HEART 2024 Tutorial

Memory-Centric Computing Systems

Processing-Near-Memory

Programming General-purpose PIM

Geraldo F. Oliveira

Dr. Juan Gómez Luna

Professor Onur Mutlu

ETH zürich

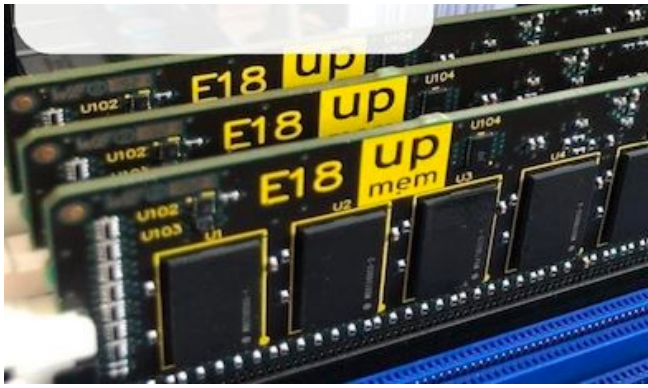
SAFARI

Friday, June 21, 2024

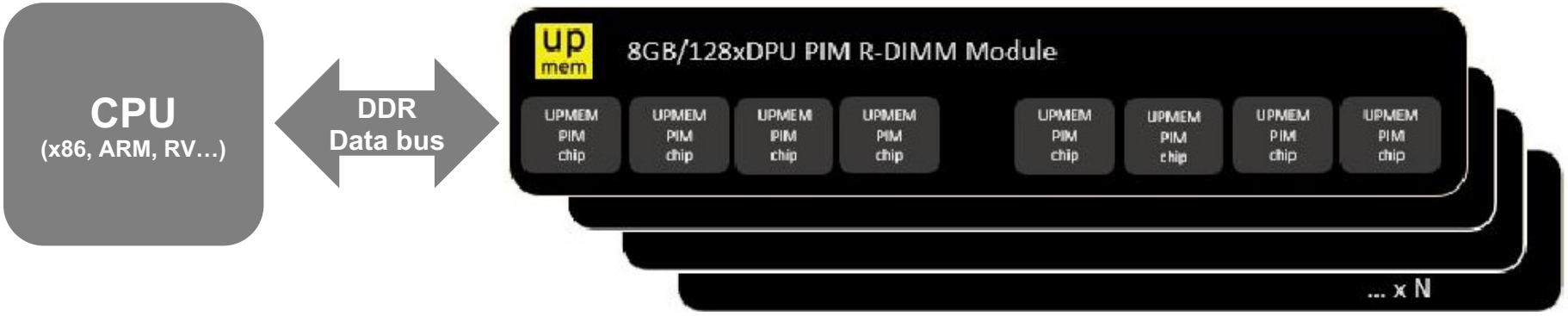
UPMEM PIM

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.



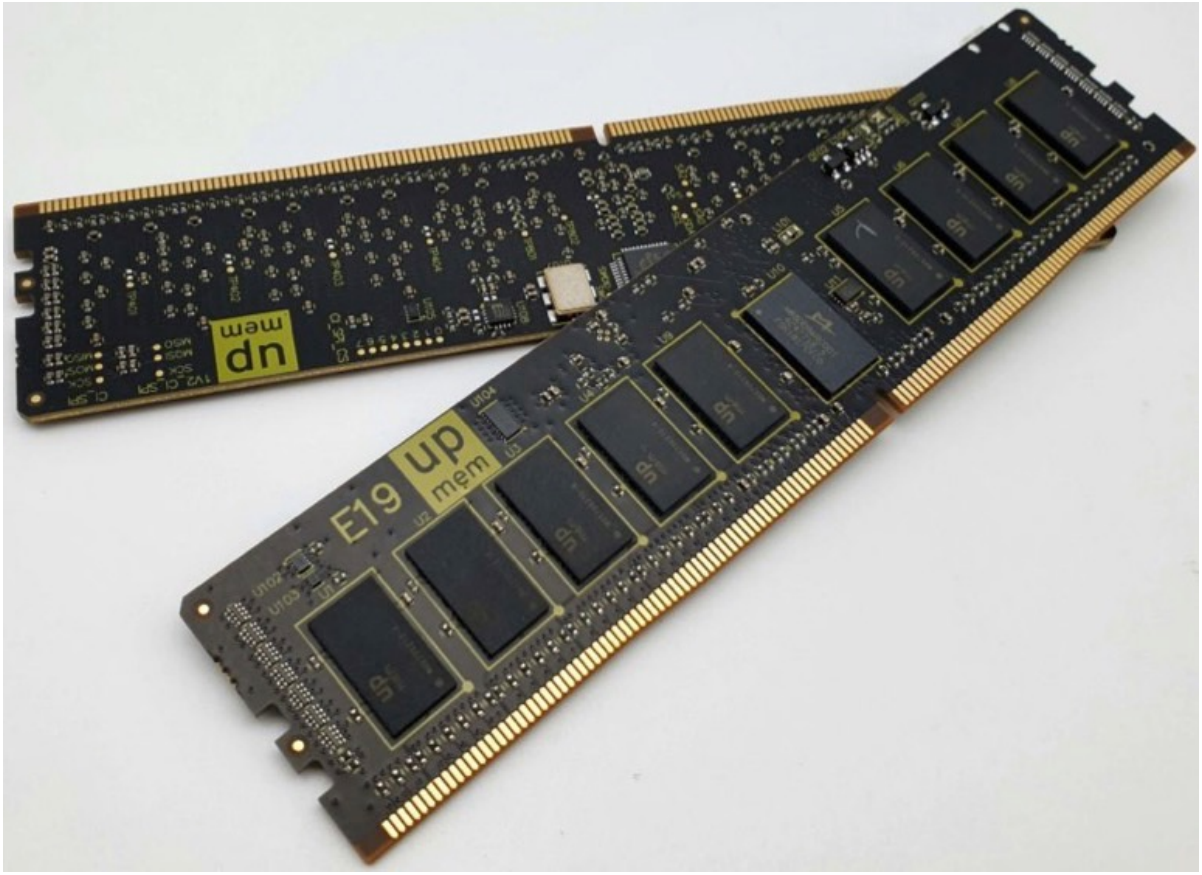
- Replaces **standard** DIMMs
  - DDR4 R-DIMM modules
    - 8GB+128 DPUs (16 PIM chips)
    - Standard 2x-nm DRAM process
  - **Large amounts of** compute & memory bandwidth



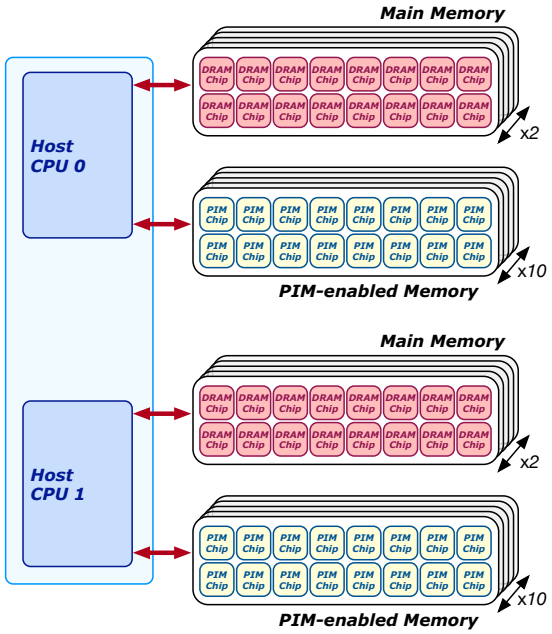
https://www.anandtech.com/show/14750/hot-chips-31-analysis-inmemory-processing-by-upmem
https://www.upmem.com/video-upmem-presenting-its-true-processing-in-memory-solution-hot-chips-2019/

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



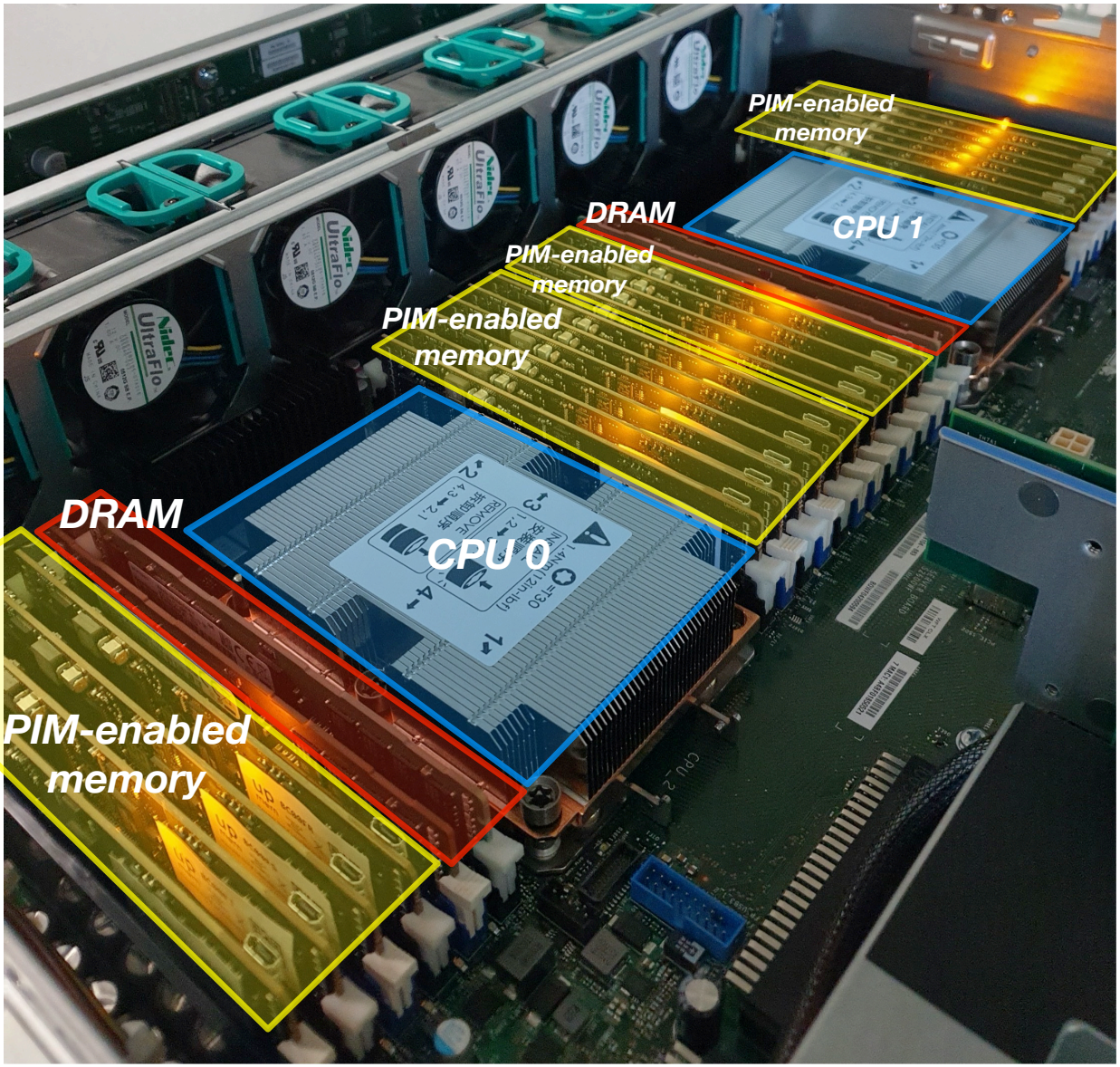
Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture

JUAN GÓMEZ-LUNA, ETH Zürich, Switzerland
IZZAT EL HAJJ, American University of Beirut, Lebanon
IVAN FERNANDEZ, ETH Zürich, Switzerland and University of Malaga, Spain
CHRISTINA GIANNOULA, ETH Zürich, Switzerland and NTUA, Greece
GERALDO F. OLIVEIRA, ETH Zürich, Switzerland
ONUR MUTLU, ETH Zürich, 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 insufficient 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 UPMEM company has designed and manufactured the first publicly-available real-world 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.

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 UPMEM-based PIM system using microbenchmarks to assess various architecture limits such as compute throughput and memory bandwidth, yielding new insights. Second, we present *PrIM* (*Processing-In-Memory benchmarks*), a benchmark suite of 16 workloads from different application domains (e.g., dense/sparse 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 PrIM benchmarks on the UPMEM 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 UPMEM-based PIM systems with 640 and 2,556 DPUs 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[1], IZZAT EL HAJJ[2], IVAN FERNANDEZ[1,3], CHRISTINA GIANNOULA[1,4], GERALDO F. OLIVEIRA[1], AND ONUR MUTLU[1]

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

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

https://arxiv.org/pdf/2105.03814.pdf

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

UPMEM Patent

(12) **United States Patent**
Devaux et al.

(10) **Patent No.: US 10,324,870 B2**
(45) **Date of Patent: Jun. 18, 2019**

(54) **MEMORY CIRCUIT WITH INTEGRATED PROCESSOR**

(71) Applicant: **UPMEM**, Grenoble (FR)

(72) Inventors: **Fabrice Devaux**, La Conversion (CH); **Jean-François Roy**, Grenoble (FR)

(73) Assignee: **UPMEM**, Grenoble (FR)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **15/551,418**

(22) PCT Filed: **Feb. 12, 2016**

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,666,485	A *	9/1997	Suresh	G06F 13/1605 710/113
6,463,001	B1	10/2002	Williams	
7,349,277	B2 *	3/2008	Kinsley	G11C 11/406 365/193
8,438,358	B1 *	5/2013	Kraipak	G11C 7/04 711/167

(Continued)

FOREIGN PATENT DOCUMENTS

EP	0780768	A1	6/1997
JP	H03109661	A	5/1991
WO	2010/141221	A1	12/2010

(57) **ABSTRACT**

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.

Fabrice Devaux, Jean-François Roy. “Memory circuit with integrated processor.” US 10,324,870 B2.

UPMEM PIM System Organization (I)

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

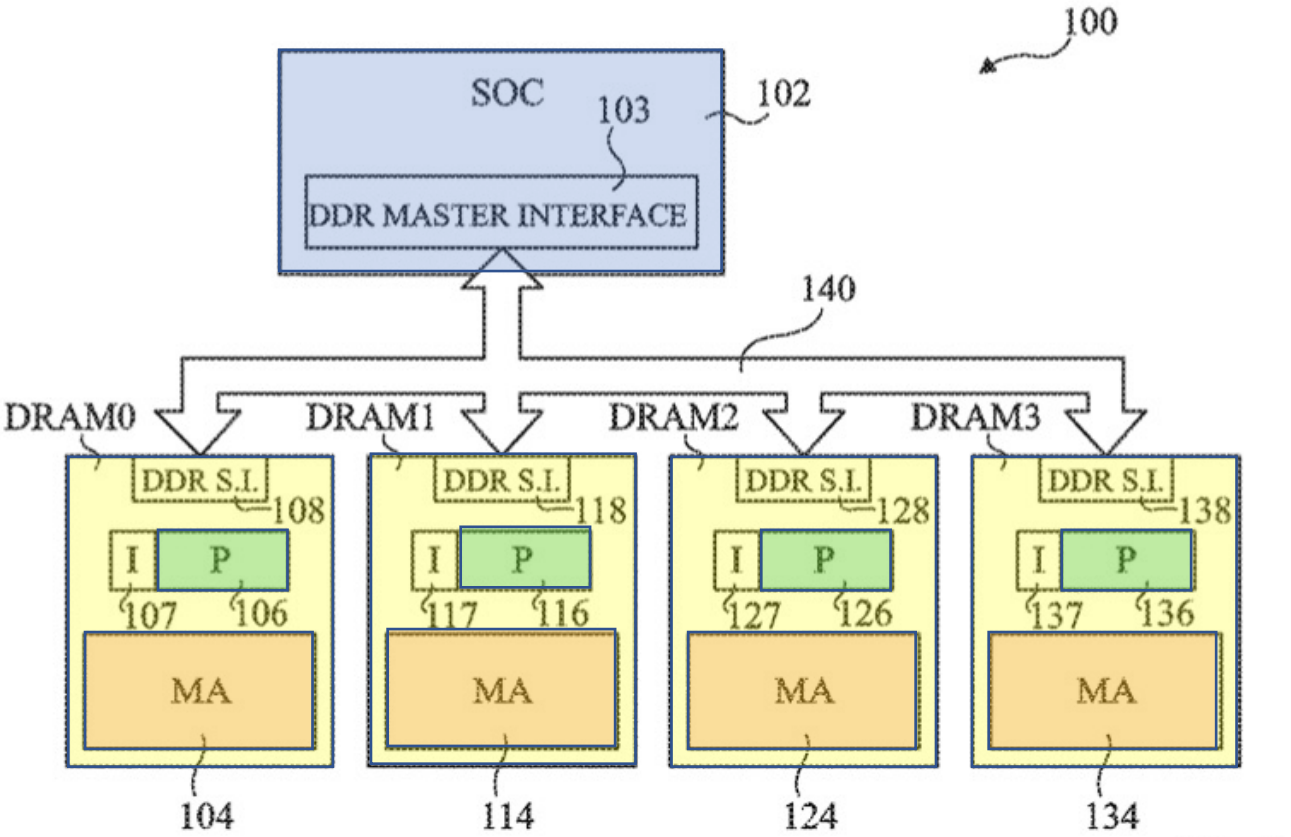
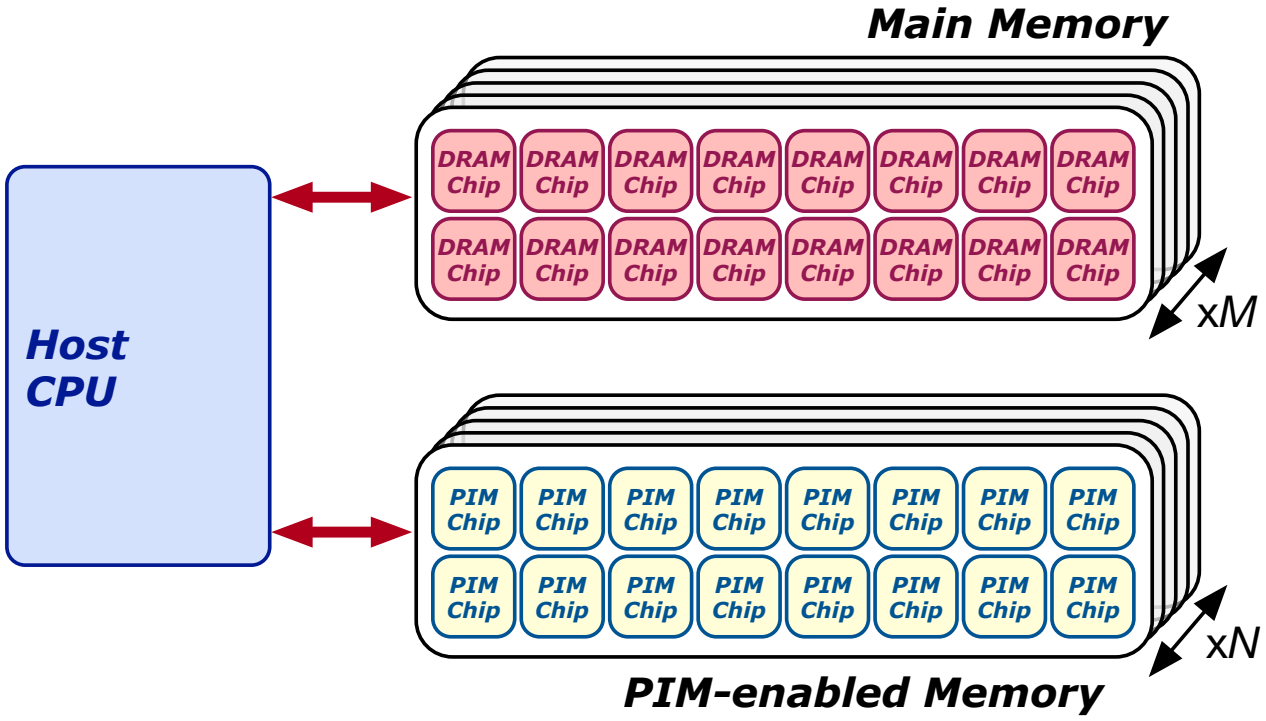


Fig 1

Fabrice Devaux, Jean-François Roy. “Memory circuit with integrated processor.” US 10,324,870 B2.

