

Improved Computation of Database Operators via Vector Processing Near-Data

Sairo Santos
Federal Rural University of the Semi-arid
Angicos, Brazil
sairo.santos@ufersa.edu.br

Tiago R. Kepe
Federal Institute of Paraná
Curitiba, Brazil
tiago.kepe@ifpr.edu.br

Marco A. Z. Alves
Federal University of Paraná
Curitiba, Brazil
mazalves@inf.ufpr.br

Abstract—Data-centric applications are increasingly more common, causing issues brought on by the discrepancy between processor and memory technologies to be increasingly more apparent. Near-Data Processing (NDP) is an approach to mitigate this issue. It proposes moving some of the computation close to the memory, thus allowing for reduced data movement and aiding data-intensive workloads. Analytical database queries are very commonly used in NDP research due to their intrinsic usage of very large volumes of data. In this paper, we investigate the migration of most time-consuming database operators to 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 Vector-In-Memory Architecture (VIMA) to a high-performance x86 baseline. We pitch VIMA against both a single-thread baseline and a modern 16-thread x86 system to evaluate its performance. Against a single-thread baseline, our experiments show that VIMA is able to speed up execution by up to $5\times$ for selection, $2.5\times$ for projection, and $16\times$ for join while consuming up to 99% less energy. When considering a multi-thread baseline, VIMA matches the execution time performance even at the largest dataset sizes considered. In comparison to existing state-of-the-art NDP platforms, we find that our approach achieves superior performance for these operators.

Index Terms—near-data processing, high performance computing, database operators

I. INTRODUCTION

After several decades of precipitous advancements in processor speed, main memory technology, Dynamic Random Access Memory (DRAM), has lagged behind significantly, failing to progress at the same rate. The latency in access of data stored in DRAMs was only reduced by 30% between 1997 and 2017 [1]. Meanwhile, processors continue to advance in speed at an average rate of 20% per year [2]. This disparity poses an issue to all modern computers: they must move all data from the memory to the processor for processing, as required by the von Neumann architecture design. The discrepancy between processor and memory speed causes a myriad issues, largely referred to as the memory wall [3].

The memory wall is even more relevant currently, as interest in big-data applications is ever-increasing. Such applications deal with enormous volumes of data, thus requiring a lot of data movement for processing, which is onerous in both time and energy consumption [3]–[5].

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

Cache hierarchies placed next to the processing cores, which are now ubiquitous in all modern computer systems, are the main mitigation strategy for the problems caused by data movement. Cache memories are used to store data that gets fetched from the memory, assuming it might be requested by the application again soon, at which point they can be provided much faster. Whenever the data access patterns of an application involve reusing the same data in close succession, this assumption greatly benefits the system, as the data is now available close to the processor and does not require fetching from the main memory again. However, it is increasingly common for applications to not present such locality of reference, accessing data in a streaming-like pattern [6]–[9]. For this class of applications, current modern computer systems are unable to mitigate the penalty of accessing the main memory to fetch required data. They will then provide poor execution time and energy consumption performance when running such applications.

The era of Big Data is mainly characterized by the increasing relevancy of applications that fit this description, as they primarily analyze large datasets. In fact, according to some authors [10], the 'big data' term itself carries the implication that such applications are ill-equipped to handle such volumes of data. Such behaviors regarding data access cause researchers to consider unorthodox methods. One such method consists in implementing processing near the data, e.g. close to the main memory, to avoid systems being forced to move data all the way to the processor whenever beneficial. Such approach enables systems to better suit applications that are data-centric, as opposed to applications that are computation-centric [4]. The field of research that studies and proposes architectures that fit that description is known as Near-Data Processing (NDP).

NDP research often uses big-data applications to evaluate architecture proposals and showcase results, as they expose the memory wall issue. Thus, several works in the literature that apply different NDP concepts and architectures to fields such as artificial intelligence, genome sequencing, and computational fluid dynamics [11].

One such field is analytical database queries, which deal with very large datasets by design and, thus, are also very commonly targeted by NDP research. 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].

Most existing work focused on analytical database applications have focused on data streaming operators, such as selection and projection, which suit NDP well due to their coalescent access patterns and low data reuse. However, operators with data reuse behavior that benefit from data caching are also critical for NDP [13].

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

- We implement near-data versions of common database operators and provide a simulation-based performance evaluation of such implementations.
- We implement a near-data bloom join database operator and provide a simulation-based performance evaluation of such implementations.
- We discuss the benefits of near-data processing when running analytical workloads over large datasets, comparing performance against a modern x86 system.
- We compare the performance of the NDP architecture with that of a modern 16-thread x86 traditional architecture.
- We simulate and evaluate the performance of database operators on a near-data multithreaded context.

Our work is, as far as we are aware, the first to use a near-data architecture based on large vectors to implement and evaluate performance of database operators, migrate the bloom join operator near-data and also the first to consider a multithreaded near-data processing environment.

In our simulation environment, VIMA is able to outperform the x86 baseline for all database query operators, considering both a single-thread x86 baseline and a 16-thread x86 baseline. It speeds up execution by up to $16\times$ for the join operator considering a single-thread baseline, while consuming up to 99% less energy. Our results are superior to the related work in reducing execution time and saving energy 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 a set of 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

Near-Data Processing (NDP) reduces data access times and energy consumption in data-intensive tasks by placing processing capabilities close to the data, extending the traditional von

Neumann architecture model. In an NDP architecture, processing occurs near the memory, eliminating the need for excessive data movement between memory and the processor. Rather than transferring large amounts of data to the processor, only relevant instructions are offloaded for near-data execution. This approach offers significant reductions in execution time and energy consumption for data-centric applications, effectively utilizing the parallelism and internal bandwidth of the main memory.

