32 threads. The high task parallelism can lead to very fast processing of the scan. Anyway, to enable the highest possible parallelism of the DPUs, it is necessary to have as little inter-DPU communication as possible and as little synchronization as possible at the tasklet level. In our approach, each tasklet worked on its own allocated memory space on the MRAM. Thus, no synchronization mechanisms were necessary. The result is a significant runtime improvement of the table scan.

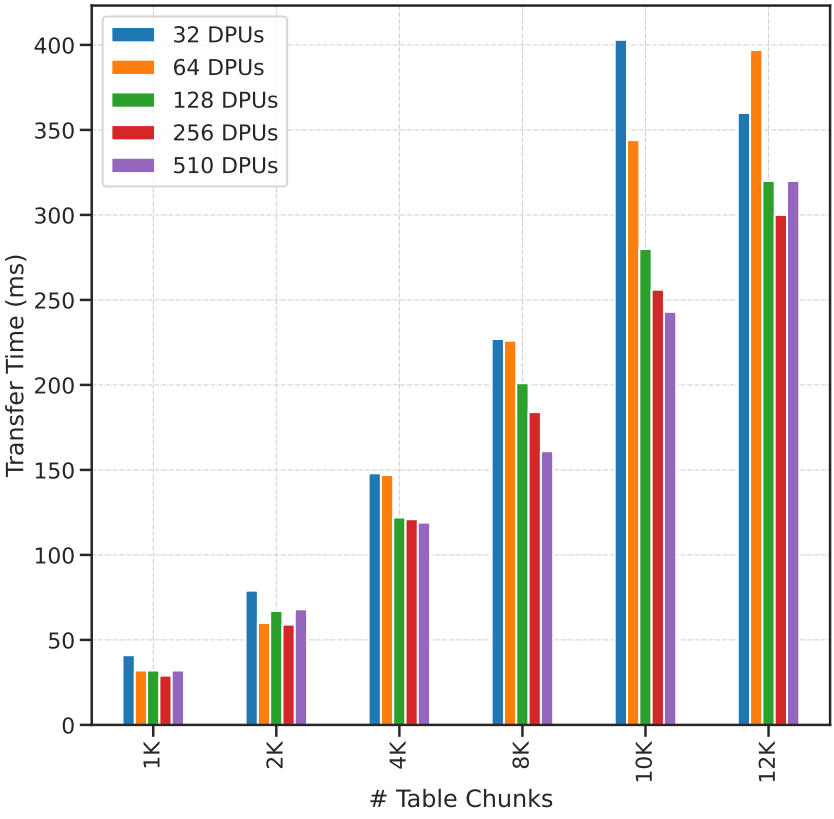

Fig. 8: Table transfer times from host to DPUs with different numbers of DPUs.

6.4 Data transfer

Fig. 8 shows the transfer times of large tables for different numbers of DPUs. For this experiment, we created several graphs with different numbers of chunks, ranging from 1000 (1K) to 12000 (12K). Each chunk contains up to 817 node records. We considered the transfer times on different numbers of DPUs to study the parallel data transfer. Furthermore, we made sure that the total number of chunks is distributed among the DPUs. If the amount of data does not fit into the memory of the DPUs, the data transfer would have to be executed multiple times. The results clearly show that the transfer times increase as the number of data increases. Parallel data transfer can reduce the transfer times by a few milliseconds. For example, the transfer time can be decreased by half if all 510 DPUs are used instead of 32.

Since the data only has to be loaded into the memory of the DPUs at the startup of the database, the result of the data transfer is acceptable. However, by using interleaved execution, the transfer times of data can be hidden. However, these possibilities are outside the scope of this paper.

7 Conclusion

The table scan is the most basic operator in the query processing of databases. In this work, we have investigated how we can accelerate table scans with filters using PIM technology. However, this approach requires an adapted design, since this new paradigm brings different characteristics with it, such as the transfer of data and the partitioning of the workloads in order to achieve the highest possible parallelism. As shown in the benchmarks presented here, PIM technology can outperform the runtime of a comparable CPU execution. To achieve this, however, high parallelism is needed. Furthermore, PIM technology can also be used to improve other operators. It is conceivable that especially the operators which are needed in graph databases for traversing can be further improved in their runtime.

