A Compiler for Automatic Selection of Suitable Processing-in-Memory Instructions

Hameeza Ahmed, Paulo C. Santos[†], João P. C. Lima[†], Rafael F. Moura[†],
Marco A. Z. Alves[‡], Antônio C. S. Beck[†], Luigi Carro[†]
Dep. of Computer and Information Systems Eng. – NED University – Karachi, Pakistan
[†]Informatics Institute – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
[‡]Department of Informatics – Federal University of Paraná – Curitiba, Brazil
Email: hameeza@neduet.edu.pk [†]{pcssjunior, jplima, rfmoura, caco, carro}@inf.ufrgs.br [‡]{mazalves}@inf.ufpr.br

***Abstract*—Although not a new technique, due to the advent of 3D-stacked technologies, the integration of large memories and logic circuitry able to compute large amount of data has revived the Processing-in-Memory (PIM) techniques. PIM is a technique to increase performance while reducing energy consumption when dealing with large amounts of data. Despite several designs of PIM are available in the literature, their effective implementation still burdens the programmer. Also, various PIM instances are required to take advantage of the internal 3D-stacked memories, which further increases the challenges faced by the programmers. In this way, this work presents the Processing-In-Memory cOmpiler (PRIMO). Our compiler is able to efficiently exploit large vector units on a PIM architecture, directly from the original code. PRIMO is able to automatically select suitable PIM operations, allowing its automatic offloading. Moreover, PRIMO concerns about several PIM instances, selecting the most suitable instance while reduces internal communication between different PIM units. The compilation results of different benchmarks depict how PRIMO is able to exploit large vectors, while achieving a near-optimal performance when compared to the ideal execution for the case study PIM. PRIMO allows a speedup of 38× for specific kernels, while on average achieves 11.8× for a set of benchmarks from PolyBench Suite.**

***Keywords*—Compiler, Processing in Memory, Near-data computing, Vector instructions, SIMD, 3D-Stacked memories**

I. Introduction

3D-stacking technology has enabled cost-effective integration of memory and logic units to support Processing-in-Memory (PIM) capabilities [1]. One promising idea in PIM concept is to improve performance and reduce energy consumption by moving costly operations as close as possible to the memory, minimizing data movements through the narrow buses existent between Central Processing Unit (CPU) cores and main memory [2], [3].

From the hardware point-of-view, an ideal PIM architecture must take advantage of the internal bandwidth available on 3D-stacked memories, which can achieve up to 320GB/s in devices such as High Bandwidth Memory (HBM) [4] and Hybrid Memory Cube (HMC) [5]. Moreover, an efficient PIM logic design must also be concerned with area and power dissipation constraints inherent to 3D-stacked memories [6], [7], [8]. To fulfill both requirements, the logic PIM design must be centered on the Functional Units (FUs) and must have reduced complexity in Instruction Level Parallelism (ILP) mechanisms [2], [8]. Nonetheless, the FU-centered PIM design requires from a compiler to infer as many Single Instruction Multiple Data (SIMD) instructions as possible to amortize the drawback of having a control unit with limited ILP capabilities. Moreover, one can only obtain the maximum advantage of exploiting a PIM device if there is no disruption in the software design cycle, hence, a compiler to seamlessly allow several PIM features is mandatory. The compiler must deal with automatic code offloading to the PIM units, and also must efficiently exploit the PIM architecture concerning the optimal usage of vector units and communication among them.

This work presents Processing-In-Memory cOmpiler (PRIMO), the first compiler to automatically generate code for PIM-based systems. This means that the original program source code will be automatically distributed between the host processor and PIM instances, without any code annotation from the programmer, external tools, or special libraries, optimizing the huge internal bandwidth utilization. As a PIM device can be composed of several SIMD units or Vector Processor Units (VPUs) distributed along 3D-stacked memory *vaults* [2], [3], the communication between these units can reduce performance and requires proper treatment. In this way, a new VPU selection technique has been proposed in the register allocation stage of PRIMO, which contributes to the reduction of unnecessary communication between PIM instances.

