On the SPEC-CPU 2017 opportunities for dynamic vectorization possibilities on PIM architectures*

Rodrigo M. Sokulski[1], **Sairo R. dos Santos**[2], **Marco A. Z. Alves**[1]

[1]Department of Informatics
Federal University of Paraná (UFPR) – Curitiba, Brazil

[2]Department of Exact Sciences and Information Technology
Federal Rural University of the Semi-arid (UFERSA) –- Angicos, Brazil

[1]{rmsokulski, mazalves}@inf.ufpr.br, [2]sairo.santos@ufersa.edu.br

Abstract. *Processing-In-Memory (PIM) devices usually implement vector instructions to efficiently utilize the large main memory bandwidth. One possible way to vectorize applications for such PIM systems is to convert CPU instructions into PIM vector instructions dynamically. In this work, we present a study on the feasibility of the dynamic conversion between these instructions for the Vector-In-Memory Architecture (VIMA). Our results show that 24 % of the loops from some SPEC-CPU 2017 applications are suitable for this conversion. Furthermore, we conclude that dynamic conversion mechanisms must to be able to efficiently deal with memory access conflicts, a problem present in 99 % of all possible conversions to VIMA.*

1. Introduction

One of the main characteristics of computer systems based on the von Neumann architecture is the separation between processing and storage units. This separation implies the frequent need for the CPU to move data from the main memory through an interconnection called the von Neumann bottleneck. This bottleneck can lead to considerable energy expenditures and performance reductions. This constant data transfer corresponds to around 60 % of the total system energy expenditure for some applications, such as rendering web pages and capturing and playing videos [Boroumand et al. 2018].

Between 2004 and 2011, the average speed of high-performance processing cores increased by 20 % per year [Preshing and Poley 2012]. As for CPUs released for desktops, this single-thread growth was 10% per year between 2005 and 2020 [Lechner 2020]. In terms of memories, the emergence of new versions of DRAM and new technologies of 3D memories led to improved bandwidth while keeping the latency practically stable [K. Chang 2017]. Figure 1 illustrates this effect, showing the average performance improvement of CPUs compared to bandwidth and access latency to different types of memory between 2004 and 2020. We consider a 10 % growth per year in CPU speed from 2012 onwards. Every year, the gap between CPU speeds and memory latency becomes more dramatic. Thus, applications with sparse accesses, which do not benefit from greater bandwidth, end up being harmed, which is one of the issues associated with the Memory Wall. On the other hand, applications with coalescent accesses may require multiple threads/processing cores to fully utilize the memory bandwidth.

*This work was supported by the Serrapilheira Institute (grant number Serra-1709-16621) and CAPES (Brazilian government).

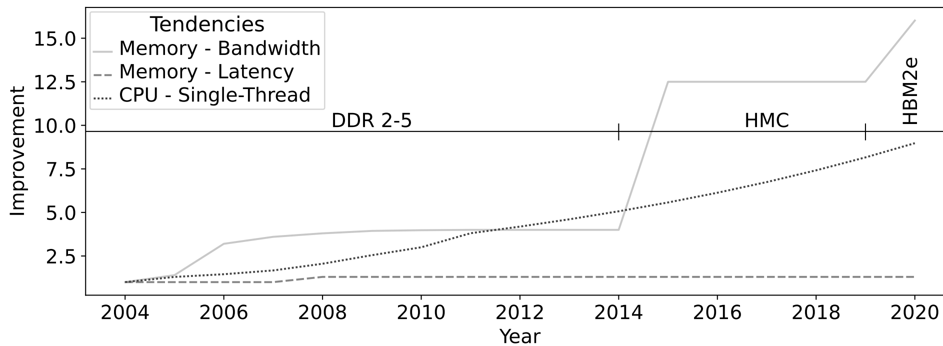

Figure 1. Relative improvement in single-thread speed of CPUs compared to bandwidth and latency offered by memories.

Processing-In-Memory (PIM) techniques emerge as an alternative to reduce these low bandwidth utilization and high latency and energy consumption problems. They consist of adding processing units close to or inside the memory module, allowing the reduction of data traffic between the memory and the processor.

Although this technology appeared over five decades ago [Stone 1970], the recent advent of 3D integrated memories, such as the Hybrid Memory Cube (HMC) [Jeddeloh and Keeth 2012] and the High-Bandwidth Memory (HBM) [Jun et al. 2017], fostered the proposal of several new PIM mechanisms inside the memory die, thus classified as In-Memory Accelerators (IMA) [Santos et al. 2021]. These new devices are implemented on top of the logical layer of 3D memories, allowing faster access to stored data through Through-Silicon Vias (TSVs) [Motoyoshi 2009].

