Large Vector Extensions inside the HMC

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

***Abstract*—One of the main challenges for embedded systems is the transfer of data between memory and processor. In this context, Hybrid Memory Cubes (HMCs) can provide substantial energy and bandwidth improvements compared to traditional memory organizations, while also allowing the execution of simple atomic instructions in the memory. However, the complex memory hierarchy still remains a bottleneck, especially for applications with a low reuse of data, limiting the usable parallelism of the HMC vaults and banks. In this paper, we introduce the HIVE architecture, which allows performing common vector operations directly inside the HMC, avoiding contention on the interconnections as well as cache pollution. Our mechanism achieves substantial speedups of up to 17.3× (9.4× on average) compared to a baseline system that performs vector operations in a 8-core processor. We show that the simple instructions provided by HMC actually hurt performance for streaming applications.**

***Keywords*—Near-data processing, vector instructions, HMC**

I. Introduction

Given the increasing memory bandwidth requirements in embedded systems, the Hybrid Memory Cube (HMC) is becoming a popular alternative to traditional memory technologies [1]. The main benefits of HMC are the low energy consumption, the high bandwidth capabilities and the processing capabilities inside the HMC [2]. According to the most recent designs, this new memory architecture is formed by 32 vaults, each consisting of 8 or 16 stacked DRAM banks [3], [4]. The memory banks are connected to the logic layer using Through-Silicon Vias (TSVs) [5]. The logic layer provides the computational capabilities (atomic instructions) and also substitutes the memory controller in the processor, as it is responsible for handling the requests and for sending the DRAM signals to the banks.

Although multi-channel DDR and HMC memories can provide high data bandwidths, the traditional way of performing the computation inside the processor core is inefficient for stream applications with low data reuse. The first reason for this inefficiency is that multiple processor cores are required to achieve the maximum performance from the memory, in order to use the internal bank, channel or vault parallelism. The second reason is that such applications can not benefit from the caches due to their low data reuse.

HMC designers have provided a limited number of atomic instructions that can be triggered by the processor. However, the HMC Instruction Set Architecture (ISA) consists of update instructions that are restricted to immediate operands, with a small data width of 16 bytes. Most applications with a streaming behavior can make use of processor SIMD extensions (such as SSE, AVX, and NEON), but the data transfers required for such operations are inefficient. Furthermore, the current HMC ISA is not suitable for such applications. Therefore, we introduce in this paper the HMC Instruction Vector Extensions (HIVE) inside the HMC memories. HIVE is designed to perform logical and arithmetic operations inside the HMC over large amounts of data with a single instruction. In contrast to previous proposals, HIVE does not require full processing cores to provide general purpose computing capabilities inside the HMC.

HIVE has the following main features:
Vector processing near memory: HIVE performs vector operations inside the memory over up to 8 KB of contiguous data with a single HIVE instruction.
HMC ISA extension: HIVE extends the HMC ISA, adding vector instructions that are suitable for vectorizable applications, covering a wide range of possible usage scenarios.
Reduced number of cores: HIVE reduces the number of cores necessary to process large data streams by avoiding huge data transfers and vector operations inside the processor.
Increased cache efficiency: HIVE improves cache efficiency by sending vector instructions to be executed inside the memory, without storing data that will not be reused in the caches.

In our simulations, using our HIVE instructions executing the applications with the biggest input sizes, we show that executing HIVE operations with a single thread performs on average 20.6× faster than 8 cores executing SSE instructions, connected to a 4-channel DDR 3 (SSE+DDR), and 9.4× faster than with SSE+HMC. However, emulating the standard HMC ISA instructions, we show that such instructions provide lower performance for stream applications, compared to the normal SSE instructions, due to the low data width.

