Opportunities and Challenges of Performing Vector Operations inside the DRAM

Marco A. Z. Alves, Paulo C. Santos, Matthias Diener, Luigi Carro
Informatics Institute
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
{mazalves, pcssjunior, mdiener, carro}@inf.ufrgs.br

ABSTRACT

In order to overcome the low memory bandwidth and the high energy costs associated with the data transfer between the processor and the main memory, proposals on near-data computing started to gain acceptance in systems ranging from embedded architectures to high performance computing. The main previous approaches propose application specific hardware or require a large amount of logic. Moreover, most proposals require algorithm changes and do not make use of the full parallelism available on the DRAM devices. These issues limits the adoption and the performance of near-data computing. In this paper, we propose to implement vector instructions directly inside the DRAM devices, which we call the Memory Vector Extensions (MVX). This balanced approach reduces data movement between the DRAM to the processor while requiring a low amount of hardware to achieve good performance. Comparing to current vector operations present on processors, our proposal enable performance gains of up to 97× and reduces the energy consumption by up to 70× of the full system.

CCS Concepts

•**Computer systems organization** → Heterogeneous (hybrid) systems;

Keywords

Near-data computing; Reducing data movement; Vector instructions;

1. INTRODUCTION

The main memory bandwidth of current systems is a known performance bottleneck. Furthermore, the data transfers between the memory and processor is a contributor to a system's power consumption, especially for embedded systems, which have stringent restrictions. Double Data Rate (DDR) memories have emerged as a major technological breakthrough, providing the ability of transmitting data at both clock edges. Furthermore, the interconnection between the main memory and the processors improved from a simple shared bus to dedicated point-to-point interconnections. Despite these advancements, memory accesses still present a large challenge for the performance and energy consumption of modern multi-core processors. Thus, besides the burst technique, sets of devices are deployed in a module to increase parallelism and increase data throughput. A widely adopted technique to integrate a large number of devices is the use of multiple channels and multiple memory controllers. This technique allows memory modules to be used in parallel, although on a limited bus width per module.

However, even for systems with a high memory bandwidth, if we consider streaming applications that present a high spatial locality and low temporal locality of memory accesses, the cache hierarchy will represent a waste of resources in terms of performance and energy consumption. The reason for this waste is that for most stream applications, data that is brought into the caches is only used once and removed as soon as possible to make room for new data. In cases where the processor is performing bursts of requests, the prefetch requests will be triggered at the same time as the normal requests, thus memory prefetchers will not reduce the average memory access latency [8].

For these two reasons, the concept of *near-data comput-*

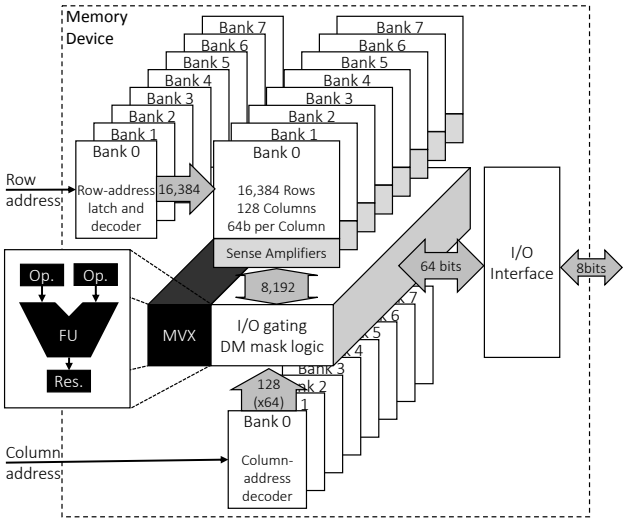


Figure 1: DDR 3 x8 functional block diagram of a single device [11] with our MVX mechanism (in black).

MEMSYS '15, October 05-08, 2015, Washington DC, DC, USA

DOI: http://dx.doi.org/10.1145/2818950.2818953

ing aims to remove the requirement of large data movements between processors and main memory [3]. Moving the logic closer to the data enables the processing elements to access wider data widths. This can be explained by internal DRAM organization. Figure 1 presents an overview of a DDR 3 x8 module. Traditional memory modules are formed by multiple devices that act in a coordinated way. The devices are composed of a set of banks, and all the devices react to an operation signal, always operating in the same bank for a given signal. These banks are composed of sub-arrays, formed by rows that are accessed per column [7]. Once a row is precharged and opened, its data is read by the sense-amplifiers and becomes available in the row buffers. At this point, the selected columns from different devices are transmitted via the interconnection to the processor. However, instead of transmitting this open row, its data could be consumed by near-data processing elements, avoiding the interconnection overhead as well as providing access to a wider amount of data.