IMA devices can perform either scalar or vector operations. Scalar instructions are more generic and simpler to apply and use. On the other hand, vector instructions are more efficient because they require less communication between the CPU and memory (when transmitting the instructions) and because they make better use of the 3D memories' bandwidth (requesting large data chunks).

VIMA is a IMA device capable of executing vector instructions. Its instructions can be adopted in several ways. As with other PIM devices, VIMA instructions can be used manually, through intrinsic functions [Cordeiro et al. 2017], with specialized compilers [Ahmed et al. 2019, Nai et al. 2017, Hadidi et al. 2017].Intrinsic functions and specialized compilers are limited when dealing with legacy code or proprietary applications, which may lack source code to be recompiled. Although compilations can be executed over the binary code, as it does not contain the complete semantics of the source code, this alternative is more limited than the previous ones. On the other hand, the conversion of instructions at runtime could be performed transparently to users and developers, regardless of the code executed. Nevertheless, as far as we know, there are no dynamic conversion mechanisms for vector PIM instructions nor papers studying the challenges of this kind of mechanism on the SPEC-CPU 2017 benchmarks.

Therefore, this work aims to define the existing possibilities and challenges for future dynamic conversion mechanisms. For this, we characterized the SPEC-CPU 2017 benchmark applications looking for the dynamic vectorization possibilities for the VIMA. We evaluated characteristics of these applications' loops that are relevant to runtime

mechanisms, such as dependencies between instructions, memory accesses, number of iterations, and control flow instructions. Our goal is to identify the most relevant difficulties to overcome during the future development of PIM dynamic conversion mechanisms. Our main finding is that the most commonly found problems faced when runtime vectorizing CPU into PIM instructions are adjacent memory accesses and flow deviations, problems present in more than 90 % of the possible convertible patterns in most of the evaluated applications. This hampers dynamic conversions by requiring mechanisms aware of these problems and capable of get around them.

The rest of this paper is organized as follows: Section 2 presents an overview of our adopted PIM mechanism and the difficulties encountered during the dynamic conversion of instructions. Section 3 presents some dynamic vectorization related work. Section 4 presents our methodology, as well as our results and analyses. Finally, Section 5 presents our conclusions, and potential future work.

2. The VIMA architecture and the dynamic conversion challenges

The rise of 3D memories paved the way for creating PIM IMA devices, encouraging several architectural proposals that implement this type of processing. Among these proposals, some use vector instructions to take full advantage of the main memory internal parallelism, efficiently utilizing its large bandwidth.

One of these IMA proposals is called VIMA, which adds instructions to the processor's Instruction Set Architecture (ISA) in the same way as Intel's Advanced Vector Extension (AVX) instructions. However, such new operations shall execute near-data. Considering that each VIMA instruction must be fetched and decoded by the processor before it triggers the operation to near-data execution, we can classify such architecture as fine-grain PIM (in contrast to architectures that implements a full processor near data).

VIMA is accessible through simulations and allows the maintenance of cache coherence, as well as the use of virtual memory, being applied to generic workloads, according to the classification proposed by Singh et al. [Singh et al. 2019].

In this Section, we present VIMA, its characteristics and restrictions, and some of the difficulties encountered when converting CPU instructions to VIMA instructions in runtime.

2.1. Vector in Memory Architecture (VIMA)

Proposed by Alves et al. [Alves et al. 2022], VIMA extends the system's ISA, allowing the execution of vector instructions near-data. Although there are other similar proposals, such as the HMC Instruction Vector Extensions (HIVE) [Alves et al. 2016] and the HMC Instruction Prediction Extensions (HIPE) [Tomé et al. 2018], VIMA provides some important advantages related to the in-memory processing, such as a simpler programming interface capable of reusing data, presenting precise exceptions, and allowing active multithreading, in addition to showing an expansive design.

The VIMA instructions are executed through a set of functional units, a small data cache memory and an instruction sequencer contained in the logical layer of 3D memories. As in the original VIMA paper, for our experiments we consider a 3D memory containing 32 vaults, each with 8 independent banks, using a row buffer of 256 B.

Thus, VIMA instructions are capable of performing operations on data operands between 256 B and 8 KB in size, which corresponds to the minimum and maximum vault parallelism $(32x256B)$ when considering a HMC 3D memory.

2.2. Conversion problems

During execution, VIMA instruction must load its operands and store the results directly from memory using physical addresses as VIMA does not have a register bank. These instructions receive only one or two operands, which allows us to classify them according to three categories: (i) memory copy, (ii) operation on a vector, and (iii) operation between vectors. Figure 2 illustrates these categories.

