Advancing Database System Operators with Near-Data Processing

Sairo R. dos Santos[†‡] Francis B. Moreira[†] Tiago R. Kepe[†§] Marco A. Z. Alves[†]
[†]Department of Informatics – Federal University of Paraná – Curitiba, Brazil
[‡]Department of Exact Sciences and Information Technology – Federal Rural University of Semi-arid – Angicos, Brazil
[§]Federal Institute of Paraná – Curitiba, Brazil
Email:[†]{fbm, mazalves}@inf.ufpr.br [‡]{sairo.santos@ufersa.edu.br} [§]{tiago.kepe@ifpr.edu.br}

***Abstract*—As applications become more data-intensive, issues like von Neumann's bottleneck and the memory wall became more apparent since data movement is the main source of inefficiency in computer systems. Looking to mitigate this issue, Near-Data Processing (NDP) moves computation from the processor to the memory, thus reducing the data movement required by many data-intensive workloads. In this paper, we look to database query operators, common targets of NDP research as database systems often need to deal with large amounts of data. We investigate the migration of most time-consuming database operators to Vector-In-Memory Architecture (VIMA), a novel 3D-stacked memory-based NDP architecture. We consider the selection, projection, and bloom join database query operators, commonly used by data analytics applications, comparing VIMA to a high-performance x86 baseline. Our results show speedups of up to $8\times$ for selection, $6\times$ for projection, and $16\times$ for join while consuming up to 99% less energy. To the best of our knowledge, these results outperform the state-of-the-art for these operators on NDP platforms.**

I. Introduction

As processor speed advanced tremendously in the last several decades, the Dynamic Random Access Memory (DRAM) technology used for main memory has failed to follow this trend. DRAM experienced only a 30% improvement in data access latency between 1997 and 2017 [1], while processor speed rises 20% per year on average [2]. Since all modern computer systems follow the von Neumann architecture design, which requires all data to be moved to the processor before processing, this discrepancy causes a multitude of issues commonly known as the memory wall [3].

As big data applications become increasingly common, the memory wall is even more relevant. These applications require significant data movement within a computer system, which is onerous in both time and energy consumption [3]–[5]. Nevertheless, most current computers add a cache hierarchy close to the processor to mitigate the latency and energy consumption issues caused by data movement.

When data is fetched from memory for processing, these cache memories store it in hopes that it will be requested again soon thereafter. This assumption causes cache hierarchies to be very beneficial for applications that present data reuse patterns. Many current applications, however, behave in a more data-centric manner, presenting a streaming-like behavior in their data accesses [6]–[9]. Computer systems present sub-optimal speed and energy consumption performance for such applications, meaning the penalty of moving data from main memory to processor cannot be mitigated.

These modern applications increasingly rely on analyzing huge datasets in what has become known as the "Era of Big Data". Fischer et al. [10] point out that the term 'big data' implies that traditional computer systems are unable to deal with such volume of data in a practical fashion, which often forces researchers to consider unorthodox methods. One emerging method relies on moving computation closer to the main memory, thus disrupting the traditional logic of a computation-centric system and causing it to become data-centric [4]. This field of research is widely referred to as NDP.

Since Big Data is at the forefront of the memory wall issue, research in NDP often targets applications under the Big Data umbrella to showcase advancements. Thus, several works can be found in the literature that apply different NDP concepts and architectures to fields such as artificial intelligence, genome sequencing and computational fluid dynamics [11].

Naturally, analytical database workloads also present compelling opportunities for NDP and have also been addressed as such. Much work is found in the literature describing efforts to filter data near the memory [12], implement major database query operators for NDP hardware [13], and provide frameworks for processing database applications near-data [14].

Previous work has analyzed the performance of database operators with a data streaming behavior, which suits NDP due of its coalescing access pattern and lack of data locality. However, operators with data reuse behavior, that benefit from data caching, have been pointed out as critical for NDP [13].

In this paper, we migrate common database query operators to run on VIMA, a novel NDP architecture [15]. We analyze how such operators perform regarding execution time and energy consumption compared to implementations for a x86 system with AVX-512 extensions. Our main contributions are:

- We simulate and evaluate the performance of database operators on a NDP architecture with very large vectors.
- We migrate the bloom join database operator to an advanced NDP architecture.
- We discuss how a NDP architecture can benefit analytical

This work was partially supported by the Serrapilheira Institute (grant number Serra-1709-16621), CAPES and CNPq (Brazilian Government).

workloads dealing with large volumes of data by comparing its performance with a traditional x86 architecture.

To the best of our knowledge, this work is the first to implement and evaluate database operators on an architecture featuring very large vectors and also the first to migrate the bloom join operator to any NDP architecture.

Our simulation results show that VIMA outperforms the baseline for all database query operators we considered, with a speedup of up to 16× at the join operator used while also reducing energy consumption by up to 99%. Regarding related work, our results are superior in both reduction in execution time and energy savings when considering large input sizes.

Outline: In Section 2, we describe the NDP architecture used for our experiments, pointing out how it enables faster processing near the memory for applications dealing with large data sets and certain behaviors. In Section 3 we detail our implementations of the NDP database query operators. In Section 4, we present and discuss our results. In Section 5, we present related work, describing other NDP work aimed at database processing. Section 6 describes our conclusions.

II. Background on Near-Data Processing

The origins of Near-Data Processing (NDP) can be traced back to the 1990s, when the first few proposals of the idea surfaced [16], [17]. However, coupling processing and storage elements on the same die was considered impractical at the time, and systems still had much performance to be gained from simply allowing Moore's Law to play out. Hence, the field saw little to no advancement for several years. With the end of Dennard scaling [18] and the advent of 3D chips enabled by TSV technology [19], NDP has made a resurgence.

