# Extending uPIMulator and Enabling Execution Pipelining to Hide Memory Transfer Latency

Weichu YANG, WEICHU@CS.WISC.EDU

**Abstract**

UPMEM-PIM is a commercial, general-purpose parallel computing architecture with processing-in-memory (PIM) capabilities. While uPIMulator is a lightweight simulator designed for UPMEM-PIM systems, it lacks support for modeling communication between the host CPU and DPUs. Prior studies have shown that DRAM ↔ PIM data transfer throughput is significantly suboptimal, often becoming a major performance bottleneck. To better analyze this limitation, this work extends uPIMulator and integrates it into the Gem5, enabling CPU–DPU interactions with a memory transfer latency model. Furthermore, we propose a potential optimization strategy that partitions a single workload into streams with smaller data blocks and executes them in a staged, pipelined fashion. This design allows the data transfer of one stream to overlap with the computation of the previous stream, effectively hiding CPU→DPU memory transfer latency. Under ideal conditions, preliminary results from representative benchmarks indicate that the proposed approach can yield up to a 1.9× performance improvement in the CPU→DPU data transfer and DPU computation phases.

## 1 Terminology

**UPMEM PIM**
UPMEM PIM is a commercial general-purpose Processing-In-Memory (PIM) technology. The general workflow of a UPMEM-PIM device is as follows: allocate a DPU, offload the executable binary from the CPU to the DPU, offload the input arguments from the CPU to the DPU, perform computation on the DPU, return the computation results from the DPU to the CPU, and then proceed to the next round of computation.

**Tasklet**
A software thread that runs on a DPU in UPMEM's PIM architecture. Tasklets execute the same code on different pieces of data and may follow different control-flow paths at runtime. Each DPU supports up to 24 concurrent tasklets.

**DPU**
A 350 MHz in-order 14-stage pipelined processor with a RISC-based ISA, supporting up to 24 software threads (tasklets).

**MRAM**
The main memory (64MB) attached to each DPU, accessible by both the host CPU and the DPU itself. It is used for input/output data exchange and persistent storage.

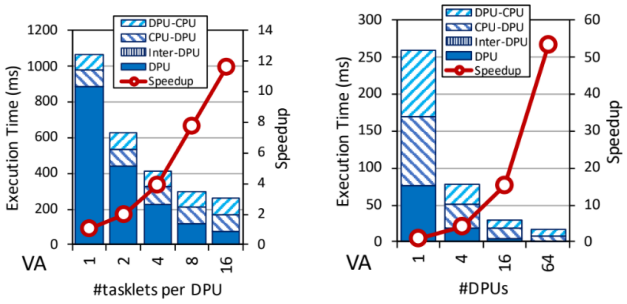
**WRAM**
A 64KB on-chip working memory inside each DPU, used for fast, temporary data access during computation. WRAM is shared among tasklets and offers significantly lower latency than MRAM.

**uPIMulator**
A lightweight, cycle-level simulator for the UPMEM-PIM architecture[1].

**VA** A vector addition benchmark from the PrIM (Processing-In-Memory) benchmark suite. It computes the element-wise sum of two vectors. In UPMEM-PIM, the CPU loads the input vectors and sends them to the DPUs, which perform the addition and return the result vector to the host.

**Workload**
In this work, a workload refers to the content offloaded from the CPU to the DPU for execution. It includes both the executable binary and the corresponding input parameters required for processing.

**Stream**
A concept introduced in this work to represent a sub-workload derived from partitioning a larger workload into smaller data blocks. In the stream mode, the executable is sent once at the beginning, subsequent transfers of stream involve only the input data.




(a) Varying #tasklet per DPU (b) Varying #DPU

Figure 1: Execution time (ms) and speedup(considering only the portion of execution time spent on the DPU, compared with that of one tasklet per DPU or one DPU) of benchmarks VA. These figures both are directly taken from Gómez-Luna et al. [2].