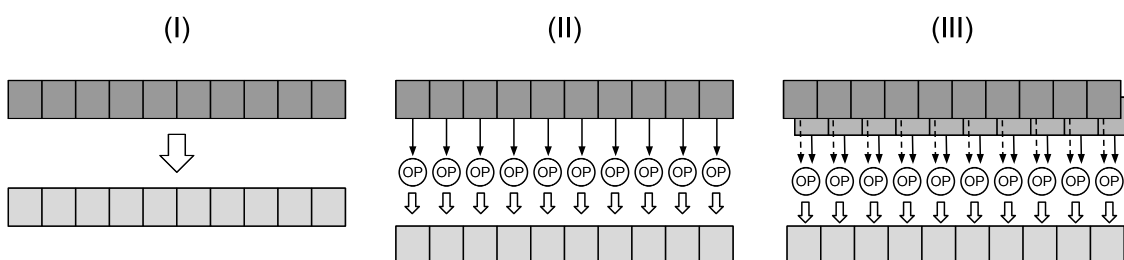

Figure 2. VIMA operations are classified into three categories: (I) memory copy; (II) operation on a vector and; (III) operation between two vectors.

Thus, in order for a VIMA conversion mechanism to function, some pattern of CPU instructions equivalent to one of these categories must be found, for example, a sequence with a load, an operation and a store instruction corresponding to the pattern (II). However, creating PIM vector instructions through dynamic conversion faces problems due to the large size of the data they operate. The converter engines must vectorize multiple CPU instructions to compose a VIMA instruction. For example, a VIMA instruction of 256 B is equivalent to 4× AVX-512 instructions.

This vectorization can be performed through the union of instructions from a basic block or several iterations of the same loop. Nevertheless, the dynamic vectorization without loops seems unfeasible due to the number of instructions required for a single VIMA conversion. Thus, in this work, we consider a mechanism that identifies sequences of instructions inside loops that could be vectorized and converted into VIMA instructions. Figure 3 illustrates in (a) a vector sum algorithm and in (b) the execution of this code in CPU. Since the sum of vectors present in multiple iterations of this loop can be vectorized into one memory instruction, (c) shows an example of such a conversion.

In some cases, this conversion can be problematic if adjacent instructions perform memory accesses, present changes in execution flow, or read registers written by previously converted instructions. These issues are further explored next.

2.2.1. Memory accesses

When vectorizing a sequence of instructions to PIM, adjacent instructions performing memory reads and writes can lead to issues. This problem happens when the memory access order changes between the vectorized sequence and its adjacent instructions. In

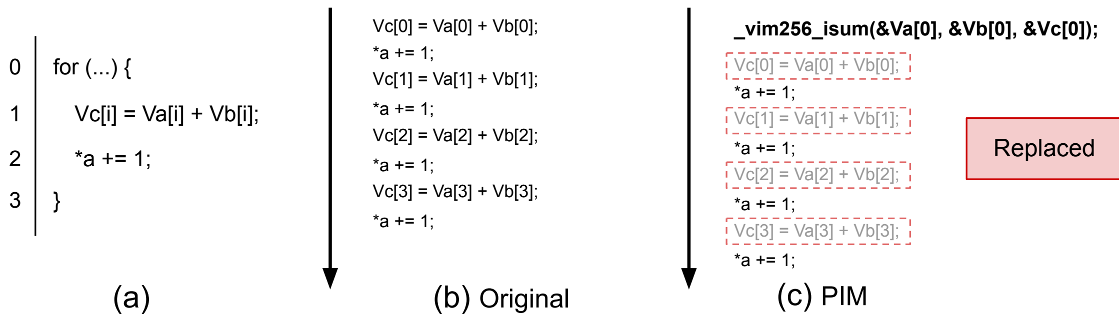

Figure 3. In (a) we have vector sum algorithm, in (b) a conventional execution of (a) and in (c) the replacement of some of the original (b) instructions with a VIMA instruction in runtime.

Figure 3, considering that the pointer a update the value of $Va[3]$, (a) presents a loop in this situation. At each iteration, line 2 increments the content of $Va[3]$ by 1, through the pointer a. Thus, as shown in (b), before the fourth iteration, that calculates $Vc[3] = Va[3] + Vb[3]$, line 2 increments $Va[3]$ 3 times. However, this is altered in (c), with the dynamic conversion of line 1 into a PIM instruction operating over 4 times more data than the CPU instruction. Therefore, $Va[3]$ is only incremented after the PIM instruction calculates the vector sum result of $Vc[3] = Va[3] + Vb[3]$, generating an incorrect result.

