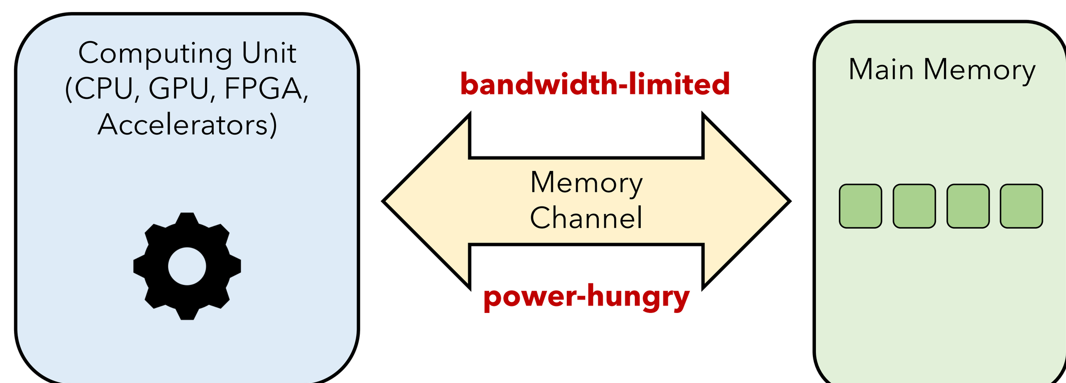
DaPPA: A Data-Parallel Framework for Processing-in-Memory Architectures

Geraldo F. Oliveira (*Ph.D. Candidate*) Onur Mutlu (*Ph.D. Advisor*)

ETH zürich SAFARI

Data Movement Bottleneck

Data movement is a major bottleneck in modern computer architectures

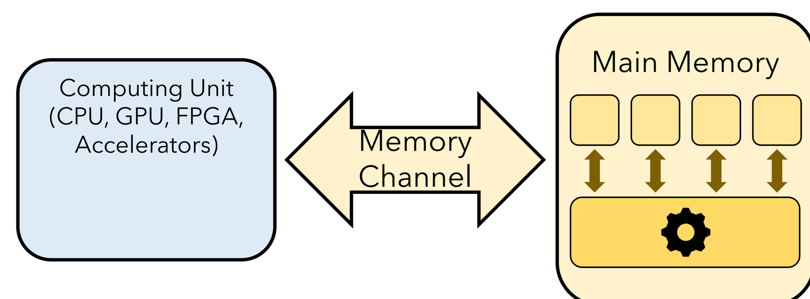


Over 60% of the total system energy is spent on data movement

Processing-in-Memory: Overview & Landscape

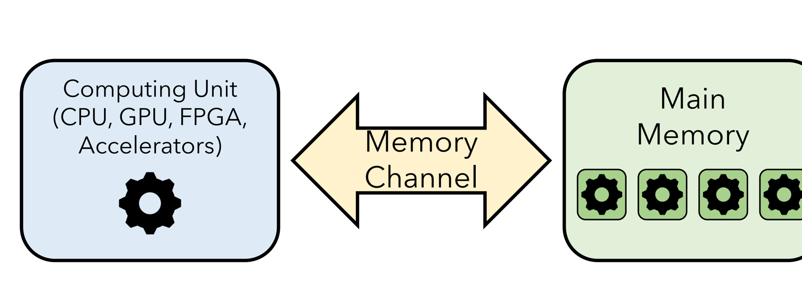
Processing-in-Memory: move computation to where the data resides

1. Processing-near-Memory



Processing-*near*-Memory leverages additional logic for computation

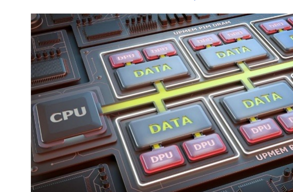
2. Processing-using-Memory



Processing-using-Memory uses memory's operational principles for computation

Processing-in-Memory architectures are now commercially available

UPMEM (2019)