The contributions proposed in this work are as follows:

- To the authors' best knowledge, the first compiler that is able to automatically exploit a PIM architecture by detecting vector instructions of different sizes from the source code;
- An optimization technique that selects the appropriate vector size to fine tune the PIM hardware use, thus reducing the PIM instructions required to execute the original program source code;
- A new technique for selection of relevant Vector Processor Unit (VPU), so that communication of partial data results within the memory is also minimized, and hence reducing execution time of the PIM hardware;

The proposed compiler can support different PIM architectures, hence it is tested using a state-of-the-art PIM architecture as case study [3]. PRIMO is experimented by compiling a set of kernels and PolyBench benchmarks, and by simulating the baseline hardware and target PIM on the GEM5 simulator [9]. For the set of kernels, PRIMO can allow a speedup of up to

[1]This study is partially financed by CAPES(Code 001), CNPq and FAPERGS.

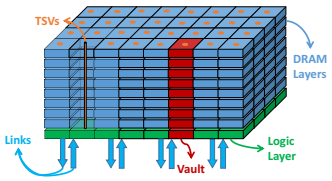

Fig. 1: HMC layout comprising of eight DRAM layers and a base logic layer connected by Through-Silicon Vias (TSVs) and vertically organized in *vaults*.

15.8×, showing the capacity of automatic offloading instructions of the compiler. For PolyBench, our compiler design is able to allow a speedup of up to 18.4×.

The rest of the paper is organized as follows: Section II gives a brief background about PIMs and compilers. Section III highlights the previous work in this field. The methodology behind PRIMO is presented in Section IV. Section V shows the performance of PRIMO with respect to a case study, which is followed by final conclusion and future work that are presented in Section VI.

II. Background

In this section, an overview of fundamental aspects of 3D-stacked memories is presented along with the description of PIM architectures. Additionally, the role of compilers in PIM architectures is highlighted.

A. 3D-stacked DRAM and Processing-in-Memory Architectures

High-density 3D memories rely on multiple stacked DRAM dies to provide high bandwidth and capacity, in order to meet the demand for today's system workloads.

As described in the latest specification [5], the HMC is a package containing up to eight Dynamic Random Access Memory (DRAM) layers and one logic layer stacked together and connected through TSV, as shown in Figure 1. Within each cube, memory is organized vertically into *vaults*, which consist of a group of corresponding memory portions on different DRAM layers combined with a *vault* controller within the logic layer. Up to 32 *vaults* can compose an HMC module, each one with an independent memory controller. Moreover, the HMC specifies its own PIM commands to compute atomic instruction limited to operands of 16 bytes. Due to the independent access in different *vaults* and simple processing capabilities in the logic layer, HMC is broadly chosen as the target device in PIM research works [10], [11], [12], [3].

Accordingly to recent works [8], PIM architectures can be categorized based on the main processing element within the logic layer: General Purpose Processor (GPP)-like cores and dedicated logic circuits. GPP-like PIM cores do not require effort to the programmer or even compilers to generate binary code since they utilize the same programming interface inherent to the GPP that composes the logic layer. However, they are neither capable of taking advantage of the available bandwidth nor fit into the area and power budget constraints related to 3D-stacked memories [8]. Alternatively, PIM architectures built on dedicated logic circuits design not only fit into 3D-stacked memories budget constraints but also require some level of programmability to control them. This opens up a challenge and opportunity for compiler researches since they have to efficiently perform offloading of PIM commands and choose the better way to use the dedicated logic circuit resources.

B. Compilers for PIM Architectures

In the PIM scenarios, the prime responsibilities of a compiler includes automatic selection of the processing units placed in multiple 3D-stacked memory *vaults*, data reorganization, reduction of *LOAD/STORE* operations, and data reuse optimizations [13].

Considering that PIM designs centered on FUs can exploit the internal 3D-stacked memory bandwidth, the natural choice is to implement a large quantity of FUs and VPUs, as supported in recent studies [2], [13], [3], [8] Hence, the compiler is expected to automatically make use of the largest vector possible to take advantage of the available resources.

Although existing compilers provide an automatic vectorization feature, it loses in efficiently exploring the architecture by fixing the vector width of the operands [14], which can either underutilize resources or waste vectorization opportunities. These issues are not so clear with small SIMD units placed in GPP, but if the compiler misses vectorization opportunities on large SIMD units placed in PIM, the penalty regarding performance will be more substantial. Additionally, the resource misuse reduces energy efficiency (for example, by increasing memory access, cache misses) which is an essential goal in any environment.

