

P&S Processing-in-Memory

Programming Processing-in-Memory Architectures

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Spring 2022

21 April 2022

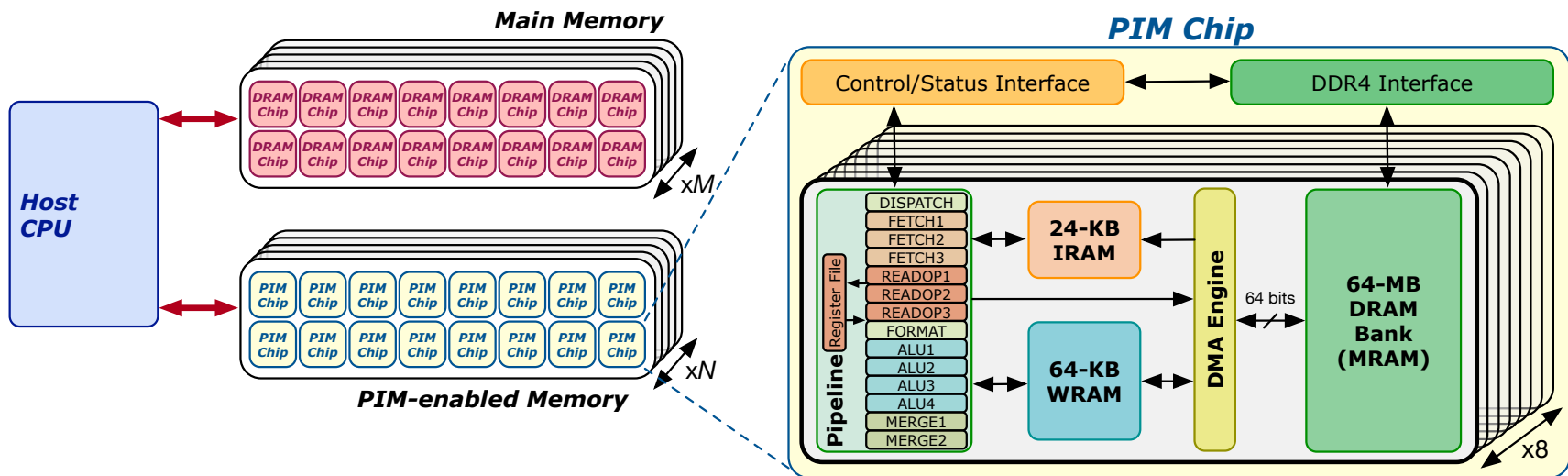
UPMEM Processing-in-DRAM Engine (2019)

- **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



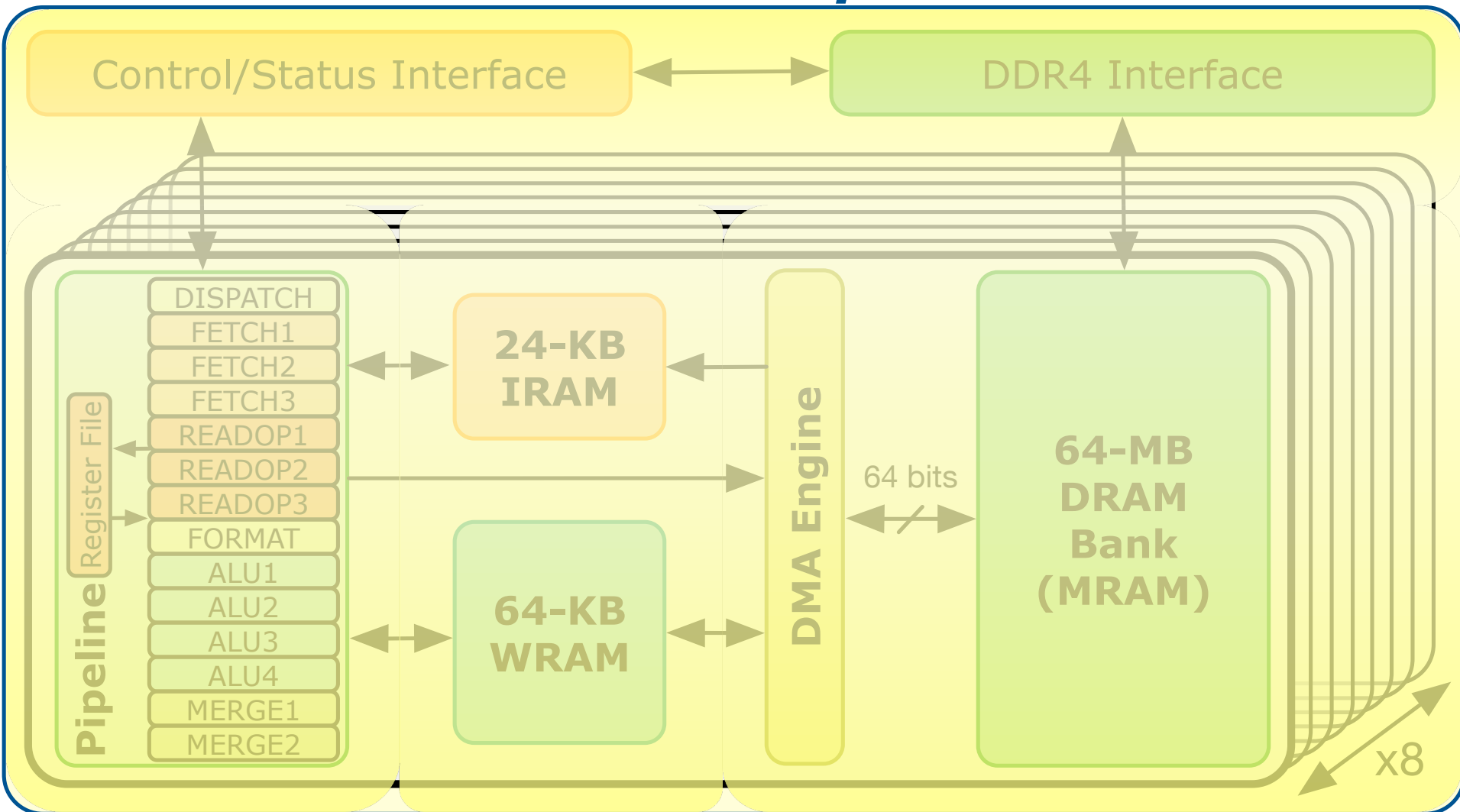
System Organization

- 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



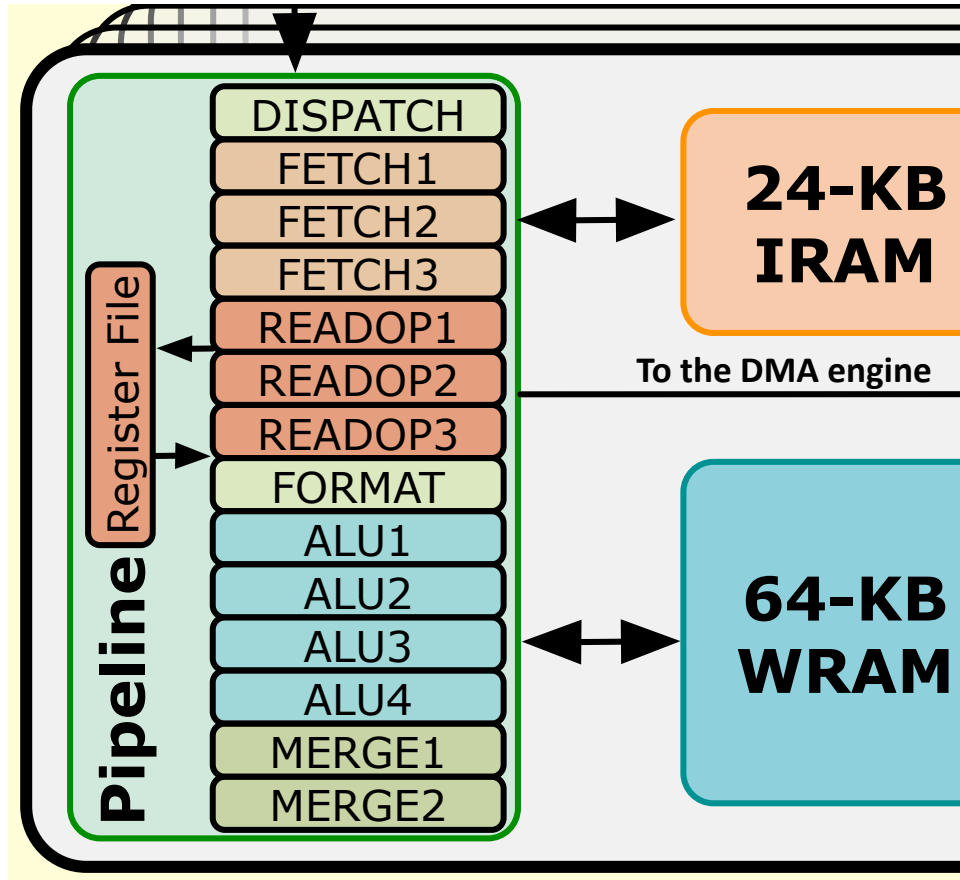
DRAM Processing Unit

PIM Chip



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



* 350 MHz in the UPMEM-based PIM system used for the experimental results shown in this lecture