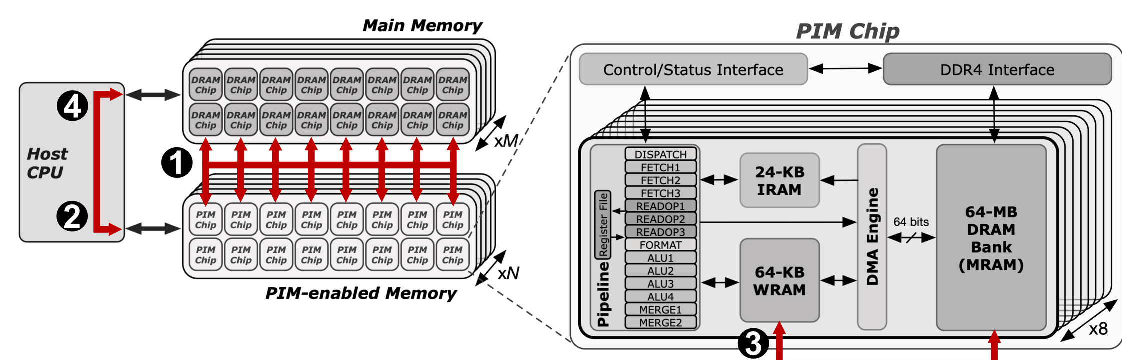
Near-DRAM-banks processing for general-purpose computing

Samsung HMB-PIM (2021)

Near-DRAM-banks processing for neural networks

The Programmability Issue

General-purpose PIM architectures (e.g., UPMEM) often employ a custom programming interface



Steps required to execute a program on UPMEM:

1. Split up input data and computation across PIM chips
2. Transfer input data from the main memory to PIM chips
3. Manually handle caching in working RAM (WRAM)
4. Transfer output data from PIM chips to main memory

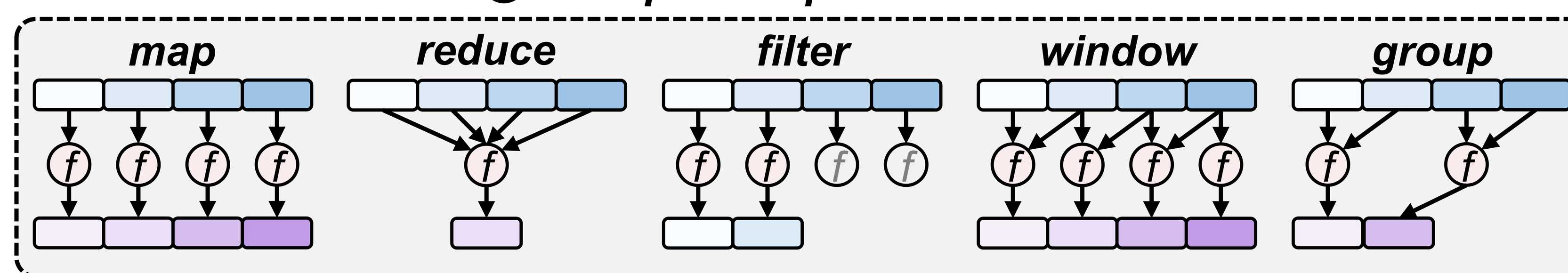
Programming a general-purpose PIM architecture leads to non-trivial effort → requires knowledge of the underlying hardware and manual fine-grained data movement handling

Our Goal

To ease programmability for the UPMEM architecture, allowing a programmer to write efficient PIM-friendly code without the need to *explicitly manage hardware resources*.

