Technische Universität Ilmenau
Department of Computer Science and Automation
Institute of Practical Computer Science
Databases and Information Systems Group

Master Research Project

Graph Processing using Processing in Memory

Submitted by:
Krishna Kumar Ranganath

Advisors:	Prof. Dr.-Ing. habil. Kai-Uwe Sattler
	M. Sc. Muhammad Attahir Jibril
Program:	RCSE
Matriculation Number:	66079
Submission Date:	3 March 2025

Contents

List of Figures

List of Tables

Abstract

The Single Source Shortest Path (SSSP) problem is a well-known graph problem with numerous applications in networking, logistics, and scientific computing. Traditional CPU-based SSSP algorithms are not very scalable because certain algorithms are inherently sequential and memory bandwidth-bound. To avoid these constraints, this research examines the implementation of SSSP algorithms on the UPMEM DPU, an innovative Processing-in-Memory (PIM) architecture designed to accelerate data-intensive applications by reducing data movement overhead.

Keywords: Processing in Memory (PIM), UPMEM, Graph Processing, Single-source shortest path, DRAM Processing Units (DPUs), Tasklets, MRAM, WRAM, IRAM.

CHAPTER 1

Introduction

In many modern workloads, a large fraction of the execution time and energy consumption of data-intensive workloads is spent moving data between the memory and the CPU cores [1]. Applications such as neural networks, graph processing are fundamentally memory bound. For such applications, processing-in-memory (PIM) is a promising solution [1]. The motivation behind PIM is to tackle the data movement bottleneck by making the memory responsible for the computation tasks, and relieving the CPU from doing any computation, hence reducing the transfer time and the amount of data transfer between the memory and CPU [2]. This Research Project focuses on Processing the Single Source Shortest Path (SSSP) algorithm using PIM. The shortest path problem involves finding the shortest path from the source vertex to all other vertices in the graph. Bellman-Ford is one of the algorithms that solves the single source shortest path problem and can be parallelized for many architectures [3]. The Bellman-Ford algorithm computes the shortest distances in a directed weighted graph from the source vertex to all the other vertices in the graph, with one or more negative edge weights. The algorithm repeatedly relaxes all the edges in the graph for shorter paths according to the number of vertices in the graph minus 1 (V - 1) [4]. This paper also discusses certain benchmarks specific to this algorithm running on the DPU such as DPU Execution time, CPU Execution time, Transfer time between the CPU and DPU, and DPU memory utilization. Finally the DPU implementation of the algorithm is validated by comparing the results to the CPU baseline.

1.1 In-Memory Graph Processing

Graph Processing in memory refers to using RAM to compute graph structures efficiently, increasing the throughput, and improving the performance of large-scale graph analytics. PIM has been considered as a promising solution to providing higher bandwidth, and reducing the data transfer rate and time. The SSSP algorithm calculates the shortest path from the source vertex to all the other vertices in the graph [5]. Given a weighted directed graph *G = (V, E)* and a source vertex *s*, the Bellman-Ford algorithm finds the shortest path from *s* to each vertex *v* in the graph. The shortest path from *s* to *v* is stored in a distance array d(v) for each vertex *v* [6]. Steps in Bellman-Ford Algorithm [7]:

1. Source vertex *s* is set to zero, and the initial distances is set to infinity for all other vertices.
2. Check if a shorter distance can be calculated for each edge, and update the distance if the calculated path is shorter.

3. Check all edges V - 1 times.
4. Loop through all the edges checking for negative weights in the graph.

Notation	Meaning
G	a graph $G = (V,E)$
V	vertices in G, $/V/ = n$
E	edges in G, $/E = m$
vi	vertex i
$ei.j$	edge from vi to vy
s	source vertex of a graph

Table 1.1: Notations of a Graph [5]

The Bellman-Ford algorithm is memory-intensive due to it's reliance on random access patterns, large adjacency structures, and iterative updates. The Traditional CPU architecture sometimes faces performance bottlenecks because of frequent data movement between CPU and memory. PIM performs the computing directly in the memory, significantly increasing the performance of graph algorithms[8]. PIM involves standard Dual In-line Memory Modules (DIMMS), with a large number of DRAM Processing Units (DPUs) combined with standard DRAM chips [1].

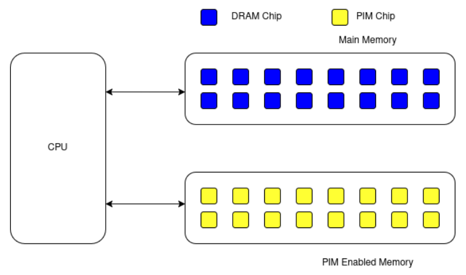

Figure 1.1: PIM Enabled Memory [1].

