Sorting Algorithms Using Processing-in-Memory

Meven MOGNOL
University of Rennes, IRISA, Inria, Genscale

Thesis Director: LAVENIER Dominique
Industrial Advisor: LEGRIEL Julien (UPMEM)

Université de Rennes UMR IRISA cnrs Inria upmem

What is PiM

Processing-in-Memory is an emerging technology:

- Integrate computational capabilities directly within the memory.
- Minimizes the need for data movement between memory and the CPU.
- It aims to improve both performance and energy efficiency.

Near-Memory

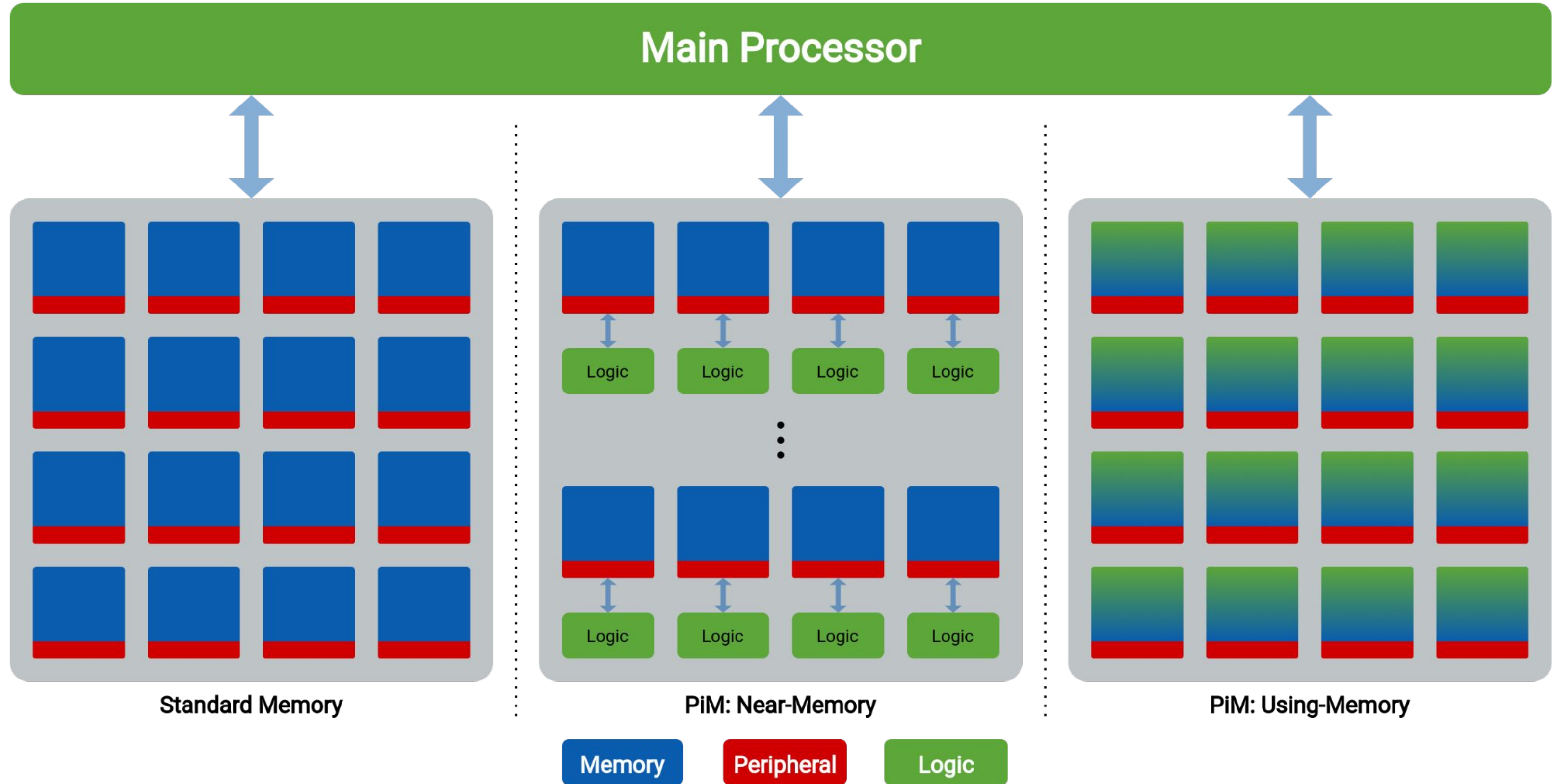
- Processing close to memory
- Add compute unit close to memory arrays
- Ex: SSD with read filtering [1]
- Ex: UPMEM Memory [2]