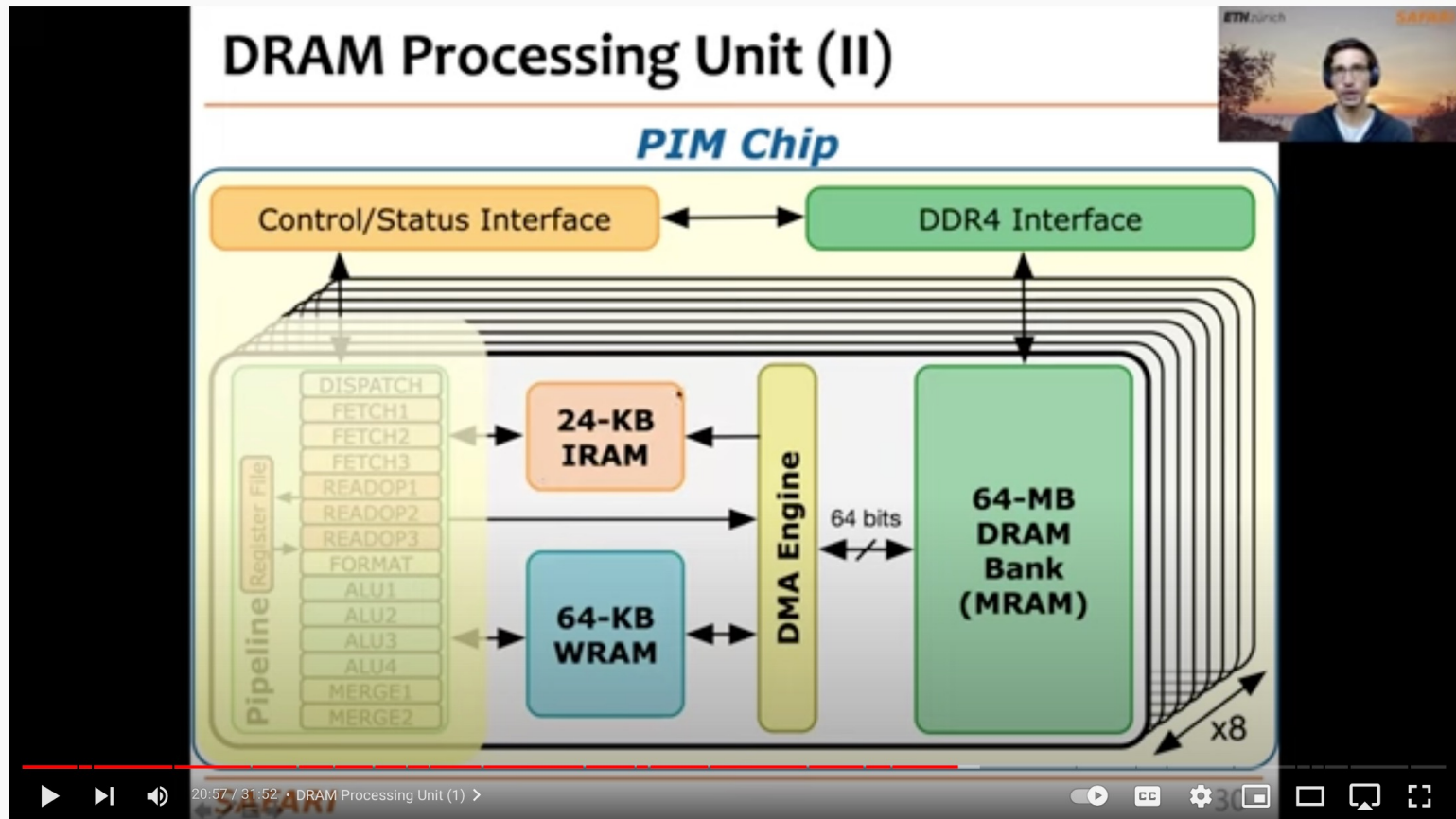
DPU Instruction Set Architecture

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

The screenshot shows a web page titled "Instruction Set Architecture" from the "UPMEM development tools documentation". The page has a blue header with a hamburger menu icon and the title. Below the header, there is a breadcrumb trail "» Instruction Set Architecture" and a link "View page source". The main content area has a heading "Instruction Set Architecture" followed by a paragraph: "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." Below this is another paragraph: "Software developers should use this section as a reference manual to develop or debug assembly code." The next section is "Resources overview" followed by "Thread registers". Under "Thread registers", there is a paragraph: "The system is composed of 24 hardware threads. Each of them owns a set of private resources:" followed by a bulleted list: "• 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.", and "• Two persistent flags, keeping information about the previous result of an arithmetic or logical instruction:" with a sub-bullet "◦ ZF: last result is equal to zero".

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

UPMEM PIM Architecture: Lectures 2 & 3



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

477 views • Premiered Mar 17, 2022

18 DISLIKE SHARE CLIP SAVE ...



Onur Mutlu Lectures
24.1K subscribers

Projects & Seminars, ETH Zürich, Spring 2022
Exploring the Processing-in-Memory Paradigm for Future Computing Systems (
https://safari.ethz.ch/projects_and_s...)

SUBSCRIBED



Accelerator Model (I)

- UPMEM DIMMs coexist with conventional DIMMs
- Integration of UPMEM DIMMs in a system follows an **accelerator model**
- 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

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

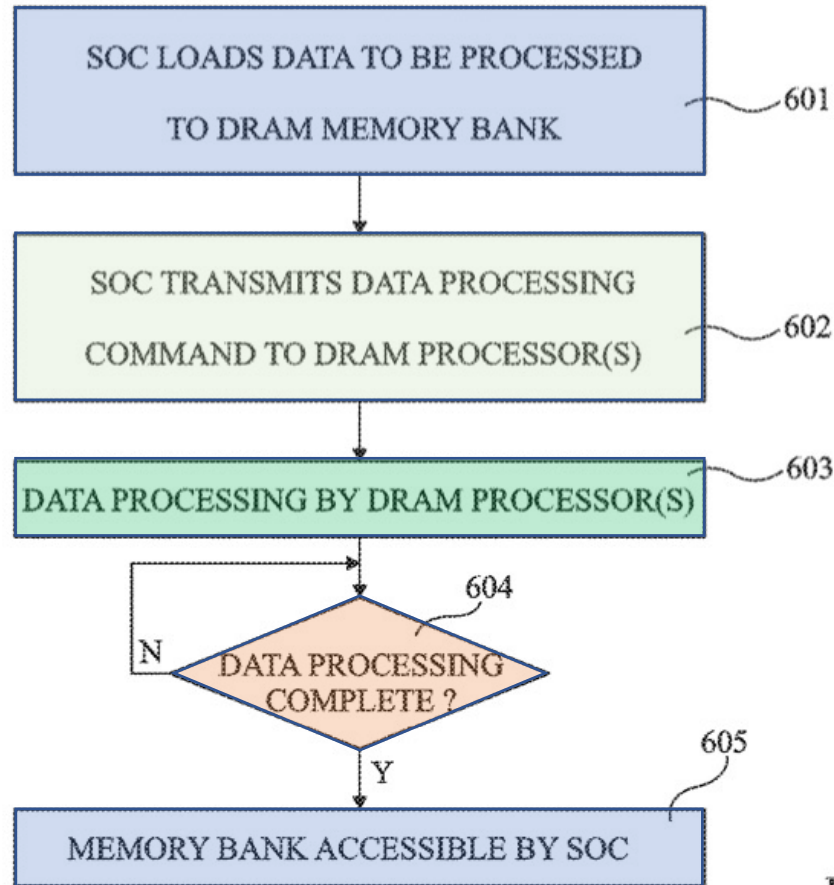
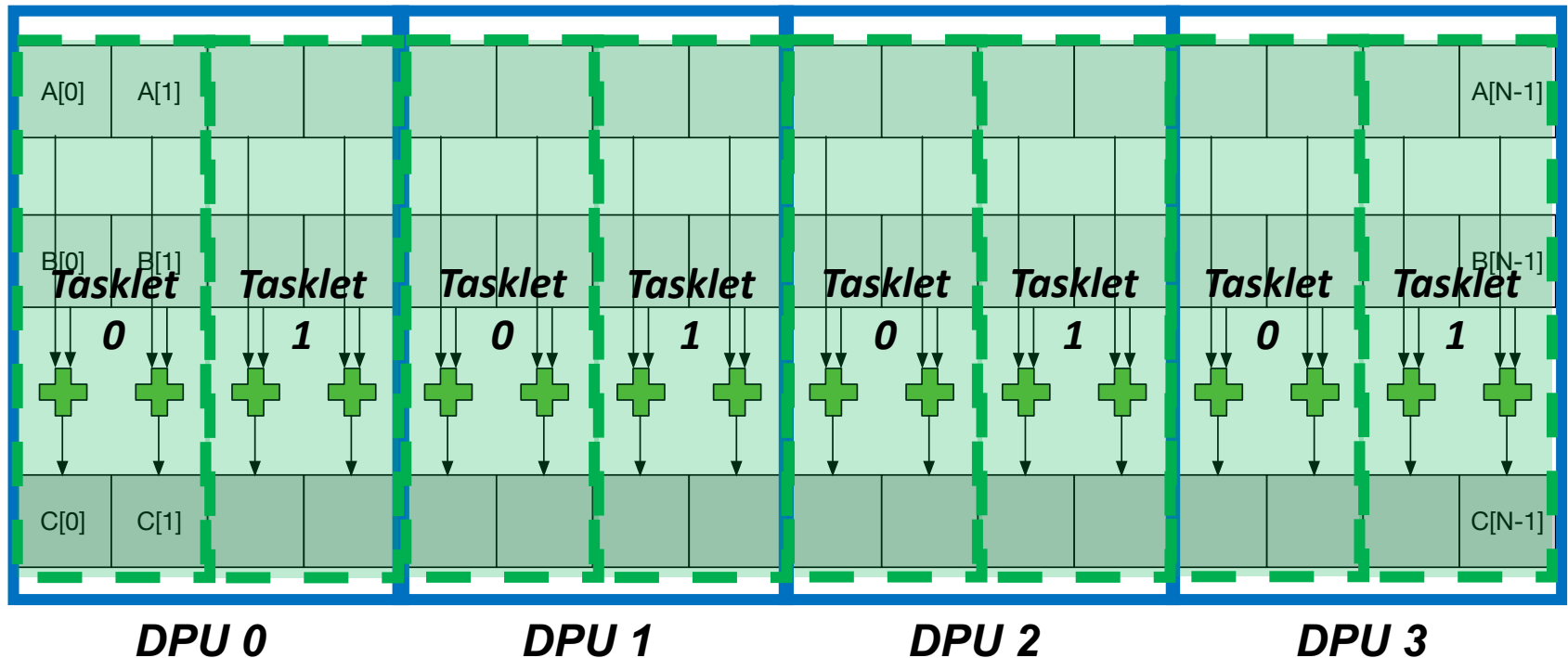