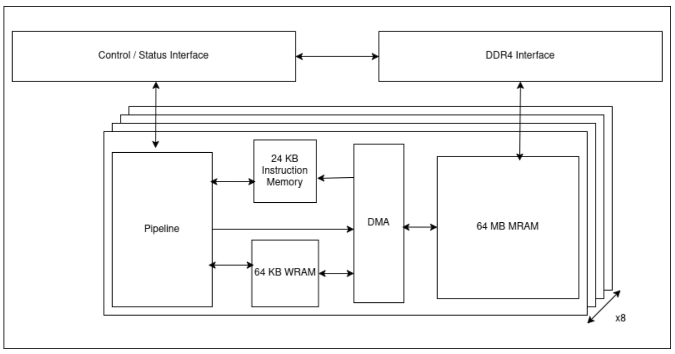

Figure 1.2: PIM Chip Block Diagram [1]

Figure 1.1 represents a UPMEM PIM system, and Figure 1.2 represents the components of a UPMEM PIM Chip. UPMEM has designed the first commercially available PIM architecture. Each UPMEM PIM chip contains 8 DPUs, and each DPU has exclusive access to 64-MB DRAM Bank called the Main RAM (MRAM), a 64-KB Working RAM (WRAM), and a 24-KB Instruction memory (IRAM). The MRAM can be accessed by the CPU for copying input data and retrieving results [1].

1.2 Task Description

Graph Processing is an important workload in data-intensive applications, the traditional CPU architecture may not be suitable for handling large scale graphs, and PIM provides an efficient solution by offloading the computational tasks to the memory instead of the CPU. This Research Project focuses on evaluating the performance of the Bellman-Ford algorithm by running it on the UPMEM Hardware and SDK, and then comparing the results with the CPU baseline implementation to check for correctness, analyze the efficiency, speed and memory utilization of executing the program on multiple DPUs.

1.3 Objectives

This Research Project investigates the emerging PIM technology and discusses the key role it plays in processing large graphs. The following are the objectives of this research project:

- Implementation of the Bellman-Ford algorithm on the UPMEM SDK to find the shortest path to all other vertices in the graph.
- Optimize the code by partitioning the graph to evenly distribute the edges across DPUs, each DPU processes a set of edges in parallel which significantly reduces the workload of each DPU and the execution time.
- Scale the algorithm across multiple DPUs and determine the DPU execution time and memory utilization.
- Validating the DPU implementation results by comparing it with the CPU implementation to ensure correctness.

1.4 Overview of the Research Project organization

- **Chapter 1:** Describes the introduction to Processing in Memory, task description, and objectives of the Research Project
- **Chapter 2:** Provides an in detail explanation on the components of UPMEM PIM, briefly discusses about traditional cpu based processing in comparison with in memory processing, and graph processing in UPMEM PIM.
- **Chapter 3:** Discusses about the Host application and DPU program, their respective functions and compilation process.

- **Chapter 4:** Briefly discusses on the Implementation approach and results analysis.
- **Chapter 5:** Details the project conclusion and explains the future optimizations.

CHAPTER 2

Processing in Memory using UPMEM

This chapter briefly describes how the Bellman-Ford algorithm was leveraged using the UPMEM PIM, all the components of the UPMEM PIM are discussed in detail, and also the approach taken in implementing the algorithm on the UPMEM SDK.

2.1 Traditional CPU-Based processing vs In-Memory processing

Processing large graphs is memory-intensive task, In a Traditional CPU-Based systems graphs workloads require frequent memory access which increases the time taken in moving the data between the CPU and the memory, this also slows down the performance since CPUs are optimized for sequential memory access [9]. CPUs have limited parallelism (8-64 cores) and scalability, while processing a graph multiple nodes and edges needs to be processed simultaneously [10]. PIM provides a solution to these challenges faced by the CPU, PIM provides low latency as the processing occurs inside the memory and there is no frequent data movement between the memory and CPU, it also offers high parallelism as nodes and edges can be processed simultaneously across multiple DPUs, the number of DPUs can be scaled according to the amount of data to ensure faster processing [11].

2.2 Components of the UPMEM PIM

A UPMEM DIMM contains contains 8 or 16 chips, inside each PIM chip there are 8 DPUs and 8 64-MB MRAM Banks [1]. This section provides an in detailed explanation about the components of the UPMEM PIM:

- **DRAM Processing Units (DPUs):** DPUs are computing cores embedded inside UPMEM's PIM chips. DPUs enable processing directly inside the memory, therefore reducing data movement overhead and improving performance for memory-intensive workloads such as graph algorithms [12]. Every PIM chip contains multiple DPUs which makes them have high parallel processing capabilities, in the case of processing graphs, graphs can be partitioned across multiple DPUs, each DPU will process a part of the graph using tasklets (software threads inside DPUs), the DPUs synchronize the boundary nodes (the nodes which are shared across DPUs), the results are then merged and sent to the CPU [13].

- **Tasklets:** The DPU contains software threads called Tasklets used for multithreading in DPUs, each DPU can run upto 24 Tasklets, since the number of hardware threads in a DPU is 24. The number of Tasklets can be set during the compile time, and assigned to each DPU [1].
- **Memory Management:** Each DPU has access to a 64-MB DRAM Bank called Main RAM (MRAM), a 64-KB Working RAM (WRAM), and a 24-KB Instruction RAM (IRAM) [1].
  - **Main RAM (MRAM):** MRAM is the memory space used by the DPU to store graph data and other workloads. The DPU processes directly from MRAM reducing the data movement to the CPU. Although MRAM is slower than the WRAM, it is much larger which makes it suitable to store large datasets. [14].
  - **Working RAM (WRAM):** The Data processing of the programs happen in the WRAM, it has lesser storage capacity compared to the MRAM which is why data is only stored in the WRAM temporarily. Direct Memory Access (DMA) is used to move data from the MRAM to WRAM [1].
  - **Instruction RAM (IRAM):** The IRAM is a small memory inside every DPU with the size of 24-KB. The CPU sends program instructions to each DPU, the IRAM is used to store these program instructions [15].

