Processing-Near-Memory
Real PNM Architectures
Programming General-purpose PIM

Dr. Juan Gómez Luna
Professor Onur Mutlu

Sunday, March 26, 2023
## Two PIM Approaches

### 5.2. Two Approaches: Processing Using Memory (PUM) vs. Processing Near Memory (PNM)

Many recent works take advantage of the memory technology innovations that we discuss in Section 5.1 to enable and implement PIM. We find that these works generally take one of two approaches, which are categorized in Table 1: (1) *processing using memory* or (2) *processing near memory*. We briefly describe each approach here. Sections 6 and 7 will provide example approaches and more detail for both.

Table 1: Summary of enabling technologies for the two approaches to PIM used by recent works. Adapted from [341] and extended.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Example Enabling Technologies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Using Memory</td>
<td>SRAM, DRAM, Phase-change memory (PCM), Magnetic RAM (MRAM), Resistive RAM (RRAM)/memristors</td>
</tr>
<tr>
<td>Processing Near Memory</td>
<td>Logic layers in 3D-stacked memory, Silicon interposers, Logic in memory controllers, Logic in memory chips (e.g., near bank), Logic in memory modules, Logic near caches, Logic near/in storage devices</td>
</tr>
</tbody>
</table>


PIM Becomes Real

- UP MEM, founded in January 2015, announces the first real-world PIM architecture in 2016
- UP MEM’s PIM-enabled DIMMs start getting commercialized in 2019

- In early 2021, Samsung announces FIMDRAM at ISSCC conference
- Samsung’s LP-DDR5 and DIMM-based PIM announced a few months later

- In early 2022, SK Hynix announces AiM and Alibaba announces HB-PNM at ISSCC conference
UPMEM PIM
UPMEM Processing-in-DRAM Engine (2019)

- Processing in DRAM Engine
- Includes **standard DIMM modules**, with a **large number of DPU processors** combined with DRAM chips.

- Replaces **standard** DIMMs
  - DDR4 R-DIMM modules
    - 8GB+128 DPUs (16 PIM chips)
    - Standard 2x-nm DRAM process
  - **Large amounts of** compute & memory bandwidth

UPMEM DIMMs

- E19: 8 chips/DIMM (1 rank). DPUs @ 267 MHz
- P21: 16 chips/DIMM (2 ranks). DPUs @ 350 MHz
2,560-DPU Processing-in-Memory System

Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture

JUAN GÓMEZ-LUNA, ETH Zurich, Switzerland
IZZAT EL HAJJ, American University of Beirut, Lebanon
IVAN FERNANDEZ, ETH Zurich, Switzerland and University of Malaga, Spain
CHRISTINA GIANNOLA, ETH Zurich, Switzerland and NTUA, Greece
GERALD F. OLIVEIRA, ETH Zurich, Switzerland
ONUR MUTLU, ETH Zurich, Switzerland

Many modern workloads, such as neural networks, databases, and graph processing, are fundamentally memory-bound. For such workloads, the data movement between main memory and CPU cores imposes a significant overhead in terms of both latency and energy. A major reason is that this communication happens through a narrow bus with high latency and limited bandwidth, and the low data reuse in memory-bound workloads is inefficient to amortize the cost of main memory access. Fundamentally addressing this data movement bottleneck requires a paradigm where the memory system assumes an active role in computing by integrating processing capabilities. This paradigm is known as processing-in-memory (PIM).

Recent research explores different forms of PIM architectures, motivated by the emergence of new 3D-stacked memory technologies that integrate memory with a logic layer where processing elements can be easily placed. Past works evaluate these architectures in simulation or, at best, with simplified hardware prototypes. In contrast, the UPNEM company has designed and manufactured the first publicly-available real-world PIM architecture. The UPNEM PIM architecture combines traditional DRAM memory arrays with general-purpose in-order cores, called DPU (Processing Units (DPU)), integrated in the same chip.

This paper provides the first comprehensive analysis of the first publicly-available real-world PIM architecture. We make two key contributions. First, we conduct an experimental characterization of the UPNEM-based PIM system using microbenchmarks to assess various architecture limits such as compute throughput and memory bandwidth, yielding new insights. Second, we present PIM (Processing-in-Memory benchmark), a benchmark suite of 16 workloads from different application domains (e.g., deep-space linear algebra, databases, data analytics, graph processing, neural networks, bioinformatics, image processing), which we identify as memory-bound. We evaluate the performance and scaling characteristics of PIM workloads on the UPNEM PIM architecture, and compare their performance and energy consumption to their state-of-the-art CPU and GPU counterparts. Our extensive evaluation conducted on two real UPNEM-based PIM systems with 640 and 2,560 DPU provides new insights about suitability of different workloads to the PIM system, programming recommendations for software designers, and suggestions and hints for hardware and architecture designers of future PIM systems.

Understanding a Modern PIM Architecture

Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System

JUAN GÓMEZ-LUNA\textsuperscript{1}, IZZAT EL HAJJ\textsuperscript{2}, IVAN FERNANDEZ\textsuperscript{1,3}, CHRISTINA GIANNOULA\textsuperscript{1,4}, GERALDO F. OLIVEIRA\textsuperscript{1}, AND ONUR MUTLU\textsuperscript{1}

\textsuperscript{1}ETH Zürich
\textsuperscript{2}American University of Beirut
\textsuperscript{3}University of Malaga
\textsuperscript{4}National Technical University of Athens

Corresponding author: Juan Gómez-Luna (e-mail: juang@ethz.ch).

https://github.com/CMU-SAFARI/prim-benchmarks
A memory circuit having: a memory array including one or more memory banks; a first processor; and a processor control interface for receiving data processing commands directed to the first processor from a central processor, the processor control interface being adapted to indicate to the central processor when the first processor has finished accessing one or more of the memory banks of the memory array, these memory banks becoming accessible to the central processor.
UPMEM PIM System Organization (I)

- FIG. 1 schematically illustrates a computing system comprising DRAM circuits having integrated processors according to an example embodiment.

Fig 1

In a UPMEM-based PIM system, UPMEM DIMMs coexist with regular DDR4 DIMMs.
UPMEM PIM System Organization (III)

- A UPMEM DIMM contains 8 or 16 chips
  - Thus, 1 or 2 ranks of 8 chips each
- Inside each PIM chip there are:
  - 8 64MB banks per chip: Main RAM (MRAM) banks
  - 8 DRAM Processing Units (DPUs) in each chip, 64 DPUs per rank
DRAM Processing Unit (I)

- FIG. 4 schematically illustrates part of the computing system of FIG. 1 in more detail according to an example embodiment.
DRAM Processing Unit (II)

**PIM Chip**

- Control/Status Interface
- DDR4 Interface
- 24-KB IRAM
- 64-KB WRAM
- 64-bit Pipeline
- 64-MB DRAM Bank (MRAM)
- DMA Engine
- x8 Memory Width
DPU Pipeline

• In-order pipeline
  - Up to 425 MHz

• Fine-grain multithreaded
  - 24 hardware threads

• 14 pipeline stages
  - DISPATCH: Thread selection
  - FETCH: Instruction fetch
  - READOP: Register file
  - FORMAT: Operand formatting
  - ALU: Operation and WRAM
  - MERGE: Result formatting
Fine-grained Multithreading
Fine-Grained Multithreading (I)

• Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
  - Branch/instruction resolution latency overlapped with execution of other threads’ instructions

+ No logic needed for handling control and data dependences within a thread
  -- Single thread performance suffers
  -- Extra logic for keeping thread contexts
  -- Does not overlap latency if not enough threads to cover the whole pipeline
Fine-Grained Multithreading (II)

- Idea: Switch to another thread every cycle such that no two instructions from a thread are in the pipeline concurrently.

- Tolerates the control and data dependence latencies by overlapping the latency with useful work from other threads.

- Improves pipeline utilization by taking advantage of multiple threads.

- Thornton, “Parallel Operation in the Control Data 6600,” AFIPS 1964

- Smith, “A pipelined, shared resource MIMD computer,” ICPP 1978
Lecture on Fine-Grained Multithreading

Fine-Grained Multithreading

- Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
  - Branch/instruction resolution latency overlapped with execution of other threads’ instructions

+ No logic needed for handling control and data dependences within a thread
  -- Single thread performance suffers
  -- Extra logic for keeping thread contexts
  -- Does not overlap latency if not enough threads to cover the whole pipeline
DPU Pipeline

• In-order pipeline
  - Up to 425 MHz

• Fine-grain multithreaded
  - 24 hardware threads

• 14 pipeline stages
  - DISPATCH: Thread selection
  - FETCH: Instruction fetch
  - READOP: Register file
  - FORMAT: Operand formatting
  - ALU: Operation and WRAM
  - MERGE: Result formatting
DPU Instruction Set Architecture

- Specific 32-bit ISA
  - Aiming at scalar, in-order, and multithreaded implementation
  - Allowing compilation of 64-bit C code
  - LLVM/Clang compiler

Instruction Set Architecture

This section covers the architecture concepts required to understand and use UPMEM DPU processor as a software developer. It is also providing an exhaustive list of the available processor instructions.

Software developers should use this section as a reference manual to develop or debug assembly code.

Resources overview

Thread registers

The system is composed of 24 hardware threads. Each of them owns a set of private resources:

- 24 general purpose 32-bits registers named $r0$ through $r23$
- A 16-bits wide program counter, named PC. Notice that the PC value does not address an instruction in memory, but the index of such an instruction directly. For example, a PC equal to 1 represents the second instruction in the DPU's program memory.
- Two persistent flags, keeping information about the previous result of an arithmetic or logical instruction:
  - ZF: last result is equal to zero

https://sdk.upmem.com/2021.2.0/201_IS.html#
### Microbenchmark for INT32 ADD Throughput

**C-based code**

```c
#define SIZE 256
int* bufferA = mem_alloc(SIZE * sizeof(int));
for(int i = 0; i < SIZE; i++){
    int temp = bufferA[i];
    temp += scalar;
    bufferA[i] = temp;
}
```

**Compiled code (UPMEM DPU ISA)**

```assembly
move r2, 0
.LBB0_1:   // Loop header
    lsl_add r3, r0, r2, 2  // Address calculation
    lw r4, r3, 0          // Load from WRAM
    add r4, r4, r1        // Add
    sw r3, 0, r4          // Store to WRAM
    add r2, r2, 1         // Index update
    jneq r2, 256, .LBB0_1  // Conditional jump
```
More on the UPMEM PIM Architecture

2,560-DPU System (1)

- UPMEM-based PIM system with 20 UPMEM DIMMs of 16 chips each (40 ranks)
  - P21 DIMMs
  - Dual x86 socket
  - UPMEM DIMMs coexist with regular DDR4 DIMMs
  - 2 memory controllers/socket (3 channels each)
  - 2 conventional DDR4 DIMMs on one channel of one controller

Processing-in-Memory Course: Lecture 2: Real-world PIM: UPMEM PIM Architecture - Spring 2022

669 views • Premiered Mar 17, 2022

Onur Mutlu Lectures
25.9K subscribers
Understanding a Modern PIM Architecture

Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System

JUAN GÓMEZ-LUNA¹, IZZAT EL HAJJ², IVAN FERNANDEZ¹,³, CHRISTINA GIANNELLA¹,⁴, GERALDO F. OLIVEIRA¹, AND ONUR MUTLU¹

¹ETH Zürich
²American University of Beirut
³University of Malaga
⁴National Technical University of Athens

Corresponding author: Juan Gómez-Luna (e-mail: juang@ethz.ch).

https://github.com/CMU-SAFARI/prim-benchmarks
KEY TAKEAWAY 1

The UPMEM PIM architecture is fundamentally compute bound. As a result, the most suitable workloads are memory-bound.
The UPMEM-based PIM system can outperform a state-of-the-art GPU on workloads with three key characteristics:

1. Streaming memory accesses
2. No or little inter-DPU synchronization
3. No or little use of integer multiplication, integer division, or floating point operations

These three key characteristics make a workload potentially suitable to the UPMEM PIM architecture.
**Key Takeaway 2**

The most well-suited workloads for the UPMEM PIM architecture use no arithmetic operations or use only simple operations (e.g., bitwise operations and integer addition/subtraction).
**KEY TAKEAWAY 3**

The most well-suited workloads for the UPMEM PIM architecture require little or no communication across DPUs (inter-DPU communication).
Key Takeaway 4

**KEY TAKEAWAY 4**

- UPMEM-based PIM systems **outperform state-of-the-art CPUs in terms of performance** (by 23.2× on 2,556 DPUs for 16 PrIM benchmarks) and **energy efficiency on most of PrIM benchmarks**.

