Adaptive Query Compilation with Processing-in-Memory

Alexander Baumstark
TU Ilmenau, Germany
alexander.baumstark@tu-ilmenau.de

Muhammad Attahir Jibril
TU Ilmenau, Germany
muhammad-attahir.jibril@tu-ilmenau.de

Kai-Uwe Sattler
TU Ilmenau, Germany
kus@tu-ilmenau.de

***Abstract*—The challenge of today's DBMS is to integrate modern hardware properly in order to provide efficiency and performance. While emerging technologies like Processing-in-Memory (PIM) reduce the bottleneck when accessing memory by offloading computation, DBMSs must adapt to the new characteristics and the provided processing models in order to make use of it efficiently. The Single Program Multiple Data (SPMD) models require a special precompiled program for PIM-enabled chips in the UPMEM architecture. Integrating this model into the query processing of a DBMS can improve the overall performance by efficiently exploiting the underlying characteristic of high parallel execution directly on memory. To address this, we propose an approach to integrate this programming model directly into the query processing by leveraging adaptive query compilation. The experiment results show an improvement in the execution times compared to the execution on non-PIM hardware.**

***Index Terms*—Processing in Memory, UPMEM, Query Compilation, Graph Database**

I. Introduction

Most modern in-memory databases aim to provide high performance with low latency and high throughput. Due to the different development processes between CPU and main memory, memory-bound operations in DBMS become increasingly a problem due to a bottleneck (or memory wall), which affects the possible performance of modern systems. Processing-in-Memory (PIM) is a new opportunity that solves this problem by moving computation directly to memory. Through the company around UPMEM, there is already real hardware that provides PIM-enabled DIMMs commercially [1]. However, UPMEM introduced hardware with new characteristics using a special programming model for the hardware. In order to use this hardware for query execution in databases efficiently, the characteristics and programming models of the hardware must be adapted into query processing. The UPMEM architecture follows the Single Program Multiple Data Model [2], which requires a specially compiled program for the PIM-enabled chips to execute. This code is compiled ahead of time (AOT), which means that only static query parts can be executed with it. Consequently, operators must be precompiled. By integrating this compilation process this problem can be solved, which however again leads to additional compilation times. Adaptive query compilation is a solution to this problem. By providing multiple execution modes for queries, the query can be interpreted during compilation and the efficient code can be called once compilation is complete. Furthermore, this approach allows the execution of different filter expressions with the PIM technology directly in the query pipeline of databases without the need to provide previously compiled programs. With this work, we show an approach to integrate new hardware with its own characteristics into an existing adaptive query engine with low effort.

The contributions of this work are as follows:

- We show an approach to generating efficient machine code from queries for the execution using the PIM technology.
- We show an approach to integrate the PIM compilation approach of the UPMEM architecture into query processing.
- We introduce a method to adaptively switch between query execution modes in order to hide the compilation times of PIM programs.
- We evaluate our approach against the execution of non-PIM optimized execution.

We built this work around the Poseidon Graph Database[1], an HTAP graph database implemented to exploit the memory hierarchy of modern systems. Initially, Poseidon was explored for exploiting persistent memory. However, the approaches developed for this purpose can also be efficiently transferred for use as an in-memory database in DRAM.

II. Related Work

a) Code Compilation: Query compilation is a widely known and researched technique for improving query performance through compilation. In general, the approach in query compilation can be divided into template-based and IR-based compilation. The template-based approach supplies query templates with the query arguments provided by the user. Then, with the use of a high-level compiler, e.g., GCC or clang, it transforms them into machine code. An example of this approach is Hekaton, a database engine for Microsoft's SQL Server using this technique [3]. It transforms algebra plans through several optimization steps into high-level C code. Due to the high compilation times of high-level compilers, the performance of this approach is low. IR-based approaches use an intermediate representation (IR) for machine-code generation. [4] provides a query compiler that is based on the LLVM framework. Based on this work, [5] proposed an

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

approach to adaptively switch between execution modes to avoid the waiting time for compilation. An interpreter is used to execute the query while the compilation runs. Especially for short-running queries, the execution time can be improved, since their compilation time can be higher than the actual processing [6]. Other approaches provide their own IR for the generation of machine code. [7] showed an approach that estimates value lifetimes before code generation. The Voodoo IR is a declarative algebra, that provides a set of vectorization instructions to generate OpenCL code [8].

