

P&S Processing-in-Memory

Real-World Processing-in-Memory Architectures:
Microbenchmarking of UPMEM PIM

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Spring 2022

24 March 2022

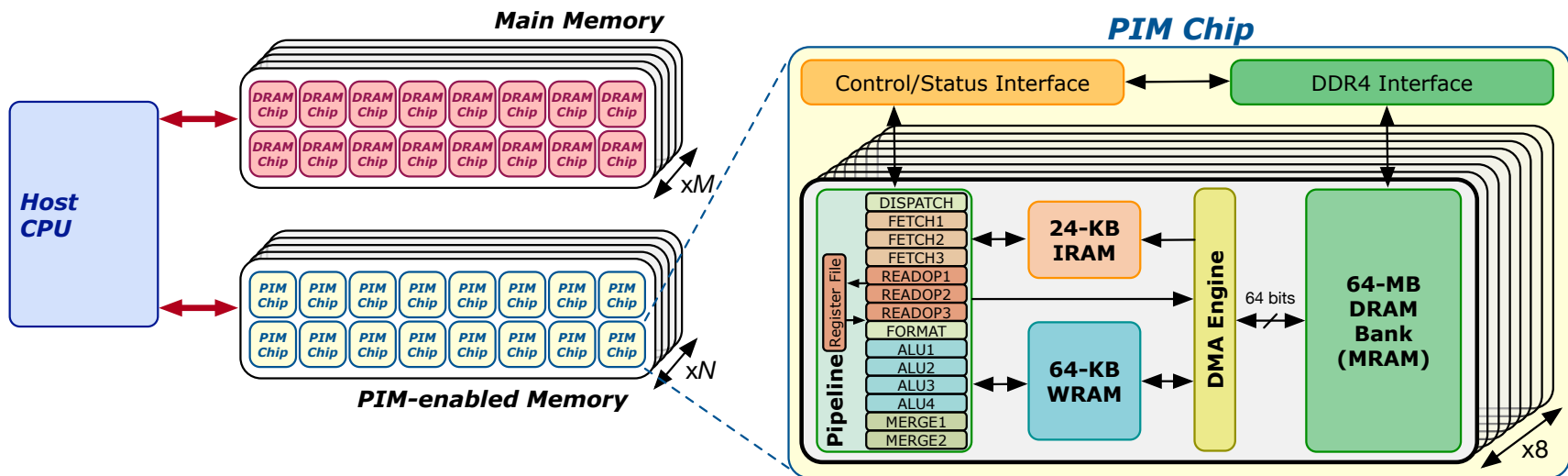
UPMEM Processing-in-DRAM Engine (2019)

- **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



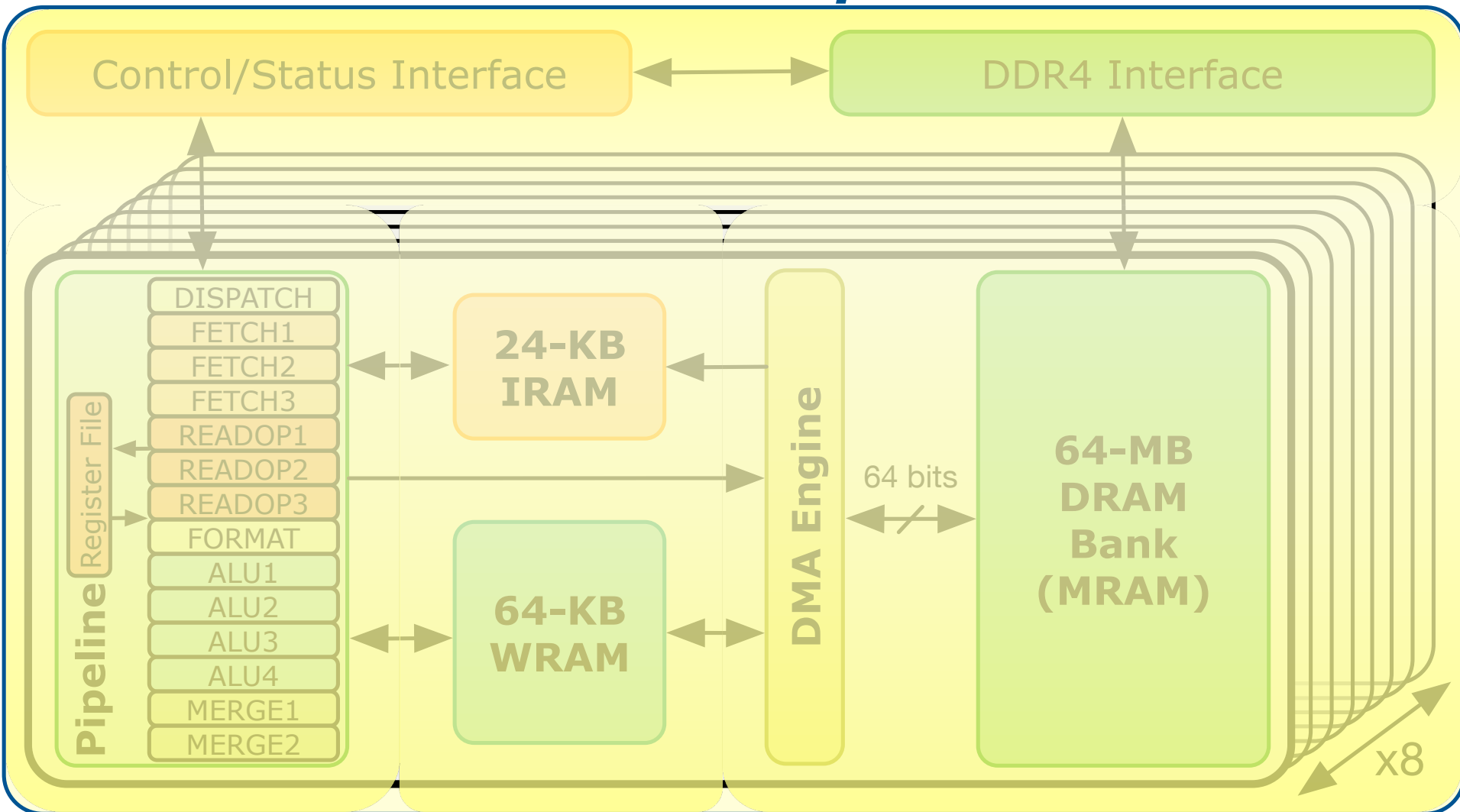
System Organization

- 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



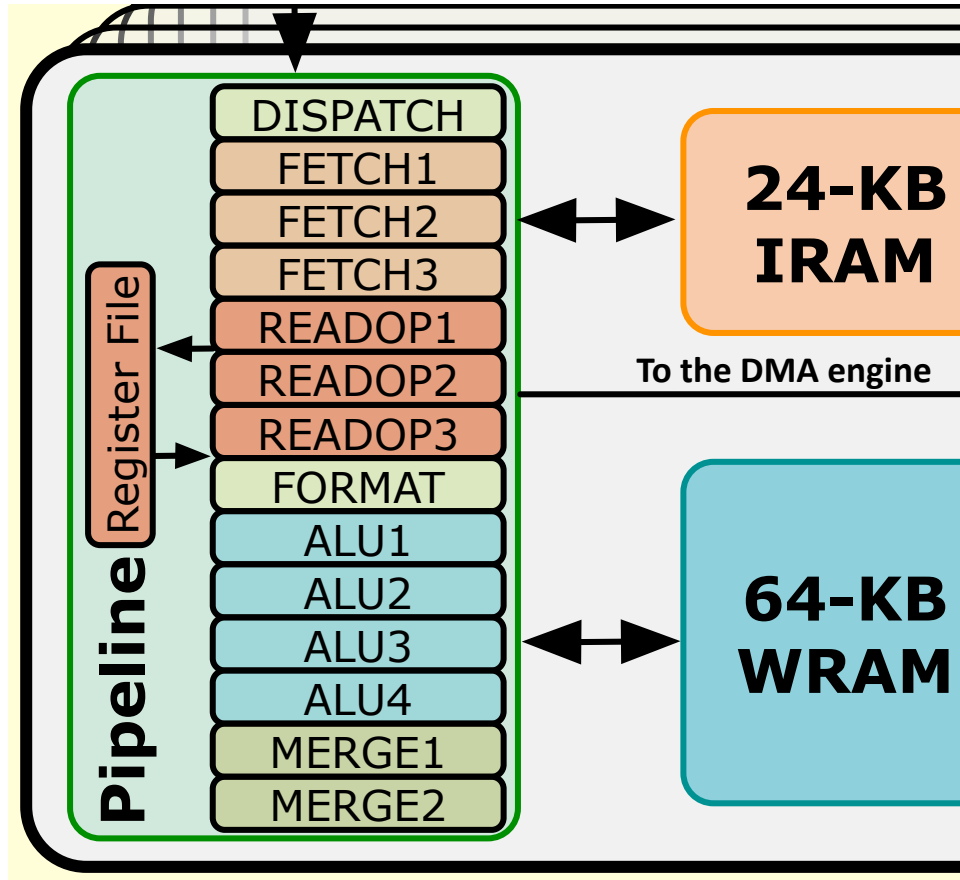
DRAM Processing Unit

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" under the "UPMEM development tools documentation" header. The page includes a navigation menu, a breadcrumb trail, and a "View page source" link. The main content area is titled "Instruction Set Architecture" and contains a paragraph about the architecture concepts, a paragraph about using the section as a reference manual, and a section titled "Resources overview" with a subsection "Thread registers" that describes the system's hardware threads and registers.

u Instruction Set Architecture — UPMEM DPU SDK 2021.2.0 Documentation

UPMEM development tools documentation

» Instruction Set Architecture [View page source](#)

Instruction Set Architecture

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

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

Resources overview

Thread registers

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

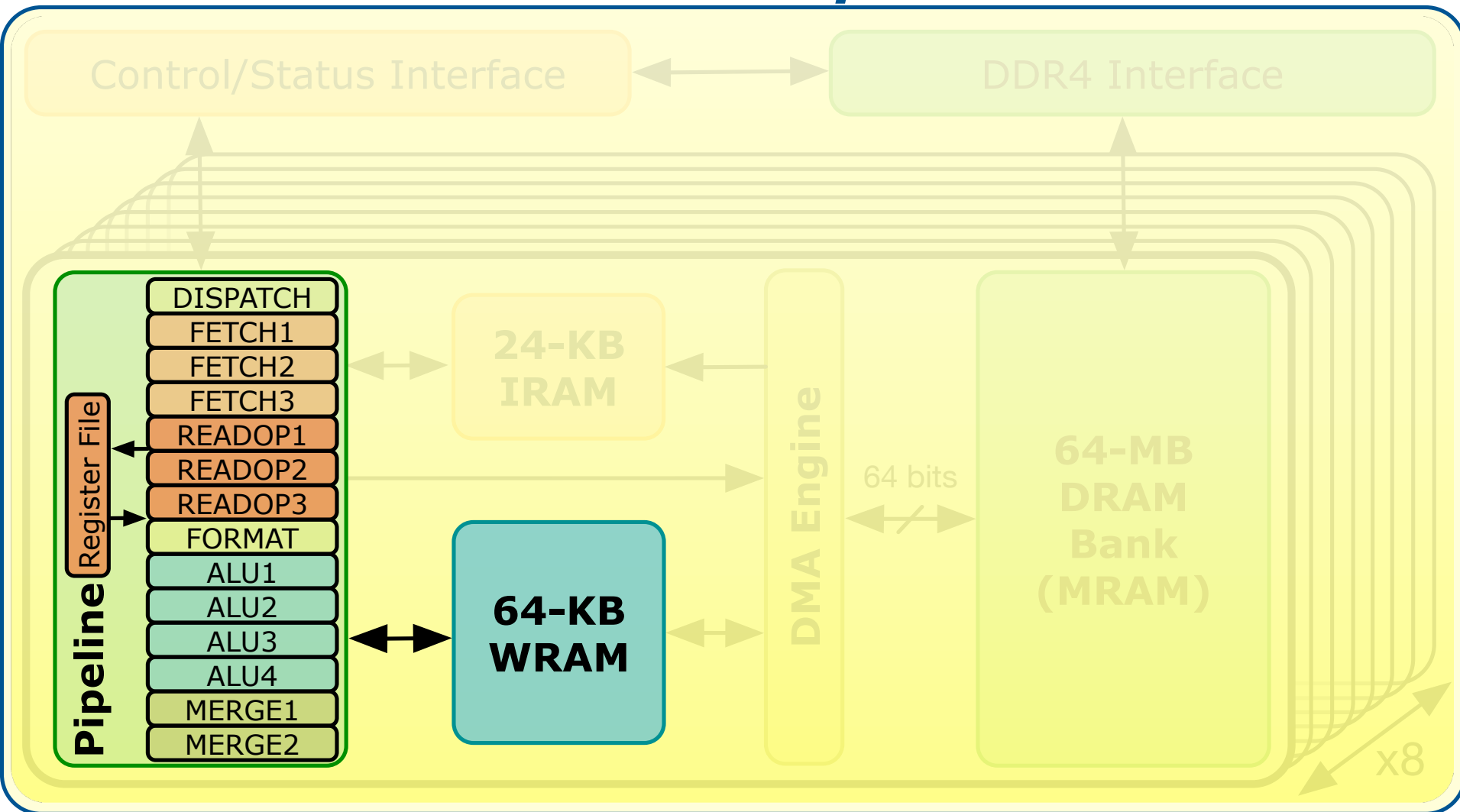
- 24 general purpose 32-bits registers named `r0` through `r23`
- A 16-bits wide program counter, named PC. Notice that the PC value does not address an instruction in memory, but the index of such an instruction directly. For example, a PC equal to 1 represents the second instruction in the DPU's program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

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

Microbenchmarking the UPMEM PIM Architecture

DPU: Arithmetic Throughput

PIM Chip



Arithmetic Throughput: Microbenchmark

- Goal
 - Measure the maximum arithmetic throughput for different datatypes and operations
- Microbenchmark
 - We stream over an array in WRAM and perform read-modify-write operations
 - Experiments on one DPU
 - We vary the number of tasklets from 1 to 24
 - Arithmetic operations: add, subtract, multiply, divide
 - Datatypes: int32, int64, float, double
- We measure cycles with an accurate cycle counter that the SDK provides
 - We include WRAM accesses (including address calculation) and arithmetic operation