Using-Memory

- Processing with memory
- Use memory for computation
- Ex:CASA (CAM based) [3]

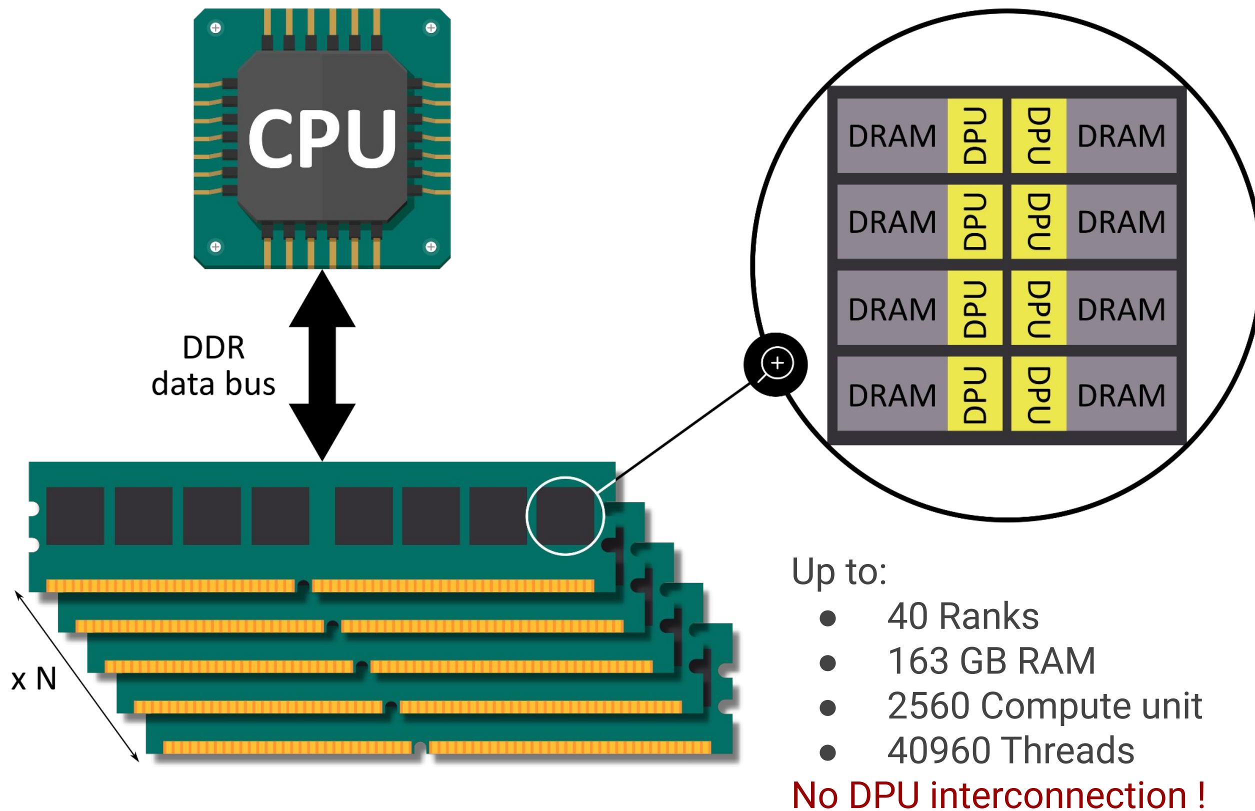
[1] Nika Mansouri Ghiasi et al. 2022. GenStore: A High-Performance in-Storage Processing System for Genome Sequence Analysis. ASPLOS '22
[2] Fabrice Devaux. The true processing in memory accelerator, 2019 IEEE Hot Chips 31 Symposium (HCS)
[3] Yi Huang et al. 2023. CASA: An Energy-Efficient and High-Speed CAM-based SMEM Seeding Accelerator for Genome Alignment. MICRO '23

What is PiM: In Pictures



[*] Khan, A. A. et al, "The Landscape of Compute-near-memory and Compute-in-memory: A Research and Commercial Overview", 2024, arXiv

UPMEM PiM System [1]



Up to:

- 40 Ranks
- 163 GB RAM
- 2560 Compute unit
- 40960 Threads

No DPU interconnection !