Integrating new hardware is also a challenging aspect when writing a query compiler as they often have other architectural characteristics or programming models than the host. The research ranges from integrating GPUs to FPGAs [9]–[12].

The work of [9] provides an approach to integrate FPGAs into a database engine without further changes in the data layout. The evaluation shows an enormous saving when executing queries using the FPGA. [10] demonstrated an approach to compiler analytical queries into FPGA programs in order to execute text analytics. For GPU-based query processing, the work of [11] provided a strategy to transform multiple operations of a query into a single GPU kernel. The authors of r3d3 provided a hybrid AOT and JIT compiled approach to compile queries into a GPU program [12]. Further, the work of [13] integrated GPUs into the query engine. The evaluation of this query engine showed that it can outperform other GPU-based query engines.

b) Processing-in-Memory: PIM has been a well-known technology for improving the CPU-memory bottleneck for several decades. On the technical side, there have been a number of concepts that have enabled PIM since the 1990s [14], [14], [15]. However, the high cost and lack of industrial support for this concept prevented the commercial production of real PIM hardware. Nonetheless, research was conducted based on prototypes. Since PIM Technology follows a similar approach to GPU Processing, the applicability of the design space of GPU-accelerated architectures to PIM was investigated in the work of [16]. The authors of [17] published a mechanism to reduce the data exchange between CPU and PIM cores by means of caching, referred to as LazyPIM. The company around UPMEM published with their architecture the first real hardware which enables PIM [1]. There are already a number of works concerning this architecture investigating its characteristics and applicability. The work of Gomez-Luna et. al under the architecture for its limitations and performance as well as energy consumption [18]. The result of the work shows that the UPMEM system achieves particularly good performance as long as the individual components (DPUs) do not require communication (DPU-to-DPU).

However, to our knowledge, there exists no DBMS that exploits adaptive query compilation with PIM to enhance query processing.

III. Background

Before considering the approach for the adaptive compilation of PIM programs, this Section is used to introduce the basics of the Poseidon Graph Database and the UPMEM architecture which are the used systems for which the proposed approach is implemented. Nevertheless, the key techniques and algorithms can be applied to any other DBMS.

A. Graph Database Poseidon

As mentioned above, Poseidon is formally optimized to exploit the characteristics of PMem. However, the optimizations for PMem can also be transferred to DRAM. Therefore, we use Poseidon as the system for the implementation of our approach.

a) Data Model: The underlying data model of the Poseidon Graph Database is the Labeled-Property Graph. Within this model, the data is organized as a graph consisting of nodes that are connected with relationships. Further, labels can be directly assigned to the nodes and relationships. Formally, we define the model for the Poseidon Graph Database as follows:

Theorem 1. *A graph G consists of nodes N and directed relationships $R \in N \times N$, denoted by $G = (N, R)$. A node $n \in N$ is identified by a unique identifier $id : N \rightarrow ID$. From the set of labels L, a label is assigned to each node and relationship using the label function $l : (N \cup R) \rightarrow L$. Further, a property is a key-value pair $(k, v) \in P$. The properties P are $P = K \times D$, where K is the set of property names and D is the property values. Properties can be assigned using $p : (N \cup R) \rightarrow \mathcal{P}(P)$, using the powerset $\mathcal{P}$.*

Using this model we are able to create, store, and process arbitrary graphs.