In this paper, we present and evaluate the possibility of integrating Functional Units (FUs) directly inside the DRAM devices to perform near-data computing. We call this mechanism *Memory Vector Extensions (MVX)*, and it is illustrated in Figure 1. MVX integrates logic close to the row buffers, which is triggered by the processor and processes wide portions of data in parallel. A typical row buffer size provides 8 KB of contiguous data by multiple devices which can be processed by a single vector instruction. We also discuss implementation possibilities and future challenges of MVX. Compared to previous proposals for near-data computing, MVX combines a small logic with a high bandwidth, low data movement, and provides general purpose processing capabilities.

MVX is designed to outperform traditional processor vector instructions for algorithms that present high spatial locality and low temporal locality, such as applications with a stream memory access behavior. This is because such applications cannot benefit from cache memories inside the processors to hide the memory access latency and limitations of the interconnection. We present experimental results in terms of execution time and energy consumption with three application kernels with different memory access behaviors (low, medium and high data re-usage), using several MVX implementation choices. We also discuss the area overhead of our mechanism. In terms of performance, our mechanism is able to provide up to 97× (39× on average) improvement over the baseline system formed by 16 low-power cores. Regarding energy consumption, MVX consumes up to 70× (13× on average) less energy than the baseline system with a moderate area overhead.

2. THE MVX MECHANISM

The main focus of this work is to move the execution of vector operations from the processor to the DRAM, reducing the data movement between processor and main memory. Our Memory Vector Extensions (MVX) obtain data directly from open rows inside the devices. To take full advantage of the data available inside the memory devices, we implement a register bank and a set of vector functional units inside the DRAM. These new functional units will answer to specific MVX instructions which we introduce, while the register bank enables the memory to operate in parallel with our mechanism.

In this section, we introduce MVX in detail, presenting the required processor and memory system modifications as well as the operations of the mechanism. In the description in the following subsections, we follow the instruction path, from the binary generation to the actual execution in the DRAM functional units.

2.1 Overview of MVX

Figure 2 depicts the full data-path of an MVX instruction from the processor to the DRAM. The new MVX instructions are fetched and decoded by the processor and then sent to the main memory to be executed in the DRAM, avoiding expensive data transfers between the memory and processor.

We assume a memory module formed by 8 devices, each device containing row buffers of 1024 bytes. With this configuration, each set of row buffers contains 8 KB of data, which corresponds to 2,048 operands of 32 bit (integer or single precision FP). In this way, each MVX instruction uses 3 internal registers to perform up to 2,048 operations of the same type (compared to 16 operations in AVX-512, for example). In our model, MVX supports all of the ARM NEON operations (integer and FP).

2.2 Binary Generation

In order to use MVX instructions, we require no changes to the source code of an application. However, the code needs to be recompiled in order to make use of the MVX instructions. The automatic vectorization techniques (similar to the ones present in current compilers [10]) from the compiler are extended to make usage of our wider operations. To avoid resource conflicts, a sequence of MVX instructions needs to be wrapped by MVX's lock and unlock instructions. These instructions will perform a lock in the MVX structures for a specific thread, unlocking it whenever an unlock is executed.

2.3 Processor Modifications

In the processor, we require an Instruction Set Architecture (ISA) extension to provide our MVX instructions. The MVX instructions use a new register bank inside the DRAM to perform operations. The MVX instructions pass through the pipeline in the same way as a memory load operation. MVX instructions that do not require memory addresses, such as MVX lock and unlock, will bypass the Address Generation Unit (AGU) and wait to be transmitted inside the Memory Order Buffer (MOB).