[1] Fabrice Devaux. The true processing in memory accelerator, 2019 IEEE Hot Chips 31 Symposium (HCS)

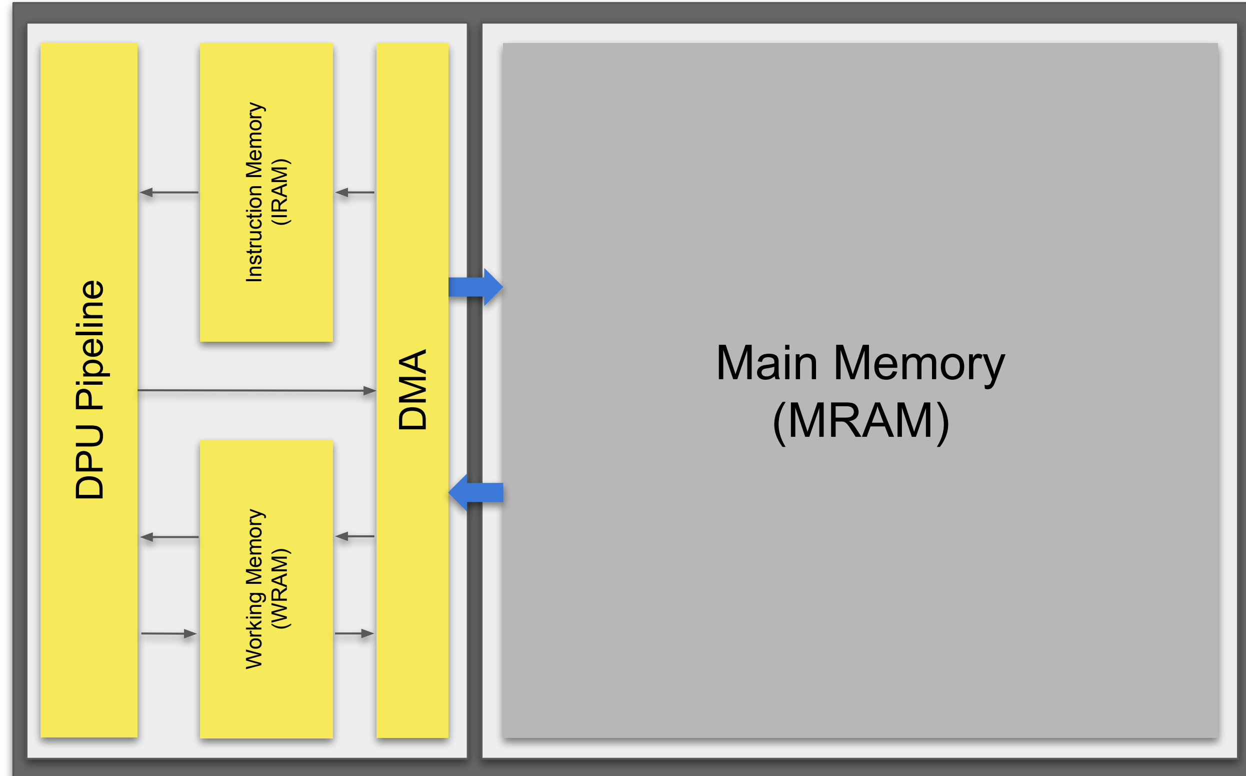
The DPU

The Memory

- WRAM: 64KB scratchpad
- IRAM: 4096 instructions
- MRAM: 64MB

The DPU:

- DPU: 350Mhz
- Custom ISA
- 16 Threads
- 11 Steps pipeline
- Direct WRAM access



PiM Programming

Host: C, C++, Python, Java

```c
int main()
{
    struct dpu_set_t dpu_set;
    struct dpu_set_t dpu;
    uint32_t nr_of_dpus = 0;
    uint32_t dpu_checksum = 0;
    bool status = true;

    dpu_alloc(nr_dpus: NR_DPUS, profile: NULL, &dpu_set);
    dpu_load(dpu_set, binary_path: DPU_BINARY, program: NULL);

    create_test_file();

    dpu_copy_to(dpu_set, symbol_name: "file",
                symbol_offset: 0, src: test_file, length: FSIZE);
    dpu_launch(dpu_set, policy: DPU_SYNCHRONOUS);

    dpu_results_t result;
    uint32_t each_dpu = 0;
    DPU_FOREACH(dpu_set, dpu, each_dpu)
    {
        dpu_prepare_xfer(dpu_set: dpu, buffer: &result);
    }
    dpu_push_xfer(dpu_set, xfer: DPU_XFER_FROM_DPU, symbol_name: "g_result",
                  symbol_offset: 0, length: sizeof(uint32_t), flags: DPU_XFER_DEFAULT);

    dpu_free(dpu_set);

    return 0;
}
```