## 2 Motivation

For a UPMEM-PIM system, the overall overhead typically arises from three sources: (i) data transfer from the DPU to the CPU, (ii) data transfer from the CPU to the DPU, and (iii) computation on the DPU. As shown in Figure 1, as the degree of parallelism increases — through more tasklets per DPU and more DPUs per system — the time spent on DPU computation is significantly reduced [2]. Consequently, the communication between the CPU and DPU becomes the dominant contributor to the total system overhead. Dongjae Lee et al. conducted

a comprehensive analysis from multiple perspectives and revealed that Host CPU ↔ DPU data transfer throughput is significantly suboptimal due to the way PIM is integrated with conventional host systems[3].

Considering the significant proportion of CPU↔DPU transfer overhead in the overall system overhead, this work extends uPIMulator and integrates it into Gem5, alongside the development of a simple runtime library for host-side applications to enable interaction between the CPU and DPU. Moreover, the simulator is equipped with a memory transfer latency model calibrated to approximate real-world hardware, facilitating more accurate simulation-based studies.

| Transfer Size (B) | CPU→DPU Bandwidth (GB/s) | DPU→CPU Bandwidth (GB/s) |
|---|---|---|
| 8 | 0.0002 | 0.0001 |
| 32 | 0.0005 | 0.0003 |
| 128 | 0.0020 | 0.0010 |
| 512 | 0.0050 | 0.0030 |
| 2K | 0.0100 | 0.0060 |
| 8K | 0.0200 | 0.0150 |
| 32K | 0.0500 | 0.0300 |
| 128K | 0.1200 | 0.0600 |
| 512K | 0.2000 | 0.1000 |
| 2M | 0.4000 | 0.1300 |
| 8M | 0.3500 | 0.1200 |
| 32M | 0.3000 | 0.1100 |

Table 1: Sustained bandwidth of CPU→DPU (host main memory to one MRAM bank) and DPU→CPU (one MRAM bank to host main memory) transfers of different sizes for one DPU. Reproduced from G´omez-Luna et al. [2].

# 3 Simulation System Design

To integrate the PIM device into Gem5, we encapsulated it as a custom SimObject and added a corresponding port on the CPU side to enable communication with PIM Simobject. To simulate the workflow of a real UPMEM-PIM system, we implemented a runtime that provides interfaces for CPU–DPU interaction, such as dpu_load and dpu_launch. Within the runtime, we leverage Gem5's built-in pseudo-instruction mechanism (M5Ops) to control CPU–PIM communication, where the CPU interacts with the PIM device by issuing packets. Figure 2 shows an overview of the system.

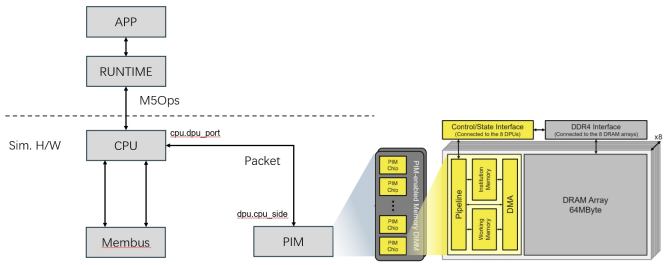



Figure 2: System Overview

To improve the accuracy of the simulation for the CPU↔DPU transportation, a model fitted using experimental data from a real UPMEM-PIM device was used to calculate the communication latency. Table 1 presents the original data, which was obtained through experiments conducted on a real-world UPMEM-PIM system[2]. Figure 3 shows the piecewise spline-fitted functions that model the relationship between transfer size and bandwidth for CPU→DPU and DPU→CPU transfers.

Based on our extended Gem5_uPIMulator and the integrated memory transfer latency model, we successfully reproduced performance behavior that closely matches that of the real device. Figure 4 presents a comparison between the simulation results and those measured on actual hardware. As shown, the trend of execution time with respect to the number of tasklets is nearly identical, and the memory transfer latency values are even more closely aligned.

