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

Geraldo F. Oliveira
Onur Mutlu

ETH Zürich
SAFARI

Wednesday, October 25, 2023
Executive Summary

Problem: Programming general-purpose processing-in-memory (PIM) systems require non-trivial effort
- The programmer needs to (1) have knowledge of the hardware and (2) manually manage data movement

Goal: Ease programmability of general-purpose PIM systems

Key Idea: DaPPA – A Data-Parallel PIM Architecture 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 works across PIM cores

Key Results: Our extensive evaluation shows that DaPPA
- outperforms hand-tuned workloads by 2.1x;
- reduces programming complexity (in lines of code) by 94.4%
Data Movement Bottleneck: The Problem

Data movement is a major bottleneck in modern computer architectures.

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

1 A. Boroumand et al., “Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks,” ASPLOS, 2018
Processing-in-Memory (PIM) architectures alleviate data movement by doing computation where the data resides.
Processing-in-Memory: Landscape of Real Systems

Processing-in-Memory architectures are now commercially available

**UPMEM**

Near-DRAM-banks processing for general-purpose computing

**HBM-PIM**

Near-DRAM-banks processing for neural networks
Processing-in-Memory: Landscape of Real Systems

Processing-in-Memory architectures are now commercially available

**UPMEM**

Near-DRAM-banks processing for general-purpose computing

**HBM-PIM**

Near-DRAM-banks processing for neural networks
Processing-in-Memory: The UPMEM Architecture

UPMEM: Near-DRAM-banks processing for general-purpose computing

Integration of UPMEM PIM in a system follows an accelerator model
The Programmability Barrier: Overview

Programming the UPMEM-based system requires:

1. **Splitting** input data and computation across PIM chips
2. **Transferring** input data from main memory to PIM chips
3. **Manually handling caching** in PIM’s scratchpad memory
4. **Transferring output data** from PIM chips to main memory
The Programmability Barrier: Vector Addition Example

Programmer’s Tasks:
The Programmability Barrier: Vector Addition Example

Programmer’s Tasks:

Align data

```
const unsigned int input_size = 1073741824; // Example value
const unsigned int input_size_8bytes =
    ((input_size * sizeof(T)) % 8) != 0
? roundup(input_size, 8) : input_size;

const unsigned int input_size_dpu = divceil(input_size, nr_of_dpus);
const unsigned int input_size_dpu_8bytes =
    ((input_size_dpu * sizeof(T)) % 8) != 0
? roundup(input_size_dpu, 8) : input_size_dpu;
```
unsigned int kernel = 0;
dpu_arguments_t input_arguments[NR_DPUS];
for(i=0; i<nr_of_dpus-1; i++) {
    input_arguments[i].size = input_size_dpu_8bytes * sizeof(T);
    input_arguments[i].transfer_size = input_size_dpu_8bytes * sizeof(T);
    input_arguments[i].kernel  = kernel;
}
input_arguments[nr_of_dpus-1].size =
    (input_size_8bytes - input_size_dpu_8bytes * (NR_DPUS-1)) * sizeof(T);
input_arguments[nr_of_dpus-1].transfer_size =
    input_size_dpu_8bytes * sizeof(T);
input_arguments[nr_of_dpus-1].kernel = kernel;
The Programmability Barrier: Vector Addition Example

Programmer’s Tasks:

<table>
<thead>
<tr>
<th>Align data</th>
<th>Collect parameters</th>
<th>Distribute parameters</th>
</tr>
</thead>
</table>

```c
DPU_ASSERT(dpu_push_xfer(dpu_set, DPU_XFER_TO_DPU, "DPU_INPUT_ARGUMENTS", 0, sizeof(input_arguments[0]), DPU_XFER_DEFAULT));
DPU_FOREACH(dpu_set, dpu, i) {
    DPU_ASSERT(dpu_prepare_xfer(dpu, bufferA + input_size_dpu_8bytes * i));
}
DPU_ASSERT(dpu_push_xfer(dpu_set, DPU_XFER_TO_DPU, DPU_MRAM_HEAP_POINTER_NAME, 0, input_size_dpu_8bytes * sizeof(T), DPU_XFER_DEFAULT));
DPU_FOREACH(dpu_set, dpu, i) {
    DPU_ASSERT(dpu_prepare_xfer(dpu, bufferB + input_size_dpu_8bytes * i));
}
DPU_ASSERT(dpu_push_xfer(dpu_set, DPU_XFER_TO_DPU, DPU_MRAM_HEAP_POINTER_NAME, input_size_dpu_8bytes * sizeof(T), input_size_dpu_8bytes * sizeof(T), DPU_XFER_DEFAULT));
```
The Programmability Barrier: Vector Addition Example

