A Fast Simulation Method for Multi-Thread PIM Applications

***Abstract*—Processing-in-Memory is a prominent accelerator for many application classes - from pattern matching to machine learning. Since these designs encompass accelerating and increasing the efficiency of critical specific and general-purposed applications, it is expected that these accelerators will be coupled to existing systems and consequently coupled to systems capable of multi-thread computing. However, few tools allow studies and experimentation of single-thread Processing-in-Memory (PIM) systems and even fewer multi-thread PIM systems. This work presents an extension to Sim^{2}PIM, a simple simulator for Processing-in-Memory accelerators, to allow for multi-thread support in a multi-core environment. Sim^{2}PIM can simulate different types of PIM accelerators while executing native host instructions on the host. We extend this support to the Operating System (OS) System Calls, and the *pthread* Application Programming Interface (API). This support allows for the lowest possible instrumentation overhead for multi-thread applications while providing per-thread instrumentation. We show significant gains in simulation speed while maintaining the original Sim^{2}PIM's high accuracy.**

***Index Terms*—Processing-In-Memory, Simulation, Threads, Multi-Threaded, Parallel**

I. Introduction

With the slowdown of Moore's law, the processing units' operating frequency cannot continue scaling at the same rate. This drawback pushed the industry to increase the number of processing cores within a single processor chip to maintain performance gains and technological generation improvements. On the other hand, the Memory-Wall appeared due to the increased performance gap between processor logic and main memory. This performance gap occurs mainly due to technological differences. However, the increase of processing cores on general-purpose systems widened this gap by putting more pressure on the memory system. To mitigate this gap, Processing-in-Memory (PIM), Near-Data Processing (NDP), and Computing-In-Memory (CIM) have emerged as a prominent solution while providing performance and efficiency improvements.

In the last decade, technological advancements have allowed creating a myriad of PIM designs [1], [2], [3], [4]. Although there have been considerable advances in the available tools to experiment and exploit these designs, the field still lacks tools capable of seamlessly simulating the integration of PIM accelerators and host processors. Even more critical are the tools that allow simulation of these designs integrated to multi-core systems and simulate these environments on multi-threading applications.

In this work, we propose to extend Sim^{2}PIM [5], a simple simulator for PIM architectures, to allow for multi-thread support in a multi-core environment. Sim^{2}PIM already works at a low level, so we extend it to identify *pthreads* and wrap them with instrumentation code for each available core.

This extension to Sim^{2}PIM provides:

- High-speed and high-accuracy simulations of multiple application threads in parallel.
- Full compliance with native Operating System (OS) Syscal and MT libraries.
- Full synchronization capabilities in targeted threads, with the native *pthread* Application Programming Interface (API).

Since our proposal minimizes overheads at the thread creation step when not simulating PIM instructions, it achieves performance much similar to performance profiling tools such as *perf* on host code, with as little as 10% run-time overhead and less than 2% metrics difference for most applications. Additionally, by utilizing the host hardware and OS resources, it can run PIM threads concurrently, exploring natural parallelism, for the tested application, achieving 2x simulation time speedup.

The paper is organized as follows: In section II, we introduce the gap that other available simulators leave for multi-thread applications and show how Sim^{2}PIM is well suited to this environment. Section III presents the type of applications that can benefit from a multi-thread compatible PIM device. Section IV explains the modifications to the original Sim^{2}PIM design. Section V evaluates the impact this modification has on Sim^{2}PIM simulation speed and accuracy. Finally, we conclude the paper on section VI.

II. PIM Simulators

Not all PIM designs are created equally, and most simulators available can handle only a tiny subset of these [6], [7], [8]. To accurately handle a multi-thread application in a multi-core environment, the simulation must be aware of the OS and the underlying hardware. Some simulators include the hardware and a virtualized operating system, while others simulate only the PIM device and use the complete host system.

Simulators such as gem5 [9], and SiNUCA [10] are capable of simulating entire micro-architectures with an elevated level of accuracy. SiNUCA [10] is a trace-based simulator, which uses traces generated on a real machine. The simulator has accurate descriptions of the hardware components down to the pipeline of the processor. However, it can not simulate the interactions with the operating system and other processes. The simulator also suffers from the flaws of other trace-

based simulators in that the benchmark can not interact with simulated hardware, as the traces have already been collected.