- UPMEM-based PIM systems **outperform state-of-the-art GPUs on a majority of PrIM benchmarks** (by 2.54× on 2,556 DPUs for 10 PrIM benchmarks), and the outlook is even more positive for future PIM systems.

- UPMEM-based PIM systems are **more energy-efficient than state-of-the-art CPUs and GPUs on workloads that they provide performance improvements** over the CPUs and the GPUs.
PrIM Repository

- All microbenchmarks, benchmarks, and scripts
- [https://github.com/CMU-SAFARI/prim-benchmarks](https://github.com/CMU-SAFARI/prim-benchmarks)

PrIM (Processing-In-Memory Benchmarks)

PrIM is the first benchmark suite for a real-world processing-in-memory (PIM) architecture. PrIM is developed to evaluate, analyze, and characterize the first publicly-available real-world processing-in-memory (PIM) architecture, the UPMEM PIM architecture. The UPMEM PIM architecture combines traditional DRAM memory arrays with general-purpose in-order cores, called DRAM Processing Units (DPUs), integrated in the same chip.

PrIM provides a common set of workloads to evaluate the UPMEM PIM architecture with and can be useful for programming, architecture and system researchers all alike to improve multiple aspects of future PIM hardware and software. The workloads have different characteristics, exhibiting heterogeneity in their memory access patterns, operations and data types, and communication patterns. This repository also contains baseline CPU and GPU implementations of PrIM benchmarks for comparison purposes.

PrIM also includes a set of microbenchmarks can be used to assess various architecture limits such as compute throughput and memory bandwidth.
Samsung FIMDRAM
(aka HBM-PIM)
Samsung Develops Industry’s First High Bandwidth Memory with AI Processing Power

The new architecture will deliver over twice the system performance and reduce energy consumption by more than 70%

Samsung Electronics, the world leader in advanced memory technology, today announced that it has developed the industry’s first High Bandwidth Memory (HBM) integrated with artificial intelligence (AI) processing power — the HBM-PIM. The new processing-in-memory (PIM) architecture brings powerful AI computing capabilities inside high-performance memory, to accelerate large-scale processing in data centers, high performance computing (HPC) systems and AI-enabled mobile applications.

Kwangil Park, senior vice president of Memory Product Planning at Samsung Electronics stated, “Our groundbreaking HBM-PIM is the industry’s first programmable PIM solution tailored for diverse AI-driven workloads such as HPC, training and inference. We plan to build upon this breakthrough by further collaborating with AI solution providers for even more advanced PIM-powered applications.”
Samsung Function-in-Memory DRAM (2021)

- FIMDRAM based on HBM2

![3D Chip Structure of HBM with FIMDRAM]

**Chip Specification**

- 128DQ / 8CH / 16 banks / BL4
- 32 PCU blocks (1 FIM block/2 banks)
- 1.2 TFLOPS (4H)
- FP16 ADD / Multiply (MUL) / Multiply-Accumulate (MAC) / Multiply-and-Add (MAD)

**ISSCC 2021 / SESSION 25 / DRAM / 25.4**

A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications

Young-Choon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, Seongil O, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, Youngmin Choi, Jin Guk Kim, Jongyoon Choi, Hyun-Sung Shin, Jin Kim, BengSeng Phua, HyungMin Kim, Myeong Jun Song, Ahn Choi, Daeho Kim, SooYoung Kim, Eun-Bong Kim, David Wang, Shinhaeng Kang, Yuhwan Ro, Seungwoo Seo, JoonHo Song, Jaeyoun Youn, Kyomin Sohn, Nam Sung Kim

1Samsung Electronics, Hwasung, Korea
2Samsung Electronics, San Jose, CA
3Samsung Electronics, Suwon, Korea
Samsung Function-in-Memory DRAM (2021)

Chip Implementation

- Mixed design methodology to implement FIMDRAM
  - Full-custom + Digital RTL
FIMDRAM: System Organization

• PIM units respond to standard DRAM column commands (RD or WR)
  - Compliant with unmodified JEDEC controllers

• They execute one wide-SIMD operation commanded by a PIM instruction with deterministic latency in a lock-step manner

• A PIM unit can get 16 16-bit operands from IOSAs, a register, and/or the result bus

Lee et al., Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology, ISCA 2021
Lecture on FIMDRAM/HBM-PIM

FIMDRAM: Bank-level Parallelism

- Unlike standard DRAM devices, all banks can be accessed concurrently for 8x higher bandwidth (with 16 pCHs)
- In AB-PIM mode, a memory command triggers a PIM instruction in the CRF

Processing-in-Memory Course: Lecture 4: Real-world PIM: Samsung HBM-PIM Architecture - Spring 2022

673 views • Streamed live on Mar 31, 2022

https://youtu.be/_CpWJGK9N04
Samsung AxDIMM
Samsung AxDIMM (2021)

- DIMM-based PIM
  - DLRM recommendation system

AxDIMM Design: Hardware Architecture

Figure 3.1 Hardware Architecture

Standard DIMM Interface

DDDR4 slave PHY receives DRAM commands and NMP instructions (via DQ pins) from the host side
AxDIMM Design: Execution Flow

Decoder
- Decode Inst

Data Fetch
- RD Psum
- RD Emb

Adder
- ADD

Accumulate
- WR Psum

Mode Change
- WR Inst
- SLS Execute

RD Status Reg

RD Psum

Host
- WR Emb Table

Ram-1

Rank-0

INST BUF
DEC
CMDGEN
ADDER

Conf Reg

Psum Buf

Acc Mode Enable
Set SLS ExeReg
Read StatusReg

DDR4 Slave PHY

Data Path

Emb Table Data

Decode Inst

Accumulate Psum_{k+1}

RD Psum

RD Emb

DRAM Cmd

Input UF

Rank-0.NMP

Rank-1.NMP

Mig (PHY)

SAFARI
Lecture on AxDIMM

Processing-in-Memory Course: Lecture 9: Real-world PIM: Samsung AxDIMM - Spring 2022

https://youtu.be/J_prUKfnv7Q
SK Hynix AiM
SK hynix Develops PIM, Next-Generation AI Accelerator

February 16, 2022

Seoul, February 16, 2022

SK hynix (or “the Company", www.skhynix.com) announced on February 16 that it has developed PIM*, a next-generation memory chip with computing capabilities.

*PIM (Processing In Memory): A next-generation technology that provides a solution for data congestion issues for AI and big data by adding computational functions to semiconductor memory

It has been generally accepted that memory chips store data and CPU or GPU, like human brain, process data. SK hynix, following its challenge to such notion and efforts to pursue innovation in the next-generation smart memory, has found a breakthrough solution with the development of the latest technology.

SK hynix plans to showcase its PIM development at the world’s most prestigious semiconductor conference, 2022 ISSCC*, in San Francisco at the end of this month. The company expects continued efforts for innovation of this technology to bring the memory-centric computing, in which semiconductor memory plays a central role, a step closer to the reality in devices such as smartphones.

*ISSCC: The International Solid-State Circuits Conference will be held virtually from Feb. 20 to Feb. 24 this year with a theme of “Intelligent Silicon for a Sustainable World”

For the first product that adopts the PIM technology, SK hynix has developed a sample of GDDR6-AiM (Accelerator” in memory). The GDDR6-AiM adds computational functions to GDDR6 memory chips, which process data at 16Gbps. A combination of GDDR6-AiM with CPU or GPU instead of a typical DRAM makes certain computation speed 16 times faster. GDDR6-AiM is widely expected to be adopted for machine learning, high-performance computing, and big data computation and storage.

SK Hynix Accelerator-in-Memory (2022)

- 4 Gb AiM die with 16 processing units (PUs)

AiM Die Photograph

1 Process Unit (PU) Area

<table>
<thead>
<tr>
<th>Area Type</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>0.19mm²</td>
</tr>
<tr>
<td>MAC</td>
<td>0.11mm²</td>
</tr>
<tr>
<td>Activation Function (AF)</td>
<td>0.02mm²</td>
</tr>
<tr>
<td>Reservoir Cap.</td>
<td>0.05mm²</td>
</tr>
<tr>
<td>Etc.</td>
<td>0.01mm²</td>
</tr>
</tbody>
</table>

MAC 58%

AF 11%

Reservoir Cap. 26%

Etc. 5%
SK Hynix AiM: System Organization (2022)

- GDDR6-based AiM architecture
Lecture on Accelerator-in-Memory

AiM: Adder Tree: Bank-wide Mantissa Shift (BWMS)

- Find MAX EX of 16 EXs
- Obtain the differences
- Shift all MAs by the differences
- Perform MA additions

Processing-in-Memory Course: Lecture 6: Real-world PIM: SK Hynix AiM - Spring 2022

607 views • Streamed live on Apr 14, 2022

https://youtu.be/NDL77Xdccbs?t=159
Alibaba HB-PNM
Alibaba HB-PNM: Overall Architecture (2022)

- 3D-stacked logic die and DRAM die vertically bonded by hybrid bonding (HB)
Alibaba HB-PNM: Compute Engines

- Match engine and neural engine for matching and ranking in a recommendation system
Lecture on HB-PNM

Match Engine: Distance Calculator

- Distance calculator obtains similarity between input query and feature vectors
  - It computes Hamming distance of two 512-bit vectors
  - Distance is filtered by root of max-heap

Processing-in-Memory Course: Lecture 10: Real-world PIM: Alibaba HB-PNM - Spring 2022

410 views • Premiered May 12, 2022

https://youtu.be/OZjKnn-DbwA
More Real PIM
NeuroBlad®es build a processing-in-memory analytics chip and server

By Chris Mellor - October 6, 2021

An Israeli startup called NeuroBlad®e has exited stealth mode, built a processing-in-memory (PIM) analytics chip combining DRAM and thousands of cores, put four of them in an analytics accelerating server appliance box, and taken in $83 million in B-round funding.