As dataset sizes grow and modern applications become more data-intensive [20], von Neumann's bottleneck and the memory wall become more significant issues, as data movement is the main source of inefficiency in current computer systems [21]. Hoping to improve both speed and energy efficiency, NDP proposes placing logic elements near or coupled with memory cells, thereby reducing the costs of moving data between storage and processing elements.

NDP works particularly well when applications access large amounts of data in a coalescent pattern. In this scenario, traditional cache memory hierarchies present low performance, thus increasing time and energy usage due to data movement. Meanwhile, such a coalescent access pattern also presents a high potential for memory bandwidth usage by NDP.

Figure 1 illustrates an experiment that showcases the effects of migrating an application from a traditional x86 architecture to NDP. The application performed a simple integer comparison over an array and was executed on both a traditional architecture with a 16 MB Last Level Cache (LLC) and a NDP-enabled system. Values greater than 1 indicate improvement over the baseline. The experiment varied input array size (memory footprint), iterations (repetitions over the same array), and threads used for the baseline x86 execution.

As expected, whenever the memory footprint of the application is smaller than the LLC size and iterated over the data

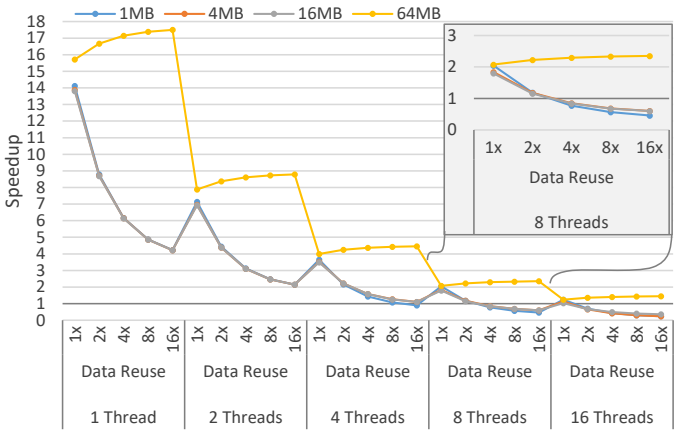

Fig. 1: NDP performance compared to traditional x86.

multiple times, the cache hierarchy of a traditional system is highly beneficial and thus preferable. However, as input dataset size grows and data-reuse opportunities become scarce, the NDP approach achieves superior performance. This can be observed in the 64 MB results, for which the speedup of the NDP increases sensibly with data reuse, as opposed to the memory footprint sizes that fit in the LLC.

Several approaches to NDP have achieved significant results. Some of the most popular are: (i) in-cell accelerators, which modify memory cells or their behavior to allow for computation [22]–[24]; (ii) in-memory accelerators, which place their logic on the same device as the memory, often using the logic layer of 3D-stacked memories to accommodate it [13], [15], [25]–[29], and; (iii) near-memory accelerators, which are devices placed in separate silicon die connected to the memory using off-chip connections [30]–[32].

Figure 2 shows a diagram of a 3D-stacked memory. These devices are composed of several Dynamic Random Access Memory (DRAM) layers vertically connected through Through-Silicon Via (TSV) and logically divided in up to 32 independent vaults, allowing for very high internal bandwidth. They also include a base layer that implements simple logic, thus enabling them to perform near-data instructions, bypassing the need for movement between memory and processor.

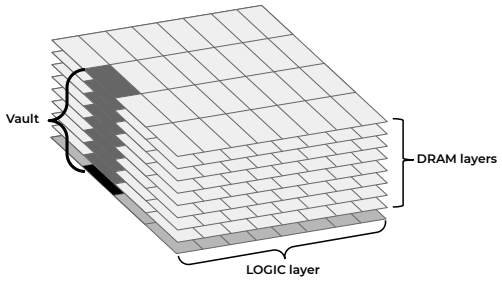

Fig. 2: Block diagram of a 3D-stacked memory.

Our target architecture is HMC Instruction Vector Extensions (HIVE) [26], a general-purpose 3D-stacked memory-based NDP architecture. This choice was motivated by the

amount of existing work in the literature that considers and extends HIVE [13], [15], [26] and its readily available simulation infrastructure. HIVE offers vector instructions that make use of the massive parallelism inside the 3D-stacked memory to fetch large vectors of data. It extends the processor Instruction Set Architecture (ISA) to add such instructions and control vector functional units near the memory, effectively delegating all front-end instruction handling to the processor.

In this paper, we consider Vector-In-Memory Architecture (VIMA) [15], an architecture that modifies HIVE by swapping its register bank with a 256 KB cache memory used to store vectorized data and provide programmers more flexibility. VIMA consists of an instruction sequencer that communicates with the host processor, a 256 KB cache memory used to store vectorized data, and a set of ARM NEON functional units used to fetch data and operate over 8 KB vectors. Figure 3 shows the architecture. The compiler inserts VIMA instructions into the application, much like happens with other vector ISA extensions, such as Advanced Vector Extensions (AVX) and NEON. VIMA instructions are handled by the processor like regular memory instructions up to the execution stage in the pipeline, at which point they are sent to the memory for near-data execution. Each instruction handles up to three 8 KB vectors, causing up to two reads and one write operation of this size. This vector size was chosen to make good use of the parallelism that is possible due to the number of independent vaults in an Hybrid Memory Cube (HMC) device.