Researchers have used the gem5 [9] simulator to evaluate a set of applications under different hardware configurations, making this setup perfect for hardware-software co-design. The simulator is divided into several independent modules, coupled and decoupled to test different combinations. However, this modularity and broad set of configuration options create a notoriously steep learning curve for using the gem5 environment. While the code maintainers strive to improve usability, testing disruptive new hardware such as PIM units on the simulator can prove a hurdle, even for simplistic experiments.

A faster alternative is to use PinTools [11]. Utilizing trace-files containing cycles, memory access, and data as input for basic models of the processor and memory hierarchy, acting much like SiNUCA, the tool can interpret each issued instruction. PinTools's huge instrumentation overheads prohibit direct code measurements and gem5's extensive simulation times and barrier of entry. None of these simulators can handle threaded applications with native system calls. Baremetal simulators as [10] can not simulate OS-level thread scheduling and system calls, while gem5 based simulators still face long simulation times and added virtualization overheads.

However, even with limited support from simulators, multi-thread applications are a majority in high-performance computing applications. Thus, the need arose for simulators capable of handling multiple memory stacks at the host and PIM side. Developed explicitly for this purpose, MultiPIM [12], which is based on two other simulators [13], [14], can simulate a multi-stacked-memory PIM device. The simulator offloads *POSIX* and *OpenMP* threads, mapping them to the PIM hardware, maintaining coherence between cores, and a PIM-side task scheduler. It can therefore handle multi-thread applications at PIM-side. However, the simulator utilizes Intel's PinTool [11] based instruction feeding mechanism, which interprets each instruction in the virtual environment at runtime. The program must then simulate all the metrics in a virtual environment, bearing a long simulation time.

Sim^{2}PIM [5], a recent simulator, proposed to overcome the downfalls of previous simulators, providing a fast, accurate, and simple-to-use experimentation environment for hardware-software co-design of PIM devices. Sim^{2}PIM acknowledges that integration with available hardware is imperative for testing and disseminating PIM technologies. To provide that, the researchers described a method to use the host's baseline hardware and its OS to execute host-only sections of the benchmarks. Like the kernel virtual machine (KVM) provided in gem5, it can execute code at host speed, but it grants access to program metrics via the Hardware Performance Counters. Sim^{2}PIM manages to directly instrument the compiled *assembly* code containing PIM instructions and insert simulation calls using a parser. It detects PIM instructions and substitutes them for host memory instructions. In the technique described in [15] the host is left as is, and integration happens in a *Plug and Play* manner. However, Sim^{2}PIM currently does not provide support for multi-threaded benchmark applications. If an MT benchmark is executed in Sim^{2}PIM, the simulator will execute each thread in series. The serial execution loses accuracy, as communication overheads between the threads will be masked. It also significantly increases the simulation overhead for Pthread mutexes and other synchronization schemes, as these require that the current thread be halted, and another one restarted.

As this simulator has the best accuracy results and fastest simulation speeds, we chose it as a basis, and as we will see, the main challenges lay in expanding the instrumentation methodology to multiple threads while maintaining low overheads. In section IV we further explain the instrumentation process and how its implementation is advantageous to a multi-thread simulation. Since the authors make the code available, it is straightforward to implement and modify with only a handful of $.c$ files.

III. Multi-thread PIM Motivation

The most prominent PIM architectures are intrinsically data-parallel Single Instruction Multiple Data (SIMD)-like units, which operate on vectors and matrices [1], [2], [3]. However, applications can explore different parallelism levels, such as task, data, and instruction parallelism [4]. These levels can be used separately or jointly.

SIMD exclusive: The most straightforward case, where SIMD units are well suited, is when exploiting the locality of large chunks of data. A single host core can dispatch instructions to a SIMD unit located inside the memory device [2], [16], [17], [18]. Most PIM architectures fall in this category, as it is the simplest to implement and to benchmark against Central Processing Units (CPUs) and Graphics Processing Units (GPUs).

SIMD + Instruction Level Parallelism (ILP): A more interesting case is when the application allows for the simultaneous execution of the different instructions in multiple data chunks, with a single core dispatching instructions to different SIMD execution units, like a multi-issue execution stage. The application itself is still serial on the host core, as it uses independent sets of data to apply different instructions. The work presented in [19], [2] are examples of this approach, where each tile is controlled and scheduled independently by the dispatching unit.