Programmer’s Tasks:

<table>
<thead>
<tr>
<th>Align data</th>
<th>Collect parameters</th>
<th>Distribute parameters</th>
<th>Launch computation</th>
</tr>
</thead>
</table>

DPU_ASSERT(dpu_launch(dpu_set, DPU_SYNCHRONOUS));
# The Programmability Barrier: Vector Addition Example

## Programmer’s Tasks:

<table>
<thead>
<tr>
<th>Align data</th>
<th>Collect parameters</th>
<th>Distribute parameters</th>
<th>Launch computation</th>
<th>Collect results</th>
</tr>
</thead>
</table>

```c
i = 0;
DPU_FOREACH(dpu_set, dpu, i) {
    DPU_ASSERT(dpu_prepare_xfer(dpu,
        bufferC + input_size_dpu_8bytes * i));
}
DPU_ASSERT(dpu_push_xfer(dpu_set, DPU_XFER_FROM_DPU,
    DPU_MRAM_HEAP_POINTER_NAME,
    input_size_dpu_8bytes * sizeof(T),
    input_size_dpu_8bytes * sizeof(T),
    DPU_XFER_DEFAULT));
```
The Programmability Barrier: Vector Addition Example

Programmer’s Tasks:

<table>
<thead>
<tr>
<th>Align data</th>
<th>Collect parameters</th>
<th>Distribute parameters</th>
<th>Launch computation</th>
<th>Collect results</th>
<th>Manage scratchpad</th>
</tr>
</thead>
</table>

```
barrier_wait(&my_barrier);

uint32_t input_size_dpu_bytes = DPU_INPUT_ARGUMENTS.size;
uint32_t input_size_dpu_bytes_transfer = DPU_INPUT_ARGUMENTS.transfer_size;
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);

T *cache_A = (T *) mem_alloc(BLOCK_SIZE);
T *cache_B = (T *) mem_alloc(BLOCK_SIZE);
```
```c
for (int byte_index = base_tasklet; byte_index < input_size_dpu_bytes;
     byte_index += BLOCK_SIZE * NR_TASKLETS)
{
    uint32_t l_size_bytes = (byte_index + BLOCK_SIZE >=
                               input_size_dpu_bytes)
                        ? (input_size_dpu_bytes - byte_index) : BLOCK_SIZE;

    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);
    vector_addition(cache_B, cache_A, l_size_bytes >> DIV);
    mram_write(cache_B, (__mram_ptr void*)(mram_base_addr_B + byte_index) +
               l_size_bytes);
}
```

---

**The Programmability Barrier:**

**Vector Addition Example**

**Programmer’s Tasks:**

- **Align data**
- **Collect parameters**
- **Distribute parameters**
- **Launch computation**
- **Collect results**
- **Manage scratchpad**
- **Orchestrate computation**
The Programmability Barrier: Vector Addition Example

Programmer’s Tasks:

- Align data
- Collect parameters
- Distribute parameters
- Launch computation
- Collect results
- Manage scratchpad
- Orchestrate computation

Goal: Just write my kernel

static void vector_addition(T *bufferB, T *bufferA, int l_size){
    for (unsigned int i = 0; i < l_size; i++){
        bufferB[i] += bufferA[i];
    }
}

