Operand Size Reconfiguration for Big Data Processing in Memory

Paulo C. Santos[†], Geraldo F. Oliveira[†], Diego G. Tomé[‡], Marco A. Z. Alves[‡], Eduardo C. Almeida[‡], Luigi Carro[†]
[†]Informatics Institute – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
[‡]Department of Informatics – Federal University of Paraná – Curitiba, Brazil
Email: [†]{pcssjunior, gfojunior, carro}@inf.ufrgs.br [‡]{dgtome, mazalves, eduardo}@inf.ufpr.br

***Abstract*—Nowadays, applications that predominantly perform lookups over large databases are becoming more popular with column-stores as the database system architecture of choice. For these applications, Hybrid Memory Cubes (HMCs) can provide bandwidth of up to 320 GB/s and represents the best choice to keep the throughput for these ever increasing databases. However, even with the high available memory bandwidth and processing power, in order to achieve the peak performance, data movements through the memory hierarchy consumes an unnecessary amount of time and energy. In order to accelerate database operations, and reduce the energy consumption of the system, this paper presents the Reconfigurable Vector Unit (RVU) that enables massive and adaptive in-memory processing, extending the native HMC instructions and also increasing its effectiveness. RVU enables the programmer to reconfigure it to perform as a large vector unit or multiple small vectors units to better adjust for the application needs during different computation phases. Due to its adaptability, RVU is capable of achieving performance increase of 27× on average and reduce the DRAM energy consumption in 29% when compared to an x86 processor with 16 cores. Compared with the state-of-the-art mechanism capable of performing large vector operations with fixed size, inside the HMC, RVU performed up to 12% better in terms of performance and improve in 53% the energy consumption.**

Keywords*—*In-Memory Processing, Big Data, Database, Reconfigurable, Vector Instructions, Hybrid Memory Cube

I. Introduction

The amount of data being created and stored by modern applications is ever increasing, such large data sets so-called big data need to be available with fast access to be searched, analyzed and modified when it is necessary, supporting a large number of parallel operations. Such rapid availability becomes a key to productivity and innovation growth in the industry [1]. Although storage space in the big data scenario is not a critical problem for modern systems, to efficiently operate over those big portions of data can be a challenging task for both hardware and software designers. In a processor-based system, data has to travel over the whole memory hierarchy, passing through the disk, main memory, cache memories and registers until it can be processed by the functional units inside the processing cores. In addition, modern processors are composed of tens or hundreds of cores, imposing high pressure in the main memory.

Traditional main memory technologies, as DDR memories, have a series of constraints associated with its architecture, from high latency to reduced bandwidth. For that reason, hardware designers have recently proposed new memory technologies and architectures, such as the Hybrid Memory Cube (HMC). They are turning out as an attractive alternative over traditional memory architectures [2] since it can be used to mitigate those limitations. HMC devices are based on the 3D stack technology, where layers of DRAM and logic elements are stacked on the same chip. It uses Through-Silicon Vias (TSVs) [3] technology to communicate through layers [4], [5]. The internal HMC architecture is formed by 32 vaults that can independently access DRAM banks. That increases overall available parallelism, providing both higher performance and considerably lower energy consumption in comparison to the current memory systems.

The logic layer inside the HMC implements the vault controller plus several Functional Units (FUs) capable of performing arithmetic and logical atomic instructions with operands size of up 16 bytes. Each vault has its independent FUs enabling in-memory processing. Previous research showed that HMC working as a memory device only, can be inefficient when executing streaming applications and some database operations due to traditional memory hierarchy and processors architecture. However, even when considering the in-memory processing capabilities, the HMC instructions are also inefficient for such stream applications, due to the reduced operand size supported. Thus, previous work proposed different ways to overcome such limitation, with large vector instructions for instance. Such large vector instructions can handle homogeneous applications, however, it waste resources for applications with dynamic behavior, such data lookups in database systems.

In this paper, we propose Reconfigurable Vector Unit (RVU), a mechanism that performs in-memory operations inside HMC devices, extending and improving the instruction set available inside these memories. Our mechanism adapts its vector units to handle each computation phase inside the applications better. RVU presents flexibility to operate over 1× vector of 8 KB of contiguous data, or 32× vectors of 256 B, where each vector can operate over different data ranges.