The idea is to take a GPU approach to big data-style analytics and AI software by employing a massively parallel core design, but take it further by layering the cores on DRAM with a wide I/O bus architecture design linking the cores and memory to speed processing even more. This design vastly reduces data movement between storage and memory and also accelerates data transfer between memory and processing cores.
**ABSTRACT**

Distributed processors and methods for compiling code for execution by distributed processors are disclosed. In one implementation, a distributed processor may include a substrate; a memory array disposed on the substrate; and a processing array disposed on the substrate. The memory array may include a plurality of discrete memory banks, and the processing array may include a plurality of processor subunits, each one of the processor subunits being associated with a corresponding, dedicated one of the plurality of discrete memory banks. The distributed processor may further include a first plurality of buses, each connecting one of the plurality of processor subunits to its corresponding, dedicated memory bank, and a second plurality of buses, each connecting one of the plurality of processor subunits to another of the plurality of processor subunits.
NeuroBlade Patent (II)

Sity et al., “Memory-based Distributed Processor Architecture,” US 10,762,034 B2
NeuroBlade: Xiphos

- PIM XRAM chip
  - IMPU (Intensive Memory Processing Unit)
- x86 CPU, 32 NVMe SSDs
- PCIe fabric: "Everything is connected on top of PCIe fabric."
- Wide I/O bus: multiple x16 PCIe buses
Variety of Current Real PIM Architectures

• Differences

- Near-bank (UPMEM, FIMDRAM, AiM, HB-PNM) vs. near-chip (AxDIMM)
- General-purpose (UPMEM) vs. special-function (FIMDRAM, AiM, HB-PNM)
- FGMT (UPMEM) vs. SIMD (FIMDRAM, AiM, AxDIMM) vs. systolic array (HB-PNM)
- Natively integer (UPMEM, HB-PNM) vs. floating point (FIMDRAM)
  • FP16 (FIMDRAM) vs. BF16 (AiM) vs. FP32 (AxDIMM)
- DDR4 (UPMEM, AxDIMM) vs. LPDDR4 (HB-PNM) vs. HBM2 (FIMDRAM) vs. GDDR6 (AiM)
Common Characteristics

- These PIM systems have **some common characteristics**:

  1. There is a **host processor** (CPU or GPU) with access to (1) standard main memory, and (2) PIM-enabled memory.

  2. PIM-enabled memory contains **multiple PIM processing elements** (PEs) with high bandwidth and low latency memory access.

  3. PIM PEs run only at **a few hundred MHz** and have a small number of registers and small (or no) cache/scratchpad.

  4. PEs may need to communicate via the **host processor**.
A State-of-the-Art PIM (PNM) System

• These PIM systems have some common characteristics:
  1. There is a host processor (CPU or GPU) with access to (1) standard main memory, and (2) PIM-enabled memory
  2. PIM-enabled memory contains multiple PIM processing elements (PEs) with high bandwidth and low latency memory access
  3. PIM PEs run only at a few hundred MHz and have a small number of registers and small (or no) cache/scratchpad
  4. PEs may need to communicate via the host processor
Programming a General-purpose PIM System
Accelerator Model (I)

• Integration of UPMEM DIMMs in a system follows an **accelerator model**

• UPMEM DIMMs coexist with conventional DIMMs

• UPMEM DIMMs can be seen as a **loosely coupled accelerator**
  - Explicit data movement between the main processor (host CPU) and the accelerator (UPMEM)
  - Explicit kernel launch onto the UPMEM processors

• This resembles GPU computing
GPU Computing

• Computation is **offloaded to the GPU**

• Three steps
  - CPU-GPU data transfer (1)
  - GPU kernel execution (2)
  - GPU-CPU data transfer (3)

https://www.youtube.com/watch?v=y40-tY5WJ8A
Accelerator Model (II)

- FIG. 6 is a flow diagram representing operations in a method of delegating a processing task to a DRAM processor according to an example embodiment.

![Flow diagram](image)

- SOC LOADS DATA TO BE PROCESSED TO DRAM MEMORY BANK
- SOC TRANSMITS DATA PROCESSING COMMAND TO DRAM PROCESSOR(S)
- DATA PROCESSING BY DRAM PROCESSOR(S)
- DATA PROCESSING COMPLETE?
- MEMORY BANK ACCESSIBLE BY SOC

Fig 6

System Organization

- **FIG. 1** schematically illustrates a computing system comprising DRAM circuits having integrated processors according to an example embodiment.

First Programming Example: Vector Addition
Observations, Recommendations, Takeaways

**GENERAL PROGRAMMING RECOMMENDATIONS**

1. Execute on the DRAM Processing Units (DPUs) portions of parallel code that are as long as possible.
2. Split the workload into independent data blocks, which the DPUs operate on independently.
3. Use as many working DPUs in the system as possible.
4. Launch at least 11 tasklets (i.e., software threads) per DPU.

**PROGRAMMING RECOMMENDATION 1**

For data movement between the DPU’s MRAM bank and the WRAM, use large DMA transfer sizes when all the accessed data is going to be used.

**KEY OBSERVATION 7**

Larger CPU-DPU and DPU-CPU transfers between the host main memory and the DRAM Processing Unit’s Main memory (MRAM) banks result in higher sustained bandwidth.

**KEY TAKEAWAY 1**

The UPMEM PIM architecture is fundamentally compute bound. As a result, the most suitable work-loads are memory-bound.
Vector Addition (VA)

• Our first programming example
• We partition the input arrays across:
  - DPUs
  - Tasklets, i.e., software threads running on a DPU
User Manual

Getting started

- The UPMEM DPU toolchain
  - Notes before starting
  - The toolchain purpose
  - dpu-upmem-dpurte-clang
    - Limitations
  - The DPU Runtime Library
  - The Host Library
  - dpu-lldb
- Installing the UPMEM DPU toolchain
  - Dependencies
    - Python
  - Installation packages
    - Installation from tar.gz binary archive
  - Functional simulator
- Hello World! Example
  - Purpose
  - Writing and building the program
General Programming Recommendations

- From UPMEM programming guide*, presentations★, and white papers☆

**GENERAL PROGRAMMING RECOMMENDATIONS**

1. Execute on the *DRAM Processing Units (DPUs)* portions of parallel code that are as long as possible.
2. Split the workload into independent data blocks, which the DPUs operate on independently.
3. Use as many working DPUs in the system as possible.
4. Launch at least **11 tasklets (i.e., software threads)** per DPU.

★ F. Devaux, "The true Processing In Memory accelerator," HotChips 2019. doi: 10.1109/HOTCHIPS.2019.8875680
☆ UPMEM, “Introduction to UPMEM PIM. Processing-in-memory (PIM) on DRAM Accelerator,” White paper
DPU Allocation

• `dpu_alloc()` allocates a number of DPUs
  - Creates a `dpu_set`

```c
struct dpu_set_t dpu_set, dpu;
uint32_t nr_of_dpus;

// Allocate DPUs
DPU_ASSERT(dpu_alloc(NR_DPUS, NULL, &dpu_set));

DPU_ASSERT(dpu_get_nr_dpus(dpu_set, &nr_of_dpus));
printf("Allocated %d DPU(s)\n", nr_of_dpus);
```

Can we allocate different DPU sets over the course of a program?

Yes, we can. We show an example next

We deallocate a DPU set with `dpu_free()`
DPU Allocation: Needleman-Wunsch (NW)

• In NW we change the number of DPUs in the DPU set as computation progresses

```c
// Top-left computation on DPUs
for (unsigned int blk = 1; blk <= (max_cols-1)/BL; blk++) {
    // If nr_of_blocks are lower than max_dpus,
    // set nr_of_dpuses to be equal with nr_of_blocks
    unsigned nr_of_blocks = blk;
    if (nr_of_blocks < max_dpus) {
        DPU_ASSERT(dpu_free(dpu_set));
        DPU_ASSERT(dpu_alloc(nr_of_blocks, NULL, &dpu_set));
        DPU_ASSERT(dpu_load(dpu_set, DPU_BINARY, NULL));
        DPU_ASSERT(dpu_get_nr_dpuses(dpu_set, &nr_of_dpuses));
    } else if (nr_of_dpuses == max_dpus) {
        ;
    } else {
        DPU_ASSERT(dpu_free(dpu_set));
        DPU_ASSERT(dpu_alloc(max_dpus, NULL, &dpu_set));
        DPU_ASSERT(dpu_load(dpu_set, DPU_BINARY, NULL));
        DPU_ASSERT(dpu_get_nr_dpuses(dpu_set, &nr_of_dpuses));
    }
    ...
}
```
Load DPU Binary

- `dpu_load()` loads a program in all DPUs of a `dpu_set`.

```c
// Define the DPU Binary path as DPU_BINARY here
 ifndef DPU_BINARY
 #define DPU_BINARY "./bin/dpu_code"
 endif

...

// Load binary
DPU_ASSERT(dpu_load(dpu_set, DPU_BINARY, NULL));
```

Is it possible to launch different kernels onto different DPUs?

Yes, it is possible. This enables:

- Workloads with **task-level parallelism**
- Different programs using different DPU sets
CPU-DPU/DPU-CPU Data Transfers

- CPU-DPU and DPU-CPU transfers
  - Between host CPU’s main memory and DPUs’ MRAM banks

- Serial CPU-DPU/DPU-CPU transfers:
  - A single DPU (i.e., 1 MRAM bank)

- Parallel CPU-DPU/DPU-CPU transfers:
  - Multiple DPUs (i.e., many MRAM banks)

- Broadcast CPU-DPU transfers:
  - Multiple DPUs with a single buffer
Serial Transfers

- `dpu_copy_to();`
- `dpu_copy_from();`
- We transfer (part of) a buffer to/from each DPU in the `dpu_set`
- `DPU_MRAM_HEAP_POINTER_NAME`: Start of the MRAM range that can be freely accessed by applications
  - We do not allocate MRAM explicitly
Parallel Transfers

• We push different buffers to/from a DPU set in one transfer
  - All buffers need to be of the same size

• First, prepare `(dpu_prepare_xfer);` then, push `(dpu_push_xfer)`

• Direction:
  - `DPU_XFER_TO_DPU`
  - `DPU_XFER_FROM_DPU`
Broadcast Transfers

• `dpu_broadcast_to();`
  - Only CPU to DPU

• We transfer the same buffer to all DPUs in the `dpu_set`

```c
// Pointer to main memory
DPU_ASSERT(dpu_broadcast_to(dpu_set, DPU_MRAM_HEAP_POINTER_NAME, 0, bufferA, input_size_dpu * sizeof(T), DPU_XFER_DEFAULT));
```
Different Types of Transfers in a Program

• An example benchmark that uses both parallel and serial transfers

• Select (SEL)
  - Remove even values

![Diagram of parallel and serial transfers]

**Parallel transfers**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPU 0</td>
<td>1</td>
</tr>
<tr>
<td>DPU 1</td>
<td>3</td>
</tr>
<tr>
<td>DPU 2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Serial transfers**

Predicate: True if it is even
Inter-DPU Communication

- There is **no direct communication channel between DPUs**

- **Inter-DPU communication takes place via the host CPU** using CPU-DPU and DPU-CPU transfers

- **Example communication patterns:**
  - Merging of partial results to obtain the final result
    - Only DPU-CPU transfers
  - Redistribution of intermediate results for further computation
    - DPU-CPU transfers and CPU-DPU transfers
How Fast are these Data Transfers?

• With a microbenchmark, we obtain the sustained bandwidth of all types of CPU-DPU and DPU-CPU transfers

• Two experiments:
  - 1 DPU: variable CPU-DPU and DPU-CPU transfer size (8 bytes to 32 MB)
  - 1 rank: 32 MB CPU-DPU and DPU-CPU transfers to/from a set of 1 to 64 MRAM banks within the same rank

• Preliminary experiments with more than one rank
  - Channel-level parallelism

**DDR4 bandwidth** bounds the maximum transfer bandwidth

The cost of the transfers can be amortized, if enough computation is run on the DPUs
CPU-DPU/DPU-CPU Transfers: 1 DPU

- Data transfer size varies between 8 bytes and 32 MB

**KEY OBSERVATION 7**

Larger CPU-DPU and DPU-CPU transfers between the host main memory and the DRAM Processing Unit’s Main memory (MRAM) banks result in higher sustained bandwidth.
CPU-DPU/DPU-CPU Transfers: 1 Rank (I)

- **CPU-DPU** (serial/parallel/broadcast) and **DPU-CPU** (serial/parallel)
- The number of DPUs varies between 1 and 64

![Graph showing sustained bandwidth](image)

**KEY OBSERVATION 8**

The sustained bandwidth of parallel CPU-DPU and DPU-CPU transfers between the host main memory and the DRAM Processing Unit’s Main memory (MRAM) banks increases with the number of DRAM Processing Units inside a rank.
CPU-DPU/DPU-CPU Transfers: 1 Rank (II)

- CPU-DPU (serial/parallel/broadcast) and DPU-CPU (serial/parallel)
- The number of DPUs varies between 1 and 64

**KEY OBSERVATION 9**

The sustained bandwidth of parallel CPU-DPU transfers is higher than the sustained bandwidth of parallel DPU-CPU transfers due to different implementations of CPU-DPU and DPU-CPU transfers in the UPMEM runtime library.