Microbenchmark for INT32 ADD Throughput

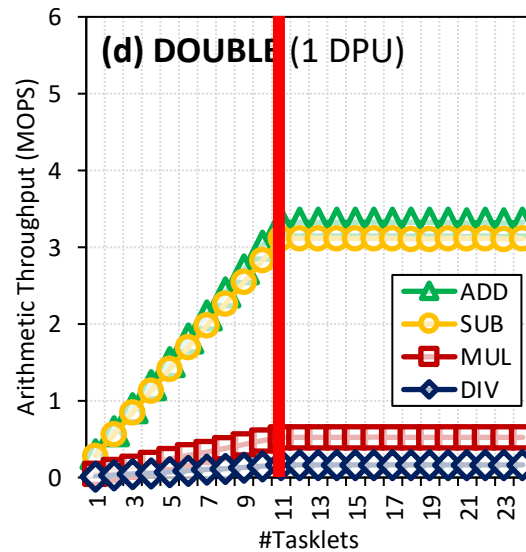
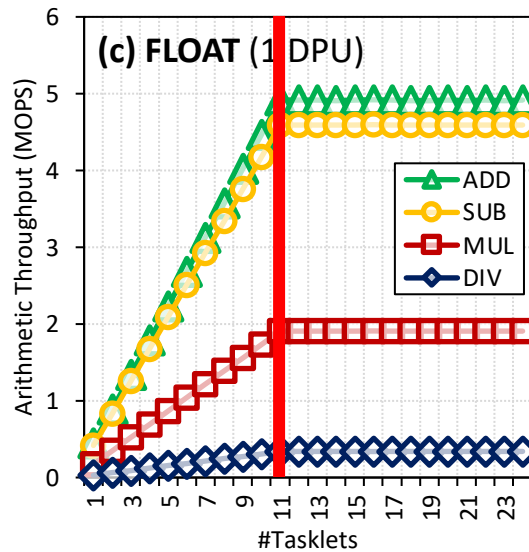
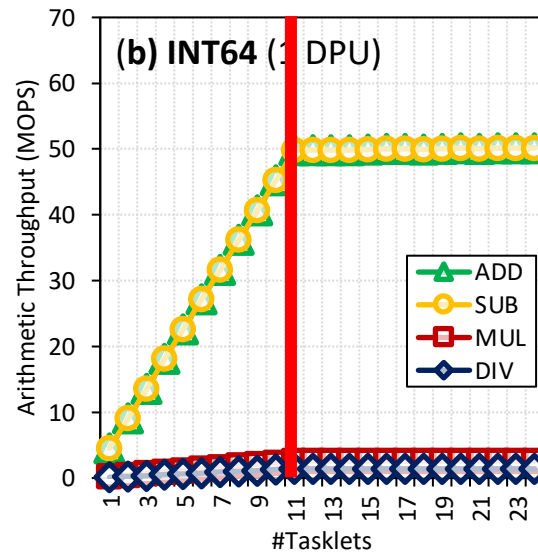
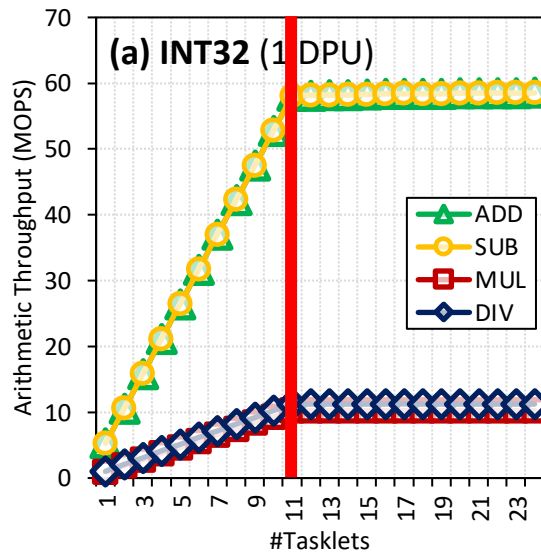
C-based code

```
1  #define SIZE 256
2  int* bufferA = mem_alloc(SIZE * sizeof(int));
3  for(int i = 0; i < SIZE; i++){
4      int temp = bufferA[i];
5      temp += scalar;
6      bufferA[i] = temp;
7  }
```

Compiled code
(UPMEM DPU ISA)

```
1  move r2, 0
2  .LBB0_1:                                // Loop header
3  lsl_add r3, r0, r2, 2                    // Address calculation
4  lw r4, r3, 0                            // Load from WRAM
5  add r4, r4, r1                           // Add
6  sw r3, 0, r4                            // Store to WRAM
7  add r2, r2, 1                           // Index update
8  jneq r2, 256, .LBB0_1                   // Conditional jump
```

Arithmetic Throughput: 11 Tasklets

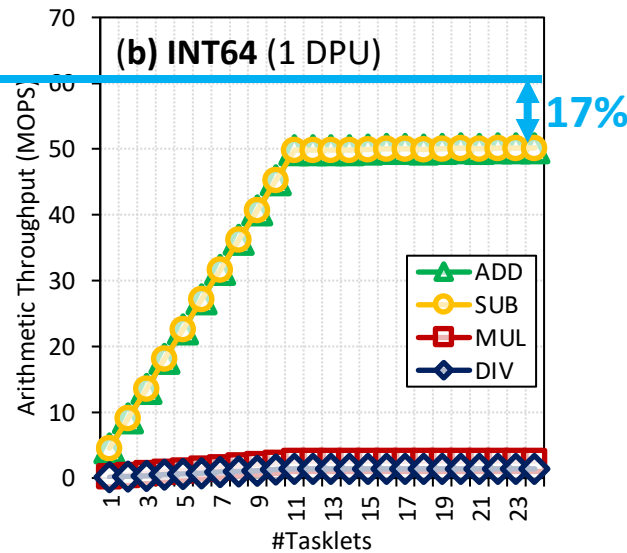
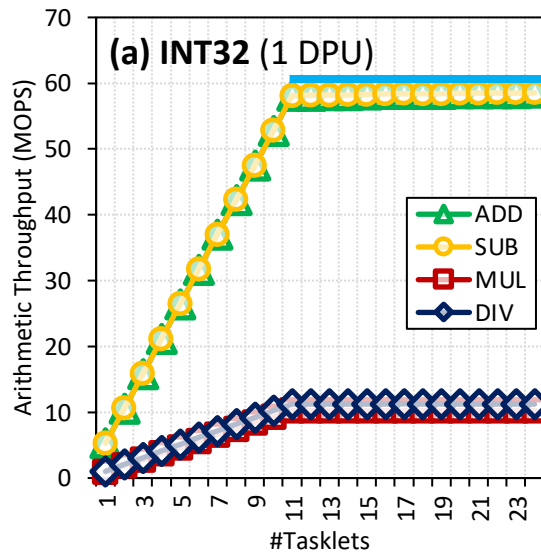


KEY OBSERVATION 1

The arithmetic throughput of a DRAM Processing Unit saturates at 11 or more tasklets.

This observation is consistent for different datatypes (INT32, INT64, UINT32, UINT64, FLOAT, DOUBLE) and operations (ADD, SUB, MUL, DIV).

Arithmetic Throughput: ADD/SUB



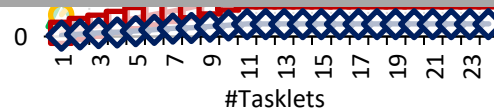
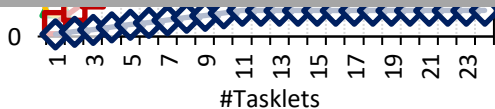
INT32 ADD/SUB are
17% faster than
INT64 ADD/SUB

Can we explain the peak throughput?

Peak throughput at 11 tasklets.

One instruction retires every cycle when the pipeline is full

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



Arithmetic Throughput: #Instructions

- Compiler explorer: <https://dpu.dev>

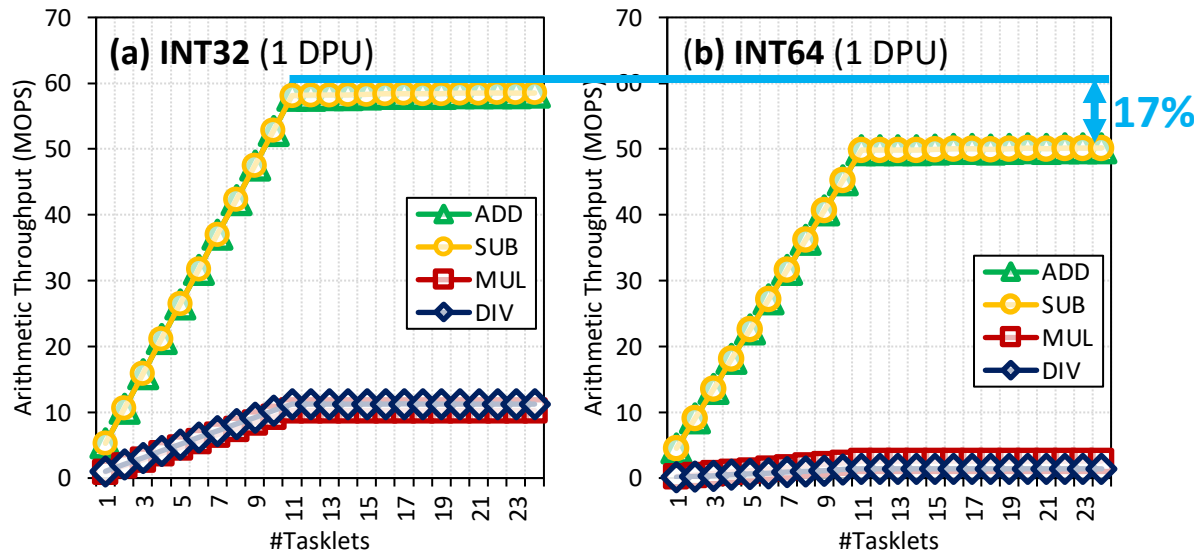
```
1  #define BLOCK_SIZE 1024
2
3  typedef int T;
4  void Benchmark__32bits(T *cache_A, T scalar) {
5      for (int i = 0; i < BLOCK_SIZE / sizeof(T); i++){
6          // WRAM READ
7          T temp = cache_A[i];
8
9          temp += scalar; // ADD
10
11         // WRAM WRITE
12         cache_A[i] = temp;
13     }
14 }
15
16 typedef long T_long;
17 void Benchmark__64bits(T_long *cache_A, T_long scalar) {
18     for (int i = 0; i < BLOCK_SIZE / sizeof(T_long); i++){
19         // WRAM READ
20         T_long temp = cache_A[i];
21
22         temp += scalar; // ADD
23     }
24
25
26
27
```

A ▾ ☐ 11010 ☐ ./a.out ☒ .LX0: ☒ .text ☒ // ☐ \

```
1 Benchmark__32bits:
2     move r2, 0
3 .LBB0_1:
4     lsl_add r3, r0, r2, 2
5     lw r4, r3, 0
6     add r4, r4, r1
7     sw r3, 0, r4
8     add r2, r2, 1
9     jneq r2, 256, .LBB0_1
10    jump r23
11 Benchmark__64bits:
12     move r1, 0
13 .LBB1_1:
14     lsl_add r4, r0, r1, 3
15     ld d6, r4, 0
16     add r7, r7, r3
17     addc r6, r6, r2
18     sd r4, 0, d6
19     add r1, r1, 1
20     jneq r1, 128, .LBB1_1
21    jump r23
```

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

Arithmetic Throughput: ADD/SUB



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

Can we explain the peak throughput?

Peak throughput at 11 tasklets.

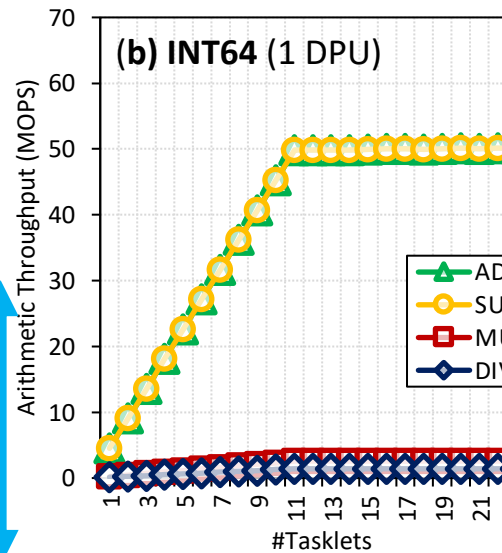
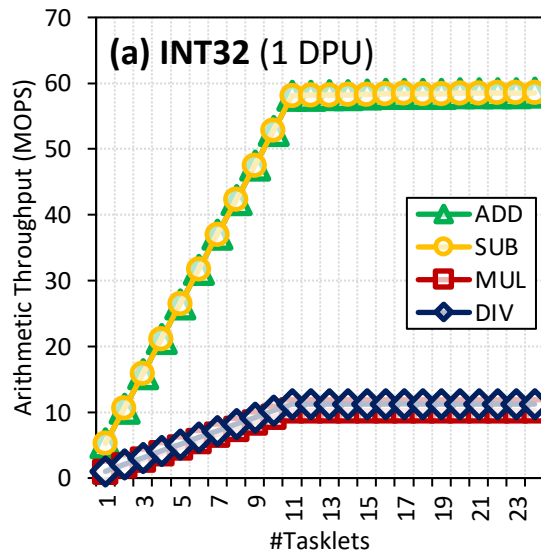
One instruction retires every cycle when the pipeline is full

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

64-bit ADD/SUB: 7 instructions \rightarrow 50.00 MOPS

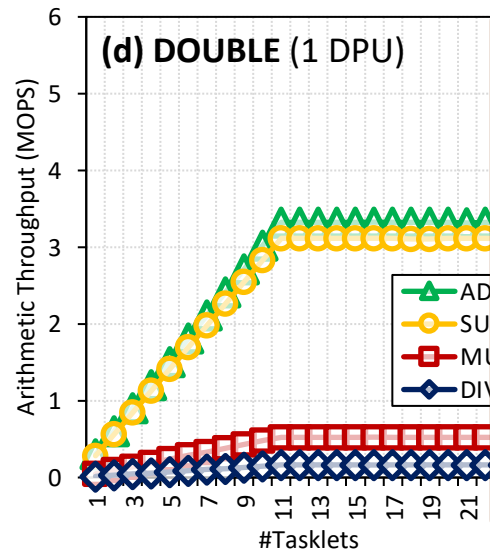
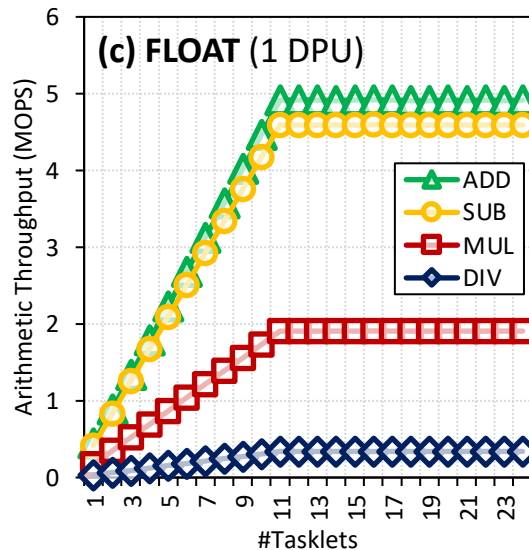
at $\text{frequency}_{DPU} = 350 \text{ MHz}$