Compared to regular x86 operations, we avoid data movements among memory and processor for streaming applications by performing in memory computation. Thus, reducing the energy consumed by the cache hierarchy while increasing the overall system performance, once cache pollution is also reduced. Compared to related work that performs fixed size vector operations inside the memory [6], RVU is capable of reducing unnecessary computation. Considering a scenario where only parts of a large vector are required to be computed,

The authors gratefully acknowledge the support of CNPq and CAPES.

previous work needs to allocate a full-width vector unit to that memory area. Meanwhile, our mechanism can be reconfigured to break the large vector unit into multiple smaller units performing operations over different data chunks, increasing the system efficiency.

Experimental results executing database predicates show that RVU performs on average 22× better than traditional x86 architectures with 16 cores and HMC memory. It also has improved performance over the previous state-of-the-art mechanism by 12% on average.

II. Background

Columnar Database System (CDBS) or column-stores for short were originated from the Decomposed Storage Manage (DSM) during the early 80's as an alternative for traditional row-oriented database systems [7]. During the recent years, the increasing amount of data being generated put column-stores under the spotlight again. In particular, decision support systems (DSS) require querying large amounts of data, where queries are less predictable, longer lasting, more read-oriented, and more attribute-focused than before with modifications in batch processing [8]. These are also common characteristics of big data programs that made column-stores as the database system architecture of choice. In fact, [9] shows from an interview with 357 Big Data experts that 33% of DBMS used among them are already column-store ones, and others 34% expect to adopt column-store systems in the next three years. In addition, several commercial and non-commercial column-store DBMS have become popular in the past years as MonetDB, HP Vertica, VectorWise and Infobright.

Column-stores create vertical even partitions of a table, where each partition maps to an attribute. The goal is to store the partitions contiguously in a block of data and only scan over the required attributes, instead of the whole tuple of the row-stores [10], as illustrated in Figure 1. Query processing requires joining partitions to reconstruct the output tuple and to apply filters. We stick to the implicit method of tuple reconstruction from MonetDB [11], where each partition is an array with the index matching each tuple in any other array (e.g., partitions $Column1$ and $Column2$ of equal lengths i for which $Column1[i] = Column2[i]$). To apply filters, when a partition $Column1$ is scanned to validate a query filter, only the matching indices are used to scan the next partition $Column2$. This process is called "late materialisation" and its goal is to avoid computing unwanted data and speedup reads (Figure 2).

Besides, column-stores enhance compression with partitions storing the same data type, speedup aggregation queries, and expose parallelism in the system to fetch partitions concurrently [12]. On the other hand, inserting and updating from such databases systems are expensive operations. Multiple memory requests are needed to recover a single tuple of the table, since attributes are spread all over the storage device [13]. Therefore, this database implementation is not a suitable choice for heavily transaction-based applications.

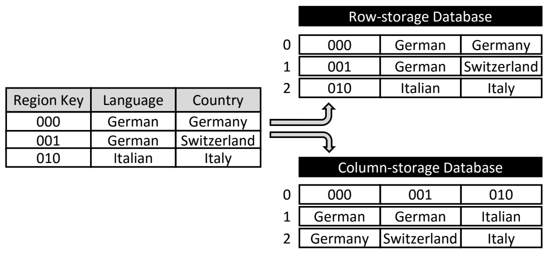
III. Performing Database Operations in HMCs

The primary focus of this work is to improve the overall performance and reduce energy consumption of traditional database operations. Our mechanism uses the Processor-in-Memory (PIM) approach taking advantage of the logic layer provided by Hybrid Memory Cube (HMC). The PIM strategy enables reducing energy consumption by decreasing the number of data movements between the main memory and host processors. Moreover, since the PIM layer is capable of processing database requests with similar performance to the host device, it is possible to achieve a significant speedup for the whole system. To achieve these goals, we focus on column-structured databases, which allow vector operations.