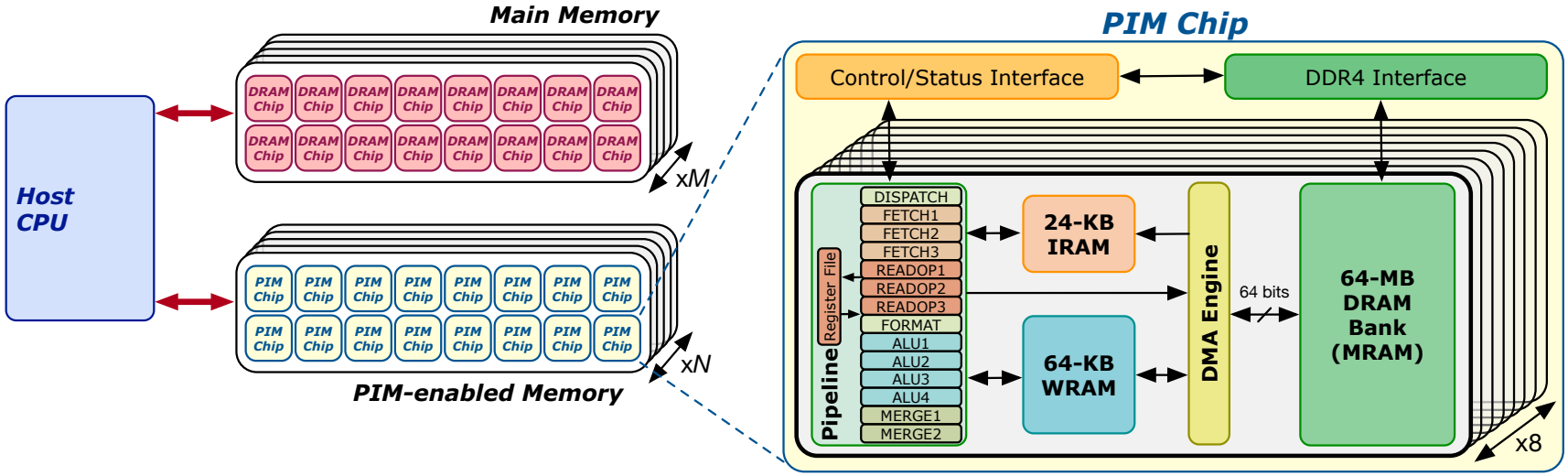
UPMEM PIM System Organization (II)

- 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*

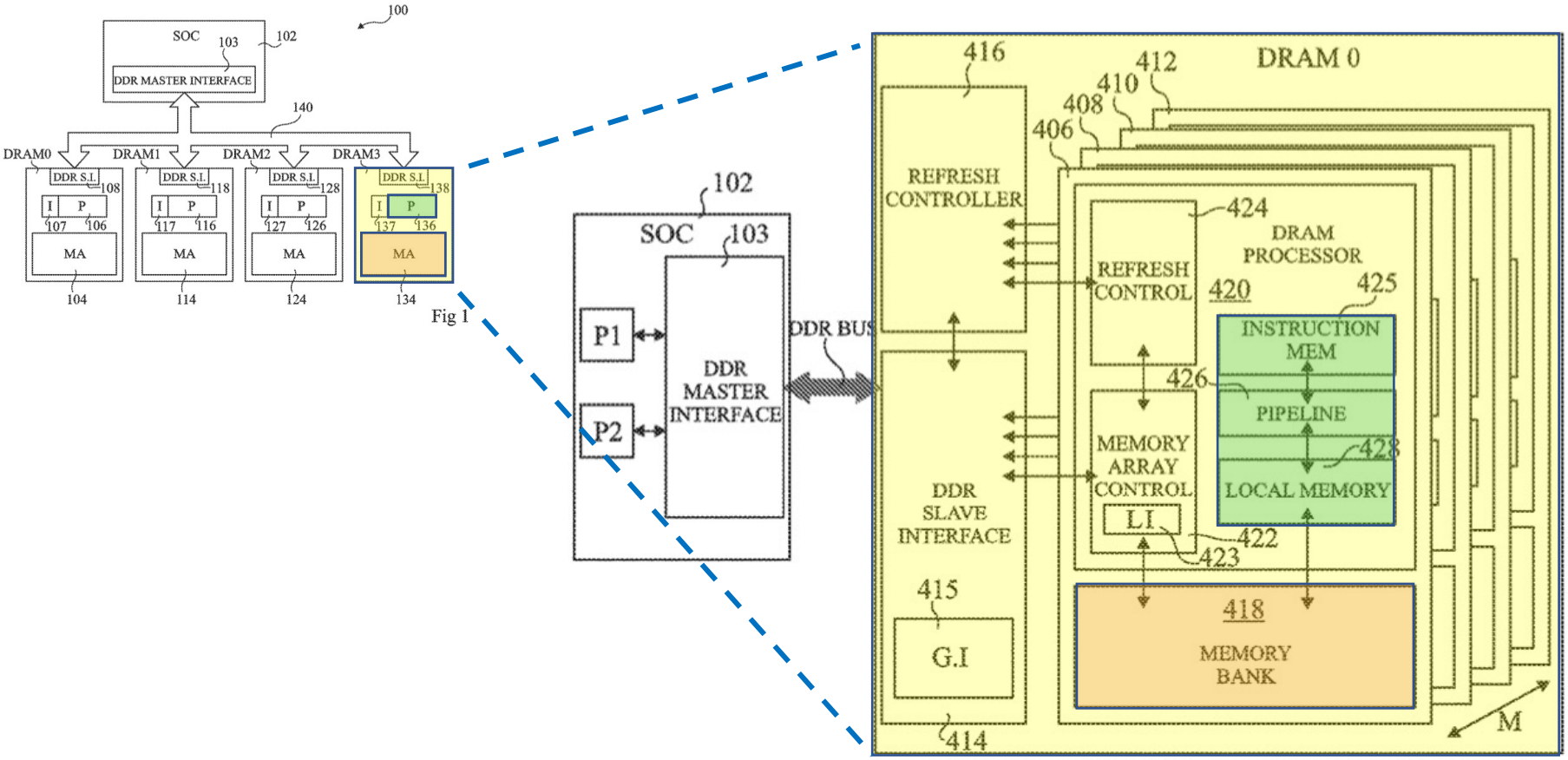
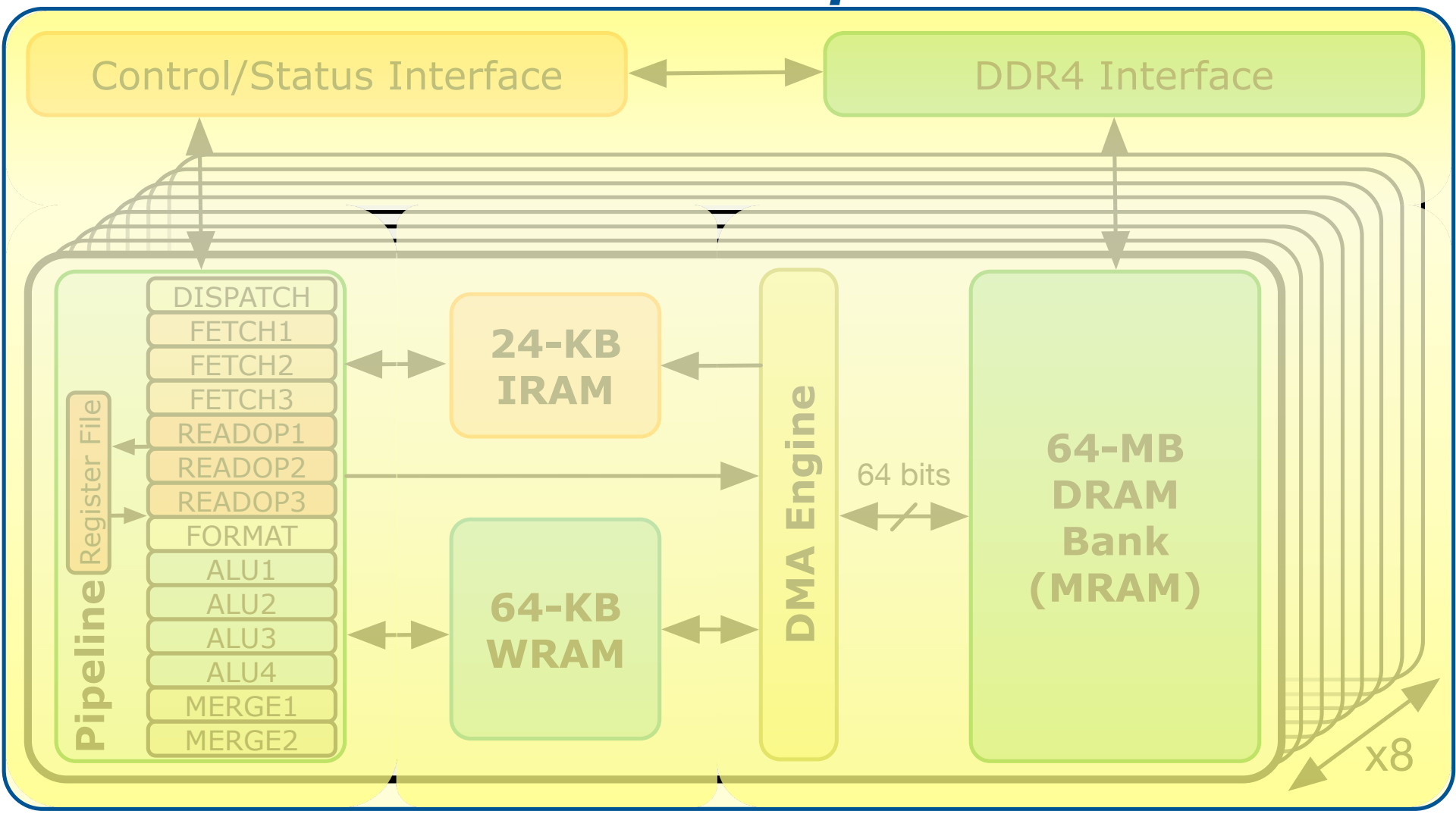


Fig 4

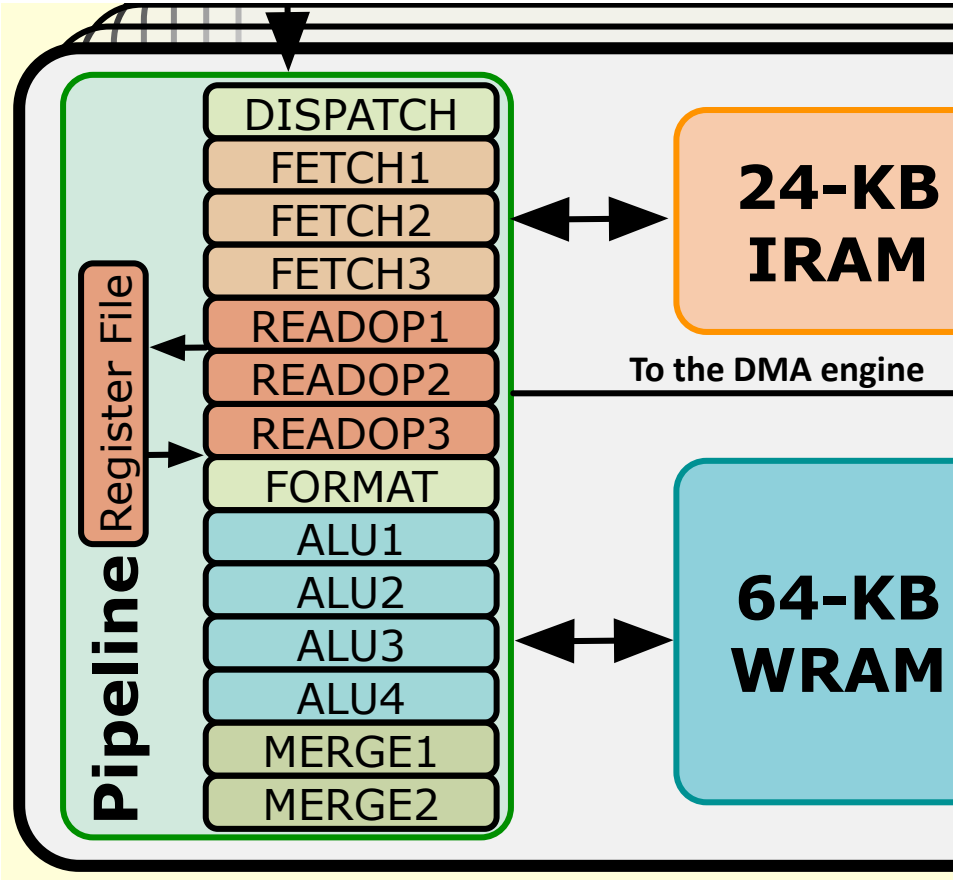
Fabrice Devaux, Jean-François Roy. “Memory circuit with integrated processor.” US 10,324,870 B2.

DRAM Processing Unit (II)



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)

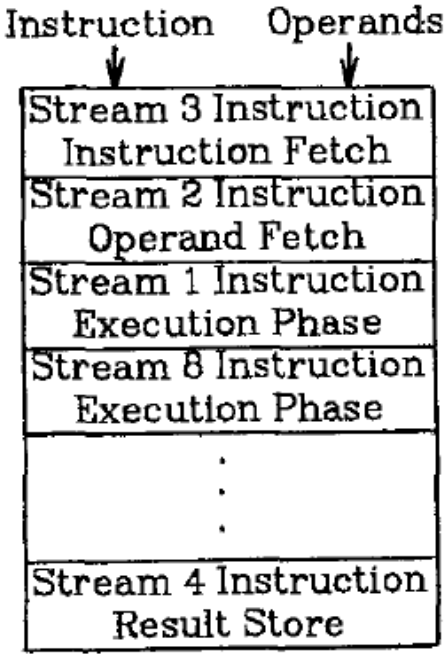
- Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
  - Branch/instruction resolution latency overlapped with execution of other threads'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



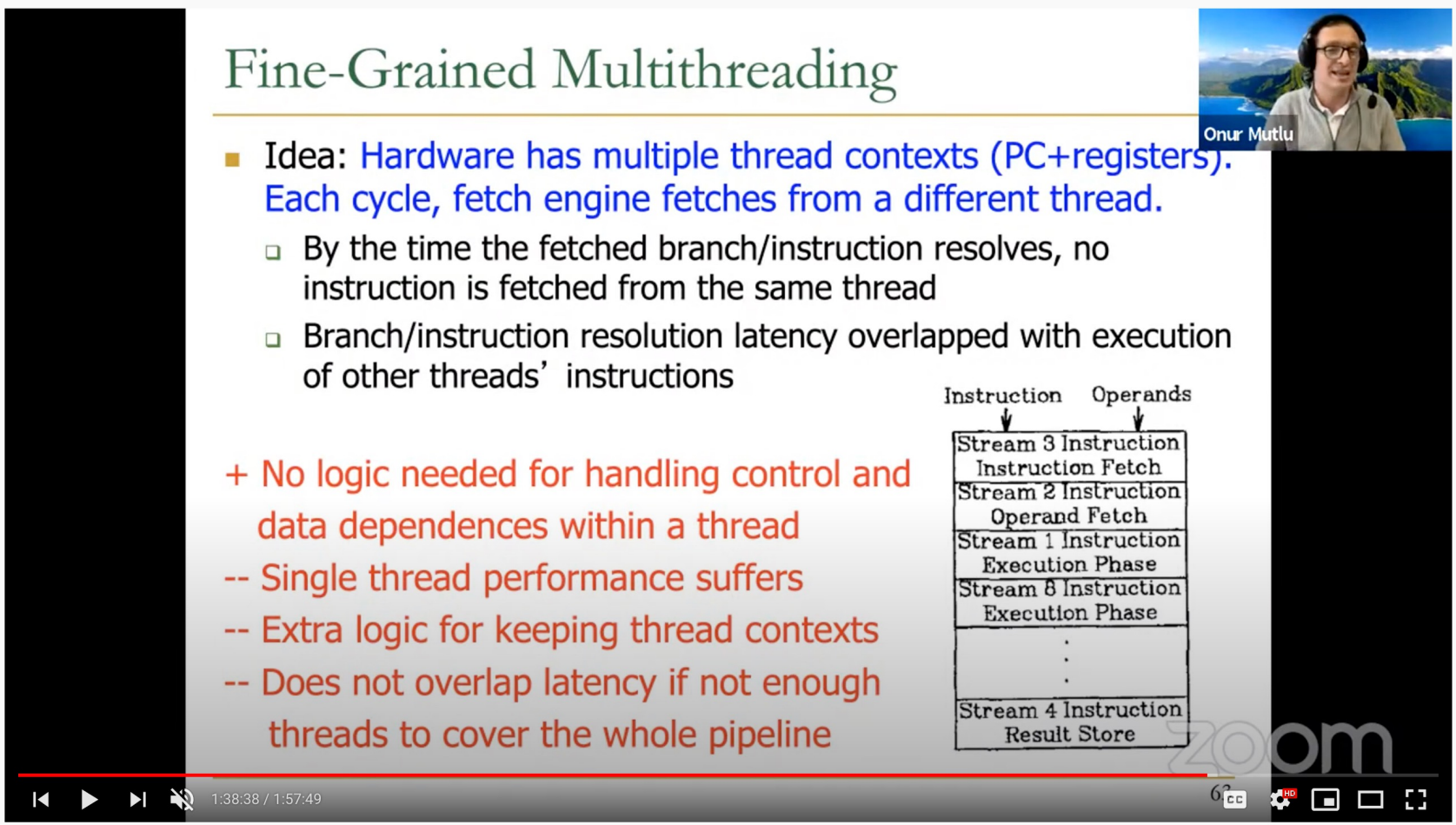
Fine-Grained Multithreading (II)