All of the MVX instructions are sent to the MOB to be delivered to the memory subsystem. These instructions wait inside the MOB for an answer from the memory system, which returns the status of the operation as successful or

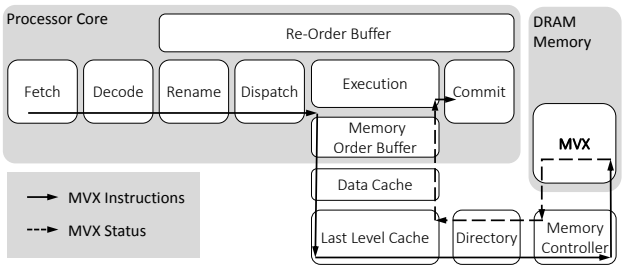


Figure 2: Data-path illustration of an MVX instruction.

raises exceptions. The processor uses these instructions' status to control execution flags, such as overflow and not-a-number, among others. MVX instructions that perform loads and stores work with virtual addresses and therefore have to be translated by the Translation Look-aside Buffer (TLB) and checked for correct permissions to access the given address range. After passing through the TLB, the requests follow the cache memory hierarchy, bypassing the memory caches. The cache directory needs to be changed as well, to ensure a write-back of all the modified data in the range at which the specific MVX instruction will operate.

2.4 Memory Controller Modifications

In our mechanism, the memory controller is responsible for handling the instructions and sending them to the DRAM in-order. This reduces the logic required inside the DRAM. Furthermore, the MVX instructions use the internal buffers inside the memory controller to wait until they can be executed.

When the memory controller receives the MVX lock operation, it has to lock the MVX mechanism inside the DRAM to operate only for the thread that requested the lock. In case the memory is already locked, the lock instruction from the requester waits until the DRAM gets unlocked. After a lock is granted, the MVX instructions are able to perform their operations. Locking the mechanisms avoids that one thread modifies the content of registers that are being used by a different thread. A simple mechanism could make use of this locking system to power gate or clock gate all MVX resources after a certain period of time, reducing energy overhead during idle periods. Normal memory access requests (both read and write) can still be serviced while the MVX is locked, such that other threads that do not use MVX can continue to execute normally.

2.5 DRAM Modifications

To perform vector instructions inside the DRAM, we require two main logic additions to the DDR device, a register bank and the vector functional units. Figure 3 illustrates MVX inside a DRAM device. This figure presents a DDR 3 x8 device, even though our mechanism can be easily adapted for different DDRx device layouts (for example, different row bank sizes, data bursts, etc.).

The additional register bank inside the DRAM device can be used to store full row buffers from any bank inside the device. Each register is capable of handling an entire row buffer (8,192 bits per device). Thus, open row signals can be issued to different banks to achieve a higher performance. MVX interacts with the DRAM only during load and store operations by copying data to and from the MVX registers. Therefore, our mechanism does not require new DRAM signals.

MVX instructions are executed in-order, however, its functional units act as a restricted data-flow processor. A given operation may start as soon as the registers are ready. To support that data-flow, we provide a flag associated with each register that indicates if the operand is ready. Each MVX instruction needs to erase this flag for its destination register, and re-enable it whenever the instruction becomes ready. This system enables the DRAM to open rows from different banks in parallel, and also ensures that once an MVX instruction requires operands that are not ready yet, execution will stall.

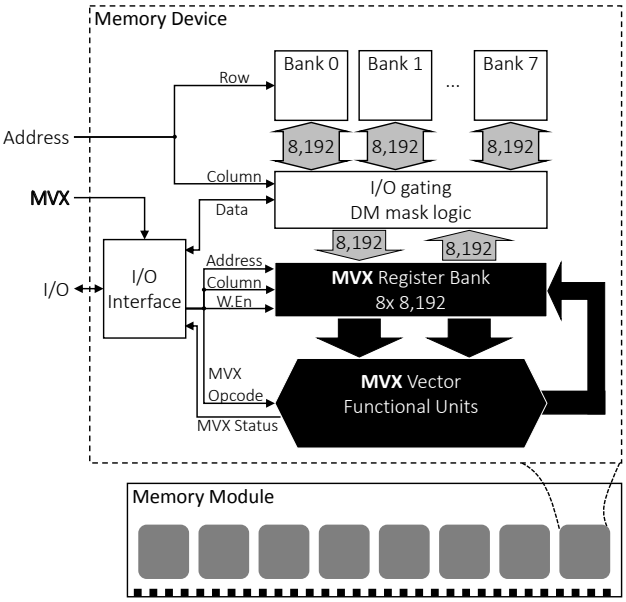


Figure 3: Memory module with DDR 3 x8 devices showing the modifications required by our mechanism. Sizes are given in bits.

