Graph Processing using Processing in Memory

- Supervisor: Muhammad Attahir Jibril
- Presenter: Krishna Kumar Ranganath

Introduction

What is Processing in Memory?

- Processing in Memory (PIM) is a technology in which the memory units such as DRAM is integrated with processing capabilities.

- Applications such as Graph Processing, Machine Learning (ML) and Artificial Intelligence (AI), Database Querying are usually memory bound, processing them in memory benefits in lower latency and higher bandwidth.

- PIM reduces the traditional data movement bottleneck between the CPU and the memory by processing tasks in the memory.

Why Processing In Memory?

Benefits of PIM

In the traditional CPU architecture data is stored in the memory and processed in the CPU.

- Reduces data movement bottleneck.
- Improves engergy efficiency, hence reducing power consumption.
- Parallel Processing.
- Accelerates processing speeds drastically.

CPU Architecture	Processing in Memory
Processing takes place in the CPU	Processes data inside the memory
Slower processing for larger datasets	Faster processing for large datasets
Limited parallelization	Massive parallelization
Frequent data movement between CPU and Memory	Minimal data movement between CPU and Memory

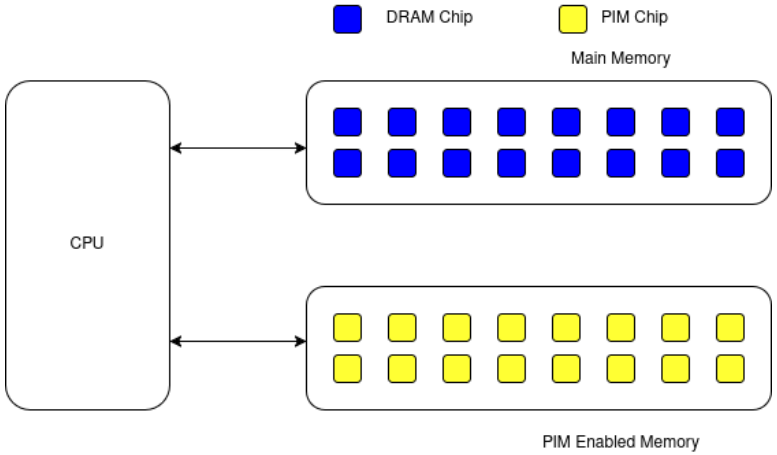
Comparison between the traditional CPU architecture and Processing in Memory

Research Objective

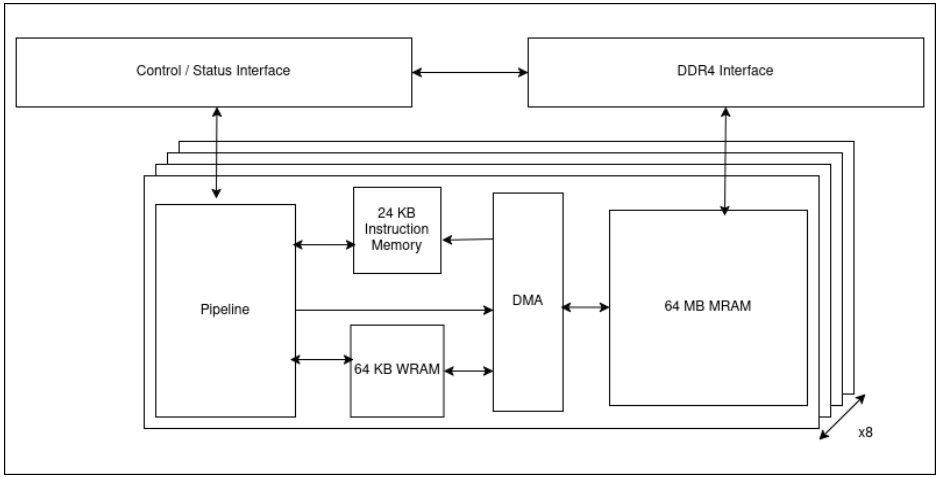
- **Processing the Single-Source Shortest path Algorithm using memory:** Implementing the Bellman-Ford Algorithm using the UPMEM SDK and executing it on the UPMEM DRAM Processing Units (DPUs).
- **Comparison between the CPU and DPU Implementation:** The algorithm is run on both the CPU and the DPU to check the time taken to execute the program and also validate the DPU implementation results using the CPU results.
- **Optimization of the algorithm:** The algorithm is optimized to run in parallel by splitting the graph across multiple DPUs.