The sustained bandwidth of broadcast CPU-DPU transfers (i.e., the same buffer is copied to multiple MRAM banks) is higher than that of parallel CPU-DPU transfers (i.e., different buffers are copied to different MRAM banks) due to higher temporal locality in the CPU cache hierarchy.
"Transposing" Library

The library feeds DPUs with correct data

Eight 64-bit “horizontal” words are turned into 8 vertical words, feeding 8 different DRAM chips
This way DPUs see full 64-bit words, not chunk of them

| Word 0 | W | W |
| Word 1 | o | o |
| Word 2 | r | r |
| Word 3 | d | d |
| Word 4 | 0 | 1 |
| Word 5 | 2 | 3 |
| Word 6 | 4 | 5 |
| Word 7 | 6 | 7 |

DRAM chip have 8-bit data bus

The transformation, a 8x8 matrix transposition, is done by the library inside a 64-byte cache line, thus very efficiently.

Copyright UPMEM® 2019
Microbenchmark: CPU-DPU

- CPU-DPU (serial/parallel/broadcast) and DPU-CPU (serial/parallel)
DPU Kernel Launch

• `dpu_launch()` launches a kernel on a `dpu_set`
  - `DPU_SYNCHRONOUS` suspends the application until the kernel finishes
  - `DPUASYNCHRONOUS` returns the control to the application
    • `dpu_sync` or `dpu_status` to check kernel completion

```c
1 printf("Run program on DPU(s) \n");
2 // Run DPU kernel
3 DPU_ASSERT(dpu_launch(dpu_set, DPU_SYNCHRONOUS));
```

What does the asynchronous execution enable?

Some ideas:
• **Task-level parallelism**: concurrent execution of different kernels on different DPU sets
• Concurrent **heterogeneous computation** on CPU and DPUs
How to Pass Parameters to the Kernel?

• We can use serial and parallel transfers
• We pass them directly to the scratchpad memory of the DPU
  - Working RAM (WRAM): 64KB per DPU
• This is useful for input parameters and some results

```c
// In DPU WRAM (dpu/task.c)
__host dpu_arguments_t DPU_INPUT_ARGUMENTS;
__host dpu_results_t DPU_RESULTS[NR_TASKLETS];
```

```c
// Host code (host/app.c)
#ifdef SERIAL
  DPU_FOREACH (dpu_set, dpu) {
    DPU_ASSERT(dpu_copy_to(dpu, "DPU_INPUT_ARGUMENTS", 0, (const void *)input_arguments[i], sizeof(input_arguments[0])));
    i++;
  }
#else
  DPU_FOREACH(dpu_set, dpu, i) {
    DPU_ASSERT(dpu_prepare_xfer(dpu, &input_arguments[i]));
  }
  DPU_ASSERT(dpu_push_xfer(dpu_set, DPU_XFER_TO_DPU, "DPU_INPUT_ARGUMENTS", 0, sizeof(input_arguments[0]), DPU_XFER_DEFAULT));
#endif
```
Recall: Vector Addition (VA)

• Our first programming example
• We partition the input arrays across:
  - DPUs
  - Tasklets, i.e., software threads running on a DPU
Programming a DPU Kernel (I)

• Vector addition

```c
// Vector addition kernel
int main_kernel1() {
    unsigned int tasklet_id = me();
    uint32_t input_size_dpu_bytes = DPU_INPUT_ARGUMENTS.size;
    uint32_t input_size_dpu_bytes_transfer = DPU_INPUT_ARGUMENTS.transfer_size;

    // Address of the current processing block in MRAM
    uint32_t base_tasklet = tasklet_id << BLOCK_SIZE LOG2;
    uint32_t mram_base_addr_A = (uint32_t)DPU_MRAM_HEAP_POINTER;
    uint32_t mram_base_addr_B = (uint32_t)(DPU_MRAM_HEAP_POINTER + input_size_dpu_bytes_transfer);

    // Initialize a local cache to store the MRAM block
    T *cache_A = (T *) mem_alloc(BLOCK_SIZE);
    T *cache_B = (T *) mem_alloc(BLOCK_SIZE);

    for(unsigned int byte_index = base_tasklet; byte_index < input_size_dpu_bytes; byte_index += BLOCK_SIZE * NR_TASKLETS){
        // Bound checking
        uint32_t l_size_bytes = (byte_index + BLOCK_SIZE >= input_size_dpu_bytes) ? (input_size_dpu_bytes - byte_index) : BLOCK_SIZE;

        // Load cache with current MRAM block
        mram_read((__mram_ptr void const*) (mram_base_addr_A + byte_index), cache_A, l_size_bytes);
        mram_read((__mram_ptr void const*) (mram_base_addr_B + byte_index), cache_B, l_size_bytes);

    // Computer vector addition
    vector_addition(cache_B, cache_A, l_size_bytes >> DIV);

    // Write cache to current MRAM block
    mram_write(cache_B, (__mram_ptr void*)(mram_base_addr_B + byte_index), l_size_bytes);
}
return 0;
```
Programming a DPU Kernel (II)

- Vector addition

```c
// vector_addition: Computes the vector addition of a cached block
static void vector_addition(T *bufferB, T *bufferA, unsigned int l_size) {
    for (unsigned int i = 0; i < l_size; i++) {
        bufferB[i] += bufferA[i];
    }
}
```
Intra-DPU Synchronization
Synchronization Primitives

- A **tasklet** is the software abstraction of a hardware thread
- Each tasklet can have its **own memory space in WRAM**
  - Tasklets can also share data in WRAM by sharing pointers
- Tasklets within the same DPU can **synchronize**
  - **Mutual exclusion**
    - `mutex_lock();` `mutex_unlock();`
  - **Handshakes**
    - `handshake_wait_for();` `handshake_notify();`
  - **Barriers**
    - `barrier_wait();`
  - **Semaphores**
    - `sem_give();` `sem_take();`
Parallel Reduction (I)

• Tasklets in a DPU can work together on a parallel reduction
Parallel Reduction (II)

- Each tasklet computes a local sum
Parallel Reduction (III)

- Each tasklet computes a local sum

```c
for(unsigned int byte_index = base_tasklet; byte_index < input_size_dpu_bytes; byte_index += BLOCK_SIZE * NR_TASKLETS){
    // Bound checking
    uint32_t l_size_bytes = (byte_index + BLOCK_SIZE >= input_size_dpu_bytes) ? (input_size_dpu_bytes - byte_index) : BLOCK_SIZE;

    // Load cache with current MRAM block
    mram_read((__mram_ptr void const*)(mram_base_addr_A + byte_index), cache_A, l_size_bytes);

    // Reduction in each tasklet
    l_count += reduction(cache_A, l_size_bytes >> DIV); // Accumulate in a local sum

    // Copy local count to shared array in WRAM
    message[tasklet_id] = l_count; // Copy local sum into WRAM
}```
Final Reduction

• A single tasklet can perform the final reduction

```
for(unsigned int byte_index = base_tasklet; byte_index < input_size_dpu_bytes; byte_index += BLOCK_SIZE * NR_TASKLETS){
    // Bound checking
    uint32_t l_size_bytes = (byte_index + BLOCK_SIZE >= input_size_dpu_bytes) ? (input_size_dpu_bytes - byte_index) : BLOCK_SIZE;

    // Load cache with current MRAM block
    mram_read((__mram_ptr void const*)(mram_base_addr_A + byte_index), cache_A, l_size_bytes);

    // Reduction in each tasklet
    l_count += reduction(cache_A, l_size_bytes >> DIV);  // Accumulate in a local sum

    // Copy local count to shared array in WRAM
    message[tasklet_id] = l_count;  // Copy local sum into WRAM
}

// Single-thread reduction
// Barrier
barrier_wait(&my_barrier);  // Barrier synchronization

if(tasklet_id == 0){
    #pragma unroll
    for (unsigned int each_tasklet = 1; each_tasklet < NR_TASKLETS; each_tasklet++){
        message[0] += message[each_tasklet];  // Sequential accumulation
    }

    // Total count in this DPU
    result->t_count = message[0];
}
```
Vector Reduction: Naïve Mapping

Thread 0  Thread 2  Thread 4  Thread 6  Thread 8  Thread 10

0   1   2   3   4   5   6   7   8   9   10  11

0+1  2+3  4+5  6+7  8+9  10+11

0...3  4..7  8..11

0..7  8..15

iterations

Slide credit: Hwu & Kirk
Using Barriers: Tree-Based Reduction

• Multiple tasklets can perform a tree-based reduction
  - After every iteration tasklets synchronize with a barrier
  - Half of the tasklets retire at the end of an iteration

```c
// Barrier
barrier_wait(&my_barrier);

#pragma unroll
for (unsigned int offset = 1; offset < NR_TASKLETS; offset <<= 1){
    if((tasklet_id & (2*offset - 1)) == 0){
        message[tasklet_id] += message[tasklet_id + offset];
    }

    // Barrier
    barrier_wait(&my_barrier);
}
```

A handshake-based tree-based reduction is also possible. We can compare single-tasklet, barrier-based, and handshake-based versions*.

*Gómez-Luna et al., “Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture,”
Parallel Reduction on GPU

Search Space of Parallel Reduction

Over 85 different versions possible!


HetSys Course: Lecture 6: Parallel Patterns: Reduction (Spring 2022)

Project & Seminar, ETH Zürich, Spring 2022
Hands-on Acceleration on Heterogeneous Computing Systems (https://safari.ethz.ch/projects_and_a_...)

https://youtu.be/Xp0HHpcDwUc
User Manual

Getting started

- The UPMEM DPU toolchain
  - Notes before starting
  - The toolchain purpose
  - dpu-upmem-dpurte-clang
    - Limitations
  - The DPU Runtime Library
  - The Host Library
  - dpu-lldb
- Installing the UPMEM DPU toolchain
  - Dependencies
    - Python
  - Installation packages
    - Installation from tar.gz binary archive
  - Functional simulator
- Hello World! Example
  - Purpose
  - Writing and building the program
Microbenchmarking of UPMEM PIM
DPU Pipeline

- In-order pipeline
  - Up to 425 MHz
- Fine-grain multithreaded
  - 24 hardware threads
- 14 pipeline stages
  - DISPATCH: Thread selection
  - FETCH: Instruction fetch
  - READOP: Register file
  - FORMAT: Operand formatting
  - ALU: Operation and WRAM
  - MERGE: Result formatting
Arithmetic Throughput: Microbenchmark

• Goal
  - Measure the maximum arithmetic throughput for different datatypes and operations

• Microbenchmark
  - We stream over an array in WRAM and perform read-modify-write operations
  - Experiments on one DPU
  - We vary the number of tasklets from 1 to 24
  - Arithmetic operations: add, subtract, multiply, divide
  - Datatypes: int32, int64, float, double

• We measure cycles with an accurate cycle counter that the SDK provides
  - We include WRAM accesses (including address calculation) and arithmetic operation
Microbenchmark for INT32 ADD Throughput

C-based code