SIMD + Thread Level Parallelism (TLP): The least explored case in PIM literature is when multiple cores or threads interact with the PIM simultaneously. TLP usually relies on exploiting multi-core architectures to compute different parts of a program simultaneously while taking care of inter-thread communication and preventing data races.

When discussing PIM performance, it is common to compare it with multiple threads executing applications with great spatial-locality in the multiple processor cores while using the SIMD units [20]. However, many applications, like N-Body simulations, layers of a neural network, among others, can be implemented making use of threads computing on different segments of data concurrently. So far, to the best of

our knowledge, there is no available tool that allows precise and fast simulation of such applications executing on a multi-thread PIM environment.

IV. Simulator Methodology

This section reviews the process that makes Sim^{2}PIM fast and accurate and then describes the added features that make it suitable for Multi-Thread simulation.

A. Flow and Integration

The simulation comprises three distinct components: the instrumented *assembly* code, the simulator backbone, and a PIM description code. Before the simulation, a parser generates the *assembly* code by replacing user-defined macros, functions, *intrinsics*, explicit compiled PIM instructions or other means to tag PIM operations with host memory instructions, as described in [15]. This process guarantees the results are comparable to a real PIM hardware interfacing with the host processor as is. This method also guarantees that the simulation environment operates on the same memory space, allowing for seamless inter-thread data coherence and virtual memory support. If the PIM hardware does not support inter-thread communication inside the PIM device, as is the case with UPMEM [21] and PIM-HBM [22], communication in the host caches can be mimicked with loads and write-backs.

However, if the PIM design demands other offloading, cache coherence, or virtual memory support methods, the inserted overheads in this step (Offload overheads in Figure 1) can be discounted and even changed for the appropriate overheads. The parser also inserts function calls to the simulator backbone, respecting calling conventions, as illustrated on Listings 1 and 2. These calls are not resolved since the instrumented *assembly* code is linked with the simulation backbone and the PIM simulation at a later stage.

Listing 1:
Original x86 + PIM Code Snippet - Annotated or Compiled

```
1  movq %rax, %r14
2  . . . . . .
3  PIM_LOAD  32512(%rbx,%rax), %PIM_REG_0 ;PIM instruction
4  . . . . . .
5  addq $2048, %rax
```

The simulator backbone handles the setup of performance counters, launching of threads, and offloading of PIM instructions from the benchmark to the PIM simulator. Also, it is responsible for launching the PIM simulation in a reserved thread. This approach assures that there will be minimal pollution of the caches during simulation time, which increases the accuracy of the hardware counters. It is also responsible for warming up the performance counters and evaluating overheads to be considered and discounted later. The simulation call inserted into the application's *assembly* code (as shown in listing 2) is part of the backbone code. Before it starts the simulation of PIM instructions, the backbone samples the performance counters of the current core and then invokes the PIM simulation environment.

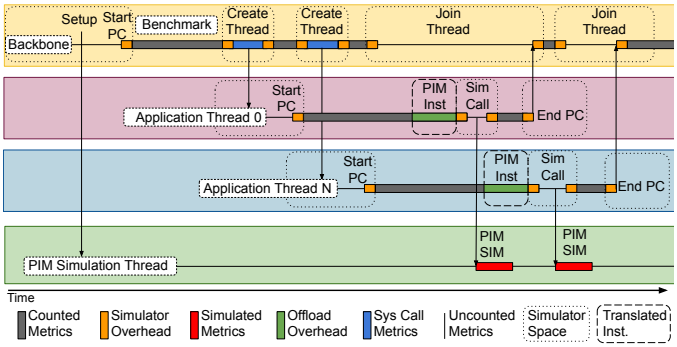

Fig. 1: Overhead diagram for a multi-thread application on the extended Sim^{2}PIM.

Listing 2:
Original x86 + Simulation Code Snippet - Parsed

```
1  movq %rax, %r14
2  . . . . . .
3  movq (PIM_LOAD), %rdi ;PIM instruction as PIM simulator
       opcode
4  callq SIM_CALL ;SIM_CALL receives %rdi as parameter
5  . . . . . .
6  addq $2048, %rax
```

The benchmark itself is defined as a function to be called in the backbone between instrumentation sections. For the simulation of a multi-thread application, each thread will perform its simulation calls, isolated in each core. After the thread offloads its instruction to the simulator, the counters are sampled again[1], and the original *assembly* code continues. The HW simulation receives the instructions through a single-consumer and multiple-provider circular buffer with atomic locks to allow the offloading of multiple control streams to the PIM. In this multi-core environment, the benchmark threads do not wait for the PIM simulation to complete, more accurately simulating the PIM as a separate, asynchronous unit.