The basic workflow between MRAM and WRAM is depicted in the below Figure 2.1:

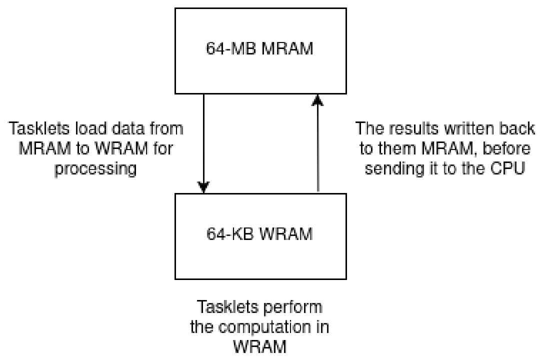
64-MB MRAM
Tasklets load data from MRAM to WRAM for processing
The results written back to them MRAM, before sending it to the CPU
64-KB WRAM
Tasklets perform the computation in WRAM

Figure 2.1: Workflow between MRAM and WRAM

CHAPTER 3
Programming in UPMEM

This chapter focuses on the key practices to follow while programming on the UPMEM SDK, it also provides an in detailed explanation of various header files and functions specific to the UPMEM SDK. The host application and the DPU program are explained in detail.

3.1 The UPMEM SDK

The purpose of the UPMEM SDK or the UPMEM DPU toolchain is to create programs embedded on DPUs, create host applications that fuel the DPU programs, and to debug the host applications and DPU programs [16]. The UPMEM DPU toolchain can be installed on Linux operating systems, the SDK also contains a functional simulator to run programs if there are no UPMEM DIMMs in the system. A program running on a machine without UPMEM DIMMs automatically falls back to the simulator, if the system contains UPMEM DIMMs, the code executes on it. This simulator is limited to 64 DPUs [17]. The two main components of programming in UPMEM are the host application and the DPU program. The host application's main task is initialize variables, receive the input and display the output, allocate the number of DPUs, transfer data to the DPUs and receive the processed data from the DPUs. The host application can be programmed in C, C++, Java and Python. The DPU program contains the main logic of a program, it is responsible for in-memory computation, storing data in MRAM, processing the data using WRAM, and running multiple tasklets on each DPU.

3.2 Host Application

The DPU Host API assists in interaction between host applications and DPUs by which host applications can define the number of DPUs, launch DPUs, and communicate with DPU program. The host application also provides the option to set the environment for the entire program whether to run it on hardware having UPMEM DIMMs or the simulator. The **<dpu.h>** header file is an essential component of the UPMEM SDK. This is a C API with which DPUs can be managed and utilized effectively. This header file also includes a set of essential functions specific to the UPMEM SDK which are listed below [18]:

- **DPU_ASSERT:** Responsible for error handling in the DPU API, and exits in case of an error.

- **dpu_alloc():** This function is used to set the number of DPUs for the program to use, and also to specify whether to use simulator or the real hardware to run the programs.
- **dpu_load():** Loads the binary executable of the DPU program into the allocated DPU set.
- **dpu_launch():** Begins the execution of the DPU program, the host application remains suspended until the execution is complete.
- **DPU_FOREACH:** Iterates over all the DPUs in the allocated set.
- **dpu_copy_to():** Sends the data from the host to the DPU MRAM.
- **dpu_copy_from():** Sends the data back to the host from the DPU MRAM after finishing the processing in the DPU.
- **dpu_free():** Upon successful completion of the execution, this function frees all the DPUs of a DPU set.

3.3 DPU Program

The DPU code is written in C and this code responsible for processing data directly in the DPUs, reducing data transfer bottlenecks, and enabling parallel execution. The DPU code uses MRAM for storing the data and WRAM to process the data received from the MRAM, tasklets which are software threads used for multi threading enables parallel execution in each DPU [14]. The header **<defs.h>** contains functions specific to the DPU program. Key components of the DPU Program are listed below [19]:

1. **Memory declarations:** The MRAM and WRAM buffers are declared at the beginning of the program.
2. **Tasklet execution:** Multi thread execution within DPUs. Using the function **me()** the tasklet system name can be retrieved.
3. **Main logic:** This section contains the computation logic of the program, this part of the code is processes inside the DPU.
4. **Computed results storage:** After processing the data, it is then stored in the MRAM and the final output is sent to the host by the MRAM.

3.4 Compilation of the Host and DPU Programs

In the UPMEM SDK, the Host application and the DPU Program are compiled separately. The SDK also provides a debugging functionality, the **LLDB** is a high-performance debugger used in the UPMEM SDK [20].