```c
#define SIZE 256
int* bufferA = mem_alloc(SIZE * sizeof(int));
for(int i = 0; i < SIZE; i++){
    int temp = bufferA[i];
    temp += scalar;
    bufferA[i] = temp;
}
```

Compiled code (UPMEM DPU ISA)

```assembly
move r2, 0
.LBB0_1:  // Loop header
    lsl_add r3, r0, r2, 2  // Address calculation
    lw r4, r3, 0         // Load from WRAM
    add r4, r4, r1       // Add
    sw r3, 0, r4         // Store to WRAM
    add r2, r2, 1        // Index update
    jneq r2, 256, .LBB0_1 // Conditional jump
```
**Arithmetic Throughput: 11 Tasklets**

**KEY OBSERVATION 1**

The arithmetic throughput of a DRAM Processing Unit saturates at 11 or more tasklets. This observation is consistent for different datatypes (INT32, INT64, UINT32, UINT64, FLOAT, DOUBLE) and operations (ADD, SUB, MUL, DIV).
Arithmetic Throughput: ADD/SUB

Can we explain the peak throughput?

Peak throughput at 11 tasklets. One instruction retires every cycle when the pipeline is full.

Arithmetic Throughput (in OPS) = \( \frac{\text{frequency}_{\text{DPU}}}{\#\text{instructions}} \)

INT32 ADD/SUB are 17% faster than INT64 ADD/SUB.
Arithmetic Throughput: #Instructions

• Compiler explorer: [https://dpu.dev](https://dpu.dev)

```c
#define BLOCK_SIZE 1024

typedef int T;
void Benchmark__32bits(T *cache_A, T scalar) {
    for (int i = 0; i < BLOCK_SIZE / sizeof(T); i++){
        ///// WRAM READ /////
        T temp = cache_A[i];

        temp += scalar; // ADD

        ///// WRAM WRITE /////
        cache_A[i] = temp;
    }
}

typedef long T_long;
void Benchmark__64bits(T_long *cache_A, T_long scalar) {
    for (int i = 0; i < BLOCK_SIZE / sizeof(T_long); i++){
        ///// WRAM READ /////
        T_long temp = cache_A[i];

        temp += scalar; // ADD
```

6 instructions in the 32-bit ADD/SUB microbenchmark
7 instructions in the 64-bit ADD/SUB microbenchmark
Arithmetic Throughput: ADD/SUB

INT32 ADD/SUB are 17% faster than INT64 ADD/SUB

Can we explain the peak throughput?

Peak throughput at 11 tasklets.
One instruction retires every cycle when the pipeline is full

Arithmetic Throughput (in OPS) = \( \frac{frequency_{DPU}}{\#instructions} \)

64-bit ADD/SUB: 7 instructions → 50.00 MOPS
at \( frequency_{DPU} = 350 \text{ MHz} \)
Arithmetic Throughput: MUL/DIV

Huge throughput difference between ADD/SUB and MUL/DIV

DPUs do not have a 32-bit multiplier

MUL/DIV implementation is based on an instruction that performs bit shifting and addition in 1 cycle (MUL/DIV take a maximum of 32 instructions)
Arithmetic Throughput: Native Support

KEY OBSERVATION 2

• DPUs provide native hardware support for 32- and 64-bit integer addition and subtraction, leading to high throughput for these operations.

• DPUs do not natively support 32- and 64-bit multiplication and division, and floating point operations. These operations are emulated by the UPMEM runtime library, leading to much lower throughput.
Microbenchmark: Arithmetic Throughput

- Arithmetic throughput for different operations and datatypes
DPU: WRAM Bandwidth

PIM Chip

Control/Status Interface

DDR4 Interface

Dispatch
Fetch1
Fetch2
Fetch3
Readop1
Readop2
Readop3
Format
Alu1
Alu2
Alu3
Alu4
Merge1
Merge2

Pipeline

Register File

64-KB WRAM

24-KB IRAM

DMA Engine

64-MB DRAM Bank (MRAM)

64 bits

x8

SAFARI
WRAM Bandwidth: Microbenchmark

• Goal
  - Measure the WRAM bandwidth for the STREAM benchmark

• Microbenchmark
  - We implement the four versions of STREAM: COPY, ADD, SCALE, and TRIAD
  - The operations performed in ADD, SCALE, and TRIAD are addition, multiplication, and addition+multiplication, respectively
  - We vary the number of tasklets from 1 to 16
  - We show results for 1 DPU

• We do not include accesses to MRAM
// COPY
for(int i = 0; i < SIZE; i++){
    bufferB[i] = bufferA[i];
}

// ADD
for(int i = 0; i < SIZE; i++){
    bufferC[i] = bufferA[i] + bufferB[i];
}

// SCALE
for(int i = 0; i < SIZE; i++){
    bufferB[i] = scalar * bufferA[i];
}

// TRIAD
for(int i = 0; i < SIZE; i++){
    bufferC[i] = bufferA[i] + scalar * bufferB[i];
}
**WRAM Bandwidth: STREAM**

How can we estimate the bandwidth?

Assuming that the pipeline is full, and *Bytes* is the number of bytes read and written:

\[
\text{WRAM Bandwidth} \left( \text{in} \frac{\text{Bytes}}{\text{S}} \right) = \frac{\text{Bytes} \times \text{frequency}_\text{DPU}}{\#\text{instructions}}
\]
WRAM Bandwidth: COPY

**COPY** executes 2 instructions (WRAM load and store).

With 11 tasklets, $11 \times 16$ bytes in 22 cycles:

\[
\text{WRAM Bandwidth} = \left( \frac{B}{S} \right) = 2,800 \frac{MB}{s} \text{ at 350 MHz}
\]
**WRAM Bandwidth:** ADD

**Formula:**

\[ \text{WRAM Bandwidth} \left( \frac{B}{S} \right) = \frac{\text{Bytes} \times \text{frequency}_{DPU}}{\text{#instructions}} \]

**ADD executes 5 instructions** (2 ld, add, addc, sd).

**With 11 tasklets,** 11 × 24 bytes in 55 cycles:

\[ \text{WRAM Bandwidth} \left( \frac{B}{S} \right) = 1,680 \frac{MB}{s} \text{ at } 350 \text{ MHz} \]
WRAM Bandwidth: Access Patterns

- All 8-byte WRAM loads and stores take one cycle when the DPU pipeline is full.

**KEY OBSERVATION 3**

The sustained bandwidth provided by the DPU’s internal Working memory (WRAM) is independent of the memory access pattern (either streaming, strided, or random access pattern).

**All 8-byte WRAM loads and stores take one cycle,** when the DPU’s pipeline is full (i.e., with 11 or more tasklets).

- Microbenchmark: $c[a[i]] = b[a[i]]$;
  - Unit-stride: $a[i] = a[i-1] + 1$;
  - Strided: $a[i] = a[i-1] + \text{stride}$;
  - Random: $a[i] = \text{rand}()$;
Microbenchmark: STREAM and WRAM

- STREAM benchmark and WRAM access patterns
DPU: MRAM Latency and Bandwidth

**PIM Chip**

- Control/Status Interface
- DDR4 Interface
- 24-KB IRAM
- 64-KB WRAM
- DMA Engine
- 64-MB DRAM Bank (MRAM)
- 64 bits
- x8

**Pipeline**

- DISPATCH
- FETCH1
- FETCH2
- FETCH3
- READOP1
- READOP2
- READOP3
- FORMAT
- ALU1
- ALU2
- ALU3
- ALU4
- MERGE1
- MERGE2

**Register File**

- 64 bits
MRAM Bandwidth

• Goal
  - Measure MRAM bandwidth for different access patterns

• Microbenchmarks
  - Latency of a single DMA transfer for different transfer sizes
    • mram_read(); // MRAM–WRAM DMA transfer
    • mram_write(); // WRAM–MRAM DMA transfer
  - STREAM benchmark
    • COPY, COPY-DMA
    • ADD, SCALE, TRIAD
  - Strided access pattern
    • Coarse-grain strided access
    • Fine-grain strided access
  - Random access pattern (GUPS)

• We do include accesses to MRAM
MRAM Read and Write Latency (I)

We can model the MRAM latency with a linear expression:

$$\text{MRAM Latency (in cycles)} = \alpha + \beta \times \text{size}$$

In our measurements, $\beta$ equals 0.5 cycles/byte.

Theoretical maximum MRAM bandwidth = 700 MB/s at 350 MHz.
**KEY OBSERVATION 4**

- The DPU’s **Main memory (MRAM) bank access latency** increases **linearly** with the transfer size.
- The maximum theoretical MRAM **bandwidth** is **2 bytes per cycle**.
Read and write access to MRAM are symmetric.

The sustained MRAM bandwidth increases with data transfer size.

**Programming Recommendation 1**

For data movement between the DPU's MRAM bank and the WRAM, **use large DMA transfer sizes when all the accessed data is going to be used.**
MRAM latency changes slowly between 8 and 128 bytes.

For small transfers, the fixed cost (\(\alpha\)) dominates the variable cost (\(\beta \times \text{size}\)).

**PROGRAMMING RECOMMENDATION 2**

For small transfers between the MRAM bank and the WRAM, **fetch more bytes than necessary within a 128-byte limit**. Doing so increases the likelihood of finding data in WRAM for later accesses (i.e., the program can check whether the desired data is in WRAM before issuing a new MRAM access).
2,048-byte transfers are only 4% faster than 1,024-byte transfers

Larger transfers require more WRAM, which may limit the number of tasklets

**PROGRAMMING RECOMMENDATION 3**

Choose the data transfer size between the MRAM bank and the WRAM based on the program’s WRAM usage, as it imposes a tradeoff between the sustained MRAM bandwidth and the number of tasklets that can run in the DPU (which is dictated by the limited WRAM capacity).
MRAM Bandwidth

• **Goal**
  - Measure MRAM bandwidth for different access patterns

• **Microbenchmarks**
  - Latency of a single DMA transfer for different transfer sizes
    - mram_read(); // MRAM–WRAM DMA transfer
    - mram_write(); // WRAM–MRAM DMA transfer
  - **STREAM** benchmark
    - COPY, COPY-DMA
    - ADD, SCALE, TRIAD
  - Strided access pattern
    - Coarse-grain strided access
    - Fine-grain strided access
  - Random access pattern (GUPS)

• We do include accesses to MRAM
STREAM Benchmark in MRAM

// COPY
// Load current MRAM block to WRAM
mram_read((__mram_ptr void const*)mram_address_A, bufferA, SIZE * sizeof(uint64_t));

for(int i = 0; i < SIZE; i++){
    bufferB[i] = bufferA[i];
}

// Write WRAM block to MRAM
mram_write(bufferB, (__mram_ptr void*)mram_address_B, SIZE * sizeof(uint64_t));

// COPY-DMA
// Load current MRAM block to WRAM
mram_read((__mram_ptr void const*)mram_address_A, bufferA, SIZE * sizeof(uint64_t));

// Write WRAM block to MRAM
mram_write(bufferA, (__mram_ptr void*)mram_address_B, SIZE * sizeof(uint64_t));
The sustained bandwidth of **COPY-DMA** is close to the theoretical maximum (700 MB/s): ~1.6 TB/s for 2,556 DPUs.

**COPY-DMA** saturates with **two tasklets**, even though the DMA engine can perform only one transfer at a time.

Using **two or more tasklets** guarantees that there is always a DMA request enqueued to keep the DMA engine busy.
STREAM Benchmark: Bandwidth Saturation (I)

**STREAM (MRAM, INT64, 1DPU)**

**COPY-DMA**

**COPY**

**ADD**

**SCALE**

**TRIAD**

**COPY and ADD** saturate at 4 and 6 tasklets, respectively.

**SCALE and TRIAD** saturate at 11 tasklets.

The latency of MRAM accesses becomes longer than the pipeline latency after 4 and 6 tasklets for **COPY** and **ADD**, respectively.

The pipeline latency of **SCALE** and **TRIAD** is longer than the MRAM latency for any number of tasklets (both use costly MUL).
KEY OBSERVATION 5

• When the access latency to an MRAM bank for a streaming benchmark (COPY-DMA, COPY, ADD) is larger than the pipeline latency (i.e., execution latency of arithmetic operations and WRAM accesses), the performance of the DPU saturates at a number of tasklets smaller than 11. This is a memory-bound workload.

• When the pipeline latency for a streaming benchmark (SCALE, TRIAD) is larger than the MRAM access latency, the performance of a DPU saturates at 11 tasklets. This is a compute-bound workload.
MRAM Bandwidth

• Goal
  - Measure MRAM bandwidth for different access patterns

• Microbenchmarks
  - Latency of a single DMA transfer for different transfer sizes
     • mram_read(); // MRAM-WRAM DMA transfer
     • mram_write(); // WRAM-MRAM DMA transfer
  - STREAM benchmark
     • COPY, COPY-DMA
     • ADD, SCALE, TRIAD
  - Strided access pattern
     • Coarse-grain strided access
     • Fine-grain strided access
  - Random access pattern (GUPS)

• We do include accesses to MRAM

SAFARI
Strided and Random Access to MRAM

// COARSE-GRAINED STRIDED ACCESS
// Load current MRAM block to WRAM
mram_read((__mram_ptr void const*)mram_address_A, bufferA, SIZE * sizeof(uint64_t));
mram_read((__mram_ptr void const*)mram_address_B, bufferB, SIZE * sizeof(uint64_t));

for(int i = 0; i < SIZE; i += stride){
    bufferB[i] = bufferA[i];
}

// Write WRAM block to MRAM
mram_write(bufferB, (__mram_ptr void*)mram_address_B, SIZE * sizeof(uint64_t));

// FINE-GRAINED STRIDED & RANDOM ACCESS
for(int i = 0; i < SIZE; i += stride){
    int index = i * sizeof(uint64_t);
    // Load current MRAM element to WRAM
    mram_read((__mram_ptr void const*)(mram_address_A + index), bufferA, sizeof(uint64_t));

    // Write WRAM element to MRAM
    mram_write(bufferA, (__mram_ptr void*)(mram_address_B + index), sizeof(uint64_t));
}

Large difference in maximum sustained bandwidth between coarse-grained and fine-grained DMA

Coarse-grained DMA uses 1,024-byte transfers, while fine-grained DMA uses 8-byte transfers

Random access achieves very similar maximum sustained bandwidth to fine-grained strided approach
Strided and Random Accesses (II)

The sustained MRAM bandwidth of coarse-grained DMA decreases as the stride increases.

The effective utilization of the transferred data decreases as the stride becomes larger (e.g., a stride 4 means that only one fourth of the transferred data is used).
For a stride of 16 or larger, the fine-grained DMA approach achieves higher bandwidth.

With stride 16, only one sixteenth of the maximum sustained bandwidth (622.36 MB/s) of coarse-grained DMA is effectively used, which is lower than the bandwidth of fine-grained DMA (72.58 MB/s).
**PROGRAMMING RECOMMENDATION 4**

- For strided access patterns with a **stride smaller than 16 8-byte elements**, fetch a **large contiguous chunk** (e.g., 1,024 bytes) from a DPU’s MRAM bank.
- For strided access patterns with **larger strides and random access patterns**, fetch **only the data elements that are needed** from an MRAM bank.
Microbenchmark: Strided and Random

- Strided and random accesses to MRAM
DPU: Arithmetic Throughput vs. Operational Intensity

**PIM Chip**

Control/Status Interface

DDR4 Interface

DISPATCH

FETCH1

FETCH2

FETCH3

READOP1

READOP2

READOP3

FORMAT

ALU1

ALU2

ALU3

ALU4

MERGE1

MERGE2

24-KB IRAM

DMA Engine

64-KB WRAM

64-MB DRAM Bank (MRAM)

64 bits

x8
Arithmetic Throughput vs. Operational Intensity (I)

- Goal
  - Characterize memory-bound regions and compute-bound regions for different datatypes and operations

- Microbenchmark
  - We load one chunk of an MRAM array into WRAM
  - Perform a variable number of operations on the data
  - Write back to MRAM

- The experiment is inspired by the Roofline model*

- We define operational intensity (OI) as the number of arithmetic operations performed per byte accessed from MRAM (OP/B)

- The pipeline latency changes with the operational intensity, but the MRAM access latency is fixed

Arithmetic Throughput vs. Operational Intensity (II)