VIMA fetches data in parallel from the several independent memory vaults of the memory, a common feature in 3D-stacked memory-based NDP solutions. The dedicated cache is checked for the data required by each instruction; execution starts immediately in case of a cache hit. Otherwise, it starts once data is successfully loaded from the memory vaults into cache. Instruction statuses are updated regarding completion or exception (similar to x86 AVX instructions) as available.

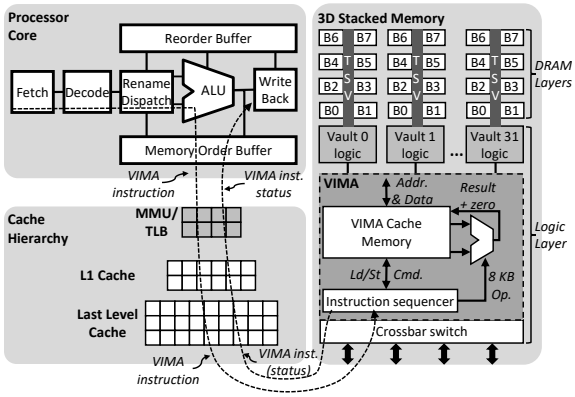

Fig. 3: 3D-memory module with VIMA architecture.

A. Intrinsics-VIMA

Intrinsics-VIMA [33] is a library used to facilitate development of programs using VIMA instructions using C/C++. Code 1 shows an example of an Intrinsics-VIMA routine.

Code 1: Intrinsics-VIMA routine example.

```
void *_vim2K_fadds(__v32f *a, __v32f *b, __v32f *c) {
    for (int i = 0; i < vima_size; ++i) {
        c[i] = a[i] + b[i];
    }
    return EXIT_SUCCESS;
}
```

Much like Intel or ARM intrinsics libraries for their respective Single Instruction Multiple Data (SIMD) extensions, it facilitates development, allowing for debugging and execution of its code on any architecture. For simulation purposes, upon trace generation each function of the library is substituted with a corresponding VIMA instruction supported by the simulation environment used for this work.

III. Near-Data Database Operators

This section describes the implementation details of the selection, projection, and join database operators. These operators are ubiquitous on analytic queries and correspond to about 70% of the execution time and memory usage of a standard database benchmark, i.e., TPC-H [13]. They represent two distinct behaviors: 1) data streaming, with the selection and projection operators, and 2) data reuse, e.g., the join operator.

A. Data Streaming

A data streaming application loads, processes, and stores each data point only once per execution. Such applications do not reuse data and thus do not benefit from the cache subsystem. Even worse, they may pollute the cache memories, degrading overall system performance.

Selection. The selection code consists of a simple comparison operation that uses an unchanging vector containing the filter value for repeated comparisons. The loop iterates over a vector with the values being considered for selection. Here we consider the late materialization scheme for this operation and thus the comparison result is stored on a bitmap of equal size for every execution of the instruction.

Code 2: VIMA selection operator code.

```
for (int i = 0; i < v_size; i += VECTOR_SIZE){
    _vim2K_isltu (filter_vec, &vector1[i], &bitmap[i]);
}
```

Projection. For the projection operator, masks generated by the selection operator are used to inform the load operation that fetches the data onto which the operator is being applied. The results are then stored in a third vector.

Code 3: VIMA projection operator code.

```
for (int i = 0; i < v_size; i += VECTOR_SIZE){
    _vim2K_ilmku (&vector2[i], &bitmap[i], &result[i]);
}
```

B. Data Reuse

Applications with some degree of data reuse benefit from cache memories by re-accessing data portions, presenting a high locality of reference. That is the case of the join operator from database systems, which merges two distinct datasets according to a join condition. It is commonly implemented

using an intermediary data structure, such as a hash table or bloom filter, that is re-accessed often.

Bloom Join. Our implementation of the join operator uses a bloom filter and is the most complex of all operators considered in this paper. We chose this specific variant of the join operation because, as far as we are aware, it has not been migrated to a NDP architecture before. It is composed of three phases: bloom filter creation, bloom filter probing, and result confirmation to generate the final result of the join operation. All bloom filter code used in our experiments is largely based on an existing implementation by Polychroniou [34], with alterations to accommodate the different architectures and chipsets available for testing.

Code 4: VIMA bloom join create operator code.

```
for (int i = 0; i < entries_size; i += VECTOR_SIZE) {
    _vim2K_ilmku (&entries[i], mask_1, bit);
    _vim2K_irmku (fun, mask_1);
    for (int j = 0; j < functions; j++){
        _vim2K_ipmtu (factors, fun, fac);
        _vim2K_ipmtu (shift_m, fun, shift_vec);
        _vim2K_imulu (bit, fac, bit);
        _vim2K_isllu (bit, shift_vec, bit);
        _vim2K_imodu (bit, bloom_filter_size, bit);
        _vim2K_isrlu (bit, shift5_vec, bit_div);
        _vim2K_iandu (bit, mask_31, bit_mod);
        _vim2K_isllu (mask_1, bit_mod, bit);
        _vim2K_iscou (bit, bit_div, bloom_filter);
        _vim2K_iaddu (fun, mask_1, fun);
    }
};
```

The VIMA implementation of the bloom filter creation algorithm shown in Code 4 consists of a loop that sets bits on an integer vector for each set of input elements. Since the creation phase of a bloom filter does not discard any elements, every data point goes through the same process. Each element undergoes multiplication and shifting steps to determine hash results, which are then used to establish the placement of bits in the filter. Filter size and number of hash functions are pre-determined, considering how many elements will be represented in the filter and an acceptable false positive rate. New elements are loaded into the vector on every iteration of the outer loop, while the inner loop iterations represent each calculated independent hash function.