III. Related Work

As described in the Section II-B, the compiler must tackle the instruction offloading decision, as well as automatically optimize the code for a given architecture. In the literature, several works deal with PIM offloading decisions, while others deal with hardware exploitation. However, to the best of our knowledge, no previous study has implemented a complete compiler support, including both the offloading and architecture-oriented optimization steps, as PRIMO does.

Regarding offloading decisions, [15] presents Compiler-Assisted technique for enabling InstRuction-level Offloading of PIM (CAIRO). In CAIRO, the PIM offloading candidates are decided by the number of cache misses, bandwidth savings, and the overhead of host atomic instructions respectively. These decisions are taken offline because it requires a cache profiler for recording the traces of previous execution of applications. Further, each version of an application needs to be compiled, profiled, and then analyzed by CAIRO to, then, include HMC-atomic operations. Although CAIRO can double the performance and prevent performance slowdown caused by incorrect offloading, it is limited to scalar HMC-atomic operations. Also, CAIRO do not consider the advantage of SIMD parallelism in the analysis, even though most of the benchmarks used in the evaluation are focused on Graphics Processing Unit (GPU). In PRIMO, both the offloading and exploitation of FUs happens automatically at compile time, without using third party profilers or increasing design time.

Similarly, the work of [16] presents Transparent Offloading and Mapping (TOM), which is a compiler-based solution that allows computation offloading to multiple 3D-stacked memories in a GPU-based system. This approach is concerned about the amount of code that will be sent to the main memory of a

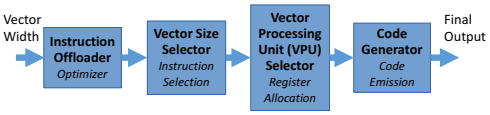

Fig. 2: Structure of Processing-In-Memory cOmpiler (PRIMO)

GPU. In TOM, the instruction blocks that have the potential for maximum memory savings are initially identified. Then, it is decided whether the selected candidates are required to be offloaded or not based on runtime system conditions. Although TOM shows an effective approach to enable near-data processing in GPU systems, still the programmer needs to write a Compute Unified Device Architecture (CUDA) annotated code which is avoided by PRIMO. Also, PRIMO incorporates both the offloading and exploitation of PIM resources whereas TOM is only focused on offloading.

Further, in [17], a new compiler pass is presented for detecting code dealing with bandwidth critical data to allocate data to HBM. The allocation calls are transformed into specific HBM allocation calls which are transparent to the user. This solution is automatic in the manner that *malloc* calls are changed to *memkind_alloc* for memory bandwidth bound codes. Despite [17] is intended for automatic data organization, its technique relies on special functions, which requires programmer attention. Moreover, this work is not concerned with instructions offloading, and hence limits a more generic PIM interface.

Finally, for exploiting Active Memory Cube (AMC) PIM hardware, a compiler is presented in [18], and it adopts a directive-based approach. A program annotated to run on parallel systems with OpenMP 4.0 directives is analyzed to discover which parts will run on the AMC. Also, AMC can exploit complete vector and pipeline features of the lanes in order to derive an effective schedule of instructions for each lane. The AMC compiler can deliver up to 71% better FU utilization than the hand-written version of some scientific kernel, such as matrix multiplication, but also generates large instruction footprints. Similar to AMC, PRIMO is intended for hardware exploitation but without the burden of using a *pragma* or directive-based approach, and its instruction footprint is comparable to the footprint of AVX-512 code.

IV. The PRIMO concept

PRIMO is designed for *Fixed* type PIMs, where the main focus remains the exploitation of vector units present in these specific machines. PRIMO provides complete compiler support for PIM architectures by enabling offloading decisions, suitable vector size and vector FUs selection, and code emission in an automatic manner.PRIMO is developed using the Low Level Virtual Machine (LLVM) compiler tool [19]. Figure 2 shows the main modules that comprises PRIMO. Besides the Intermediate Representation (IR) code, vector width parameter is passed as an input to PRIMO modules and in the end PIM code is emitted as final output. For implementing these modules, both the back-end and middle-end of the compiler are modified while front-end is used without any changes. The *Instruction Offloader* component is implemented in the optimizer phase, while the *Vector Size Selector* acts in the instruction selection phase, and relevant VPUs are selected by *VPU Selector* module in register allocation phase. Finally, *Code Generator* functionality is enabled in code emission phase in PRIMO's back-end.