```c
int repetitions = input_repeat >= 1.0 ? (int)input_repeat : 1;
int stride = input_repeat >= 1.0 ? 1 : (int)(1 / input_repeat);

// Load current MRAM block to WRAM
mram_read((__mram_ptr void const*)mram_address_A, bufferA, SIZE * sizeof(T));

// Update
for(int r = 0; r < repetitions; r++) {
    for(int i = 0; i < SIZE; i += stride) {
        #ifdef ADD
            bufferA[i] += scalar; // ADD
        #elif SUB
            bufferA[i] -= scalar; // SUB
        #elif MUL
            bufferA[i] *= scalar; // MUL
        #elif DIV
            bufferA[i] /= scalar; // DIV
        #endif
    }
}

// Write WRAM block to MRAM
mram_write(bufferA, (__mram_ptr void*)mram_address_B, SIZE * sizeof(T));
```

input_repeat greater or equal to 1 indicates the (integer) number of repetitions per input element

input_repeat smaller than 1 indicates the fraction of elements that are updated
We show results of arithmetic throughput vs. operational intensity for (a) 32-bit integer ADD, (b) 32-bit integer MUL, (c) 32-bit floating-point ADD, and (d) 32-bit floating-point MUL (results for other datatypes and operations show similar trends).
In the **memory-bound region**, the arithmetic throughput increases with the operational intensity.

In the **compute-bound region**, the arithmetic throughput is flat at its maximum.

The **throughput saturation point** is the operational intensity where the transition between the memory-bound region and the compute-bound region happens.

The throughput saturation point is as low as \( \frac{1}{4} \) OP/B, i.e., 1 integer addition per every 32-bit element fetched.
Arithmetic Throughput vs. Operational Intensity (V)

**KEY OBSERVATION 6**

The arithmetic throughput of a DRAM Processing Unit (DPU) saturates at low or very low operational intensity (e.g., 1 integer addition per 32-bit element). Thus, the DPU is fundamentally a compute-bound processor. We expect most real-world workloads be compute-bound in the UPMEM PIM architecture.
Microbenchmark: Arithmetic Throughput vs. Operational Intensity

- Arithmetic Throughput versus Operational Intensity
PrIM Benchmarks

• Goal
  - A common set of workloads that can be used to
    • evaluate the UPMEM PIM architecture,
    • compare software improvements and compilers,
    • compare future PIM architectures and hardware

• Two key selection criteria:
  - Selected workloads from different application domains
  - Memory-bound workloads on processor-centric architectures

• 14 different workloads, 16 different benchmarks*

*There are two versions for two of the workloads (HST, SCAN).
## PrIM Benchmarks: Application Domains

<table>
<thead>
<tr>
<th>Domain</th>
<th>Benchmark</th>
<th>Short name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense linear algebra</td>
<td>Vector Addition</td>
<td>VA</td>
</tr>
<tr>
<td></td>
<td>Matrix-Vector Multiply</td>
<td>GEMV</td>
</tr>
<tr>
<td>Sparse linear algebra</td>
<td>Sparse Matrix-Vector Multiply</td>
<td>SpMV</td>
</tr>
<tr>
<td>Databases</td>
<td>Select</td>
<td>SEL</td>
</tr>
<tr>
<td></td>
<td>Unique</td>
<td>UNI</td>
</tr>
<tr>
<td>Data analytics</td>
<td>Binary Search</td>
<td>BS</td>
</tr>
<tr>
<td></td>
<td>Time Series Analysis</td>
<td>TS</td>
</tr>
<tr>
<td>Graph processing</td>
<td>Breadth-First Search</td>
<td>BFS</td>
</tr>
<tr>
<td>Neural networks</td>
<td>Multilayer Perceptron</td>
<td>MLP</td>
</tr>
<tr>
<td>Bioinformatics</td>
<td>Needleman-Wunsch</td>
<td>NW</td>
</tr>
<tr>
<td>Image processing</td>
<td>Image histogram (short)</td>
<td>HST-S</td>
</tr>
<tr>
<td></td>
<td>Image histogram (large)</td>
<td>HST-L</td>
</tr>
<tr>
<td>Parallel primitives</td>
<td>Reduction</td>
<td>RED</td>
</tr>
<tr>
<td></td>
<td>Prefix sum (scan-scan-add)</td>
<td>SCAN-SSA</td>
</tr>
<tr>
<td></td>
<td>Prefix sum (reduce-scan-scan)</td>
<td>SCAN-RSS</td>
</tr>
<tr>
<td></td>
<td>Matrix transposition</td>
<td>TRNS</td>
</tr>
</tbody>
</table>
Roofline Model

- Intel Advisor on an Intel Xeon E3-1225 v6 CPU

All workloads fall in the memory-bound area of the Roofline
## PrIM Benchmarks: Diversity

- PrIM benchmarks are diverse:
  - Memory access patterns
  - Operations and datatypes
  - Communication/synchronization

<table>
<thead>
<tr>
<th>Domain</th>
<th>Benchmark</th>
<th>Short name</th>
<th>Memory access pattern</th>
<th>Computation pattern</th>
<th>Communication/synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Sequential</td>
<td>Strided</td>
<td>Random</td>
</tr>
<tr>
<td>Dense linear algebra</td>
<td>Vector Addition</td>
<td>VA</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Matrix-Vector Multiply</td>
<td>GEMV</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sparse linear algebra</td>
<td>Sparse Matrix-Vector Multiply</td>
<td>SpMV</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Databases</td>
<td>Select</td>
<td>SEL</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unique</td>
<td>UNI</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data analytics</td>
<td>Binary Search</td>
<td>BS</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Time Series Analysis</td>
<td>TS</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graph processing</td>
<td>Breadth-First Search</td>
<td>BFS</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Neural networks</td>
<td>Multilayer Perceptron</td>
<td>MLP</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bioinformatics</td>
<td>Needleman-Wunsch</td>
<td>NW</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Image processing</td>
<td>Image histogram (short)</td>
<td>HST-S</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Image histogram (long)</td>
<td>HST-L</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Parallel primitives</td>
<td>Reduction</td>
<td>RED</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Prefix sum (scan-scan-add)</td>
<td>SCAN-SSA</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Prefix sum (reduce-scan-scan)</td>
<td>SCAN-RSS</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Matrix transposition</td>
<td>TRNS</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
### PrIM Benchmarks: Inter-DPU Communication

#### Inter-DPU communication

- **Result merging:**
  - SEL, UNI, HST-S, HST-L, RED
  - Only DPU-CPU transfers

- **Redistribution of intermediate results:**
  - BFS, MLP, NW, SCAN-SSA, SCAN-RSS
  - DPU-CPU and CPU-DPU transfers

<table>
<thead>
<tr>
<th>Domain</th>
<th>Benchmark</th>
<th>Short name</th>
<th>Memory access pattern</th>
<th>Computation pattern</th>
<th>Communication/synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Sequential</td>
<td>Strided</td>
<td>Random</td>
</tr>
<tr>
<td>Dense linear algebra</td>
<td>Vector Addition</td>
<td>VA</td>
<td>Yes</td>
<td></td>
<td>add, int32_t</td>
</tr>
<tr>
<td></td>
<td>Matrix-Vector Multiply</td>
<td>GEMV</td>
<td>Yes</td>
<td></td>
<td>add, mul, uint32_t</td>
</tr>
<tr>
<td>Sparse linear algebra</td>
<td>Sparse Matrix-Vector Multiply</td>
<td>SpMV</td>
<td>Yes</td>
<td>Yes</td>
<td>add, mul, float</td>
</tr>
<tr>
<td>Databases</td>
<td>Select</td>
<td>SEL</td>
<td>Yes</td>
<td></td>
<td>add, compare, int64_t</td>
</tr>
<tr>
<td></td>
<td>Unique</td>
<td>UNI</td>
<td>Yes</td>
<td></td>
<td>add, compare, int64_t</td>
</tr>
<tr>
<td>Data analytics</td>
<td>Binary Search</td>
<td>BS</td>
<td>Yes</td>
<td>Yes</td>
<td>compare, int64_t</td>
</tr>
<tr>
<td></td>
<td>Time Series, Analysis</td>
<td>TS</td>
<td>Yes</td>
<td></td>
<td>add, sub, mul, div, int32_t</td>
</tr>
<tr>
<td>Graph processing</td>
<td>Breadth-First Search</td>
<td>BFS</td>
<td>Yes</td>
<td>Yes</td>
<td>bitwise logic, uint64_t</td>
</tr>
<tr>
<td>Neural networks</td>
<td>Multilayer Perceptron</td>
<td>MLP</td>
<td>Yes</td>
<td></td>
<td>add, mul, compare, int32_t</td>
</tr>
<tr>
<td>Bioinformatics</td>
<td>Needle, by Woman</td>
<td>NW</td>
<td>Yes</td>
<td>Yes</td>
<td>add, sub, compare, int32_t</td>
</tr>
<tr>
<td>Image processing</td>
<td>Image histogram (short)</td>
<td>HST-S</td>
<td>Yes</td>
<td>Yes</td>
<td>add, uint32_t</td>
</tr>
<tr>
<td></td>
<td>Image histogram (long)</td>
<td>HST-L</td>
<td>Yes</td>
<td></td>
<td>add, uint32_t</td>
</tr>
<tr>
<td>Parallel primitives</td>
<td>Reduction</td>
<td>RED</td>
<td>Yes</td>
<td>Yes</td>
<td>add, int64_t</td>
</tr>
<tr>
<td></td>
<td>Prefix sum (scan, data, add)</td>
<td>SCAN-AX</td>
<td>Yes</td>
<td></td>
<td>add, int64_t</td>
</tr>
<tr>
<td></td>
<td>Prefix sum (reduce, scan, scan)</td>
<td>SCAN-RSS</td>
<td>Yes</td>
<td></td>
<td>add, int64_t</td>
</tr>
<tr>
<td></td>
<td>Mass transportation</td>
<td>SCAN</td>
<td>Yes</td>
<td></td>
<td>add, sub, mul, int64_t</td>
</tr>
</tbody>
</table>
PrIM Benchmarks

- 16 benchmarks and scripts for evaluation