II. Related Work

Near-data processing is becoming a promising technique to reduce data transfers [6]. Adding processing capabilities inside or close to the DRAM has a high potential for performance and energy efficiency improvements by avoiding huge and inefficient data transfers. Considering SDR and DDR memories, several techniques were proposed to process data inside these memories [7], [8], [9], [10], [11], [12], by increasing the number of internal buses together with full processor cores or integrating the processing logic into the DRAM row-buffers. However, integrating processing logic inside the traditional SDR or DDR memories is a challenging task, because logic circuits fabricated in DRAM-optimized process technology are much slower than similar circuits in a logic-optimized process technology [13].

Other proposals integrate reconfigurable logic or full processors outside the memory, relying on wider data buses of TSVs to transfer data between the memory device and the

The authors gratefully acknowledge the support of CNPq and CAPES.

processing logic [14], [15]. Such proposals limit their processors to access only the data available on the devices to which they are attached. Because of the limitations, these mechanisms require a very specific memory mapping of the application's data by the OS in order to work with multiple channels. Finally, some proposals [16], [17] take advantage of 3D integration technologies to insert a custom logic in a layer of DRAM devices. These circuits are custom ASICs (Application Specific Integrated Circuits) and are able to accelerate only specific applications.

For the new HMC, a logic layer is already available inside the memory module, showing that the integration challenge has been overcome. Such a logic layer embeds the memory controller as well as a few functional units capable of performing simple atomic update instructions [3]. One of the first proposals to include general purpose processors inside the HMC, NDCores [18], embedded 512 cores in the HMC, split among multiple modules. Their objective was to make better use of the full bandwidth provided by the vaults in order to compute map/reduce applications inside the memory. Another work that proposes to include full processors inside the HMC, Tesseract [19], aims to speed up the processing of graphs. Such processors are also integrated together with the HMC vaults. The recent Active Memory Cube [20], propose a pipeline organization with an instruction buffer inside the logic layer of the HMC. These proposals introduce fully pipelined processors with cache memories or internal data forwarding, while we only require functional units and a register bank, greatly simplifying the implementation. Moreover, the cache memories are inefficient for many streaming applications.

In order to provide better support for near-data computing, Ahn et al. [21] present an automated way to detect if a piece of code should run near the cache memories (in case it presents temporal locality) or near the main memory (in case it presents spatial locality). Several applications were implemented using different ISAs in order to evaluate the benefits from their mechanism. Such a system is orthogonal to ours and could be used together with HIVE in order to obtain the maximum performance from our mechanism.

III. HIVE: Performing Vector Operations in HMCs

The main focus of this work is to introduce general-purpose vector operations inside the HMC, to take advantage of HMC's intrinsic parallelism to enhance stream-like computations, while also reducing the data movement between processor and main memory. Our mechanism, HMC Instruction Vector Extensions (HIVE), obtains data directly from the multiple memory vaults, to take advantage of the high parallelism inside the HMC. We implement an instruction sequencer, a register bank and a set of vector functional units inside the HMC. These new functional units answer to the HIVE instructions that we introduce, while the register bank enables the memory to operate in parallel with our mechanism. Thus, computation can be overlapped with memory accesses. In this section, we introduce HIVE, discussing the required processor and HMC 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.

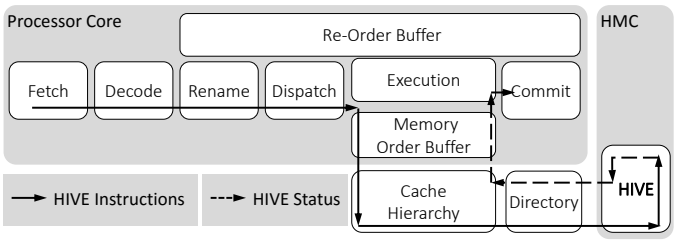


Fig. 1: Datapath of a HIVE instruction.

A. Overview of HIVE

The new HIVE instructions are inserted inside the application code by the compiler, in a similar way that SSE, AVX and NEON are used. When executing the code, HIVE instructions are fetched and decoded by the processor and then sent to the main memory to be executed inside the HMC, avoiding expensive data transfers between the memory and processor. Figure 1 depicts an overview of the datapath of a HIVE instruction from the processor to the HMC.