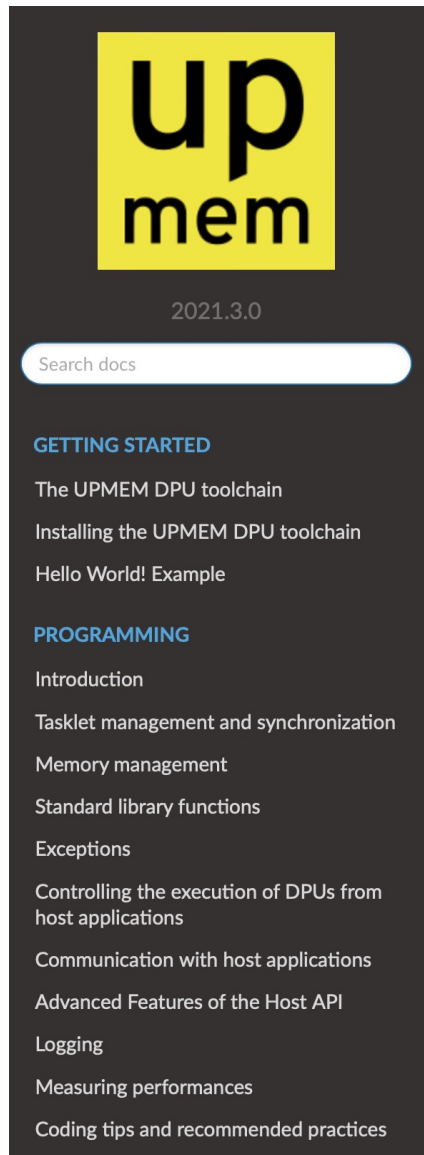
Fig 6

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



[Home](#) » User Manual

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-lddb
- 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
 - Running and testing hello world
 - Creating a host application to drive 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.

* <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
```

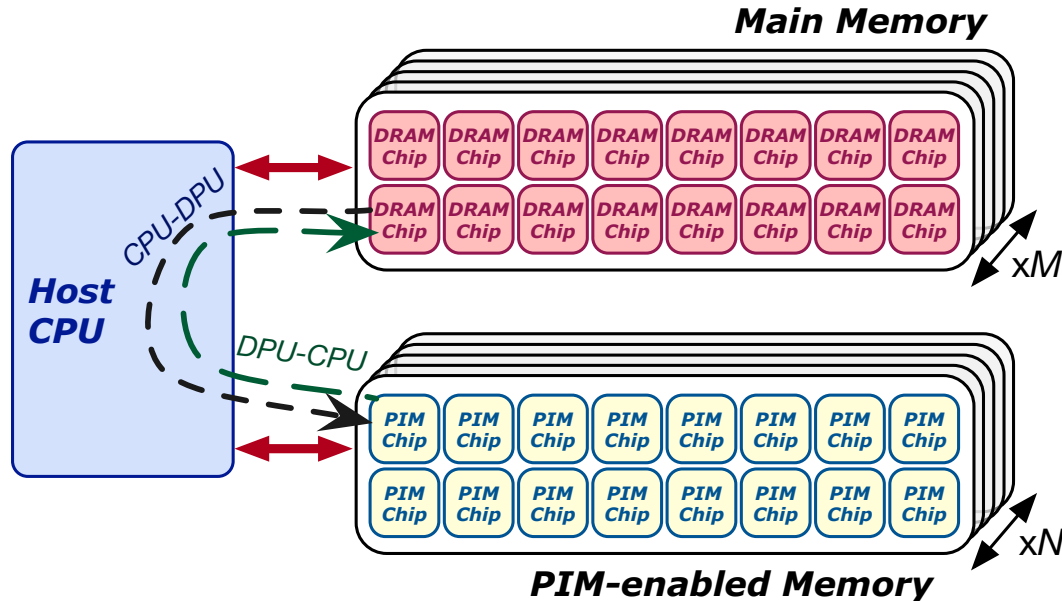
Is it possible to launch different kernels onto different DPUs?

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

```
1 ▾ DPU_FOREACH (dpu_set, dpu) {  
2   DPU_ASSERT(dpu_copy_to(dpu, DPU_MRAM_HEAP_POINTER_NAME 0, bufferA + input_size_dpu_8bytes * i, input_size_dpu_8bytes * sizeof(T));  
3   DPU_ASSERT(dpu_copy_from(dpu, DPU_MRAM_HEAP_POINTER_NAME input_size_dpu_8bytes * sizeof(T), bufferB + input_size_dpu_8bytes * i, input_size_dpu_8bytes * sizeof(T));  
4   i++;  
5 ▲ }  
6
```

	Offset within MRAM	Pointer to main memory	Transfer size
	0	bufferA + input_size_dpu_8bytes * i	input_size_dpu_8bytes * sizeof(T)
	input_size_dpu_8bytes * sizeof(T)	bufferB + input_size_dpu_8bytes * i	input_size_dpu_8bytes * sizeof(T)

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