Upon registers being ready, the functional units operate in several steps (DRAM cycles) to process the entire row buffer. The number of steps depends on the number of available functional units. We further explore the trade-offs of the number of functional units in Section 3. All functional units operate at the same frequency as the DRAM device. After completion, every MVX instruction sends a status to the processor, such that our instructions behave similar to a normal memory request. These acknowledgment signals provide important information for the processor regarding the status of each operation, such as overflow, division-by-zero and other exceptions. For instance, in the Intel AVX instruction set, 17 bits are enough to provide the information regarding the operation status [6]. During our mechanism evaluation, we considered an acknowledgment of 64 bits in order to correctly simulate the impact of this transmission on the final performance.

3. EXPERIMENTAL EVALUATION OF MVX

This section presents the simulation details, the application kernels and the evaluation results comparing MVX to a baseline system.

3.1 Configuration Parameters and Baseline

To evaluate our MVX mechanism, we used an in-house cycle accurate simulator. [1, 2]. The simulation parameters are inspired by Intel's Atom processor with the Silvermont micro-architecture. Table 1 shows the simulation parameters used for our tests. In order to model the energy consumption, we feed our performance results into the McPAT 1.3 and CACTI 6.5P energy modeling tools [9]. We modeled the components with the embedded system and power gating parameters enabled. Our modeled processor uses a 32 nm integration technology, while the DRAM and MVX models use a 45 nm integration technology.

The Atom processor with the Silvermont micro-architecture only supports SSE 4.2 instructions on up to 8 cores and 2 memory channels. To build a possible future scenario for comparison, we added support for AVX vector instructions using 512 bit registers (the same used on Xeon-Phi processors). We also extrapolated the baseline to include up to 16 cores, and support up to 8 memory channels. As the MVX hardware is a set of vectorial functional units and a register bank, we performed the experiments with the MVX working at the DRAM device operating frequency, with all the vectorial operations executing in parallel. However, experiments with a reduced number of functional units are also presented, where execution requires multiple iterations. We consider the worst case for the performance, where no pipelining is performed between the iterations.

3.2 Application Kernels

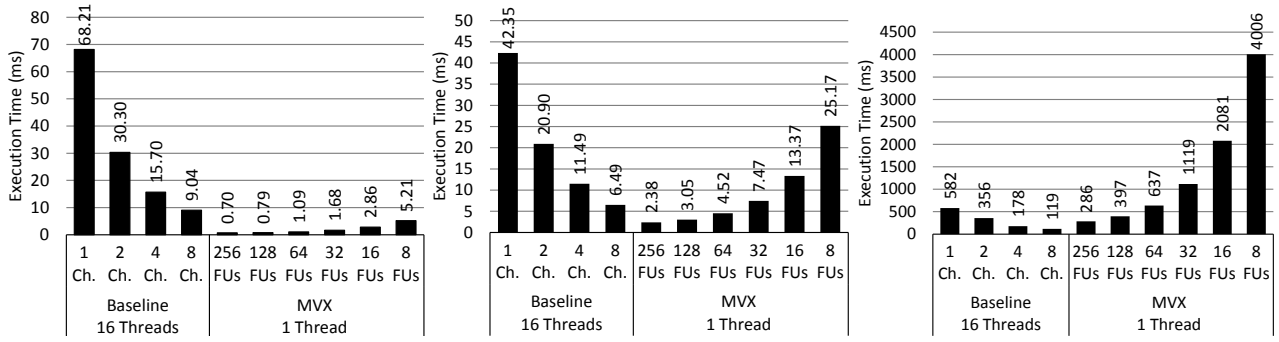
For our evaluations we used three different floating point applications kernels: A **vector sum** using three vectors of 64 MB each. A **stencil** with 5-points over a matrix of 64 MB, adding up the 5 neighboring elements, multiplying the result by two and then storing every result in an output matrix. A **matrix multiplication** using three square matrices (2 source and 1 result) of 9 MB each. All kernels use floating-point operands.

The vector sum application represents the most favorable case for our mechanism since it does not reuse data and only performs a stream over contiguous vector elements. The stencil benchmark presents some data reuse and can make use of the cache memories. The matrix multiplication application has a high amount of data reuse, thus benefiting greatly from the cache memories. The assembly code of the three application kernels was obtained from the gcc compiler using auto-vectorization and was then manually adapted to use MVX instructions.