- [https://github.com/CMU-SAFARI/prim-benchmarks](https://github.com/CMU-SAFARI/prim-benchmarks)
Outline

• Introduction
  - Accelerator Model
  - UPMEM-based PIM System Overview

• UPMEM PIM Programming
  - Vector Addition
  - CPU-DPU Data Transfers
  - Inter-DPU Communication
  - CPU-DPU/DPU-CPU Transfer Bandwidth

• DRAM Processing Unit
  - Arithmetic Throughput
  - WRAM and MRAM Bandwidth

• PrIM Benchmarks
  - Roofline Model
  - Benchmark Diversity

• Evaluation
  - Strong and Weak Scaling
  - Comparison to CPU and GPU

• Key Takeaways
Evaluation Methodology

• We evaluate the 16 PrIM benchmarks on two UPMEM-based systems:
  - 2,556-DPU system
  - 640-DPU system

• Strong and weak scaling experiments on the 2,556-DPU system
  - 1 DPU with different numbers of tasklets
  - 1 rank (strong and weak)
  - Up to 32 ranks

*Strong scaling* refers to how the execution time of a program solving a particular problem varies with the number of processors for a fixed problem size.

*Weak scaling* refers to how the execution time of a program solving a particular problem varies with the number of processors for a fixed problem size per processor.
Evaluation Methodology

• We evaluate the 16 PrIM benchmarks on two UPMEM-based systems:
  - 2,556-DPU system
  - 640-DPU system
• Strong and weak scaling experiments on the 2,556-DPU system
  - 1 DPU with different numbers of tasklets
  - 1 rank (strong and weak)
  - Up to 32 ranks
• Comparison of both UPMEM-based PIM systems to state-of-the-art CPU and GPU
  - Intel Xeon E3-1240 CPU
  - NVIDIA Titan V GPU
2,560-DPU System

- UPMEM-based PIM system with 20 UPMEM DIMMs of 16 chips each (40 ranks)
  - P21 DIMMs
  - Dual x86 socket
    - UPMEM DIMMs coexist with regular DDR4 DIMMs
    - 2 memory controllers/socket (3 channels each)
    - 2 conventional DDR4 DIMMs on one channel of one controller

* There are 4 faulty DPUs in the system that we use in our experiments. Thus, the maximum number of DPUs we can use is 2,556.
640-DPU System

• UPMEM-based PIM system with 10 UPMEM DIMMs of 8 chips each (10 ranks)
  - E19 DIMMs
  - x86 socket
    • 2 memory controllers (3 channels each)
    • 2 conventional DDR4 DIMMs on one channel of one controller
## Datasets

- Strong and weak scaling experiments

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Strong Scaling Dataset</th>
<th>Weak Scaling Dataset</th>
<th>MRAM-WRAM Transfer Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA</td>
<td>1 DPU-1 rank: 2.5M elem. (10 MB)</td>
<td>2.5M elem./DPU (10 MB)</td>
<td>1024 bytes</td>
</tr>
<tr>
<td>GEMV</td>
<td>1 DPU-1 rank: 8192 \times 1024 elem. (32 MB)</td>
<td>1024 \times 2048 elem./DPU (8 MB)</td>
<td>1024 bytes</td>
</tr>
<tr>
<td>SpMV</td>
<td>bcstk30 [253] (12 MB)</td>
<td>bcstk30 [253]</td>
<td>64 bytes</td>
</tr>
<tr>
<td>SEL</td>
<td>1 DPU-1 rank: 3.8M elem. (30 MB)</td>
<td>3.8M elem./DPU (30 MB)</td>
<td>1024 bytes</td>
</tr>
<tr>
<td>UNI</td>
<td>1 DPU-1 rank: 3.8M elem. (30 MB)</td>
<td>3.8M elem./DPU (30 MB)</td>
<td>1024 bytes</td>
</tr>
<tr>
<td>BS</td>
<td>2M elem. (16 MB). 1 DPU-1 rank: 256K queries. (2 MB)</td>
<td>2M elem. (16 MB). 256K queries/DPU (2 MB)</td>
<td>8 bytes</td>
</tr>
<tr>
<td>TS</td>
<td>256 elem. query. 1 DPU-1 rank: 512K elem. (2 MB)</td>
<td>512K elem./DPU (2 MB)</td>
<td>256 bytes</td>
</tr>
<tr>
<td>BFS</td>
<td>loc-gowalla [254] (22 MB)</td>
<td>rMat [255] (=100K vertices and 1.2M edges per DPU)</td>
<td>8 bytes</td>
</tr>
<tr>
<td>MLP</td>
<td>3 fully-connected layers. 1 DPU-1 rank: 2K neurons (32 MB)</td>
<td>3 fully-connected layers. 1K neuron/DPU (4 MB)</td>
<td>1024 bytes</td>
</tr>
<tr>
<td>NW</td>
<td>1 DPU-1 rank: 2560 bps (50 MB), large/small sub-block = ( \frac{2560}{# DPU} )</td>
<td>512 bps/DPU (2MB), 1/s = 512/2</td>
<td>8, 16, 32, 40 bytes</td>
</tr>
<tr>
<td>HST-S</td>
<td>1 DPU-1 rank: 1536 \times 1024 input image [256] (6 MB)</td>
<td>1536 \times 1024 input image [256]/DPU (6 MB)</td>
<td>1024 bytes</td>
</tr>
<tr>
<td>HST-L</td>
<td>1 DPU-1 rank: 1536 \times 1024 input image [256] (6 MB)</td>
<td>1536 \times 1024 input image [256]/DPU (6 MB)</td>
<td>1024 bytes</td>
</tr>
<tr>
<td>RED</td>
<td>1 DPU-1 rank: 6.3M elem. (50 MB)</td>
<td>6.3M elem./DPU (50 MB)</td>
<td>1024 bytes</td>
</tr>
<tr>
<td>SCAN-SSA</td>
<td>1 DPU-1 rank: 3.8M elem. (30 MB)</td>
<td>3.8M elem./DPU (30 MB)</td>
<td>1024 bytes</td>
</tr>
<tr>
<td>SCAN-RSS</td>
<td>1 DPU-1 rank: 3.8M elem. (30 MB)</td>
<td>3.8M elem./DPU (30 MB)</td>
<td>1024 bytes</td>
</tr>
<tr>
<td>TRNS</td>
<td>1 DPU-1 rank: 12288 \times 16 \times 64 \times 8 (768 MB)</td>
<td>12288 \times 16 \times 1 \times 8/DPU (12 MB)</td>
<td>128, 1024 bytes</td>
</tr>
</tbody>
</table>

The **PrIM benchmarks** repository includes all datasets and scripts used in our evaluation

[https://github.com/CMU-SAFAIR/prim-benchmarks](https://github.com/CMU-SAFAIR/prim-benchmarks)
Strong Scaling: 1 DPU (I)

- **Strong scaling experiments on 1 DPU**
  - We set the number of tasklets to 1, 2, 4, 8, and 16
  - We show the breakdown of execution time:
    - **DPU**: Execution time on the DPU
    - **Inter-DPU**: Time for inter-DPU communication via the host CPU
    - **CPU-DPU**: Time for CPU to DPU transfer of input data
    - **DPU-CPU**: Time for DPU to CPU transfer of final results
  - Speedup over 1 tasklet
Strong Scaling: 1 DPU (II)

A number of tasklets greater than 11 is a good choice for most real-world workloads we tested (16 kernels out of 19 kernels from 16 benchmarks), as it fully utilizes the DPU’s pipeline.

**KEY OBSERVATION 10**

Speedups 1.5-2.0x as we double the number of tasklets from 1 to 8. Speedups 1.2-1.5x from 8 to 16, since the pipeline throughput saturates at 11 tasklets.
Strong Scaling: 1 DPU (III)

VA, GEMV, SpMV, BS, TS, MLP, HST-S do not use intra-DPU synchronization primitives

In SEL, UNI, NW, RED, SCAN-SSA (Scan kernel), SCAN-RSS (both kernels), synchronization is lightweight

BFS, HST-L, TRNS (Step 3) use mutexes, which cause contention when accessing shared data structures
Strong Scaling: 1 DPU (IV)

- **VA, GEMV, SpMV, BS, TS, MLP, HST**: do not use intra-DPU synchronization primitives.
- In SEL, UNI, NW, RED, SCAN-SSA (Scan kernel), SCAN-RSS (both kernels), synchronization is lightweight.
- BFS, HST-L, TRNS (Step 3) use mutexes, which cause contention when accessing shared data structures.

**KEY OBSERVATION 11**

Intensive use of intra-DPU synchronization across tasklets (e.g., mutexes, barriers, handshakes) may limit scalability, sometimes causing the best performing number of tasklets to be lower than 11.
SCAN-SSA (Add kernel) is not compute-intensive. Thus, performance saturates with less than 11 tasklets (recall STREAM ADD). BS shows similar behavior.

**KEY OBSERVATION 12**

Most real-world workloads are in the compute-bound region of the DPU (all kernels except SCAN-SSA (Add kernel) and BS), i.e., the pipeline latency dominates the MRAM access latency.
**Strong Scaling: 1 DPU (VI)**

The amount of time spent on CPU-DPU and DPU-CPU transfers is low compared to the time spent on DPU execution.

TRNS performs step 1 of the matrix transposition via the CPU-DPU transfer.

Using small transfers (8 elements) does not exploit full CPU-DPU bandwidth.

**KEY OBSERVATION 13**

Transferring large data chunks from/to the host CPU is preferred for input data and output results due to higher sustained CPU-DPU/DPU-CPU bandwidths.
Strong Scaling: 1 Rank (I)

- Strong scaling experiments on 1 rank

  - We set the number of tasklets to the best performing one
  - The number of DPUs is 1, 4, 16, 64
  - We show the breakdown of execution time:
    - **DPU**: Execution time on the DPU
    - **Inter-DPU**: Time for inter-DPU communication via the host CPU
    - **CPU-DPU**: Time for CPU to DPU transfer of input data
    - **DPU-CPU**: Time for DPU to CPU transfer of final results

- Speedup over 1 DPU
Strong Scaling: 1 Rank (II)

VA, GEMV, SpMV, SEL, UNI, BS, TS, MLP, HST-S, HSTS-L, RED, SCAN-SSA (both kernel), SCAN-RSS (both kernels), and TRNS (both kernels) scale linearly with the number of DPUs.

Scaling is sublinear for BFS and NW.

BFS suffers load imbalance due to irregular graph topology.

NW computes a diagonal of a 2D matrix in each iteration. More DPUs does not mean more parallelization in shorter diagonals.
Strong Scaling: 1 Rank (III)

VA, GEMV, SpMV, BS, TS, TRNS do not need inter-DPU synchronization

SEL, UNI, HST-S, HST-L, RED, SCAN-SSA, SCAN-RSS need inter-DPU synchronization but 64 DPUs still obtain the best performance

BFS, MLP, NW require heavy inter-DPU synchronization, involving DPU-CPU and CPU-DPU transfers
Strong Scaling: 1 Rank (IV)