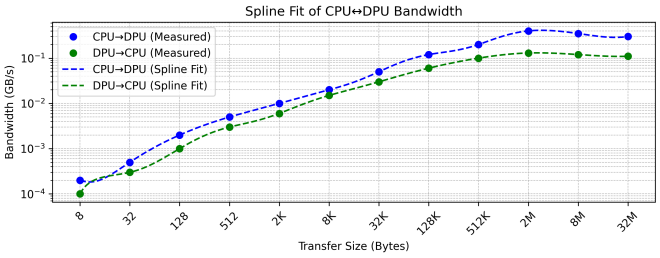
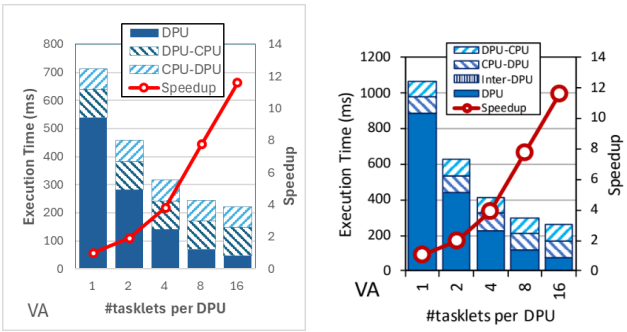



Figure 3: Sustained bandwidth (log scale x- and y-axes) of CPU→DPU (host main memory to one MRAM bank) and DPU→CPU (one MRAM bank to host main memory) transfers of different sizes for one DPU. The curves in are obtained by applying spline fitting to the scatter data presented in Table 1.




(a) Simulation result (b) Real-device result

Figure 4: Comparison of execution time (ms) and DPU-side speedup between simulation and the real device for the VA benchmark. Speedup is computed based on the portion of execution time spent on the DPU and is normalized to the performance of one tasklet per DPU or one DPU. Figures 4.4b is directly taken from Gómez-Luna et al. [2].

# 4 Pipelined Execution

## 4.1 Overview

Consider the execution of a workload on a UPMEM-PIM system. The process typically involves three sequential steps: (i) loading the workload from the host CPU to the DPUs, (ii) executing the workload on the DPUs, and (iii) transferring the computed results back to the CPU. These steps are performed serially, with each step depending on the completion of the previous one. However, if the workload is further partitioned into tasks(hereafter referred to as stream) with smaller data block and executed in a pipelined manner, it can be divided into distinct stages, such as CPU→DPU data transfer and DPU computation. By overlapping the data transfer of one stream with the computation of the previous stream, the CPU→DPU transfer latency can be effectively hidden.

Figure 5 shows how streams-based pipelined execution work compared with sequential execution. For workloads that support data partitioning followed by reduction, a workload can be divided into N streams based on input data blocks and executed in a staged manner. Before entering the stream-based execution phase, the DPU executable should be loaded. Then, the input data for each stream is sequentially transferred to the DPU, and the DPU begins execution for each stream whenever its compute unit becomes idle. The intermediate results of all streams are temporarily stored in the DPU. Once all stream computations are completed, a reduction is performed within the DPU, and the final result is then returned to the CPU.

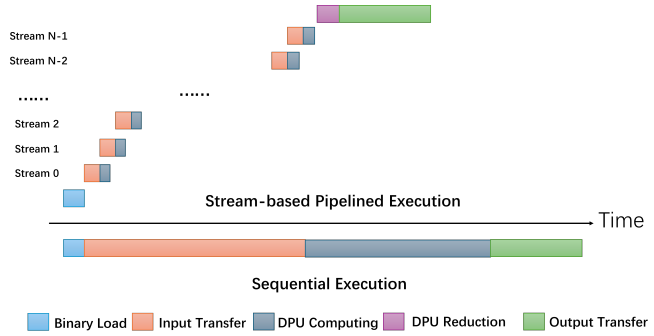



Figure 5: Comparison of Pipelined vs Sequential Execution

## 4.2 Theoretical Performance Modeling

The binary load and output transfer phases have direct dependencies on preceding or succeeding steps and thus cannot be optimized. Therefore, the following discussion focuses only on the input transfer, DPU computing, and DPU reduction processes.

For sequential execution, the input transfer takes $T^{input}_{sequential}$ and the computation takes $T^{computing}_{sequential}$.

In contrast, for stream-based execution with $N$ streams, the $i$-th stream requires $T^{input}_i$ for input transfer, $T^{computing}_i$ for stream computation, and a final $T^{reduction}$ after all streams complete.