In a traditional system composed of the host processor, several levels of caches, main memory and disk, during full column scans a Database Management System (DBMS) will spend a significant amount of time requesting data from memory. First, it will load all data from each column into the memory hierarchy (*i.e.* caches L3, L2, and L1). Once data is available, it will compute the query and return selected data. These steps suffer from two major problems. First, the collected data have a streaming behavior and presents poor temporal locality, which hurts cache efficiency [14]. This happens because it is likely that the DBMS will load a large amount of data before the materialization, when some data will get reused. Second, for a query that involves more than one column, the host processor will have to move unnecessary data through the system to check whether its value is relevant to the computation or not. Figure 2 illustrates a query over 3 columns, showing that for the second and third columns, some portions of data are not relevant, since they represent a mismatch in the previous column. In a composed database query, data movements can be reduced by checking the previous predicate before requesting data from memory. However, in this case, the cache line size defines the minimum aligned chunk of data to be accessed.

On the other hand, using a near-data approach like the one proposed by [6] would possibly mitigate the first issue cited above. However, the static nature of the [6] will not solve the second issue. The reason is that [6] always operates over 8 KB of contiguous data at the time, accessing, in many cases, large portions of data not required.

In contrast, our mechanism Reconfigurable Vector Unit (RVU) aims at solving both issues by enabling the applications to configure the appropriate data block size, and also by reducing the total amount of data being transferred between memory and processor. It is important to notice that, if RVU is configured with the same size of HMC Instruction Vector Extensions (HIVE) (*i.e.* 8 KB), it can achieve similar results. However, if reconfigured to use smaller vector sizes, it can be better adjusted for the application needs. RVU works as



Row-storage Database
0 | 000 | German | Germany
1 | 001 | German | Switzerland
2 | 010 | Italian | Italy
Region Key | Language | Country
000 | German | Germany
001 | German | Switzerland
010 | Italian | Italy
Column-storage Database
0 | 000 | 001 | 010
1 | German | German | Italian
2 | Germany | Switzerland | Italy

Fig. 1: Storage model alternatives for the logical database.

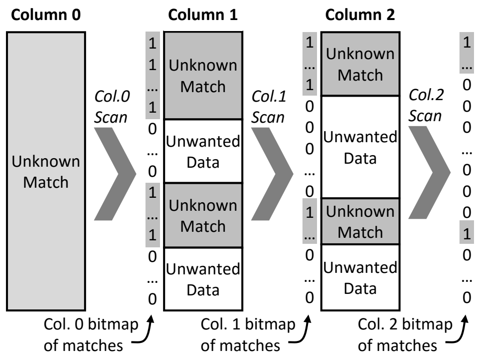


Fig. 2: Late materialization of tuples in query processing over column-stores.

follows:

1. The host device sends a configuration parameter to RVU. This configuration parameter will dictate the appropriate vector size, ranging between 8 KB to 256 bytes. A large vector is suitable when the data being accessed is contiguous through the memory. On the other hand, small vectors will benefit from a sparse memory request. The application can decide the most suitable vector size for one column by analyzing the previously one.

2. Once configured, RVU will access and operate over data blocks of the defined size. Furthermore, it will compute the requests and return to the host device a bitmap vector that represents the results. This bitmap is the simplest form to inform for each entry if there was a match or not, reducing thus the amount of data being transmitted to the processor.

3. Based on how zeros and ones are distributed over the resultant bitmap vector, the application can reconfigure RVU with the appropriated size for the next step.

For example, consider the following query against the table in Figure 1:

```
SELECT
    sum (l_extendedprice * l_discount)
FROM
    lineitem
WHERE
    l_shipdate < date '1995-01-01'
    AND [0.06] + 0.01-01
    AND l_quantity < 24
```

First, the application does not have any information about data pattern, so it will likely configure our mechanism to operate over the maximum vector size supported, *i.e.*, 8 KB. RVU will compute all data from the *l_shipdate* column inside RVU checking for dates before *1995-01-01*, and returning to the host the bitmap corresponding to the matches in that column. Secondly, based on the previous bitmap, the host can generate a proper RVU configuration for the next search, for instance 2 vectors of 4 KB. Then, only requests for the appropriate memory address will be created for the *l_discount* column. In this stage, RVU will check for discounts ranging between 0.07 and 0.05, and return to the host the bitmap corresponding to that second column. For the third column, RVU can be reconfigured again to a smaller size and will access the requested addresses and return the last query result bitmap. Thus, during the query execution, RVU could be adapted to reduce the waste of resources and reduce the access over unwanted data.