The PRIMO tool provides an abstract and generic implementation such that for any target PIM, the tool can be extended as a function of the architecture specific features. Traditionally, the PIM is treated as an accelerator, however due to the nature of the case study PIM (presented in Section V), the hybrid code approach can be adopted. In this way, the PIM is seen as an extension of host's FUs, allowing host to trigger PIM instructions. Therefore, x86 back-end is extended to add PIM support, like an added extension to the present Advanced Vector Extensions (AVX) instruction set. This approach allows the direct interpretation and communication with the PIM accelerator, as the target PIM requires hybrid code (host + accelerator code) [20], [3]. Also, PRIMO can control the generation of PIM code, by allowing a simple *PIM-Enabled* parameter to be used at compile-time. With this approach, code annotations such as *pragmas* and compiler *directives* are totally avoided by PRIMO which results in acceleration of legacy codes along with reduction in the user interventions, complex languages and the necessity of the specialized programmers.

For a better understanding of how PRIMO works, Figure 3 shows the instructions generated by the compilation of a simple *vec-sum* kernel for both the GPP x86 AVX-512 and a PIM architecture with the vector width of 64 Bytes. As it can be seen in Figure 3b, each iteration of the *vec-sum* kernel is decomposed into eight *LOAD* (four explicit with *vmovdqu32* instructions and four implicit in the four *vpaddd* second source instructions), four *ADD* and four *STORE* instructions, respectively. The maximum vector size computed by x86 (GPP) AVX-512 SIMD units is 16 doubleword elements (16×4Bytes). This means that, for instance, the 4 iterations of the *vec-sum* kernel generate a total of 32 vectorial *LOAD*, 16 vectorial *ADD*, and 16 vectorial *STORE* instructions. These numbers remain the same for any *vec-width* $\geqslant$ 16 32-bit elements, in case of AVX. Similarly, the PIM equivalent instructions emitted by PRIMO are shown in Figure 3c. Each iteration of the *vec-sum* kernel is decomposed into two *LOAD*, one *ADD* and one *STORE* instructions, respectively. In the case of *LOAD*, the PIM instruction is *PIM_256B_LOAD_DWORD V_0_R2048b_0, [rax+b+16384]*. Here, **256B** indicates the loading of 256 Bytes which is 64× 4Bytes at a time whereas register IDs are indicated as *V_0_R2048b_0* which means *register 0* of *vault 0* of size 2048 bits. This instruction moves the 256 Bytes from memory location *rax+b+16384* to *register 0* of *vault 0* in a specific PIM machine distributed along HMC memory *vaults*.

Similarly, instruction *PIM_256B_VADD_DWORD V_0_R2048b_1, V_0_R2048b_0, V_0_R2048b_1* adds the contents of *register 0* and *register 1* of *vault 0* and puts the result in *register 1* of *vault 0*. The similar semantics can be observed for other vector size instructions as well, with some absolute address adjustments. Finally, in case of PIM with *vector-width=64* 4 Byte elements, a total of 8 vectorial *LOAD*, 4 vectorial *ADD* and 4 vectorial *STORE* instructions are emitted for a 4 iterations kernel as depicted through Figure 3c. The size of vector instructions is dependent on the SIMD vector widths of selected PIM hardware. Hence, one can observe that by using larger vector PIM units, the number of instructions get reduced, with obvious implications in the processing speed given the extra parallelism.

```
for(int i=0; i<N; i++)
    c[i] = a[i] + b[i];
```