While the first few NDP proposals first surfaced back in the last 1990s [16], [17], implementing processing and storage elements on the same hardware was not feasible at the time and, since systems still has much performance to gain from allowing Moore’s law to follow its course, the idea was not widely pursued and thus saw very little advancement for many years. However, as Dennard scaling began to show signs of exhaustion [18] and Through-Silicon Via (TSV) technology [19] became viable, yielding the first few 3D-stacked memories, NDP has again sparked the interest of researchers.

The era of Big Data has meant that applications are increasingly more data-centric [20], which means the von Neumann bottleneck and the memory wall are ever more relevant, seeing as the most significant source of inefficiency and energy consumption in modern systems is data movement [21]. In hopes of mitigating the impact of such inefficiency in both execution time and energy consumption, the NDP approach brings computation to the data by placing processing elements near the memory, thus reducing most costs associated with moving data across the system.

In general, NDP is better suited to applications that access large volumes of data in a coalescent fashion, meaning they do not benefit from traditional cache hierarchies. Considering a traditional system, this means such programs access the main memory for nearly every data access, thus experiencing longer execution times and increased energy consumption due to this constant data movement between memory and processor. On the other hand, such a situation is oftentimes ideal for near-data execution.

A simple experiment can illustrate the effects of NDP execution of a data-hungry application in comparison to a traditional system. Figure 1 shows the results of an experiment that compares the performance of a traditional system with a 16 MB last level cache with that of a NDP architecture. Both systems run an application that performs a simple integer comparison over a large vector. Observed variables were input size (memory footprint), iterations (repetitions over the same data) and number of baseline threads.

Whenever the input data fits the last level cache of the baseline system, as one would expect, execution on the baseline system is aided by the cache hierarchy and is thus preferable to the near-data option. However, when input data overwhelms the last level cache, data reuse is no longer possible, meaning the baseline is forced to reload data for repeated iterations of the application. From this point on, near-data execution achieves better performance and is, therefore, preferable. This improvement can be seen in Figure 1 when

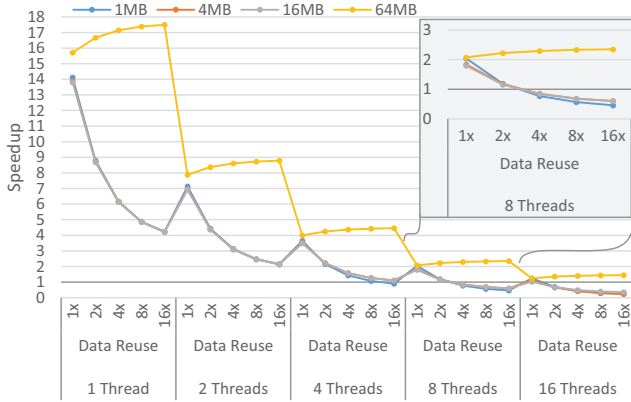


Fig. 1. NDP performance compared to traditional x86.

observing the 64 MB results, where the improvement the NDP alternative offers increases sensibly with the number of application iterations on the baseline, as opposed to the other data sizes, which fit in the last level cache.

Some of the most common approaches to NDP are: (i) in-cell accelerators, which modify the behavior of memory cells to enable in-memory processing [22]–[24]; (ii) in-memory accelerators, which add logic to memory devices, oftentimes to the logic later of 3D-stacked memories [13], [15], [25]–[29], and; (iii) near-memory accelerators, which place separate devices close to the memory using off-chip connections [30]–[32].

Figure 2 shows a diagram of a 3D-stacked memory. Such devices are made possible by TSV connection technology, which allows for vertical integration of Dynamic Random Access Memory (DRAM) layers. Memory space is split into up to 32 logically independent vaults, allowing for high internal bandwidth. The device also includes an underlying logic layer where processing elements can be placed, thus enabling near-data computation and bypassing the need for data movement.

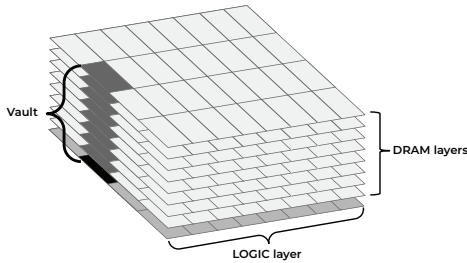


Fig. 2. Block diagram of a 3D-stacked memory.

For our experiments, we consider HMC Instruction Vector Extensions (HIVE) [26], a 3D-stacked memory-based NDP architecture. HIVE is a general-purpose architecture with a readily available simulation environment and several existing works in the literature documenting and extending its capabilities [13], [15], [27]. It uses large vector instructions that

leverage the large internal bandwidth of 3D-stacked memories for improved performance, extending the processor ISA with its own specific instructions for simplicity of front-end instruction handling. We further extend this architecture by adding a dedicated near-data cache memory to the architecture, which we use to store and reuse vectorized data. Such storage is added in place of the register bank used in the original research paper that describes HIVE [26]. The resulting design is called Vector-In-Architecture (VIMA) [15].