VA, GEMV, TS, MLP, HST-S, HST-L, RED, SCAN-SSA, SCAN-RSS, TRNS use parallel transfers. CPU-DPU and DPU-CPU transfer times decrease as we increase the number of DPUs.

BS, NW use parallel transfers but do not reduce transfer times:
- BS transfers a complete array to all DPUs.
- NW does not use all DPUs in all iterations.

SpMV, SEL, UNI, BFS cannot use parallel transfers, as the transfer size per DPU is not fixed.

**PROGRAMMING RECOMMENDATION 5**
Parallel CPU-DPU/DPU-CPU transfers inside a rank of DPUs are recommended for real-world workloads when all transferred buffers are of the same size.
Strong Scaling: 32 Ranks (I)

- Strong scaling experiments on 32 rank
  - We set the number of tasklets to the best performing one
  - The number of DPUs is 256, 512, 1024, 2048
  - We show the breakdown of execution time:
    - DPU: Execution time on the DPU
    - Inter-DPU: Time for inter-DPU communication via the host CPU
    - We do not show CPU-DPU/DPU-CPU transfer times
  - Speedup over 256 DPUs
Strong Scaling: 32 Ranks (II)

VA, GEMV, SEL, UNI, BS, TS, MLP, HST-S, HSTS-L, RED, SCAN-SSA (both kernel), SCAN-RSS (both kernels), and TRNS (both kernels) scale linearly with the number of DPUs.

SpMV, BFS, NW do not scale linearly due to load imbalance.

**KEY OBSERVATION 14**

Load balancing across DPUs ensures linear reduction of the execution time spent on the DPUs for a given problem size, when all available DPUs are used (as observed in strong scaling experiments).
Strong Scaling: 32 Ranks (III)

**KEY OBSERVATION 15**

The overhead of merging partial results from DPUs in the host CPU is tolerable across all PrIM benchmarks that need it.

**KEY OBSERVATION 16**

Complex synchronization across DPUs (i.e., inter-DPU synchronization involving two-way communication with the host CPU) imposes significant overhead, which limits scalability to more DPUs.
Weak Scaling: 1 Rank

**KEY OBSERVATION 17**

Equally-sized problems assigned to different DPUs and little/no inter-DPU synchronization lead to linear weak scaling of the execution time spent on the DPUs (i.e., constant execution time when we increase the number of DPUs and the dataset size accordingly).

**KEY OBSERVATION 18**

Sustained bandwidth of parallel CPU-DPU/DPU-CPU transfers inside a rank of DPUs increases sublinearly with the number of DPUs.
CPU/GPU: Evaluation Methodology

• Comparison of both UPMEM-based PIM systems to state-of-the-art CPU and GPU
  - Intel Xeon E3-1240 CPU
  - NVIDIA Titan V GPU

• We use state-of-the-art CPU and GPU counterparts of PrIM benchmarks
  - https://github.com/CMU-SAFARI/prim-benchmarks

• We use the largest dataset that we can fit in the GPU memory

• We show overall execution time, including DPU kernel time and inter DPU communication
The 2,556-DPU and the 640-DPU systems outperform the CPU for all benchmarks except SpMV, BFS, and NW.

The 2,556-DPU and the 640-DPU are, respectively, 93.0x and 27.9x faster than the CPU for 13 of the PrIM benchmarks.
The 2,556-DPU outperforms the GPU for 10 PrIM benchmarks with an average of 2.54x.

The performance of the 640-DPU is within 65% the performance of the GPU for the same 10 PrIM benchmarks.
The UPMEM-based PIM system can outperform a state-of-the-art GPU on workloads with three key characteristics:

1. Streaming memory accesses
2. No or little inter-DPU synchronization
3. No or little use of integer multiplication, integer division, or floating point operations

These three key characteristics make a workload potentially suitable to the UPMEM PIM architecture.
The 640-DPU system consumes on average 1.64x less energy than the CPU for all 16 PrIM benchmarks. For 12 benchmarks, the 640-DPU system provides energy savings of 5.23x over the CPU.
The UPMEM-based PIM system provides large energy savings over a state-of-the-art CPU due to higher performance (thus, lower static energy) and less data movement between memory and processors.

The UPMEM-based PIM system provides energy savings over a state-of-the-art CPU/GPU on workloads where it outperforms the CPU/GPU. This is because the source of both performance improvement and energy savings is the same: the significant reduction in data movement between the memory and the processor cores, which the UPMEM-based PIM system can provide for PIM-suitable workloads.
Outline

• Introduction
  - Accelerator Model
  - UPMEM-based PIM System Overview

• UPMEM PIM Programming
  - Vector Addition
  - CPU-DPU Data Transfers
  - Inter-DPU Communication
  - CPU-DPU/DPU-CPU Transfer Bandwidth

• DRAM Processing Unit
  - Arithmetic Throughput
  - WRAM and MRAM Bandwidth

• PrIM Benchmarks
  - Roofline Model
  - Benchmark Diversity

• Evaluation
  - Strong and Weak Scaling
  - Comparison to CPU and GPU

• Key Takeaways
Key Takeaway 1

The UPMEM PIM architecture is fundamentally compute bound. As a result, the most suitable workloads are memory-bound.

The throughput saturation point is as low as ¼ OP/B, i.e., 1 integer addition per every 32-bit element fetched.
The most well-suited workloads for the UPMEM PIM architecture use no arithmetic operations or use only simple operations (e.g., bitwise operations and integer addition/subtraction).
**Key Takeaway 3**

The most well-suited workloads for the UPMEM PIM architecture require little or no communication across DPUs (inter-DPU communication).
Key Takeaway 4

**KEY TAKEAWAY 4**

- UPMEM-based PIM systems **outperform state-of-the-art CPUs in terms of performance** (by 23.2× on 2,556 DPUs for 16 PrIM benchmarks) and **energy efficiency on most of PrIM benchmarks**.

- UPMEM-based PIM systems **outperform state-of-the-art GPUs on a majority of PrIM benchmarks** (by 2.54× on 2,556 DPUs for 10 PrIM benchmarks), and the outlook is even more positive for future PIM systems.

- UPMEM-based PIM systems are **more energy-efficient than state-of-the-art CPUs and GPUs on workloads that they provide performance improvements** over the CPUs and the GPUs.
Understanding a Modern PIM Architecture

Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System

JUAN GÓMEZ-LUNA¹, IZZAT EL HAJJ², IVAN FERNANDEZ¹,³, CHRISTINA GIANNIOULA¹,⁴, GERALDO F. OLIVEIRA¹, AND ONUR MUTLU¹

¹ETH Zürich
²American University of Beirut
³University of Malaga
⁴National Technical University of Athens

Corresponding author: Juan Gómez-Luna (e-mail: juang@ethz.ch).

https://github.com/CMU-SAFARI/prim-benchmarks
Benchmarking Memory-Centric Computing Systems: Analysis of Real Processing-in-Memory Hardware

Juan Gómez-Luna  
ETH Zürich

Izzat El Hajj  
American University of Beirut

Ivan Fernandez  
University of Malaga

Christina Giannoula  
National Technical University of Athens

Geraldo F. Oliveira  
ETH Zürich

Onur Mutlu  
ETH Zürich

https://github.com/CMU-SAFARI/prim-benchmarks
Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture

Juan Gómez-Luna\textsuperscript{1} Izzat El Hajj\textsuperscript{2} Ivan Fernandez\textsuperscript{1,3} Christina Giannoula\textsuperscript{1,4} Geraldo F. Oliveira\textsuperscript{1} Onur Mutlu\textsuperscript{1}

\textsuperscript{1}ETH Zürich \hspace{1em} \textsuperscript{2}American University of Beirut \hspace{1em} \textsuperscript{3}University of Malaga \hspace{1em} \textsuperscript{4}National Technical University of Athens

https://github.com/CMU-SAFARI/prim-benchmarks
PrIM Repository

• All microbenchmarks, benchmarks, and scripts
• [https://github.com/CMU-SAFARI/prim-benchmarks](https://github.com/CMU-SAFARI/prim-benchmarks)

PrIM (Processing-In-Memory Benchmarks)

PrIM is the first benchmark suite for a real-world processing-in-memory (PIM) architecture. PrIM is developed to evaluate, analyze, and characterize the first publicly-available real-world processing-in-memory (PIM) architecture, the UPMEM PIM architecture. The UPMEM PIM architecture combines traditional DRAM memory arrays with general-purpose in-order cores, called DRAM Processing Units (DPUs), integrated in the same chip.

PrIM provides a common set of workloads to evaluate the UPMEM PIM architecture with and can be useful for programming, architecture and system researchers all alike to improve multiple aspects of future PIM hardware and software. The workloads have different characteristics, exhibiting heterogeneity in their memory access patterns, operations and data types, and communication patterns. This repository also contains baseline CPU and GPU implementations of PrIM benchmarks for comparison purposes.

PrIm also includes a set of microbenchmarks can be used to assess various architecture limits such as compute throughput and memory bandwidth.
Processing-Near-Memory
Real PNM Architectures
Programming General-purpose PIM

Dr. Juan Gómez Luna
Professor Onur Mutlu

ETH Zürich
SAFARI

Sunday, March 26, 2023