b) Storage: In order to store arbitrary graphs efficiently we store the nodes, relationships, and their properties in separate tables. As the underlying data structure for the storage of the tables, we use the same data structure that is referred to as a chunked vector [19]. A chunked vector is fundamentally a linked list of fixed-size arrays (chunks) where available slots are indicated using a bitset. A new record will be inserted into the first available space according to the position of the first unset bit of the bitset. Whenever a chunk is full, a new chunk will be allocated and linked with the last chunk in the vector. The node, relationship, and property records have always a fixed size. This is done by using a dictionary for variable-sized fields, like strings. The appropriate dictionary code is then placed in the field of the record. The connection between nodes is done via relationships. Each node record maintains two offsets, which indicate the first ingoing and outgoing relationship of the node. Further, the relationship records maintain the source and destination node id and as a field, as well as the next relationship of the node records. This enables traversing the graph by processing through the appropriate relationship lists of the nodes. A similar approach is used for the properties: each node maintains the first property item id as a field. Belonging properties items are linked with their identifiers.

c) Query Processing: For the processing of queries, the Poseidon Graph Database provides operators that are based on graph algebra. Graph algebra extends the relational algebra by

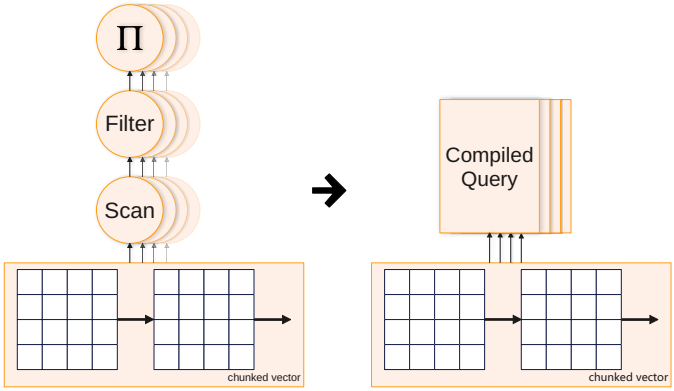

Fig. 1. Query Pipeline of the Poseidon Graph Database

more operators in order to traverse the underlying graph data. For Poseidon, we implemented three additional operators in order to traverse the graph:

- `NodeScan`: scans the nodes table and filters them by a given label
- `ForeachRelationship`: iterates through the relationship list of a node by a given direction (to, from)
- `Expand`: extracts the source or destination node of a relationship

Further, the query operators in Poseidon are organized in a push-based pipeline, according to their position in the query. Figure 1 shows the query pipeline of Poseidon using the push-based approach. With push-based processing, the control flow of the operators is the same as the data flow, which is beneficial to generate efficient machine code. The results of an operator are pushed to the next operator in the pipeline until a pipeline breaker is reached. A pipeline breaker can be a materializing operator that writes data to the storage or a collecting operator to return the results to the user. Additionally, we make use of Morsel-driven parallelism for the execution of queries [20]. The chunks of the tables (Morsels) are assigned to a task and pushed into a task pool. The available threads execute the query by pulling a task from the pool and executing the query on the assigned chunks of the task until no task is left.

The Poseidon Graph Database provides three modes for the execution of graph queries:

- Interpretation
- Compilation
- Adaptive Query Compilation

The interpretation mode uses AOT-compiled operators to execute a query. For each operator, the query engine executes the appropriate compiled operator code. The compilation modes transform the graph query into LLVM IR and compile it into an optimized machine code using the LLVM framework. The adaptive query compilation approach combines both previous approaches [21]. First, it starts the query execution in the interpretation mode and compiles the query in a background thread. As soon as the compilation is complete, it switches to compiled machine code.