VIMA communicates with the host processor through an instruction sequencer, which emits memory requests to the memory and handles vector operands. All data is stored in a 256 KB dedicated cache and processed with a set of 512-bit vector units used to operate over 8 KB vectors. Figure 3 shows the architecture.

VIMA instructions, like other NDP proposals such as Intel AVX or ARM NEON, extend the ISA of host processors. These instructions work like regular memory instructions, offloading to a near-memory device during execution. VIMA assumes 8 KB vector operands, causing two data loads and one data store operation of 8 KB to main memory. This vector size suits a 3D-stacked memory with 32 vaults and a 256 B row buffer, where 8 KB operands request 256 B from each vault.

Improved parallelism in data access is one of the main features of 3D-stacked memories, which is another reason why such devices are so well suited to NDP. Thus, much like many other NDP solutions, VIMA fetches data in parallel from the several independent vaults, taking advantage of both the internal parallelism and increased bandwidth of the 3D-stacked memory. All data is stored in the dedicated cache memory, which is checked for existing data before load and store requests are sent to the main memory. Data is only fetched from the memory if it is not yet stored in the cache. Instruction execution starts once all operand data is successfully stored in the cache. Whenever an instruction finishes execution or causes an exception, its status is updated accordingly.

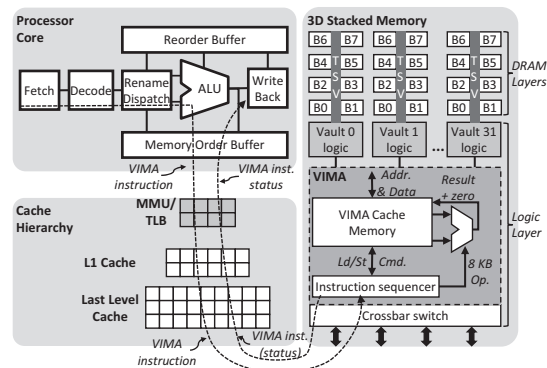


Fig. 3. 3D-memory module with VIMA architecture.

A. Intrinsic-VIMA

We provide a library that can be used to core and debug applications using VIMA instruction in C/C++, Intrinsic-

VIMA. Code 1 shows an example of an Intrinsic-VIMA routine.

```
Code 1. Intrinsic-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;
}
```

The library functions similarly to the libraries provided by Intel or ARM to access their own Single Instruction Multiple Data (SIMD) extensions, meaning the function calls are substituted for their associated SIMD instructions by the compiler. We use this for simulation purposes, where each function call is swapped for its corresponding VIMA instruction during trace generation for our simulation environment.

III. NEAR-DATA DATABASE OPERATORS

Here we describe the three database operators selected for our experiments: selection, projection and join. These specific operators were chosen because of how ubiquitous they are on analytic queries, accounting for about 70% of the total execution time of TPC-H, a standard database benchmark [13]. The three operators display distinct two behaviors we aim to investigate: (i) the selection and projection operators represent a data streaming behavior, and (ii) the join operator represent a data reuse behavior.

TABLE I
VIMA INSTRUCTION USED IN THE IMPLEMENTATION OF THE DATABASE OPERATORS.

Instruction	Description
<code>_vim2K_iaddu</code>	Addition operation
<code>_vim2K_imulu</code>	Multiplication operation
<code>_vim2K_imovu</code>	Move operation
<code>_vim2K_iandu</code>	Bitwise AND
<code>_vim2K_iorun</code>	Bitwise OR
<code>_vim2K_isllu</code>	Bitwise shift to the left
<code>_vim2K_isrly</code>	Bitwise shift to the right
<code>_vim2K_isltu</code>	Set if lower than
<code>_vim2K_icmqu</code>	If equal comparison
<code>_vim2K_imodu</code>	Modulo division by immediate value
<code>_vim2K_icpyu</code>	Copy operation
<code>_vim2K_igtru</code>	Gather operation
<code>_vim2K_iscou</code>	Scatter operation
<code>_vim2K_ilmku</code>	Loads data from memory into vector according to set indices in the mask
<code>_vim2K_ismku</code>	Stores data from vector into memory according to set indices in the mask
<code>_vim2K_irmku</code>	Sets vector positions to zero according to set indices in the mask
<code>_vim2K_ipmtu</code>	Permutates elements from another vector according to indices in the mask
<code>_vim2K_idptu</code>	Dot product of all elements in a vector

Table I lists all the functions used in our code. Our experiments considered random data, and the code was used to generate simulation traces within the simulation environment. Results are presented in the next section.

A. Data Streaming

Data streaming applications only load and process each data point once per execution, thus not reusing data or benefiting from the cache hierarchy of a system. Instead, since all data is loaded to the processor, it gets stored in the cache to never be reused, thus polluting the cache memory without providing any benefit.