(a) C Code

```
mov rax, -16384
.LBB0_1:
vmovdqu32 zmm8, [rax+c+16576]
vmovdqu32 zmm4, [rax+c+16512]
vmovdqu32 zmm3, [rax+c+16448]
vmovdqu32 zmm0, [rax+c+16384]
vpaddd zmm9, zmm0, [rax+b+16384]
vpaddd zmm6, zmm3, [rax+b+16448]
vpaddd zmm3, zmm4, [rax+b+16512]
vpaddd zmm0, zmm8, [rax+c+16576]
vmovdqu32 [rax+a+16384], zmm0
vmovdqu32 [rax+a+16448], zmm3
vmovdqu32 [rax+a+16512], zmm6
vmovdqu32 [rax+a+16576], zmm9
add rax, 4096
jne .LBB0_1
```

(b) AVX-512 ASM Code

```
mov rax, -16384
.LBB0_1:
PIM_256B_LOAD_DWORD V_0_R2048b_0, [rax+b+16384]
PIM_256B_LOAD_DWORD V_0_R2048b_1, [rax+c+16384]
PIM_256B_VADD_DWORD V_0_R2048b_1, V_0_R2048b_0, V_0_R2048b_1
PIM_256B_STORE_DWORD [rax+a+16384], V_0_R2048b_1
add rax, 4096
jne .LBB0_1
```

(c) PIM ASM Code

Fig. 3: PRIMO Code Generation

A. Instruction Offloader

It is important to decide whether instructions are suitable for execution on the PIM accelerator or on the GPPs. For this, PRIMO's *Instruction Offloader* module uses the instruction vector width as a metric in order to select the PIM instructions. At the same time, PRIMO tries to avoid *LOAD/STORE* operations by taking advantage of huge vector registers available inside PIM. The main principle followed is to perform operations with large size vector operations using PIM, and smaller size operations are performed in the GPP. Let us consider a GPP with SIMD units of size v16i32 (a vector of 16 32-bit integer elements), and a PIM accelerator of sizes v32i32, v64i32, v128i32, v256i32, v512i32, v1024i32, v2048i32. In such scenario, if the vector width is set to values less than or equal to 16 32-bit elements, PRIMO will emit routine GPP SIMD instructions. However, as the vector width is increased, such that if it is $\geqslant 32\times$4Bytes elements, PRIMO generates code for the PIM accelerator, which will be executed by respective hardware. Following this metric, the selection of PIM instruction is done by operand sizes. The selection algorithm is applied in the instruction selection stage.

B. Vector Size Selector

In PRIMO the *Vector Size Selector* module is able to select vector sizes by providing larger vector data-types in the middle-end including vectors of 64, 128, 256, 512, and 1024 elements of 32/64bits (integer and floating-point), and vector of 2048 elements of 32bits. PRIMO has enabled generic support for large vectors because the vector width is a feature of the target PIM architecture. Currently, vector operand is selected manually in PRIMO using *force-vector-width* compiler flag. In this manner, if the number of iterations (N) in a given code is $\geqslant$ 2048 PRIMO will select operand size to be v2048 instead of conventional v16, v8, v4. Hence, the same behavior can be observed for remaining vector sizes through Equation 1. These examples show that the selection of vector widths selection with respect to the number of iterations is beneficial in reducing the issued scalar operations and non optimal vector size operations. This leads to the overall reduction in under or over utilization of SIMD units.

$$vectorsize = \begin{cases} v2048, & \text{if } N \geqslant 2048 \\ v1024, & \text{if } 1024 \leqslant N \leqslant 2047 \\ \dots & \dots \\ v8, & \text{if } 8 \leqslant N \leqslant 15 \\ v4, & \text{if } 4 \leqslant N \leqslant 7 \\ scalar, & \text{otherwise} \end{cases} \quad (1)$$

C. Vector Processor Unit (VPU) Selector

HMC is divided into 32 *vaults* to achieve higher bandwidth by exploring *vault* parallelism. Due to this division, several designs take advantage of this organization by including a PIM *per vault* in order to improve the parallelism of memory access and processing. Hence, in these cases 32 independent PIM and their respective 32 independent register files exist.

This section presents another contribution provided as part of PRIMO. It proposes a *VPU Selector* component for mapping the computations to the suitable vector units by selecting the most suitable *vault* (or PIM) to execute determined instruction. This selection is performed during the register allocation phase of the PRIMO, which is responsible for mapping each Virtual Register (VR) of an instruction to PIM physical register. The physical register nomenclature is formatted as *Vault_no_Reg_no*.