3.4.1 Host application compilation

The host program runs on the CPU and uses a standard GCC or Clang compiler. Below is an example of the host program compilation command:

```
$gcc --std=c99  host_code.c host_code `dpu-pkg-config --cflags
   --libs dpu`
```

Host program compilation command

3.4.2 DPU Program compilation

The DPU program cannot be compiled with the standard gcc compiler, the UPMEM SDK provides a separate command to compile the DPU program.

```
$dpu-upmem-dpurte-clang -o dpu_code dpu_code.c
```

DPU program compilation command

Generally the number of tasklets to be used by each DPU can only be specified during the compilation time in the terminal while compiling the DPU program. The number of tasklets can range from 1 to 24, if the number of tasklets is not mentioned the program chooses a single tasklet by default. After the host program and the DPU program are successfully compiled, the host program can be executed.

```
$dpu-upmem-dpurte-clang -DNR_TASKLETS=24 -o dpu_code dpu_code.c
```

Specifying the number of tasklets

```
$./host_code
```

Executing the host program

Debugging can also be performed by using the following commands:

```
$dpu-lldb
(lldb) file dpu_code
(lldb) process launch
(lldb) exit
```

Debugging the DPU program

CHAPTER 4

Implementation and Results

This chapter demonstrates the approach used, and provides in depth analysis of the host program and DPU program. The results are then analyzed with different use case scenarios. CPU and DPU computation is tested with multiple graph files ranging from medium to large graphs, and then the results of the CPU and DPU implementation are compared.

4.1 Use case scenarios and Benchmarks

This Research Project focuses on implementing the Single-source shortest path algorithm on the UPMEM PIM and evaluating benchmarks such as the total execution time taken by the DPUs, CPU-DPU and DPU-CPU transfer time. The execution time of the DPU and CPU are compared to determine which architecture performs efficiently when provided with graph data having medium or high load. Since the main objective of the algorithm is to find the shortest path from the source vertex to all other vertices, a directed weighted graph with non-negative weights is used. The graph is stored in a text file and is formatted in a Edge List manner:

```
Number of Vertices Number of Edges
u v weight
u v weight
....
```

Graph input format

The Number of Vertices and Edges are declared at the beginning of the file, each line contains a directed edge from u to v having a non-negative weight. The graph input file is generated with the help of a python script.

```
import random

def generate_graph(num_vertices, num_edges, weight_range,
    output_file="graph_input.txt"):
    edges = set()

    max_edges = num_vertices * (num_vertices - 1)  # Directed graph
        without self-loops
    if num_edges > max_edges:
        raise ValueError(f"Too many edges! Max possible edges for
            {num_vertices} vertices: {max_edges}")

    # Generate random edges
    while len(edges) < num_edges:
        u = random.randint(0, num_vertices - 1)
        v = random.randint(0, num_vertices - 1)
        weight = random.randint(1, weight_range)

        if u != v and (u, v) not in edges:  # Avoid self-loops and
            duplicate edges
            edges.add((u, v, weight))

    # Write to file
    with open(output_file, "w") as f:
        f.write(f"{num_vertices} {num_edges}\n")
        for u, v, weight in edges:
            f.write(f"{u} {v} {weight}\n")

    print(f"Graph with {num_vertices} vertices and {num_edges}
        edges saved to {output_file}.")

# Function call
generate_graph(num_vertices=30000, num_edges=100000,
    weight_range=100, output_file="graph5.txt")
```

Python program to generate a suitable Graph

Three different graph input files are used for this task, **small_scale_graph**, **medium_scale_graph**, and **large_scale_graph**. Benchmarks taken into consideration are:

- **CPU Execution Time:** The time taken by the CPU to execute the SSSP algorithm in ms.
- **DPU Execution Time:** The time taken by the DPUs to execute the SSSP algorithm in ms.
- **CPU-DPU Transfer Time:** The time taken to transfer the graph data to the DPU program.
- **DPU-CPU Transfer Time:** The time taken to transfer the processed graph data back to the CPU.

4.2 Host program analysis

The Host program manages the execution of DPUs, initialization of variables and data structures, calculates the Benchmarks, and reads the graph file input. The Host program contains three key functions, the first function takes the graph text file as input, the second function contains the CPU code for the SSSP algorithm (which will be later used as a baseline to validate the DPU implementation of the algorithm), and the third function manages the DPU execution, partitions the graph and evaluates the benchmarks.

4.2.1 Reading the graph data from a text file

Since the graph used in this task is too large to be declared in an array, it is read from a text file. Below is the function which reads the graph from a text file:

```
void read_graph(const char *filename, int *num_vertices, int
    *num_edges, Edge **edges) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("Failed to open file");
        exit(EXIT_FAILURE);
    }

    fscanf(file, "%d %d", num_vertices, num_edges);
    *edges = (Edge *)malloc((*num_edges) * sizeof(Edge));
    if (!(*edges)) {
        perror("Failed to allocate memory for edges");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < *num_edges; i++) {
        fscanf(file, "%d %d %d", &(*edges)[i].u, &(*edges)[i].v,
            &(*edges)[i].weight);
    }

    fclose(file);
}
```

Function to read the graph from a text input file

The function **void read_graph()** reads the graph from a file and stores it in memory. The function parameter contains the name of the file containing the graph, two different pointers to store number of vertices and edges. The function reads each edge from the file and stores it in the allocated array. Each edge consists of u (source vertex), v (destination vertex), weight (edge weight). It loops **num_edges times**, reading all edges into memory.