- 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

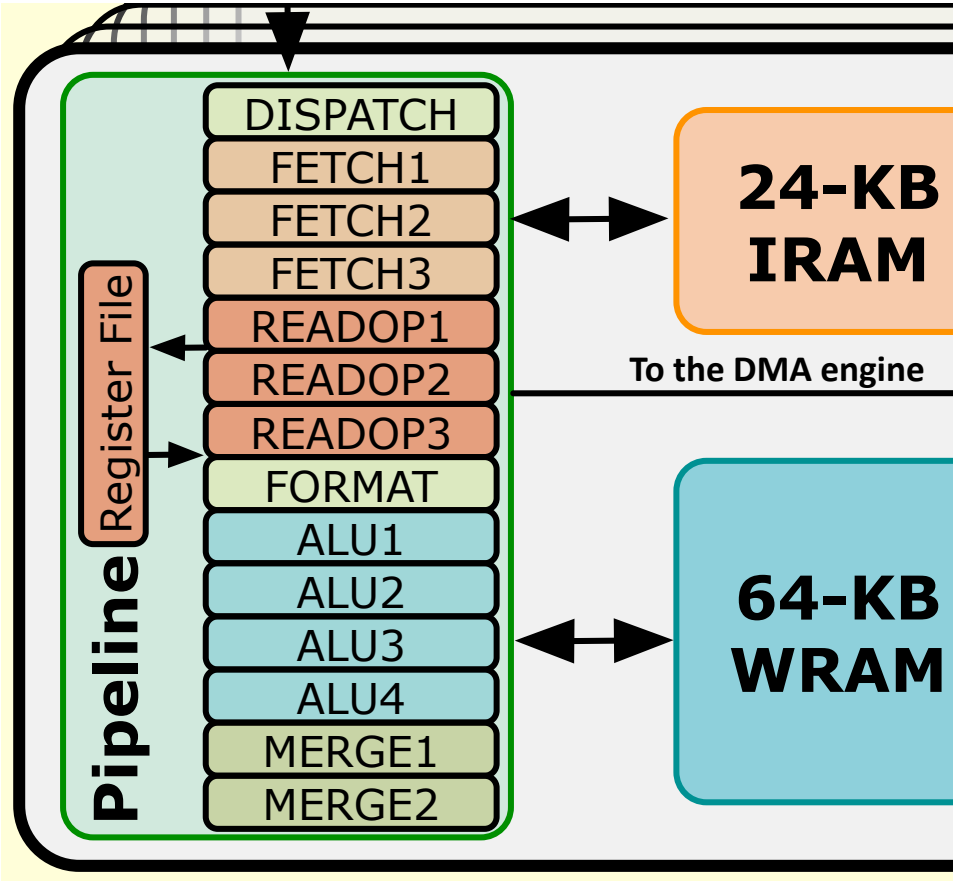
- Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964
- Smith, "A pipelined, shared resource MIMD computer," ICPP 1978

Lecture on Fine-Grained Multithreading



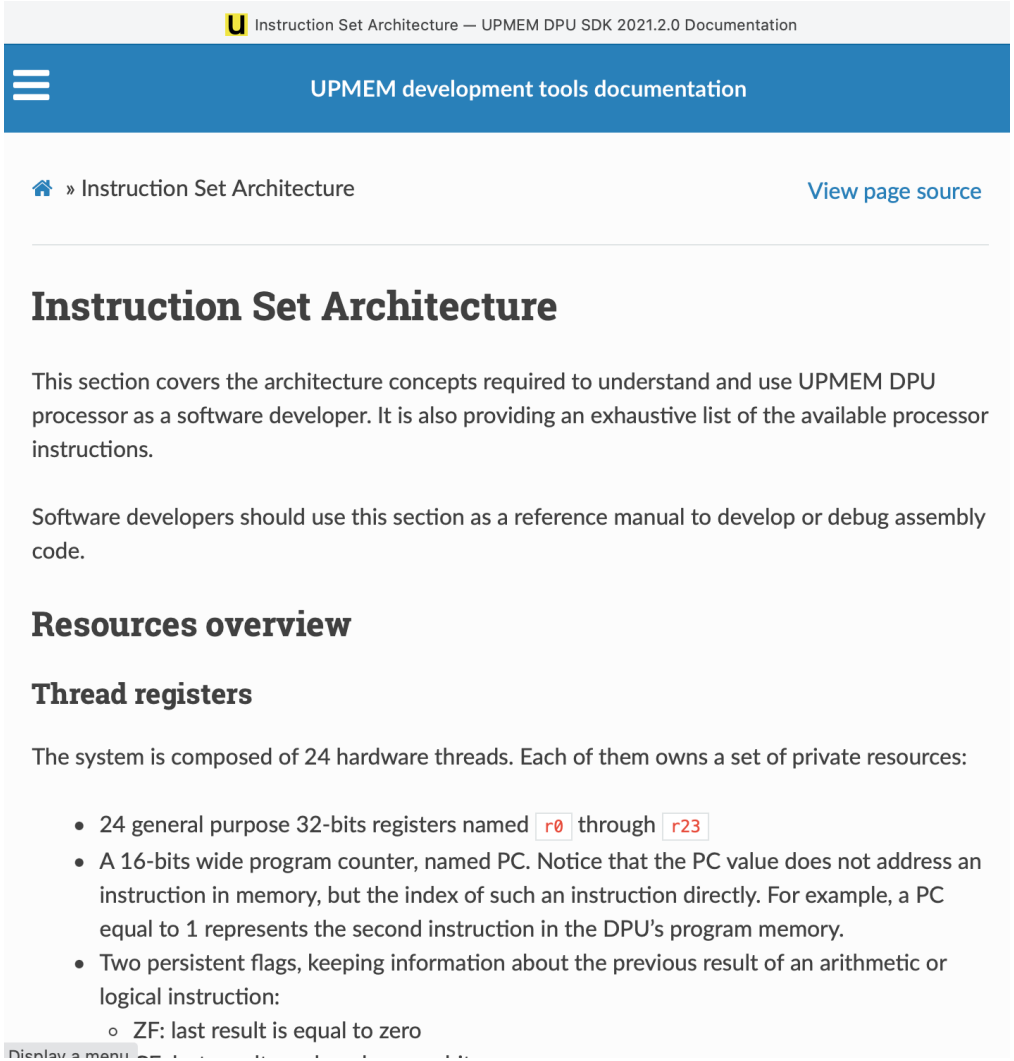
DPU Pipeline

- 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



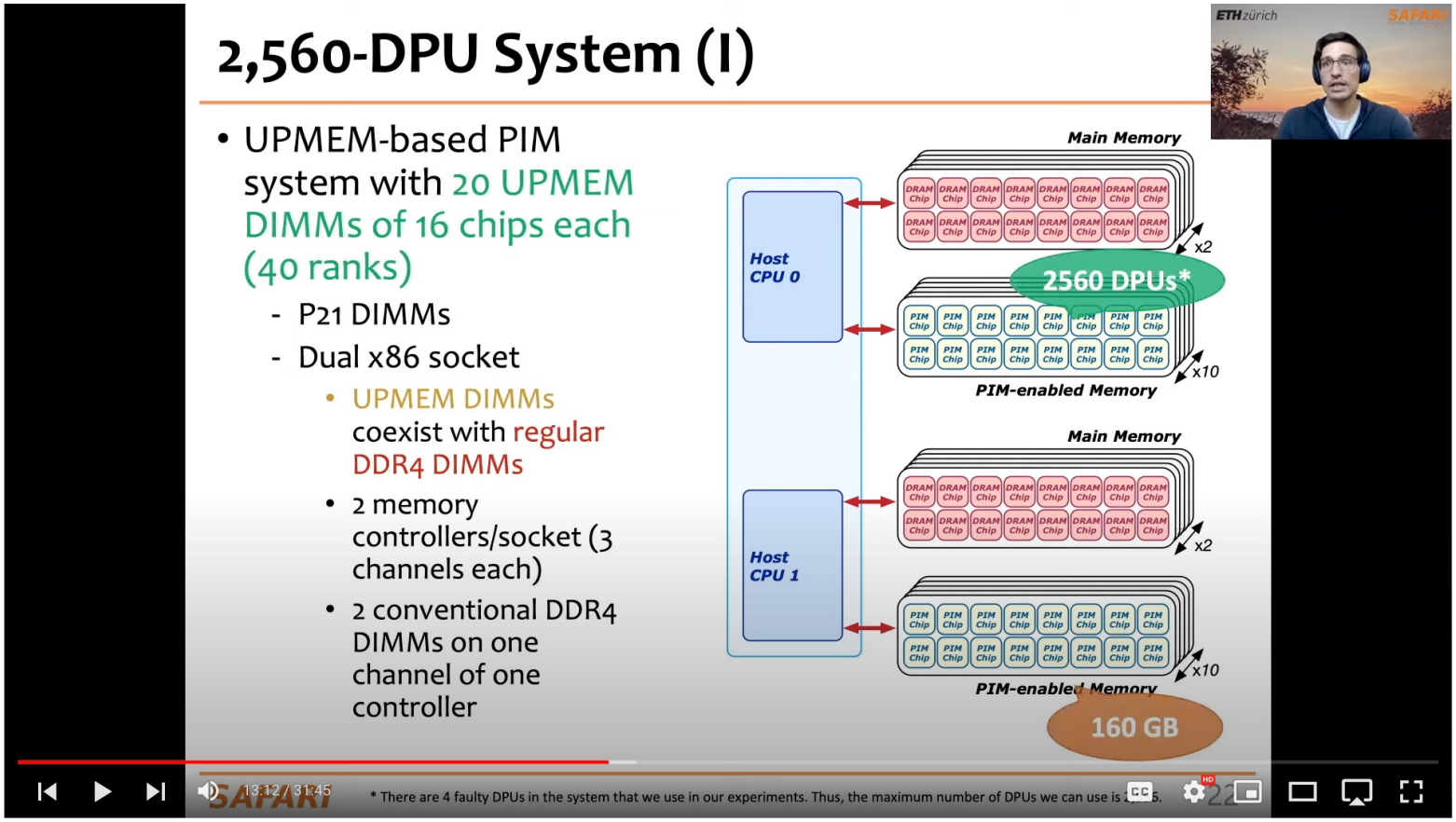
DPU Instruction Set Architecture

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



https://sdk.upmem.com/2021.2.0/201_IS.html#

More on the UPMEM PIM Architecture



PIM Course: Lecture 3: Real-world PIM: UPMEM PIM - Fall 2022

Onur Mutlu Lectures
31.7K subscribers
Subscribed
18
Share
Clip
Save

564 views 4 months ago Livestream - P&S Data-Centric Architectures: Fundamentally Improving Performance and Energy (Fall 2022)
Projects & Seminars, ETH Zürich, Fall 2022
Data-Centric Architectures: Fundamentally Improving Performance and Energy
(https://safari.ethz.ch/projects_and_s...) Show more

https://youtu.be/p_sLhKeo6ys
https://youtu.be/7c6x5GJG6dw

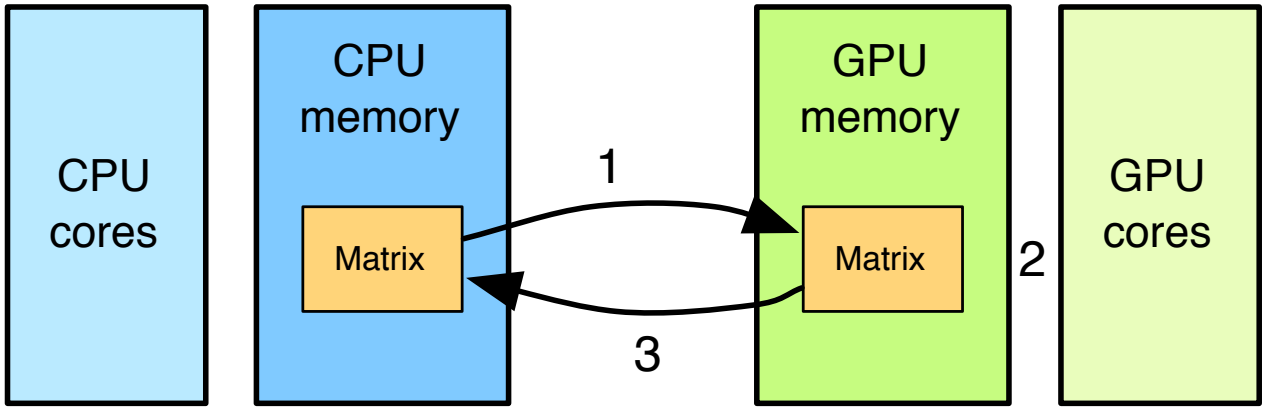
Programming a General-purpose PIM System

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
https://safari.ethz.ch/digitaltechnik/spring2018/lib/exe/fetch.php?media=digitaldesign-2018-lecture22-gpuprogramming-afterlecture.pdf

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*

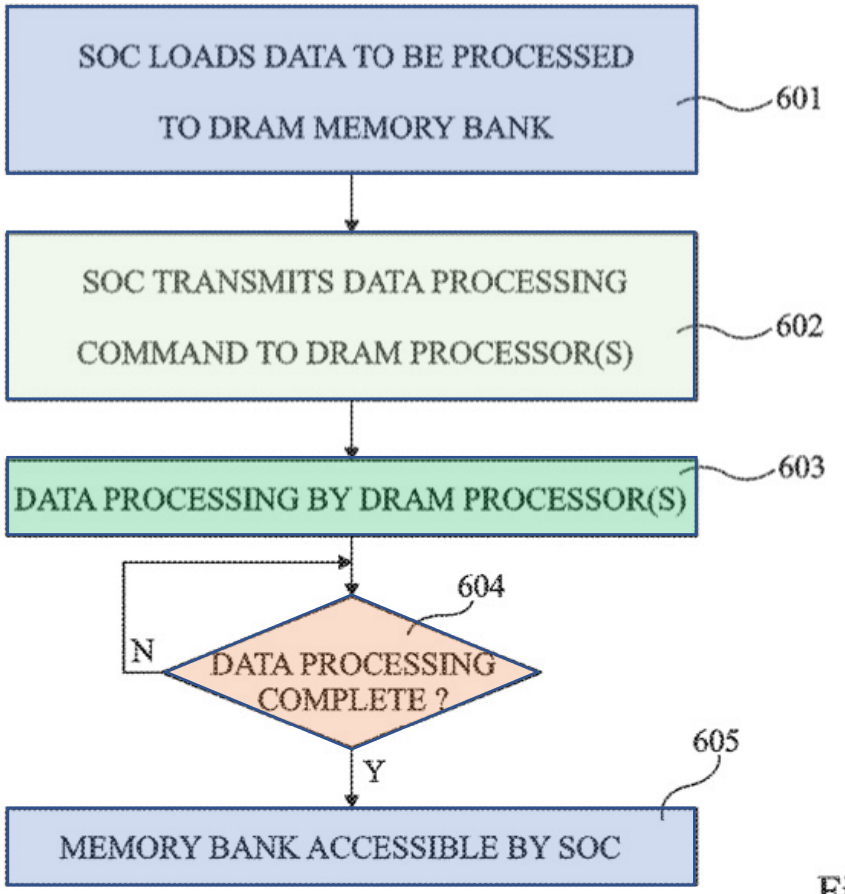


Fig 6

Fabrice Devaux, Jean-François Roy. "Memory circuit with integrated processor." US 10,324,870 B2.

System Organization

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

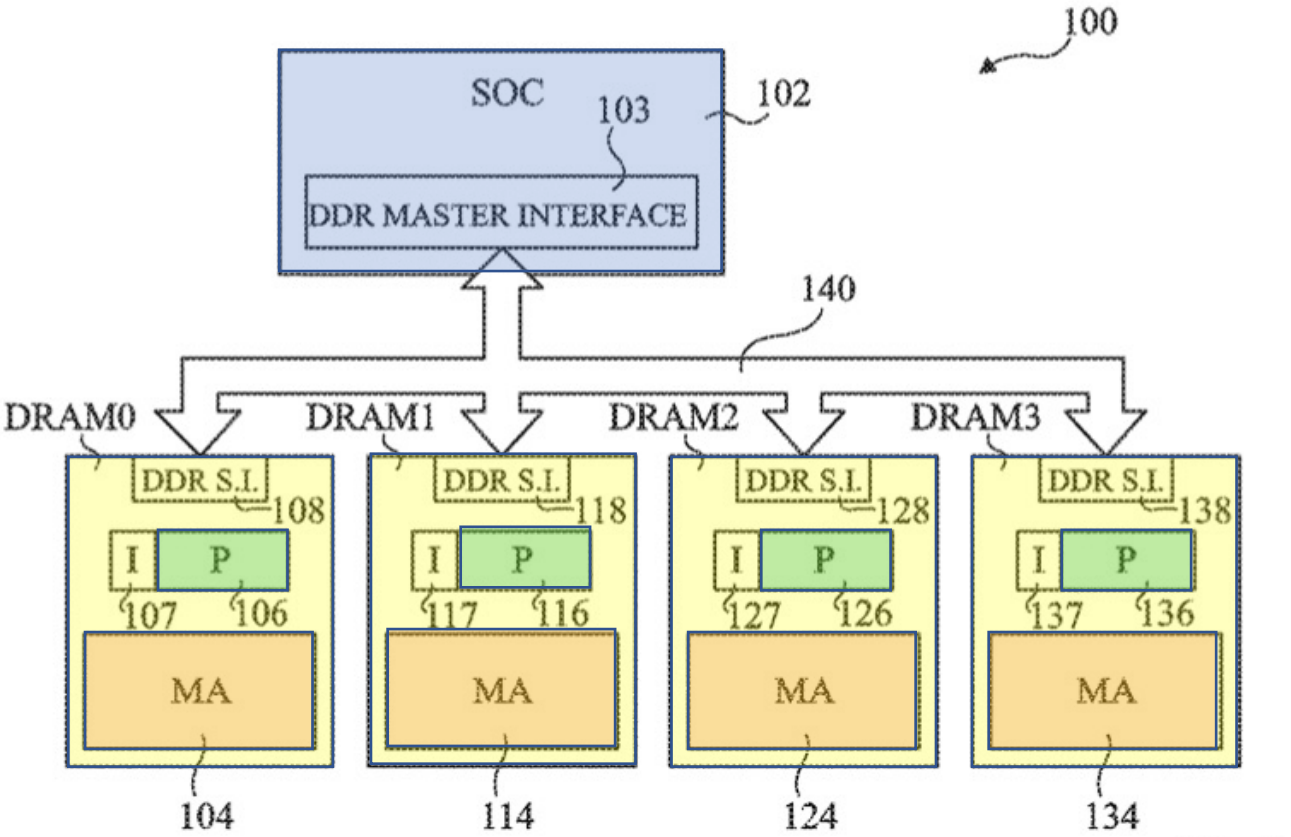


Fig 1

Fabrice Devaux, Jean-François Roy. “Memory circuit with integrated processor.” US 10,324,870 B2.

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

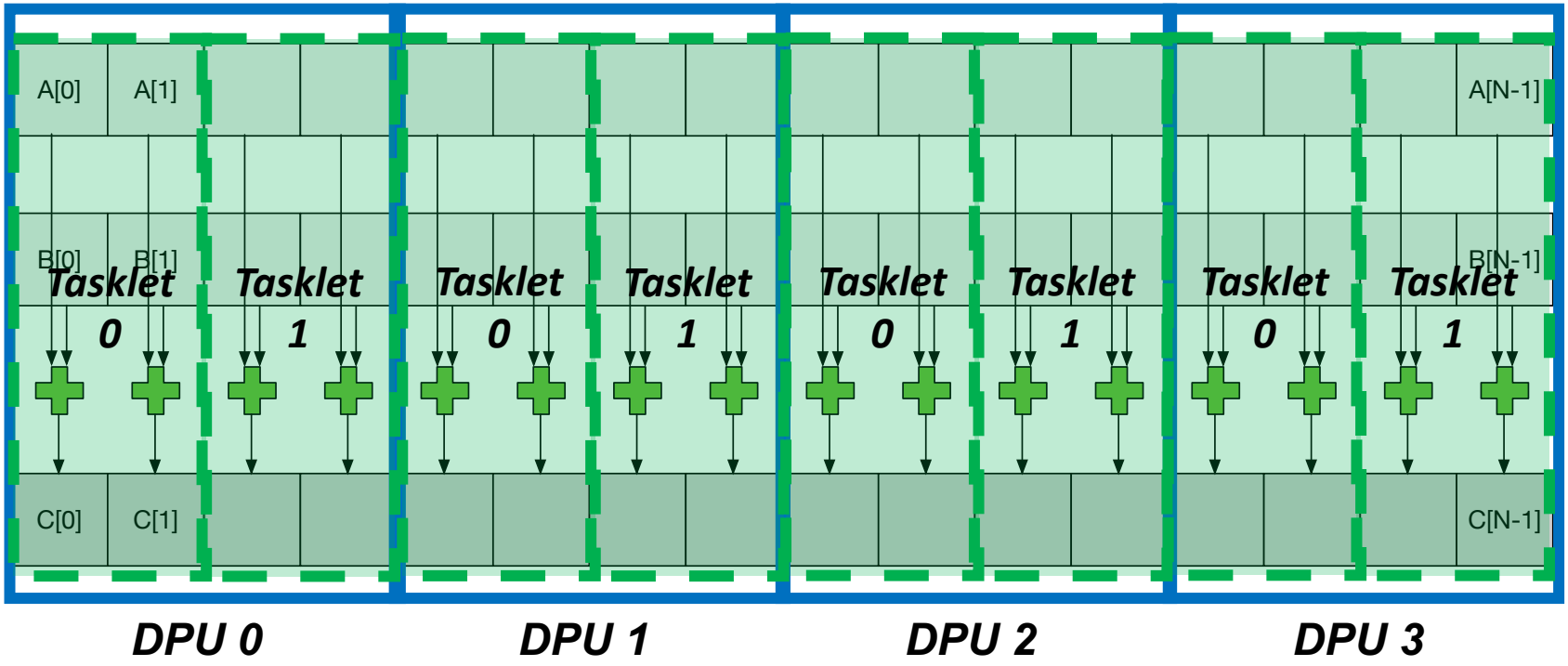
Larger CPU-DPU and DPU-CPU transfers between the host main memory and the DRAM Processing Unit's Main memory (MRAM) banks **result in higher sustained bandwidth**.