Selection. For the selection operator, the Vector-In-Memory Architecture (VIMA) code performs a simple comparison between a constant vector containing a filter and a second vector into which all input data is loaded. The application iterates over the input data and stores results considering a late materialization model, meaning the result of the operation is a bitmap the same length as the input dataset.

```
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. The projection operator considers a bitmap mask such as the one created by the selection operator. It is used to inform a conditional loading operation that fetches and stores data from the memory according to the positions of the bits set in the mask. The results are stored in a separate 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

Cache memories benefit applications that present some degree of data reuse, e.g. locality of reference. The database join operator, which merges two datasets according to a specific condition, behaves as such. It commonly relies on an intermediary data structure to keep track of join elements, and this data structure is repeatedly accessed for checking and updating.

Bloom Join. The join operator has many different implementations. We chose to implement the bloom filter-based implementation, e.g. bloom join, because it is not commonly implemented near-data. The bloom join has three distinct phases: (i) creation, when the bloom filter data structure is set, usually using the smaller of the two datasets in the join operation; (ii) probing, when the bloom filter is used to check for whether elements of the larger dataset are in the smaller one (and therefore are part of the result of the join operation); and (iii) confirmation, when elements with a positive result in the probing phase are checked against the actual original dataset to confirm the result. The confirmation phase is necessary due to the nature of bloom filters, which are based on hash functions and thus risk false positive results, although negative results are guaranteed. All bloom filter code used is based on an existing algorithm by Polychroniou [33] with alterations to account for the different Instruction Set Architecture (ISA) available for our experiments.

Code 4 contains VIMA code for the bloom filter creation phase. It iterates over data elements, calculating bit positions

in the filter based on input elements. Each data point goes through the same calculations to determine its mapping in the filter, which is then set. Factors like the number of elements and acceptable positive rate affect settings such as filter size and number of hash functions. The outer loop loads new elements into the vector, while the inner loop handles bit positioning for each hash function.

Code 4. VIMA bloom join create operator code.

```
for (int i = 0; i < entries_size; i += VECTOR_SIZE) {
    _vim2K_ilmku (&entries[i], mask_l, bit);
    _vim2K_irmku (fun, mask_l);
    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_isrllu (bit, shift5_vec, bit_div);
        _vim2K_iandlu (bit, mask_3l, bit_mod);
        _vim2K_isllu (mask_l, bit_mod, bit);
        _vim2K_iscou (bit, bit_div, bloom_filter);
        _vim2K_iaddu (fun, mask_l, fun);
    }
};
```

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_isrllu (bit, shift5_vec, bit_div);
    _vim2K_iandlu (bit, mask_3l, bit_mod);
    _vim2K_isllu (mask_l, bit_mod, bit);
    _vim2K_igtru (bloom_filter, bit_div, bit_div);
    _vim2K_iandlu (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_l, fun);
};
```

In Code 5, the probing phase’s VIMA implementation is depicted. It verifies if elements in the second dataset exist in the bloom filter set created earlier. In each iteration, elements undergo identical hash calculations, their resulting bit positions checked to ascertain presence or absence. Hash functions determine the bloom filter index to probe for each data point. This is calculated individually for each vector element, directing a gather instruction to fetch relevant bloom filter indices for bit assessment. Bit-wise operations isolate specific bits, with their values indicating presence in the bloom filter.

A vector is used to keep track of which hash function is currently being calculated for each input value, and its elements are updated according to the result of each loop iteration of the probing loop. This value is incremented every

time the bit probed for its associated element is found in the bloom filter and resets to zero when it is not, meaning the element in the corresponding index of the input data vector is deemed absent. If this value reaches the total number of hash functions used in the bloom filter, the corresponding element is stored as a possible positive result. The vector is also used as a mask to load new data for data, replacing elements that have been determined to not fit the condition of the join operation, to not waste any processing time. Once every data point has reached one of the two possible outcomes, all elements deemed present in the bloom filter are eligible to go through the confirmation phase.

The confirmation phase takes every positive result from the probing phase and compares them against the entire original data used to set the bloom filter structure. This step is necessary to remove all possible false positives from the probing phase due to the nature of the hash functions used in the bloom filter. The VIMA implementation is seen on Code 6.