Even when utilizing the CPU with its maximum possible parallelism, the results cannot come close to the runtimes of a table scan on multiple DPUs. With very large tables, this effect becomes even more pronounced. This has several implications for the execution of table scans. To save as much runtime as possible, as much data as possible, if not all of it, must be transferred to the DPUs' memory. A problem that comes along with this is data transfer. As shown in the evaluation, the transfer times increase with increasing table size. To take advantage of this as much as possible, the data must be transferred to the memory when the DBMS is started. The execution of updates and the maintenance of the consistency of this data is another problem, which is outside the scope of this paper. Furthermore, the data transfer can be optimized by using the possible bandwidth of the DPUs to transfer as much as possible in parallel.

The economic aspect of the currently available PIM hardware cannot be fairly measured and compared to the baseline hardware at the time of this work. Current hardware is currently only prototypically distributed by the UPMEM company.

8 Outlook

In future work, plan to integrate the compilation process of DPU programs directly into the query compiler. This would allow more flexible filter operations instead of only pre-coded filters. Furthermore, the design of an approach for asynchronous execution can be subject to investigation. The data transfer times are particularly high for very large tables. If this data were to be transferred to the DPUs before query execution, this would have a negative impact on the overall response time of the system. With an adaptive design, which transfers data asynchronously and executes it in parallel on different DPUs or on the rank level, this problem can be efficiently overcome. Finally, having multiple memory technologies including PIM available in a database server raises the question of data placement and/or efficient data transfer.

Acknowledgements. This work was partially funded by the German Research Foundation (DFG) in the context of the project "Hybrid Transactional/Analytical Graph Processing in Modern Memory Hierarchies (#TAG)" (SA 782/28-2) as part of the priority program "Scalable Data Management for Future Hardware" (SPP 2037), "Processing-In-Memory Primitives for Data Management (PIMPMe)" (SA 782/31) as part of the priority program "Disruptive Memory Technologies" (SPP 2377), and by the Carl-Zeiss-Stiftung under the project "Memristive Materials for Neuromorphic Electronics (MemWerk)".

Bibliography

[BJS21] Baumstark, Alexander; Jibril, Muhammad Attahir; Sattler, Kai-Uwe: Adaptive Query Compilation in Graph Databases. In: 37th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2021, Chania, Greece, April 19-22, 2021. IEEE, pp. 112–119, 2021.

[Bo17] Boroumand, Amirali; Ghose, Saugata; Patel, Minesh; Hassan, Hasan; Lucia, Brandon; Hsieh, Kevin; Malladi, Krishna T.; Zheng, Hongzhong; Mutlu, Onur: LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory. IEEE Computer Architecture Letters, 16(1):46–50, 2017.

[Dr02] Draper, Jeff; Chame, Jacqueline; Hall, Mary; Steele, Craig; Barrett, Tim; LaCoss, Jeff; Granacki, John; Shin, Jaewook; Chen, Chun; Kang, Chang Woo; Kim, Ihn; Daglikoca, Gokhan: The Architecture of the DIVA Processing-in-Memory Chip. In: Proceedings of the 16th International Conference on Supercomputing. ICS '02, Association for Computing Machinery, New York, NY, USA, p. 14–25, 2002.

[Gi22] Giannoula, Christina; Fernandez, Ivan; Gómez-Luna, Juan; Koziris, Nectarios; Goumas, Georgios; Mutlu, Onur: , Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Systems, 2022.

[Gó22] Gómez-Luna, Juan; Hajj, Izzat El; Fernandez, Ivan; Giannoula, Christina; Oliveira, Geraldo F.; Mutlu, Onur: Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. IEEE Access, 10:52565–52608, 2022.