Since the benchmark, the backbone, and the PIM simulation are all threads in the same program, they share the same memory space. Thus it is trivial for the PIM to operate over the program data as if both PIM and data were on the same memory device. Also, this feature allows to seamlessly simulate PIM devices that support virtual memory. Additionally, the host's cache hierarchy guarantees cache coherence for the simulation itself. The process described in [15] can be used to implement the coherence and virtual memory support in standard hardware, or the simulation could be coupled with other methods of cache coherence (e.g., [23]). The PIM simulator code takes the complexity level the designer desires, from a cycle-accurate simulation (e.g., SystemC) to an instruction-level look-up table, and in our tests, a functional description of the hardware with estimated execution times.

B. Simulation Thread Mapping

The single thread version of Sim^{2}PIM environment relies on a multi-core system so that the hardware simulation does

[1]The Hardware Performance Counter (HPC) are overflow counters; this means one must acquire the difference between two consecutive measures instead of an absolute value.

not affect the metrics from the benchmark. The simulation uses two dedicated cores, one exclusive for the PIM hardware simulation and the other for the backbone and benchmark. To guarantee that each thread's metrics are accounted for correctly, we must apply the same principle of isolating each part of the application in a different core. As each core has its performance counters, we must ensure that the threads do not migrate cores, which would result in a wrong calculation of the metrics.

If the number of threads is greater than the number of available cores, Sim^{2}PIM can offer two distinct approaches: **1)** it can allow for all the threads to be launched by the main thread and dispute core time with the other thread. This approach would allow the OS to optimize thread context switches and simultaneously keep a more significant number of threads alive. Alternatively, **2)** it can allow the execution of only one thread per physical core at a time. Thus, this solution provides the best accuracy for individual threads, even allowing for better profiling of the PIM instructions.

We extend the pre-linkage parser as shown in Section IV-A to identify and modify calls to the *pthread* library made in the benchmark code, especially the *pthread_create()* and *pthread_join()* functions. They are replaced by *wrapper* functions, which extend the *pthread* calls, making them capable of setting the core affinity, marking the physical core as in-use (if using strategy **2)**), and acquiring metrics before the start of the thread function itself. Inside this function, the only measurement made is around the *pthread_create()* call, as we make sure to measure the thread's launch, as this poses a significant overhead in multi-thread applications, shown in Figure 1 as *Sys Call Metrics*.

Each thread counts its metrics, and we assume all the threads are executed concurrently. For the *elapsed-cycles* and *elapsed-time* metrics, only the largest value is considered to the final count, as it is the bottleneck for program conclusion. For the other types of metrics available on the host, core/thread-wise metric counts are valuable (e.g., emitted instructions, cache misses) and are thus counted separately for each thread. For all metrics, the wrapper function around *pthread_join()* guarantees the waiting time for the simulations to end is not counted on the benchmark's main thread metrics, as shown in Figure 1.

A note on PIM hardware thread mapping: There are three main ways to map a thread to the PIM hardware; hard-coded user assignment, automatic compiler mapping, and OS/Driver mapping (General-Purpose/GPU-like). The programmer or the compiler can map the threads to the hardware directly in the benchmark code. However, if the designers desire it, a scheduler module could be added in several places to handle scheduling in a specialized manner. If added to the thread creation wrapper, the scheduler could handle a static but run-time assignment. If the module is added to the simulation environment, it could handle the threads dynamically, or if the scheduler is added to the PIM HW functional simulation, it could handle thread assignment inside the device. Sim^{2}PIM provides much flexibility, as all the steps of execution are in control of the user.

C. Code Offloading

We adopt the design presented in [15] that provides code offloading, cache coherence, and virtual memory support. We chose this design since it can be extended to several modern PIM designs, from PIM designs that exploit cache memories [1], logic layers in 3D stacked memories [2], memristors [3] and in-memory 3D-stacked accelerators, such as Samsung's PIM-HBM [22]. The overheads of this method can be easily estimated and can also be discounted if the designer wishes. Since the delivery of instructions is guaranteed, other offloading means (e.g., Instruction Set Architecture (ISA) extension) could be added if we can estimate the overhead and create a functional simulation.