In order to prevent this issue, conversion mechanisms for PIM should perform pointer aliasing checks or not convert sequences with other instructions performing memory accesses in the same loop.

2.2.2. Execution flow deviations

For a sequence of loop instructions to be converted to PIM, there must be a sufficient number of consecutive iterations enabling vector opportunities. Furthermore, loops with conditional statements between converted statements can lead to incorrect conversions. Figure 4 (a) presents a algorithm that illustrates this situation. Depending on the value contained in $Va[i]$, line 3 changes the value of tmp, so if a set of iterations of line 4 is turned into a PIM instruction, executing in-memory, without considering this flow deviation, this may lead into an incorrect result. This situation inhibits the conversion of these loops, requiring complex checks, something not viable considering lightweight dynamic conversion mechanisms.

2.2.3. Register readings

Since instructions converted to PIM are executed near-data, their resulting values are unavailable within the CPU registers. As a result, loops containing dependent instructions must be converted entirely to PIM, or the conversion must be avoided. Figure 4 presents an example where (b) if instructions 2 and 3, performing a memory copy, were converted to PIM, the result of instruction 2 would not be available for instruction 4. Thus, instructions and their dependents must be converted into PIM operations, or the conversion should not be performed.

```
0 | for (i=0; i < N; ++i) {
1 |     int tmp = Va[i];
2 |     if (tmp == 3)
3 |         tmp = 0;
4 |     Vc[i] = tmp + Vb[i];
5 | }
                (a)
```

```
int g = 0;                      | 0
for (i=0; i < N; ++i) {         | 1
    int tmp = Va[i];            | 2
    Vc[i] = tmp;                | 3
    g = Vb[i] - tmp + 3;        | 4
}                               | 5
              (b)
```

Figure 4. Vector sum and memory copy algorithms with conversion problems due to flow deviations during conversion (a) and register readings by conversion external instructions (b), respectively.

3. Related Work

This paper presents an analysis of the loops present in SPEC-CPU 2017 for the creation of a runtime converter mechanism between CPU instructions and VIMA instructions. As far as we know, there are no dynamic vectorization mechanisms for in-memory instructions. Thus, our related work analysis consists of proposals for runtime vectorization of CPU instructions performed by software or hardware.

Among the software-based mechanisms, Yardimci and Franz [Yardimci and Franz 2008] propose a virtual machine that converts the application binary into an intermediate representation before execution. During execution, profiling is performed, and the loops of this representation can be parallelized at several levels, such as simultaneous threads, processes, or vector instructions. Nakamura et al. [Nakamura et al. 2011] also perform profiling to identify the most expensive functions of the application, performing vectorizations on these functions basic blocks and loops. Likewise, Hallou et al. [Hallou et al. 2016] use profiling information to identify hot spots on which to apply its optimizations. Despite this, conversions focus on updating existing Single Instruction-Multiple Data (SIMD) instructions, while Nakamura et al. [Nakamura et al. 2011] aim to vectorize application scalar instructions.

Considering hardware-based proposals, Pajuelo et al. [Pajuelo et al. 2002] identify loads with a fixed stride and instructions that operate over such data. Thus, their mechanism recursively vectorizes these instructions. However, due to the speculative nature of the mechanism, writing instructions are not vectorized. Similarly, Call Barreiro [Call Barreiro 2014] identifies operations of the same type to be vectorized. However, these vector operations are accumulated in the Reorder Buffer (ROB) and executed when confirmed that all scalar instructions would execute, avoiding speculative vectorizations. In addition, it does not vectorize loads and stores. Other approaches are adopted by Kalathingal et al. [Kalathingal et al. 2016], which use instructions present in different threads to compose large shared vector instructions. Stephens et al. [Stephens et al. 2017] present a set of size-agnostic vector instructions. Thus, the same vectorization can be used by different systems containing vector instructions of different sizes.

These vectorization techniques presented lead to promising results. Nevertheless, during vectorization for VIMA, new problems arise because in-memory processing is only advantageous for memory-bound code, as CPU-bound code benefits from the more complex Out-of-Order (OoO) execution existing in CPUs. Furthermore, the lack of a

VIMA register bank limits the conversions performed to simple operations linked to memory reading and writing operations. Besides, the distance between the CPU and the PIM execution also poses a challenge when considering moving results. Because of this, the following sections present benchmarks characterizations to understand essential aspects and limitations of real applications conversions.

4. Methods and Results

This section details the workload applications evaluated in this work and the process used for identifying and evaluating loops.