```
1 DPU_FOREACH(dpu_set, dpu, i) {
2     DPU_ASSERT(dpu_prepare_xfer(dpu, bufferA + input_size_dpu_8bytes * i))
3 }
4 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));
5
6 DPU_FOREACH(dpu_set, dpu, i) {
7     DPU_ASSERT(dpu_prepare_xfer(dpu, bufferB + input_size_dpu_8bytes * i))
8 }
9 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));
10
```

Pointer to main memory

Offset within MRAM

Transfer size

Direction

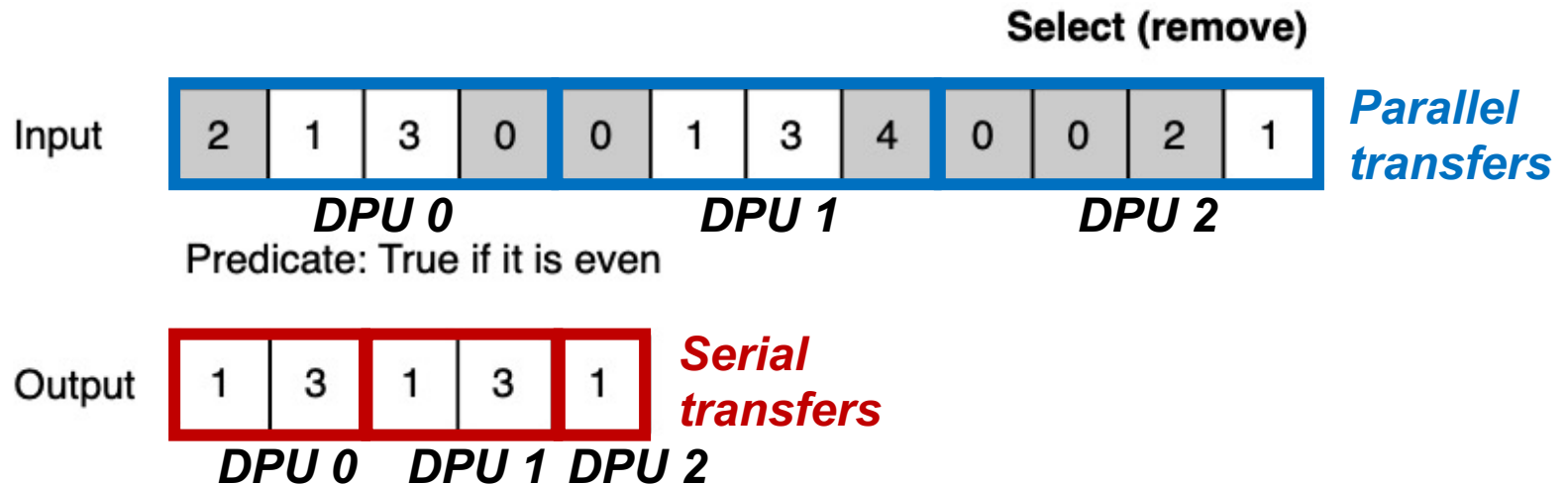
Broadcast Transfers

- `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
```

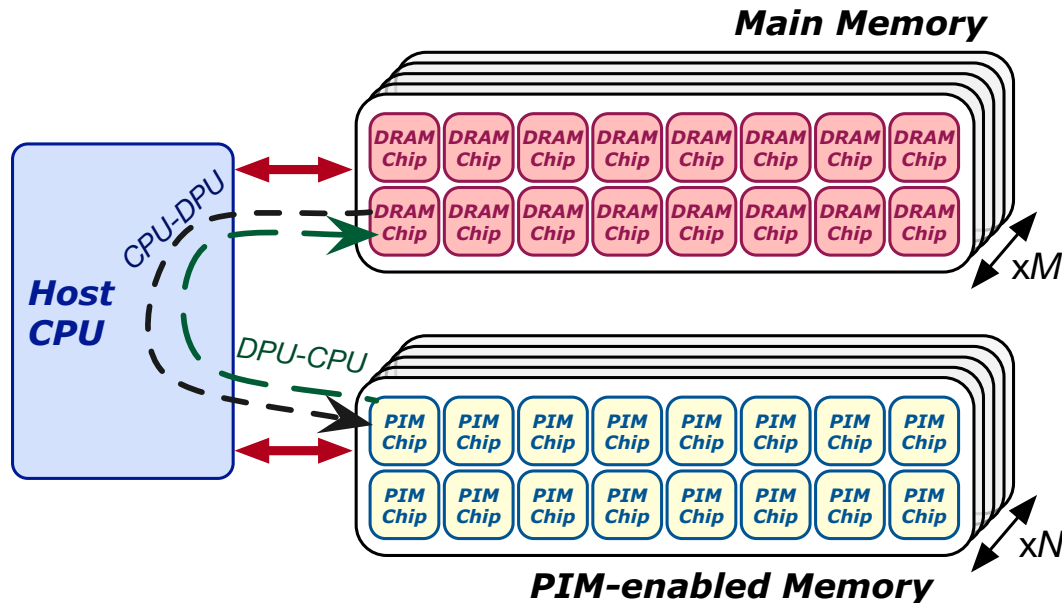
Different Types of Transfers in a Program

- 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?

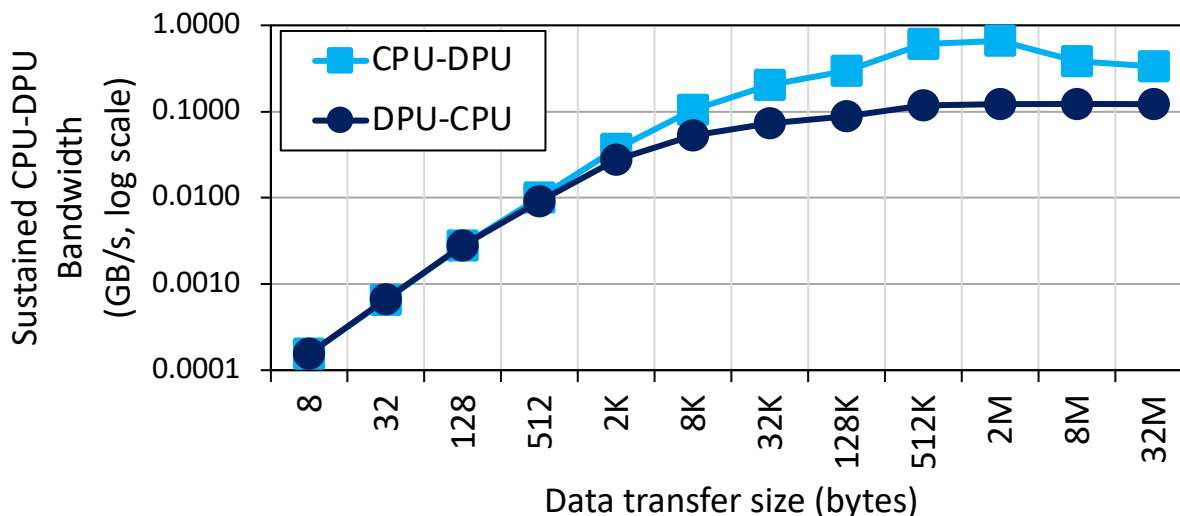
- With a microbenchmark, we obtain the **sustained bandwidth of all types of CPU-DPU and DPU-CPU transfers**
- Two experiments:
 - **1 DPU**: variable CPU-DPU and DPU-CPU transfer size (**8 bytes to 32 MB**)
 - **1 rank**: 32 MB CPU-DPU and DPU-CPU transfers to/from a set of **1 to 64 MRAM banks** within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