D. Synchronization

When using pthreads, the programmer primarily handles synchronization between threads, with mutexes, semaphores, and other atomic operations. This behavior does not change on the simulator, as the caches are still used for shared memory space between threads, and synchronization, in general, happens at the host-side and is thus handled natively.If shared data is used concurrently by multiple threads (e.g., a shared vector), the typical approach would be to use pthread barriers or other atomic operations to avoid using outdated values in other threads, maintaining synchronization. This approach remains valid for most situations in the simulation environment, as all synchronization mechanisms still execute natively. However, in cases where the last issued PIM instruction before the barrier was a store, the thread must await its completion before proceeding with further instructions. The same is valid for in-thread data checks.

A simple solution is for the simulator to consider store instructions synchronization barriers in the simulation. It must wait for the PIM hardware simulator to finish the instruction before returning to benchmark code. Thus the instrumented simulation call must be aware of the store instruction and await its completion by the PIM thread. The time this takes is not accounted for, as this happens in simulation space. This behavior mirrors the real-hardware as described by the offloading mechanism [15], and the cache coherence works as well.

V. Evaluation

Sim^{2}PIM merits lay on top of its high simulation speeds, high-accuracy host hardware metrics, and the backbone's structure high modularity. These characteristics make Sim^{2}PIM especially suited to evaluate the interactions between host hardware and the PIM device, including the system's memory hierarchy and technology. We show this by evaluating different host systems, with a known PIM [2] architecture built upon the HMC [24] memory. The specifications can be seen in Table I. We set out to showcase the low-overheads our modifications incur in *pthread* calls and the speedup of simulating multiple threads in Sim^{2}PIMs truly multi-core

environment. For the backbone overhead evaluation tests, we used some of the algorithms on the PolyBench benchmark suite [25]. As for the PIM performance measurements, we used an ideal VecSum kernel operating over a large (64MB) vector.

A. Pthread Overheads

The extension to Sim^{2}PIM's backbone can be seen as thread-level instrumentation, akin to the original simulator for a single-thread benchmark. Therefore the costs and overhead evaluation within the thread and for PIM instructions are similar to the original publication. The new simulation overheads happen at the *wrapper* functions for thread creation and destruction, as aforementioned in Section IV-B. To evaluate the impact of these overheads on PIM multi-thread simulation, we compared the same set of applications executing with and without the simulator. The simulator instrumentation is as presented in Section IV, while for the comparison, we selected *perf* as an easy to deploy, low-overhead and high-accuracy performance profiling tool. The results are shown in Table II.

For both the single thread and multi-thread results, we can see that for most benchmarks, Sim^{2}PIM and *perf* show similar results, with a trend towards smaller results in Sim^{2}PIM. We leverage this trend is due to the instrumentation provided by Sim^{2}PIM to be more accurate due to the use of hard-coded HPC, not depending on slower system calls. The cycles metric is influenced by several factors, including the congestion of the memory subsystem and frequency fluctuations. Thus some applications may present more variation than others. Splitting the application between threads might also affect this behavior, increasing the traffic in the processor caches. We can also see the execution time collected with the *time* command for Sim^{2}PIM and from *perf* itself. If we consider the overhead of the *perf* application negligible on larger run-times, we can see that Sim^{2}PIM's performance for host code is very competitive, especially if we consider the usual run-times of other simulators.

B. Multi-Thread Simulation

We test the multi-thread PIM simulation on a simple embarrassingly parallel kernel that performs the vector sum of a large array. This way, we avoid complications dealing with complex vector algorithms and inter-thread communication. As the number of threads increases, the size of the array for each thread gets smaller. In Figure 2 we can see the advantages of executing PIM code with thread-level parallelism as there is an apparent speedup in the collected metrics (Simulated Cycles). Moreover, we can see that the simulation does not suffer a performance penalty for simulating this scenario. It becomes even faster. The advantages of simulating each thread in parallel are evident when we evaluate the costs of executing a multi-thread benchmark, dividing work between a different number of cores.