The proposed selector algorithm is implemented in the register allocation stage of PRIMO and enables the maximum parallelism by selecting multiple registers belonging to register files of different *vaults*. This is achieved by analyzing the code in order to identify the existing dependencies for mapping the computations to the suitable vector units. Figure 4 illustrates the proposed selection algorithm, which is generic for a PIM consisted of large VPUs. In the proposed algorithm the VR number is treated as the input. The algorithm operates by reading the VR number and checking whether the accessed register is the first one. In case the accessed register is the first VR of the set, the VR gets directly mapped to the available

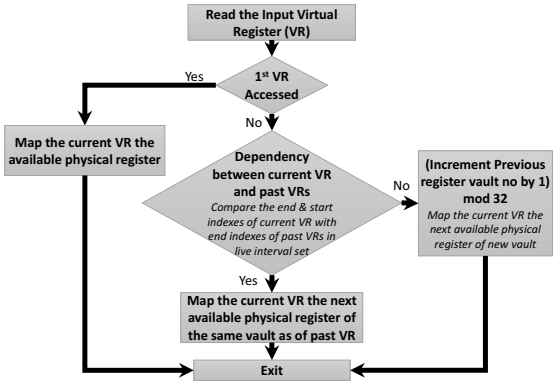

Fig. 4: VPU Selection Technique implemented in Register Allocation Stage

physical register belonging to a certain vault of PIM, without any further checking. However, in scenarios where the accessed VR is not the first register, dependency checking is performed between current and past VRs. In such checking, the end and start indexes of the current VR are compared with end indexes of previous VRs. These indexes are obtained through the live interval set. In case there exists a dependency between the current VR and previous VRs, then the current VR is mapped to a physical register belonging to the same *vault* as the previously dependent register. However, when there is no dependency between current and past VR the *vault* number of the previous VR is incremented by 1 and bounded to 32, which is the *vault* count. In this scenario, the current VR is mapped to the next available physical register of a new *vault*, resulting in increased vault level parallelism. Finally, after mapping the given VR to a physical register, the algorithm is terminated.

D. Code Generator

In the end, the PRIMO code generator component is responsible to emit assembly or binary code for the target PIM. In order to emit assembly no further modifications are required after register allocation. However, for the emission of object code the complete PIM based instruction encoding is required in PRIMO. In the end, the linker creates an executable file containing both the host and PIM instructions after acquiring the object code. When this executable is run on host platform, it automatically offloads the PIM instructions to the PIM hardware and the remaining instructions are executed on host processor.

V. Experimental Setup and Results

This section presents the methodology used to evaluate PRIMO.

A. Simulation Environment

As a case study, we selected a simple PIM architecture presented in [8], which claims to provide enough computing power to exploit the available bandwidth of 3D-stacked memories. The Reconfigurable Vector Unit (RVU) [3] design is a *fixed-function* PIM core that give priority to FUs instead of spending area with ILP techniques in the logic layer of such memories. RVU enables massive and adaptive in-memory processing by extending the native HMC instructions with an AVX-based ISA, hence extracting huge data-level parallelism by using a vast number FUs. RVU allows the vector width to be adjusted according to the current instruction. Therefore, the RVU mechanism can be accessed per *vault*, processing over scalar operands of 4 Bytes or 8 Bytes, and over vector operands with up to 256 Bytes of size. Furthermore, it is possible to concatenate several RVUs at once to increase the vector operation size, achieving operations with up to 8192 Bytes of size. However, to limit the modeling efforts, we limited the maximum vector operator to 4096 Bytes to match the Operating System (OS) page size.

To properly evaluate PRIMO, a complete simulation environment was developed using the GEM5 Simulator [9]. The simulator supports x86 Instruction Set Architecture (ISA) with Intel AVX-512 ISA extension, a complete HMC module with 32 instances (one per *vault*) of the RVU core according to the description presented in [3], [8]. The hardware setup is summarized in the Table I. The evaluation of PRIMO initially is made by a set of small kernels. Further, a subset of the PolyBench Suite [21] is presented.

TABLE I: Baseline and Design system configuration.