3.3 Performance Results

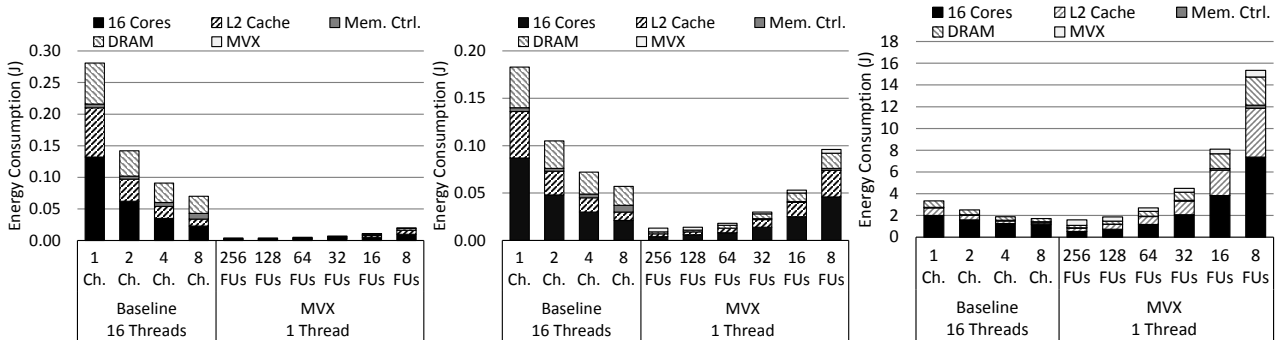
Figure 4 evaluates the baseline system performance scalability when the number of memory channels increases, for the three applications kernels. It also shows the performance changes for our MVX mechanism with different numbers of functional units per device. With this experiment, we intend to show the break-even point for the implementation of MVX when compared to parallel architectures with increasing numbers of memory channels (higher data bandwidths). For the vector sum, we found that MVX with 256 FUs performs 12.8× faster than a baseline that uses 8 channels and 16 cores. When increasing the amount of data reuse, for the stencil application, our mechanism still performs 2.7× faster than the baseline that uses 8 channels and 16 cores. For the matrix multiplication, which has a high amount of data reuse, the break-even point occurs with 16 cores with 4 or more memory channels.

These performance results also give us insight regarding



(a) Vector sum. (b) Stencil. (c) Matrix multiplication.

Figure 4: Execution time of the kernels, varying the baseline number of channels and number of functional units per device of MVX.



(a) Vector sum. (b) Stencil. (c) Matrix multiplication.

Figure 5: Energy consumption of the kernels, varying the baseline number of channels and number of functional units per device of MVX.

a project decision considering that our mechanism could be implemented using fewer functional units, performing the operations over fewer operands in a multi-iteration way. In this case, the latency to perform a single operation would be multiplied, depending on the number of functional units. For vector sum and stencil, which present zero or low amount of data reuse, we can observe that even reducing the number of functional units to 8, our mechanism continues to deliver a higher performance than the baseline system with a single channel and 16 cores. Executing the matrix multiplication application, our mechanism with half of the functional units (128 per device) is still better than a system with 1 or 2 channels and 16 cores. We can see with these results that our mechanism presents a wide design space with sustained performance gains.

3.4 Energy Results

During the execution, using only one processor core is enough to achieve high performance gains with MVX, while the other 15 cores are idle. This reduces the processor power consumption. On the other hand, the power consumption of the main memory is increasing due to the execution of the MVX instructions. On average, the combined power consumption of the processor and DRAM keeps at the same level.

Figure 5 evaluates the baseline and our mechanism energy consumption when executing the three applications kernels. As expected, the energy consumption has a high correlation with the performance results. MVX reduces the energy consumption by 70× for vector sum, 14× for stencil and 2× for matrix multiplication compared to the baseline with a single memory channel. Higher energy savings could be achieved by fully disabling unused processor cores or reducing the number of processor cores present in the MVX system. McPAT models the same power leakage for idle and executing cores, which explains part of the higher processor energy consumption when the execution time increases. However, for applications with higher data reuse, our mechanism requires extra memory accesses due to the lack of cache memories, reducing performance and consuming more energy.

Table 1: Baseline and MVX system configuration.