A. *Processor to Memory Interface*

When the application needs to perform a data search through the database, it makes the processor to trigger special instructions to the memory system. These instructions are similar to the HMC or HIVE instructions proposed by [6]. RVU operates as a load-store architecture, where the internal registers can be used during computations. For each instruction, the processor can add the base read address or register. Like in the HMC, some operations will return their status to the processor after being executed, for instance, during comparisons a bitmap is returned to the processor as part of the status.

The processor also adds within each instruction the vector size to be used. This information dictates the memory window size. For example, if the processor decides that one small vector is enough to perform the actual computation, the vector size will be 256 bytes. In other words, it means that only one RVU unit will read and operate through 256 bytes of data at the time. Otherwise, if the processor issues an instruction where it is necessary to use 32 vectors, all available RVUs will be enabled, each one reading 256 bytes of data, in a total of 8 KB of data at the time.

Once the memory computation finishes, the processor will receive an acknowledge signal from our device. Then, the processor can either emit another special instruction or read the resultant bitmap vector from the store address it has indicated. With the bitmap information, the processor can also decide how many PIMs units it should use at the next request by checking how sparse or dense the resulted bitmap vector is.

Likewise, the processor is responsible for deciding how many RVU instructions it needs to trigger to compute the whole database column. For example, if the target column has a size of 80 KB of data, and it has been decided that the most suitable vector size to use during the computation was 8 KB, the processor will need to issue 10 instructions for our device over time. Also, the processor needs to manage the address fields in the instruction, doing appropriate memory calculation at each new instruction, until it reaches the end of the column. We consider that the compiler can generate code for our mechanism in a similar way it generates code to use different AVX vector sizes.

B. *Memory Interface*

Figure 3 depicts our proposed design. RVU only adds elements to the logic layer of the HMC device. In total, we have added 32 RVU units. Each unit is composed simply of a small register bank and a set of Functional Units (FUs). The register bank has 8 internal registers, used to store data from memory partitions (working as input register), or to store temporary computation results (working as an output register). Each register is 256 bytes wide, being able to read all elements of the memory row buffer at once. In our design we consider to use the FUs as the related work in order not to restrict the applications handled by RVU. Finally, each RVU is connected directly to the HMC vaults.

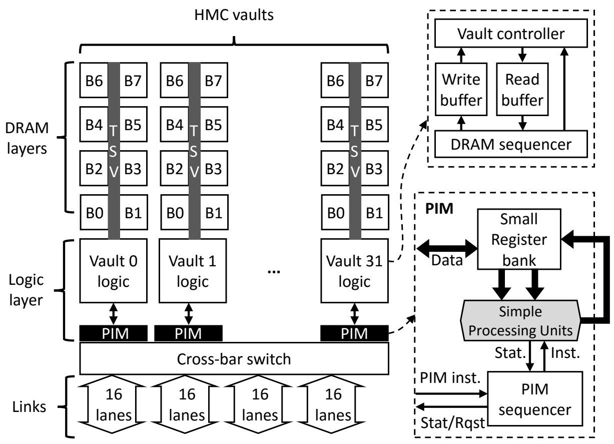


Fig. 3: RVU architecture.

Once a vector unit receives an RVU load instruction from the host processor, it will issue load requests to the vault logic. When the vault responds to the request, the loaded data will be stored in the register file. Thus, the next instructions can perform operations over data inside the register file, and also store the data whenever it is necessary. After each operation finishes, RVU sends an acknowledge signal to the host processor, in a similar way performed by AVX instructions. For each 8192 byes of data requested, only 2048 bits of data will be generated as a result. Therefore, at the end of the operation, less data needs to be send to the processor.

The reconfigurable nature of our mechanism can be accomplished due to an extra field in the instruction. Once the host processor decides the appropriate vector size to work with, it triggers the instruction to RVU indicating how many vector units are suitable for the actual computation.