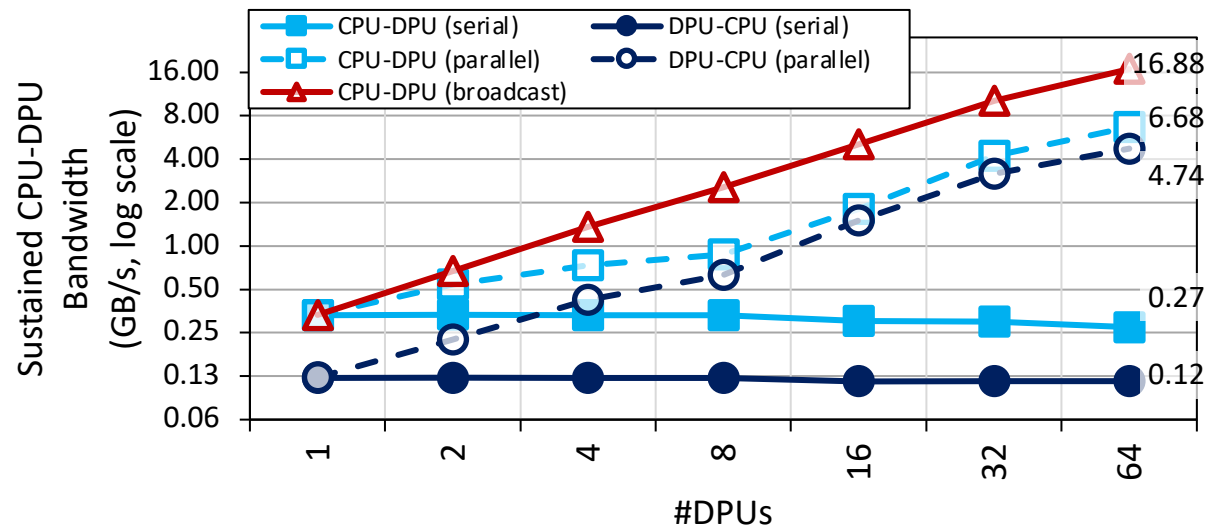


KEY OBSERVATION 7

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

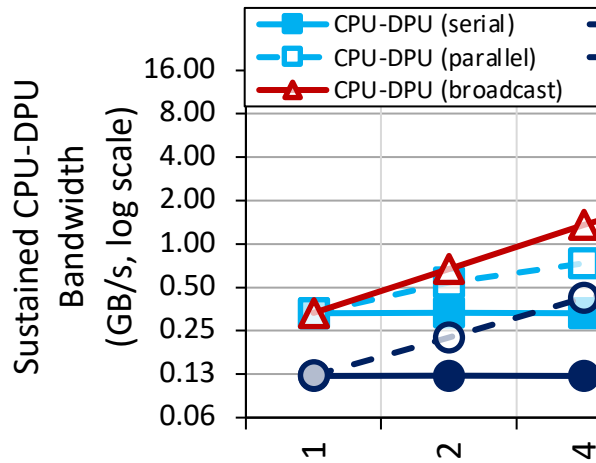


KEY OBSERVATION 8

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

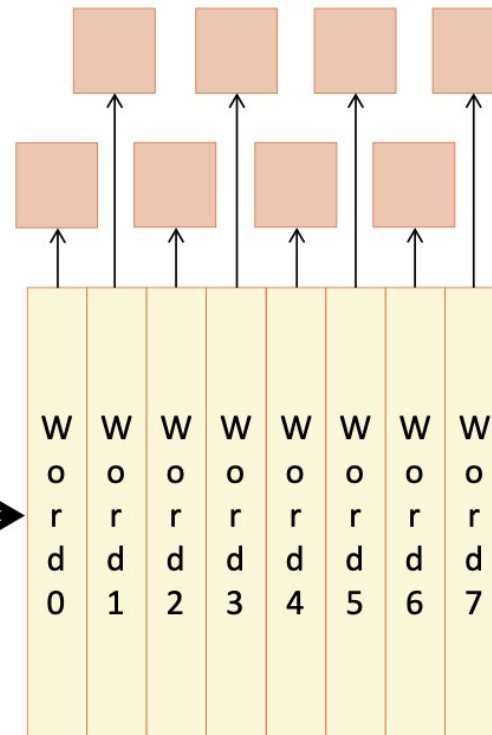
Eight 64-bit “horizontal” words are turned into 8 vertical words, feeding 8 different DRAM chips

This way DPUs see full 64-bit words, not chunk of them

DRAM chip have 8-bit data bus

Word 0
Word 1
Word 2
Word 3
Word 4
Word 5
Word 6
Word 7

Library



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

Copyright UPMEM® 2019

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 04, 2020 at 13:55:41 UTC from IEEE Xplore. Restrictions apply.

HOT CHIPS 31

up
mem

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

..		
folder dpu	PRIM -- first commit	7 days ago
folder host	PRIM -- first commit	7 days ago
folder support	PRIM -- first commit	7 days ago
file Makefile	PRIM -- first commit	7 days ago
file 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?

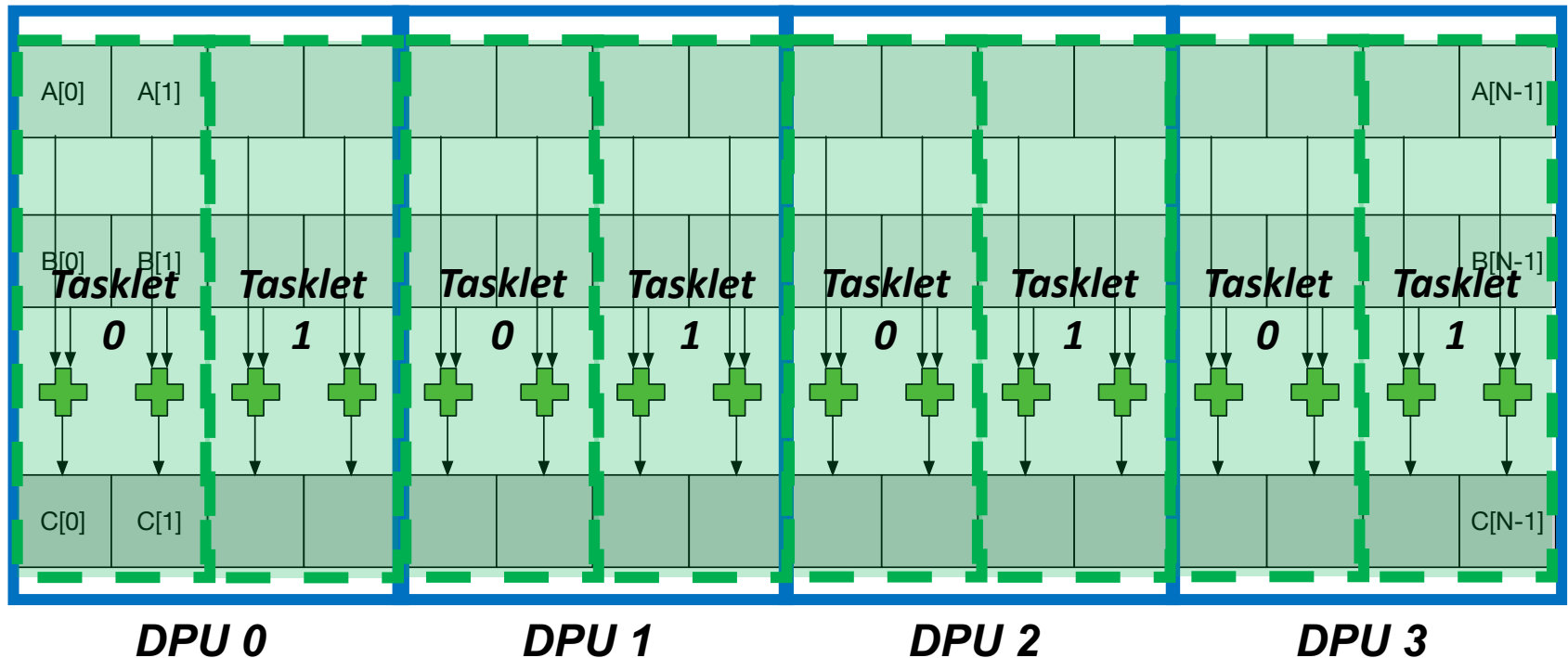
- We can use serial and parallel transfers
- We pass them directly to the scratchpad memory of the DPU
 - **Working RAM (WRAM)**: We introduce it in the next slides
- This is useful for **input parameters and some results**

```
1 // In DPU WRAM (dpu/task.c)
2 __host dpu_arguments_t DPU_INPUT_ARGUMENTS;
3 __host dpu_results_t DPU_RESULTS[NR_TASKLETS];
4
```

```
1 // Host code (host/app.c)
2 #ifdef SERIAL
3     DPU_FOREACH (dpu_set, dpu) {
4         DPU_ASSERT(dpu_copy_to(dpu, "DPU_INPUT_ARGUMENTS", 0, (const void *)&input_arguments[i], sizeof(input_arguments[0])));
5         i++;
6     }
7 #else
8     DPU_FOREACH(dpu_set, dpu, i) {
9         DPU_ASSERT(dpu_prepare_xfer(dpu, &input_arguments[i]));
10    }
11    DPU_ASSERT(dpu_push_xfer(dpu_set, DPU_XFER_TO_DPU, "DPU_INPUT_ARGUMENTS", 0, sizeof(input_arguments[0]), DPU_XFER_DEFAULT));
12 #endif
13
```

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

```
1 // Vector addition kernel
2 int main_kernel1() { Tasklet ID
3     unsigned int tasklet_id = me() Size of vector tile processed by a DPU
4     uint32_t input_size_dpu_bytes = DPU_INPUT_ARGUMENTS.size; // Input size per DPU in bytes
5     uint32_t input_size_dpu_bytes_transfer = DPU_INPUT_ARGUMENTS.transfer_size; // Transfer input size per DPU in bytes
6
7     // Address of the current processing block in MRAM
8     uint32_t base_tasklet = tasklet_id << BLOCK_SIZE_LOG2; MRAM addresses of arrays A and B
9     uint32_t mram_base_addr_A = (uint32_t)DPU_MRAM_HEAP_POINTER;
10    uint32_t mram_base_addr_B = (uint32_t)(DPU_MRAM_HEAP_POINTER + input_size_dpu_bytes_transfer);
11
12    // Initialize a local cache to store the MRAM block
13    T *cache_A = (T *) mem_alloc(BLOCK_SIZE); WRAM allocation
14    T *cache_B = (T *) mem_alloc(BLOCK_SIZE);
15
16    for(unsigned int byte_index = base_tasklet; byte_index < input_size_dpu_bytes; byte_index += BLOCK_SIZE * NR_TASKLETS){
17        // Bound checking
18        uint32_t l_size_bytes = (byte_index + BLOCK_SIZE >= input_size_dpu_bytes) ? (input_size_dpu_bytes - byte_index) : BLOCK_SIZE;
19
20        // Load cache with current MRAM block
21        mram_read((__mram_ptr void const*)(mram_base_addr_A + byte_index), cache_A, l_size_bytes); MRAM-WRAM DMA
22        mram_read((__mram_ptr void const*)(mram_base_addr_B + byte_index), cache_B, l_size_bytes); transfers
23
24        // Computer vector addition
25        vector_addition(cache_B, cache_A, l_size_bytes >> DIV); Vector addition (see next slide)
26
27        // Write cache to current MRAM block
28        mram_write(cache_B, (__mram_ptr void*)(mram_base_addr_B + byte_index), l_size_bytes); WRAM-MRAM DMA transfer
29
30    }
31    return 0;
32 }
```