SAFARI
The Programmability Barrier: Summary

Programmer’s Tasks:
- Align data
- Collect parameters
- Distribute parameters
- Launch computation
- Collect results
- Manage scratchpad
- Orchestrate computation

Goal:
- Just write my kernel

Problem:

Programming the UPMEM system leads to non-trivial effort → requires knowledge of the underlying hardware and manual fine-grained data movement handling

```c
static void vector_addition(T *bufferB, T *bufferA, int l_size) {
    for (unsigned int i = 0; i < l_size; i++) {
        bufferB[i] += bufferA[i];
    }
}
```
Our Goal

To ease programmability for the UPMEM system, allowing a programmer to write efficient PIM-friendly code without the need to explicitly manage hardware resources.
**DaPPA:**
Key Idea & Overview

Leverage an intuitive data-parallel pattern-based interface for PIM programming

**DaPPA**, a Data-Parallel PIM Architecture that automatically distributes input and gathers output data, handles memory management, and parallelizes work across PIM cores

**DaPPA is composed of three main components:**

1. Data-Parallel Pattern APIs
2. Dataflow Programming Interface
3. Dynamic Template-Based Compilation

*SAFARI*
DaPPA:
Key Idea & Overview

Leverage an intuitive data-parallel pattern-based interface for PIM programming

DaPPA, a Data-Parallel PIM Architecture that automatically distributes input and gathers output data, handles memory management, and parallelizes work across PIM cores

DaPPA is composed of three main components:

1. Data-Parallel Pattern APIs
2. Dataflow Programming Interface
3. Dynamic Template-Based Compilation

SAFARI
DaPPA: Data-Parallel Pattern APIs

- Skeleton and pattern-based parallel programming are a common abstraction for parallel architectures

- DaPPA supports five primary data-parallel patterns

  - map
  - reduce
  - filter
  - window
  - group

The user can combine all five data-parallel primitives to describe complex data transformations
DaPPA: Key Idea & Overview

**Key Idea**

Leverage an intuitive data-parallel pattern-based interface for PIM programming

DaPPA, a Data-Parallel PIM Architecture that automatically distributes input and gathers output data, handles memory management, and parallelizes work across PIM cores

DaPPA is composed of three main components:

1. Data-Parallel Pattern APIs
2. Dataflow Programming Interface
3. Dynamic Template-Based Compilation

SAFARI
DaPPA: Dataflow Programming Interface

DaPPA exposes a **dataflow-based programming interface** to the user

- Defines a collection of transformations over the input data
- A pipeline of stages, each representing a parallel pattern
- Data flows sequentially across each stage

**Pipeline**

- **Input**: Transformation stages
- **Output**: Final data

Each stage is a parallel pattern, and data flows sequentially across each stage.
DaPPA: Key Idea & Overview

Leverage an intuitive data-parallel pattern-based interface for PIM programming

DaPPA, a Data-Parallel PIM Architecture that automatically distributes input and gathers output data, handles memory management, and parallelizes work across PIM cores

DaPPA is composed of three main components:

1. Data-Parallel Pattern APIs
2. Dataflow Programming Interface
3. Dynamic Template-Based Compilation

SAFARI
DaPPA: Dynamic Template-Based Compilation

DaPPA uses a dynamic template-based compilation to generate PIM code in two main steps.

1. **Templating**: DaPPA creates a base UPMEM code based on a basic skeleton of a UPMEM application.
   - We use the Inja C++ templating engine.

2. **Optimizations**: DaPPA uses a series of transformations to
   - extract data required by the UPMEM code template
   - calculate the memory offsets for MRAMs and WRAMs
   - divide computation between CPU and PIM cores

DaPPA compiles and executes each stage in a Pipeline per time → allows for runtime optimizations.
DaPPA: Putting All Together

Example of DaPPA’s implementation of a **vector dot product operation**
DaPPA: Putting All Together

Example of DaPPA’s implementation of a vector dot product operation

Example of DaPPA’s implementation of a vector dot product operation
DaPPA: Putting All Together

Example of DaPPA’s implementation of a vector dot product operation

\[ C = A_0B_0 + A_1B_1 + A_2B_2 + A_3B_3 \]