Arithmetic Throughput: MUL/DIV



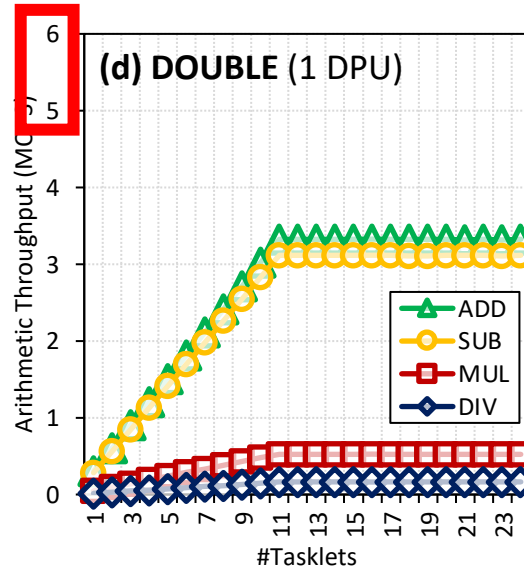
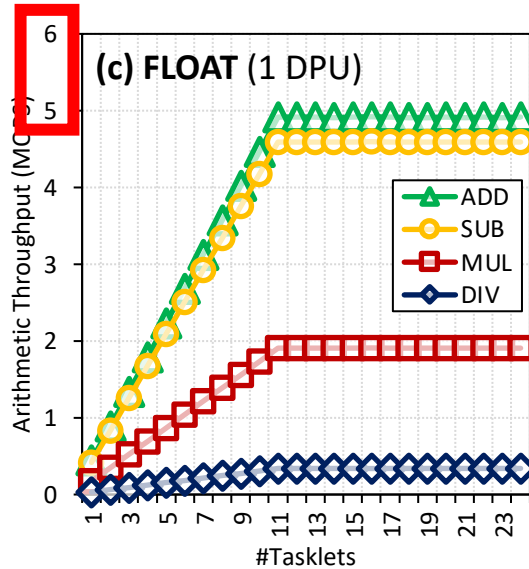
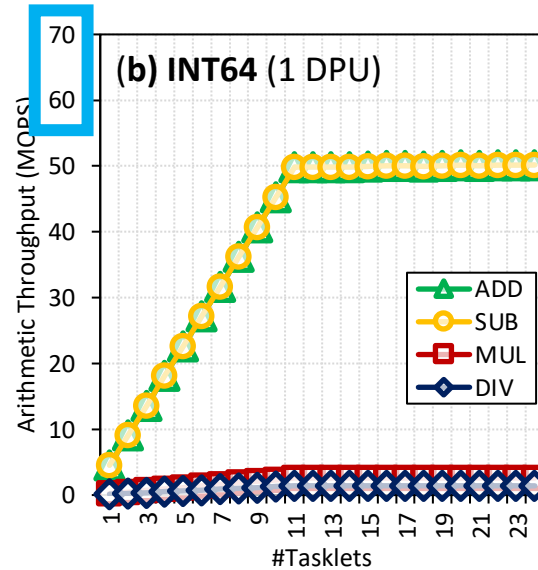
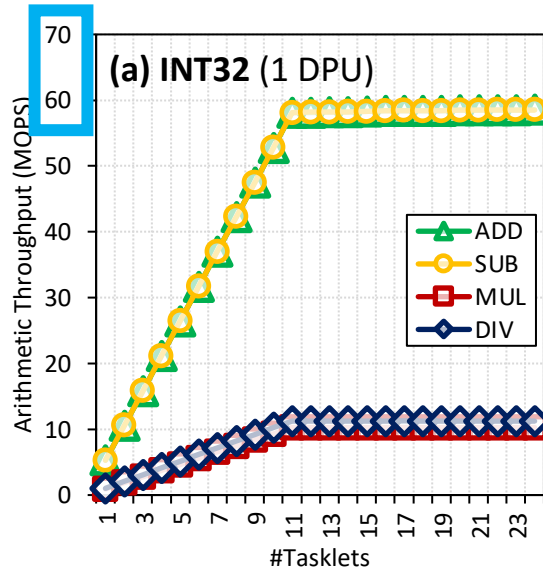
Huge throughput difference between ADD/SUB and MUL/DIV

DPU's do *not* have a 32-bit multiplier



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

Arithmetic Throughput: Native Support



KEY OBSERVATION 2

- DPUs provide **native hardware support for 32- and 64-bit integer addition and subtraction**, leading to high throughput for these operations.
- DPUs do ***not* natively support 32- and 64-bit multiplication and division, and floating point operations**. These operations are **emulated by the UPMEM runtime library**, leading to much lower throughput.

Microbenchmark: Arithmetic Throughput

- Arithmetic throughput for different operations and datatypes

CMU-SAFARI / prim-benchmarks

Unwatch

2

Star

2

Fork

1

<> Code

Issues

Pull requests

Actions

Projects

Wiki

Security

Insights

Settings

main

prim-benchmarks / Microbenchmarks / Arithmetic-Throughput /

Go to file

Add file

...

Juan Gomez Luna PRIM -- first commit

3de4b49 9 days ago

History

..

folder

dpu

PRIM -- first commit

9 days ago

folder

host

PRIM -- first commit

9 days ago

folder

support

PRIM -- first commit

9 days ago

file

Makefile

PRIM -- first commit

9 days ago

file

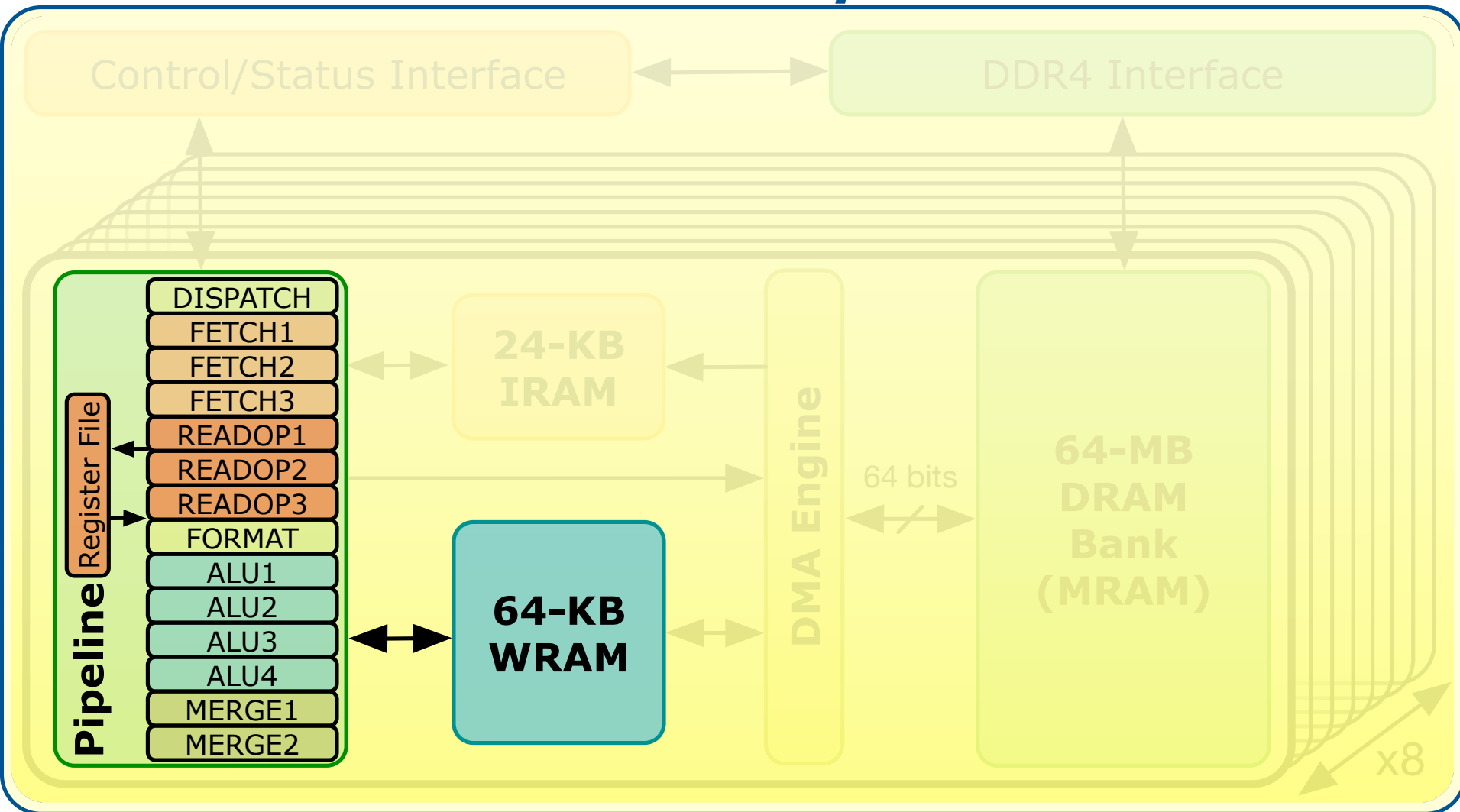
run.sh

PRIM -- first commit

9 days ago

DPU: WRAM Bandwidth

PIM Chip



WRAM Bandwidth: Microbenchmark

- Goal
 - Measure the **WRAM bandwidth** for the STREAM benchmark
- Microbenchmark
 - We implement the four versions of STREAM: **COPY, ADD, SCALE, and TRIAD**
 - The operations performed in ADD, SCALE, and TRIAD are **addition, multiplication, and addition+multiplication**, respectively
 - We vary the number of tasklets from 1 to 16
 - We show results for 1 DPU
- We do not include accesses to MRAM

STREAM Benchmark in WRAM

// COPY

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

8 bytes read, 8 bytes written,
no arithmetic operations

// ADD

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

16 bytes read, 8 bytes written,
ADD

// SCALE

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

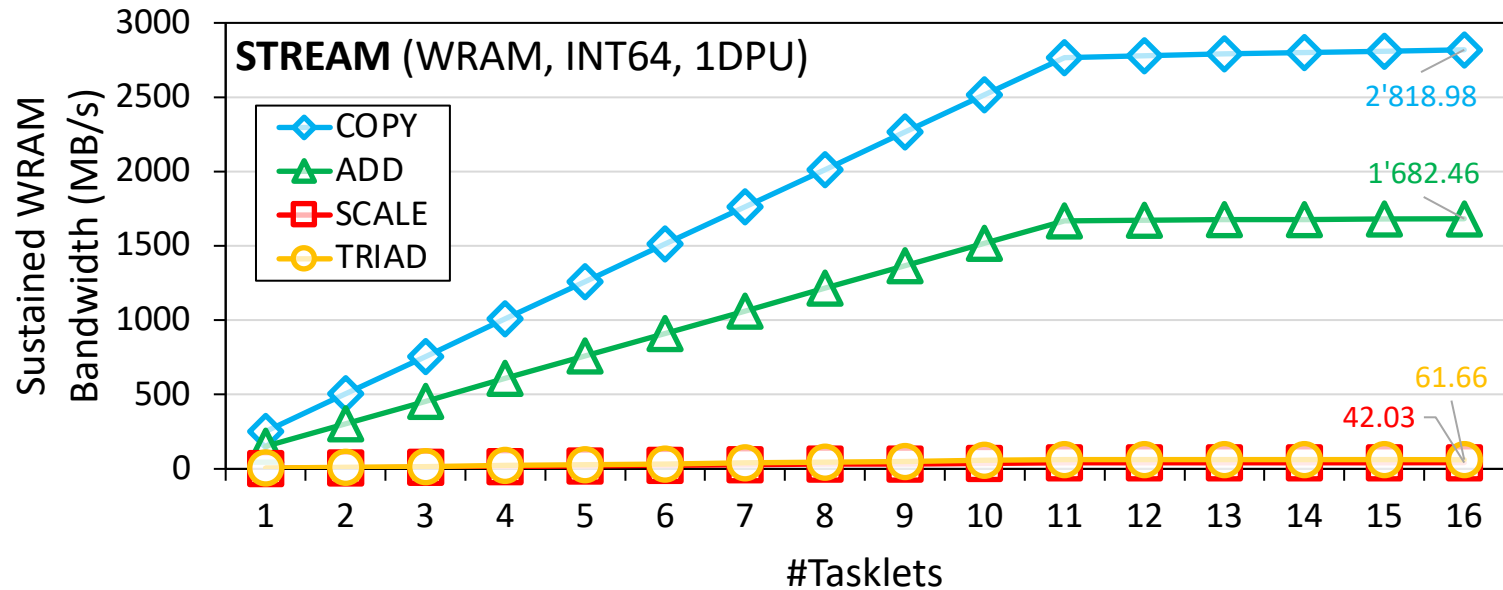
8 bytes read, 8 bytes written,
MUL

// TRIAD

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

16 bytes read, 8 bytes written,
MUL, ADD

WRAM Bandwidth: STREAM

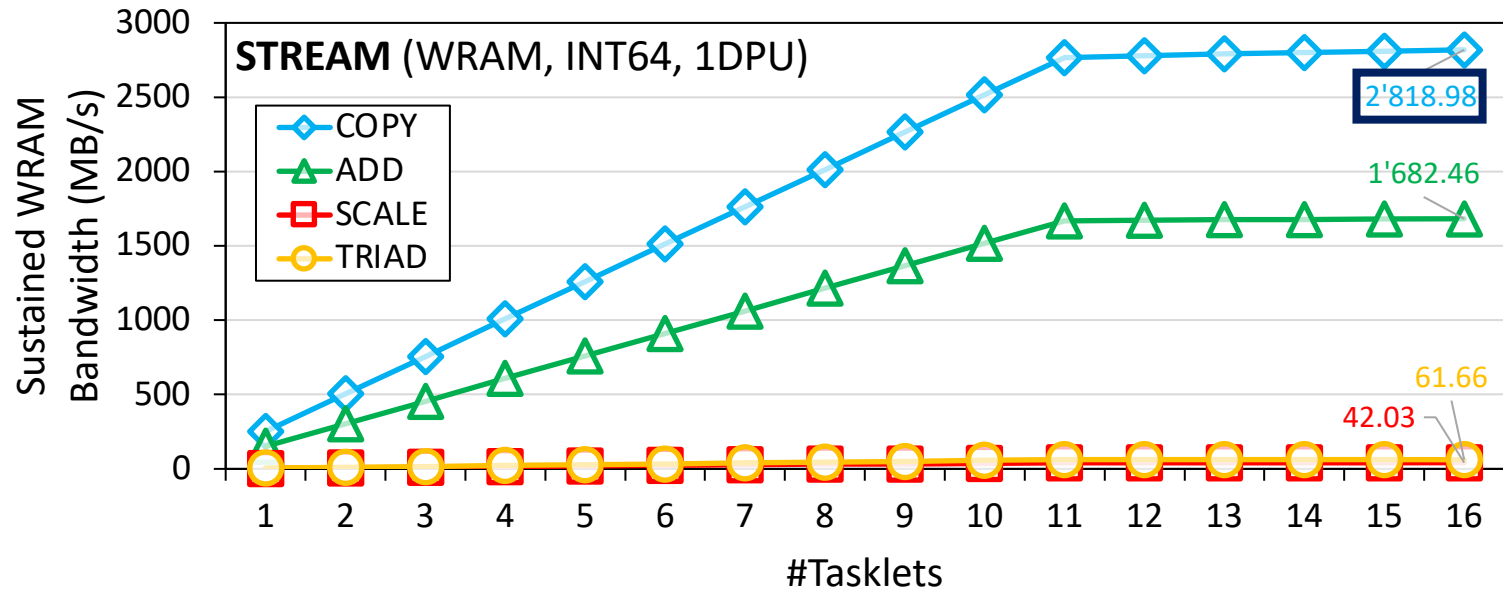


How can we estimate the bandwidth?