The probing algorithm, shown in Code 5, iterates until all elements are either found or discarded according to the bloom filter algorithm. On every iteration, as elements are checked for whether their corresponding bit is set in the bloom filter, some of them may be discarded while others are not. This means different hash functions may be being calculated for different elements in an iteration. Consequently, every loop either resets or increments elements in a vector of indexes which inform what hash function is being calculated for each element loaded in the input vector. The results of the hash functions inform which indices of the bloom filter must be loaded for probing. A gather instruction then fetches 32-bit integers corresponding to these indices in the filter. Once the specific bit being consulted on each index is determined, it is isolated through a series of bit-wise operations. Each value determines whether its corresponding element in the working vector can be discarded.

Code 5: VIMA bloom join probe operator code.

```
int j = 0;
for (int i = 0; i <= entries_size; ) {
    _vim2K_ilmku (&entries[i], mask_k, key);
    i += j;
    _vim2K_irmku (fun, mask_k);
    _vim2K_icpyu (key, bit);
    _vim2K_ipmtu (factors, fun, fac);
    _vim2K_ipmtu (shift_m, fun, shift_vec);
    _vim2K_imulu (bit, fac, bit);
    _vim2K_isllu (bit, shift_vec, bit);
    _vim2K_imodu (bit, bloom_filter_size, bit);
    _vim2K_isrlu (bit, shift5_vec, bit_div);
    _vim2K_iandu (bit, mask_31, bit_mod);
    _vim2K_isllu (mask_1, bit_mod, bit);
    _vim2K_igtru (bloom_filter, bit_div, bit_div);
    _vim2K_iandu (bit, bit_div, bit);
    _vim2K_icmqu (bit, mask_0, mask_k);
    _vim2K_icmqu (fun, fun_max, mask_kk);

    _vim2K_idptu (mask_kk, &j);
    if (j > 0) {
        _vim2K_ismku (key, mask_kk, &output[*output_count]);
        *output_count += j;
    }

    _vim2K_iorun (mask_k, mask_kk, mask_k);
    _vim2K_idptu (mask_k, &j);
    _vim2K_iaddu (fun, mask_1, fun);
};
```

Values in the hash function vector are reset or incremented according to a mask updated with the result of the bloom filter consultations made on every loop iteration. Whenever an element is discarded, this value resets to zero. When this index is equal to the total number of hash functions being used, its corresponding element is stored as a possible positive result. This mask is also used to load new data into the working elements vector as elements are discarded. This policy guarantees no processing goes to waste, except at the very end of the process when the number of elements left is smaller than the vector size. The algorithm moves on once every element in the vector has reached one of the two possible outcomes. Elements found in the bloom filter are stored in an output vector that will be used in the confirmation phase of the join operation.

The confirmation step compares the positive results of the bloom filter probing phase with the actual elements in the dataset used to create the filter. This is necessary due to the nature of hashing algorithms, which are used for the bloom filter and may generate false-positive results. Code 6 shows the implementation.

Code 6: VIMA bloom join confirmation operator code.

```
for (int i = 0; i < positives_size; i++){
    _vim2K_imovu (positives[i], vector);
    for (int j = 0; j < entries_size; j += VECTOR_SIZE){
        count = 0;
        _vim2K_icmqu (vector, &entries[j], check);
        _vim2K_idptu (check, &count);
        if (count > 0){
            result++;
            break;
        }
    }
}
```

All Intrinsics-VIMA functions used in the code are described on Table I. We used these implementations to process

TABLE I: VIMA instruction used in the implementation of the database operators.

Instruction	Description
_vim2K_iaddu	Addition operation
_vim2K_imulu	Multiplication operation
_vim2K_imovu	Move operation
_vim2K_iandu	Bitwise AND
_vim2K_iorun	Bitwise OR
_vim2K_isllu	Bitwise shift to the left
_vim2K_isrlu	Bitwise shift to the right
_vim2K_isltu	Set if lower than
_vim2K_icmqu	If equal comparison
_vim2K_imodu	Modulo division by immediate value
_vim2K_icpyu	Copy operation
_vim2K_igtru	Gather operation
_vim2K_iscou	Scatter operation
_vim2K_ilmku	Loads data from memory into vector according to set indexes in the mask
_vim2K_ismku	Stores data from vector into memory according to set indexes in the mask
_vim2K_irmku	Sets vector positions to zero according to set indexes in the mask
_vim2K_ipmtu	Permutates elements from another vector according to indexes in the mask
_vim2K_idptu	Dot product of all elements in a vector

TABLE II: Baseline and VIMA system configuration.