KEY TAKEAWAY 1

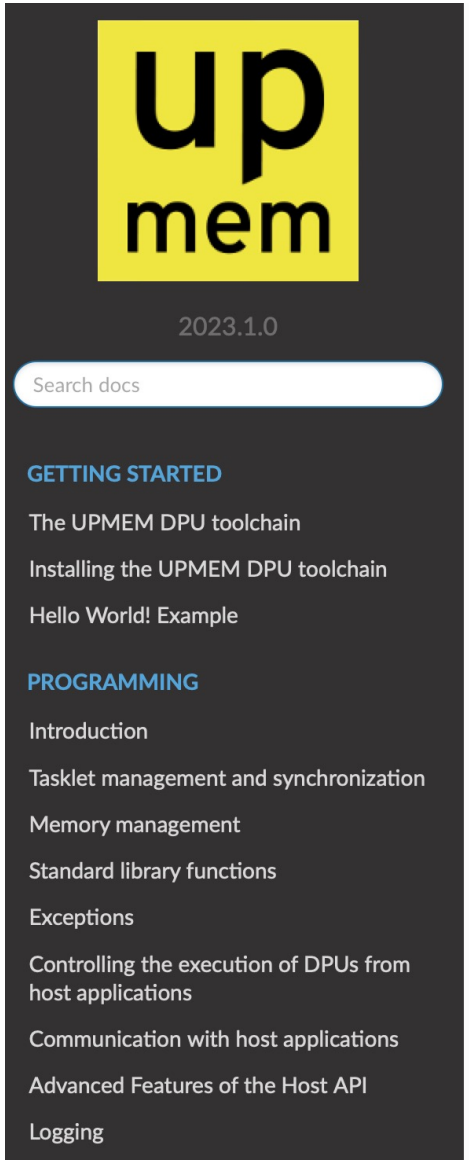
The UPMEM PIM architecture is fundamentally compute bound. As a result, **the most suitable work- loads are memory-bound.**

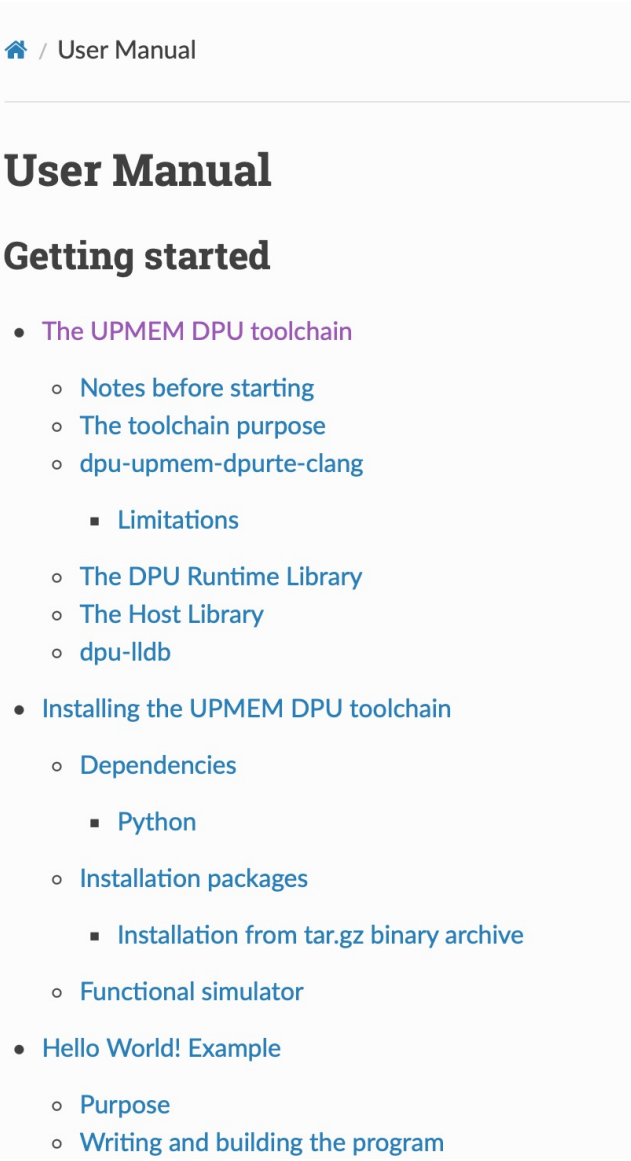
Vector Addition (VA)

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



UPMEM SDK Documentation





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.

* https://sdk.upmem.com/2021.1.1/index.html
★ 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`

```
1   struct dpu_set_t dpu_set, dpu;
2   uint32_t nr_of_dpus;
3
4   // Allocate DPUs
5   DPU_ASSERT(dpu_alloc(NR_DPUS, NULL, &dpu_set));
6
7   DPU_ASSERT(dpu_get_nr_dpus(dpu_set, &nr_of_dpus));
8   printf("Allocated %d DPU(s)\n", nr_of_dpus);
9
```

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

```
1   // Top-left computation on DPUs
2   for (unsigned int blk = 1; blk <= (max_cols-1)/BL; blk++) {
3
4       // If nr_of_blocks are lower than max_dpus,
5       // set nr_of_dpus to be equal with nr_of_blocks
6       unsigned nr_of_blocks = blk;
7       if (nr_of_blocks < max_dpus) {
8           DPU_ASSERT(dpu_free(dpu_set));
9           DPU_ASSERT(dpu_alloc(nr_of_blocks, NULL, &dpu_set));
10          DPU_ASSERT(dpu_load(dpu_set, DPU_BINARY, NULL));
11          DPU_ASSERT(dpu_get_nr_dpus(dpu_set, &nr_of_dpus));
12      } else if (nr_of_dpus == max_dpus) {
13          ;
14      } else {
15          DPU_ASSERT(dpu_free(dpu_set));
16          DPU_ASSERT(dpu_alloc(max_dpus, NULL, &dpu_set));
17          DPU_ASSERT(dpu_load(dpu_set, DPU_BINARY, NULL));
18          DPU_ASSERT(dpu_get_nr_dpus(dpu_set, &nr_of_dpus));
19      }
20
21      ...
22  }
```

Load DPU Binary

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

```
1  // Define the DPU Binary path as DPU_BINARY here
2  #ifndef DPU_BINARY
3  #define DPU_BINARY "./bin/dpu_code"
4  #endif
5
6      ...
7
8      // Load binary
9      DPU_ASSERT(dpu_load(dpu_set, DPU_BINARY, NULL));
10
```

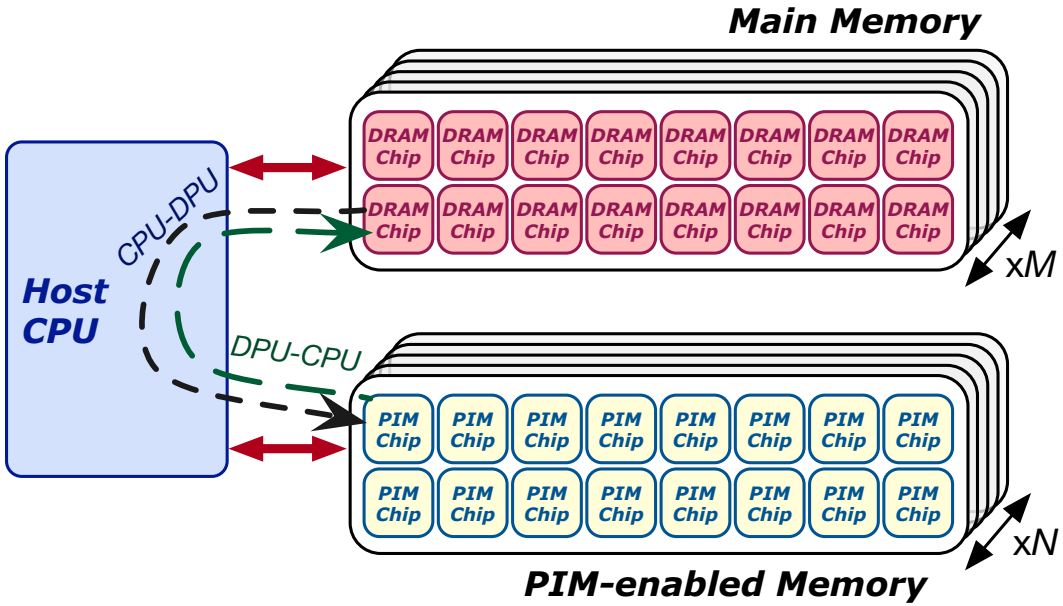
Is it possible to launch different kernels onto different DPUs?

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

```
DPU_FOREACH (dpu_set, dpu) {
    DPU_ASSERT(dpu_copy_to(dpu, DPU_MRAM_HEAP_POINTER_NAME, 0,                                  bufferA + input_size_dpu_8bytes * i, input_size_dpu_8bytes * sizeof(T)));
    DPU_ASSERT(dpu_copy_to(dpu, DPU_MRAM_HEAP_POINTER_NAME, input_size_dpu_8bytes * sizeof(T), bufferB + input_size_dpu_8bytes * i, input_size_dpu_8bytes * sizeof(T)));
    i++;
}
```

Offset within MRAM | Pointer to main memory | Transfer size

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`

```
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));
```

Pointer to main memory

Offset within MRAM

Transfer size

Direction

Broadcast Transfers

- `dpu_broadcast_to();`
  - Only CPU to DPU
- We transfer the same buffer to all DPUs in the `dpu_set`

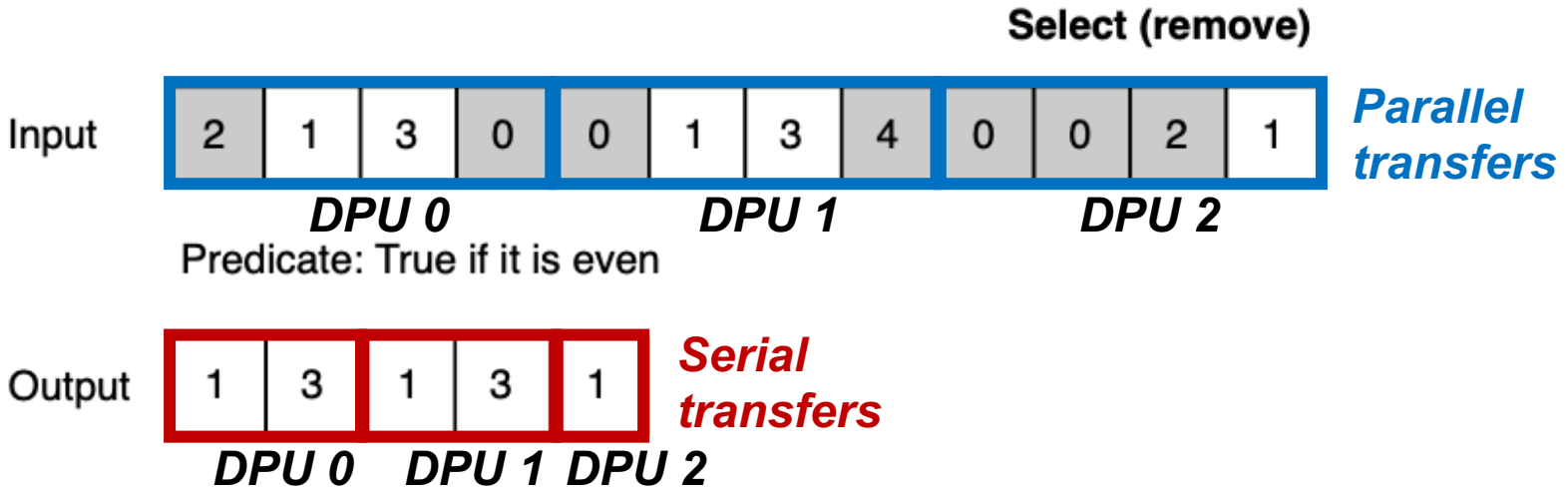
```
1  DPU_ASSERT(dpu_broadcast_to(dpu_set, DPU_MRAM_HEAP_POINTER_NAME, 0, bufferA, input_size_dpu * sizeof(T), DPU_XFER_DEFAULT));
2
```

Pointer to main memory

Transfer size

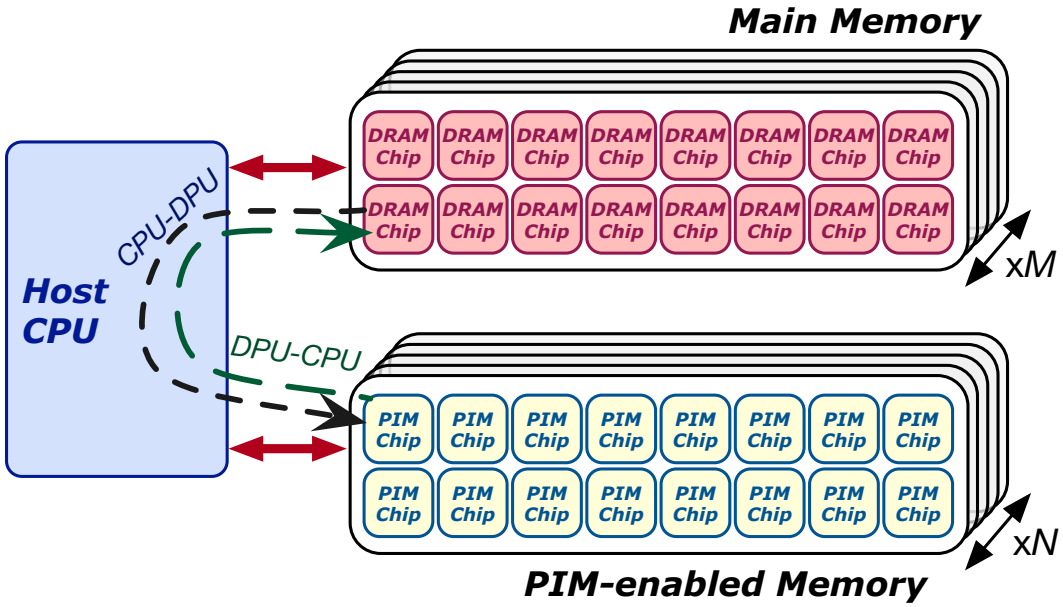
Different Types of Transfers in a Program

- An example benchmark that uses both parallel and serial transfers
- Select (SEL)
  - Remove even values



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?

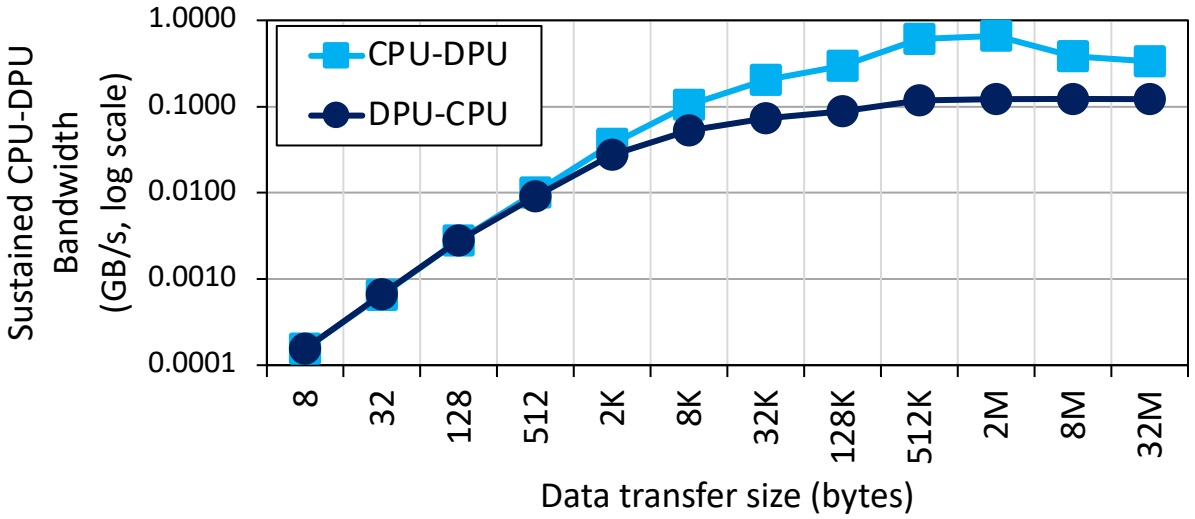
- With a microbenchmark, we obtain the sustained bandwidth of all types of CPU-DPU and DPU-CPU transfers
- Two experiments:
  - 1 DPU: variable CPU-DPU and DPU-CPU transfer size (8 bytes to 32 MB)
  - 1 rank: 32 MB CPU-DPU and DPU-CPU transfers to/from a set of 1 to 64 MRAM banks within the same rank
- 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