DaPPA: A Data-Parallel Processing-in-Memory Architecture

1 data-parallel pattern APIs



reduce

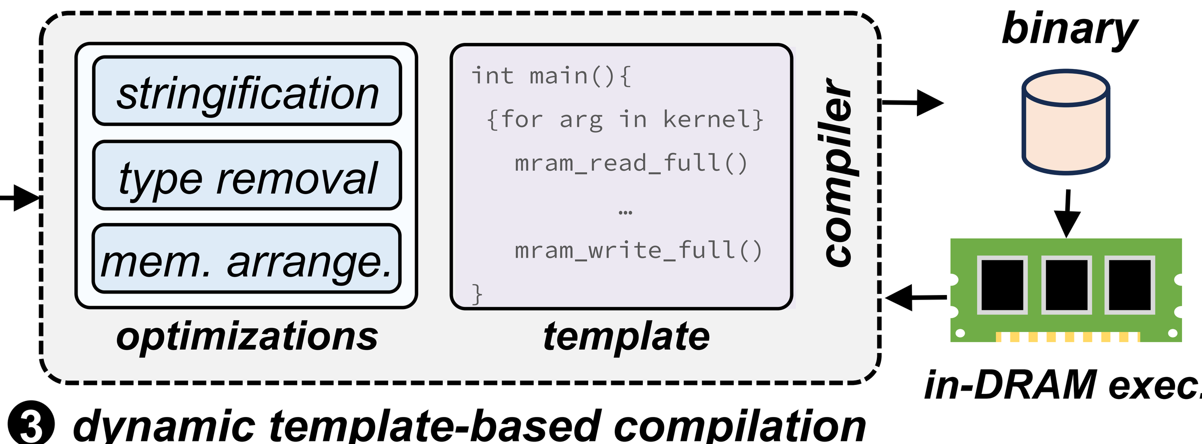
$C = A_0B_0 + A_1B_1 + A_2B_2 + A_3B_3$

map

target computation

```
UPMEM::Pipeline p(datasize);
p.stage(MAP(([](int *c, int *a, int *b){
        *c = *a * *b;
}), OUTPUT(int, &c),
INPUT(int, a), INPUT(int, b)));
p.stage(REDUCE(([](int *sum, int *c){
        *sum += *c;
}), REDUCE_OUT(int, &sum), INPUT(int, &c)));
```

2 dataflow programming interface



3 dynamic template-based compilation

DaPPA: Overview & Execution Steps

1 Data-Parallel Pattern APIs

- Pre-defined functions that implement high-level ***data-parallel pattern primitives***.
- DaPPA supports five primary data-parallel pattern primitives, including *(i)* ***map***, *(ii)* ***filter***, *(iii)* ***reduce***, *(iv)* ***window***, *(v)* ***group***.
- The user *can* ***combine all five data-parallel primitives*** to describe complex data transformations.

2 Dataflow Programming Interface

- DaPPA exposes a dataflow-based programming interface to the user.
- Main component: the ***Pipeline class*** - represents a sequence of data-parallel patterns.
- A Pipeline has one or more ***stages***, each of which utilizes a given data-parallel pattern that is executed in a pipeline fashion.

3 Dynamic Template-Based Compilation

- ***Template***: DaPPA creates a base UPMEM code based on a ***basic skeleton*** of a UPMEM application.
- ***Optimizations***: DaPPA uses a series of transformations to *(i)* ***extract*** data required by the UPMEM code template; *(ii)* calculate the ***memory offsets*** for MRAMs and WRAMs; *(iii)* ***divide computation*** between CPU and PIM cores.

Putting All Together

- Above, we showcase an example of implementing a simple ***vector dot product*** application using DaPPA.
- DaPPA ***generates*** the appropriate binary for the UPMEM system, ***executes*** the target computation in the PIM cores, and ***copies*** the final output from the PIM cores to the CPU.