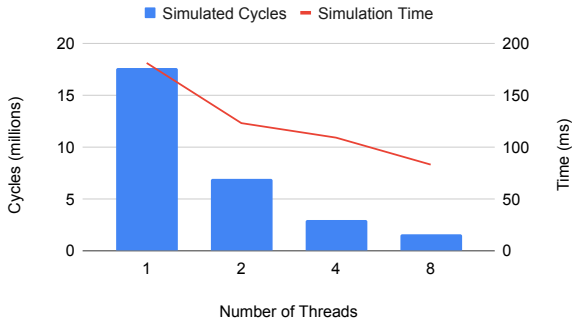

Fig. 2: Simulated Cycles and Simulation Time speedup for a 64MB VecSum application offloaded by the Xeon CPU to the PIM device achieved by increasing the number of threads for the application.

Even though simulators like Gem5 can simulate very different architectures, even enabling the design of new architectures, the added complexity limits the ease of use for simple tests like these. Quickly changing a parameter and rerunning the simulation is not an option as the simulation can take hours. Intel's PinTool offers very fine control over the code currently executing, allowing the user to follow branches and accessed memory addresses. However, once again, the slow simulation speed makes quick testing a nuisance. Furthermore, PinTool and other trace-based simulations suffer from another challenge, as changes in the benchmark require the traces to be reacquired. Figure 3 presents a comparison between our previous experiments with these tools showing the accuracy of the metrics in relation to estimates of the actual hardware performance [2] on the y-axis. On the x-axis, the simulation speeds are normalized to the run-time of the multi-threaded version of Sim^{2}PIM.

Like many other simulators, the original version of Sim^{2}PIM could simulate multi-thread applications by executing each thread sequentially. However, it could not simulate the interaction of these multiple requests on shared resources, such as memory access bandwidth. Sim^{2}PIM is now capable of doing so in a fast simulation time, with minimal simulation overheads.

TABLE I: Baselines and Case Study PIM Parameters

Baseline/Host Intel Xeon Silver-4214 @ 2.2GHz; Cache per Core L1 = 32kB; L2 = 1024kB; Last Level Cache = 16MB; Main Memory DDR4 2x32GB 2400MHz CL16;
Baseline/Host AMD R5-1600 @ 3.2GHz; Cache per Core L1 = 32kB; L2 = 512kB; Last Level Cache = 8MB; Main Memory DDR4 2x8GB 2666MHz CL16;
RVU Processing Logic [2] Operation frequency: 1 GHz; Up to 32x 64 functional units (integer + floating-point); Vector sizes (bytes): 32x 256, 16x 512, 8x 1024, 4x 2048, 2x 4096, 1x 8192 Latency (cycles): 1-alu, 3-mul. and 20-div. int. units; Latency (cycles): 5-alu, 5-mul. and 20-div. fp. units; Register bank: 8 sets of 32 composable registers of 256 bytes each;

TABLE II: Simulated Cycles vs Simulation time for Sim^{2}PIM and *perf* on the AMD processor.

Benchmark - data size	Perf cycles		Sim^{2}PIM cycles		cycles % increase		Perf Time (s)		Sim^{2}PIM Time (s)	
	1T	4T - Average	1T	4T - Average	1T	4T - Average	1T	4T - Average	1T	4T - Average
vecsum - 32MB	1.25E+07	3.06E+06	1.23E+07	3.05E+06	-1.799	-0.541	0.0165	0.0083	0.037	0.035
gemm - 1,5MB	2.89E+07	9.68E+06	2.84E+07	7.99E+06	-1.577	-17.497	0.0173	0.0089	0.029	0.033
2mm - 750kB	1.57E+08	3.93E+07	1.57E+08	3.92E+07	-0.018	-0.255	0.0524	0.0219	0.039	0.046
covariance - 16MB	1.48E+09	4.30E+08	1.66E+09	4.23E+08	12.491	-1.734	0.7163	0.4740	1.041	0.5
Floyd-Warshall - 8MB	1.18E+10	3.93E+09	1.18E+10	3.88E+09	-0.018	-1.261	3.2186	1.1955	3.232	1.22
Nussinov - 8MB	2.86E+10	7.23E+09	2.88E+10	7.21E+09	0.597	-0.319	7.7568	1.9769	7.825	1.998

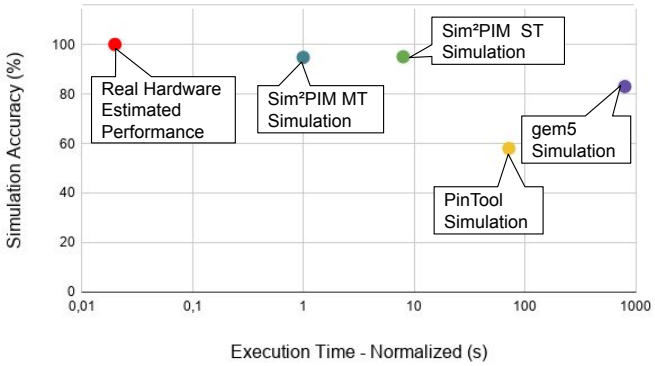

Fig. 3: Execution time and accuracy for a VecSum application with 8 threads in different simulators.