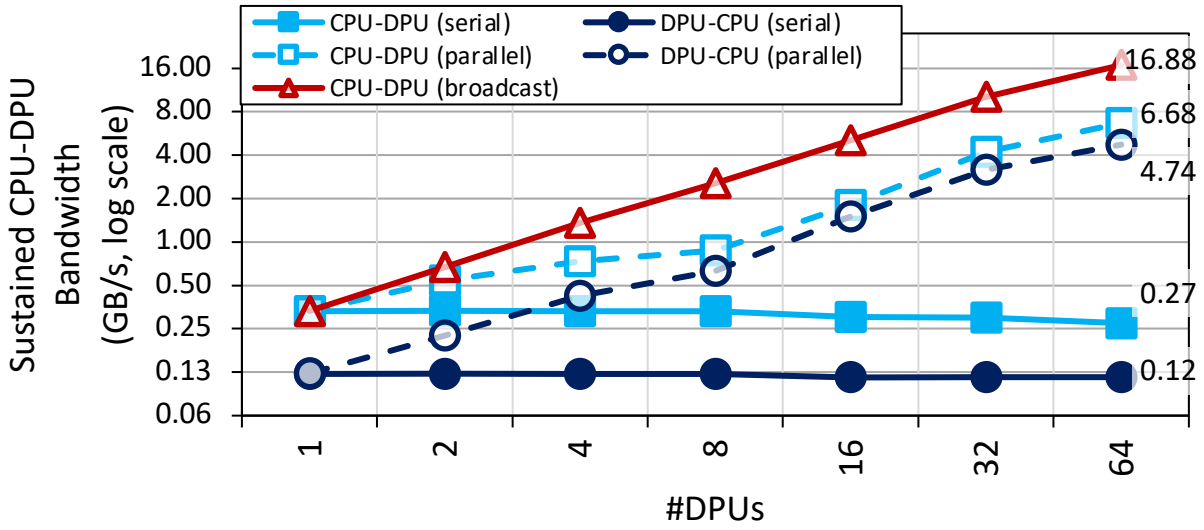


KEY OBSERVATION 7

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

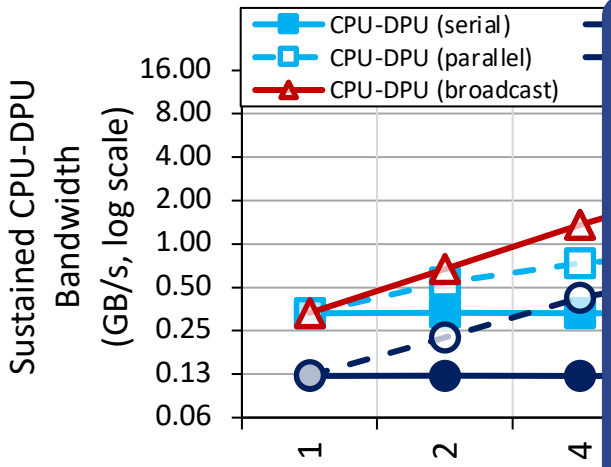


KEY OBSERVATION 8

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.**

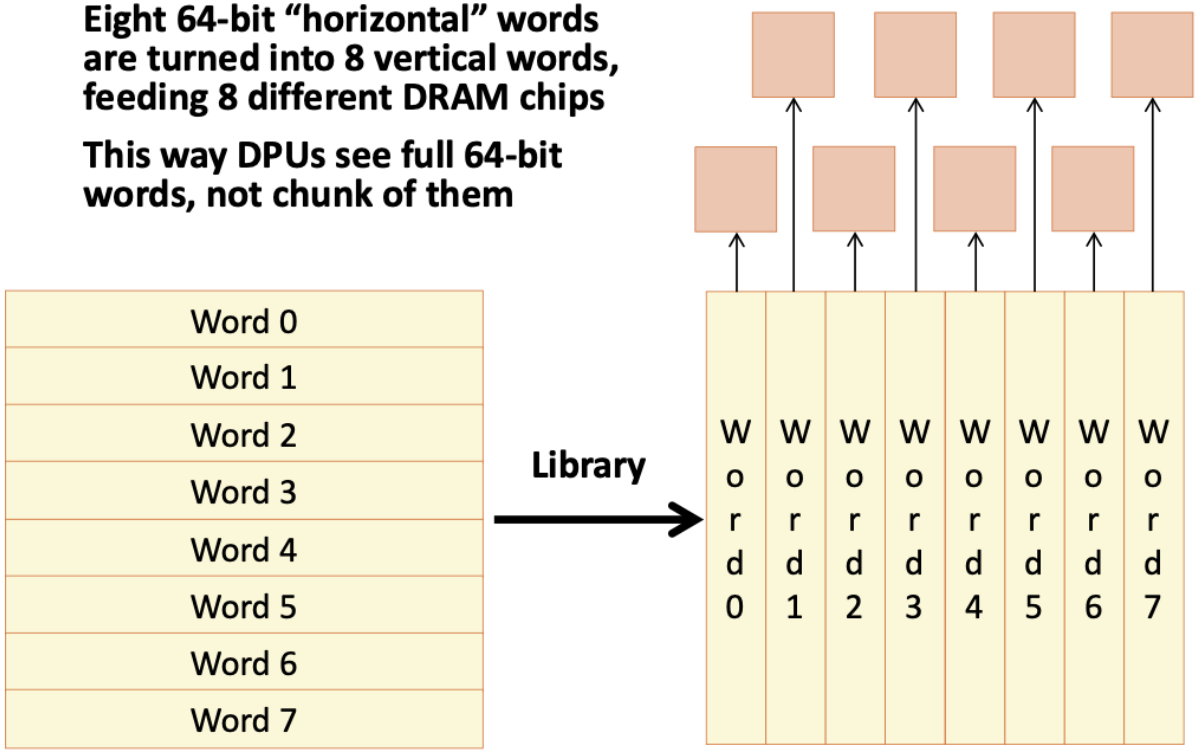
The sustained bandwidth of broadcast CPU-DPU transfers (i.e., the same buffer is copied to multiple MRAM banks) **is higher than that of parallel CPU-DPU transfers** (i.e., different buffers are copied to different MRAM banks) **due to higher temporal locality** in the CPU cache hierarchy.

“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



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.

F. Devaux, "The true Processing In Memory accelerator," HotChips 2019. doi: 10.1109/HOTCHIPS.2019.8875680

Microbenchmark: CPU-DPU

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

CMU-SAFARI / **prim-benchmarks**

Unwatch 2 | Star 1 | Fork 0

Code | Issues | Pull requests | Actions | Projects | Wiki | Security | Insights | Settings

main | prim-benchmarks / Microbenchmarks / **CPU-DPU** / | Go to file | Add file | ...

Juan Gomez Luna PrIM -- first commit 3de4b49 7 days ago **History**

..		
dpu	PrIM -- first commit	7 days ago
host	PrIM -- first commit	7 days ago
support	PrIM -- first commit	7 days ago
Makefile	PrIM -- first commit	7 days ago
run.sh	PrIM -- first commit	7 days ago

DPU Kernel Launch

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

```
1   printf("Run program on DPU(s) \n");
2   // Run DPU kernel
3   DPU_ASSERT(dpu_launch(dpu_set, DPU_SYNCHRONOUS));
4
```

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?

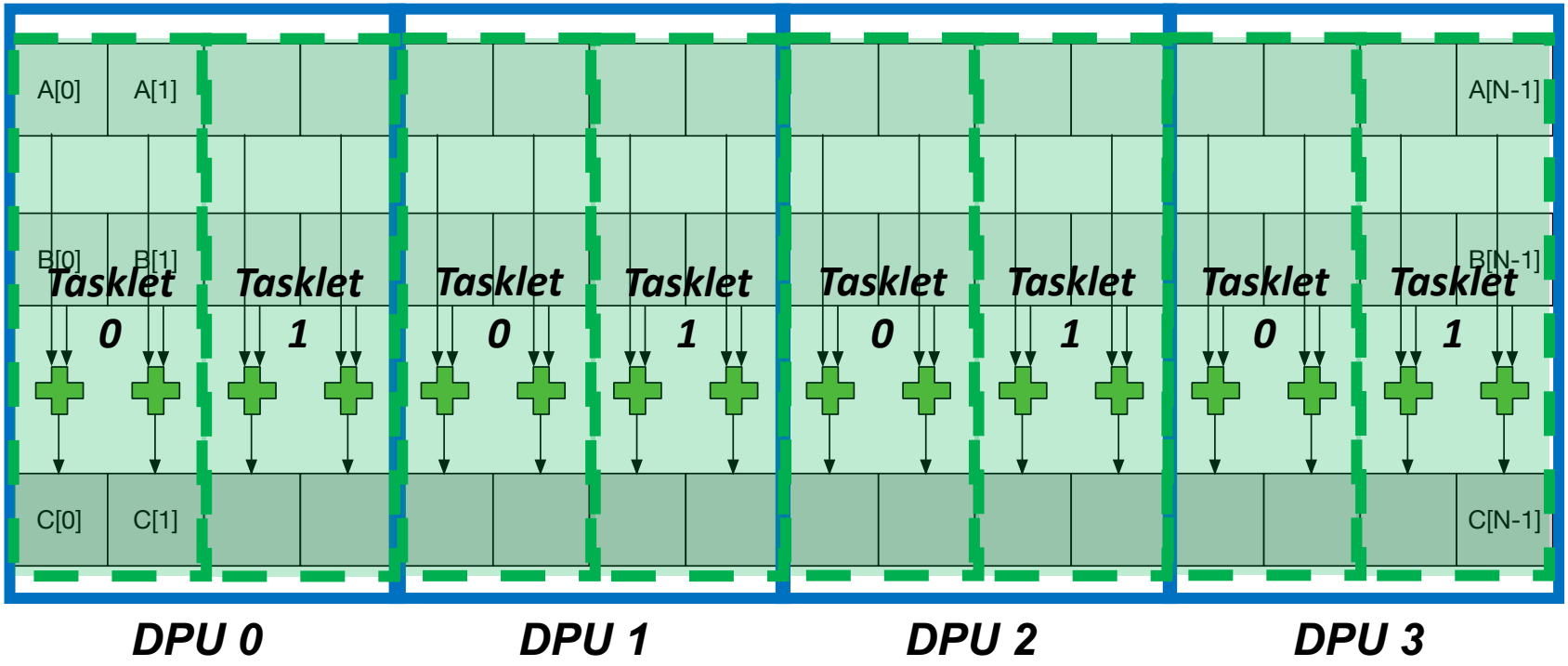
- We can use serial and parallel transfers
- We pass them directly to the scratchpad memory of the DPU
  - Working RAM (WRAM): 64KB per DPU
- This is useful for input parameters and some results

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

```
// 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; // Input size per DPU in bytes
    uint32_t input_size_dpu_bytes_transfer = DPU_INPUT_ARGUMENTS.transfer_size; // Transfer input size per DPU in bytes

    // 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;
}
```

Tasklet ID

Size of vector tile processed by a DPU

MRAM addresses of arrays A and B

WRAM allocation

MRAM-WRAM DMA transfers

Vector addition (see next slide)

WRAM-MRAM DMA transfer

Programming a DPU Kernel (II)

- Vector addition

```
1  // vector_addition: Computes the vector addition of a cached block
2  static void vector_addition(T *bufferB, T *bufferA, unsigned int l_size) {
3
4      for (unsigned int i = 0; i < l_size; i++){
5          bufferB[i] += bufferA[i];
6      }
7
8  }
```

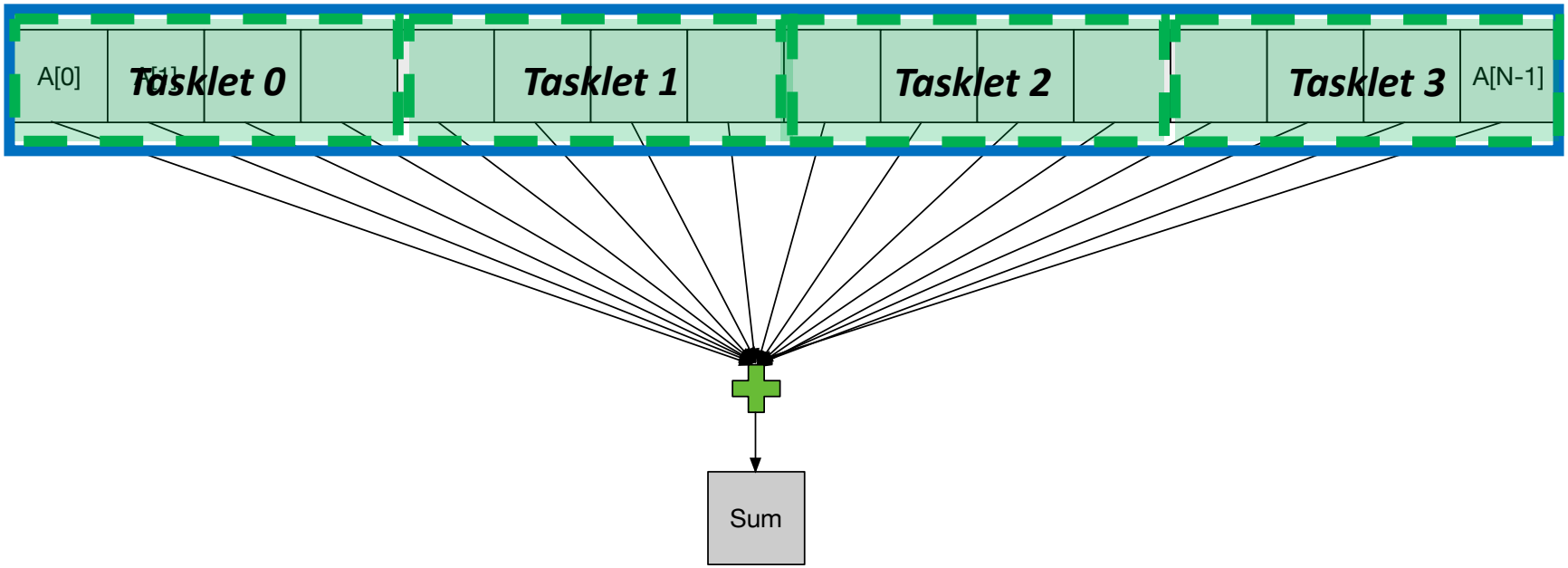
Intra-DPU Synchronization

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();`

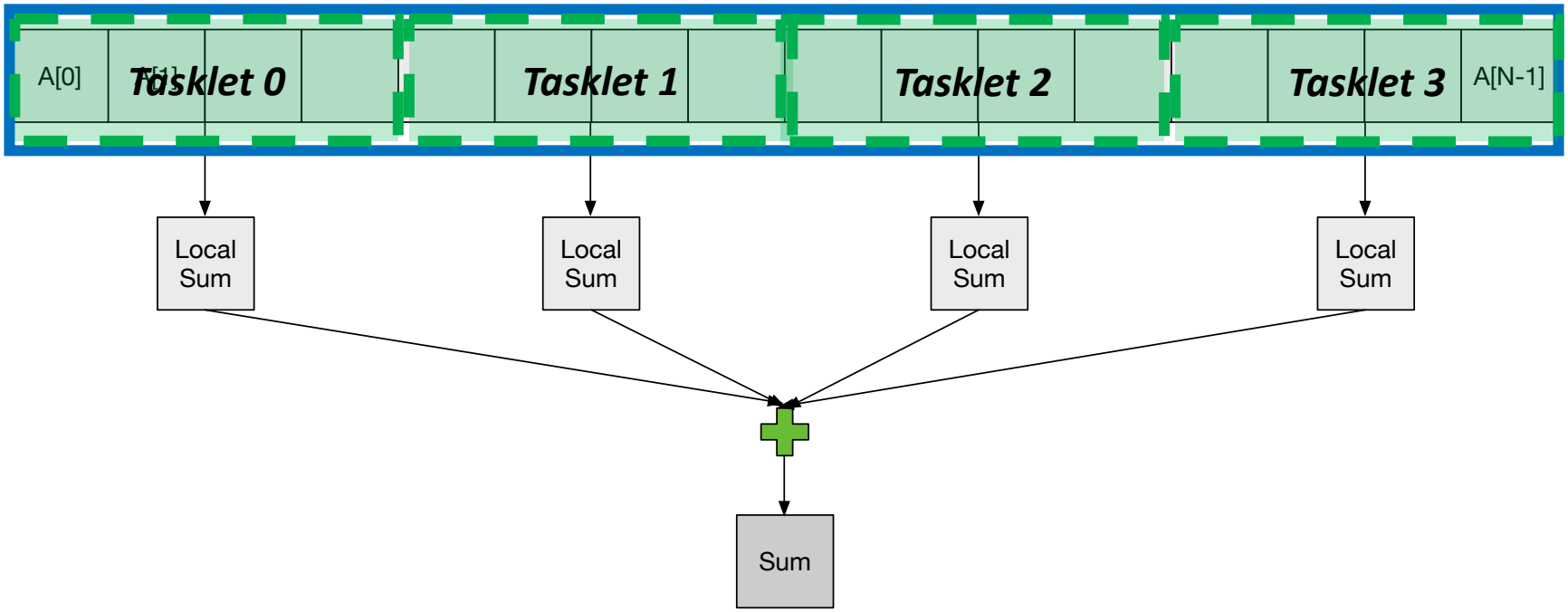
Parallel Reduction (I)

- 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

```
1  for(unsigned int byte_index = base_tasklet; byte_index < input_size_dpu_bytes; byte_index += BLOCK_SIZE * NR_TASKLETS){
2
3      // Bound checking
4      uint32_t l_size_bytes = (byte_index + BLOCK_SIZE >= input_size_dpu_bytes) ? (input_size_dpu_bytes - byte_index) : BLOCK_SIZE;
5
6      // Load cache with current MRAM block
7      mram_read((__mram_ptr void const*)(mram_base_addr_A + byte_index), cache_A, l_size_bytes);
8
9      // Reduction in each tasklet
10     l_count += reduction(cache_A, l_size_bytes >> DIV);
11
12 }
13 // Copy local count to shared array in WRAM
14 message[tasklet_id] = l_count;
```

Accumulate in a local sum

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

Accumulate in a local sum

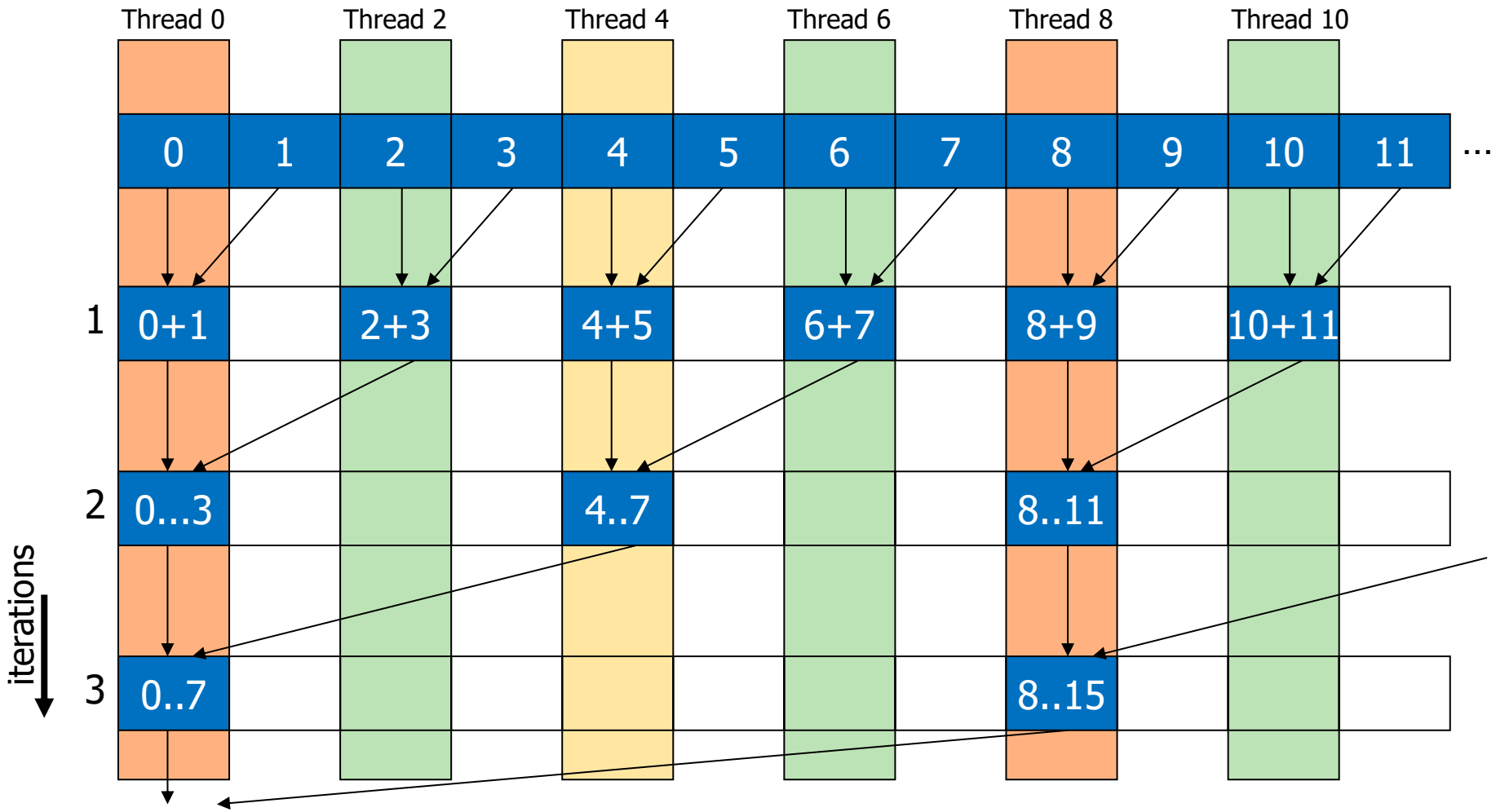
Copy local sum into WRAM

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

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

    // Total count in this DPU
    result->t_count = message[0];
}
```

Barrier synchronization

Sequential accumulation

Vector Reduction: Naïve Mapping



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