We consider version 2.0 of the HMC specification [3], which is formed by 32 memory vaults and at least 8 DRAM banks per vault, each bank with a row buffer of 256 bytes. Two main data interleaving possibilities can be used for HMC memories [22]. We can store adjacent cache lines inside the same row buffer, and just after using the full row jump to the next memory vault. The second mode is to store adjacent cache lines in different memory vaults, and after using all the memory vaults jump to the next bank, making better use of the DRAM bank parallelism among the vaults. In our experiments, we used the second interleaving mode, because it results in the best performance overall [22].

HIVE instructions operate over 8 KB of data. We selected this size since it allows using the full parallelism inside the HMC vaults and banks. Other sizes can be easily implemented as well. With this configuration, each HIVE instruction uses 3 internal registers to perform up to 2,048 operations of the same type (4-byte integer/FP operations, compared to 16 operations in AVX-512). In our model, HIVE supports all of the ARM NEON operations (integer and FP).

B. Binary Generation

In order to use the HIVE 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 new vector instructions. The automatic vectorization techniques (similar to the ones present in current compilers [23]) from the compiler are extended to make usage of our wider operations. To avoid usage conflicts on the HIVE registers, a sequence of HIVE instructions needs to be wrapped by HIVE's lock and unlock instructions. These instructions will perform a lock in the HIVE structures only, and the lock enables operations from a specific thread, unlocking it whenever an unlock is executed. Normal memory access requests issued by non-HIVE instructions can still be serviced while HIVE is locked. Lock mechanism is discussed in Section III-D.

C. Processor Modifications

In the processor, we require an ISA extension to provide our HIVE instructions. The HIVE instructions use a new

register bank inside the HMC to perform operations. The HIVE instructions pass through the pipeline in the same way as a memory load operation. HIVE instructions that do not require memory addresses, such as HIVE lock and unlock, will bypass the Address Generation Unit (AGU) and wait to be transmitted inside the Memory Order Buffer (MOB). All HIVE instructions are placed inside the MOB to be delivered to the memory subsystem. These instructions wait inside the MOB for an answer from the HMC, which returns the status of the operation as successful or raises exceptions. The processor uses these instructions' status to control execution flags, such as overflow and not-a-number, among others.

HIVE instructions that perform loads and stores work with virtual addresses. Therefore, the addresses 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 HIVE instruction will operate. Although we implement HIVE in an out-of-order processor, in-order processors could also be modified to work with the HIVE instructions. It is important to note that all modifications inside the processor are also required to make use of the new ISA present in the HMC specification. Thus, we expect that only minor changes inside the processor are required to support the HIVE instructions.

D. HMC Modifications

When HIVE receives a HIVE lock instruction, it has to lock the mechanism to operate only for the thread that requested the lock. In case the memory is already locked, the lock instruction is sent back to the requester with a fail status. When a lock is granted, the HIVE 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. This locking system can also be used to power gate or clock gate all HIVE resources after a certain period of time, reducing the energy overhead during idle periods. Normal memory access requests (both reads and writes) can still be serviced while HIVE is locked, such that other threads that do not use HIVE can continue to execute.

To perform vector instructions inside the DRAM, we require three main logic additions to the HMC, a HIVE sequencer, a register bank, and the vector functional units. Figure 2 illustrates HIVE inside an HMC 2.0 module, with 32 vaults and 8 banks per vault. HIVE can be easily adapted to different HMC layouts (such as different numbers of banks per vault or row buffer sizes). In our mechanism, the HIVE sequencer handles the instructions in-order until they can be executed and sends the status after the instructions are executed. During HIVE load/store operations, the sequencer is also responsible for broadcasting the 8 KB request split into 128 sub-requests of cache line size (64 bytes). Each sub-request is sent to its respective vault controller.