4.2.2 CPU Implementation of the SSSP Algorithm

The CPU implementation of the program is written in the **double cpu_sssp()** function in the host program, the purpose of this function is to determine the CPU execution time, the output of this function is used to validate the DPU program for correctness. The algorithm follows the Bellman-Ford approach.

```
double cpu_sssp(int num_vertices, int num_edges, Edge *edges, int
    *cpu_distances) {
    for (int i = 0; i < num_vertices; i++) {
        cpu_distances[i] = INFINITY; // all the distances are set
            to infinity until the nodes have been discovered
    }
```

Operation	Complexity
Initialization	O(V)
Relaxation (V-1 iterations)	O(VE)
Overall Complexity	O(VE)

Table 4.1: Time Complexity of Bellman-Ford [21]

```
    cpu_distances[0] = 0; // vertex 0's distance is set to 0 as it
        is the source node

    double start_time = get_time_in_seconds(); // CPU execution
        time starts

    for (int i = 0; i < num_vertices - 1; i++) { // Perform
        (num_vertices - 1) iterations to relax all edges
        for (int j = 0; j < num_edges; j++) { // Iterate through
            all edges in the graph
            int u = edges[j].u;  // Source vertex of the edge
            int v = edges[j].v; // Destination vertex of the edge
            int weight = edges[j].weight; // Weight of the edge (u
                -> v)

        // Relaxation Step:
        // If the current shortest distance to 'u' is not INFINITY,
        // and the path through 'u' gives a shorter distance to 'v',
        // then update the shortest distance to 'v'.
            if (cpu_distances[u] != INFINITY && cpu_distances[u] +
                weight < cpu_distances[v]) {
                cpu_distances[v] = cpu_distances[u] + weight;
            }
        }
    }

    double end_time = get_time_in_seconds(); // CPU execution time
        ends
    return (end_time - start_time) * 1000; // Execution time in ms
}
```

CPU implementation of the SSSP algorithm

The variable **double start_time** starts the timer to determine the CPU execution time, the algorithm performs **V-1** iterations, each iteration attempts to relax all the edges. The variable **double end_time** stops the timer and computes the execution time of the CPU implementation.

4.2.3 DPU Management in the Host code

In this section of the code the DPUs are allocated and loaded into the DPU program for processing. To make complete utilization of every DPUs, the graph is partitioned across multiple DPUs to ensure parallel processing and reduce the execution time. The edges are divided among the DPUs, and each DPU processes a subset of edges independently.

```
     int32_t edges_per_dpu = (num_edges + NR_DPUS - 1) / NR_DPUS;
    int dpu_id = 0;

    double transfer_start_time = get_time_in_seconds(); // start
       CPU-DPU transfer timer

    // Transfer edge partitions to each DPU
    DPU_FOREACH(set, dpu) {
        int32_t start_idx = dpu_id * edges_per_dpu;
        int32_t end_idx = (start_idx + edges_per_dpu < num_edges) ?
           (start_idx + edges_per_dpu) : num_edges;
        int32_t partition_size = end_idx - start_idx;
        printf("DPU %d is handling edges [%d - %d] (%d edges)\n",
           dpu_id, start_idx, end_idx - 1, partition_size);

        // Transfer edge data to DPU
        DPU_ASSERT(dpu_copy_to(dpu, "edges", 0, &edges[start_idx],
           partition_size * sizeof(Edge)));
        DPU_ASSERT(dpu_copy_to(dpu, "NUM_VERTICES", 0,
           &num_vertices, sizeof(num_vertices)));
        DPU_ASSERT(dpu_copy_to(dpu, "NUM_EDGES", 0,
           &partition_size, sizeof(partition_size)));

        dpu_id++;
    }
    // Transfer initial distances array to DPUs
    DPU_FOREACH(set, dpu) {
        DPU_ASSERT(dpu_copy_to(dpu, "distances", 0, dpu_distances,
           num_vertices * sizeof(int)));
    }
```

Partitioning the data across DPUs

The above code shows the approach used in this project to distribute data among all the DPUs, the total number of edges are divided equally among all the DPUs by **(num_edges + NR_DPUS - 1) / NR_DPUS;**. For partitioning the graph some key variables are declared:
This particular line in the program ensures that the end index does not exceed the number of edges. If the last DPU gets fewer edges, it prevents overflow.

```
    end_idx = (start_idx + edges_per_dpu < num_edges) ? (start_idx
       + edges_per_dpu) : num_edges;
```

Edge partitioning

using **dpu_copy_to()**function the partitioned data is transferred to the MRAM of the DPU. The **DPU_FOREACH** iterates depending on the number of DPUs

Variables	Functionality
num_edges	Total number of edges in the graph
NR_DPUs	Number of DPUs allocated
edges_per_dpu	Number of edges assigned to each DPU
start_idx	The starting index of the edges assigned to a particular DPU
end_idx	The ending index for that DPU's edges

Table 4.2: Variables used in partitioning the graph

and sends the data to all the DPUs. The DPUs are launched and the processing begins in the DPU program, each DPU only updates distances for its assigned edges. The host gathers the updated shortest path from all the DPUs, since each DPU does not process all the edges it has only a partial knowledge of the graph. The dpu_distances are copied back to the DPU program to ensure all DPUs start with the latest shortest path before processing edges again.

```
    for (int iter = 0; iter < num_vertices - 1; iter++) {
       DPU_ASSERT(dpu_launch(set, DPU_SYNCHRONOUS)); // launch DPUs

       bool any_updates = false;
       int temp_distances[num_vertices];

       // retreive update distances from DPUs
       DPU_FOREACH(set, dpu) {
           DPU_ASSERT(dpu_copy_from(dpu, "distances", 0,
              temp_distances, num_vertices * sizeof(int32_t)));

           for (int i = 0; i < num_vertices; i++) {
               if (temp_distances[i] < dpu_distances[i]) {
                   dpu_distances[i] = temp_distances[i];
                   any_updates = true;
               }
           }
       }

       if (!any_updates) {
           break;
       }
        // Copy updated distances back to DPUs for the next
           iteration
       DPU_FOREACH(set, dpu) {
           DPU_ASSERT(dpu_copy_to(dpu, "distances", 0,
              dpu_distances, num_vertices * sizeof(int32_t)));
       }
   }
```