1. data-parallel pattern APIs

2. dataflow programming interface
DaPPA:
Putting All Together

Example of DaPPA’s implementation of a vector dot product operation
Evaluation: Methodology Overview

• Evaluation Setup
  - **Host CPU:** 2-socket Intel® Xeon Silver 4110 CPU
  - **PIM Cores:** 20 UPMEM PIM DIMMs (160 GB PIM memory)
  - **2560 DPUs** in total

• **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**)

• **Metrics**
  - **End-to-end execution time** (average of 10 runs)
  - **Programming complexity** (in lines of code)
Evaluation: Performance Analysis

Large performance improvement due to parallel data transfers & host+PIM collaborative exec.
DaPPA significantly improves end-to-end performance compared to hand-tuned implementations.

**Evaluation:**
Performance Analysis

Normalized Performance

- VA
- SEL
- UNI
- RED
- GEMV
- HST-S
- GMEAN

PrIM
DaPPA

DaPPA significantly improves end-to-end performance compared to hand-tuned implementations.
Evaluation: Programming Complexity Analysis

DaPPA significantly reduces programming complexity by abstracting hardware components.
SimplePIM [Chen+, PACT’23]: a framework that uses (1) iterator functions and (2) primitives for communication to aid PIM programmability

Compared to SimplePIM, DaPPA provides three key benefits

1. **Higher abstraction level** → The programmer does not need to manually specify communication patterns used during computation

2. **Support for more parallel patterns** → DaPPA supports two more parallel primitives (window and group), and allows the mixing of parallel patterns

3. **Further execution optimizations** → DaPPA allows using idle host resources for collaborative execution

**DaPPA improves state-of-the-art frameworks for PIM programmability**
Conclusion

**Problem:** Programming general-purpose processing-in-memory (PIM) systems require **non-trivial effort**
- The programmer needs to (1) have **knowledge of the hardware** and (2) **manually manage** data movement

**Goal:** Ease **programmability** of general-purpose PIM systems

**Key Idea:** DaPPA – A **Data-Parallel PIM Architecture** 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** works across PIM cores

**Key Results:** Our extensive evaluation shows that DaPPA
- **outperforms** hand-tuned workloads by **2.1x**;
- reduces programming complexity (in lines of code) by **94.4%**
Next Steps & Follow-Up

• Support **irregular access patterns** with custom data parallel patterns

• Leverage **previously-implemented real-world workloads** implemented using parallel patterns in DaPPA

• **Improve evaluation**
  - More workloads from the PrIM benchmark suite
  - More metrics: energy, coding timing
  - Quantitative comparison to SimplePIM

• **Extend DaPPA** to other PIM architectures
  - Processing-using-DRAM architectures (e.g., SIMDRAM [Hajinazar & Oliveira+, ASPLOS’21])
  - Samsung HBM-PIM
Real PIM Tutorial (MICRO 2023)

- October 29th: Lectures + Hands-on labs + Invited lectures

https://events.safari.ethz.ch/micro-pim-tutorial/doku.php?id=start

https://youtube.com/live/ohUooNSIxOI?feature=share
PIM Review and Open Problems

A Modern Primer on Processing in Memory

Onur Mutlu\textsuperscript{a,b}, Saugata Ghose\textsuperscript{b,c}, Juan Gómez-Luna\textsuperscript{a}, Rachata Ausavarungnirun\textsuperscript{d}

SAFARI Research Group

\textsuperscript{a}ETH Zürich  
\textsuperscript{b}Carnegie Mellon University  
\textsuperscript{c}University of Illinois at Urbana-Champaign  
\textsuperscript{d}King Mongkut’s University of Technology North Bangkok

Onur Mutlu, Saugata Ghose, Juan Gomez-Luna, and Rachata Ausavarungnirun, "A Modern Primer on Processing in Memory"  

SAFARI

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

Geraldo F. Oliveira
Onur Mutlu

ETH Zürich
SAFARI

Wednesday, October 25, 2023