The additional register bank inside HIVE is used to store the sub-requests coming from any vault/bank inside the HMC. Each register can handle 128 positions of 64 bytes each (8,192 bytes in total). Thus, sub-requests can be issued to different

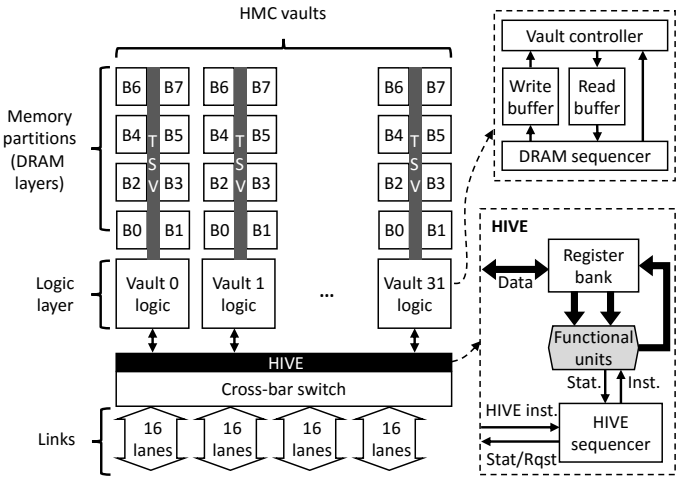


Fig. 2: HMC module with our mechanism architecture.

vaults/banks to increase performance. HIVE interacts with the DRAM devices only during load and store operations by copying data to and from the HIVE registers. Therefore, our mechanism does not require new DRAM signals.

HIVE executes instructions in-order, but its functional units act as a restricted data-flow processor. A given operation can start as soon as the registers are ready. To support this data-flow, we use a flag associated with each register that indicates if the operand is ready. Each HIVE instruction needs to erase this flag for its destination register, and re-enables it whenever the instruction becomes ready. This system allows the DRAM to open rows from different banks in parallel, and also ensures that once a HIVE instruction requires operands that are not ready yet, execution will stall. When registers are ready, the functional units operate in several steps (HIVE cycles) to process the entire register. The number of steps depends on the number of functional units. We explore the performance trade-offs of the number of functional units in Section IV.

All functional units operate at the frequency of the HMC vault controller. After completion, each HIVE instruction sends a status to the processor, such that our instructions behave similarly 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-512 instruction set, 17 bits are enough to provide the information regarding the operation status [24]. For the evaluation of our mechanism, we consider an acknowledgment of 64 bits in order to correctly simulate the impact of the transmission of the status bits on the final performance. Note that the number of status bits does not increase linearly with the operation size. For example, between AVX-128 and AVX-512, only two bits were added, because only a single flag is raised if one or more sub-operations cause an exception.

IV. Experimental Evaluation of HIVE

This section presents the simulation environment, the application kernels and the evaluation results of HIVE. To simplify the explanations, we refer to SSE+DDR and SSE+HMC when executing an application that uses SSE instructions in a system with DDR or HMC memory modules respectively. We refer to

HMC$_{inst}$+HMC and HIVE+HMC when talking about applications that use the HMC ISA or our HIVE instructions in a system with HMC memory modules.

A. Methodology

1) Configuration Parameters and Baseline: To evaluate our architecture, we used a cycle-accurate simulator [25]. The simulation parameters are inspired by Intel's Atom processor with the Silvermont microarchitecture. This processor only supports up to 2 memory channels. To build a possible future scenario for comparison, we added support for 4 memory channels in the baseline. Table I shows the simulation parameters. For the baseline, we simulate 128-bit SSE instructions.

TABLE I: Baseline and HIVE system configuration.