```
1   // Barrier
2   barrier_wait(&my_barrier);
3
4   #pragma unroll
5   for (unsigned int offset = 1; offset < NR_TASKLETS; offset <<= 1){
6
7       if((tasklet_id & (2*offset - 1)) == 0){
8           message[tasklet_id] += message[tasklet_id + offset];
9       }
10
11      // Barrier
12      barrier_wait(&my_barrier);
13  }
```

"offset" tasklets working

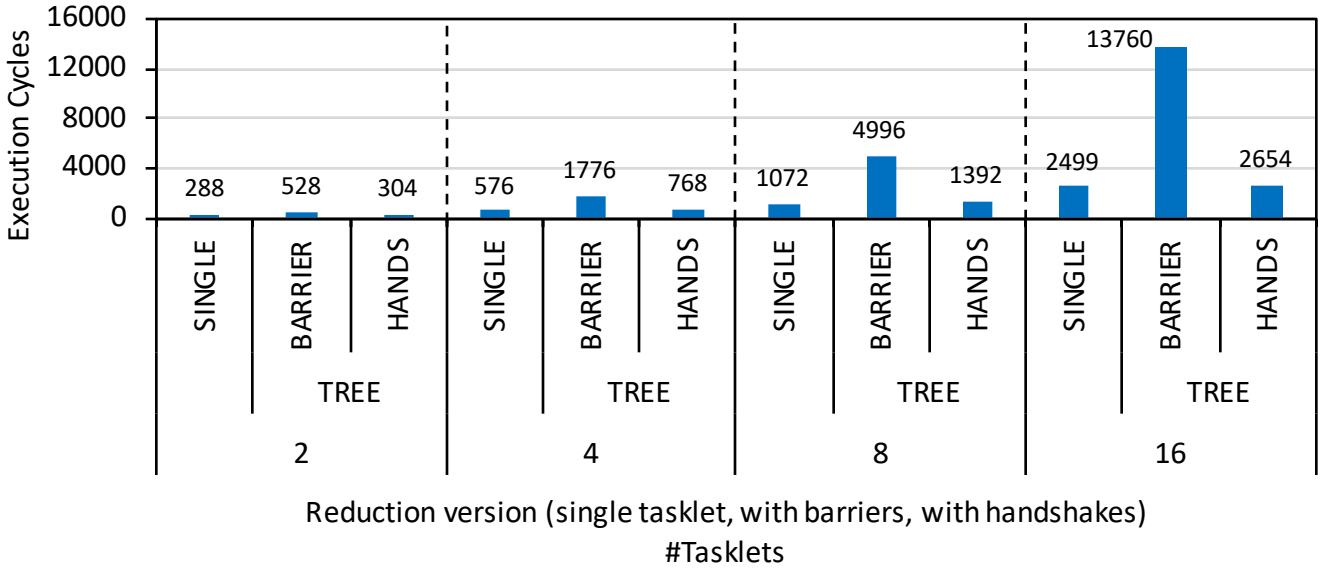
Barrier synchronization

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," https://arxiv.org/pdf/2105.03814.pdf

Tree-Based Reduction on UPMEM PIM (I)

- Single-thread vs. Barrier-based vs. Handshake-based on 1 DPU

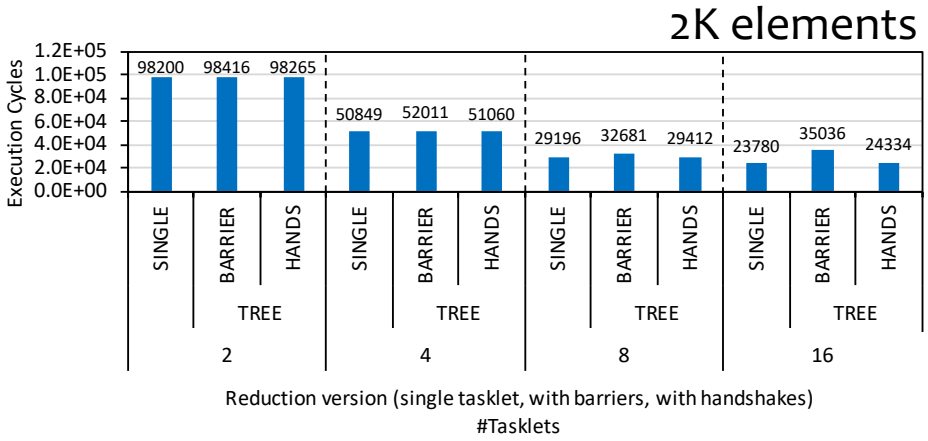


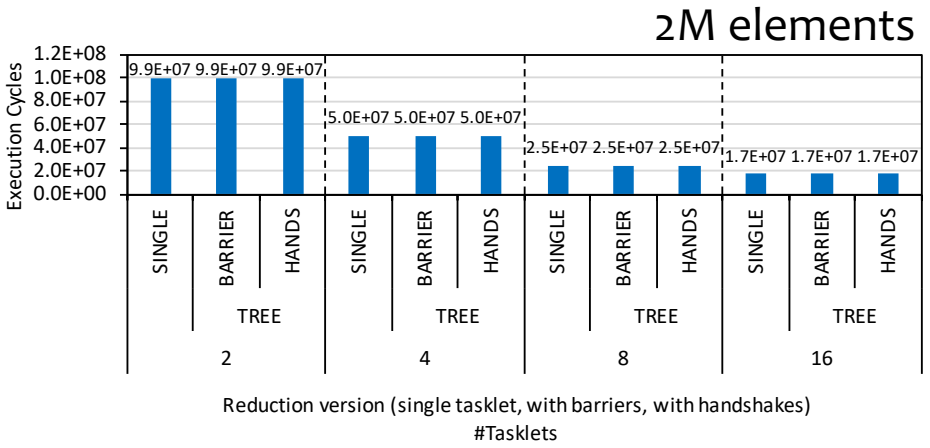
High cost of intra-DPU synchronization
(especially, barrier synchronization)
when there is small amount of computation

Gómez-Luna et al. "Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture." *arXiv preprint arXiv:2105.03814* (2021). https://arxiv.org/pdf/2105.03814.pdf

Tree-Based Reduction on UPMEM PIM (II)

- Single-thread vs. Barrier-based vs. Handshake-based on 1 DPU

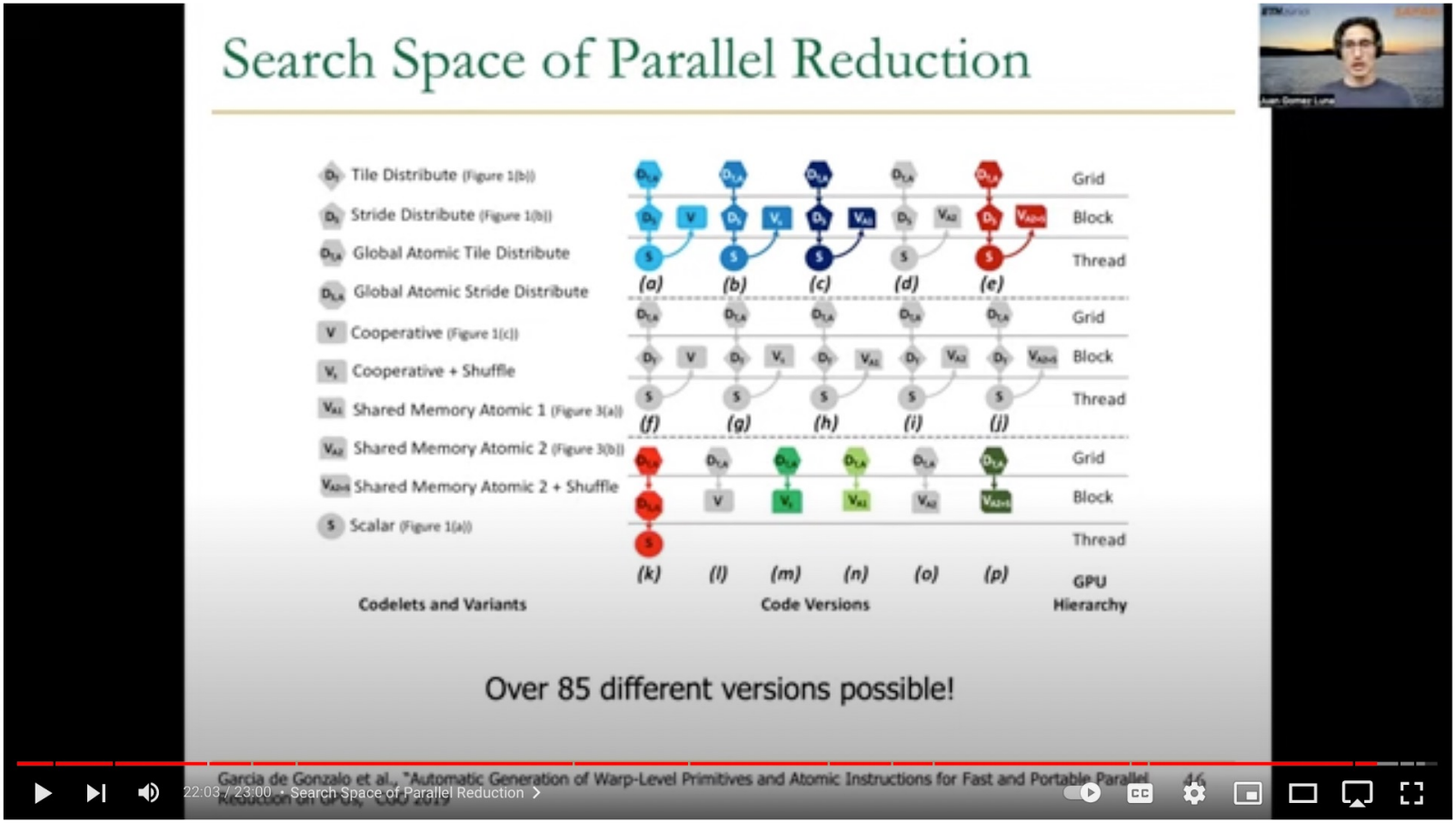




Cost of intra-DPU synchronization
gets amortized when there is large amount of computation

Gómez-Luna et al. "Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture." *arXiv preprint arXiv:2105.03814* (2021). https://arxiv.org/pdf/2105.03814.pdf

Parallel Reduction on GPU



HetSys Course: Lecture 6: Parallel Patterns: Reduction (Spring 2023)

Onur Mutlu Lectures
32.6K subscribers

Subscribed | 4 | Share | Clip | Save

197 views 2 weeks ago Livestream - Programming Heterogeneous Computing Systems with GPUs and other Accelerators (Spring 2023)
Project & Seminar, ETH Zürich, Spring 2023
Programming Heterogeneous Computing Systems with GPUs and other Accelerators (https://safari.ethz.ch/projects_and_s...)

https://youtu.be/tJmcV9AMfp4

Prefix-Sum (Scan)

Input

1	2	3	4	1	1	1	1	0	1	2	3	2	2	2	2

Output (Exclusive Scan)

```
out[0] = 0; // Identity value
for(int i=1; i<n; i++)
    out[i] = out[i-1] + in[i-1];
```

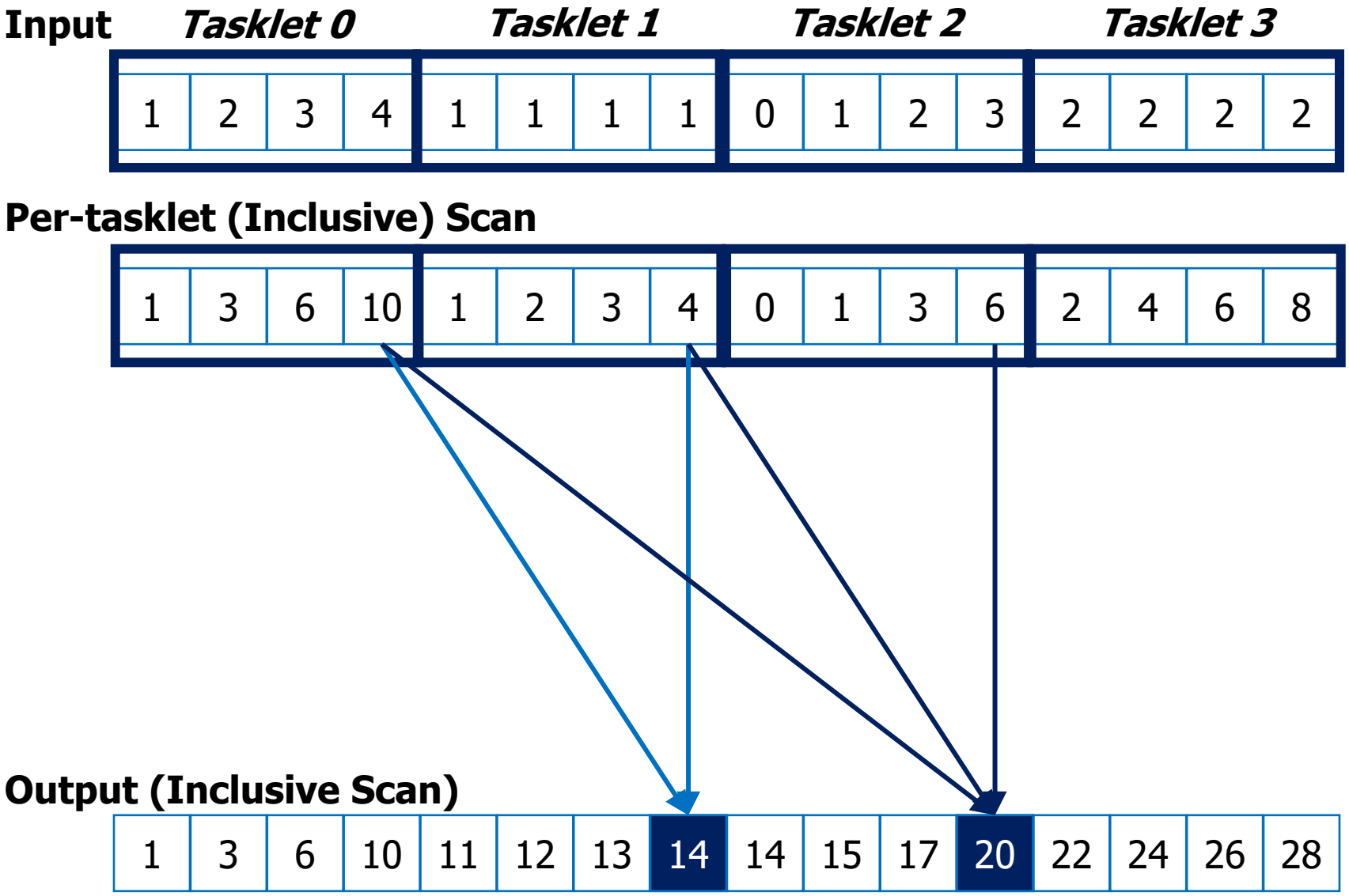
0	1	3	6	10	11	12	13	14	14	15	17	20	22	24	26

Output (Inclusive Scan)

```
out[0] = in[0];
for(int i=1; i<n; i++)
    out[i] = out[i-1] + in[i];
```

1	3	6	10	11	12	13	14	14	15	17	20	22	24	26	28

Hierarchical (Inclusive) Scan: 1 DPU



Per-DPU (Inclusive) Scan (I)

- Each tasklet computes scan locally

```
1  // Load cache with current MRAM block
2  mram_read((const __mram_ptr void*)(mram_base_addr_A + byte_index), cache_A, BLOCK_SIZE);
3
4  // Scan in each tasklet
5  T l_count = scan(cache_B, cache_A);
6
7  // Sync with adjacent tasklets
8  T p_count = handshake_sync(l_count, tasklet_id);
9
10 // Add in each tasklet
11 add(cache_B, p_count);
12
13 // Write cache to current MRAM block
14 mram_write(cache_B, (__mram_ptr void*)(mram_base_addr_B + byte_index), BLOCK_SIZE);
```

Per-tasklet scan

```
17 // Scan in each tasklet
18 static T scan(T *output, T *input){
19     output[0] = input[0];
20     #pragma unroll
21     for(unsigned int j = 1; j < REGS; j++) {
22         output[j] = output[j - 1] + input[j];
23     }
24     return output[REGS - 1];
25 }
```

Per-DPU (Inclusive) Scan (II)

- Each tasklet communicates with adjacent tasklets

```
1  // Load cache with current MRAM block
2  mram_read((const __mram_ptr void*)(mram_base_addr_A + byte_index), cache_A, BLOCK_SIZE);
3
4  // Scan in each tasklet
5  T l_count = scan(cache_B, cache_A);
6
7  // Sync with adjacent tasklets
8  T p_count = handshake_sync(l_count, tasklet_id);
9
10 // Add in each tasklet
11 add(cache_B, p_count);
12
13 // Write cache to current MRAM bl
14 mram_write(cache_B, (__mram_ptr v
```

Per-tasklet scan

Handshake-based synchronization

```
28 // Handshake with adjacent tasklets
29 static T handshake_sync(T l_count, unsigned int tasklet_id){
30     T p_count;
31
32     // Wait and read message
33     if(tasklet_id != 0){
34         handshake_wait_for(tasklet_id - 1);
35         p_count = message[tasklet_id];
36     }
37     else
38         p_count = 0;
39
40     // Write message and notify
41     if(tasklet_id < NR_TASKLETS - 1){
42         message[tasklet_id + 1] = p_count + l_count;
43         handshake_notify();
44     }
45     return p_count;
46 }
```

Per-DPU (Inclusive) Scan (III)

- Each tasklet adds an offset to each own element