IV. Experimental Setup and Results

In this section, the methodology and evaluation results are presented. To evaluate RVU we used a cycle-accurate simulator [15]. This simulator allows us to model our custom hardware inside the HMC providing the necessary consideration about all timing issues to simulate our mechanism accurately.

A. Baselines and Configuration Parameters

To compare with our design, two baselines were chosen. The first baseline considered was inspired by the Intel Sandy Bridge processor microarchitecture. The Sandy Bridge was configured with 16 cores and AVX512 instruction set capabilities, and in all cases, the main memory used was HMC which provides high bandwidth. The second baseline was the related work HIVE mechanism, presented in [6]. HIVE is capable of processing 8192 bytes per operation in-memory.

We focused our experiments on TPC-H [16], a decision support database benchmark widely adopted to assess the performance of column-stores. TPC-H consists of a suite of twenty-two business oriented ad-hoc queries that have broad industry-wise relevance and examine large volumes of data. In our experiments, we stick to a mix of three particular queries: TPC-H 04, 06 and 22. Due to the target in near-data processing, these queries were chosen by their feature in expression

TABLE I: Baseline, HMC and PIM configurations.

OoO Execution Cores	16 cores @ 2.0 GHz, 32 nm; 6-wide out-of-order; Buffers: 18-entry fetch, 28-entry decode; 168-entry ROB; MOB entries: 64-read, 36-write; 1-load and 1-store units (1-1 cycle); 3-alu, 1-mul. and 1-div. integer units (1-3-32 cycle); 1-alu, 1-mul. and 1-div. floating-point units (3-5-10 cycle); Branch predictor: Two-level GAs. 4,096 entry BTB; 1 branch per fetch;
L1 Data + Inst. Cache	32 KB, 8-way, 2-cycle; 64 B line; LRU policy; MSHR size: 10-request, 10-write, 10-eviction; Stride prefetch: 1-degree;
L2 Cache	Private 256 KB, 8-way, 4-cycle; 64 B line; LRU policy; ; MSHR size: 20-request, 20-write, 10-eviction; Stream prefetch: 2-degree;
L3 Cache	Shared 40 MB (16-banks), 2.5 MB per bank, 16-way, 6-cycle; 64 B line; LRU policy; Bi-directional ring; Inclusive; MOESI protocol; MSHR size: 64-request, 64-write, 64-eviction;
HMC v2.0 Module	32 vaults, 8 DRAM banks/vault, 256 B Row buffer; 8 GB total size; DRAM@166 MHz; 4-links@8 GHz; Inst. lat. 1 CPU cycle 8 B burst width at 2:1 core-to-bus freq. ratio; Closed-row policy; DRAM: CAS, RP, RCD, RAS and CWD latency (9-9-9-24-7 cycles);
HIVE Processing Logic	Operation frequency: 1 GHz; Up to 2,048 functional units (integer + floating-point); Latency (cycles): INT: 1-alu, 3-mul, 20-div; FP: 5-alu, 5-mul, 20-div; Register bank with 8 registers of 8,192 bytes each;
RVU Processing Logic	Operation frequency: 1 GHz; Up to 32x 64 functional units (integer + floating-point); Vector sizes (bytes): 32x 256, 16x 512, 8x 1024, 4x 2048, 2x 4096, 1x 8192 Latency (cycles): INT: 1-alu, 3-mul, 20-div; FP: 5-alu, 5-mul, 20-div; Register bank: 8 sets of 32 composable registers of 256 bytes each;

calculation and data access locality [17]. They implement complex boolean expressions consisting of conjunctions and disjunctions in large volume tables without join operations between tables. We let join operations for future work as it requires understanding the impact of each one of the many different join algorithms on HMC.

Moreover, we did not implement all of the columns of the TPC-H database. However, this does not bring about any influence on the results as column-stores only access the required columns to solve a query, as discussed in Section II. In our experiments, we are limited to the columns accessed by our query-mix. Although our experiments focus only on database systems, we consider that regarding other applications our mechanism would behave with similar performance than previous proposals when using the same vector size.

B. Performance Results

The performance results for the baselines and RVU executing the TPC-H query 06 are presented on Figure 4. To better detail the explanation, the results are divided on a column basis.