4.1. Benchmark SPEC-CPU 2017

Our evaluations consider over 200 million instruction traces from SPEC-CPU 2017 applications. These traces were obtained using the PinPoints [Patil et al. 2004] tool, present in Intel's Pin [Intel 2018] dynamic binary instrumentation application. This tool uses SimPoints [Calder et al. 2005] technology to select the most significant portions of an application, accelerating our analysis while maintaining the results' reliability.

We evaluated 18 of the 20 SPEC-CPU 2017 applications. Due to conflicts within Pin during trace generation, we did not evaluate the applications pop2 and cactuBSSN. All applications have been compiled with default options and flags to enable SSE, AVX, AVX2, and AVX-512 instructions. Since PinPoints had problems generating traces with gather and scatter instructions, the applications cam4, deepsjeng, fotonik3d, gcc, leela, nab, perlbench, rooms and wrf had their vectorization restricted. That was accomplished with the following flags: -mno-avx2 -mno-avx512f -mno-avx512pf -mno-avx512vl.

4.2. Loop identification

We consider a loop instructions between a backward branch and its target instruction, although function calls and returns are not considered branches. Once we identify all instructions between a branch and its target, these are considered the loop's body. Also, in loops with flow control instructions, the first path was considered the loop's body.

After the initial loop body identification, we count every iteration until the loop's end. Future executions of this loop are accounted for as new loops since loop's outside instructions are executed, hardening inter executions vectorizations. In the final step, we discarded loops with only one iteration since we can not vectorize them.

4.3. Iterations distribution

In order to convert CPU instructions from loops into more extensive VIMA instructions, the converted loop needs a sufficient number of iterations. This number depends on the size of the data processed by the CPU instruction and the PIM device, as shown in Table 1.

Table 1. Number of iterations required to form each type of VIMA instruction, according to the size of the data processed by CPU and VIMA instructions.

	32 bits	64 bits	128 bits	256 bits	512 bits
VIMA-256 B	64	32	16	8	4
VIMA-512 B	128	64	32	16	8
VIMA-1024 B	256	128	64	32	16

In order to evaluate the existence of possibly convertible loops, we analyze the number of iterations present in each loop of our traces. We categorize the loops into some types: i) scalar loops without any vector instruction; ii) vector loops with most vector instructions over 128-bit data; iii) vector loops with most vector instructions over 256-bit data and; iv) vector loops with most vector instructions over 512-bit data.

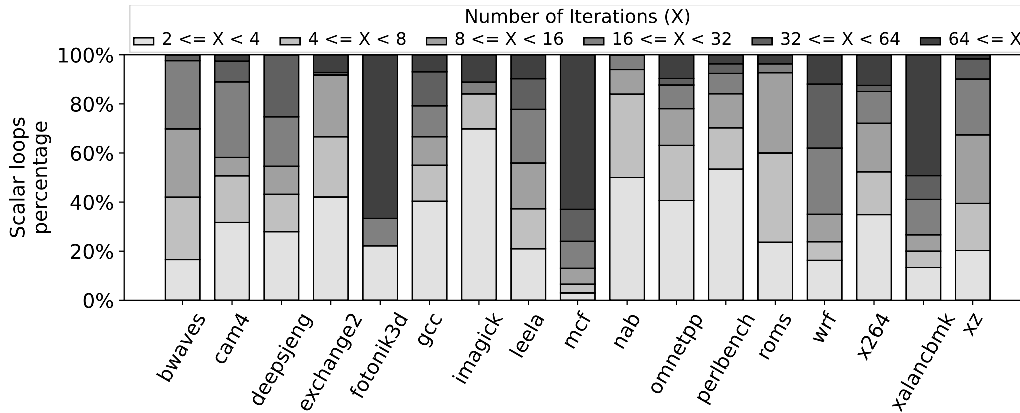

Figure 5. Iterations distribution in scalar loops of some SPEC-CPU 2017 applications.