Intel Skylake Microarchitecture
4GHz; AVX-512 Instruction Set Capable; L3 Cache 16MB;
8GB HMC; 4 Memory Channels;
HMC
HMC version 2.0 specification;
Total DRAM Size 8GBytes - 8 Layers - 8Gbit per layer
32 Vaults - 16 Banks per Vault; 4 high speed Serial Links;
RVU
1GHz; 32 Independent Functional Units; Integer and Floating-Point Capable;
Instructions from 128Bytes to 4096Bytes;
Vectorial Operations up to 256Bytes per Functional Units;
32 Independent Register Bank of 8x256Bytes each;
Latency (cycles): 1-alu, 3-mul. and 20-div. integer units;
Latency (cycles): 5-alu, 5-mul. and 20-div. floating-point units;
Interconnection between vaults: 5 cycles latency;

B. Performance Evaluation

A critical analysis present in this work is the choice of the most suitable operand size for the targeted PIM. Figure 5 depicts the behavior of four kernels compiled for different operand sizes. It is possible to notice the performance scaling with the size of the vector in the *VECSUM* kernel, which means that a larger vector operand can efficiently exploit the available resources. For this kernel, the PIM setup computing with 4096-byte operand achieves a speedup of 15.8×.

On the other hand, for a Matrix-Vector Multiplication with a workload of 2048x2048 elements, the performance saturates with 2048-byte operands and achieves a speedup of 4.7×. Moreover, as shown in the Figure 5, it is noticed that the use of a vector of 4096 Bytes even slightly harms the performance of this application, achieving 98% of the AVX-512 performance. This occurs when PRIMO tries to use vector operators in a loop, but the data must be gathered from different memory regions in small parts, harming the overall performance. These behaviors show that an optimal operand size selection is an important requirement to allow efficient use of the resources.

Another important issue on multi-instances PIM style is the efficient exploitation of distributed FUs. The use of one or few parallel FUs avoids communication between different register banks. However, it increases the cross-references to distinct *vaults* for memory requests, which reduces the efficiency of the setup. By distributing instructions to the PIM instances matching the vault that data resides, it is possible

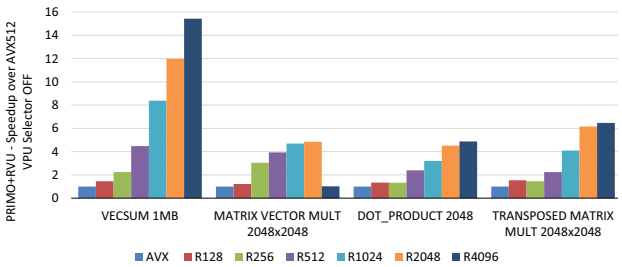

Fig. 5: Speedup of PIM over AVX-512
PRIMO with different vector operand sizes

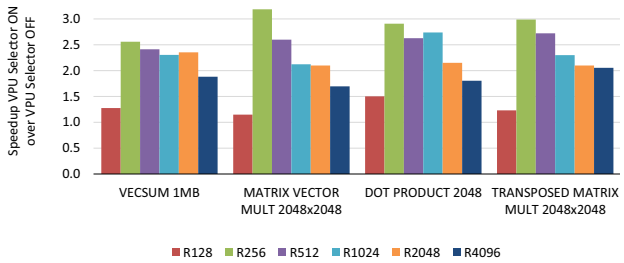

Fig. 6: VPU Selector improvement
VPU Selector ON vs VPU Selector OFF

to exploit the inherent parallelism of FUs, while it reduces the communication between different *vaults* that impacts on performance improvement. Figure 6 illustrates the improvement provided by the PRIMO's VPU Selector algorithm, which focuses on a uniform distribution of instruction along PIM instances to increase performance. The proposed VPU Selector technique can improve in up to 3.1× the utilization of PIM instances.

To widely illustrate the capabilities of PRIMO in exploiting PIM architecture, the Figure 7 shows the speedup achieved for PolyBench. Also, the Figure 7 directly compares the ability of the VPU Selector in extracting more performance from the case study architecture. As illustrate, PRIMO without its VPU Selector feature achieves an average speedup of 6.1×, while by activating its VPU Selector mechanism the average speedup jumps to 11.8×, which shows the importance of efficiently use the PIM instances.