Based on this setup, we can derive the following expressions:

$$T_{sequential} = T^{input}_{sequential} + T^{computing}_{sequential}$$

$$T_{streams} = T^{input}_0 + \sum_{i=1}^{N-1} max(T^{input}_i, T^{computing}_{i-1}) + T^{computing}_{N-1} + T^{reduction}$$

$$Speedup = \frac{T_{sequential}}{T_{streams}}$$

Under ideal conditions, data partitioning does not affect either the transfer speed or the computing speed, and each stream takes the same amount of time to execute. In this case, we have:

$$i \in \{1, 2, \ldots, N\}, T^{computing}_i = \frac{T^{computing}_{sequential}}{N}$$

$$i \in \{1, 2, \ldots, N\}, T^{input}_i = \frac{T^{input}_{sequential}}{N}$$

When the computing time and the input transfer time for a workload become approximately equal, the maximum speedup can be achieved:

$$T^{input}_{sequential} = T^{computing}_{sequential} = T$$

$$i \in \{1, 2, \ldots, N\}, T^{input}_i = T^{computing}_i = \frac{T}{N}$$

$$Speedup = \frac{T_{sequential}}{T_{streams}} = \frac{2 \cdot T}{(N+1) \cdot \frac{T}{N} + T^{reduction}} = 2 - \frac{\frac{T}{N} + T_{\text{reduction}}}{T + \frac{T}{N} + T_{\text{reduction}}}$$

This optimization approach provides a speedup upper bound of 2. In the ideal case, input transfer and stream computation times are balanced, $N$ is large, and reduction cost is relatively negligible, then the speedup can closely approach 2.

## 4.3 Performance Evaluation

Based on our previously extended Gem5_uPIMulator, we were able to implement this optimization scheme relatively easily within the simulation environment and conduct thorough experimental evaluations. We selected the VA benchmark, which performs vector addition, as the test case. One notable characteristic of this benchmark is that, after partitioning the workload into streams and executing them independently on the DPU,


Figure 6: Execution time (ms) of benchmarks VA of different input size on 1, 2, 4, 8, and 16 streams in one DPU with 16 tasklets (left y-axis), and speedup (considering only the portion of execution time and input transfer spent on the DPU) provided by more tasklets normalized to the performance of 1 tasklet (right y-axis). Input size refers the number of input elements, and each element is of 4B.

the intermediate results can be stored sequentially in PIM memory. Once computations of all streams are completed, the results can be directly returned to the CPU without requiring an additional reduction step. This significantly simplifies the implementation of the proposed scheme and makes it easier to achieve the ideal speedup under optimal conditions.

Figure 6 presents the breakdown of input transfer and DPU computing times under different input sizes and different number of streams(#streams). "DPU_only" represents the time during which only DPU computation is performed, "Write_only" represents the time spent solely on transferring input data to the PIM ranks, and "Overlap" denotes the period during which both operations are active simultaneously. When the #streams is set to 1, it corresponds to conventional sequential execution. When the #streams is greater than 1, a single workload is partitioned into multiple streams, each executed in a staged, pipelined manner. The speedup is defined as the acceleration factor for completing both the input transfer and DPU computing steps, relative to the sequential execution baseline (#streams = 1).

It can be observed that when the input size is small, increasing the #streams can actually lead to longer overall execution time. This phenomenon is mainly due to two reasons: (1) When the input size is small, the memory bandwidth utilization is also low, as shown in Figure 3, resulting in longer transfer time because the partitioned input blocks are much smaller and thus less bandwidth-efficient. (2) When the input size is small, both DPU computing and input transfer are relatively fast. In this case, the CPU's overhead in continuously offload streams to the DPUs becomes the new performance bottleneck, while the DPU-side computing and data transfer to the rank are comparatively faster. As a result, the overlap between computation and data transfer is significantly reduced or even disappears.

As the input size increases and each stream becomes larger, the advantages of executing streams in a pipelined manner become evident. The partitioned input data remains sufficiently large such that the transfer bandwidth is not significantly impacted. As a result, the overlap between data transfer and DPU computing becomes increasingly prominent, leading to a substantial reduction in overall execution time.