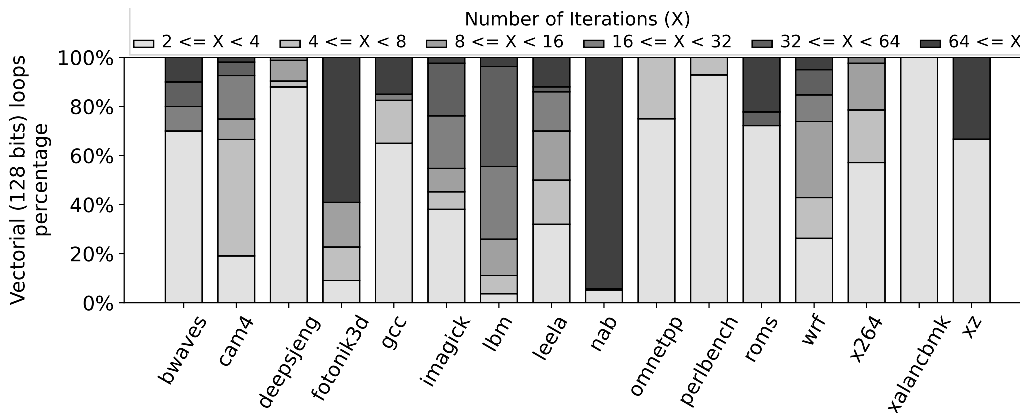

Figure 6. Iterations distribution in loops containing vector instructions, mostly over 128-bit data, in some SPEC-CPU 2017 applications.

Figure 5 presents the iterations distribution in scalar loops, where only more than 32 iterations could be converted to VIMA-256 B instructions. Figures 6 and 7 present the same analysis for loops with most vector instructions on 128 and 256-bit data, respectively. Since the evaluated traces contains only the most representative portions of each application, some applications may not contain some loop types, being excluded from the correspondent figure. Due to compilation restrictions, only the x264 application presented 512-bit vector loops, with all these loops executing at least 8 iterations.

Considering scalar loops, only mcf, fotonik3d, leela, wrf, deepsjeng, xalancbmk and gcc applications have more than 20 % of their loops convertible. On the other hand, for vector loops over 128-bit data, only omnetpp, deepsjeng, xalancbmk, perlbench, x264 and gcc applications do not present conversion opportunities to VIMA-256 B in more

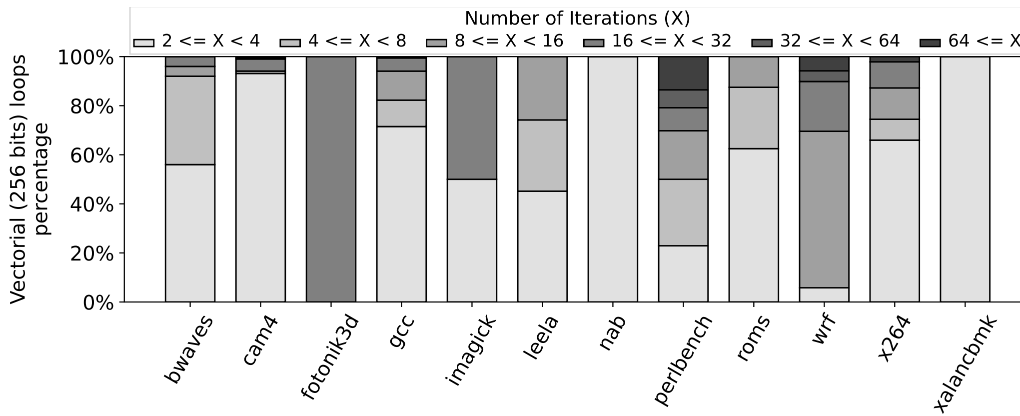

Figure 7. Iterations distribution in loops containing vector instructions, mostly over 256-bit data, in some SPEC-CPU 2017 applications.

than 20 % of their loops. This also occurs for bwaves, nab, xalancbmk, roms, gcc and cam4 applications on 256 bit data loops.

Despite this, when considering the mean of all applications, both, scalar and vector loops present 24 % of their occurrences with a sufficient number of iterations for conversion. Therefore, in order to fully utilize these applications conversions opportunities, a dynamic conversion mechanism should be prepared for utilize both, scalar and vector CPU instructions as sources for PIM conversions.

4.4. Representativeness of loops

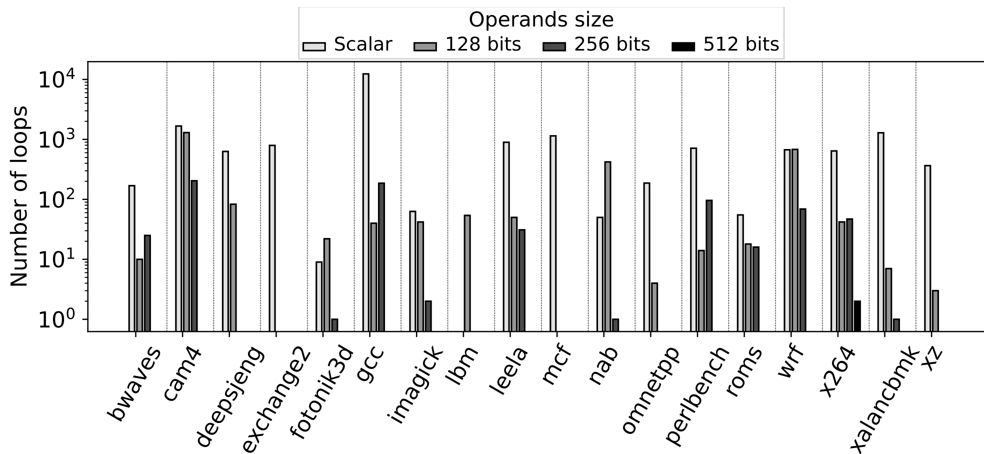

Figure 8. Number of executions of each type of loop for the evaluated applications.