OoO Execution Cores 8 cores @ 2.0 GHz, 2-wide out-of-order; Buffers: 24-entry fetch, 32-entry decode; 32-entry ROB; MOB entries: 10-read, 10-write; 1-load and 1-store units (1-1 cycle); 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); Branch predictor: Two-level PAs. 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: 8-request, 8-write, 8-eviction; Stride prefetcher: 1-degree;
L2 Cache 1 MB shared for every 2 cores, 16-way, 4-cycle; 64 B line; LRU policy; MSHR size: 32-request, 32-write, 16-eviction; MOESI protocol; Inclusive; Stream prefetcher: 2-degree;
DDR3-1333 Module 8 DRAM banks, 8 KB row buffer (1 KB / device); On-core DDR controller; 4 GB total size; DRAM@166 MHz; 4-channels; 8 B burst width at 2:1 core-to-bus freq. ratio; Open-row first policy; DRAM: CAS, RP, RCD, RAS and CWD latency (9-9-9-24-7 cycles);
HMC v2.0 Module 32 vaults, 8 DRAM banks/vault, 256 B row buffer; 4 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): 1-alu, 3-mul. and 20-div. int. units; Latency (cycles): 5-alu, 5-mul. and 20-div. fp. units; Register bank with 8 registers of 8,192 bits each;

2) Application Kernels: We used 6 different application kernels. Three kernels (vector sum, stencil, and matrix multiplication) use floating point operations, the other three (search for immediate, memory reset and memory set) use integer operations. The assembly code of the kernels was obtained from the *gcc* compiler using SSE auto-vectorization and was then manually adapted to use HMC or HIVE instructions.

The floating point application kernels are: A **vector sum** using three vectors of up to 64 MB each. A **stencil** with 5-points over a matrix of up to 64 MB, adding up the 5 neighboring elements, multiplying the result by two and then storing each result in an output matrix. A **matrix multiplication** using three square matrices (2 source and 1 result) of up to 9 MB each. These three kernels use floating-point operations and were implemented using SSE or HIVE instructions. 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 caches. The matrix multiplication application has a high amount of data reuse, thus benefiting highly from the caches.

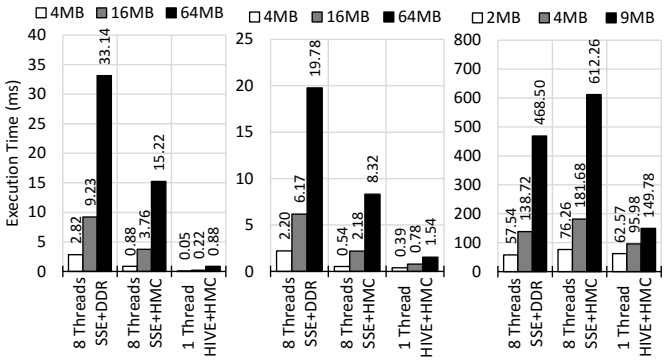
The integer application kernels are: A **search for immediate** over a vector of up to 64 MB. A **memory reset** over a vector of up to 64 MB using logical *and* operations to set all positions to zero. A **memory set** over a vector of up to 64 MB using logical *and* and *or* operations to set all position to a fixed value. These three integer kernels were implemented using SSE, HMC or HIVE instructions, enabling us to compare the HIVE instructions performance to the HMC ISA. Analyzing the HMC ISA, the search for immediate represents a case where the HMC ISA needs only to perform one load and one single operation (`cmp`) for every 16 bytes. The memory reset performs one load, one operation (`and`) and one update. The memory set requires two loads, two operations (`and` + `or`) and two updates every 16 bytes. Although a memory reset and set could be implemented using a simple memory store operation, we force its implementation using logical operations in order to observe the performance of the HMC ISA.

B. Results

1) Floating-Point Kernels: Figure 3 presents the results for the floating point kernels, showing the execution time for SSE+DDR and SSE+HMC executing with 8 cores and HIVE+HMC running with a single thread. Compared to SSE+DDR, on average SSE+HMC performs 2.6× faster for vector sum, 3.1× for stencil and 24% slower for matrix multiplication. The performance degradation happens due to the reduced row buffer size and closed row policy used for HMC, requiring to open multiple row buffers, while DDR 3 can service multiple requests from the same row.