DPU: C, C++ (limited, no std::lib)

```c
__dma_aligned uint8_t DPU_CACHES[NR_TASKLETS][CSIZE];
__host uint32_t tasklet_checksum[NR_TASKLETS];
__mram_noinit uint8_t DPU_BUFFER[BSIZE];

__mram uint32_t g_result;

BARRIER_INIT(barrier, NR_TASKLETS);

int main()
{
    uint32_t tid = me();
    uint8_t *cache = DPU_CACHES[tid];
    uint32_t checksum = 0;

    for (uint32_t buf_id = tid * CSIZE; buf_id < BSIZE; buf_id += (NR_TASKLETS * CSIZE))
    {
        mram_read(from: &DPU_BUFFER[buf_id], to: cache, nb_of_bytes: CSIZE);

        for (uint32_t cache_idx = 0; cache_idx < CSIZE; cache_idx++)
            checksum += cache[cache_idx];
    }

    tasklet_checksum[tid] = checksum;

    barrier_wait(&barrier);

    if (tid == 0)
    {
        g_result = checksum;
        for (uint32_t i = 1; i < NR_TASKLETS; i++)
            g_result += tasklet_checksum[i];
    }

    return 0;
}
```

Host: C, C++, Python, Java

DPU: C, C++ (limited, no std::lib)

```
int main()
{
    struct dpu_set_t dpu_set;
    struct dpu_set_t dpu;
    uint32_t nr_of_dpus = 0;
    uint32_t dpu_checksum = 0;
    bool status = true;
    ...
    dpu_results_t result;
    uint32_t each_dpu = 0;
    DPU_FOREACH(dpu_set, dpu, each_dpu)
    {
        dpu_prepare_xfer(dpu_set: dpu, buffer: &result);
    }
    dpu_push_xfer(dpu_set, xfer: DPU_XFER_FROM_DPU, symbol_name: "g_result",
                  symbol_offset: 0, length: sizeof(uint32_t), flags: DPU_XFER_DEFAULT);

    dpu_free(dpu_set);

    return 0;
}
```

```
__dma_aligned uint8_t DPU_CACHES[NR_TASKLETS][CSIZE];
__host uint32_t tasklet_checksum[NR_TASKLETS];
__mram_noinit uint8_t DPU_BUFFER[BSIZE];

__mram uint32_t g_result;

BARRIER_INIT(barrier, NR_TASKLETS);
...
            checksum += cache[cache_idx];
    }

    tasklet_checksum[tid] = checksum;

    barrier_wait(&barrier);

    if (tid == 0)
    {
        g_result = checksum;
        for (uint32_t i = 1; i < NR_TASKLETS; i++)
            g_result += tasklet_checksum[i];
    }

    return 0;
}
```

ETH Zürich: SimplePiM [1] - C

Institut Pasteur & IRISA: BPL - C++

[1] J. Chen et al, "SimplePIM: A Software Framework for Productive and Efficient Processing-in-Memory", PACT 2023

Area of research: Exploring Genomic Algorithms for PiM Architecture

Sort Applications in Genomic Pipelines:

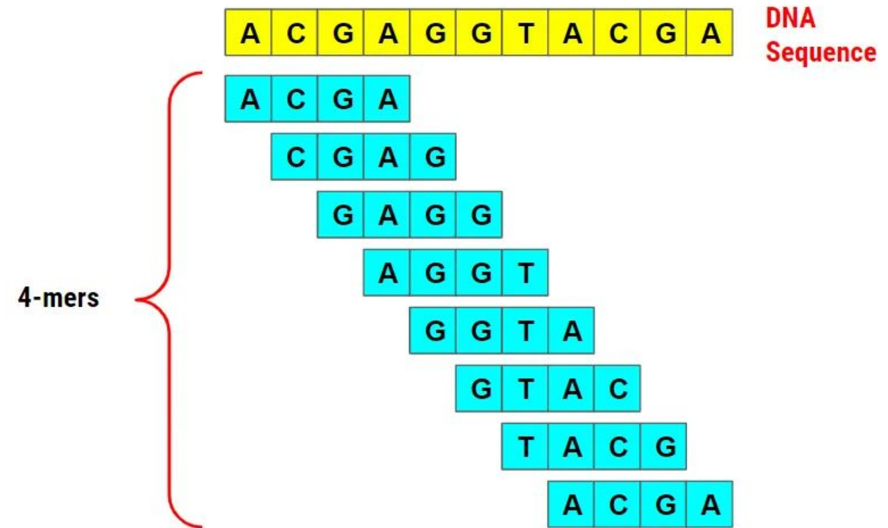
- K-mer Counting
- BAM File Sorting

Importance of Sorting Algorithms

- Fundamental of computer science.

Sorting Challenges in PiM:

- Directly sorting large arrays is impractical.



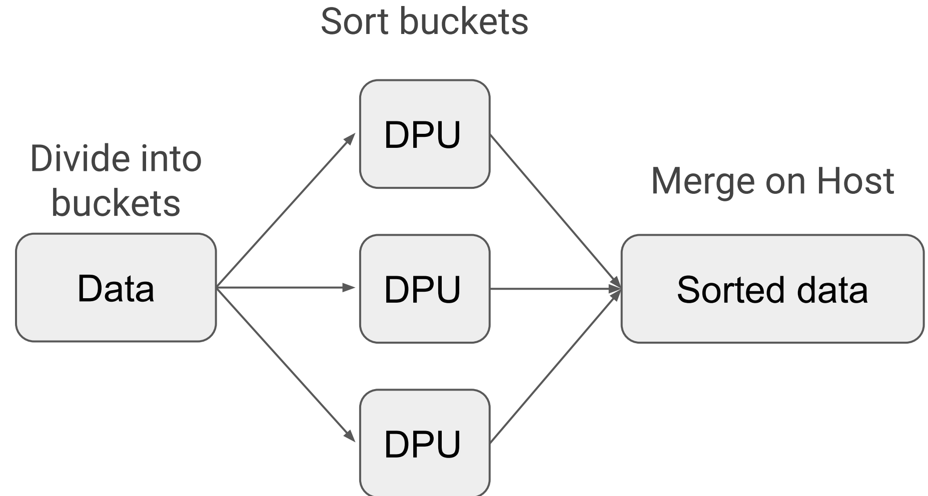
Sorting Strategies for PiM

Solution to Sorting Challenges:

- Sort smaller buckets
- Merge them.

Algorithms Tested for Bucket Sorting:

- Quick Sort
- Heap Sort
- Radix Sort (used in KMC2 [1])



[1] Deorowicz S. et al. KMC 2: fast and resource-frugal *k*-mer counting, *Bioinformatics*, Volume 31, Issue 10, May 2015

Comparison environment

Dataset:

- 40960 buckets
- 500k 32 bits integers per buckets (2MB)
- ~80 Go

Systems:

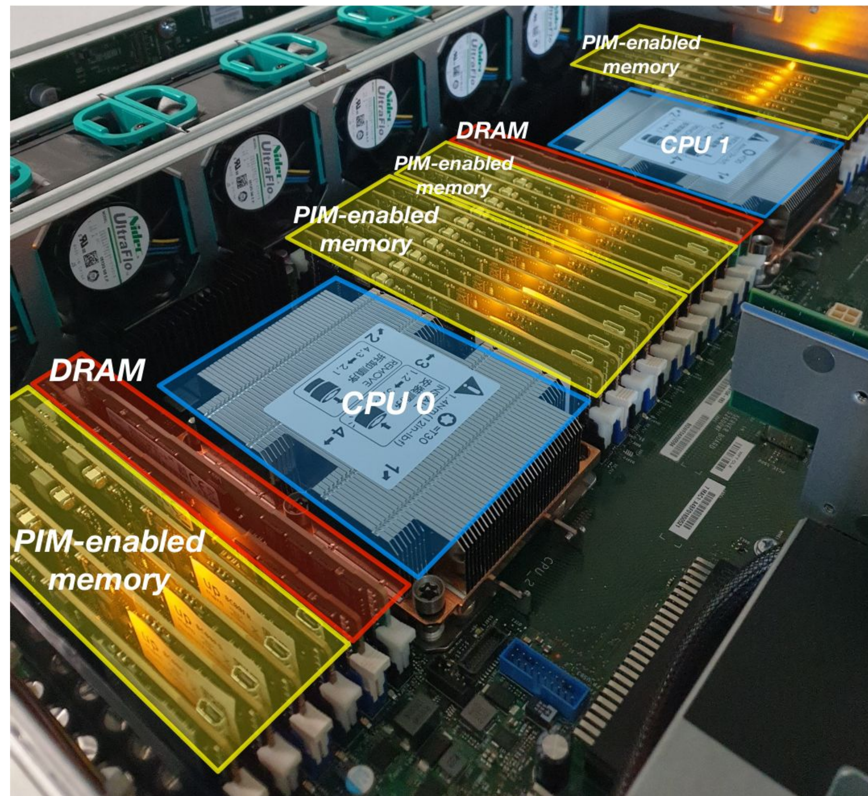
- Intel Xeon 4216 (64 cores)
- 40 UPMEM PiM ranks

Benchmarks:

- In-house implementation
- Intel AVX2 quicksort

Analysis:

- Architecture usage
- Performance



CPU Analysis: Vtune

Speculation Bound Algorithms:

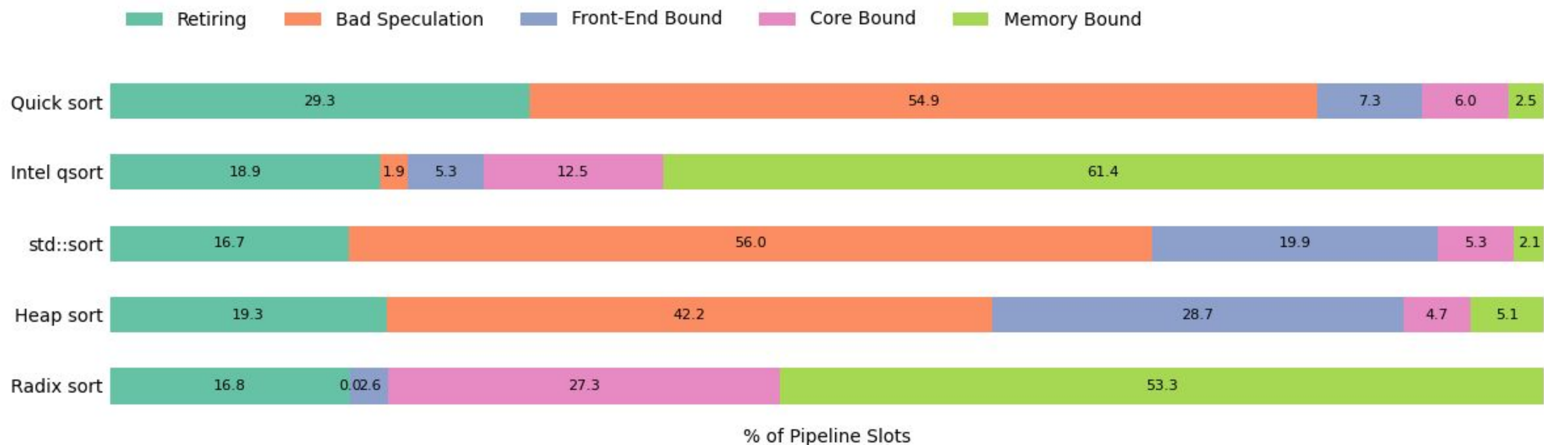
- Quick Sort and Heap Sort:
- Naive implementations (including C++ standard sort) are speculation bound.
- Utilizing AVX2 instructions eliminates this bottleneck for quick sort.

Memory Bound Algorithms:

- Radix Sort is inherently memory bound.
- Intel Quick Sort (AVX2).

Additional Information:

- CPU L3 Cache: 22MB.