```
1  // Load cache with current MRAM block
2  mram_read((const __mram_ptr void*)(mram_base_addr_A + byte_index), cache_A, BLOCK_SIZE);
3
4  // Scan in each tasklet
5  T l_count = scan(cache_B, cache_A);
6
7  // Sync with adjacent tasklets
8  T p_count = handshake_sync(l_count, tasklet_id);
9
10 // Add in each tasklet
11 add(cache_B, p_count);
12
13 // Write cache to current MRAM block
14 mram_write(cache_B, (__mram_ptr void*)(mram_base_addr_B + byte_index), BLOCK_SIZE);
```

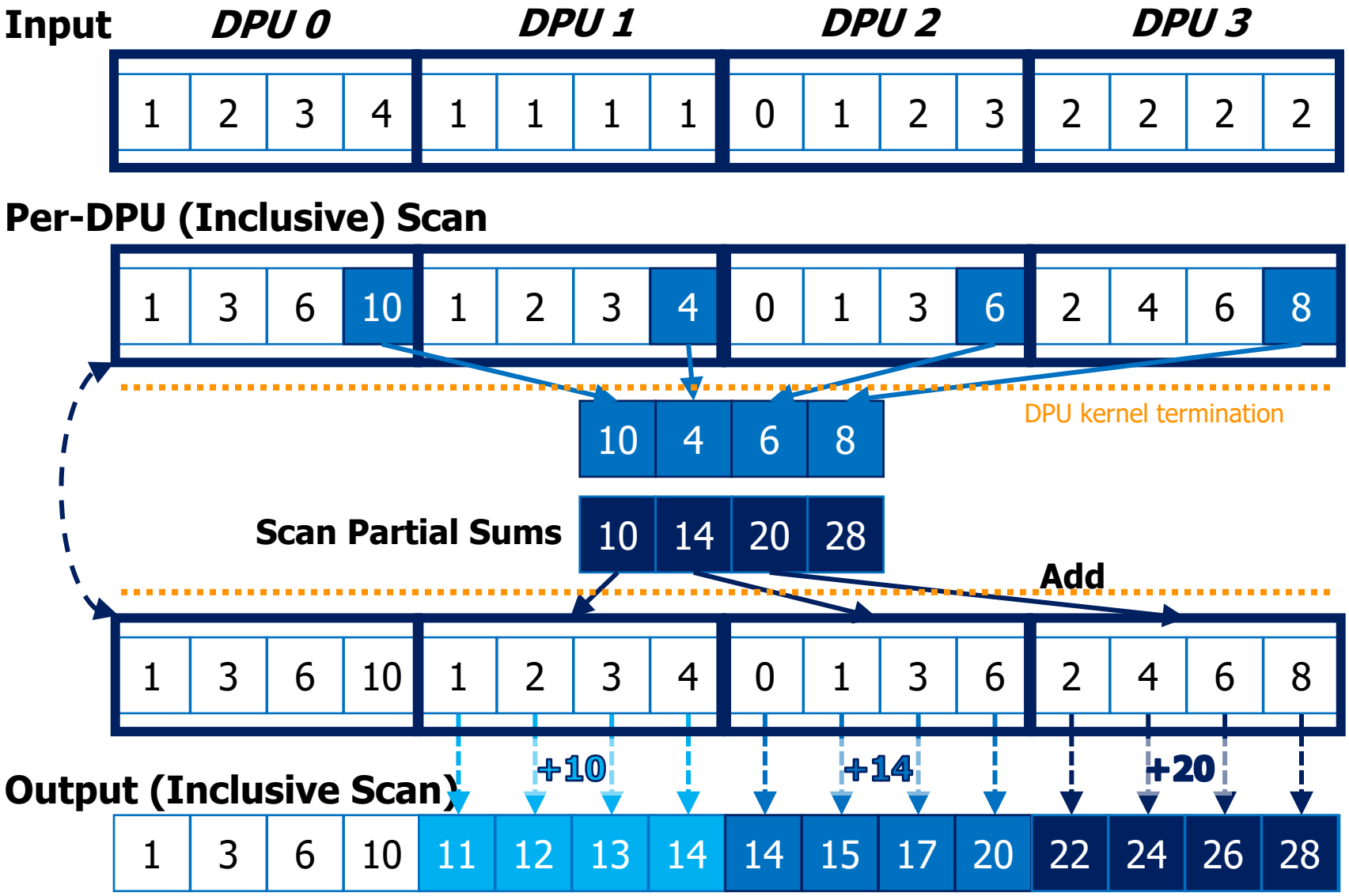
Per-tasklet scan

Handshake-based synchronization

Per-tasklet add

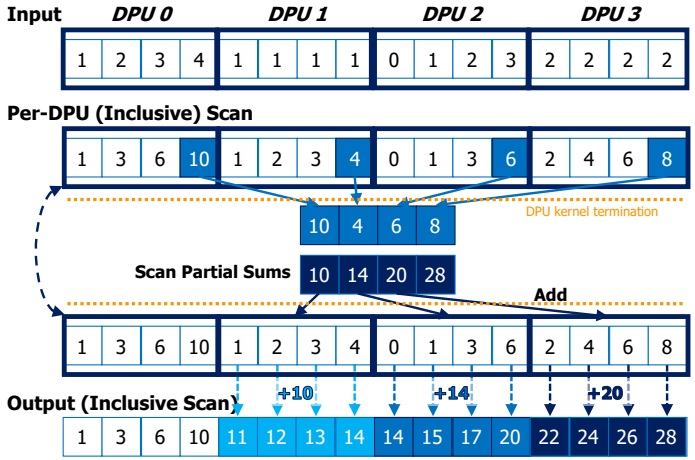
```
49 // Add in each tasklet
50 static void add(T *output, T p_count){
51     #pragma unroll
52     for(unsigned int j = 0; j < REGS; j++) {
53         output[j] += p_count;
54     }
55 }
```

Scan-Scan-Add (SSA)

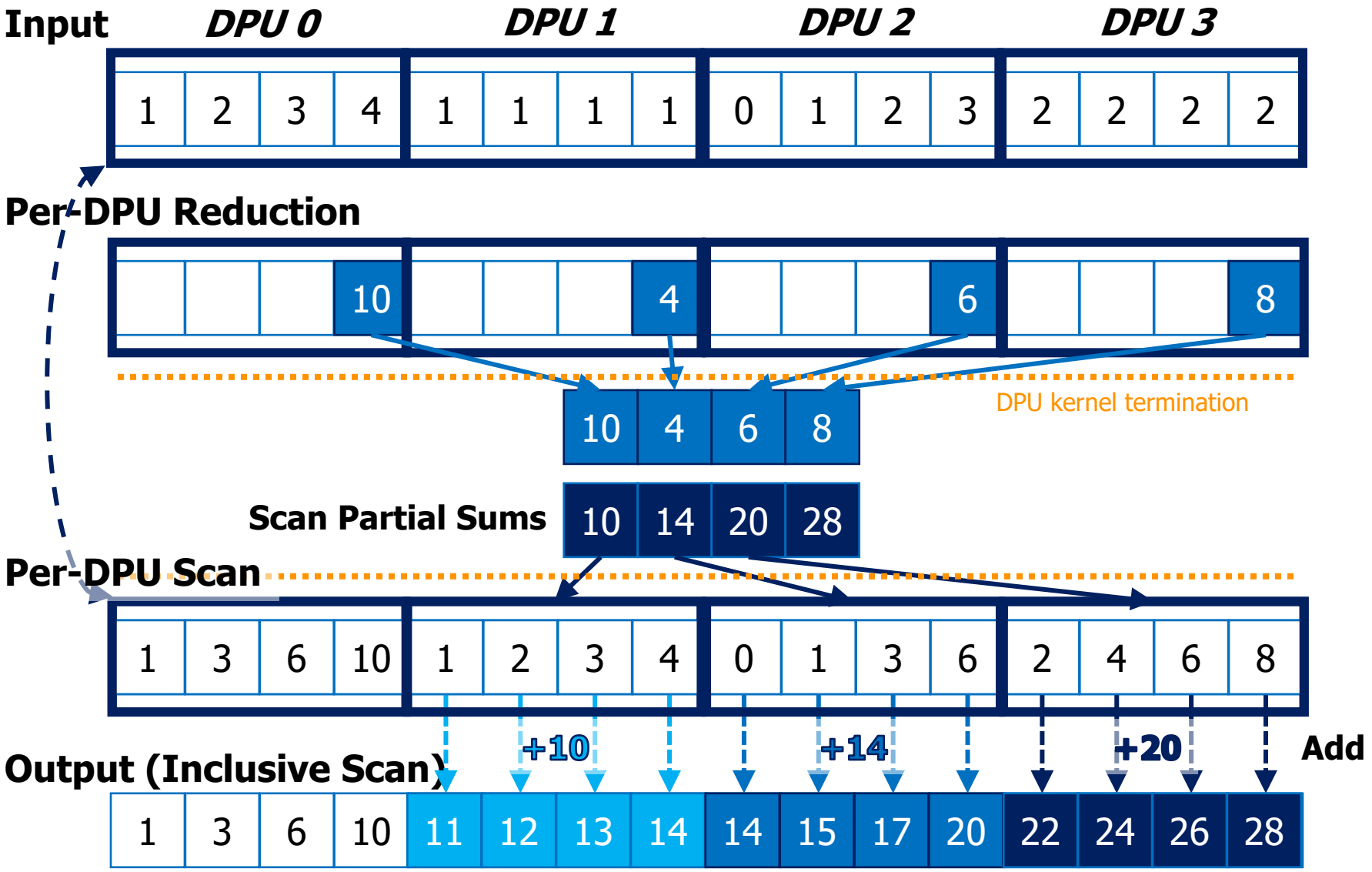


SSA: Memory Accesses

- Scan
  - First kernel reads input array (N elements) and writes array with per-DPU prefix sums (N elements)
- Scan
  - Second kernel reads and writes N / PER_DPU_SIZE elements
- Add
  - Third kernel reads array with per-DPU prefix sums (N elements) and writes output (N elements)
- 4N elements are read/written

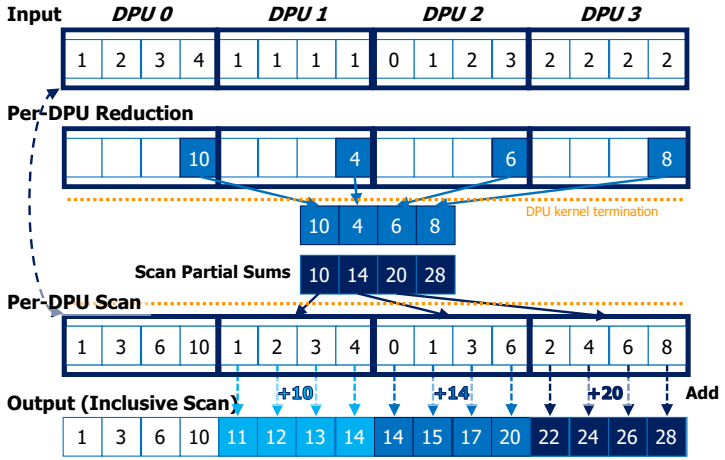


Reduce-Scan-Scan (RSS)



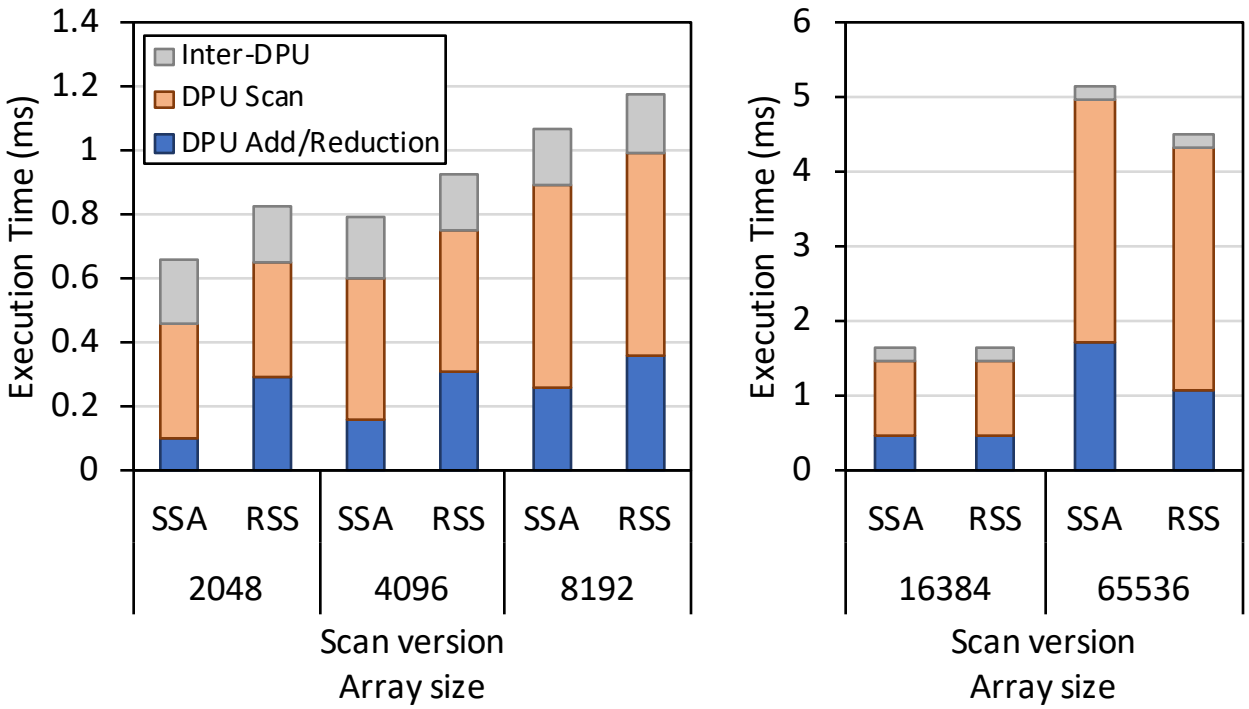
RSS: Memory Accesses

- Reduce
  - First kernel reads input array (N elements) and writes per-DPU reduction (N / PER_DPU_SIZE elements)
- Scan
  - Second kernel reads and writes N / PER_DPU_SIZE elements
- Scan
  - Third kernel reads input array (N elements) and scan partial sums (N / PER_DPU_SIZE elements), and writes output (N elements)
- 3N elements are read/written



SCAN-SSA vs. SCAN-RSS on UPMEM PIM

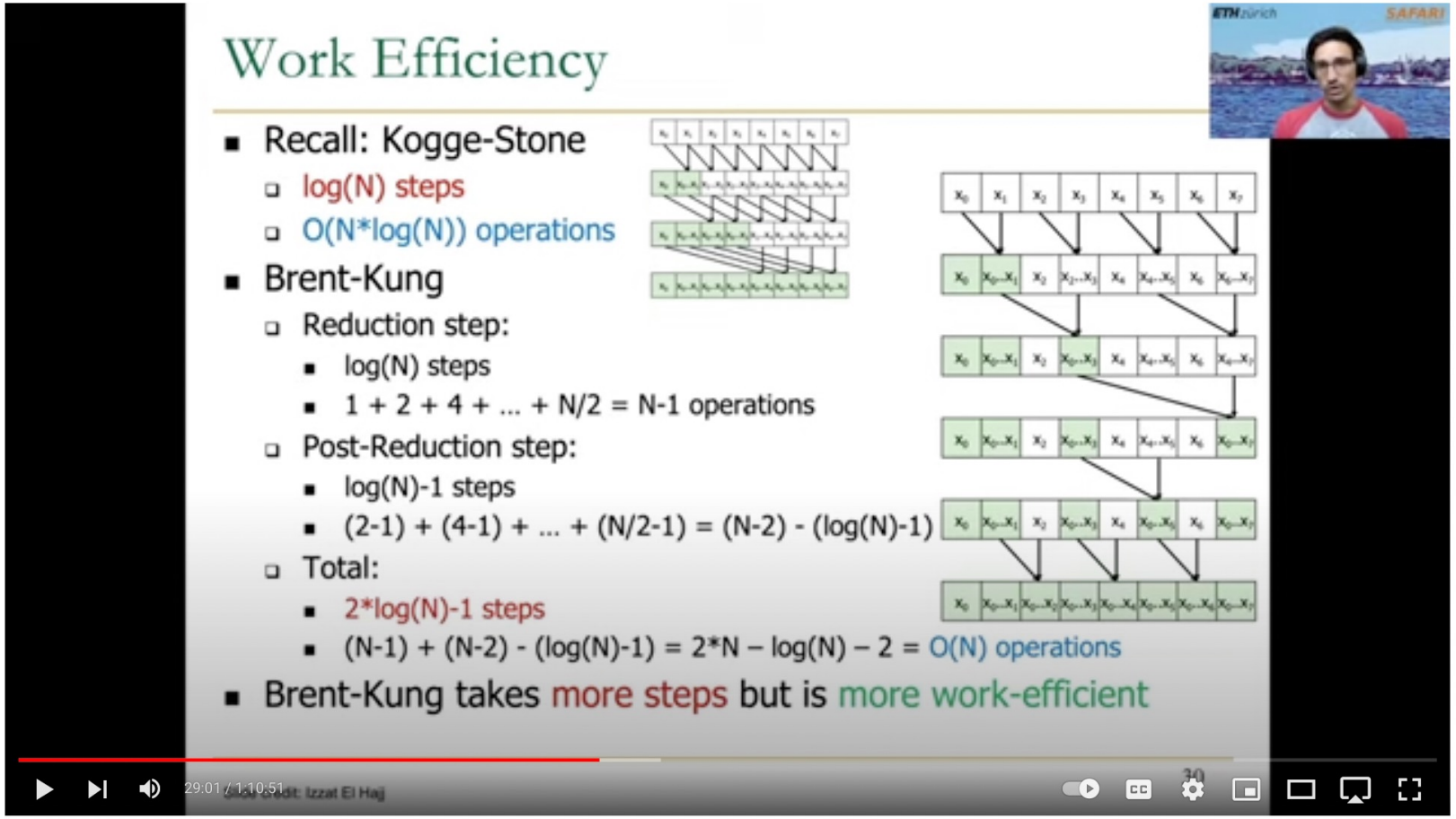
- SCAN-SSA vs. SCAN-RSS on 1 DPU



The cost of **intra-DPU synchronization** in RSS (in Reduce step) may be **noticeable for small arrays.**
For large arrays, RSS is faster than SSA, since it saves memory accesses

Gómez-Luna et al. "Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture." *arXiv preprint arXiv:2105.03814* (2021). https://arxiv.org/pdf/2105.03814.pdf

Parallel Prefix-Sum (Scan) on GPU

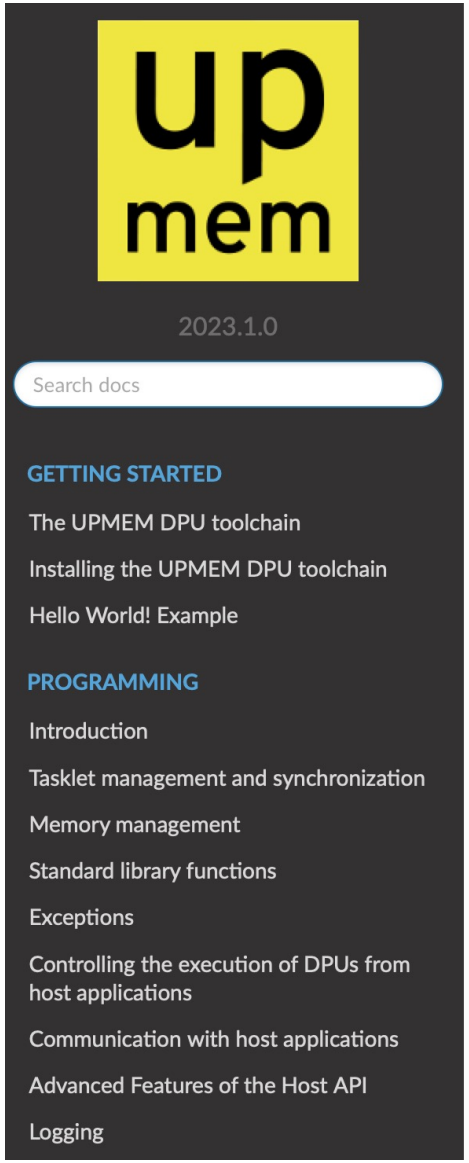


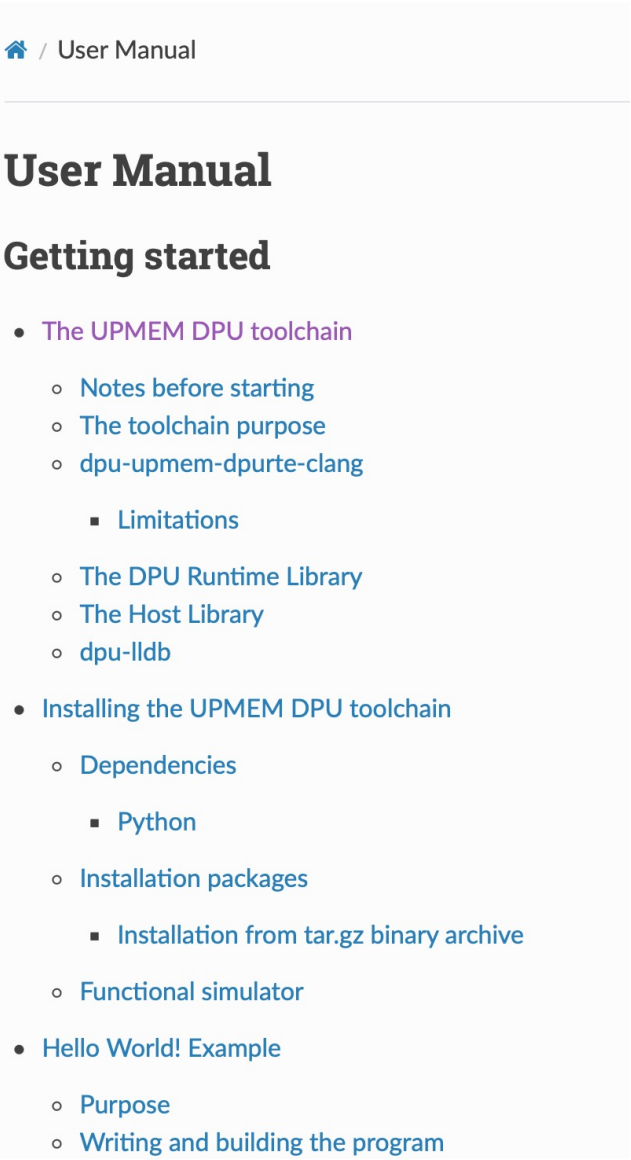
HetSys Course: Lecture 10: Parallel Patterns: Prefix Sum (Scan) (Fall 2022)

Onur Mutlu Lectures
32.6K subscribers
Subscribed
15
Share
Clip
Save

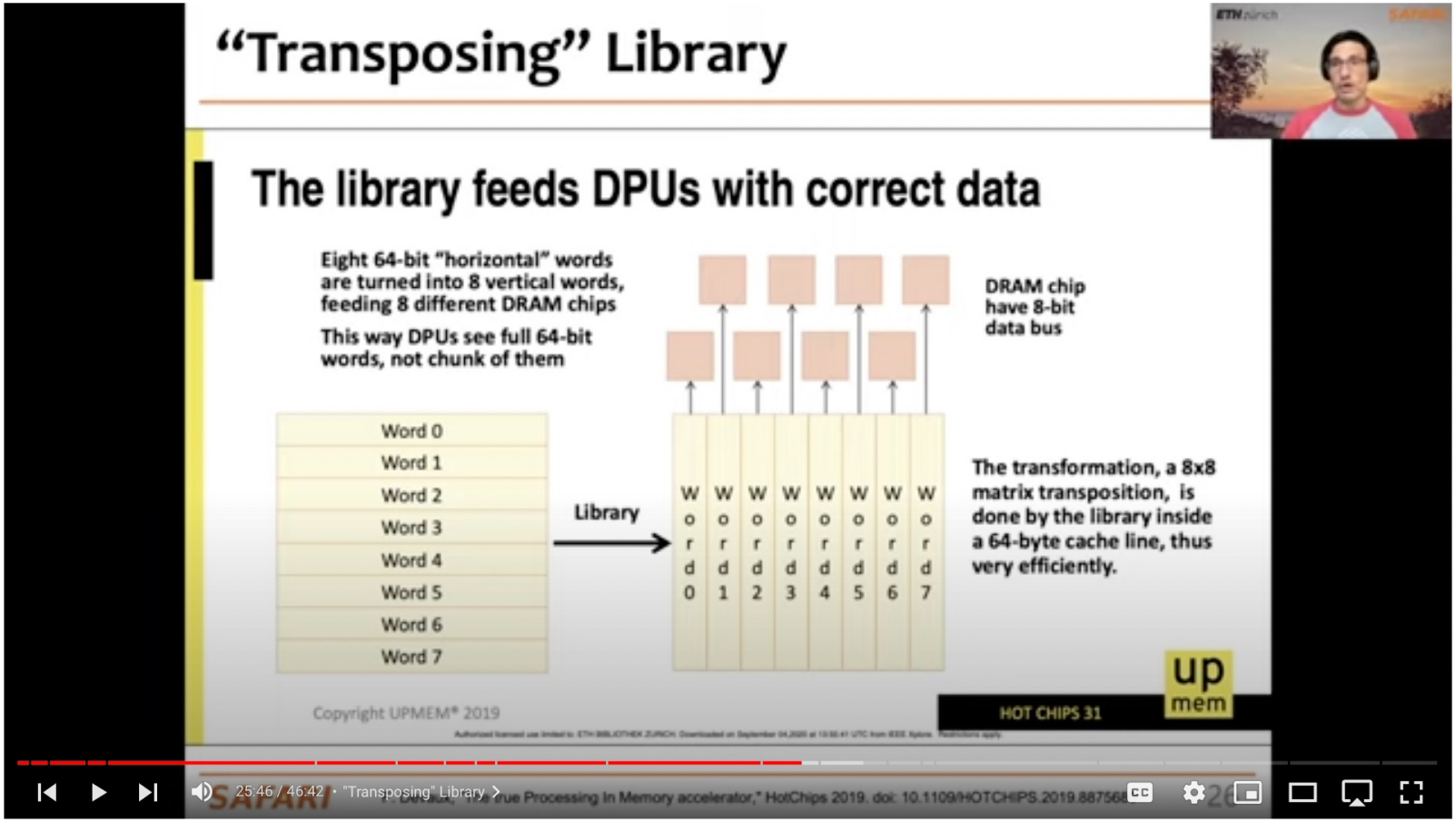
302 views 4 months ago Livestream - P&S Programming Heterogeneous Computing Systems with GPUs and other Accelerators (Fall 2022)
Project & Seminar, ETH Zürich, Fall 2022
Programming Heterogeneous Computing Systems with GPUs and other Accelerators (https://safari.ethz.ch/projects_and_s...)

UPMEM SDK Documentation





Programming UPMEM PIM (I)



PIM Course: Lecture 9: Programming PIM Architectures - Fall 2022

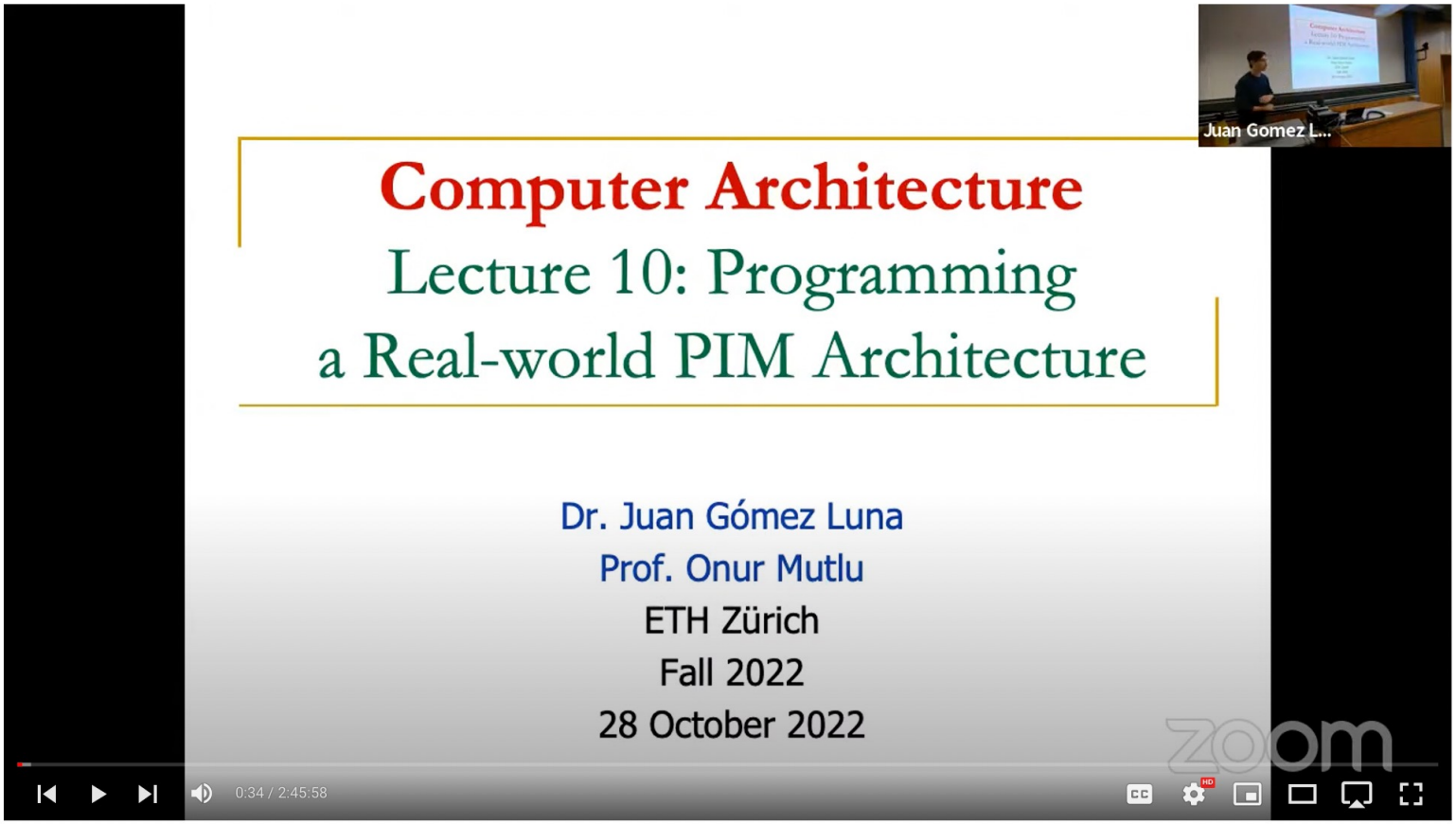
Onur Mutlu Lectures
32.6K subscribers

Subscribed | 10 | Share | Clip | Save

424 views 4 months ago Livestream - P&S Data-Centric Architectures: Fundamentally Improving Performance and Energy (Fall 2022)
Projects & Seminars, ETH Zürich, Fall 2022
Data-Centric Architectures: Fundamentally Improving Performance and Energy

https://youtu.be/zF70xuhesME

Programming UPMEM PIM (II)



Livestream - Computer Architecture - ETH Zürich (Fall 2022)

Computer Architecture - Lecture 10: Real Processing in Memory Systems: UPMEM Case Study (Fall 2022)

Onur Mutlu Lectures
29.4K subscribers
Subscribed
18
Share
Clip
Save

830 views Streamed live on Oct 28, 2022
Computer Architecture, ETH Zürich, Fall 2022 (https://safari.ethz.ch/architecture/f...)

Lecture 10: Real Processing in Memory Systems: UPMEM Case Study

https://youtu.be/pHEHdLsnNdk

Real PIM Tutorial: Hands-on Lab

Programming and Understanding a Real Processing-in-Memory Architecture

Instructors: Geraldo F. Oliveira, Prof. Onur Mutlu

1. Introduction

In this lab, you will work hands-on with a real processing-in-memory (PIM) architecture. You will program the UPMEM PIM architecture [1, 2, 3, 4] for several workloads and will experiment with them. Your main goals are (1) to become familiar with the UPMEM PIM system organization (as an example of real-world memory-centric computing system), (2) to understand the UPMEM programming model and write your own code, and (3) to understand the microarchitecture and instruction set architecture (ISA) of UPMEM's PIM core (called *DRAM Processing Unit, DPU*).

As we introduced in this tutorial, the UPMEM PIM architecture is composed of multiple DPUs (up to 2,560), each of which has access to its own DRAM bank (called *Main RAM, MRAM*) and its own scratchpad memory (called *Working RAM, WRAM*). You can find a full description of the UPMEM PIM system in [3, 4].

2. Lab Resources

You can download the necessary materials for this lab from here: `https://events.safari.ethz.ch/heart24-memorycentric-tutorial/lib/exe/fetch.php?media=template.zip`

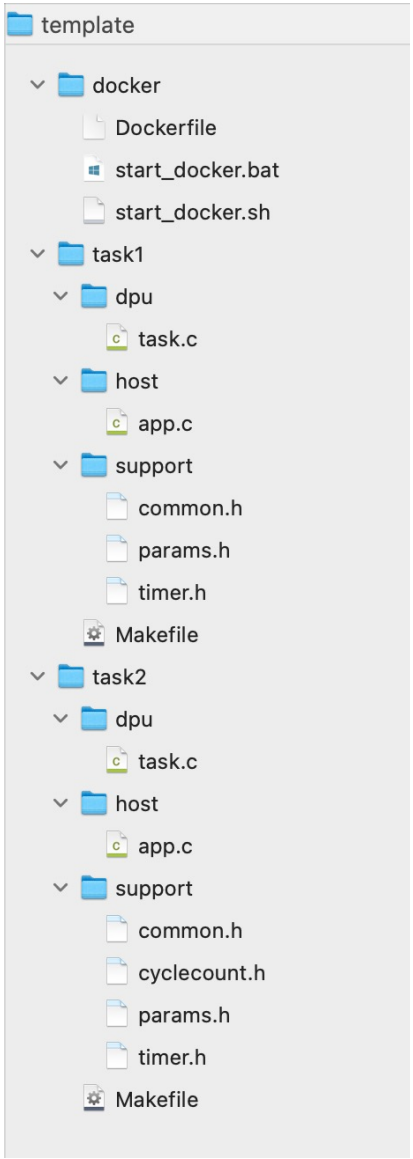
2.1. Source Code

The source code that we provide contains templates for tasks 1 (Section 4) and 2 (Section 5). For the rest of tasks, you can use the same template as for task 2. You can find the templates in the folder `template`. Look for `//@@` to find the places where you need to insert code. Do **NOT** modify any files or folders unless explicitly specified in the list below.

- `task1`
  - `Makefile`
  - `host`
    - `app.c`: Host CPU code (**modifiable**).
  - `dpu`
    - `task.c`: DPU kernel code. It is empty in this template because it is not needed for task 1.
  - `support`
    - `common.h`: Common definitions. Note that `T` is `int64_t` for this task.
    - `params.h`: Functions to read input parameters from command line.
    - `timer.h`: Timing functions.
- `task2`
  - `Makefile`
  - `host`
    - `app.c`: Host CPU code (**modifiable**).
  - `dpu`
    - `task.c`: DPU kernel code (**modifiable**).
  - `support`
    - `common.h`: Common definitions. Note that there are definitions for different data types and size of transfers between MRAM and WRAM.

Template Files

- Contain templates for task 1 and task 2
- Task 2's template can be used for the remaining tasks



Task 1: CPU-DPU and DPU-CPU Transfers

- Use serial, parallel, and broadcast transfers

Your tasks are as follows:

1. Write a host program that exercises all types of data transfers between the host main memory and one or multiple MRAM banks. Concretely, there are three types of data transfers [2]: (1) serial, (2) parallel, and (3) broadcast. Serial and parallel transfers move data from main memory to the MRAM banks or vice versa. Broadcast transfers can only happen from the main memory to the MRAM banks.
2. Evaluate all different types of data transfers for data transfers of size (1) 1MB, (2) 24MB, (3) 48MB per DPU. Use different numbers of DPUs between 1 and 64.

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

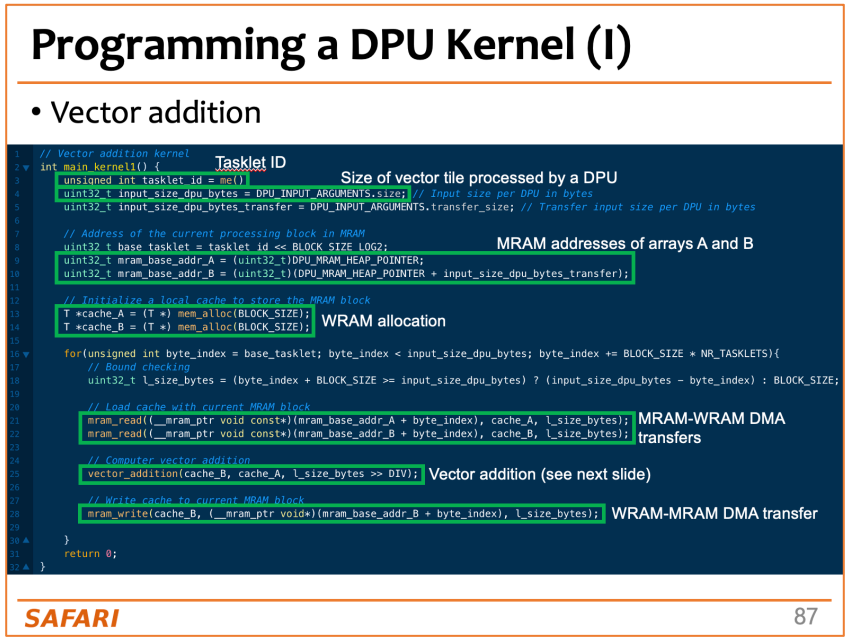
- `dpu_broadcast_to();`
  - Only CPU to DPU
- We transfer the same buffer to all DPUs in the `dpu_set`

Task 2: AXPY

Your tasks are as follows:

1. Write a DPU kernel that executes the AXPY operation ($y = y + alpha \times x$) [5] on every element of a vector. You have to (1) transfer two input vectors, `Y` and `X`, to the MRAM bank/s, (2) perform the AXPY operation with a variable number of tasklets, (3) write the results to the output vector, `Y`, and (4) transfer the output vector back to the host main memory.

- VA is a good reference code for this task



Task 3: Operations and Datatypes

Your tasks are as follows:

1. Modify your AXPY DPU kernel to make it a vector addition ($y = y + x$) and to support other operations besides addition (i.e., subtraction, multiplication, division).
2. Evaluate the performance of your new kernel for different operations (addition, subtraction, multiplication, division) and data types (`char`, `short`, `int`, `long long int`, `float`, `double`).

- You will observe significant variations in arithmetic throughput for different operations and datatypes

Task 4: Vector Reduction

Your tasks are as follows:

1. Your vector reduction DPU kernel should have four different versions: (1) final reduction with a single tasklet, (2) final tree-based reduction with barriers, (3) final tree-based reduction with handshakes, (4) final reduction with mutexes.

- Performance differences due to the final reduction step

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

Accumulate in a local sum

Copy local sum into WRAM

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

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

    // Total count in this DPU
    result->t_count = message[0];
}
```