OoO Execution Cores 1 core @ 2.0 GHz, 32 nm; Power: 6W/core; 6-wide issue; Buffers: 40-entry fetch, 128-entry decode; 168-entry ROB; MOB entries: 72-read, 56-write; 2-load, 1-store units (1-1 cycle); 4-alu, 1-mul. and 1-div. int. units (1-3-32 cycle); 2-alu, 2-mul. and 1-div. fp. units (3-5-10 cycle); 1 branch per fetch; Branch predictor: Two-level GAs. 4096 entry BTB;
L1 Inst. Cache 64 KB, 8-way, 4-cycle; 64 B line; LRU policy; Dynamic energy: 194pJ per line access; Static power: 30mW;
L1 Data Cache 64 KB, 8-way, 6-cycle; 64 B line; LRU policy; Dynamic energy: 194pJ per line access; Static power: 30mW;
L2 Cache 128 KB, 16-way, 34-cycle; 64 B line; LRU policy; Dynamic energy: 340pJ per line access; Static power: 130mW;
LLC Cache 16 MB, 16-way, 52-cycle; 64 B line; LRU policy; Dynamic energy: 3.01nJ per line access; Static power: 7W;
3D Stacked Mem. 32 vaults, 8 DRAM banks/vault, 256 B row buffer; 4 GB; DRAM@1666 MHz; 4-links@8 GHz; Inst. lat. 1 CPU cycle 8 B burst width at 2.5:1 core-to-bus freq. ratio; Open-row policy; DRAM: CAS, RP, RCD, RAS and CWD latency (9-9-9-24-7 cycles); Avg. energy per access: x86:10.8pJ/bit; VIMA:4.8pJ/bit; Static power 4W;
VIMA Processing Logic Operation frequency: 1 GHz; Power: 3.2W; 256 int. units: alu, mul. and div. (8-12-28 cycles for 8 KB pipelined) 256 fp. units: alu, mul. and div. (13-13-28 cycle for 8 KB pipelined); VIMA cache: 256 KB, fully assoc., 2-cycle (1-tag, 1-per data); Dynamic energy: 194pJ per line access; Static power: 134mW;

random data and generate simulation traces. The simulation environment and results are presented in the next section.

IV. Evaluation Methodology and Results

In this section we describe our methodology and results for the evaluation of our query operators for the VIMA architecture. **AVX** and **VIMA** stand for Intel AVX-512 and Vector-In-Memory Architecture (VIMA) results, respectively.

A. Methodology

Table II shows the parameter details used in our simulations. Parameters were set to be similar to Intel's Skylake microarchitecture. All simulations were run on SiNUCA [35], a validated cycle-accurate simulator. The original paper that describes SiNUCA reports only a 9% average error when compared with actual hardware performance, and thus we believe it to be appropriate for our evaluation purposes.

Test workloads used random 32-bit integer elements generated with standard C/C++ math functions. The dataset sizes chosen consider the capabilities of the architectures being compared: since NDP offers good performance when the dataset exceeds cache capacity, we ensure for every operator that at least one dataset size would surpass the x86 architecture's LLC size. The sizes used are 1, 20, 64, and 80 MB.

B. Results

Figure 4 shows the speedup (higher the better) of VIMA over AVX for selection and projection query operators. Percentages over bars refer to energy savings over baseline (higher the better). The speedup for both operators happens mainly because of the superior use of memory parallelism fetching data inside the memory. VIMA requires two operands on the instructions used for these operators, which causes two 8 KB vectors to be loaded from memory. These requests use most of 3D-memory's vault parallelism, thus achieving speedups of up to 8× for the selection operator and 6.5× for the projection operator, as pictured. For the selection operator, VIMA spends 75% less energy, while the projection operator consumes roughly half the energy used by the baseline.

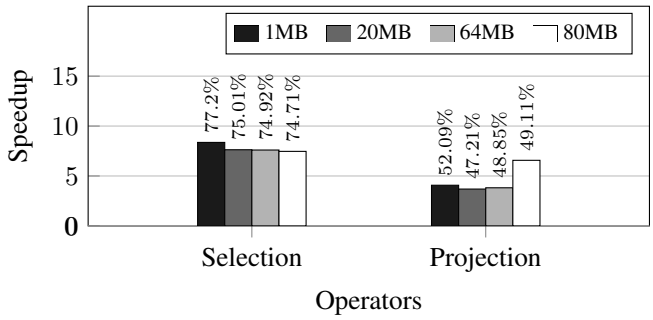

Fig. 4: Speedup over baseline for selection and projection operators, percentages indicate energy savings over baseline.

The bloom join operation used for our experiments features two columns, one being four times larger than the other. Here, dataset sizes refer to the size of the larger column between the two used by the operator. The bloom filter is created using the smaller column, while the larger one is used for probing. We devised datasets with varying selectivity to design tests that represent real-world situations and see how the architectures compare at a realistic data-join operation situation. We generated all data randomly and controlled selectivity by purposely adding elements from the smaller column into the larger according to the desired selectivity. Selectivity ranges

from 0% to 100%, varying by 10% for each test. As for the bloom filter, we used a simple hash function consisting of a multiplication step followed by a shifting step [36]. The number of hash functions varied according to the number of elements being processed so the false positive rate was kept under 1%. All multiplication and shifting factors were kept the same for both VIMA and AVX implementations. All speedup results are shown in Figure 5. Percentages over bars refer to energy savings over baseline, negative figures meaning consumption was higher than the baseline.

Data selectivity affects the execution length of each phase of the operator's execution (bloom filter creation, bloom filter probing, and confirmation), which impacts performance directly. The creation phase is defined by the hash function calculations being applied to the first column of the join operation, setting in the bloom filter vector the bits corresponding to the results. This first phase behaves the same no matter the expected results from the dataset because every element goes through all calculations regardless of its value. Nevertheless, the bloom filter probing phase is much more efficient at determining that data elements are not part of its represented dataset than otherwise. This behavior is because distinct operations are executed according to data patterns.