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;
        }
    }
}
```

IV. EVALUATION METHODOLOGY AND RESULTS

This section describes the methodology of our work and the simulation results we obtained to evaluate our query operator implementations using the Vector-In-Memory Architecture (VIMA).

Theoretically, VIMA is able to function with any 3D-stacked memory device, observing its features and limitations. We must note, though, that the organization of the devices directly impacts VIMA performance. Since VIMA is a monolithic device that moves data out of the vaults of the 3D-memory, we expect performance to be superior on memory devices that favor vault parallelism, as opposed to bank parallelism. For our experiments, we consider that the memory controller maps the least significant address bits to vaults and most significant bits to memory banks (similar to what occurs on multichannel systems with DDR-x devices).

To maximize performance in NDP architectures using DRAM-based memories, the most efficient approach is to directly access data on the memory row buffers during each access. This allows for optimal utilization of the internal bandwidth available in the memory. In the case of a SIMD instruction approach like VIMA, adjusting the width of vector operands based on the number of independent vaults and the size of their row buffers is crucial for achieving the best possible performance. Table II presents the relevant features and the theoretically optimal vector size for each memory con-

TABLE II
NDP VECTOR SIZE RECOMMENDED FOR DIFFERENT 3D MEMORY ARCHITECTURES.

Memory	# of Vaults	Buffer Size	Banks	Max. Req. Size	Vector Size
HMC 1.0	16	256 B	8	128 B	4096 B
HMC 2.1	32	256 B	16	256 B	8192 B
HBM	8	2 KB	16	128 B	16384 B
HBM2E	8	1 KB	32	128 B	8192 B
HBM3	16	1 KB	64	128 B	16384 B

figuration, considering both the internal memory bandwidth and the benefits of an NDP architecture.

For instance, if we consider the HMC 2.1 [34], we have 32 independent vaults, each with a 256 B row buffer. Assuming parallel accesses to all 32 vaults, 8192 B are available on the row buffers per access. This is the reasoning behind the 8 KB size of VIMA vector operands, since we assume an HMC 2.1 underlying memory. Since each vault in this configuration has 8 banks that can be accessed in a pipeline fashion, the device could possibly provide 8192 B per access and thus, a NDP architecture could consider this size for its instruction operands in order to extract as much performance from the memory as possible. We could also expect that most of the latency to fetch the next chunk of 8192 B would be hidden by bank parallelism. It should be noted, however, that this line of thought does not necessarily translate to actual performance for every device as it ignores constraints such as internal transmission speed, maximum supported request sizes and the width of the connections between devices. For the HMC 2.1 3D-stacked memory device, however, this is theoretically possible since it supports a maximum request size that is the same size as its row buffers.

A. Methodology

For our testing workloads, we used standard C/C++ math functions and libraries to generate random 32-bit integers. Dataset sizes were chosen for each experiment according to the Last Level Cache (LLC) capabilities of each architecture involved. Since Near-Data Processing (NDP) will usually achieve good performance against a traditional baseline when the dataset being processed overwhelms cache capacity, we ensure that, for every operator, at least one dataset size would overwhelm the capacity of the x86 architecture’s LLC size.

For our experiments we consider three distinct situations: (i) a single-thread x86 system against a single-thread system with VIMA, (ii) a 16-thread x86 system against a single-thread system with VIMA, and (iii) a 16-thread x86 against a multithreaded system with VIMA.

B. Single-Thread Baseline

Table III shows the parameter details used in our simulations with a single-thread baseline. We set parameters to be similar to Intel’s Skylake microarchitecture. We used SiNUCA [35] for all simulations. Its original paper reports only a 9% average

error in comparison with the performance of a real machine, thus being adequate for our evaluation goals.

TABLE III
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;

In Figure 4, VIMA demonstrates higher speedup compared to AVX for the selection and projection query operators. The figures above each bar represent estimated energy savings compared to the baseline. VIMA achieves improved execution speed by effectively utilizing the internal parallelism of the memory during data fetching. Both operators require fetching two operands, leading VIMA to fetch two 8 KB vectors for each instruction. As shown in the figure, VIMA accelerates the selection operator by over 5 \times and the projection operator by 2.5 \times . This performance enhancement is accomplished by leveraging the vault parallelism of the 3D-stacked memory while significantly reducing energy consumption by 75% for the selection operator and approximately 50% for the projection operator compared to the baseline.

For each experiment of the bloom join operation, we use two columns that differ in size by 4 \times . All sizes mentioned in the results refer to the size of the largest column of the two. The smaller column is used to set the bloom filter structure, while the larger one is used for probing.

To simulate real-world conditions, we created datasets with varying selectivity to evaluate the performance differences between systems during data-join operations. The datasets were generated randomly, and selectivity was controlled by intentionally adding elements from the smaller column to the larger column based on the desired selectivity level. Selectivity ranged from 0% to 100%, with increments of 10% for each test. The bloom filter implementation employed a