Although conversions over vector operations require a smaller number of iterations, these vector loops may be rare. In order to verify this, Figure 8 presents the number of loop executions classified in each type for our analysis. Most loops do not have vector instructions, which points to previous compiler vectorization difficulties that could prevent runtime conversions. Thus, conversion engines must be aware of these issues, being able to undo incorrect conversions.

Furthermore, in several applications, such as cam4, fotonik3d and wrf, the number of vector loops is significant, reinforcing the hypothesis that vectorizing CPU vector instructions is a promising path for future dynamic conversion mechanisms. Despite that, vector loops correspond to only 28 % of all executed loops when considering all applications. So in a effective runtime conversion mechanism, scalar instructions conversion cannot be ruled out, since it represents the conversion opportunities majority.

4.5. Conversion difficulties

Regardless of the loop's iterations, some factors can entangle the conversion to PIM, as presented in Section 2.2. In order to evidence these problems' frequent occurrence, we identified all the loops with memory copy, vector operation, and operation over two vectors. These patterns consist of straightforward conversions to VIMA, as they perform memory reads, writes, and, at most, one operation.

For each pattern found, we evaluated the existence of other memory accesses, flow deviations, or register reading, blocking its conversion. From this, we noticed that the biggest conversions problem is the existence of other memory accesses, since more than 99 % of the patterns found have an external load and more than 98 % have an external store. The second most common problem are flow deviations, present in more than 90 % of the loops in 16 of 18 applications. Finally, the most unusual problem is reading conversion instruction registers, which occurs in less than 50 % of the patterns for 17 of the 18 evaluated applications.

Based on this, we can conclude that any runtime vectoring mechanism for VIMA needs to be able to handle other memory accesses and flow deviations, due to their frequent occurrence with most convertible patterns. This could be done by runtime checking, but these verifications need to be fast and optimized so as not to degrade system performance. Furthermore, if a verification indicates a conversion failure, the engine must be able to undo the conversion, returning the processing to the CPU and ensuring the correct application execution. Since the reading of conversion instruction registers is not so frequent, a simple alternative to work around this problem would be to avoid converting patterns with this characteristic.

5. Conclusions and future work

With the emergence of 3D memories and the rise of devices such as the HMC and HBM, proposals for in-memory processing have become more common. Despite this, to use their instructions, these mechanisms usually rely on APIs, burdening developers, or recompilation, which is not always viable.

An alternative is to use hardware or software dynamic translation mechanisms. This, however, faces difficulties if the data size operated by the CPU instructions is different from that operated by the PIM device. Thus, we explored some of the difficulties faced in this scenario, presenting an evaluation of relevant characteristics for dynamic conversion to vector PIM instructions in loops present in SPEC-CPU 2017 applications.

Our results show that the majority of dynamic conversion opportunities for vector PIM devices focus on vectorizing the CPU's scalar instructions. Also, for a dynamic conversion device to be successful, it needs to be able to manage memory access conflicts between conversion and external instructions, which occurs in the vast majority of VIMA

conversion patterns found in loops. It should also be able to detect flow deviations during the conversion, being able to undo the conversion if a problem is encountered.

Our results should be considered when designing future dynamic conversion mechanisms, which may be able to vectorize CPU scalar instructions as well as CPU vector instructions. Since it must also handle unconverted instruction memory accesses and changes in the execution flow, being able to deal with or avoid conversions that lead to CPU and PIM dependencies. For future work, we consider the creation of our dynamic conversion mechanism dedicated to overcoming these barriers.

References

Ahmed, H., Santos, P. C., Lima, J. P. C., Moura, R. F., Alves, M. A. Z., Beck, A. C. S., and Carro, L. (2019). A compiler for automatic selection of suitable processing-in-memory instructions. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 564–569.

Alves, M. A. Z., Diener, M., Santos, P. C., and Carro, L. (2016). Large vector extensions inside the hmc. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1249–1254.