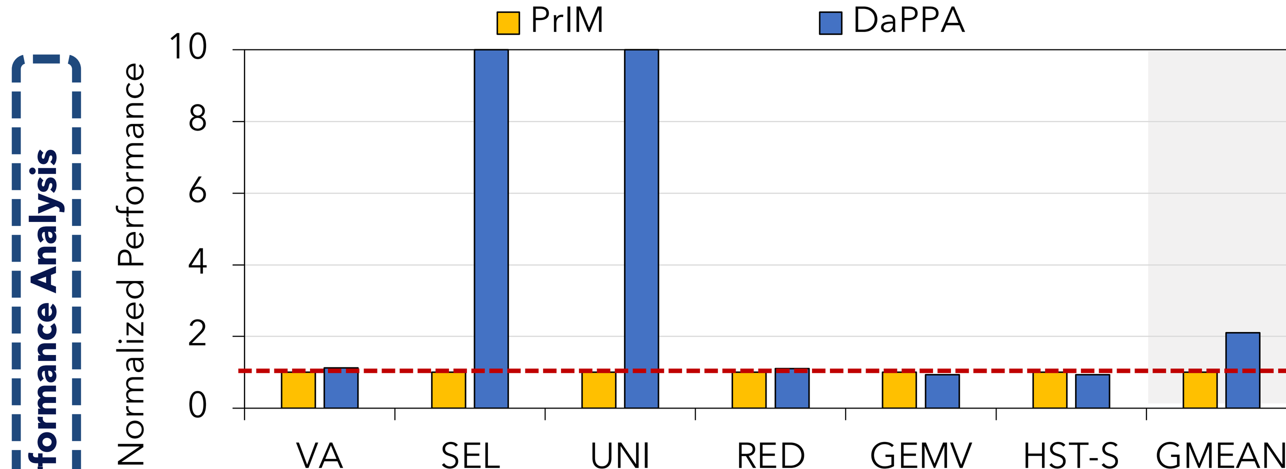
Evaluation

Experimental Setup

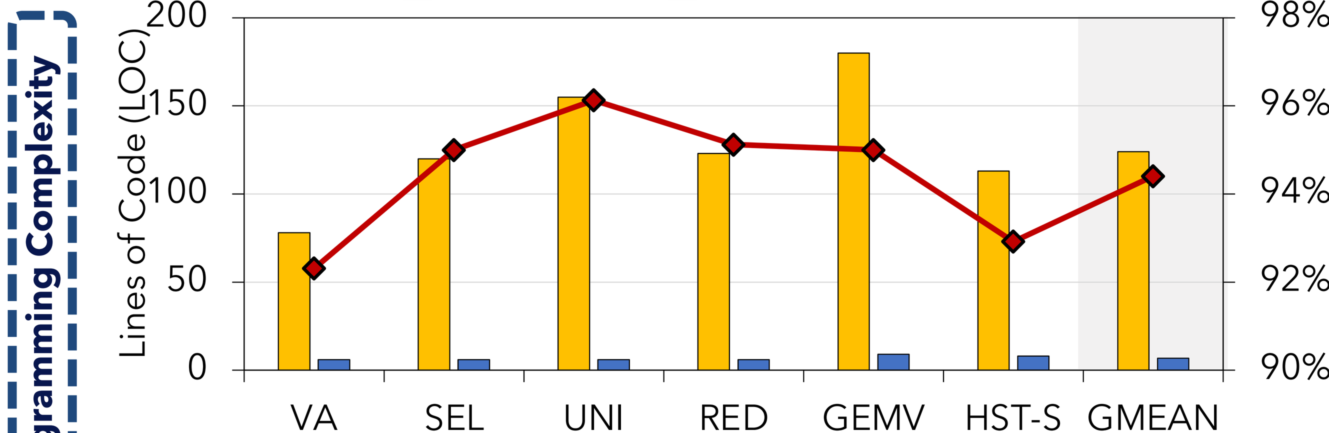
- **UPMEM system**
  - **Host CPU:** 2-socket Intel® Xeon Silver 4110 CPU
  - **PIM Cores:** 20 UPMEM PIM DIMMs (160 GB PIM memory)
  - **2560 DPUS**
- **We evaluate:**
  - Performance
  - Programming complexity (in lines of code)

Workloads

- **6 workloads from the PrIM benchmark suite**
  - Vector addition (VA)
  - Select (SEL)
  - Unique (UNI)
  - Reduce (RED)
  - General matrix-vector multiply (GEMV)
  - Histogram small (HST-S)
- **We compare:**
  - Hand-tuned PrIM implementations
  - DaPPA implementations



DaPPA *significantly improves* end-to-end performance compared to hand-tuned implementations



DaPPA *significantly reduces programming complexity*

Conclusion

Background: Processing-in-Memory (PiM) alleviates the performance and energy bottlenecks caused by data

- *The UPMEM system is an example of a real general-purpose PiM architecture*

Problem: Programming the UPMEM system requires non-trivial effort from the programmer

- The programmer needs to have knowledge of the hardware and manage data movement manually

Goal: Ease programmability for the UPMEM architecture, allowing one to write PIM code without prior knowledge of the hardware

DaPPA: A **D**ata-**P**arallel **P**rocessing-in-memory **A**rchitecture that

- provides a **data-parallel pattern-based programming interface** that abstracts hardware components;
- automatically ***distributes*** input and gathers output data, ***handles memory management***, and ***parallelizes work*** across PIM cores

Key Results: Our extensive evaluation shows that DaPPA

- ***outperforms*** hand-tuned UPMEM workloads by 2.1x
- ***reduces*** programming complexity by 94.4%