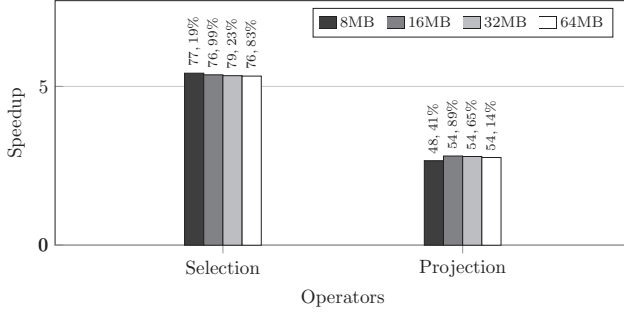


Fig. 4. Speedup over baseline for selection and projection operators, percentages indicate energy savings over baseline.

hash function that utilized multiplications and bit shifting operations from the VIMA ISA [36]. The number of hash functions used in each experiment varied based on the dataset size to maintain a low false-positive rate across all selectivities. The multiplication and shifting factors were identical for both VIMA and AVX implementations. Figure 5 provides an overview of the speedup results, and the numbers atop each bar indicate the estimated energy savings compared to the baseline execution.

Each phase in the execution of the bloom join operator is greatly affected by selectivity in the data, directly impacting performance. The three execution phases are bloom filter creation, bloom filter probing, and confirmation. The bloom filter is set during the creation phase. All data elements in the smaller column of the join go through the hash functions, and the results are used to set the corresponding bits in the bloom filter vector. Since every data element goes through all hash function calculations regardless of numerical value, the creation phase has the same behavior no matter the results expected from the selectivity in the data. On the other hand, bloom filters are much more efficient at determining that any one data element is represented in the data structure than when it is not. This behavior happens as the bloom join executes distinct operations according to data patterns.

At the probing phase, data content directly impacts performance. Here, the bloom join uses the hash result of elements in the second column 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 we can discard it. Consequently, data selectivity determines the length of the probing and confirmation phases. This relationship explains 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’s dedicated cache comes into play. The cache can house the vectors used for the hash function computations in the probing phase, as the bloom join repeatedly reuses them.

The 0% selectivity dataset shows superior results during

the confirmation phase of the bloom join operator. This phase compares positive results from the probing phase to the filter creation data, which can be time-consuming as each element is compared to all elements in the dataset. In datasets with positive results, the confirmation phase becomes a larger portion of the execution time as more elements pass the probing phase. However, the all-negative dataset has fewer positive results, mostly false-positives, resulting in a shorter confirmation phase. The highly efficient probing phase on VIMA explains the significantly better result at 0% selectivity. As selectivity increases, the confirmation phase takes up a larger portion of the execution time, and the architecture’s reuse capabilities start to impact overall performance.

Another factor is the smaller column size, which the bloom join operator repeatedly accesses 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. These sizes mean that for all datasets but the largest one, the baseline architecture’s LLC can store the entire column.

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 its cache hierarchy. Energy consumption follows the same pattern, with VIMA using much more energy as it reloads data from the main memory repeatedly. Meanwhile, 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 under evaluation 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× 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. Thus, AVX no longer benefits from the LLC locality and is forced to reload data directly from the main memory. At this dataset size, the 0% selectivity workload still yields a few thousand false-positive results from its probing phase. Thus, VIMA’s large vectors coupled with the baseline’s fetching inefficiency results in this considerable performance improvement. As selectivity grows, the confirmation phase grows, and VIMA’s advantage drops. However, VIMA continues to outperform AVX by at least 3.5× at 100% selectivity while consuming 54% less energy.

C. Multi-Thread Results

Multithreaded systems traditionally benefit greatly from their ability to fetch and process data in parallel. Since each core is equipped with its own set of functional units

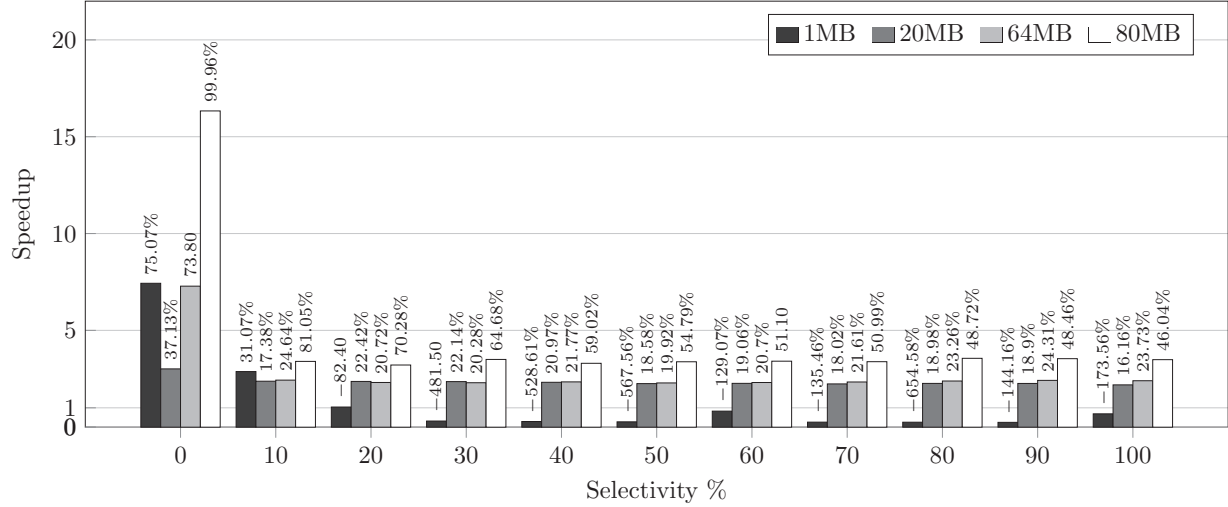


Fig. 5. Speedup over baseline for the bloom join operator with varying selectivity rates, figures over 1 indicate speedup. Percentages over the bars indicate energy savings over baseline, negative values indicate energy consumption exceeded baseline.

and register banks, such systems are able to issue numerous memory requests in parallel, applying increased pressure to the main memory and using much of its bandwidth. For this reason, we now consider a 16-thread system as our baseline, constructing a tough case against VIMA. We assume all 16 cores in the baseline follow the same specifications determined in Table III.

A functional units-based near-data architecture like VIMA, in order to favor simplicity and energy efficiency, is unable to behave like a superscalar processor. Therefore, to provide an execution time performance improvement over such systems, the vector size used by the device must be large enough to match or surpass such levels of parallelism by leveraging as much of the memory bandwidth as possible. Nevertheless, the vector size also impacts on the amount of VIMA instructions the processor needs to trigger to our architecture, which also impacts energy and time. The smaller the operand size, the more instructions the processor must trigger to fully process a given dataset. This is the reasoning behind the 8 KB size of the vector operands we use for VIMA.