OoO Execution Cores
2 GHz; 16 cores, Integration tech. 32 nm;
In-order front-end and commit; Front-end 2-wide;
14 stages (3-fetch, 3-decode, 3-rename, 2-issue, 3-commit);
Buffers: 24-entry fetch, 32-entry decode, 32-entry ROB;
2-alu, 1-mul. and 1-div. integer units (1-3-20 cycle);
1-alu, 1-mul. and 1-div. floating-point units (5-5-20 cycle);
1-load and 1-store functional units (1-1 cycle);
MOB entries: 10-read and 10-write;
Branch Predictor
1 branch per fetch; 8 parallel in-flight branches;
4 K-entry 4-way set-associative, LRU policy BTB;
Two-Level PAs predictor; 16 K-entry BHT, 2-bits;
L1 Data + Inst. Cache
32 KB, 8-way, 2-cycle; 64 bytes line; LRU policy;
MSHR entries: 8-request, 8-write-back, 1-prefetch;
Stride Prefetcher: 1-degree, 16-strides table;
L2 Cache
1 MB shared for every 2 cores, 16-way, 4-cycle;
64 bytes line; LRU policy;
MSHR entries: 16-request, 8-write-back, 2-prefetch;
Inclusive LLC; MOESI coherence protocol;
Stream Prefetcher: 2-degree, 16-distance, 32-streams;
Memory Controller and Interconnection
Bi-directional ring; On-chip DRAM controller;
Open-row first policy, 1-channel;
DRAM Module
DDR3-1333, 8 burst length at 3:1 bus frequency ratio;
8 GB total size; Integration tech. 45 nm; Device@166 MHz
8 DRAM banks, 8 KB row buffer (1 KB / device);
CAS, RP, RCD, RAS and CWD latency (9-9-9-24-7 cycles);
MVX Processing Logic
Operation frequency: 166 MHz; Integration tech. 45 nm;
Latency (cycles): 1-alu, 3-mul. and 20-div. int. units;
Latency (cycles): 5-alu, 5-mul. and 20-div. fp. units;
Up to 256 sets of functional units (int. + fp) / device;
1 register bank / device, with 8 registers of 8,192 bits each;

3.5 Area Results

MVX can be implemented with different number of vector FUs, creating a trade-off between performance and area. Figure 6 presents the estimated area (in mm^2) required for each different implementation of MVX and the baseline system with different numbers of memory channels. For our baseline configuration with 256 FUs, MVX requires about 49 mm^2 per device, which represents 14% of the total DRAM size (8 devices). We can observe that with a reduced amount of functional units per device, the overhead is reduced almost linearly due to the fixed size of the register bank, which accounts for only 1% of the total DRAM size. In our modeling, we considered a set of integer units plus ARM NEON units able to support the all the integer and floating point instructions. However, a subset of operations could be selected in order to reduce the logic overhead.

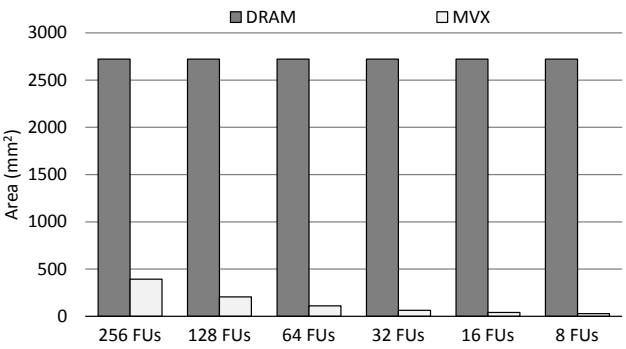


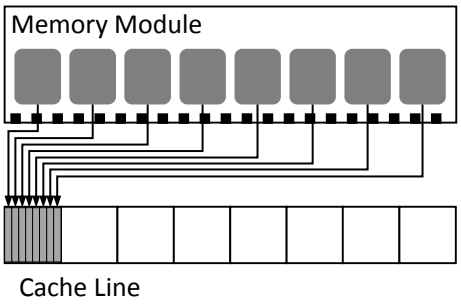
Figure 6: Area consumption (in mm^2) of the DRAM and MVX for different MVX configurations.

4. FUTURE CHALLENGES

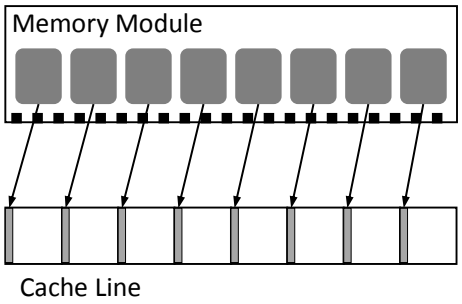
As presented in the previous section, near-data computing with functional units integrated in the DRAM is capable of providing substantial performance and energy consumption improvements. However, for the adoption of MVX, several implementation issues have to be considered, which are discussed in this section.