UPMEM DIMM Architecture

The UPMEM DIMM contains 8 or 16 PIM chips, inside each PIM chip there are 8 DRAM Processing Units (DPUs) which coexist in the system along with the standard DIMMs.



PIM Enabled Memory



Components of a PIM Chip

Important components of a DPU

- **Main Ram (MRAM):** 64MB per DPU used to store the graph, and responsible for data transfer between host (CPU) and MRAM.
- **Working Ram (WRAM):** Used for computation, tasklets load data from MRAM into WRAM for processing, volatile memory which holds frequently used data temporarily with a capacity of 64KB.
- **Instruction Memory (IRAM):** Stores the executable code of the program having a capacity of 24KB.
- **Tasklets:** Parallels threads inside each DPU, allowing parallelization in each DPU, each DPU can have upto 24 tasklets. Tasklets read data from MRAM into WRAM, and write the results back to MRAM.

Tools used for this implementation

Programming Language: C (for writing the Host and DPU programs), Python 3 (for generating input Graph text files)

Environment: UPMEM SDK

Operating System: Linux (Ubuntu server 20.04 LTS)

Text Editor: Visual Studio Code

Graph algorithms on the UPMEM DPUs

- Processing large scale graphs on the CPU is challenging and takes a lot of time, whereas processing them in memory is quicker because there is minimal data transfer and there is an added advantage of parallelization.
- To compute the shortest path from the source vertex to all other vertices in the graph the Bellman-Ford algorithm is implemented on the UPMEM DPUs
- **Why?** Because it is easier to parallelize as it updates all edges in each iteration.
- The input is a directed, weighted, dense graph read from a **.txt** file with the source vertex as 0 and all the vertices are connected either directly or indirectly to the source vertex.

Host Program and DPU Program

- Writing programs using the UPMEM SDK involves two key files the Host program file and the DPU program file.
- **Host Program file:** The role of the Host Program in this implementation is as follows:
  - Reading the Graph from the input text file.
  - Allocating the Number of DPUs and specifying the backend as simulator or hardware using the **dpu_alloc()** function.
  - Loading the DPU program into each DPU.
  - Partitioning the Graph across multiple DPUs.
  - Transferring the Graph Data to DPUs using **dpu_copy_to()** function.
  - Retrieving the updated distances from the DPUs using the **dpu_copy_from()** function.

Host Program and DPU Program

- The updated distances processed by the DPU are compared with the CPU‘s result to ensure correctness.
- The Host Program also calculates the execution time for both CPU and the DPU implementation.

- **DPU Program file:** The DPU program processes the Bellman-Ford algorithm, and has the following role:
  - Contains the logic of the algorithm.
  - **Tasklet Execution:** Multi thread execution within DPUs.
  - **Memory Declarations:** The MRAM and WRAM buffers are declared at the beginning of the program.

Graph Partitioning Strategy

- To process the Graphs, the edges are split across DPUs, each DPU gets allocated a subset of the Graph.
- This ensures in reducing redundancy, decrease in execution time, lower power and memory usage.
- The following formula is implemented in the program:

int edges_per_dpu = (no_of_edges + no_of_dpus-1) / no_of_dpus;

For Example:

Number of Edges = 20000 and Number of DPUs = 3, each DPU receives 6667 edges.
Since the int datatype is used it truncates the value from 6667.33 to 6667.