HIVE+HMC performs on average 16.8× faster for vector sum, 3.2× faster for stencil and 2.4× faster for matrix multiplication compared to SSE+HMC. Moreover, as the input size of the application increases, HIVE+HMC becomes more attractive, because the data stops fitting into the caches. For the largest input size evaluated, we can observe HIVE+HMC gains of 17.3× for vector sum, 5.4× for stencil and 4.1× for matrix multiplication compared to SSE+HMC.

It is important to note that these results use a single core to issue HIVE instructions and 8 cores for SSE+HMC, which means that HIVE achieves a high performance with only a single core. If we compare HIVE+HMC to SSE+HMC both with a single thread only, executing the biggest input size,

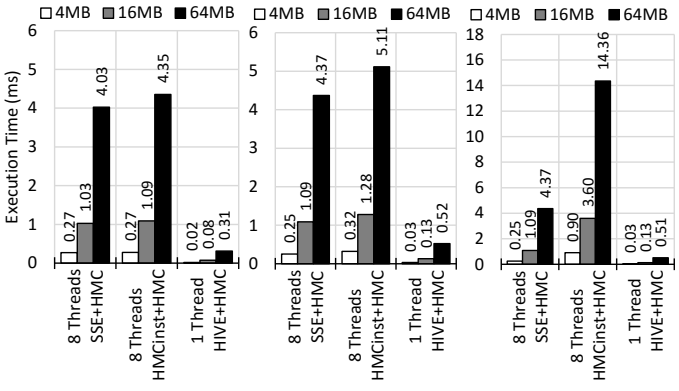


(a) Vector sum (b) Stencil (c) Matrix mul.

Fig. 3: Execution time for SSE+DDR, SSE+HMC and HIVE+HMC for different inputs of the floating-point kernels.

our mechanism achieves average gains of 58× (with 99× for vector sum, 34× for stencil and 42× for matrix multiplication). Such results are possible because HIVE is capable of using all the parallelism inside the HMC vaults and banks with a single instruction requiring thus less threads to make usage of all the parallelism.

2) Integer Kernels: Figure 4 presents the results for the integer kernels, showing the execution time for SSE+HMC, HMC$_{inst}$+HMC and HIVE+HMC, executing in a system with HMC memory. HMC$_{inst}$+HMC performed on average 30% worse than the SSE+HMC, showing that major changes are required in HMC to prevent multiple row accesses during streaming applications using the HMC ISA. Our simulator implements a smart closed row policy, which prevents the row to be closed if it is detected that another instruction wants to operate in the same row. However, this is not enough to prevent the overhead of not operating on the cache line size. We believe that a bigger Miss-Status Handling Registers (MSHR) inside the caches and inside the processor are required to expose more HMC instructions to the vaults. HIVE+HMC achieved consistent gains of 10× on average for search immediate, memory reset, and memory set compared to SSE+HMC.



(a) Search immediate (b) Memory reset (c) Memory set

Fig. 4: Execution time for SSE+HMC, HMC$_{inst}$+HMC and HIVE+HMC for different inputs of the integer kernels.

3) Comparing Atom and Sandy Bridge: Considering that our model of the Atom processor has a small number of entries in the MOB and the MSHR, we performed a second experiment with a model of a Sandy Bridge machine, to check if the gains of HIVE are consistent even when considering a high performance system. A list of parameters for the Sandy Bridge model is presented in Table II. The HMC and HIVE parameters are the same as before.

Figure 5 shows the results for the biggest input size of all kernels on the Atom and Sandy Bridge machines. We can observe that HIVE+HMC performance is independent of the processor and cache hierarchy, with a maximum difference of 29% between HIVE using Atom or Sandy Bridge. However, the average difference between our mechanism and the SSE+HMC going from Atom to Sandy Bridge decreases from 9.4× to 4.8×, due to the higher parallelism extraction of SSE+HMC.

The maximum bandwidth achieved for Atom processor executing vector sum was 7.7 GB/s for SSE+DDR, 16.8 GB/s for SSE+HMC and 290.7 GB/s for HIVE+HMC. For the Sandy-

TABLE II: Sandy Bridge system configuration.