B. Processing-in-Memory

PIM is a technology to solve the memory-wall phenomenon of today's system architectures. This problem describes the performance gap between the CPU and the memory over time. While the CPU performance of modern CPUs increased over time, memory performance in terms of bandwidth and latency stabilized. The effect of this is that modern CPUs wait most of their execution time for memory transfer. With PIM, the processing of data is directly performed on the memory, enabling a higher bandwidth, lower latency, and less energy consumption. For a long time, no commercialized implementation of PIM-enabled hardware existed. However, this has changed with the release of PIM-enabled chips by UPMEM.

a) UPMEM Architecture: The UPMEM architecture is the first real PIM-enabled hardware released. The core of this architecture is built by UPMEM DIMMs which are usual DDR4 DRAM DIMMs with a frequency of 2400 MHz and equipped with PIM-enabled chips. Further, the chips are organized into ranks, where each DIMM comprises up to two ranks in the current release. Each rank can hold up to 8 PIM-enabled chips. Then, a PIM-enabled chip holds up to 8 DRAM Processing Units (DPUs), where each DPU has access to its own Main RAM (MRAM), Instruction RAM (IRAM), and Working RAM (WRAM). These different memories have different restrictions in terms of accessibility and size. The MRAM has a size of 64 MB. It is accessible by the DPU itself and the host. The purpose of this memory is to communicate with the host, by host-to-DPU and DPU-to-host data transfer. The IRAM has a size of 24 KB and stores the actual program with the instructions of the DPU, while the WRAM with a size of 64 KB is meant to store the stack and heap data while executing the DPU program. DPUs have only access to their own memories. Therefore, there is no direct communication between the DPUs possible.

The DPU itself is a 32-bit RISC core with a maximum frequency of 400 MHz. It supports multithreading with up to 24 hardware threads. As these threads share the same memory, synchronization mechanisms like mutexes, barriers, or semaphores are required when accessing critical sections.

b) Programming Model: The programming model for the UPMEM DPUs follows the Singe Program Multiple Data model, which implies that every DPU is executed with the same single program but with different data. DPU environment allocation, data preparation, and execution are managed by the host application. For development, the UPMEM SDK provides two different libraries, one for the host, and the other for the DPU program. Both are for development using C or C++. The library for the DPU program comprises the usual C standard library adapted to the DPU architecture. Further, it contains synchronization primitives like mutexes, barriers, and semaphores. The number of used threads, also referred to as tasklets, must be defined before the compilation of the DPU program. For memory management of the DPU programs, the programmer must define the placement of memory regions

explicitly, i.e., placement of the memory region in MRAM or WRAM.

```
#include <mram.h>
__mram_noinit struct mram_chunk mrc[24];
__mram_noinit uint64_t found_results[24];
int main() {
   int tasklet_id = me();
   found_results[tasklet_id] = 0;
      for(int i = 0; i < 1024; i++) {
      if(mrc[tasklet_id].record[i].id = 42)
         found_results[tasklet_id]++;
   }
}
```

The Listing above shows a simplified DPU program for the scan of a table from the Poseidon Graph Database. The developer has to specify the memory regions in MRAM using the `__mram` or `__mram_noinit` specifier. The latter does not provide an initialization with the effect of a smaller binary size of the program. The data from the host is placed inside the specified MRAM region `mrc`. The program itself is executed with all available 24 threads. A thread can retrieve its id using the `me()` function provided by the runtime library. In order to improve the concurrent execution each thread works on its own memory region when writing the results. Therefore, no further synchronization is necessary. The results written in the `found_results` variable can be retrieved by the host using a DPU-to-host data transfer. This program can be executed on all, specific ranks or set of DPUs by the host but has to be pre-compiled using the special compiler for DPU programs given by the UPMEM SDK. The workflow of the host for the execution of PIM programs with UPMEM DPUs can be summarized into the following steps: 1) DPU (ranks) and program allocation, 2) Host-to-DPU(s) buffer population, data transfer, 3) Execution of DPU program, 4) DPU(s)-to-host data transfer.