Figure 6 shows the results for the experiments considering a 16-threaded baseline. The selection query is a clear example of a data streaming application, being composed of mainly one operation that stores an immediate value in each entry of a vector. As can be seen on the graph, the advantage VIMA has over the baseline shrinks as the input size grows. This happens due to the multithreaded nature of the baseline we are considering, as it suffers from the overhead of splitting the workload at the start of processing and aggregating all results when processing is finished. As input size grows, this overhead becomes a less significant portion of the overall execution time and thus the extent of the advantage of the NDP approach becomes more realistic. This applies to every application with primarily data streaming behavior when considering a multi-

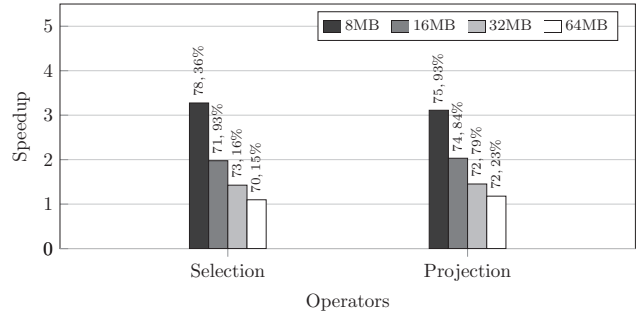


Fig. 6. Speedup over baseline for the selection and projection operators, figures over 1 indicate speedup. Percentages over the bars indicate energy savings over baseline, negative values indicate energy consumption exceeded baseline.

thread baseline.

Although the advantage of VIMA over the traditional architecture is not as pronounced as it was for the single-threaded results, it is still fairly advantageous. Regarding execution time, VIMA is able to at least match the performance of the 16-thread baseline using a single-thread even at the largest input size considered in our experiments. It is able to achieve this result while consuming 70% less energy for the selection operator and 72% for the projection operator. This suggests that, by using VIMA in a system such as the baseline considered here, one could free up 15 cores for other uses while still achieving the same performance regarding execution time and consuming 70% less energy.

Figure 7 shows results for relative average throughput for our experiments considering selection and projection queries. The graph considers relative values of average data throughput achieved by VIMA and the 16-thread x86 baseline. The data throughput and execution time results graphs almost exactly



Fig. 7. Data throughput results of x86 system and VIMA executing selection and projection database queries, normalized to 16-thread x86 baseline.

mirror each other, which shows how better usage of available data throughput is the main reason why VIMA performs better than a traditional architecture when running data streaming applications.

D. Near-Data Multi-Threading

We ran experiments to analyze the performance of a multithreaded system using VIMA and the speedup and data throughput results can be seen on Figures 8 and 9. Our experiments considered the selection and projection database query operators running on a VIMA-enabled system with increasing vector widths (256 B, 512 B and 1024 B) and number of cores (1, 2, 4 and 8 cores). The VIMA cache is shared among all threads, making the bloom join experiment susceptible to thrashing due to the limited memory size. Consequently, our experiments exclusively focused on data streaming operators. In the future, a potential avenue for investigation could involve allocating distinct cache lines to cores for conducting data reuse experiments. The underlying memory chip we used was the HMC 2.1, as it has generally shown the most advantageous results so far, and consider the 64 MB input size for both workloads.

The speedup results, which are normalized to a 16-thread x86 system, show that when using smaller vector operands, VIMA is unable to match the baseline performance when running on a single-threaded system. However, even with only one additional core, it outperforms the baseline for the selection workload and almost matches baseline performance for projection. This advantage scales with larger vectors and a higher number of threads, achieving a $3.7\times$ improvement in execution time over the baseline for the selection database query workload at 8 threads with a 1024 B vector operand width.

The data throughput results in Figure 9 highlight the significant improvement in execution time performance with additional cores. The single-thread system using smaller VIMA vectors lacks the ability to fully utilize the memory and achieve high data throughput. Despite the load-ahead mechanism improving VIMA’s throughput, there are not enough instructions in the instruction buffer to exploit the vault parallelism in the HMC 2.1 memory chip. However, with extra cores, more instructions are issued simultaneously, enabling VIMA to load operands out-of-order and utilize more of the memory’s bandwidth.

As a result, even with a 256 B vector, VIMA outperforms the baseline by 44% even with only 2 cores at the selection database query. Under the same conditions, it gets to 82% of the execution time performance of the baseline for the projection workload, achieving a 13% speedup when using 8 cores. This trend remains true for the results of the experiments with 512 B and 1024 B operands.

V. RELATED WORK

Ailamaki et al. [37] and Boncz et al. [38] were among the early researchers discussing the impact of the memory wall on database system performance during the late 1990s. They observed that processor advancements outpaced storage technology, prompting the development of software techniques aligned with hardware structures to maximize resource utilization. Consequently, strategies like columnar data storage, bulk query relational algebra, cache-conscious algorithms, and automatic optimization became prevalent in database applications, enabling efficient utilization of hardware resources [39].