Graph Partitioning Strategy

- Next, inside the **DPU_FOREACH()** function the following logic is implemented:

  int start_index = dpu_id * edges_per_dpu;

  Int end_index = (start_index + edges_per_dpu < num_edges) ? (start_index + edges_per_dpu) : num_edges;

Initially dpu_id is set to 0 and it keeps incrementing upto the number of dpus because of the **DPU_FOREACH()** function.

According to this logic the following result is obtained:

- dpu 0 receives 0 to 6667, 6667 edges.
- dpu 1 receives 6667 to 13334, 6667 edges.
- dpu 2 receives 13334 to 20000, 6666 edges.

Graph Partitioning Strategy

- For dpu 2 after applying the formula it is observed that the end_index will be 20001 **((13334 + 6667 = 20001 < 20000) ? 20001 : 20000),** the value is greater than the number of edges.
- The ternary operator ‘?’ checks the condition and it results in false, in this case the number of edges is set as the end_index for the final DPU, and the final DPU receives 1 less edge than the rest.
- After every relaxation step the Host (CPU) sends the updated distances retrieved from the DPU back to the DPU until (V – 1) times.

Result Analysis

The result of the implementation is displayed below with different test cases:

Small scale graph with 5000 vertices and 20000 edges using 16 DPUs and 16 tasklets:

```
krishna@upmem:~/Graph-processing-using-Processing-in-Memory$ sudo ./sssp_host graph1.txt

Running SSSP on CPU...
CPU Execution Time: 399.419 ms
Number of DPUs: 16
Number of Vertices: 5000
Number of Edges: 20000

Comparison of CPU and DPU Results:
Matching distances: 5000
Mismatched distances: 0

Metrics:
CPU Execution Time: 399.419 ms
DPU Execution Time: 248.413 ms
CPU to DPU Transfer Time: 6.388 ms
DPU to CPU Transfer Time: 5.764 ms
```

Result Analysis

Medium scale graph with 20000 vertices and 50000 edges using 40 DPUs and 16 tasklets:

```
krishna@upmem:~/Graph-processing-using-Processing-in-Memory$ sudo ./sssp_host graph2.txt

Running SSSP on CPU...
CPU Execution Time: 5280.193 ms
Number of DPUs: 40
Number of Vertices: 20000
Number of Edges: 50000

Comparison of CPU and DPU Results:
Matching distances: 20000
Mismatched distances: 0

Metrics:
CPU Execution Time: 5280.193 ms
DPU Execution Time: 1887.741 ms
CPU to DPU Transfer Time: 23.773 ms
DPU to CPU Transfer Time: 48.082 ms
```

Result Analysis

Large scale graph with 40000 vertices and 70000 edges using 90 DPUs and 16 tasklets:

```
krishna@upmem:~/Graph-processing-using-Processing-in-Memory$ sudo ./sssp_host graph5.txt

Running SSSP on CPU...
CPU Execution Time: 21309.389 ms
Number of DPUs: 90
Number of Vertices: 40000
Number of Edges: 70000

Comparison of CPU and DPU Results:
Matching distances: 40000
Mismatched distances: 0

Metrics:
CPU Execution Time: 21309.389 ms
DPU Execution Time: 12543.312 ms
CPU to DPU Transfer Time: 90.737 ms
DPU to CPU Transfer Time: 200.857 ms
```

Future Optimization

- Optimize the program to handle larger and complex datasets.
- The Bellman-Ford algorithm can handle negative weights, the DPU program can be modified further to detect and handle negative weights in the Graph.
- Graph partitioning method can further be optimized to distribute edges across the DPUs and synchronize them in a more efficient manner to reduce the execution time even further.

Conclusion

- From the results it is observed that the DPU implementation reduces the execution time drastically.
- Each DPU is assigned a subset of the Graph and the workload is getting distributed evenly and the edges are being processed simultaneously.
- It is more efficient to use lesser number of DPUs for small scale graphs and increase the number of DPUs gradually depending on the size of the dataset.

Thank you!

Discussions