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

$$WRAM\ Bandwidth\ \left(in\ \frac{B}{S}\right) = \frac{Bytes \times frequency_{DPU}}{\#instructions}$$

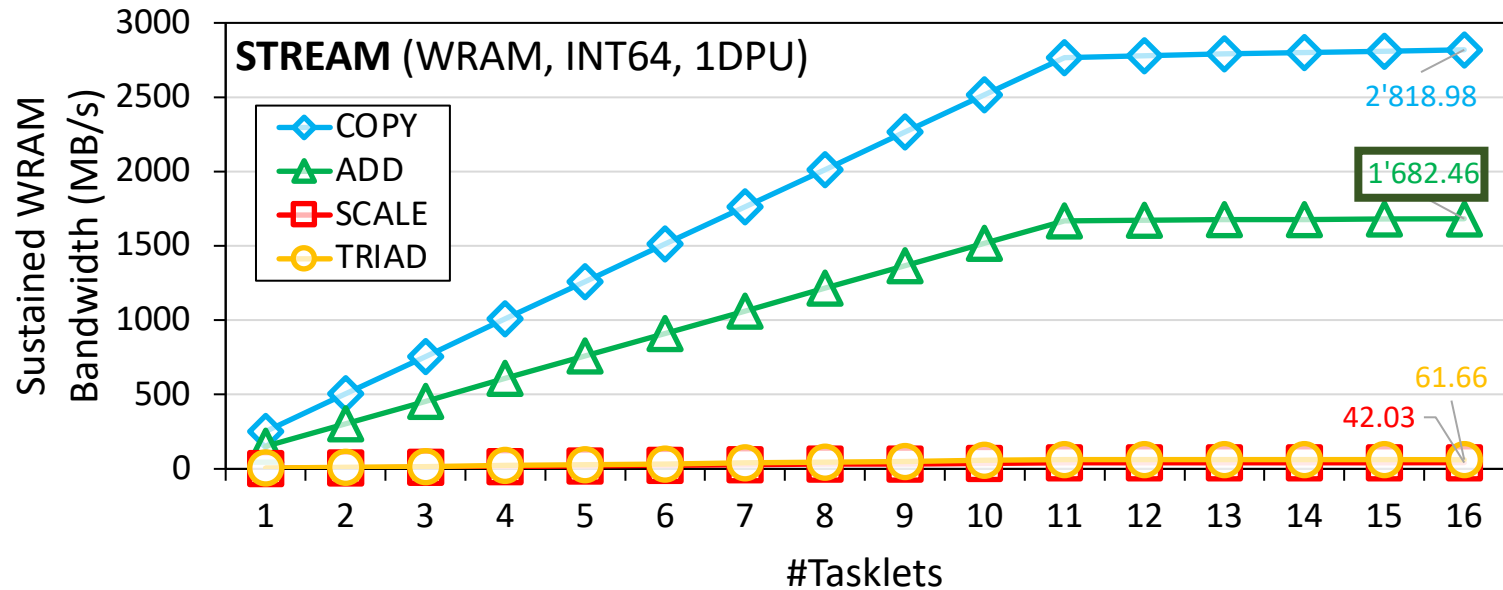
WRAM Bandwidth: COPY



COPY executes **2 instructions** (WRAM load and store).
With 11 tasklets, **11 × 16 bytes** in **22 cycles**:

$$\text{WRAM Bandwidth} \left(\text{in } \frac{B}{S} \right) = 2,800 \frac{MB}{s} \text{ at } 350 \text{ MHz}$$

WRAM Bandwidth: ADD



$$WRAM \text{ Bandwidth } \left(in \frac{B}{S} \right) = \frac{Bytes \times frequency_{DPU}}{\#instructions}$$

ADD executes 5 instructions (2 ld, add, addc, sd).
With 11 tasklets, 11×24 bytes in 55 cycles:

$$WRAM \text{ Bandwidth } \left(in \frac{B}{S} \right) = 1,680 \frac{MB}{s} \text{ at } 350 \text{ MHz}$$

WRAM Bandwidth: Access Patterns

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

KEY OBSERVATION 3

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

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

- Microbenchmark: `c[a[i]]=b[a[i]];`
 - Unit-stride: `a[i]=a[i-1]+1;`
 - Strided: `a[i]=a[i-1]+stride;`
 - Random: `a[i]=rand();`

Microbenchmark: STREAM and WRAM

- STREAM benchmark and WRAM access patterns

CMU-SAFARI / prim-benchmarks

Unwatch 2

Star 2

Fork 1

<> Code

Issues

Pull requests

Actions

Projects

Wiki

Security

Insights

Settings

main

prim-benchmarks / Microbenchmarks / STREAM /

Go to file

Add file

main

prim-benchmarks / Microbenchmarks / WRAM /

Go to file

Add file

Juan Gomez Luna PRIM -- first commit

3de4b49 9 days ago History

..

dpudpuPRIM -- first commit9 days ago

hosthostPRIM -- first commit9 days ago

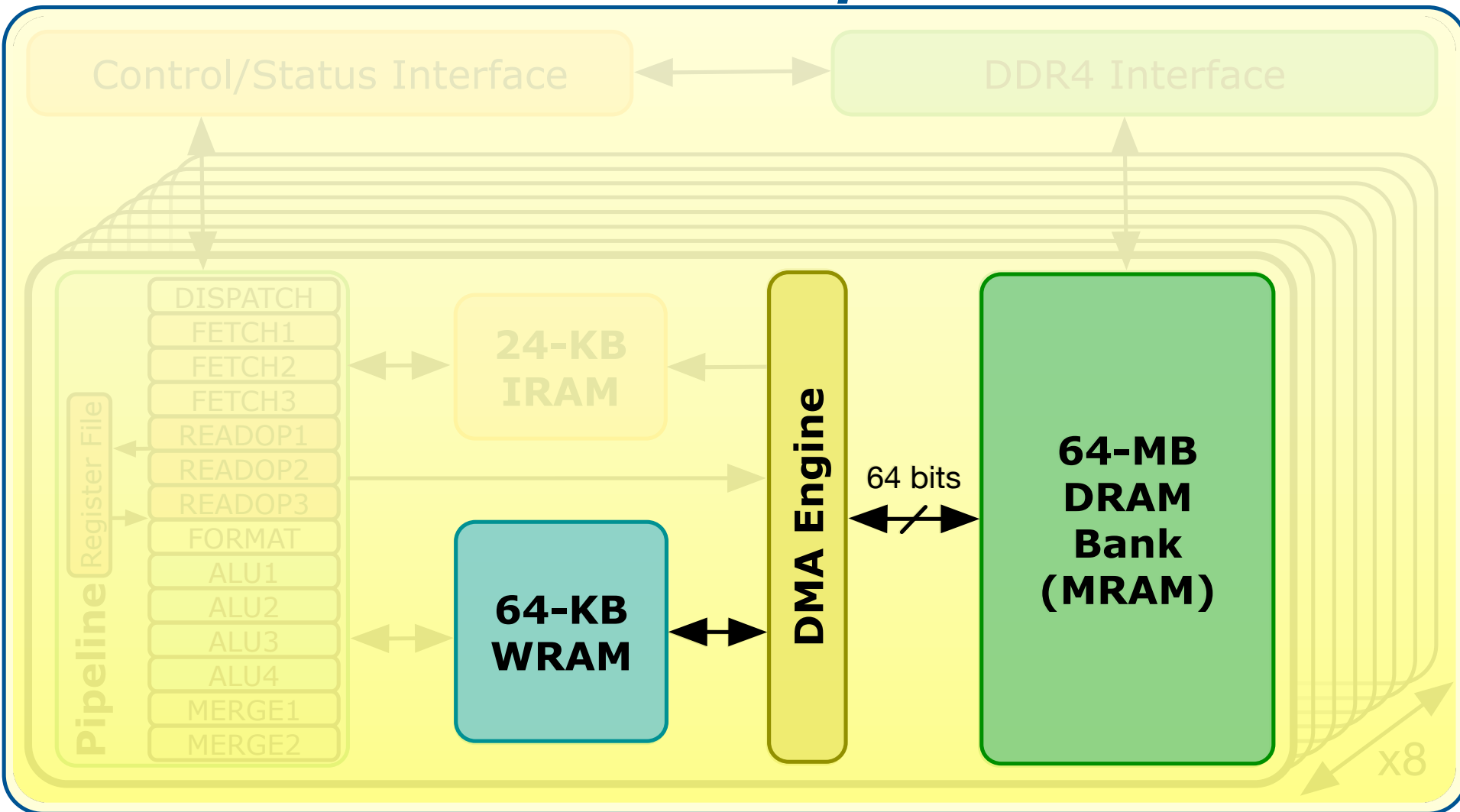
supportsupportPRIM -- first commit9 days ago

MakefileMakefilePRIM -- first commit9 days ago

run.shrun.shPRIM -- first commit9 days ago

DPU: MRAM Latency and Bandwidth

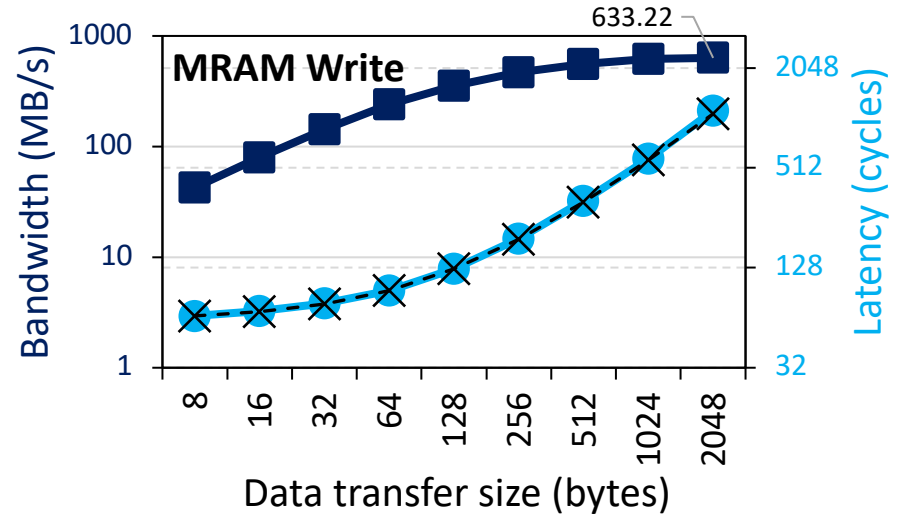
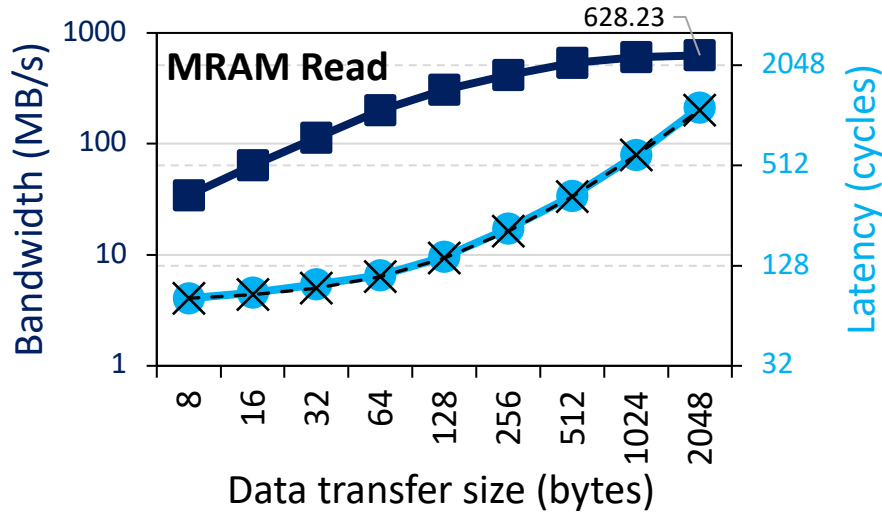
PIM Chip



MRAM Bandwidth

- Goal
 - Measure **MRAM bandwidth** for different access patterns
- Microbenchmarks
 - **Latency of a single DMA transfer** for different transfer sizes
 - `mram_read();` // MRAM-WRAM DMA transfer
 - `mram_write();` // WRAM-MRAM DMA transfer
 - **STREAM** benchmark
 - COPY, COPY-DMA
 - ADD, SCALE, TRIAD
 - **Strided** access pattern
 - Coarse-grain strided access
 - Fine-grain strided access
 - **Random** access pattern (GUPS)
- We do include accesses to MRAM

MRAM Read and Write Latency (I)



$$MRAM \text{ Bandwidth } \left(\text{in } \frac{B}{S} \right) = \frac{\text{size} \times \text{frequency}_{DPU}}{MRAM \text{ Latency}}$$