Programming a DPU Kernel (II)

- Vector addition

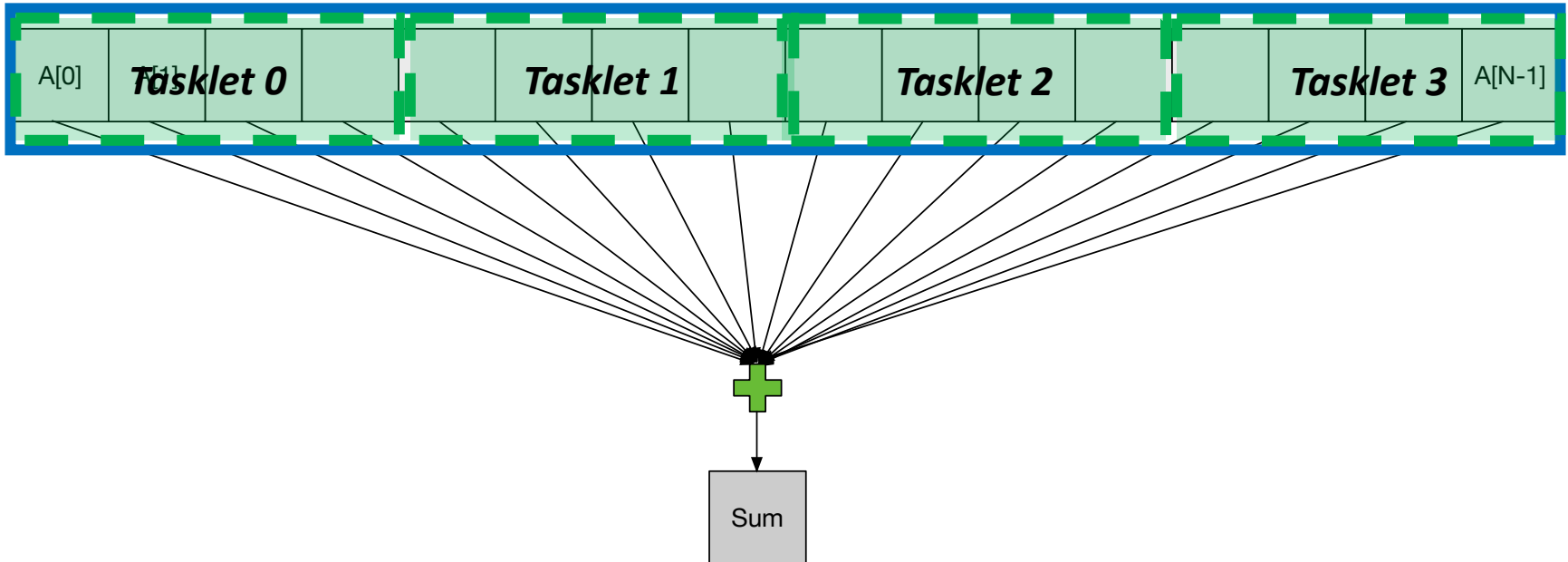
```
1 // vector_addition: Computes the vector addition of a cached block
2 ▼ static void vector_addition(T *bufferB, T *bufferA, unsigned int l_size) {
3
4 ▼     for (unsigned int i = 0; i < l_size; i++){
5         bufferB[i] += bufferA[i];
6 ▲     }
7
8 ▲ }
```


Programming a DPU Kernel (III)

- 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();`

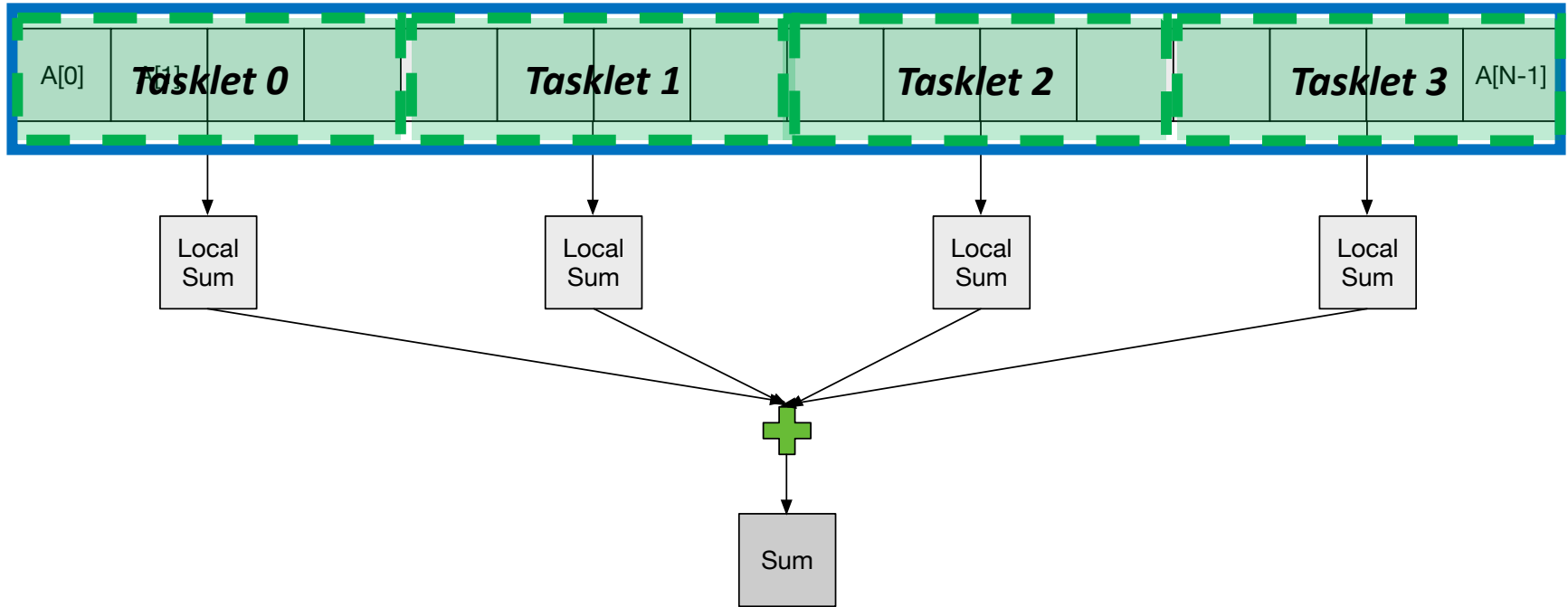
Parallel Reduction (I)

- 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); Accumulate in a local sum
11
12 ▲ }
13 // Copy local count to shared array in WRAM
14 message[tasklet_id] = l_count; Copy local sum into WRAM
```

Final Reduction

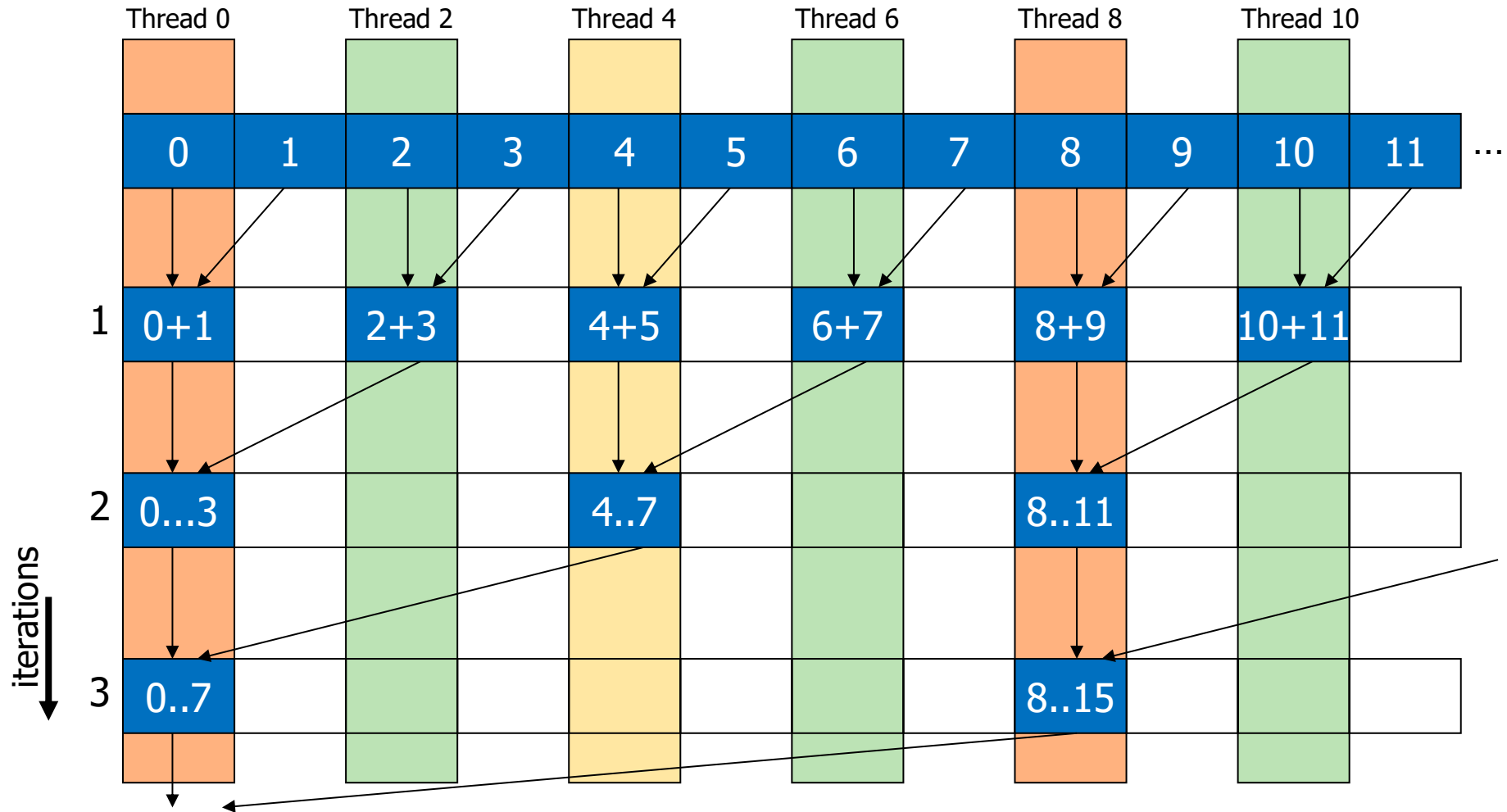
- A single tasklet can perform the final reduction