In this case, *column 0* is entirely scanned and its results are used to guide the search in the next column, as previously explained in Section III. In this way, as this first column needs to be fully scanned the maximum vector instruction size are adopted, that is the reason why RVU performance is equivalent to HIVE. It means that both operated over 8192 bytes. At this first step, it is possible to see the pure performance of the in-memory approach compared to the multi-core x86 processor.

After processing the *column 0*, the host processor can analyze the resulting bitmap and determine the new operand

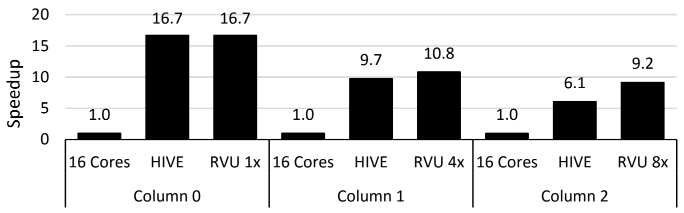


Fig. 4: Detailed performance results for TPC-H query 06.

size, while HIVE architecture keeps the same operation size. The best configuration to access *column 1* from the analysis of the bitmap is 2048 bytes. Thus, four vectors of 2048 bytes may access different memory ranges. That reduces the load footprint from the main memory, reducing the amount of unwanted data accessed and also providing acceleration to the column search. On the conclusion of the *column 1* search, a new bitmap is available, so the main processor can use that information. For the *column 2*, eight operands of 1024 bytes each represents the optimal configuration.

Figure 5 presents the performance results for queries 04, 06 and 22 from TPC-H. For query 04, the small gain shown by RVU occurs due to the fact that most of the entries in the first column did not match the predicate. Thus, a limited number of accesses are required to scan all remaining addresses, which reduces the possibilities of acceleration. In this way, our mechanism executing query 04 could achieve a speedup of 18× compared to 16 cores and 8% when compared to HIVE. For query 06, RVU performed 13× better than 16 cores and 16% better than HIVE. For query 22, it was possible to achieve a speedup of 50× compared to 16 cores, and to accelerate 13% when compared to HIVE.

C. Energy Results

This subsection presents the impact of our mechanism on the DRAM memory energy consumption. Regardless of PIM approach, our reconfigurable techniques presents a substantial reduction in data access, compared to the related work. RVU contribution can be translated into a reduction of the unwanted data access, exploring the near-optimal operand size for each operation. To evaluate the energy consumption from the DRAM layers of HMC we used the Cacti 6.5P tool available inside the McPAT toolset [18]. Considering both data accesses from each application and RVU configuration, we evaluate the impact of our mechanism on energy reduction.

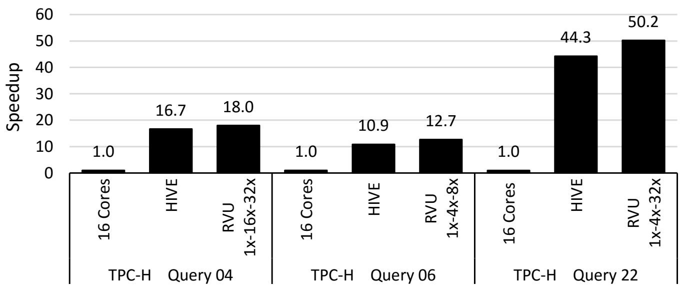


Fig. 5: Performance results for TPC-H queries.

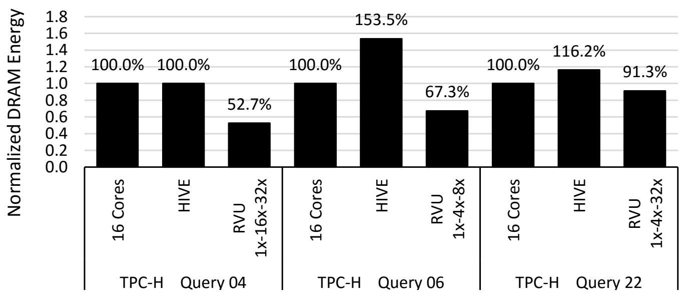


Fig. 6: Energy consumption results for TPC-H queries.