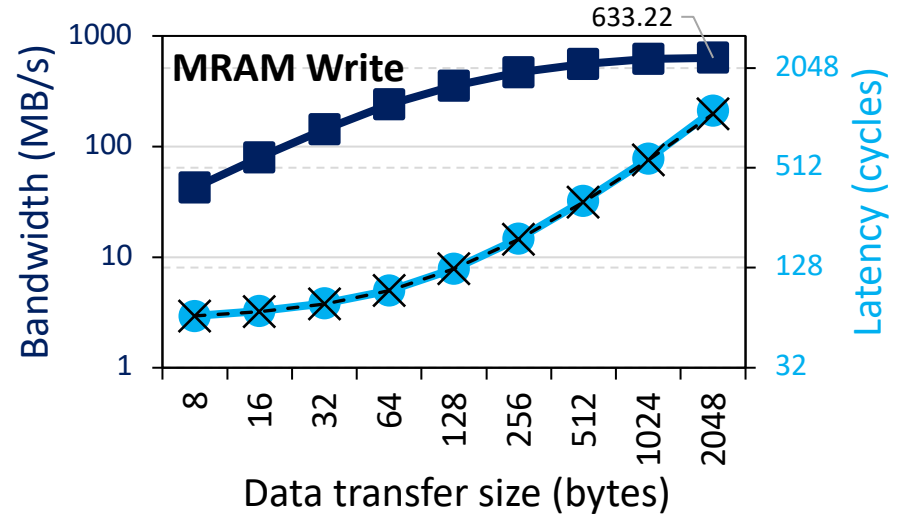
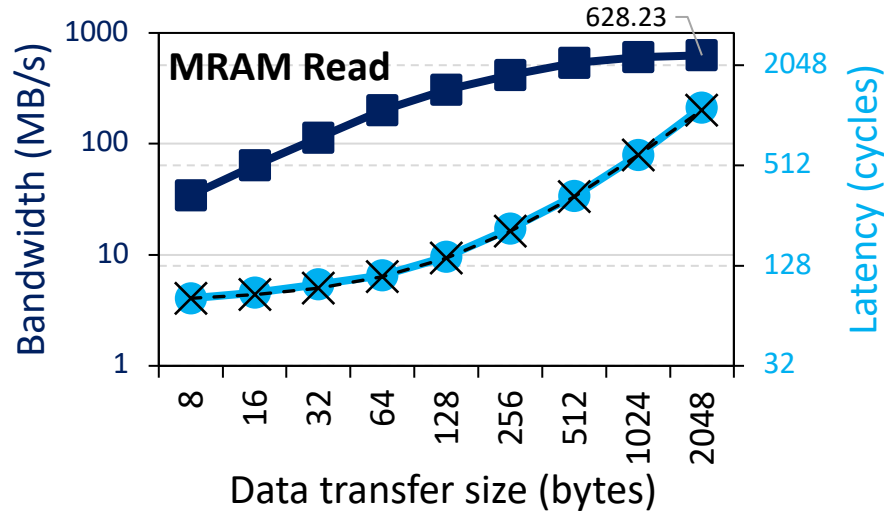
We can model the MRAM latency with a linear expression

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

In our measurements, β equals 0.5 cycles/byte.

Theoretical maximum MRAM bandwidth = 700 MB/s at 350 MHz

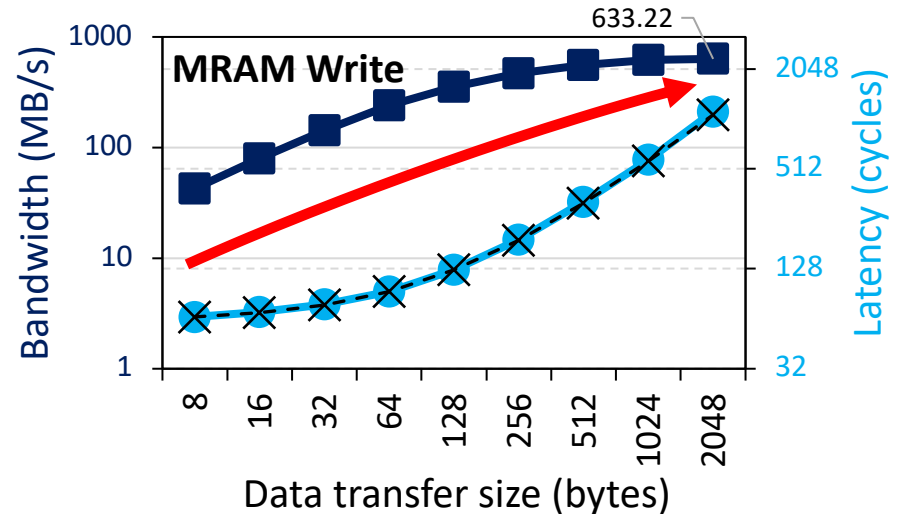
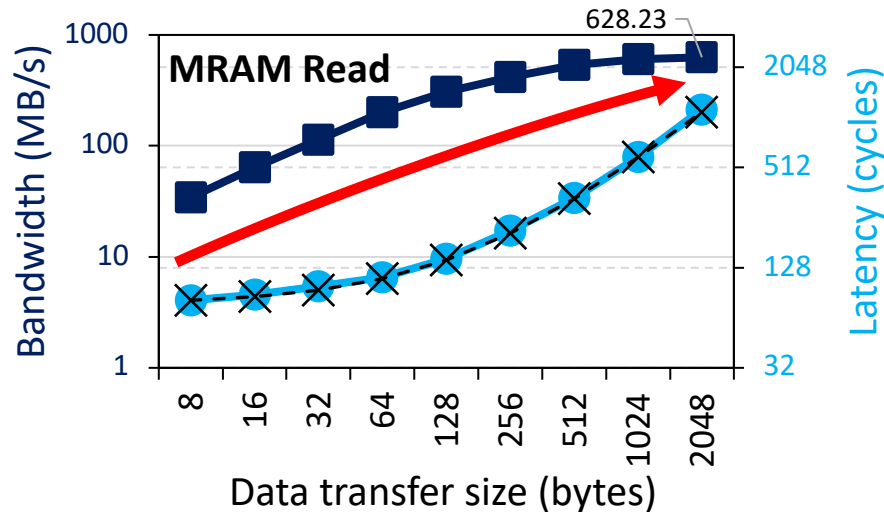
MRAM Read and Write Latency (II)



KEY OBSERVATION 4

- The DPU's Main memory (MRAM) bank access latency increases **linearly** with the transfer size.
- The maximum theoretical MRAM **bandwidth** is 2 bytes per cycle.

MRAM Read and Write Latency (III)



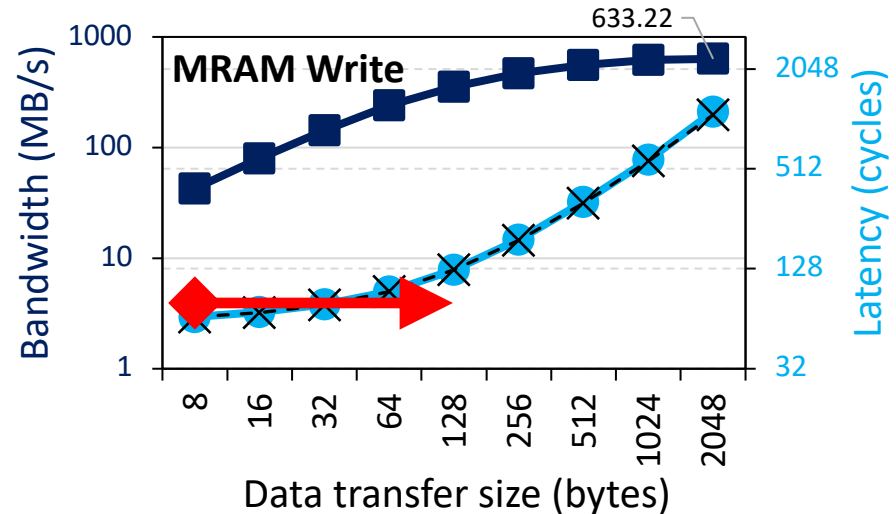
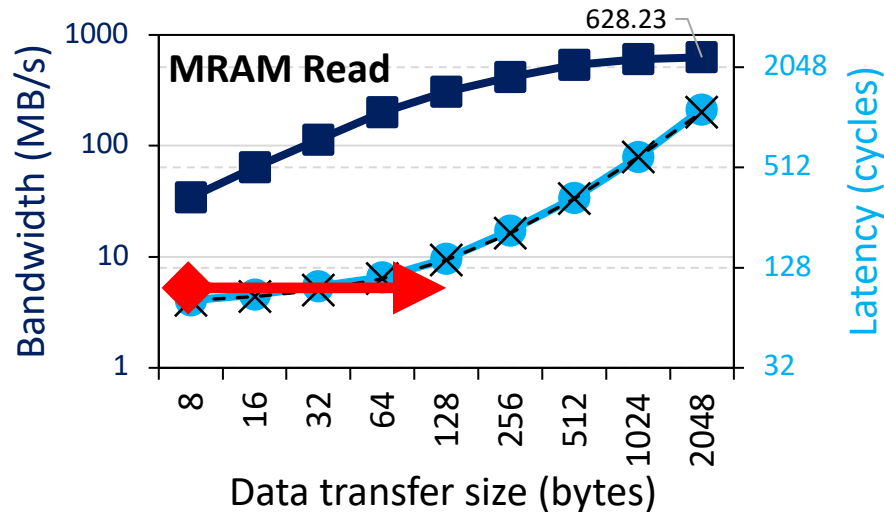
Read and write accesses to MRAM are symmetric

The sustained MRAM bandwidth increases with data transfer size

PROGRAMMING RECOMMENDATION 1

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

MRAM Read and Write Latency (IV)



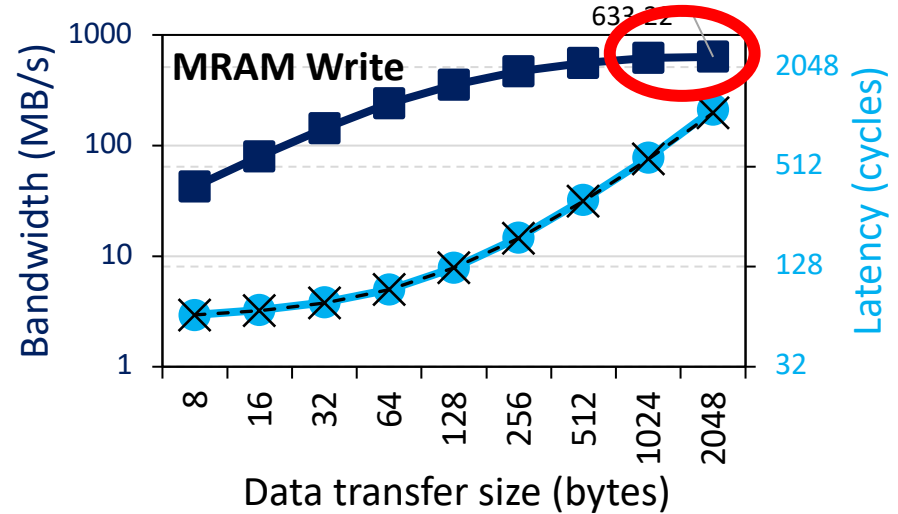
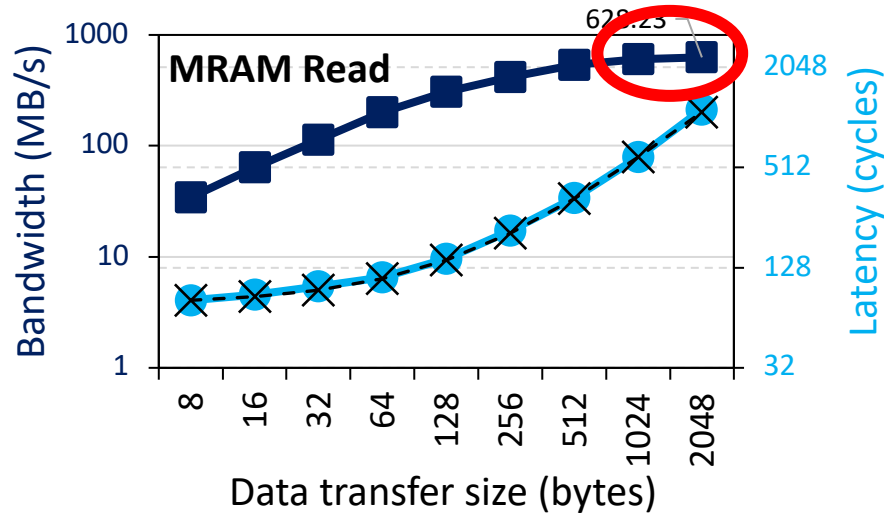
MRAM latency changes slowly between 8 and 128 bytes

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

PROGRAMMING RECOMMENDATION 2

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

MRAM Read and Write Latency (V)



2,048-byte transfers are only 4% faster than 1,024-byte transfers

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

PROGRAMMING RECOMMENDATION 3

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

MRAM Bandwidth

- Goal
 - Measure **MRAM bandwidth** for different access patterns
- Microbenchmarks
 - **Latency of a single DMA transfer** for different transfer sizes
 - `mram_read();` // MRAM-WRAM DMA transfer
 - `mram_write();` // WRAM-MRAM DMA transfer
 - **STREAM** benchmark
 - COPY, COPY-DMA
 - ADD, SCALE, TRIAD
 - **Strided** access pattern
 - Coarse-grain strided access
 - Fine-grain strided access
 - **Random** access pattern (GUPS)
- We do include accesses to MRAM