Processor Cores: 8 cores @ 2.0 GHz, 32 nm; 4-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 16 MB (8-banks), 2 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;

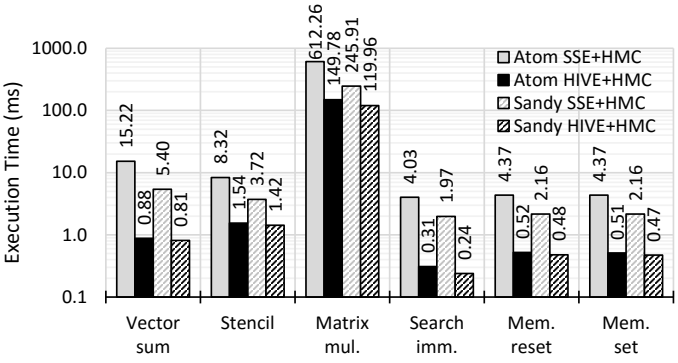


Fig. 5: Execution time of SSE+HMC and HIVE+HMC for the Atom and Sandy Bridge systems.

Bridge machine, the maximum bandwidth was 11.1 GB/s for SSE+DDR, 47.4 GB/s for SSE+HMC and 315.9 GB/s for HIVE+HMC. We can notice that the bandwidth variations between the evaluated systems are consistent with the achieved speedups. Although HMC improves the memory bandwidth considerably in both architectures, overall SSE+HMC performance is starting to be limited by the processor throughput and its complex cache hierarchy. This indicates that future processors should be designed to take into account the new HMC architecture to fully leverage its potential. HIVE overcomes this limitation, using up to 98% of the theoretical bandwidth (320 GB/s [3]) of the DRAM inside HMC.

4) Varying the Number of Functional Units: Figure 6 evaluates, using the Atom system, the speedup of HIVE with different numbers of functional units (FUs), varying the number of parallel operations between 2,048 and 128 (1/16). When implementing fewer FUs, multiple iteration are needed for each operation. In this case, the latency to perform a single operation would be multiplied, depending on the number of functional units. With this experiment, we intend to show the break-even point for the HIVE+HMC single-thread implementation when compared to SSE+HMC with 8 cores.

On average, the speedup decreases from 9.4× to 5.8× when reducing the number of FUs from 2,048 to 128. Applications with lower data reuse are less affected by the reduction of FUs, because our architecture enables the computation to be overlapped with memory accesses. However, the matrix multiplication, which has a high amount of data reuse, was the only application with a break-even point, which occurred between 512 and 256 FUs. These results show that HIVE

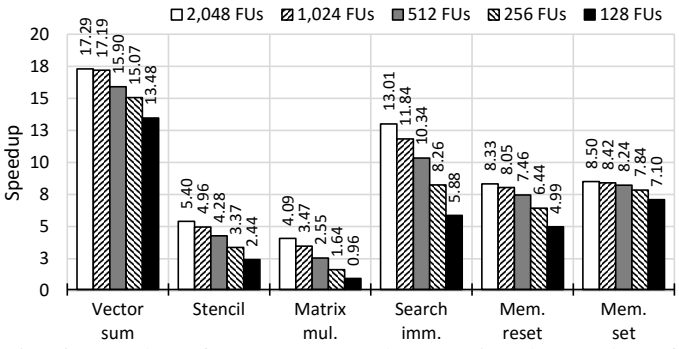


Fig. 6: Speedup of HIVE+HMC when varying the number of functional units compared to SSE+HMC.

presents a wide design space with sustained performance gains.

5) Area Overhead: We calculate the area required by HIVE with the McPAT tool [26], modeling the functional units and the register bank of our mechanism. For the functional units, we consider a set of integer units plus ARM NEON units to support integer and floating point instructions. A subset of operations or a reduced number of functional units can be selected in order to reduce the area overhead. The total area occupied by HIVE to fully expose the parallelism of the HMC vaults (2,048 functional units) results in an area of 377mm^2 when modeled at 45nm.