VI. Conclusions and Future Work

This paper presents PRIMO, a compiler able to exploit PIM mechanisms. PRIMO is able to automatically offload instructions by selecting available PIM FUs directly from native code, requiring no programmers interventions. The compiler can select different vector operand sizes, which allows the optimal operand size for each kernel. Moreover, PRIMO contributes with an automatic VPU Selector that allows the distribution of instructions along several PIM instances, which increases performance. The PRIMO tool has been tested on a state-of-the-art PIM by compiling the PolyBench benchmark suite. The results show that by using PRIMO coupled with the case study PIM it is possible to achieve an average speedup of 6.1× for a set of the PolyBench Suite. By exploring the VPU Selector mechanism, PRIMO further improves performance achieving an average speedup of 11.8×.

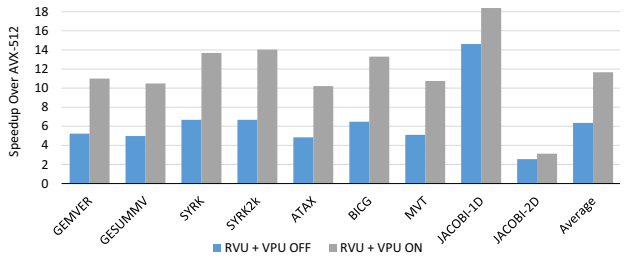

Fig. 7: Speedup of PIM over AVX-512

In future work, more studies are required to improve offloading decisions. These improvements can increase PRIMO accuracy with respect to data reuse, memory access operations, and energy consumption of specific instructions.

References

[1] R. Nair, "Evolution of memory architecture," *Proceedings of the IEEE*, vol. 103, no. 8, pp. 1331–1345, 2015.

[2] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The mondrian data engine," in *Int. Symp. on Computer Architecture (ISCA)*, 2017.

[3] P. C. Santos, G. F. Oliveira, D. G. Tomé, M. A. Alves, E. C. Almeida, and L. Carro, "Operand size reconfiguration for big data processing in memory," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.

[4] J. Standard, "High bandwidth memory (hbm) dram," *JESD235*, 2013.

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

[6] Y. Eckert, N. Jayasena, and G. H. Loh, "Thermal feasibility of die-stacked processing in memory," in *2nd Workshop on Near-Data Processing (WoNDP)*, 2014.

[7] J. T. Pawlowski, "Hybrid memory cube (hmc)," in *Hot Chips 23 Symposium (HCS)*. IEEE, 2011.

[8] J. P. C. de Lima, P. C. Santos, M. A. Alves, A. Beck, and L. Carro, "Design space exploration for pim architectures in 3d-stacked memories," in *Int. Conf. on Computing Frontiers*. ACM, 2018, pp. 113–120.

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, 2011.

[10] 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 (ISCA)*, 2015.

[11] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, "Design and evaluation of a processing-in-memory architecture for the smart memory cube," in *Int. Conf. on Architecture of Computing Systems (ARCS)*, 2016.

[12] M. Scrbak, M. Islam, K. M. Kavi, M. Ignatowski, and N. Jayasena, "Exploring the processing-in-memory design space," *Journal of Systems Architecture*, vol. 75, 2017.

[13] B. Akin, F. Franchetti, and J. C. Hoe, "Data reorganization in memory using 3d-stacked dram," in *Int. Symp. on Computer Architecture (ISCA)*, 2015.

[14] V. Porpodas and T. M. Jones, "Throttling automatic vectorization: When less is more," in *Int. Conf. on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 432–444.

[15] R. Hadidi, L. Nai, H. Kim, and H. Kim, "CAIRO: A compiler-assisted technique for enabling instruction-level offloading of processing-in-memory," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 4, p. 48, 2017.

[16] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 204–216, 2016.

[17] D. Khaldi and B. Chapman, "Towards automatic hbm allocation using llvm: a case study with knights landing," in *Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 2016.

[18] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. Costa, J. Doi, C. Evangelinos *et al.*, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, 2015.

[19] LLVM-Admin Team, *The LLVM Compiler Infrastructure*, 2018-03-08 (accessed March 8, 2018). [Online]. Available: https://llvm.org/

[20] 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 (CASES)*, 2015.

[21] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.