VI. Conclusion

It is more apparent than ever that new architectural paradigms must be proposed to keep up with computing trends. PIM is a clear contender to take the top spot of energy efficiency and acceleration. However, there must be leaps in architectural designs and the support environment to accelerate this development, including simulators. This extension to Sim^{2}PIM brings a fast and accurate simulation tool for multi-thread applications to the hands of researchers and designers. It makes few compromises, guaranteeing fast simulation speeds and high accuracy, as long as the host hardware is available for testing. Currently, the simulator is perfectly capable of exploring dual-socket systems, and as future work, we aim to explore the use of more than one HW simulation thread to emulate multiple PIM devices in the same system.

References

[1] S. Aga *et al.*, "Compute caches," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[2] P. C. Santos *et al.*, "Operand size reconfiguration for big data processing in memory," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017.

[3] S. Hamdioui *et al.*, "Memristor based computation-in-memory architecture for data-intensive applications," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015.

[4] H. A. D. Nguyen *et al.*, "A classification of memory-centric computing," *J. Emerg. Technol. Comput. Syst.*, vol. 16, Jan. 2020.

[5] P. C. Santos, B. E. Forlin, and L. Carro, "Sim²pim: A fast method for simulating host independent pim agnostic designs," ser. DATE '21, 2021.

[6] G. F. Oliveira, P. C. Santos, M. A. Z. Alves, and L. Carro, "A generic processing in memory cycle accurate simulator under hybrid memory cube architecture," in *SAMOS*, 2017.

[7] L. Xia *et al.*, "Mnsim: Simulation platform for memristor-based neuromorphic computing system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[8] J. D. Leidel and Y. Chen, "Hmc-sim-2.0: A simulation platform for exploring custom memory cube operations," in *2016 IEEE Int Parallel and Distributed Processing Symp Workshops (IPDPSW)*, 2016.

[9] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, 2011.

[10] M. A. Z. Alves *et al.*, "Sinuca: A validated micro-architecture simulator," in *2015, 17th Int. Conf. on High Performance Computing and Communications*, 2015.

[11] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation." Association for Computing Machinery, 2005.

[12] C. Yu, S. Liu, and S. Khan, "Multipim: A detailed and configurable multi-stack processing-in-memory simulator," *IEEE Computer Architecture Letters*, 2021.

[13] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Int. Symp. on Computer Architecture*, 2013.

[14] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, 2016.

[15] P. C. Santos, B. E. Forlin, and L. Carro, "Providing plug n' play for processing-in-memory accelerators," in *Asia and South Pacific Design Automation Conference (ASPDAC)*, 2021.

[16] M. Drumond *et al.*, "The mondrian data engine," in *Int. Symp. on Computer Architecture*. ACM, 2017.

[17] G. F. Oliveira, P. C. Santos, M. A. Alves, and L. Carro, "Nim: An hmc-based machine for neuron computation," in *Int. Symp. on Applied Reconfigurable Computing (ARC)*. Springer, 2017.

[18] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "Computedram: In-memory compute using off-the-shelf drams," in *Proceedings of the 52nd Annual IEEE/ACM Int Symp on Microarchitecture*, ser. MICRO '52, 2019.

[19] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Int. Symp on Computer Architecture (ISCA)*, 2016.

[20] V. Seshadri *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *Int. Symp. on Microarchitecture (MICRO)*, 2017.

[21] J. Nider *et al.*, "A case study of processing-in-memory in off-the-shelf systems," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 117–130. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/nider

[22] S. Lee *et al.*, "Hardware architecture and software stack for pim based on commercial dram technology : Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 43–56.

[23] A. Boroumand *et al.*, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," *IEEE Computer Architecture Letters*, 2016.

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

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