When the input size reaches 4,194,304 elements, corresponding to 16 MB (with each element being 4 bytes), the benefit of stream partitioning becomes particularly significant. With 16 streams, a speedup of 1.92 is achieved, which is very close to the theoretical speedup limit: 2. The near-ideal speedup achieved in this case is attributed to several essential conditions: (1) The DPU computing time and input transfer time for this benchmark are very close, allowing a high degree of overlap between the two operations. (2) This benchmark does not require reduction across the results of different streams, thereby avoiding the additional overhead typically associated with stream partitioning. (3) The input data size is sufficiently

large, such that partitioning into multiple streams does not negatively impact the memory transfer bandwidth.

# 5 Conclusion

We integrated uPIMulator into Gem5 and further extended it to enable CPU–DPU interaction within our Gem5_uPIMulator system. A simple memory transfer latency model was developed based on data from real hardware and incorporated into the simulation framework, resulting in performance outcomes that closely match those of the actual device. These extensions make it possible to study CPU–DPU communication issues through simulation, which is now considered a major source of overhead in commercial PIM systems (e.g., UPMEM-PIM and Samsung HBM-PIM)[3].

In addition, to address the overhead associated with CPU–DPU data transfers, we propose a new execution paradigm that further partitions a single workload into multiple streams and organizes them into staged execution. This pipelined execution model enables overlap between stages, thereby partially hiding the input transfer latency. Through theoretical analysis, we derive an ideal maximum speedup of 2×, and simulation results demonstrate a maximum speedup of 1.92×, highlighting the potential of the proposed approach.

# 6 Future Work

Although the integrated Gem5_uPIMulator system enables interaction between the CPU and DPU, its communication mechanism still differs from that of real hardware. In actual UPMEM-PIM devices, the PIM memory and control interfaces are memory-mapped into the host address space, allowing the host CPU to directly interact with the PIM device by performing memory operations. In contrast, our simulation system relies on Gem5's M5Ops pseudo-instruction mechanism, achieving only a functional-level simulation of the communication process. In the future, it would be worthwhile to design a more general PIM simulation framework within Gem5, where PIM device memory is properly mapped, enabling the CPU to directly control and communicate with the PIM device through regular memory accesses.

Another limitation of the current simulation system is that it does not account for concurrent accesses to the PIM MRAM by both the CPU and DPU cores. The MRAM acts as a single bank, capable of maintaining only one read/write queue at a time. In the actual UPMEM-PIM architecture, a locking mechanism is enforced when accessing MRAM: whether the access comes from the CPU or the DPU, it must be ensured that only one core accesses the MRAM at any given moment. If both the CPU and DPU attempt to access the MRAM simultaneously, the CPU's access requests may be serialized behind the DPU's, potentially causing unpredictable delays and violating DDR protocol timing constraints. This issue warrants further investigation. One potential solution is to equip each DPU with multiple MRAM banks. During CPU offloading, input data could be alternately distributed across two banks, such that the input data required by DPU computing and the input data being transferred for the next stream reside in separate banks. This would allow the CPU-to-DPU data transfer and the DPU computation to proceed concurrently without contention.

Another challenge introduced by stream-based execution is the need for a memory address management scheme within the PIM system. Specifically, it is necessary to manage how the same executable binary handles different input data addresses and temporary result storage addresses for each stream during execution. This aspect of the system design is not covered in this work and is left for future investigation.

# References


[1] Bongjoon Hyun, Taehun Kim, Dongjae Lee, and Minsoo Rhu. Pathfinding future pim architectures by demystifying a commercial pim technology. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 263–279. IEEE, 2024.

[2] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F Oliveira, and Onur Mutlu. Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture. *arXiv preprint arXiv:2105.03814*, 2021.

[3] Dongjae Lee, Bongjoon Hyun, Taehun Kim, and Minsoo Rhu. Analysis of data transfer bottlenecks in commercial pim systems: A study with upmem-pim. *IEEE Computer Architecture Letters*, 23(2):179–182, 2024.