Synchronization between host and DPU

Towards the end of the code the benchmarks are calculated and the DPU result is verified with the CPU result to ensure correctness.

```

    double dpu_to_cpu_start_time = get_time_in_seconds();
    DPU_FOREACH(set, dpu) {
        DPU_ASSERT(dpu_copy_from(dpu, "distances", 0, dpu_distances, num_vertices * sizeof(int)));
    }
    double dpu_to_cpu_end_time = get_time_in_seconds();
    double dpu_to_cpu_time_ms = (dpu_to_cpu_end_time - dpu_to_cpu_start_time) * 1000;

    double end_time = get_time_in_seconds();
    double total_execution_time_ms = (end_time - start_time) * 1000;

    printf("\nComparison of CPU and DPU Results:\n");
    int correct = 0, incorrect = 0;
    for (int i = 0; i < num_vertices; i++) {
        if (cpu_distances[i] == dpu_distances[i]) {
            correct++;
        } else {
            incorrect++;
        }
    }
    printf("Matching distances: %d\n", correct);
    printf("Mismatched distances: %d\n", incorrect);

    printf("\nMetrics:\n");
    printf("CPU Execution Time: %.3f ms\n", cpu_execution_time_ms);
    printf("DPU Execution Time: %.3f ms\n", total_execution_time_ms);
    printf("CPU to DPU Transfer Time: %.3f ms\n", cpu_to_dpu_time_ms);
    printf("DPU to CPU Transfer Time: %.3f ms\n", dpu_to_cpu_time_ms);



    DPU_ASSERT(dpu_free(set));
    free(edges);
    free(dpu_distances);
    free(cpu_distances);

    return 0;
}
```

Benchmark calculation and validation of the DPU program

4.3 DPU Program

The DPU program in this project is responsible for processing the graph and finding the shortest distance from source vertex to all the other vertices in the graph. The host program partitions the graph and sends it to the DPU program to process a subset of the graph, after processing the subset of the graph the shortest path is sent back to the host program. The host program then performs the partitioning once again and sends it to the DPU program, and the process is repeated until the shortest path is found. The DPU program is responsible for management of tasklets, MRAM, and WRAM.

```c
#include <defs.h>
#include <mram.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <barrier.h>

#define MAX_EDGES 2000000
#define MAX_VERTICES 5000000


typedef struct {
    int u, v, weight;
        int32_t padding;
} Edge;

__mram_noinit Edge edges[MAX_EDGES];
__mram_noinit int distances[MAX_VERTICES];
__host int NUM_VERTICES;
__host int NUM_EDGES;



int main() {
    bool updated = true;
    uint32_t tasklet_id = me();
    printf("tasklet_id %u: stack = %u \n", tasklet_id,
        check_stack());


    int edges_per_tasklet = NUM_EDGES / NR_TASKLETS;
    int start = tasklet_id * edges_per_tasklet;
    int end = (tasklet_id == NR_TASKLETS - 1) ? NUM_EDGES : (start
        + edges_per_tasklet);


    for (int iter = 0; iter < NUM_VERTICES - 1; iter++) {
        updated = false;

        for (int i = start; i < end; i++) {
            int u = edges[i].u;
            int v = edges[i].v;
            int weight = edges[i].weight;

            if (distances[u] != 1000000 && distances[u] + weight <
                distances[v]) {
                distances[v] = distances[u] + weight;
                updated = true;
            }
        }

        if (!updated) break;
}

    return 0;
}
```

DPU Program

In the above code MRAM is initialized and stores the shortest path distances and the edges transferred from the host program. Multiple tasklets process a subset of edges. The program is also optimized for early exit **if (!updated) break;**, which means if no updates are made in an iteration, the program exits early, this reduces unnecessary iterations, therefore improving performance.

4.4 Result Analysis

The algorithm was tested on the UPMEM simulator and the real hardware with UPMEM DIMMS, three graph input files were used to evaluate the performance of the DPU compared to the CPU. The DPU implementation's result was compared with the CPU's result to check for correctness or any mismatches in any of the vertices, the DPU's result successfully matched with the CPU's result which ensures correctness of the program. All the graphs used in this project are directed weighted and non-negative. The different use case scenarios are discussed below:

4.4.1 Small scale graph

This graph contains 5000 vertices and 20000 edges. This graph was tested using 16 DPUs and 16 tasklets. The result is displayed below

```
Running SSSP on CPU...
CPU Execution Time: 410.650 ms
First 10 distances:
Vertex 0: CPU = 0, DPU = 0
Vertex 1: CPU = 229, DPU = 229
Vertex 2: CPU = 236, DPU = 236
Vertex 3: CPU = 244, DPU = 244
Vertex 4: CPU = 147, DPU = 147
Vertex 5: CPU = 251, DPU = 251
Vertex 6: CPU = 208, DPU = 208
Vertex 7: CPU = 137, DPU = 137
Vertex 8: CPU = 168, DPU = 168
Vertex 9: CPU = 168, DPU = 168
Comparison of CPU and DPU Results:
Matching distances: 5000
Mismatched distances: 0
Metrics:
CPU Execution Time: 410.650 ms
DPU Execution Time: 266.327 ms
CPU to DPU Transfer Time: 7.477 ms
DPU to CPU Transfer Time: 5.627 ms
```

Running the program on 16 DPUs

From the above output it is observed that the CPU takes 410.650 ms to execute the program, and the DPU takes 266.327ms which is lesser than the CPU execution time, the DPU execution time is the total time taken including the transfer time between CPU and DPU.