4.1 Interleaved Data on Row Buffers

DRAM devices have a transmission width, which defines the column size and thus the number of devices needed in a module to handle requests of a predetermined size (currently,



(a) Original interleaving granularity with 1 byte per device.



(b) Proposed interleaving granularity with 8 bytes per device (required by MVX).

Figure 7: Different data interleaving granularities.

a cache line, 64 bytes). Since memory modules usually have 64 pins, the 8-prefetch-burst implemented in DDR 3 will make the devices answer enough times to constitute the cache line. Typically, devices have x4, x8 or x16 device widths. Assuming an x8 width per device, this means the 64 bits sent by the module each cycle are interleaved in the 8 devices, and therefore, integer or floating point values (4 or 8 bytes) are not stored entirely on one device.

This poses an important issue, as our mechanism requires entire elements to be present in the row buffers of a single device for calculation. To resolve this problem, we propose that the memory controller transforms the cache lines before reading or writing to the module, reordering the information such that the datum gets interleaved with 8 bytes per device. Figure 7 illustrates the original interleaving granularity of DDR 3, where the set of all 8 devices provides 8 bytes of contiguous data per transmission (Figure 7a). The figure also shows the proposed interleaving granularity, where on each transmission non-contiguous blocks of data are transmitted (Figure 7b). This scheme enables each device to store 8 bytes of contiguous data, which can hold full operands (up to 8 bytes integer or floating point operands) for the MVX operations. It is important to note that this modification changes only the memory controller operation, while the DRAM still operates in the same fashion, with the same data burst performance.

4.2 Interleaved Data on Multiple Memory Modules and Channels

In this paper, we only presented results with our mechanism implemented in a single memory module and a single channel, which is a common scenario for embedded systems. However, in order to implement our mechanism in a system with multiple channels and multiple modules, further investigation is required. In systems with multiple modules or channels, an operation might require operands from different channels. In this situation, an operand might need to be copied between the modules, generating additional overhead. This is a very common unresolved issue with most near-data computing proposals.

This issue could be resolved by the programmer for a particular system and application, but it would not fit multiple systems. Moreover, it should be performed in a synchronized way together with the Operating System (OS). A more generic solution could be to set a range of page frames on the memory to have an explicit data mapping in the memory modules. Thus, with some indication in the algorithm, the OS could place the Memory Vector Extensions (MVX) operands in page frames to be mapped to a single memory module. This solution would require annotations in the source code, and changes on the Memory Management Unit (MMU), in order to avoid data transfers between the memory modules during MVX operations.

4.3 Low Data Re-Usage Requirements

For applications that have none or low temporal locality, our mechanism avoids the data transfer overhead gracefully. However, for applications with higher temporal locality, the mechanism starts to gain less performance due to the extra DRAM accesses. The register bank inside the DRAM devices can help in this issue by storing the operands to be used during an MVX transaction (between a lock and an unlock). Moreover, the register bank also enables the MVX load/store operations to be performed in parallel with calculations. For the future, we intend to evaluate applications with low data temporal locality and low spatial locality, such as databases and graph algorithms, in order to better understand the behavior of executing scalar instructions near the data.

5. RELATED WORK

Several studies have discussed near-data computation, generally aiming to reduce the costs related to data transfer between the processing units and DRAM. Since off-chip data movement is a major bottleneck for computer systems [15], the main goals are usually increasing performance and reducing energy consumption. Table 2 presents an overview of the characteristics of the related work presented in this section and our proposal (MVX). MVX provides general purpose processing capabilities while requiring a reasonable amount of embedded logic to operate on a high data bandwidth.

Table 2: Summary of related work characteristics.

Mechanism name	Small logic	High bandwidth	General purpose	Low data movement
IRAM [12]			•	
C-RAM [4]		•	•	•
NMP [14]			•	
LiM [16]	•	•		•
DRAMA [5]	•		•	•
NDCores [13]			•	
MVX	•	•	•	•

The second column considers the processing element details used in each proposed implementation. It can be seen that most previous work relies on a large amount of logic, because these proposals require full multi-core processors (including caches and all pipeline stages) close to the DRAM devices [13, 14], while MVX requires the addition of only the functional units themselves in the DRAM. In this way, MVX can achieve a substantial amount of parallelism while maintaining a reasonable area and energy consumption. The third column presents the characteristics related to data bandwidth of each proposal, showing that only few previous proposals consider to access the full row buffer [4, 16]. Most previous proposals use the same data bandwidth present outside the memory devices or memory modules, that is, their bandwidths are much smaller than the row buffer width.