Barrier synchronization

Sequential accumulation

SAFARI 94

Real PIM Tutorial: Hands-on Lab

References

[1] UPMEM. UPMEM Software Development Kit (SDK). https://sdk.upmem.com, 2023.

[2] UPMEM. UPMEM User Manual. https://sdk.upmem.com/2023.1.0/, 2023.

[3] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernández, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture. arXiv:2105.03814 [cs.AR], 2021.

[4] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access*, 2022.

[5] Docker Inc. Docker. https://www.docker.com, 2023.

[6] Wikipedia. Basic Linear Algebra Subprograms. Level 1. https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms#Level_1, 2023.

[7] Wikipedia. Grayscale. https://en.wikipedia.org/wiki/Grayscale, 2023.

[8] LLVM. llvm-objdump - LLVM's Object File Dumper. https://llvm.org/docs/CommandGuide/llvm-objdump.html, 2023.

[9] Compiler Explorer. Compiler Explorer for DPU. https://dpu.dev, 2023.

HEART 2024 Tutorial

Memory-Centric Computing Systems

Processing-Near-Memory

Programming General-purpose PIM

Geraldo F. Oliveira

Dr. Juan Gómez Luna

Professor Onur Mutlu

ETH zürich

SAFARI

Friday, June 21, 2024