V. Conclusions

The large data requirements of parallel applications demand novel solutions to keep on increasing performance. In this paper, we presented HMC Instruction Vector Extensions (HIVE), an implementation of general-purpose vector operations inside Hybrid Memory Cubes (HMCs) that helps to mitigate the latency and bandwidth bottlenecks of traditional memory and interconnection techniques. HIVE performs common vector operations directly in the HMC memory vaults and requires only small changes to the processor and application. Compared to HMC memories and traditional vector operations implemented in the processor, HIVE improved the performance by up to 17.3× (9.4× on average), with gains depending on the amount of data reuse of the application. We also showed that the simple operations provided by HMC are not wide enough to result in significant speedups, enabling HIVE to improve performance by up to 28× (17.2× on average) compared to the HMC ISA.

References

[1] Altera, "Hybrid Memory Cube Controller IP Core User Guide," 2015, https://www.altera.com/solutions/technology.html.

[2] K. Khalifa, H. Fawzy, S. El-Ashry, and K. Salah, "Memory controller architectures: A comparative study," in *Int. Design and Test Symp.*, 2013.

[3] Hybrid Memory Cube Consortium, "Hybrid Memory Cube Specification Rev. 2.0," 2013, http://www.hybridmemorycube.org/.

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

[5] 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.

[6] R. Balasubramonian, J. Chang, T. Manning *et al.*, "Near-Data Processing: Insights from a MICRO-46 Workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, July 2014.

[7] A. Saulsbury, F. Pong, and A. Nowatzyk, "Missing the Memory Wall: The Case for Processor/Memory Integration," in *Int. Symp. on Computer Architecture*, 1996.

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

[9] N. Bowman, N. Cardwell, C. Kozyrakis *et al.*, "Evaluation of existing architectures in IRAM systems," in *Int. Symp. Computer Architecture - Workshop on Mixing Logic and DRAM*, 1997.

[10] D. Elliott, M. Stumm, W. Snelgrove, C. Cojocaru, and R. McKenzie, "Computational RAM: Implementing Processors in Memory," *Design and Test of Computers*, vol. 16, no. 1, pp. 32–41, Jan-Mar 1999.

[11] M. A. Z. Alves, P. C. Santos, M. Diener, and L. Carro, "Opportunities and challenges of performing vector operations inside the dram," in *Int. Symp. on Memory Systems*, ser. MEMSYS '15, 2015, pp. 22–28.

[12] M. A. Z. Alves, P. C. Santos, F. B. Moreira, M. Diener, and L. Carro, "Saving memory movements through vector processing in the dram," in *Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, 2015, pp. 117–126.

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

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

[15] A. Farmahini-Farahani, J. Ahn, K. Compton, and N. Kim, "DRAMA: An Architecture for Accelerated Processing near Memory," *Computer Architecture Letters*, no. 99, 2014.

[16] Q. Zhu, B. Akin, H. E. Sumbul *et al.*, "A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing," in *Int. 3D Systems Integration Conf.*, 2013.

[17] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating Sparse Matrix-Matrix Multiplication with 3D-stacked Logic-in-Memory Hardware," in *High Performance Extreme Computing Conf.*, 2013.

[18] S. 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.

[19] 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.

[20] R. Nair, S. 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, pp. 17:1–17:14, March 2015.

[21] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture," in *Int. Symp. on Computer Architecture*, 2015.

[22] P. Rosenfeld, "Performance exploration of the hybrid memory cube," Ph.D. dissertation, University of Maryland, 2014.

[23] S. Maleki, Y. Gao, M. Garzaran, T. Wong, and D. Padua, "An Evaluation of Vectorizing Compilers," in *Int. Conf. on Parallel Architectures and Compilation Techniques*, 2011.

[24] Intel, "Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual," Tech. Rep., 2012.

[25] M. A. Z. Alves, M. Diener, F. B. Moreira *et al.*, "SiNUCA: A Validated Micro-Architecture Simulator," in *High Performance Computation Conference*, 2015.

[26] 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.