Although these adaptations achieve their efficiency goal, they fail to reduce data movement, which renders them still susceptible to the increasing issue of the memory wall. However, due to their data-intensive nature, database applications are intrinsically well-suited for near-data execution, and are treated as such by Near-Data Processing (NDP) researchers as near-data technology becomes viable [40].

Columnar storage in modern databases can be harnessed through near-data strategies. JAFAR [40], [41] uses a column-store approach for near-data selection operations, achieving a speedup of up to $9\times$. Their solution features an adjacent accelerator enabling direct data access, executing efficient filtering via simple comparison and predication. This technique produces a bit-mask for tuple selection, further processable with late materialization. While promising, its extensibility to other query operators might pose challenges.

Biscuit [14] translates the MySQL database engine to a near-data implementation on SSD disks, offering a comprehensive framework with dynamic task loading, language support, multicore capabilities, and an expressive programming model. Their reported speedup for all TPC-H queries is $3.6\times$. However, Biscuit relies on complex modifications, including adding processing cores to SSD devices. In contrast, Vector-In-Memory (VIMA) achieves comparable performance improvements with simpler requirements.

One approach that also considers a 3D-stacked memory is HIPE [27], which adds predication to the Hybrid Memory Cube (HMC). This modification, which considers an already modified HMC [26], enables it to compute database algebra queries by allowing control-flow dependencies to be solved near-data. The authors report an $6.46\times$ execution time improvement over an x86 architecture for the selection operator, with 5% higher energy-efficiency. While this approach is very similar to VIMA both in architecture and simulation infrastructure, it fails to match its energy-efficiency and is much more limited, only considering the selection operator.

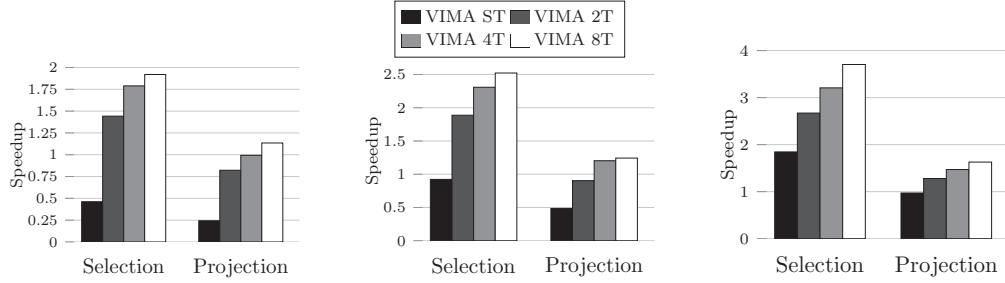


Fig. 8. Speedup of VIMA over baseline running the selection and projection database queries with a varying number of processing threads and (a) 256 B vector operands, (b) 512 B vector operands and (c) 1024 B vector operands. Values higher than 1 indicate improvement in performance over the baseline.

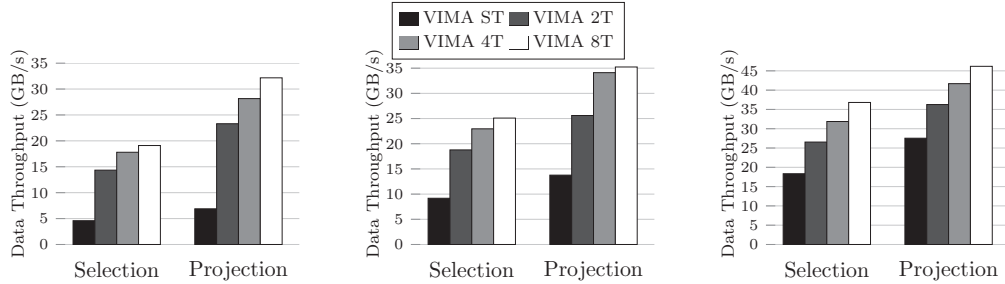


Fig. 9. Data throughput of VIMA running the selection and projection database queries with a varying number of processing threads and (a) 256 B vector operands, (b) 512 B vector operands and (c) 1024 B vector operands. Values higher than 1 indicate improvement in performance over the baseline.

Kepe et al. [13] investigated HMC-based near-data database query operators (selection, projection, aggregation, sorting, and join) against a state-of-the-art x86 system. They employed the HIVE [35] near-data architecture with an Intel AVX-512-equipped baseline. Notably, most operators showed significant improvements, except aggregation. Selection outperformed the baseline, with over $3\times$ faster execution and 45% energy reduction. Projection achieved a $7\times$ to $10\times$ improvement over the baseline, while being $3\times$ more energy-efficient. The join operator, with hash, sort-merge, and nested loop implementations, demonstrated superior near-data performance in execution and energy usage compared to the baseline. Although the aggregation operator underperforms in time and energy, their approach doesn't exploit architecture parallelism via vector sizes. Our work achieves superior execution time and greater energy savings across operators, especially in joins with diverse selectivity. We show speed enhancements from $3.5\times$ to $16\times$, and energy savings of 46% to 99% for the highest input size. Furthermore, we compare to a 16-thread x86 baseline, reinforcing VIMA's superiority over their single-thread baseline.

VI. CONCLUSIONS AND FINAL CONSIDERATIONS

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

Our approach outperforms a single-thread baseline with speedup of up to $5\times$ for selection, $2.5\times$ for projection, and $16\times$ for join operators. It achieves energy savings of 75% for selection, 50% for projection, and 99% for join. These results surpass the state-of-the-art while using a simpler and more programmer-friendly architecture. This work is the first to implement and evaluate database operators on an architecture with large vectors, and also the first to migrate the bloom join operator to an NDP architecture.

Unlike our closest related work, we also consider a modern 16-thread x86 baseline in our experiments, which we also manage to outperform. According to our results, even in a single-thread system, our approach matches the performance of a 16-thread x86 system, meaning our strategy could free up 15 entire cores for processing while maintaining the same execution time performance.

Future work includes migrating other database operators and implementations of the join operator, as other implementations can better suit certain situations. This migration should enable us to evaluate our approach with the entire TPC-H benchmark.

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 rram 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 to 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] O. Polychroniou, *Analytical Query Execution Optimized for all Layers of Modern Hardware*. Columbia University, 2018.
- [34] Hybrid Memory Cube Consortium, “Hybrid memory cube specification 2.1,” 2014, <http://www.hybridmemorycube.org/>.
- [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, 1997.
- [37] A. Ailamaki *et al.*, “Dbms 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.