4.4.2 Medium scale graph

The Medium scale graph contains 20000 vertices and 50000 edges, for processing this graph more number of DPUs are used. The result is displayed below:

```
Running SSSP on CPU...
CPU Execution Time: 5270.968 ms
First 10 distances:
Vertex 0: CPU = 0, DPU = 0
Vertex 1: CPU = 435, DPU = 435
Vertex 2: CPU = 1000000, DPU = 1000000
Vertex 3: CPU = 396, DPU = 396
Vertex 4: CPU = 304, DPU = 304
Vertex 5: CPU = 448, DPU = 448
Vertex 6: CPU = 367, DPU = 367
Vertex 7: CPU = 416, DPU = 416
Vertex 8: CPU = 611, DPU = 611
Vertex 9: CPU = 430, DPU = 430
Comparison of CPU and DPU Results:
Matching distances: 20000
Mismatched distances: 0
Metrics:
CPU Execution Time: 5270.968 ms
DPU Execution Time: 2066.886 ms
CPU to DPU Transfer Time: 26.280 ms
DPU to CPU Transfer Time: 48.367 ms
```

Running the program on 40 DPUs

The DPU execution time is significantly less than the CPU execution time, 40 DPUs and 16 tasklets were used to process the graph. The first 10 distances of both CPU and DPU are displayed to ensure both implementations are traversing in a correct manner. The CPU execution time is almost double the DPU execution time.

4.4.3 Large scale graph

The Large scale graph is used to test the DPUs potential by using 90 DPUs and 16 tasklets to perform the execution. This graph contains 40000 vertices and 70000 edges. The result is displayed below:

```
Running SSSP on CPU...
CPU Execution Time: 21084.518 ms
First 10 distances:
Vertex 0: CPU = 0, DPU = 0
Vertex 1: CPU = 975, DPU = 975
Vertex 2: CPU = 954, DPU = 954
Vertex 3: CPU = 825, DPU = 825
Vertex 4: CPU = 754, DPU = 754
Vertex 5: CPU = 808, DPU = 808
Vertex 6: CPU = 741, DPU = 741
Vertex 7: CPU = 824, DPU = 824
Vertex 8: CPU = 1000000, DPU = 1000000
Vertex 9: CPU = 672, DPU = 672
Comparison of CPU and DPU Results:
Matching distances: 40000
Mismatched distances: 0
Metrics:
CPU Execution Time: 21084.518 ms
DPU Execution Time: 15117.194 ms
CPU to DPU Transfer Time: 94.697 ms
DPU to CPU Transfer Time: 207.473 ms
```

Running the Program on 90 DPUs

Since the graph is large the execution time also increases but comparing the DPU and CPU execution time, the DPU comparatively takes lesser time than the CPU to perform the execution.

CHAPTER 5

Conclusion and Future Optimizations

5.1 Conclusion

This research project utilizes the Processing in Memory paradigm to leverage graph processing. PIM significantly improves performance over the traditional CPU architecture by processing graphs in memory and reducing data movement bottlenecks and leading to faster execution time. It can be easily scalable by adding more number of DPUs for larger datasets, therefore increasing the execution time. Through the results of this project it is observed that by using the right amount of DPUs according to the size of the dataset processing is more efficient and less time consuming compared to the CPUs. Processing in DPUs also enables the option of parallel processing of the data which can be beneficial in different use cases such as graph processing, data analytics, and machine learning. PIM also avoids unnecessary data transfers, therefore reducing energy usage.

5.2 Future Optimizations

This section discusses the potential optimizations and future work which can be carried alongside with the existing implementation:

- An even larger and complex dataset can be used to explore the UPMEM DPUs complete potential.
- Since the Bellman-Ford algorithm approach is used in this implementation, the DPUs can be optimized to handle negative weights in a graph.
- Partitioning of the graph can be optimized to reduce the data transfer and DPU execution time.

Bibliography

[1] J. Gómez-Luna, I. El Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking memory-centric computing systems: Analysis of real processing-in-memory hardware", in *2021 12th International Green and Sustainable Computing Conference (IGSC)*, 2021, pp. 1–7. DOI: `10.1109/IGSC54211.2021.9651614`.

[2] A. Baumstark, M. A. Jibril, and K.-U. Sattler, "Processing-in-memory for databases: Query processing and data transfer", in *Proceedings of the 19th International Workshop on Data Management on New Hardware*, ser. DaMoN '23, Seattle, WA, USA: Association for Computing Machinery, 2023, pp. 107–111, ISBN: 9798400701917. DOI: `10.1145/3592980.3595323`. [Online]. Available: `https://doi.org/10.1145/3592980.3595323`.