STREAM Benchmark in MRAM

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

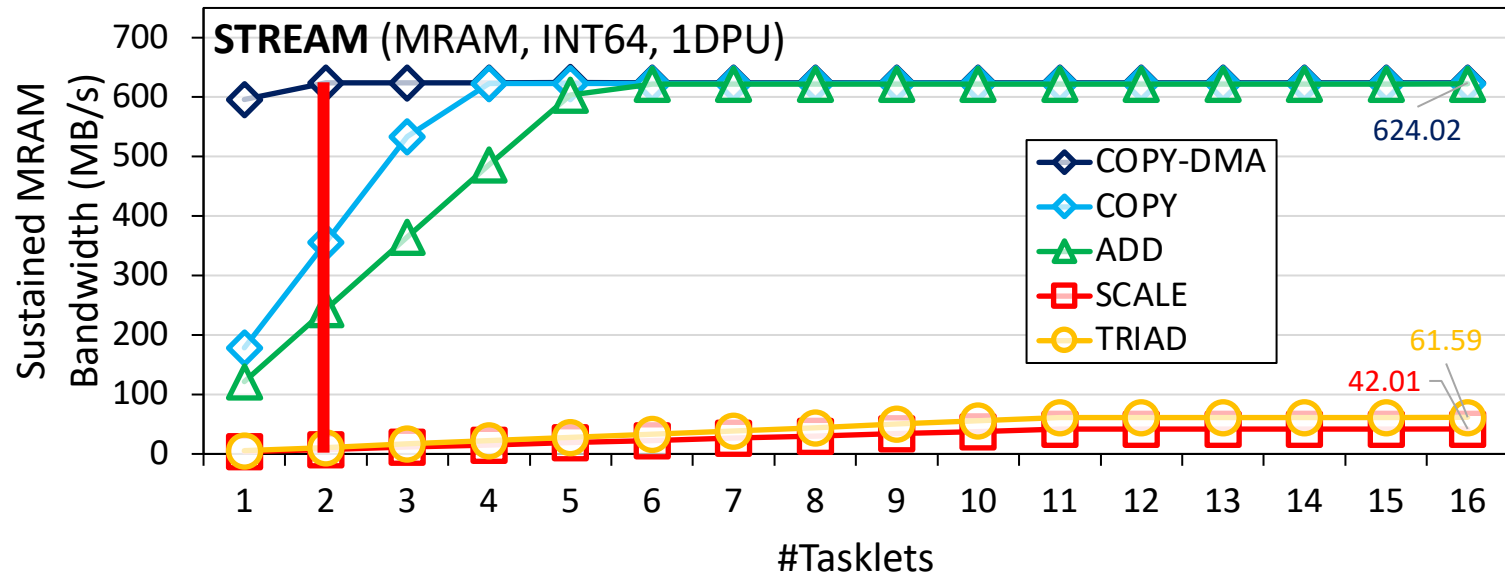
for(int i = 0; i < SIZE; i++){
    bufferB[i] = bufferA[i];
}

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

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

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

STREAM Benchmark: COPY-DMA

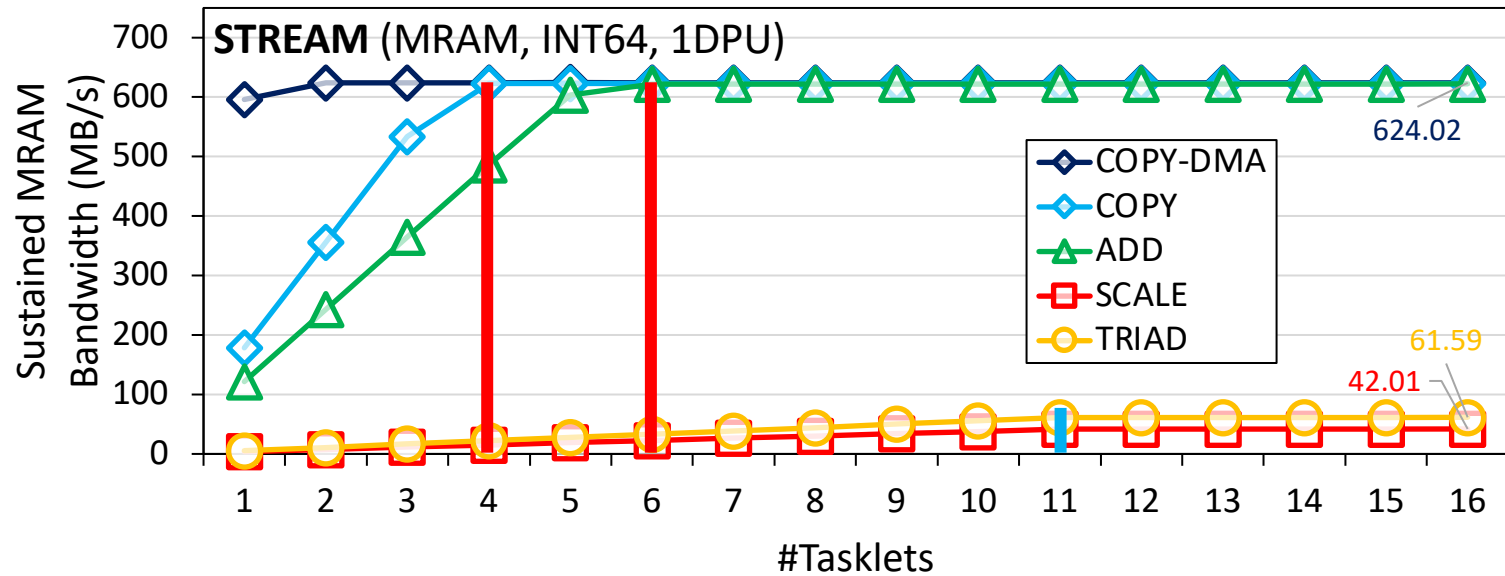


The sustained bandwidth of **COPY-DMA** is close to the theoretical maximum (700 MB/s): **~1.6 TB/s for 2,556 DPUs**

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

Using **two or more tasklets** guarantees that there is always a DMA request enqueued to keep the DMA engine busy

STREAM Benchmark: Bandwidth Saturation (I)



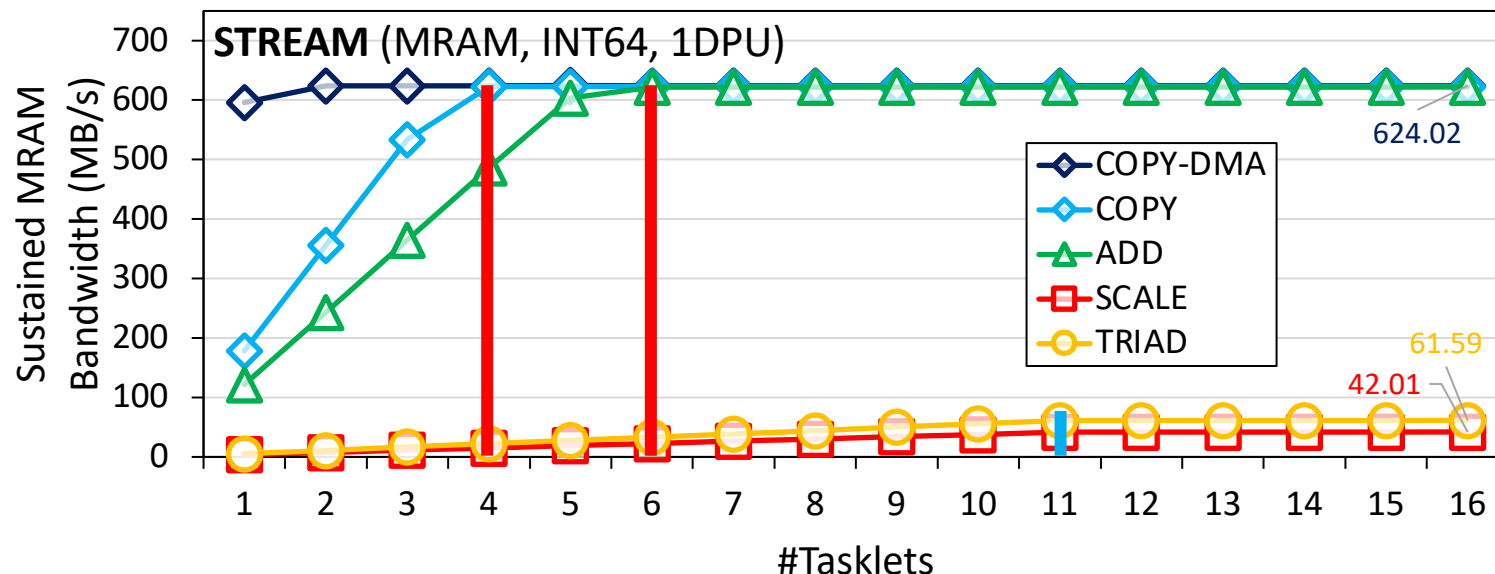
COPY and **ADD** saturate at **4** and **6** tasklets, respectively

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

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

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

STREAM Benchmark: Bandwidth Saturation (II)



KEY OBSERVATION 5

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

MRAM Bandwidth

- Goal
 - Measure **MRAM bandwidth** for different access patterns
- Microbenchmarks
 - **Latency of a single DMA transfer** for different transfer sizes
 - `mram_read();` // MRAM-WRAM DMA transfer
 - `mram_write();` // WRAM-MRAM DMA transfer
 - **STREAM** benchmark
 - COPY, COPY-DMA
 - ADD, SCALE, TRIAD
 - **Strided** access pattern
 - Coarse-grain strided access
 - Fine-grain strided access
 - **Random** access pattern (GUPS)
- We do include accesses to MRAM