At the probing phase, data content directly impacts performance. Here, the hash result of elements in the second column is used to check whether a specific bit is set in the bloom filter. If any hash result for an element points to a bit that is not set, that element is confirmed a negative and can be discarded. Consequently, data selectivity determines the length of the probing and also confirmation phases. This explanation is why results for the 0% selectivity datasets show a considerable advantage for VIMA over AVX. For VIMA, each loop iteration discards up to 2048 elements, and therefore, the probing process moves fast. Meanwhile, for the 100% selectivity dataset, all elements go through all hash computations, meaning the probing phase lasts very long. Here, VIMA is also aided by its dedicated cache, which is able to house the vectors used for the hash function computations in the probing phase which are repeatedly reused.

The confirmation phase is another reason why the 0% selectivity dataset has such superior results. During this phase, positive results from the probing phase are checked against the data used to create the bloom filter. Since each such element must be checked against all elements in this dataset, this phase can be very time-consuming. In datasets with positive results, many elements pass the probing phase and go through the confirmation phase, representing a more significant portion of the execution time with each increasing stride in selectivity. However, with the all negative dataset, the probing phase yields very few positives, most of these being false positives, causing only a very short confirmation phase. Since the probing phase is highly efficient on VIMA, it represents most of the execution time in these cases, explaining the sharply superior 0% selectivity result. These gains decrease with selectivity, with the confirmation phase representing a larger portion of the entire execution time and the reuse capabilities of each architecture start to influence overall performance.

Results are also influenced by the smaller column size, which is repeatedly accessed in full for the confirmation phase. Since this column is one fourth of the dataset size, its size is 256 KB, 5 MB, 16 MB and 20 MB for the datasets considered here. This means that, for all datasets but the largest one, this data can be fully stored in the baseline architecture's LLC.

The benefits of the LLC are clear on the results for the 1 MB dataset. While VIMA outperforms AVX at low selectivity levels, the advantage disappears as selectivity rises, which shows how much the baseline benefits from the faster access provided by the its cache hierarchy. Energy consumption follows the same pattern, with VIMA using much more energy as it reloads data from the main memory repeatedly while this data is kept in the baseline's LLC, translating into a significant advantage maintained from 20% selectivity onward.

Looking at the results for the 20 MB and 64 MB datasets, VIMA remains advantageous even with growing selectivity due to the effect of its large vectors. As the amount of data that must be considered for the confirmation phase grows (original data column and positives from the probing phase), VIMA's ability to load and process large vectors at once starts to surpass the effect of AVX's cache hierarchy.

For more extensive datasets (e.g. 80 MB), VIMA offers superior performance in both metrics. For example, when looking at the 80 MB results, we observe that VIMA outperforms AVX by $16\times$ at 0% selectivity while consuming over 99% less energy. Here, the data through which the application must iterate to confirm probing phase results is larger than the LLC in the baseline architecture, which means AVX no longer benefits from it and is forced to reload data directly from the main memory. Since at this dataset size even the 0% selectivity workload still yields a few thousand false positive results from its probing phase, VIMA's large vectors coupled with the baseline's fetching inefficiency results in this considerable performance improvement. As selectivity grows, the confirmation phase becomes longer, and this advantage declines, but VIMA continues to outperform AVX by at least $3.5\times$ at 100% selectivity while consuming 54% less energy.

V. Related Work

In the late 1990s, researchers began considering the performance of database systems on modern computers. For instance, Ailamaki et al. [37] and Boncz et al. [38] discuss how the memory wall impacted database systems as processors evolved much faster than main memory technology. This led to a tendency for database software to increasingly consider and fit the hardware architecture on which they were run, by redesigning all database system elements with a modern hardware architecture in mind. This redesign led to techniques such as columnar data storage, bulk query relational algebra, cache-conscious algorithms, and automatic optimization strategies to finely tune behavior to take advantage of the hardware [39].

Such adaptations, however, focus only on using existing hardware to the best of its ability, making no attempt to reduce

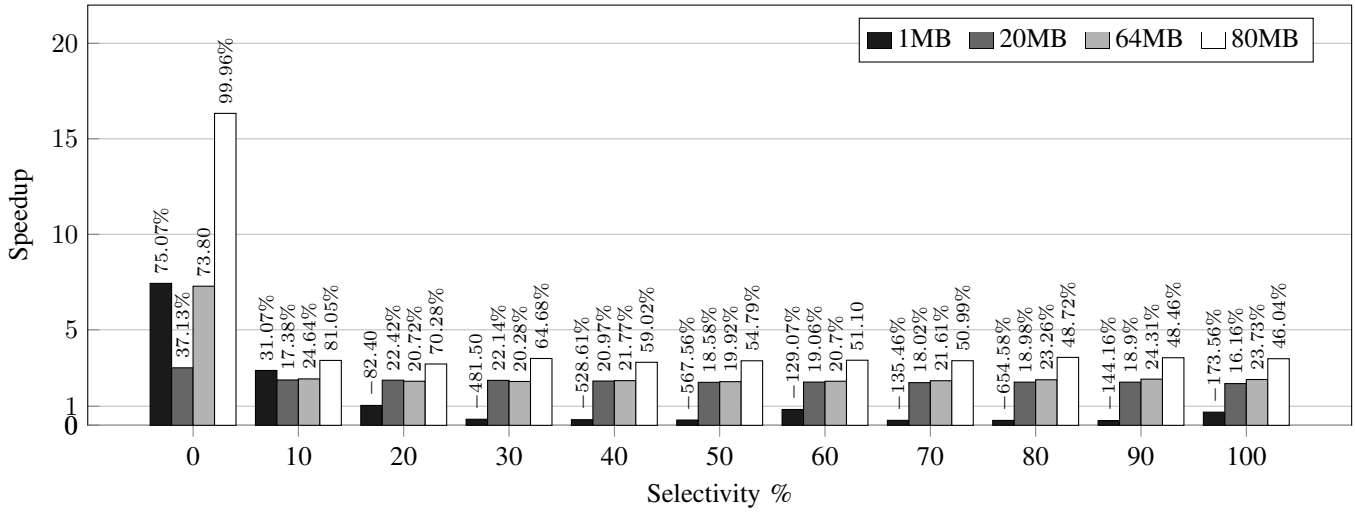

Fig. 5: Speedup over baseline for the bloom join operator with varying selectivity rates, figures over 1 indicate speedup. Percentages over the bars indicate energy savings over baseline, negative values indicate energy consumption exceeded baseline.