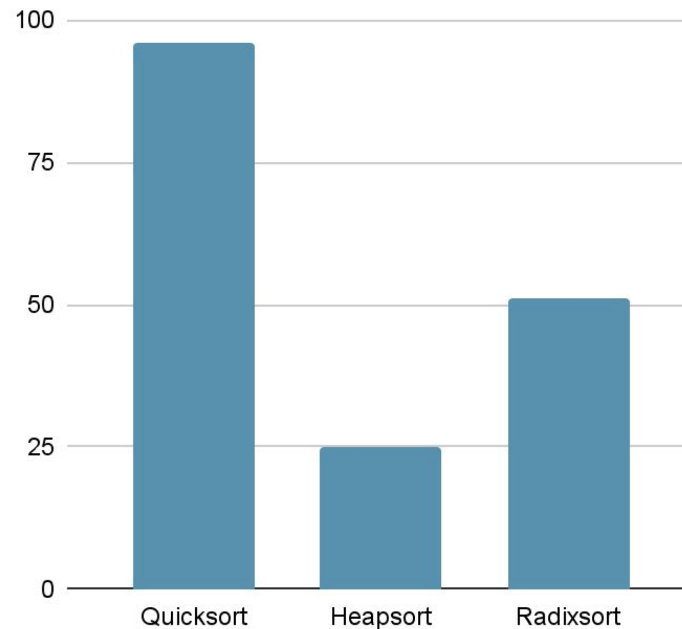
DPU Analysis

PiM Implementation:

- All implementations use manual cache to limit MRAM-WRAM transfers.
- Heapsort access patterns are more random.
- Radix sort's final step is also random in nature.

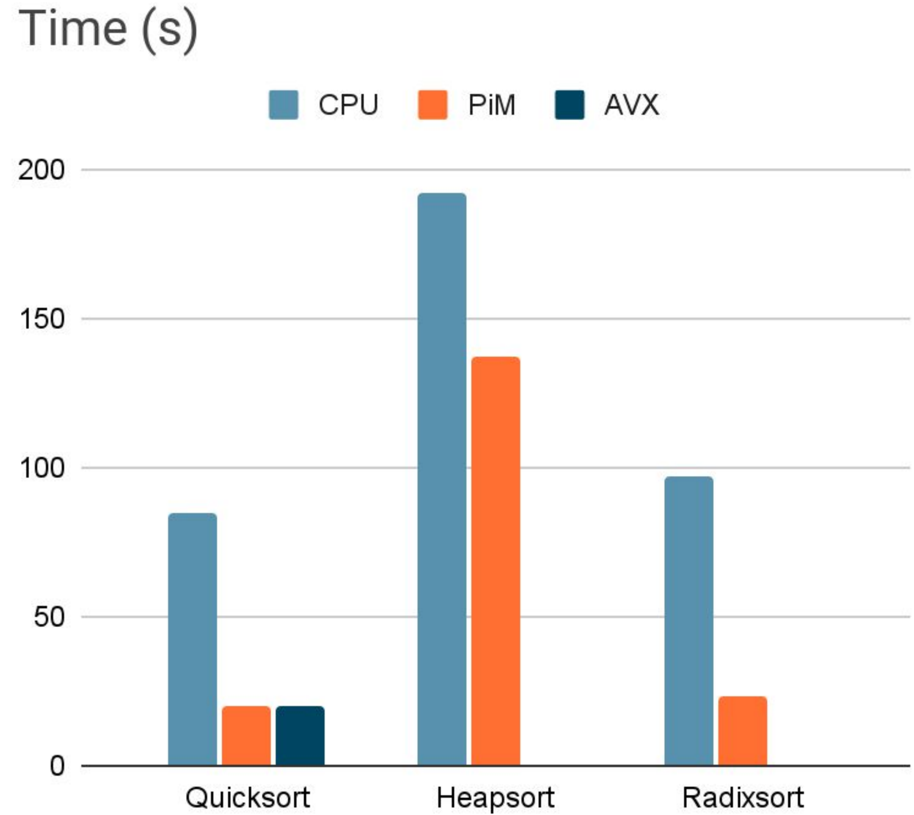
PiM Analysis:

- Quick sort is compute bound on DPU.
- Both heap sort and radix sort are memory bound on DPU due to random memory accesses limiting caching mechanism.
- Host to PiM transfer overhead is small.



Performance comparison

- PiM is 1x to 4.2x faster for sorting buckets.



Conclusion

Benefits:

- UPMEM PiM can enhance speculation bound applications

Limitations:

- Not suitable for standalone sort acceleration.

Use of sorting in PiM:

- Can be integrated into more complex DPU applications without performance loss.