Alves, M. A. Z., Santos, S., Cordeiro, A. S., Moreira, F. B., Santos, P. C., and Carro, L. (2022). Vector in memory architecture for simple and high efficiency computing.

Boroumand, A., Ghose, S., Kim, Y., Ausavarungnirun, R., Shiu, E., Thakur, R., Kim, D., Kuusela, A., Knies, A., Ranganathan, P., and Mutlu, O. (2018). Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 316–331, New York, NY, USA. Association for Computing Machinery.

Calder, B. et al. (2005). Simpoint 3.0: Faster and more flexible program analysis.

Call Barreiro, A. (2014). Dynamic vectorization of instructions. Master's thesis, Universitat Politècnica de Catalunya, Barcelona, Spain.

Cordeiro, A. S., Kepe, T. R., Tomé, D. G., de Almeida, E. C., and Alves, M. A. Z. (2017). Intrinsics-hmc: An automatic trace generator for simulations of processing-in-memory instructions. *XVIII Simpósio em Sistemas Computacionais de Alto Desempenho - WSCAD*.

Hadidi, R., Nai, L., Kim, H., and Kim, H. (2017). Cairo: A compiler-assisted technique for enabling instruction-level offloading of processing-in-memory. In *ACM Transactions on Architecture and Code Optimization*.

Hallou, N., Rohou, E., and Clauss, P. (2016). Runtime vectorization transformations of binary code. *International Journal of Parallel Programming*, 45(6):1536–1565.

Intel (2018). Pin - a dynamic binary instrumentation tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`.

Jeddeloh, J. and Keeth, B. (2012). Hybrid memory cube new dram architecture increases density and performance. In *2012 Symposium on VLSI Technology (VLSIT)*, pages 87–88.

Jun, H., Nam, S., Jin, H., Lee, J., Park, Y. J., and Lee, J. J. (2017). High-bandwidth memory (hbm) test challenges and solutions. *IEEE Design Test*, 34(1):16–25.

K. Chang, K. (2017). *Understanding and Improving the Latency of DRAM-Based Memory Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh - USA.

Kalathingal, S., Collange, S., Swamy, B. N., and Seznec, A. (2016). Dynamic inter-thread vectorization architecture: Extracting dlp from tlp. In *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 18–25.

Lechner, M. (2020). Evolution of single-threaded x86 cpu performance. `https://mlech26l.github.io/pages/2020/12/17/cpus.html`.

Motoyoshi, M. (2009). Through-silicon via (tsv). *Proceedings of the IEEE*, 97(1):43–48.

Nai, L., Hadidi, R., Sim, J., Kim, H., Kumar, P., and Kim, H. (2017). Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 457–468.

Nakamura, T., Miki, S., and Oikawa, S. (2011). Automatic vectorization by runtime binary translation. In *2011 Second International Conference on Networking and Computing*, pages 87–94.

Pajuelo, A., González, A., and Valero, M. (2002). Speculative dynamic vectorization. *SIGARCH Comput. Archit. News*, 30(2):271–280.

Patil, H., Cohn, R., et al. (2004). Pinpointing representative portions of large intel ® itanium ® programs with dynamic instrumentation. In *Int. Symp. on Microarchitecture*.

Preshing, J. and Poley, H. (2012). A look back at single-threaded cpu performance. `http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/`.

Santos, P. C., Moreira, F. B., Cordeiro, A. S., Santos, S. R., Kepe, T. R., Carro, L., and Alves, M. A. Z. (2021). Survey on near-data processing: Applications and architectures. *Journal of Integrated Circuits and Systems*, 16(2):1–17.

Singh, G., Chelini, L., Corda, S., Awan, A. J., Stuijk, S., Jordans, R., Corporaal, H., and Boonstra, A.-J. (2019). Near-memory computing: Past, present, and future. *Microprocessors and Microsystems*, 71:102868.

Stephens, N., Biles, S., Boettcher, M., Eapen, J., Eyole, M., Gabrielli, G., Horsnell, M., Magklis, G., Martinez, A., Premillieu, N., Reid, A., Rico, A., and Walker, P. (2017). The arm scalable vector extension. *IEEE Micro*, 37(2):26–39.

Stone, H. S. (1970). A logic-in-memory computer. *IEEE Transactions on Computers*, C-19(1):73–78.

Tomé, D. G., Santos, P. C., Carro, L., Almeida, E. C., and Alves, M. A. Z. (2018). Hipe: Hmc instruction predication extension applied on database processing. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 261–264.

Yardimci, E. and Franz, M. (2008). Dynamic parallelization and vectorization of binary executables on hierarchical platforms. *J. Instr. Level Parallelism*, 10.