The fourth column depicts whether the mechanism supports executing general purpose operations. Some related work can only perform very specific operations (such as FFT for the LiM mechanism [16]), which limits its adoption to specific algorithms. The last column shows how close to the main memory each proposal is implemented. In general, more complex processing elements are implemented farther away from the memory device due to integration limitations and power dissipation issues. This implies a reduction of data access bandwidth and more data movement. Summarizing our analysis, MVX achieves a good balance between logic size, operation parallelism, and energy consumption. Regarding the available design space, we consider that MVX represents an interesting alternative for near-data computing.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced the Memory Vector Extensions (MVX), a new approach to perform near-data computing that is implemented directly in the DRAM devices over a large amount of data inside the DRAM. MVX is capable of achieving high performance gains by executing vector instructions triggered by the processor. Through our experiments, we showed that MVX is capable of executing applications up to 97× faster than a 16-core processor. Moreover, when we reduced the number of functional units by up to 16× , MVX was still a better choice for applications with zero or low amount of data reuse. Energy consumption results show that our mechanism has a competitive consumption compared to the baseline. In the future, we plan to implement our mechanism in a Hybrid Memory Cube (HMC) environment, which could present different trade-offs between area and energy consumption. Furthermore, we want to evaluate more complex benchmarks with different locality patterns.

Acknowledgments

The authors gratefully acknowledge the support of CNPq.

7. REFERENCES

[1] M. Alves. *Increasing Energy Efficiency of Processor Caches via Line Usage Predictors*. PhD thesis, Universidade Federal do Rio Grande do Sul, May 2014.

[2] M. A. Z. Alves, M. Diener, F. B. Moreira, C. Villavieja, and P. O. A. Navaux. Sinuca: A validated micro-architecture simulator. In *High Performance Computation Conference*, 2015.

[3] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro*, 34(4):36–42, July 2014.

[4] D. Elliott, M. Stumm, W. Snelgrove, C. Cojocaru, and R. McKenzie. Computational ram: Implementing processors in memory. *Design and Test of Computers, IEEE*, 16(1):32–41, Jan-Mar 1999.

[5] A. Farmahini-Farahani, J. Ahn, K. Compton, and N. Kim. Drama: An architecture for accelerated processing near memory. *Computer Architecture Letters*, PP(99):1–1, 2014.

[6] Intel. Intel ® xeon phi tm coprocessor instruction set architecture reference manual. Technical report, 2012.

[7] B. Jacob, S. Ng, and D. Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2008.

[8] C. J. Lee, O. Mutlu, V. Narasiman, and Y. Patt. Prefetch-aware dram controllers. In *International Symposium on Microarchitecture (MICRO)*, pages 200–209, Nov 2008.

[9] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. The mcpat framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(1):5, 2013.

[10] S. Maleki, Y. Gao, M. Garzaran, T. Wong, and D. Padua. An evaluation of vectorizing compilers. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 372–382, 2011.

[11] Micron. 1gb: x4, x8, x16 ddr3 sdram features. Technical report, 2006.

[12] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent ram. *IEEE Micro*, 17(2):34–44, Mar 1997.

[13] S. Pugsley, J. Jestes, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. Comparing implementations of near-data computing with in-memory mapreduce workloads. *IEEE Micro*, 34(4):44–52, July 2014.

[14] M. Wei, M. Snir, J. Torrellas, and R. B. Tremaine. A near-memory processor for vector, streaming and bit manipulation workloads. Technical report, University of Illinois at Urbana-Champaign, Dept. of Computer Science, 02 2005.

[15] D. P. Zhang, N. Jayasena, A. Lyashevsky, et al. A new perspective on processing-in-memory architecture design. In *Workshop on Memory Systems Performance and Correctness (MSPC)*, page 71–73, 2013.

[16] Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. C. Hoe, L. Pileggi, and F. Franchetti. A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing. In *IEEE International 3D Systems Integration Conference (3DIC), 2013*, pages 1–7, 2013.