```
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); Accumulate in a local sum
11
12 ▲ }
13 // Copy local count to shared array in WRAM
14 message[tasklet_id] = l_count; Copy local sum into WRAM
```



```
1 // Single-thread reduction
2 // Barrier
3 barrier_wait(&my_barrier); Barrier synchronization
4
5 ▼ if(tasklet_id == 0){
6     #pragma unroll
7     for (unsigned int each_tasklet = 1; each_tasklet < NR_TASKLETS; each_tasklet++){
8         message[0] += message[each_tasklet]; Sequential accumulation
9     }
10
11 // Total count in this DPU
12 result->t_count = message[0];
13 ▲ }
```

Vector Reduction: Naïve Mapping



Using Barriers: Tree-Based Reduction

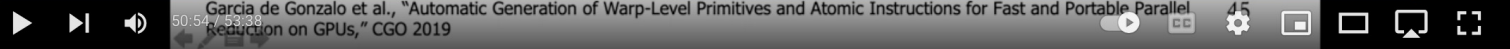
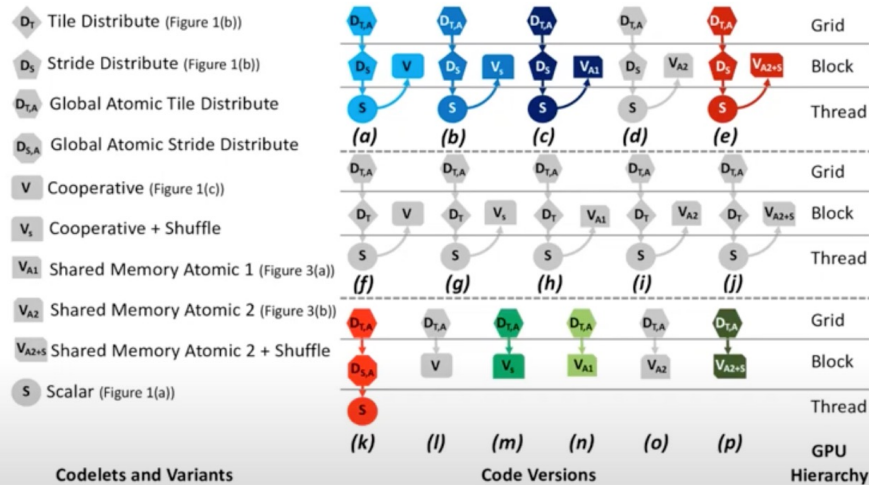
- Multiple tasklets can perform a tree-based reduction
 - After every iteration tasklets synchronize with a barrier
 - Half of the tasklets retire at the end of an iteration