The host has to specify the number of DPUs for the future execution of the DPU program. Then, the host has to load the pre-compiled DPU binary using the `dpu_load` instructions. The data transfer between the host and DPUs access always the MRAM which requires an 8-byte alignment of the data. Further, there are several ways to transfer data. The first way is to transfer the same buffer from the host to the DPUs. This requires executing the `dpu_copy_to` instruction from the host runtime library. In some cases, this instruction is not sufficient, for example, when each DPU should have a different chunk of the table in order to process data in parallel. In this case, there exist the `dpu_prepare_xfer` instruction which assigns a buffer to one or multiple DPUs. The data is then transferred in parallel by executing the `dpu_push_xfer` instruction. The execution of the DPU program can be triggered by executing the `dpu_launch` instruction. It then starts the previously loaded DPU binary on all allocated DPUs. Additionally, it is possible to execute the data transfer and launch asynchronously which gives the control back to the host. This can be beneficial to process other steps, i.e., data preparation for the next iteration. Using the `dpu_sync` method waits for the completion of all DPUs.

c) Query Processing with PIM: The Poseidon Graph Database supports the execution of AOT-compiled PIM programs for the execution of queries. For this, the whole graph data (the tables) are transferred on startup on the DPUs. In order to provide a high level of parallelism with all available DPUs and threads, the chunks of the tables are distributed across all available DPUs. For the AOT-compiled execution, the database provides for each operator an own DPU program. To process a complete query, each program has to be executed in the order according to the query plan. The state of the DPUs is preserved after each execution of the DPU programs, requiring only the results and arguments to be (re-)initialized. If required data is not present in the DPUs because it would not be transferred due to size, it is transferred before the corresponding operator is executed. After execution, the results can be transferred via a DPU-to-host data transfer and further processed by the host. This allows parallel execution of a scan operator with all DPUs and threads as well as the execution of further queries like filters or graph traversals. However, this procedure requires, in addition to the database, an AOT-compiled DPU program for each operator. Furthermore, not all requests from the user can be executed with this procedure. Therefore, a provision of an adaptive query compilation approach is advantageous.

IV. Adaptive Code Compilation for PIM

Looking at the execution of programs using UPMEM technology, an already compiled program must be available for execution on DPUs. In many cases, especially when executing UDFs, it is not possible to provide these programs because a special program is required by the user. The provision of a generic program that answers every possible request configured by parameters is inherently difficult. This Section describes an approach that integrates the JIT compilation of DPU programs into the query pipeline. However, the compilation of queries is often accompanied by a problem regarding the overall performance since the compilation of small queries can be larger than the actual execution time. Therefore, we show an adaptive approach that uses the usual execution mode when compiling the DPU program in the background and switches to the mode after it.

A. Execution Modes

The Poseidon Graph Database already provides a set of execution modes in order to answer graph queries: interpretation, compilation, and adaptive compilation. The latter mode hides the compilation times of queries by executing the query in the background while processing the query in the interpretation mode. The same approach can be applied to the compilation of PIM programs by two further modes:

- PIM mode
- Adaptive PIM mode

Figure 2 shows the PIM mode query pipeline and compilation. The PIM mode compiles the appropriate part of the

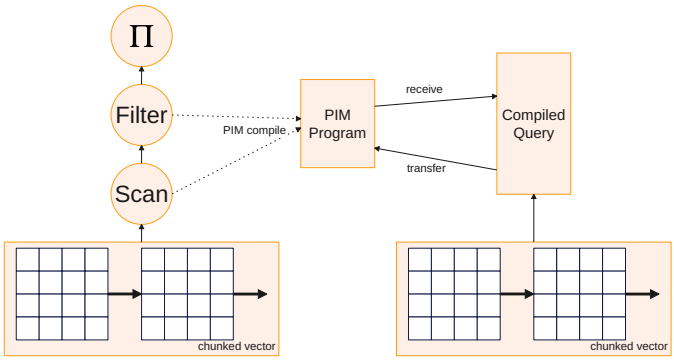

Fig. 2. Query Pipeline of the Poseidon Graph Database

query into a PIM program using the built-in PIM compiler of the query compiler in Poseidon. Currently, the built-in PIM program compiler is only able to transform `Scan`, `Filter`, and `ForeachRelationship` operations into a PIM program. An extension of the compiler to support further operators is planned for future work. The output of the PIM compiler is placed in a memory buffer that will be executed by the host using the provided functions of the UPMEM SDK.