Strided and Random Access to MRAM

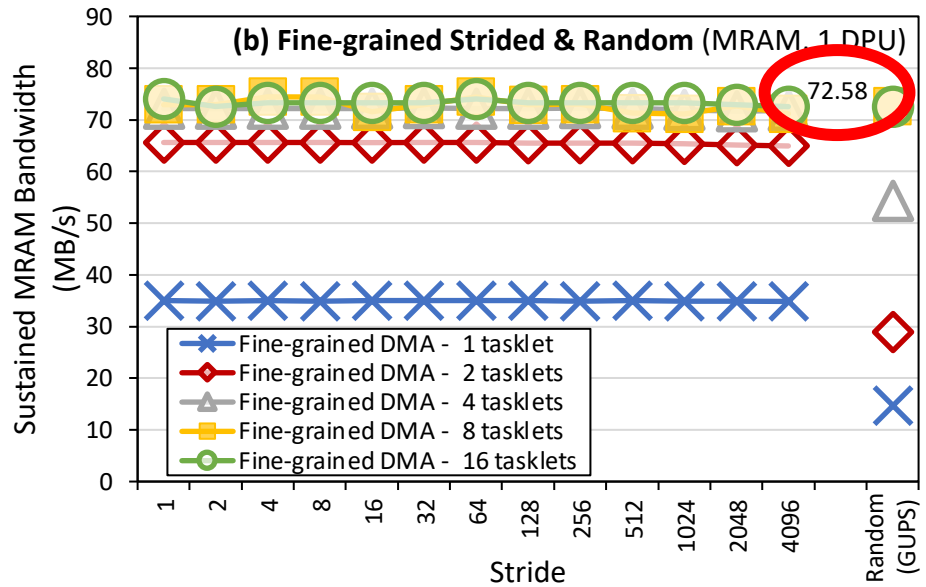
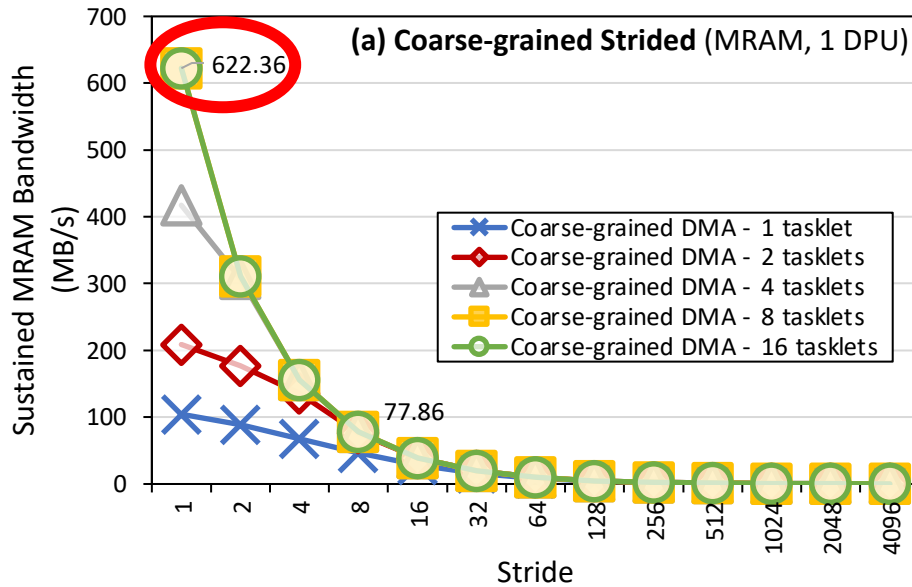
```
// COARSE-GRAINED STRIDED ACCESS
// Load current MRAM block to WRAM
mram_read((__mram_ptr void const*)mram_address_A, bufferA,
          SIZE * sizeof(uint64_t));
mram_read((__mram_ptr void const*)mram_address_B, bufferB,
          SIZE * sizeof(uint64_t));

for(int i = 0; i < SIZE; i += stride){
    bufferB[i] = bufferA[i];
}
// Write WRAM block to MRAM
mram_write(bufferB, (__mram_ptr void*)mram_address_B,
          SIZE * sizeof(uint64_t));

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

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

Strided and Random Accesses (I)

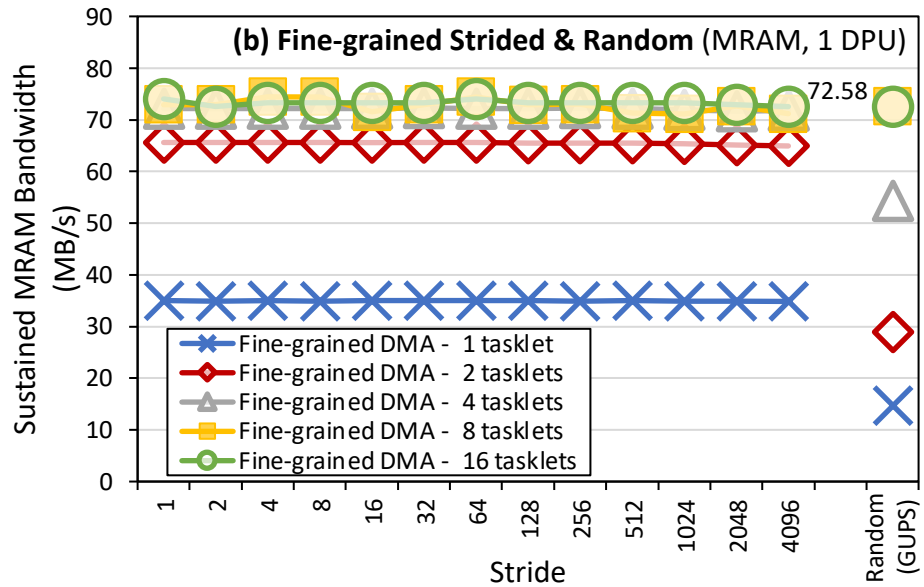
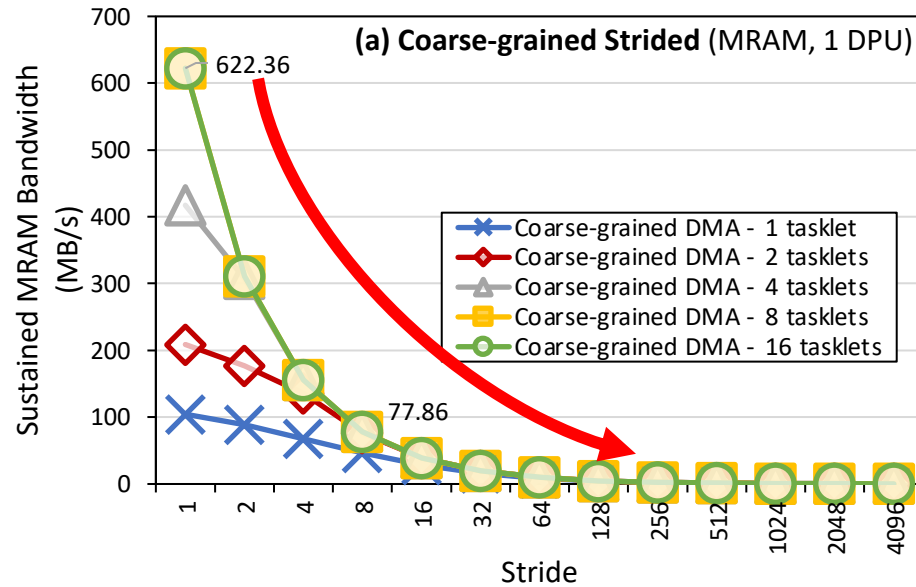


Large difference in maximum sustained bandwidth between coarse-grained and fine-grained DMA

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

Random access achieves very similar maximum sustained bandwidth to fine-grained strided approach

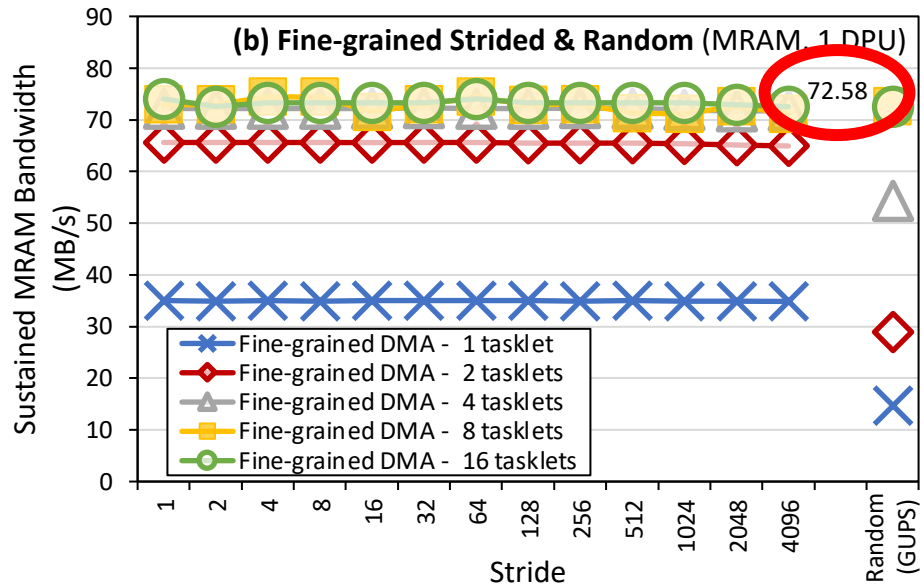
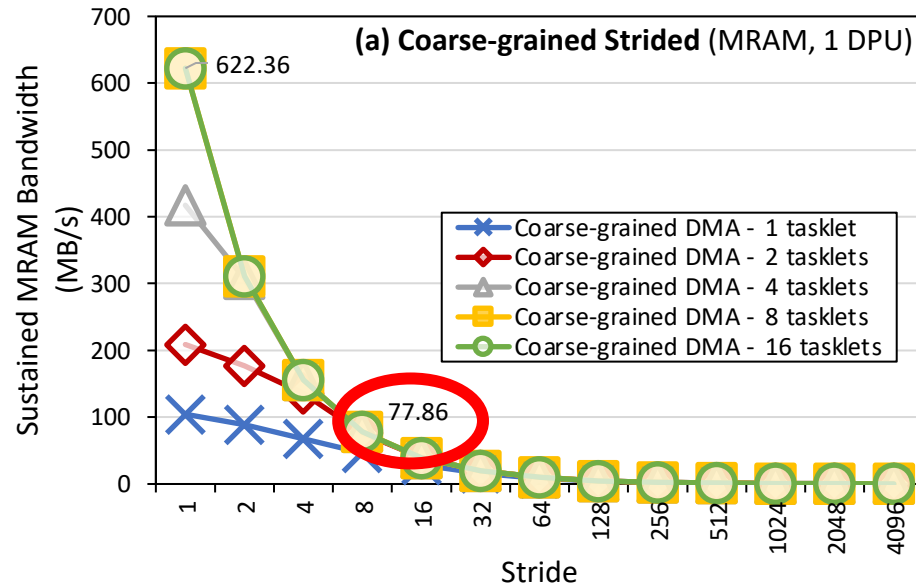
Strided and Random Accesses (II)



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

The effective utilization of the transferred data decreases as the stride becomes larger (e.g., a stride 4 means that only one fourth of the transferred data is used)

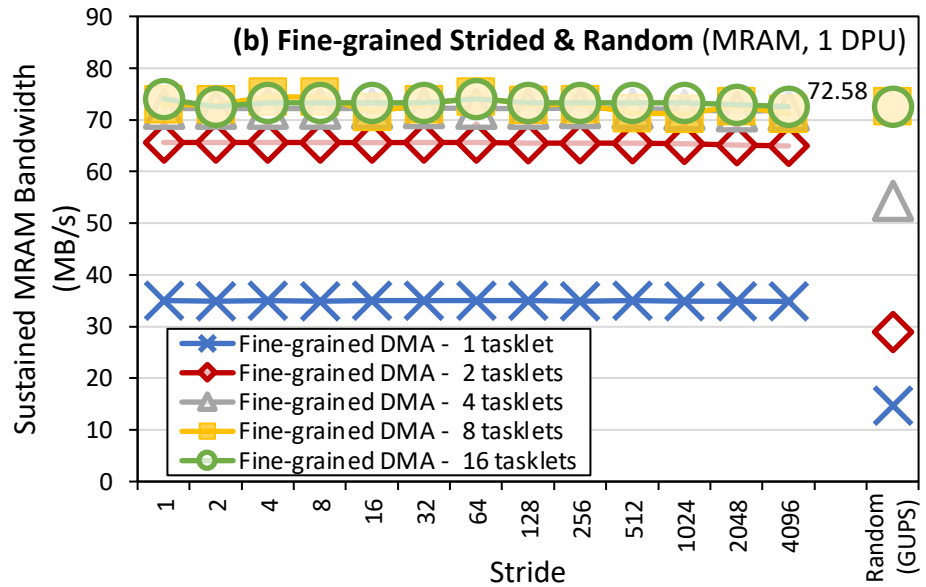
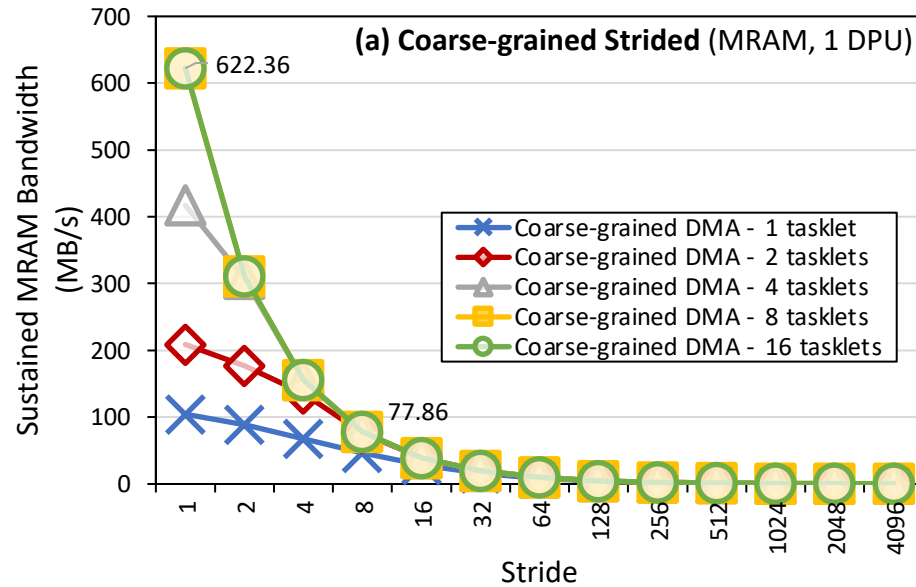
Strided and Random Accesses (III)



For a stride of 16 or larger, the fine-grained DMA approach achieves higher bandwidth

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

Strided and Random Accesses (IV)




PROGRAMMING RECOMMENDATION 4

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

Microbenchmark: Strided and Random

- Strided and random accesses to MRAM

 [CMU-SAFARI](#) / [prim-benchmarks](#)

Unwatch 2

Star 2

Fork 1

[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#) [Insights](#) [Settings](#)

main

[prim-benchmarks](#) / [Microbenchmarks](#) / [STRIDED](#) /

Go to file Add file ...

main

[prim-benchmarks](#) / [Microbenchmarks](#) / [Random-GUPS](#) /

Go to file Add file ...

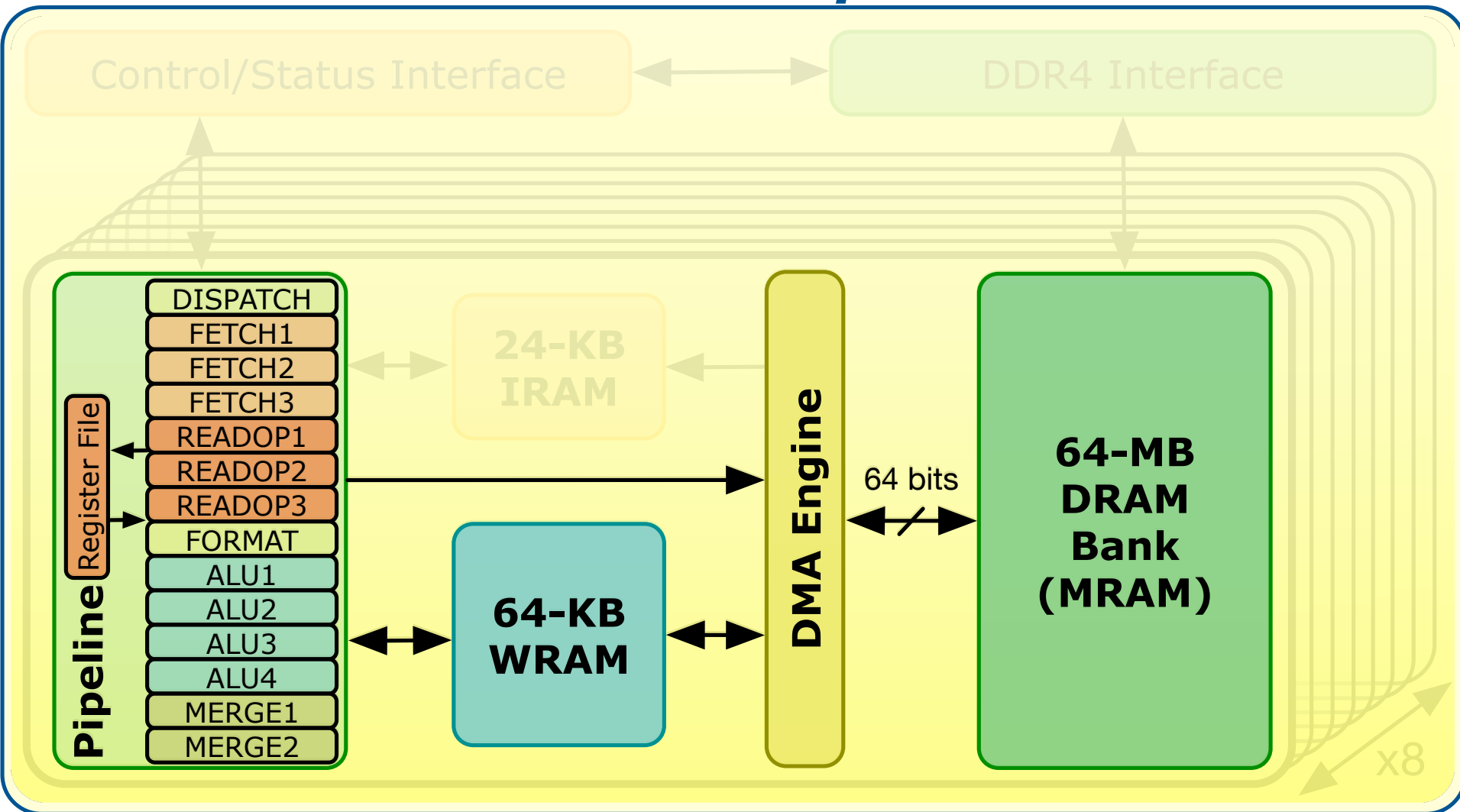
Juan Gomez Luna PRIM -- first commit 3de4b49 9 days ago History

..

dp	PRIM -- first commit	9 days ago
host	PRIM -- first commit	9 days ago
support	PRIM -- first commit	9 days ago
Makefile	PRIM -- first commit	9 days ago
run.sh	PRIM -- first commit	9 days ago

DPU: Arithmetic Throughput vs. Operational Intensity

PIM Chip



Arithmetic Throughput vs. Operational Intensity (I)

- Goal
 - Characterize **memory-bound regions** and **compute-bound regions** for different datatypes and operations
- Microbenchmark
 - We **load one chunk of an MRAM array into WRAM**
 - Perform a **variable number of operations on the data**
 - **Write back** to MRAM
- The experiment is inspired by the **Roofline model***
- We define **operational intensity** (OI) as the number of arithmetic operations performed per byte accessed from MRAM (OP/B)
- The pipeline latency changes with the operational intensity, but the MRAM access latency is fixed

Arithmetic Throughput vs. Operational Intensity (II)

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

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

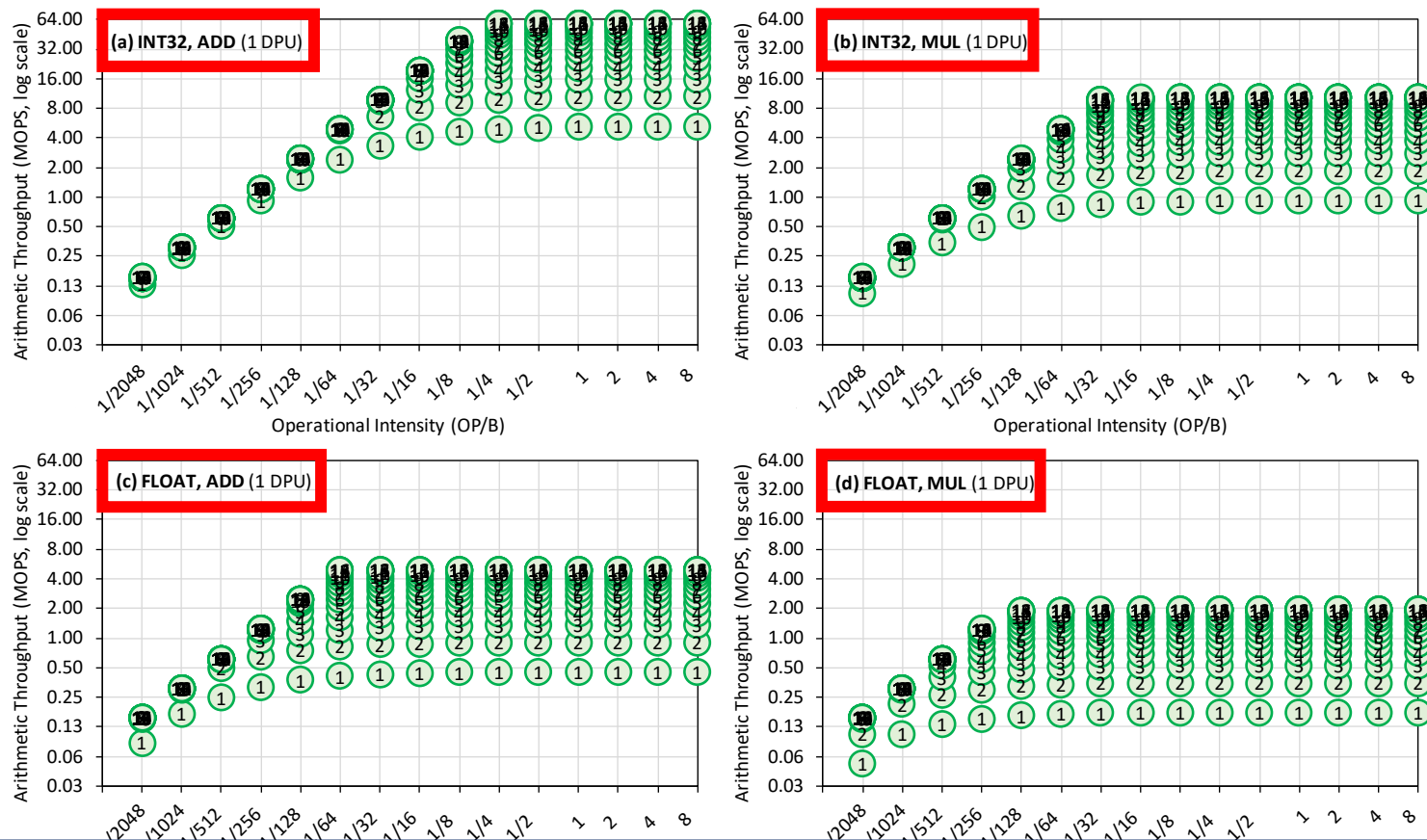
// Update
for(int r = 0; r < repetitions; r++){
    for(int i = 0; i < SIZE; i+=stride){
#ifdef ADD
        bufferA[i] += scalar; // ADD
#elif SUB
        bufferA[i] -= scalar; // SUB
#elif MUL
        bufferA[i] *= scalar; // MUL
#elif DIV