data movement across the memory hierarchy. On the other hand, as NDP technology becomes a reality, strategies to move database computations closer to the data start to look attractive, especially as data set sizes to continue to grow [40].

JAFAR [40], [41], for instance, is a near-data accelerator that considers column-store databases and implements the selection operator, providing a speed improvement of up to $9\times$ for selects. This design places the accelerator close to a DRAM chip, being able to receive data directly from it and filter this data according to a comparison value and predicate type. This way, it builds a bit-mask that indicates which tuples of a relational database table passed the filter, which can then be used for further query computation. While JAFAR shows promising results for the selection operator, its design cannot be easily extended to support the execution of other operators.

Biscuit [14] uses a different approach, proposing NDP on Solid State Drive (SSD) disks. The authors point out that most existing NDP research focuses mainly on providing proof-of-concept, failing to consider or propose features that would make NDP adoption practical. They then propose an entire framework implementing an expressive programming model, dynamic task loading, support for high-level programming languages, and multi-core and threading capabilities. Furthermore, the authors fully port the MySQL database engine to their framework and report a speedup of up to $15.4\times$ for query execution and a $3.6\times$ reduction of execution time for all TPC-H queries. Biscuit offers comprehensive experiments and results, but its solution for database processing requires extensive modifications of SSD devices through the addition of complete cores. However, our work focuses on in-memory execution. Meanwhile, it can easily be extended to feature all database operators using a much simpler hardware architecture to achieve similar improvements in performance.

HIPE [27] considers the HMC and proposes adding support for predicated instructions inside an HMC so it is able to perform database queries. HIPE is also based on HIVE [26], which proposed adding vector functional units and a register bank to the logic layer of an HMC chip. HIPE adds architectural predication support to this design, thus enabling control-flow dependencies to be solved in the memory. With this design, the authors implement a database selection operator and report results $6.46\times$ superior to an x86 architecture while being 5% more energy-efficient than the state-of-the-art. Although we use a similar architecture and simulation infrastructure, as is evidenced by the difference between our energy consumption results, their approach is not as efficient. The authors also only experiment with the selection operator.

Kepe et al. [13] implement five database query operators (selection, projection, aggregation, sorting, and join) to evaluate how an NDP architecture performs against a state-of-the-art x86 architecture. The architecture used for the experiments is HIVE [26], using a similar infrastructure used in the present work. Considering a baseline x86 architecture with Intel AVX-512 extensions, the authors report: the selection operator runs at least $3\times$ faster, regardless of input size, while consuming 45% less energy; the projection operator runs $7\times$ faster if the dataset fits the baseline's last level cache otherwise it is $10\times$ faster while reducing energy use by $3\times$; performance for the join operator varies according to implementation (nested loop, hash, and sort-merge), but the NDP implementation is significantly more energy-efficient in all cases; the aggregation operator performs moderately worse at both execution time and energy consumption. In comparison to our work, although the authors feature a wider variety of database operators and use a similar architecture, they do not consider vector sizes that fully utilize the parallelism opportunities possible with such an architecture. We report superior execution time results for the selection and join operators (though this work implements

different versions of the join operator) and superior energy savings across all operators when considering large input sizes. Here, we highlight our join results according to selectivity, something this work also investigates. Our results are superior in both execution time and energy savings. The authors report speed improvements ranging from $1.6\times$ to $3\times$ with energy savings between 5 and 70%, while our results range from $3.5\times$ to $16\times$ in speed improvement with energy savings ranging from 46% to 99% for the largest input size considered.

VI. Conclusions and Final Considerations

With the growing relevancy of analytics applications that process vast sets of data, NDP emerges as a solution for the memory wall problem. In this paper, we migrate execution of standard database query operators to a near-data architecture.

Our approach improves execution time up to $8\times$ for the selection operator, $6\times$ for the projection operator, and $16\times$ for the join operator used. We also achieved energy savings of 75% for the selection, 50% for the projection, and up to 99% for the join operator. These results are superior to the state-of-the-art and consider a simpler and more programmer-friendly architecture. To the best of our knowledge, this work is the first to implement and evaluate database operators on an architecture featuring very large vectors and also the first to migrate the bloom join operator to any NDP architecture.

Future work includes migrating other database operators and implementations of the join operator, as some implementations can suit certain situations better. This should enable us to evaluate this NDP approach with the entire TPC-H benchmark and better assess how NDP may benefit database processing.

References

[1] K. K. Chang, "Understanding and improving the latency of dram-based memory systems," Ph.D. dissertation, Carnegie Mellon University, 2017.

[2] J. Preshing, "A look back at single-threaded cpu performance," *Preshing on Programming Blog, February*, vol. 8, pp. 821–828, 2012.

[3] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, 1995.