As already mentioned, the compilation of PIM programs is accompanied by further compilation times. Waiting for the completion of the compilation would slow down the execution times enormously. The introduces Adaptive PIM mode is an approach to hide the compilation times of the PIM program. To do this, this execution mode extracts again the relevant parts from the query for the PIM program and starts the compilation in a background thread. While the thread compiles the PIM program in the background, the query is executed in the (non-PIM) adaptive compilation mode, which again starts a thread in the background to compile the query into optimized machine code. While compiling, the query is executed using the interpretation mode.

B. Code Generation

When compiling queries in the Poseidon Graph Database we have to consider two different compilation processes: the usual compilation process to generate X86 machine code from a query, and the compilation process to generate a program for the UPMEM DPUs. The LLVM compilation framework can be used for both purposes since the JIT compiler of Poseidon and the UPMEM program compiler are based on it. This makes it convenient to integrate both compilation processes into a single one. Further, LLVM provides a set of optimization algorithms for the built-in IR that can be used to optimize the code for the host and the UPMEM DIMMs.

a) Non-PIM Code: The workflow of the UPMEM architecture must be integrated into the query pipeline of Poseidon, in order to communicate with the UPMEM DPUs efficiently. As mentioned already, the workflow of a DPU program consists of allocating the DPU, host-to-DPU data transfer, DPU program execution, and DPU-to-host data transfer to retrieve the results. However, the static part like the DPU allocation and the host-to-DPU transfer of the graph can be placed outside the query pipeline, since it is more efficient to execute this part before query execution. For example, when providing a scan operator using PIM technology, the actual data must be copied to the DPUs beforehand. However, there exist cases where the data must be copied during query processing, for example, copying query arguments like IDs or predicates to the DPUs for selection, or when obtaining the processed results from the DPUs. Therefore, we require three additional operators for host-to-DPU and DPU-to-host data transfer and the actual launch of the DPU program:

- TransferPIM: transfers data from the host to all or individual DPUs
- ReceivePIM: obtains the results from all or a specific DPUs
- LaunchPIM: executes the DPU program

Using these three operators in the query pipeline of the Poseidon graph database is sufficient to execute queries using the PIM technology.

b) PIM Code: Generating LLVM IR code for a PIM program is similar to the generation of IR code for the usual architectures. We make again use of the LLVM framework on which the DPU runtime library is based. Again we abstract the processing into several operators. For the current implementation, we are only supporting two additional operators which make use of PIM using the UPMEM DIMMs: `NodeScanDPU` and `FilterDPU`.

Both operators require the data to be transferred before their execution, i.e., the table chunks and the appropriate arguments like labels and filter predicates. The `NodeScanDPU` operator iterates over the given set of chunks containing the nodes and filters the nodes by the given label. Further, it writes the results in a bit vector according to the position of the found nodes in the chunk. The host application obtains the found nodes by accessing their position according to the flipped positions in the bit vector. `FilterDPU` filters the properties of a node by a given predicate. The properties have to be transferred to the DPUs beforehand. The arguments for this operator are the id of the node, the label code for the property, and the value for comparison. A boolean result is written at the end of the execution.

For both operators, we generate LLVM IR similar to the appropriate operators for the non-PIM mode. Additionally, we optimize the code using the same optimization as used for the -O3 optimization.

C. Adaptive Switching

Whenever a DBMS tries to compile queries into efficient code one problem is the compilation time. Especially when compiling short running queries the actual compilation time may be much higher than the processing time of the query which decreases the resulting performance of query processing. We compile the query for two different platforms: the host and the DPUs. As the compilation for the host and DPUs needs time, we start the processing using the interpretation mode. In the meantime, the query will be compiled for the different

platforms in a background thread. As soon as the compilation is complete, the processing switches to the compiled machine code for the host. For the actual switching, we make use of the Morsel-driven parallelism of the query engine of Poseidon. Each chunk will be assigned a task which will be pushed into a task pool. Each task contains a static function pointer to the function that executes the query. Initially, it is set to the interpreter that triggers the query execution in the interpretation mode using AOT-compiled code. When switching modes, the function pointer will be set to the new compiled function, executing the compiled query code. The switch takes effect when the next task is pulled from the task pool.