        bufferA[i] /= scalar; // DIV
#endif
    }
}

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

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

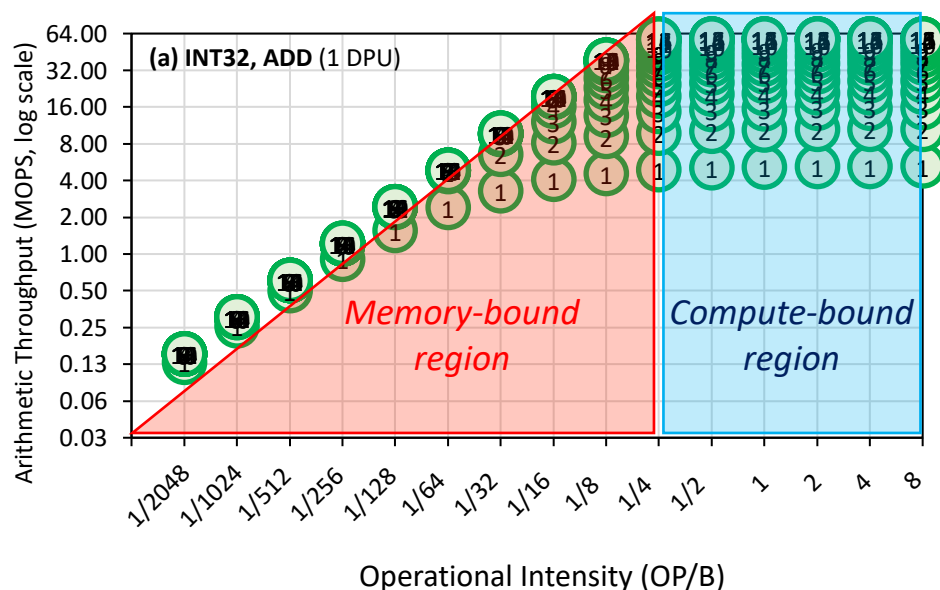
input_repeat smaller than 1 indicates the fraction of elements that are updated

Arithmetic Throughput vs. Operational Intensity (III)



We show results of arithmetic throughput vs. operational intensity for (a) 32-bit integer ADD, (b) 32-bit integer MUL, (c) 32-bit floating-point ADD, and (d) 32-bit floating-point MUL (results for other datatypes and operations show similar trends)

Arithmetic Throughput vs. Operational Intensity (IV)



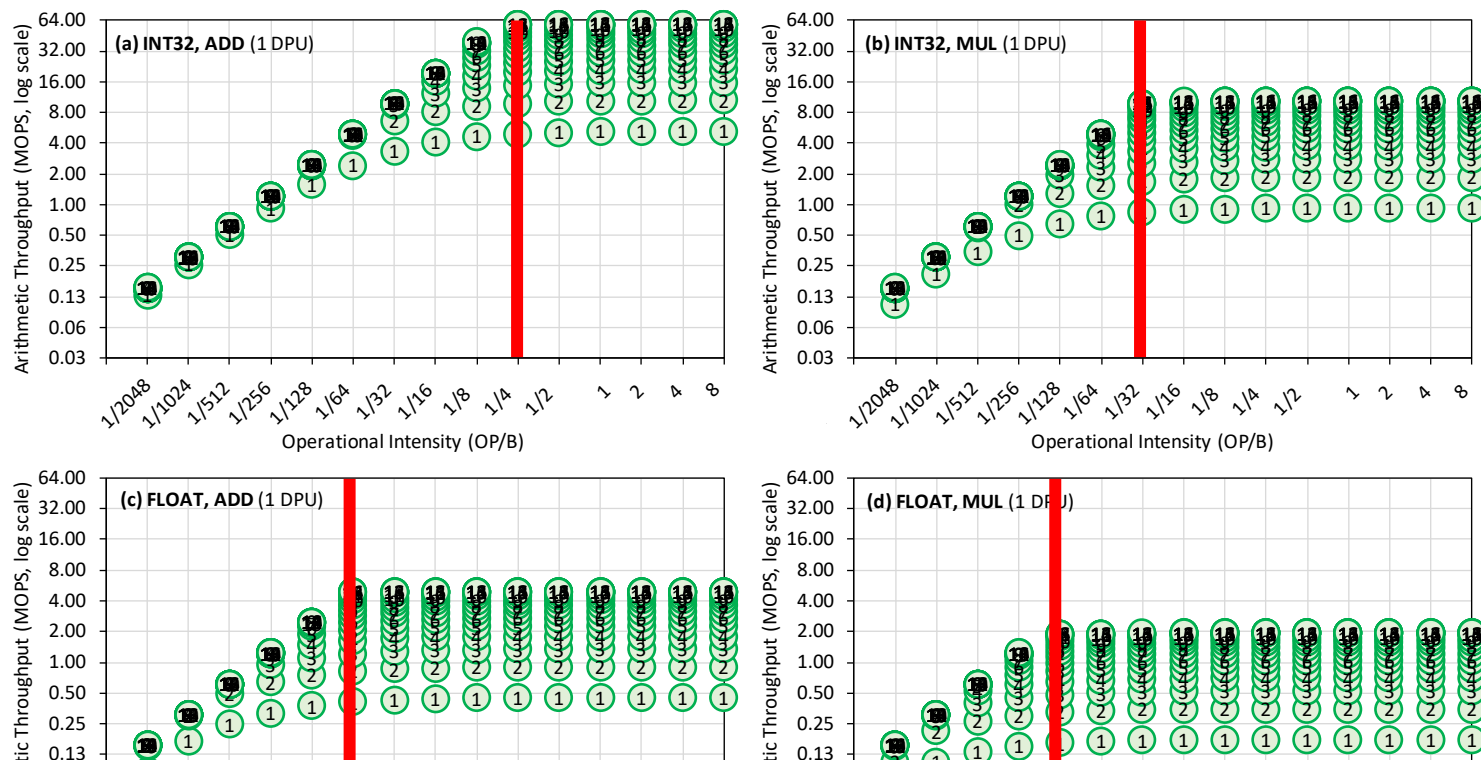
In the **memory-bound region**, the arithmetic throughput increases with the operational intensity

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

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

The throughput saturation point is as low as $\frac{1}{4}$ OP/B, i.e., **1 integer addition per every 32-bit element** fetched

Arithmetic Throughput vs. Operational Intensity (V)



KEY OBSERVATION 6

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

Microbenchmark: Arithmetic Throughput vs. Operational Intensity

- Arithmetic Throughput versus Operational Intensity

CMU-SAFARI / prim-benchmarks

Unwatch

2

Star

2

Fork

1

<> Code

Issues

Pull requests

Actions

Projects

Wiki

Security

Insights

Settings

main

prim-benchmarks / Microbenchmarks / Operational-Intensity /

Go to file

Add file

...

Juan Gomez Luna PRIM -- first commit

3de4b49 9 days ago

History

..

folder

dpu

PRIM -- first commit

9 days ago

folder

host

PRIM -- first commit

9 days ago

folder

support

PRIM -- first commit

9 days ago

file

Makefile

PRIM -- first commit

9 days ago

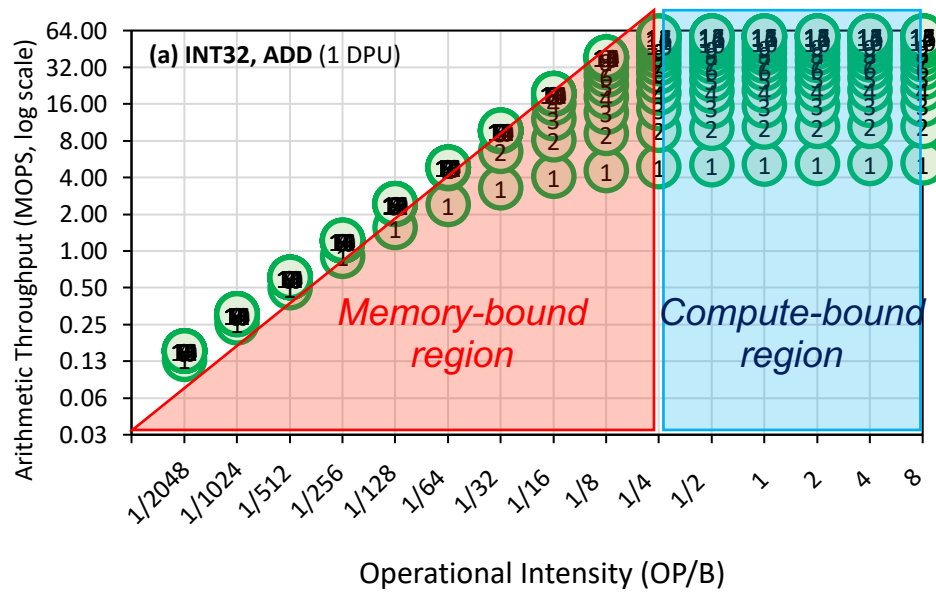
file

run.sh

PRIM -- first commit

9 days ago

Key Takeaway 1



The throughput saturation point is as low as $\frac{1}{4}$ OP/B, i.e., 1 integer addition per every 32-bit element fetched

KEY TAKEAWAY 1

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

Experimental Analysis of the UPMEM PIM Engine

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



The video player shows a seminar titled "Understanding a Modern Processing-in-Memory Architecture: Benchmarking and Experimental Characterization". The speaker is Juan Gómez Luna, with co-presenters Izzat El Hajj, Ivan Fernandez, Christina Giannoula, and Geraldo F. Oliveira, and Onur Mutlu. The video includes links to an arXiv paper and a GitHub repository. The player interface shows a progress bar at 2:26 / 2:57:10, and the channel name "Onur Mutlu Lectures" with 18.7K subscribers. The video has 2,579 views and was streamed live on Jul 12, 2021.

Understanding a Modern Processing-in-Memory Architecture: Benchmarking and Experimental Characterization

Juan Gómez Luna, Izzat El Hajj,
Ivan Fernandez, Christina Giannoula,
Geraldo F. Oliveira, Onur Mutlu

<https://arxiv.org/pdf/2105.03814.pdf>
<https://github.com/CMU-SAFARI/prim-benchmarks>

ETH Zürich SAFARI

2:26 / 2:57:10

SAFARI Live Seminar: Understanding a Modern Processing-in-Memory Architecture

2,579 views • Streamed live on Jul 12, 2021

93 0 SHARE SAVE ...

Onur Mutlu Lectures
18.7K subscribers

SUBSCRIBED

Upcoming Lectures

- Introduction to Samsung's and SK Hynix's PIM devices
- Programming an UPMEM-based PIM system
- Workload characterization for PIM suitability

P&S Processing-in-Memory

Real-World Processing-in-Memory Architectures:
Microbenchmarking of UPMEM PIM

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Spring 2022

24 March 2022