Figure 6 presents the normalized energy results for queries 04, 06, and 22 from TPC-H. In this figure, we can observe mainly two aspects of vector operations. First, since they operate over large chunks of data, in the case of a column with few entries of interest split among the column, the x86 operations will have a smaller footprint, consuming thus less DRAM energy. In the case of RVU this overhead can be reduced by using smaller vectors. The second aspect is regarding the memory alignment. Since the caches operate aligned in the cache line size factor, the vector units inside the memory do not have such constraint, enabling the program to reduce the memory footprint of data accessed by the vectors.

Energy results for query 04 show that both baselines, with 16 cores or with HIVE had the same DRAM energy consumption. However, the better adjustment of RVU for each phase will reduce the amount of unwanted data accessed, hence reducing the energy consumption by 48 %. For query 06, HIVE accessed more data than required by the 16 core system, due to the fixed large operand size. In contrast, RVU was capable of reducing the amount of access to unused data reducing the energy consumption by 22%. Finally, for query 22, RVU reduced by 9% the energy consumed compared to the 16 core system.

V. Related Work

A significant portion of efforts to mitigate the so-called memory wall addresses the different speeds between traditional Dynamic Random Access Memory (DRAM) memories and today's CPU cores. One could categorize all these related works into two broad classes, one that places logic inside the DRAM structure, and another one outside the memory. In the former one, [19] proposes to increase the number of memory ports and data buses. Then, a vector processing unit can be added inside the DRAM module to take advantage of the extra bandwidth. On the other hand, [20] places several functional units together with the memory sensor amplifiers. Each functional unit can operate at the bit level. Similar to [19], [20], Alves et. al [21] implement vector extensions inside the memory with a set of vector units per devices, sharing resources through row buffers. The main problem with all these approaches is to merge logic together within the already challenging DDR development task. In the latter class of related work, one could find studies such as [22], [23], and [24]. In the first one, Farmahini-Farahani et. al. present a 3D DRAM-processor accelerator that connects a lightweight CPU-core with a 2D DRAM die using Through-Silicon Via (TSV). Similarly, [23] proposes a Coarse-Grain Reconfigurable

Array (CGRA) on the top the memory, also connecting both systems by TSV. Finally, in JAFAR [24], a simple functional unit designed to compute relational operations, is connected to the memory I/O buffer, aiming to accelerate column-stores database systems. In brief, those methodologies suffer from not taking full advantage of the internal memory bandwidth and relying on the programmer to decide the application portion to be optimized.

In the past years, the now feasible implementation of 3D-DRAM stacked memories has motivated several new studies in the Near-Data Processing (NDP) field. For instance, [25] adds to the logic layer of a HMC device, sixteen ARM A5-like processors to improve MapReduce workloads. [26] introduces a sophisticated pipelined processor in the HMC logic layer. In [6], the authors present HIVE, a vector processor instruction extension that allows the HMC system to perform vector operations. Finally, [27] shows an accelerator system to speed up graph-based databases. It uses sixteen HMC devices to store and compute graphs operands. All HMC modules are seen as an accelerator by the system, and computation is scheduled explicitly using a proposed programming model.

All those works mentioned, whether based on 2D-DRAM memories or 3D-HMC technology, have one major issue in common: they assume that the in-memory computation will mostly have a streaming style behavior. The number of computational resources is decided during the design phase, expecting to make use of all internal or external provided bandwidth. However, this methodology can cause a waste of logic elements, since applications have different needs for data at a given period. Without our provided reconfiguration, unnecessary power dissipation due to unmatched bandwidth is required from the system, decreasing also the overall system performance. Our approach aims to eliminate this static behavior by dynamically deciding the appropriate number of accelerator units and their necessary data bus width.

VI. Conclusions and Future Work

In this paper, we presented RVU, a reconfigurable mechanism that is capable of better matching different application phases. Due to its reconfigurability, RVU can operate over either large or small chunks of data, being suitable for applications that have such dynamic behavior, like column-store databases. When compared to another state-of-the-art vector-processing mechanism, RVU shows a significant performance improvement of 12%. Our system can enhance overall performance by reducing the number of operations over unnecessary data while enabling its functional units to perform the required computation using only wanted data. As a future work, we plan to propose a metric to remove from the software layer the responsibility of choosing the best size to reconfigure the vector units during the runtime.