V. Evaluation

A. System & Workload

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.

For the workload, we use the Social Network Benchmark (SNB) dataset from the Linked Data Benchmark Council (LDBC) and the Interactive Short Read queries 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 first three Interactive Short Read queries, as the provided approach is only able to generate code for such queries. In the future, we plan to extend our system to support all the provided queries. Nevertheless, these queries are suitable to show the efficiency of the provided approach as they cover the code generation for the basic graph query operators.

B. Benchmarks

In the following benchmarks, we compare the PIM and Adaptive PIM execution modes against the baseline which does not use the PIM technology.

1) Compilation Time: Figure 3 shows the compilation times for the queries Q1-Q3 for the host and the appropriate part of the queries for the DPUs. The benchmark shows directly the significant difference between the two compilation processes. While compiling the whole query into the target X86 machine code takes several msecs, compiling the DPU program is done in less than a millisecond. The reason for this is the small size of the DPU program. The DPU program consists of only one operator, the actual node scan and filtering by the label, and therefore, of only a few instructions. The small complexity of such programs results in fast compilation.

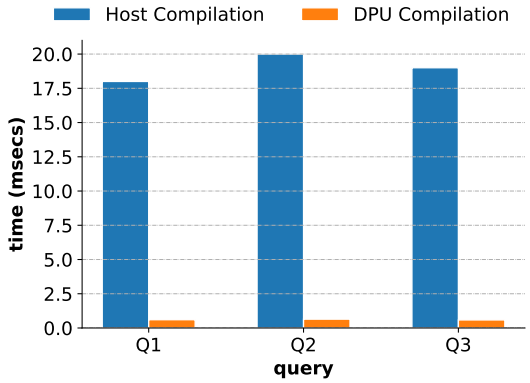

Fig. 3. Comparison of Compilation between Host and PIM Code.

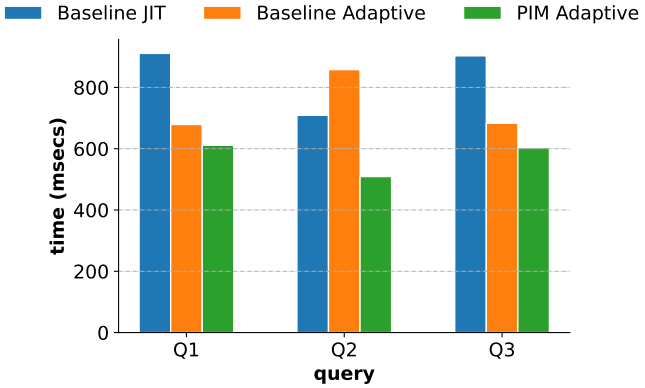

Fig. 4. Execution time of the queries in different execution modes.

2) Adaptive PIM Mode: Figure 4 shows the execution times of the queries Q1-Q3, executed in different execution modes. The Baseline JIT is the JIT compilation mode, where the query processing is waiting for the compilation process to be complete. Baseline Adaptive shows the execution times for the adaptive execution mode where the interpretation is started while compiling the query in the background and switched after the compilation is complete. PIM adaptive compiles the query into the two programs and starts in the same way with the interpretation. The adaptive PIM mode outperforms the other execution modes. This is due to the high level of parallelism using the 24 Threads on all 510 DPUs, while the other modes can only make use of the 32 CPU threads.

VI. Conclusion & Outlook

PIM is a modern technology that is expected to improve the performance of DBMS. However, modern DBMS needs to adopt the new characteristics and programming models of this technology to support its full potential of it. This work has shown an approach to integrate the technology into the query processing of DBMSs. Further, with adaptive query compilation for PIM, this work has shown that it is a reliable technique to enhance the query processing of DBMS and is suitable to adopt new technologies like PIM into the query

processing. Using this approach can improve the execution of certain queries.