```
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]; "offset" tasklets working
9 ▲     }
10
11 // Barrier
12 barrier_wait(&my_barrier); Barrier synchronization
13 ▲ }
```

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

Parallel Reduction on GPU

Search Space of Parallel Reduction



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

201 views • Premiered Apr 19, 2022

15 DISLIKE SHARE CLIP SAVE ...



Onur Mutlu Lectures

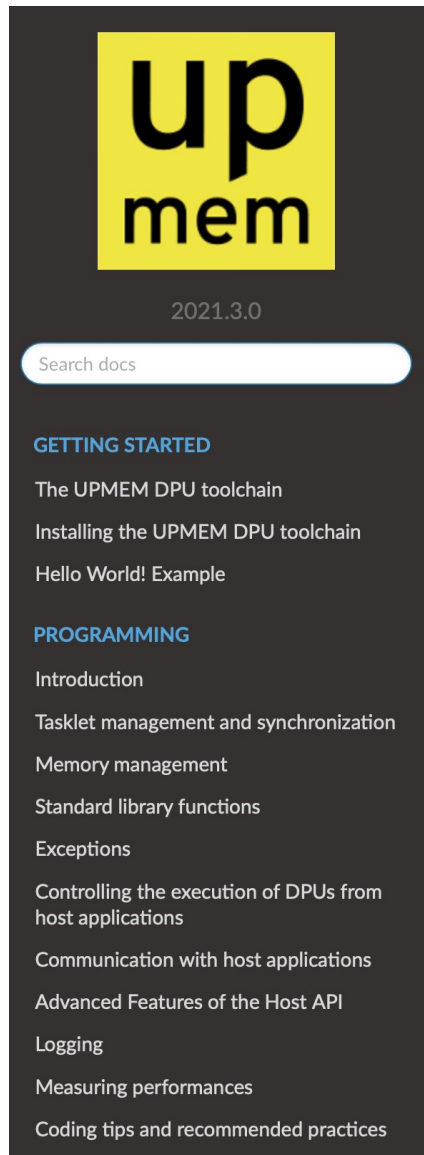
24.1K subscribers

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

SUBSCRIBED



UPMEM SDK Documentation



[Home](#) » User Manual

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-lddb
- 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
 - Running and testing hello world
 - Creating a host application to drive the program

PrIM Benchmarks

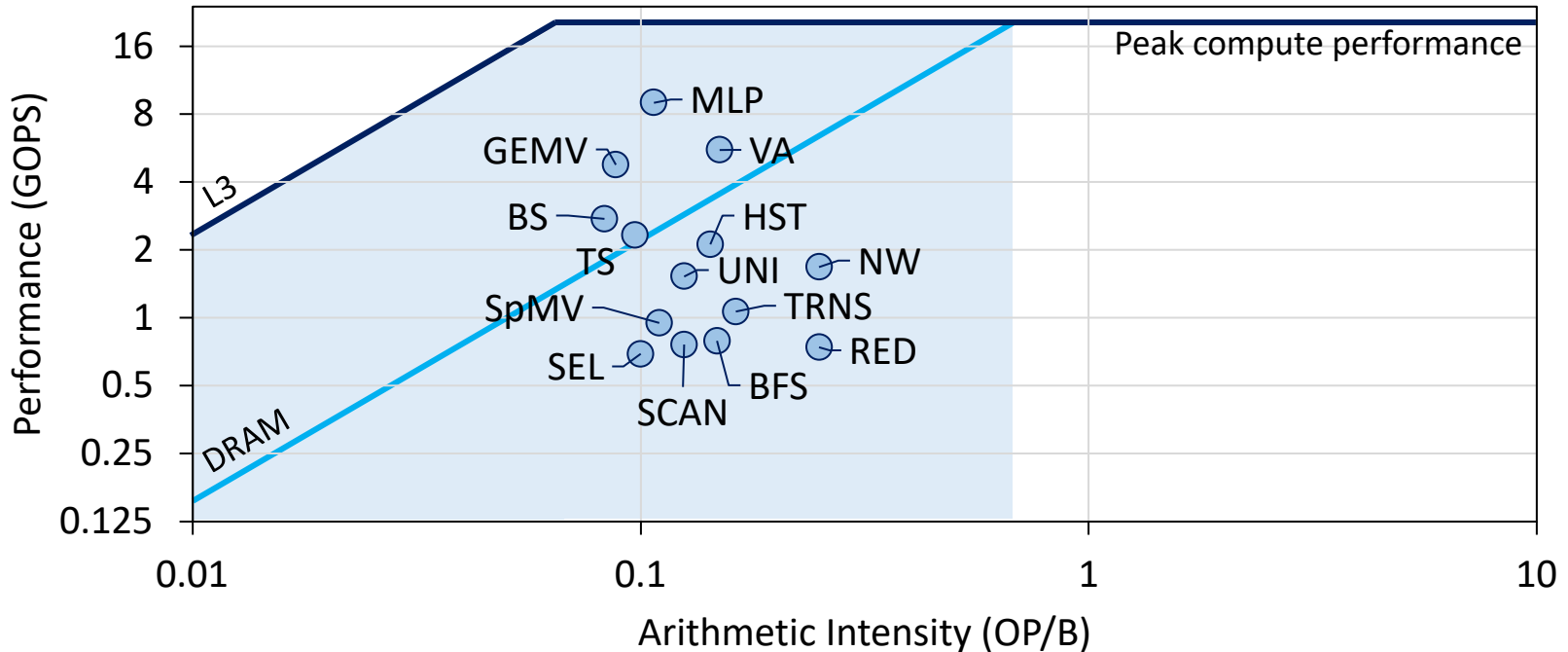
- Goal
 - A **common set of workloads** that can be used to
 - evaluate the UPMEM PIM architecture,
 - compare software improvements and compilers,
 - compare future PIM architectures and hardware
- Two key selection criteria:
 - Selected workloads from **different application domains**
 - **Memory-bound workloads** on processor-centric architectures
- 14 different workloads, 16 different benchmarks*

PrIM Benchmarks: Application Domains

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

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

Domain	Benchmark	Short name	Memory access pattern			Computation pattern		Communication/synchronization	
			Sequential	Strided	Random	Operations	Datatype	Intra-DPU	Inter-DPU
Dense linear algebra	Vector Addition	VA	Yes			add	int32_t		
	Matrix-Vector Multiply	GEMV	Yes			add, mul	uint32_t		
Sparse linear algebra	Sparse Matrix-Vector Multiply	SpMV	Yes		Yes	add, mul	float		
Databases	Select	SEL	Yes			add, compare	int64_t	handshake, barrier	Yes
	Unique	UNI	Yes			add, compare	int64_t	handshake, barrier	Yes
Data analytics	Binary Search	BS	Yes		Yes	compare	int64_t		
	Time Series Analysis	TS	Yes			add, sub, mul, div	int32_t		
Graph processing	Breadth-First Search	BFS	Yes		Yes	bitwise logic	uint64_t	barrier, mutex	Yes
Neural networks	Multilayer Perceptron	MLP	Yes			add, mul, compare	int32_t		
Bioinformatics	Needleman-Wunsch	NW	Yes	Yes		add, sub, compare	int32_t	barrier	Yes
Image processing	Image histogram (short)	HST-S	Yes		Yes	add	uint32_t	barrier	Yes
	Image histogram (long)	HST-L	Yes		Yes	add	uint32_t	barrier, mutex	Yes
Parallel primitives	Reduction	RED	Yes	Yes		add	int64_t	barrier	Yes
	Prefix sum (scan-scan-add)	SCAN-SSA	Yes			add	int64_t	handshake, barrier	Yes
	Prefix sum (reduce-scan-scan)	SCAN-RSS	Yes			add	int64_t	handshake, barrier	Yes
	Matrix transposition	TRNS	Yes		Yes	add, sub, mul	int64_t	mutex	

PrIM Benchmarks: Inter-DPU Communication

Domain	Benchmark	Short name	Memory access pattern			Computation pattern		Communication/synchronization	
			Sequential	Strided	Random	Operations	Datatype	Intra-DPU	Inter-DPU
Dense linear algebra	Vector Addition	VA	Yes			add	int32_t		
	Matrix-Vector Multiply	GEMV	Yes			add, mul	uint32_t		
Sparse linear algebra	Sparse Matrix-Vector Multiply	SpMV	Yes		Yes	add, mul	float		
Databases	Select	SEL	Yes			add, compare	int64_t	handshake, barrier	Yes
	Unique	UNI	Yes			add, compare	int64_t	handshake, barrier	Yes
Data analytics	Binary Search	BS	Yes		Yes	compare	int64_t		
	Time Series Analysis	TS	Yes			add, sub, mul, div	int32_t		
Graph processing	Breadth-First Search	BFS	Yes		Yes	bitwise logic	uint64_t	barrier, mutex	Yes
Neural networks	Multilayer Perceptron	MLP	Yes			add, mul, compare	int32_t		
Bioinformatics	Needleman-Wunsch	NW	Yes	Yes		add, sub, compare	int32_t	barrier	Yes
Image processing	Image histogram (short)	HST-S	Yes		Yes	add	uint32_t	barrier	Yes
	Image histogram (long)	HST-L	Yes		Yes	add	uint32_t	barrier, mutex	Yes
Parallel primitives	Reduction	RED	Yes	Yes		add	int64_t	barrier	Yes
	Prefix sum (scan-scan-add)	SCAN-SSA	Yes			add	int64_t	handshake, barrier	Yes
	Prefix sum (reduce-scan-scan)	SCAN-RSS	Yes			add	int64_t	handshake, barrier	Yes
	Matrix transposition	TRNS	Yes		Yes	add, sub, mul	int64_t	mutex	

PrIM Benchmarks: Inter-DPU Communication

Domain	Benchmark	Short name	Memory access pattern			Computation pattern		Communication/synchronization	
			Sequential	Strided	Random	Operations	Datatype	Intra-DPU	Inter-DPU
Dense linear algebra	Vector Addition	VA	Yes			add	int32_t		
	Matrix-Vector Multiply	GEMV	Yes			add, mul	uint32_t		
Sparse linear algebra	Sparse Matrix-Vector Multiply	SpMV	Yes		Yes	add, mul	float		
Databases	Select	SEL	Yes			add, compare	int64_t	handshake, barrier	Yes
	Unique	UNI	Yes			add, compare	int64_t	handshake, barrier	Yes
Data analytics	Binary Search	BS	Yes		Yes	compare	int64_t		
	Time Series Analysis	TS	Yes			add, sub, mul, div	int32_t		
Graph processing	Breadth-First Search	BFS	Yes		Yes	bitwise logic	uint64_t	barrier, mutex	Yes
Neural networks	Multilayer Perceptron	MLP	Yes			add, mul, compare	int32_t		
Bioinformatics	Needleman-Wunsch	NW	Yes	Yes		add, sub, compare	int32_t	barrier	Yes
Image processing	Image histogram (short)	HST-S	Yes		Yes	add	uint32_t	barrier	Yes
	Image histogram (long)	HST-L	Yes		Yes	add	uint32_t	barrier, mutex	Yes
Parallel primitives	Reduction	RED	Yes	Yes		add	int64_t	barrier	Yes
	Prefix sum (scan-scan-add)	SCAN-SSA	Yes			add	int64_t	handshake, barrier	Yes
	Prefix sum (reduce-scan-scan)	SCAN-RSS	Yes			add	int64_t	handshake, barrier	Yes
	Matrix transposition	TRNS	Yes		Yes	add, sub, mul	int64_t	mutex	

- 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

PrIM Benchmarks

- 16 benchmarks and scripts for evaluation
- <https://github.com/CMU-SAFARI/prim-benchmarks>

CMU-SAFARI / prim-benchmarks

<> Code Issues Pull requests Actions Projects Wiki Security Insights Settings

main 1 branch 0 tags Go to file Add file Code

Juan Gomez Luna PrIM -- first commit		3de4b49 15 days ago 2 commits
BFS	PrIM -- first commit	15 days ago
BS	PrIM -- first commit	15 days ago
GEMV	PrIM -- first commit	15 days ago
HST-L	PrIM -- first commit	15 days ago
HST-S	PrIM -- first commit	15 days ago
MLP	PrIM -- first commit	15 days ago
Microbenchmarks	PrIM -- first commit	15 days ago
NW	PrIM -- first commit	15 days ago
RED	PrIM -- first commit	15 days ago
SCAN-RSS	PrIM -- first commit	15 days ago
SCAN-SSA	PrIM -- first commit	15 days ago
SEL	PrIM -- first commit	15 days ago
SpMV	PrIM -- first commit	15 days ago
TRNS	PrIM -- first commit	15 days ago
TS	PrIM -- first commit	15 days ago
UNI	PrIM -- first commit	15 days ago
VA	PrIM -- first commit	15 days ago
LICENSE	PrIM -- first commit	15 days ago
README.md	PrIM -- first commit	15 days ago
run_strong_full.py	PrIM -- first commit	15 days ago
run_strong_rank.py	PrIM -- first commit	15 days ago
run_weak.py	PrIM -- first commit	15 days ago

Upcoming Lectures

- More real-world PIM architectures
- More on workload characterization for PIM suitability
 - Benchmarking and workload suitability on the UPMEM PIM architecture
- PUM architectures and prototypes

P&S Processing-in-Memory

Programming Processing-in-Memory Architectures

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Spring 2022

21 April 2022