References

[1] M. Stonebraker, S. Madden, and P. Dubey, "Intel big data science and technology center vision and execution plan," *ACM SIGMOD Record*, vol. 42, 2013.

[2] Altera, "Hybrid memory cube controller IP core user guide," 2015, https://www.altera.com/solutions/technology.html.

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

[4] Hybrid Memory Cube Consortium, "Hybrid memory cube specification rev. 2.0," 2013, http://www.hybridmemorycube.org/.

[5] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *Symp. on VLSI Technology*, 2012.

[6] M. A. Z. Alves, M. Diener, P. C. Santos, and L. Carro, "Large vector extensions inside the HMC," in *Conf. on Design, Automation & Test in Europe*, 2016.

[7] G. P. Copeland and S. N. Khoshafian, "A decomposition storage model," *SIGMOD Rec.*, vol. 14, no. 4, May 1985.

[8] P. B. S. K. Dhindsa, "A comparative study of database systems," *Int. Journal of Engineering and Innovative Technology*, pp. 267–269, 2012.

[9] P. Russom, *Managing big data*, The Data Warehousing Institute, 2013.

[10] S. Manegold, P. A. Boncz, and M. L. Kersten, "Optimizing database architecture for the new bottleneck: Memory access," *The VLDB Journal*, vol. 9, no. 3, Dec. 2000.

[11] P. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: hyper-pipelining query execution," in *Conf. on Innovative Data Systems Research*, 2005.

[12] P. Russom, *Analytic Database for Big Data*, The Data Warehousing Institute, 2012.

[13] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, "Column-oriented database systems," *VLDB Endowment*, vol. 2, no. 2, Aug. 2009.

[14] P. C. Santos, M. A. Z. Alves, M. Diener *et al.*, "Exploring cache size and core count tradeoffs in systems with reduced memory access latency," in *Euromicro Int. Conf. on Parallel, Distributed, and Network-Based Processing*, 2016.

[15] M. A. Z. Alves, M. Diener, F. B. Moreira *et al.*, "SiNUCA: a validated micro-architecture simulator," in *High Performance Computation Conf.*, 2015.

[16] *TPC BENCHMARK H*, 2nd ed., Transaction Processing Performance Council (TPC), 2014.

[17] P. A. Boncz, T. Neumann, and O. Erling, "TPC-H analyzed: hidden messages and lessons learned from an influential benchmark," in *TPC Technology Conf. Performance Evaluation Benchmarking*, 2013.

[18] S. Li, J. H. Ahn, R. D. Strong *et al.*, "The McPAT Framework for Multicore and Manycore Architectures: Simultaneously Modeling Power, Area, and Timing," *Transactions on Architecture and Code Optimization*, vol. 10, no. 1, p. 5, 2013.

[19] D. Patterson, T. Anderson, N. Cardwell *et al.*, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, Mar. 1997.

[20] D. G. Elliott, M. Stumm, W. M. Snelgrove *et al.*, "Computational RAM: Implementing Processors in Memory," *Design and Test of Computers*, vol. 16, no. 1, pp. 32–41, Jan. 1999.

[21] M. A. Z. Alves, P. C. Santos, F. B. Moreira, and opthers, "Saving memory movements through vector processing in the dram," in *Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, 2015.

[22] A. Farmahini-Farahani, J. H. Ahn, K. Compton, and N. S. Kim, "DRAMA: an architecture for accelerated processing near memory," *Computer Architecture Letters*, no. 99, 2014.

[23] H. Asghari-Moghaddam, A. Farmahini-Farahani, K. Morrow *et al.*, "Near-DRAM acceleration with single-ISA heterogeneous processing in standard memory modules," *IEEE Micro*, vol. 36, no. 1, Jan. 2016.

[24] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos, "Beyond the wall: Near-data processing for databases," in *Int. Workshop on Data Management on New Hardware*, 2015.

[25] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian *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*, 2014.

[26] R. Nair, S. F. Antao, C. Bertolli, P. Bose *et al.*, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, 2015.

[27] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Int. Symp. on Computer Architecture*, 2015.