[Gu] Guo, Juan Gómez-Luna1 Yuxin; Brocard, Sylvan; Legriel, Julien; Cimadomo, Remy; Oliveira, Geraldo F; Singh, Gagandeep; Mutlu, Onur: Machine Learning Training on a Memory-Centric Computing System.

[HG16] Hölsch, Jürgen; Grossniklaus, Michael: An Algebra and Equivalences to Transform Graph Patterns in Neo4j. In (Palpanas, Themis; Stefanidis, Kostas, eds): Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference, EDBT/ICDT Workshops 2016, Bordeaux, France, March 15, 2016. volume 1558 of CEUR Workshop Proceedings. CEUR-WS.org, 2016.

[Ji21] Jibril, Muhammad Attahir; Baumstark, Alexander; Götze, Philipp; Sattler, Kai-Uwe: JIT happens: Transactional Graph Processing in Persistent Memory meets Just-In-Time Compilation. In (Velegrakis, Yannis; Zeinalipour-Yazti, Demetris; Chrysanthis, Panos K.; Guerra, Francesco, eds): Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021. OpenProceedings.org, pp. 37–48, 2021.

[Ka22] Kang, Hongbo; Zhao, Yiwei; Blelloch, Guy E; Dhulipala, Laxman; Gu, Yan; McGuffey, Charles; Gibbons, Phillip B: PIM-tree: A Skew-resistant Index for Processing-in-Memory. arXiv preprint arXiv:2211.10516, 2022.

[La16] Lavenier, Dominique; Deltel, Charles; Furodet, David; Roy, Jean-François: BLAST on UPMEM. PhD thesis, INRIA Rennes-Bretagne Atlantique, 2016.

[Le14] Leis, Viktor; Boncz, Peter A.; Kemper, Alfons; Neumann, Thomas: Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In (Dyreson, Curtis E.; Li, Feifei; Özsu, M. Tamer, eds): International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014. ACM, pp. 743–754, 2014.

[Ng20] Nguyen, Hoang Anh Du; Yu, Jintao; Lebdeh, Muath Abu; Taouil, Mottaqiallah; Hamdioui, Said; Catthoor, Francky: A Classification of Memory-Centric Computing. ACM J. Emerg. Technol. Comput. Syst., 16(2):13:1–13:26, 2020.

[Ni21] Nider, Joel; Mustard, Craig; Zoltan, Andrada; Ramsden, John; Liu, Larry; Grossbard, Jacob; Dashti, Mohammad; Jodin, Romaric; Ghiti, Alexandre; Chauzi, Jordi; Fedorova, Alexandra: A Case Study of Processing-in-Memory in off-the-Shelf Systems. In (Calciu, Irina; Kuenning, Geoff, eds): 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021. USENIX Association, pp. 117–130, 2021.

[NL14] Neumann, Thomas; Leis, Viktor: Compiling Database Queries into Machine Code. IEEE Data Eng. Bull., 37(1):3–11, 2014.

[Pa97] Patterson, D.; Asanovic, K.; Brown, A.; Fromm, R.; Golbus, J.; Gribstad, B.; Keeton, K.; Kozyrakis, C.; Martin, D.; Perissakis, S.; Thomas, R.; Treuhaft, N.; Yelick, K.: Intelligent RAM (IRAM): the industrial setting, applications, and architectures. In: Proceedings

International Conference on Computer Design VLSI in Computers and Processors. pp. 2–7, 1997.

[UP22] UPMEM: , https://www.upmem.com/, 2022.

[WM95] Wulf, William A.; McKee, Sally A.: Hitting the memory wall: implications of the obvious. SIGARCH Comput. Archit. News, 23(1):20–24, 1995.

[Zh14] Zhang, Dongping; Jayasena, Nuwan; Lyashevsky, Alexander; Greathouse, Joseph L.; Xu, Lifan; Ignatowski, Michael: TOP-PIM: Throughput-Oriented Programmable Processing in Memory. In: Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing. HPDC '14, Association for Computing Machinery, New York, NY, USA, p. 85–98, 2014.