For future work, as we expect that this approach can improve the execution times of all types of queries, we plan to extend the support for more operators and types of queries. Further, the placement of PIM or non-PIM operators can improve the query processing and compilation approach. There can be cases, where the compilation of a PIM program is not necessary, for example, when the data is not available or transferred to the DPUs.

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)".

References

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

[2] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.

[3] C. Freedman, E. Ismert, and P. Larson, "Compilation in the microsoft SQL server hekaton engine," *IEEE Data Eng. Bull.*, vol. 37, no. 1, pp. 22–30, 2014.

[4] T. Neumann and V. Leis, "Compiling database queries into machine code," *IEEE Data Eng. Bull.*, vol. 37, no. 1, pp. 3–11, 2014.

[5] A. Kohn, V. Leis, and T. Neumann, "Adaptive execution of compiled queries," in *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, 2018, pp. 197–208.

[6] R. Y. Tahboub, G. M. Essertel, and T. Rompf, "How to architect a query compiler, revisited," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 2018, pp. 307–322.

[7] H. Funke, J. Mühlig, and J. Teubner, "Efficient generation of machine code for query compilers," in *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*. ACM, 2020, pp. 6:1–6:7.

[8] H. Pirk, O. R. Moll, M. Zaharia, and S. Madden, "Voodoo - A vector algebra for portable database performance on modern hardware," *Proc. VLDB Endow.*, vol. 9, no. 14, pp. 1707–1718, 2016.

[9] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad, "Database analytics acceleration using fpgas," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 411–420. [Online]. Available: https://doi.org/10.1145/2370816.2370874

[10] R. Polig, K. Atasu, H. Giefers, and L. Chiticariu, "Compiling text analytics queries to fpgas," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–6.

[11] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner, "Pipelined query processing in coprocessor environments," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1603–1618. [Online]. Available: https://doi.org/10.1145/3183713.3183734

[12] A. Krolik, C. Verbrugge, and L. Hendren, "r3d3: Optimized query compilation on gpus," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 277–288.

[13] J. Paul, B. He, S. Lu, and C. T. Lau, "Improving execution efficiency of just-in-time compilation based query processing on gpus," *Proc. VLDB Endow.*, vol. 14, no. 2, p. 202–214, nov 2020. [Online]. Available: https://doi.org/10.14778/3425879.3425890

[14] D. Patterson, K. Asanovic, A. Brown, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, C. Kozyrakis, D. Martin, S. Perissakis, R. Thomas, N. Treuhaft, and K. Yelick, "Intelligent ram (iram): the industrial setting, applications, and architectures," in *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, 1997, pp. 2–7.

[15] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, "The architecture of the diva processing-in-memory chip," in *Proceedings of the 16th International Conference on Supercomputing*, ser. ICS '02. New York, NY, USA: ACM, 2002, p. 14–25.

[16] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "Top-pim: Throughput-oriented programmable processing in memory," in *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '14. New York, NY, USA: ACM, 2014, p. 85–98.

[17] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu, "Lazypim: An efficient cache coherence mechanism for processing-in-memory," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 46–50, 2017.

[18] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system," *IEEE Access*, vol. 10, pp. 52 565–52 608, 2022.

[19] M. A. Jibril, A. Baumstark, P. Götze, and K.-U. Sattler, "Jit happens: Transactional graph processing in persistent memory meets just-in-time compilation," in *24th Int. Conference on Extending Database Technology (EDBT) 2021, Nicosia, Cyprus*, 2021.

[20] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann, "Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. ACM, 2014, pp. 743–754.

[21] A. Baumstark, M. A. Jibril, and K.-U. Sattler, "Adaptive query compilation in graph databases," in *2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW)*, 2021, pp. 112–119.