[4] R. Balasubramonian *et al.*, "Near-data processing: Insights from a micro-46 workshop," *IEEE Micro*, vol. 34, 2014.

[5] M. Hashemi *et al.*, "Accelerating dependent cache misses with an enhanced memory controller," in *Int. Symp. on Computer Architecture*, 2016.

[6] P. Xie *et al.*, "V-pim: An analytical overhead model for processing-in-memory architectures," in *Non-Volatile Memory Systems and Applications Symp.*, 2018.

[7] M. K. Qureshi *et al.*, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Computer Architecture News*, vol. 35, 2007.

[8] ——, "Line distillation: Increasing cache capacity by filtering unused words in cache lines," in *Int. Symp. on High Performance Computer Architecture*, 2007.

[9] A. Boroumand *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2018.

[10] D. Fisher *et al.*, "Interactions with big data analytics," *interactions*, vol. 19, no. 3, pp. 50–59, 2012.

[11] P. C. Santos *et al.*, "Survey on near-data processing: Applications and architectures," *Journal of Integrated Circuits and Systems*, vol. 16, no. 2, pp. 1–17, 2021.

[12] D. G. Tomé *et al.*, "Near-data filters: Taking another brick from the memory wall." in *ADMS@ VLDB*, 2018, pp. 42–50.

[13] T. R. Kepe *et al.*, "Database processing-in-memory: An experimental study," in *Proc. VLDB Endow.*, 2019.

[14] B. Gu *et al.*, "Biscuit: A framework for near-data processing of big data workloads," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 153–165, 2016.

[15] A. S. Cordeiro *et al.*, "Machine learning migration for efficient near-data processing," in *Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP)*, 2021.

[16] D. Patterson *et al.*, "A case for intelligent ram," *IEEE Micro*, vol. 17, 1997.

[17] D. G. Elliott *et al.*, "Computational ram: Implementing processors in memory," *IEEE Design & Test of Computers*, vol. 16, 1999.

[18] H. Esmaeilzadeh *et al.*, "Dark silicon and the end of multicore scaling," in *Int. Symp. on Computer Architecture*, 2011.

[19] J. V. Olmen *et al.*, "3D stacked IC demonstration using a through silicon via first approach," in *Int. Electron Devices Meeting*, 2008.

[20] A. Labrinidis and H. V. Jagadish, "Challenges and opportunities with big data," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 2032–2033, 2012.

[21] D. P. Zhang *et al.*, "A new perspective on processing-in-memory architecture design," in *SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2013.

[22] S. Angizi *et al.*, "Pim-assembler: A processing-in-memory platform for genome assembly," in *Design Automation Conf. (DAC)*, 2020.

[23] S. Gupta *et al.*, "Rapid: A reram processing in-memory architecture for dna sequence alignment," in *Int. Symp. on Low Power Electronics and Design (ISLPED)*, 2019.

[24] Y. Huang *et al.*, "A heterogeneous pim hardware-software co-design for energy-efficient graph processing," in *Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2020.

[25] M. A. Alves *et al.*, "Saving memory movements through vector processing in the dram," in *Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2015.

[26] M. A. Z. Alves *et al.*, "Large vector extensions inside the hmc," in *Design, Automation & Test in Europe Conf.*, 2016.

[27] D. G. Tomé *et al.*, "Hipe: Hmc instruction predication extension applied on database processing," in *Design, Automation & Test in Europe Conf.*, 2018.

[28] G. F. Oliveira *et al.*, "Nim: An hmc-based machine for neuron computation," in *Int. Symp. on Applied Reconfigurable Computing*, 2017.

[29] P. C. Santos *et al.*, "Operand size reconfiguration for big data processing in memory," in *Design, Automation & Test in Europe Conf.*, 2017.

[30] M. Alian *et al.*, "Application-transparent near-memory processing architecture with memory channel network," in *Int. Symp. on Microarchitecture (MICRO)*, 2018.

[31] M. Drumond *et al.*, "Algorithm/architecture co-design for near-memory processing," *Operating Systems Review*, 2018.

[32] S. H. Pugsley *et al.*, "NDC: analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads," in *Int. Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[33] A. S. Cordeiro *et al.*, "Intrinsics-hmc: An automatic trace generator for simulations of processing-in-memory instructions," *Simp. em Sistemas Computacionais de Alto Desempenho*, 2017.

[34] O. Polychroniou, *Analytical Query Execution Optimized for all Layers of Modern Hardware*. Columbia University, 2018.

[35] M. A. Z. Alves *et al.*, "Sinuca: A validated micro-architecture simulator," in *Int. Conf. on High Performance Computing and Communications*, 2015.

[36] M. Dietzfelbinger *et al.*, "A reliable randomized algorithm for the closest-pair problem," *Journal of Algorithms*, vol. 25, no. 1, pp. 19–51, 1997.

[37] A. Ailamaki *et al.*, "Dbmss on a modern processor: Where does time go?" in *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, no. CONF. Citeseer, 1999, pp. 266–277.

[38] P. A. Boncz *et al.*, "Database architecture optimized for the new bottleneck: Memory access," in *VLDB*, vol. 99, 1999, pp. 54–65.

[39] ——, "Breaking the memory wall in monetdb," *Communications of the ACM*, vol. 51, no. 12, pp. 77–85, 2008.

[40] S. L. Xi *et al.*, "Beyond the wall: Near-data processing for databases," in *Proceedings of the 11th International Workshop on Data Management on New Hardware*, 2015, pp. 1–10.

[41] A. Augusta and S. Idreos, "Jafar: Near-data processing for databases," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 2069–2070.