[3] F. Busato and N. Bombieri, "An efficient implementation of the bellman-ford algorithm for kepler gpu architectures", *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2222–2233, 2016. DOI: `10.1109/TPDS.2015.2485994`.

[4] M. J. Bannister and D. Eppstein, "Randomized speedup of the bellman–ford algorithm", in *2012 Proceedings of the Meeting on Analytic Algorithmics and Combinatorics (ANALCO)*, pp. 41–47. DOI: `10.1137/1.9781611973020.6`. eprint: `https://epubs.siam.org/doi/pdf/10.1137/1.9781611973020.6`. [Online]. Available: `https://epubs.siam.org/doi/abs/10.1137/1.9781611973020.6`.

[5] G. Dai, T. Huang, Y. Chi, *et al.*, "Graphh: A processing-in-memory architecture for large-scale graph processing", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 640–653, 2019. DOI: `10.1109/TCAD.2018.2821565`.

[6] W. Zhang, H. Chen, C. Jiang, and L. Zhu, "Improvement and experimental evaluation bellman-ford algorithm", in *Proceedings of the 2013 International Conference on Advanced ICT and Education*, Atlantis Press, 2013/08, pp. 138–141, ISBN: 978-90786-77-79-6. DOI: `10.2991/icaicte.2013.29`. [Online]. Available: `https://doi.org/10.2991/icaicte.2013.29`.

[7] A. Maqbool and M. Kumar, "Study and analysis in dijkstra's algorithm and bellman ford algorithm on run-time basis implementation in angiography system", vol. 16, Apr. 2018.

[8] P. Aguilera, D. P. Zhang, N. S. Kim, and N. Jayasena, "Fine-grained task migration for graph algorithms using processing in memory", in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 489–498. DOI: `10.1109/IPDPSW.2016.205`.

[9] A. V. Jamet, G. Vavouliotis, D. A. Jiménez, L. Alvarez, and M. Casas, "Practically tackling memory bottlenecks of graph-processing workloads", in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024, pp. 1034–1045. DOI: `10.1109/IPDPS57955.2024.00096`.

[10] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems", in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, Seattle, Washington: Association for Computing Machinery, 2011, ISBN: 9781450307710. DOI: `10.1145/2063384.2063471`. [Online]. Available: `https://doi.org/10.1145/2063384.2063471`.

[11] UPMEM, *Upmem use cases*, Accessed: 01-Mar-2025, 2025. [Online]. Available: `https://www.upmem.com/use-cases/`.

[12] UPMEM, *Upmem technology overview*, Accessed: 01-Mar-2025, 2025. [Online]. Available: `https://www.upmem.com/technology/`.

[13] S. Cai, B. Tian, H. Zhang, and M. Gao, "Pimpam: Efficient graph pattern matching on real processing-in-memory hardware", *Proc. ACM Manag. Data*, vol. 2, no. 3, May 2024. DOI: `10.1145/3654964`. [Online]. Available: `https://doi.org/10.1145/3654964`.

[14] B. Hyun, T. Kim, D. Lee, and M. Rhu, "Pathfinding future pim architectures by demystifying a commercial pim technology", in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 263–279. DOI: `10.1109/HPCA57654.2024.00029`.

[15] B. Friesel, M. Lütke Dreimann, and O. Spinczyk, "A full-system perspective on upmem performance", in *Proceedings of the 1st Workshop on Disruptive Memory Systems*, ser. DIMES '23, Koblenz, Germany: Association for Computing Machinery, 2023, pp. 1–7, ISBN: 9798400703003. DOI: `10.1145/3609308.3625266`. [Online]. Available: `https://doi.org/10.1145/3609308.3625266`.

[16] UPMEM, *Upmem sdk toolchain at a glance*, Accessed: 01-Mar-2025, 2025. [Online]. Available: `https://sdk.upmem.com/2025.1.0/00_ToolchainAtAGlance.html`.

[17] UPMEM, *Upmem sdk installation guide*, Accessed: 01-Mar-2025, 2025. [Online]. Available: `https://sdk.upmem.com/2025.1.0/01_Install.html`.

[18] UPMEM, *Upmem sdk - dpu.h api reference*, Accessed: 01-Mar-2025, 2025. [Online]. Available: `https://sdk.upmem.com/2025.1.0/CAPI/dpu_8h.html#afadbfbc268622c7ac416281901b0efad`.

[19] UPMEM, *Upmem sdk - defs.h api reference*, Accessed: 01-Mar-2025, 2025. [Online]. Available: `https://sdk.upmem.com/2025.1.0/DPURT/defs_8h.html#details`.

[20] UPMEM, *Upmem sdk - about dpulldb*, Accessed: 01-Mar-2025, 2025. [Online]. Available: `https://sdk.upmem.com/2025.1.0/080_AboutDPULLDB.html`.

[21] GeeksforGeeks, *Time and space complexity of bellman-ford algorithm*, Accessed: 01-Mar-2025, 2025. [Online]. Available: `https://www.geeksforgeeks.org/time-and-space-complexity-of-bellman-ford-algorithm/`.

Declaration of Originality

I hereby declare that this thesis is my own original work, which has been composed by me without using aids other than those specified. I have clearly referenced all sources, both published or unpublished, which have been directly or indirectly used in the work. This work has not been, previously or concurrently, submitted to any other examination authority.

Ilmenau, 3 March 2025

Krishna Kumar Ranganath