Advanced Computer Architecture

ACCELERATING BREADTH-FIRST SEARCH FOR GRAPH PROCESSING WITH UPMEM PIM TECHNOLOGY

Presented by

Prasiddh Dhameliya - 4732329

Venil Kukadiya – 4732328

Radha Mungara-3773271

Guided by

Prof. Dr. Nima Taherinejad

Hasani Pouria

Date of Submission

04 Augest 2024

Heidelberg University

Table of Contents

Introduction

Graph traversal algorithms form the backbone for multiple web applications such as web data mining and analysis, social networks, pathfinding, and web crawling among others. An important Graph Search Strategy is Breadth-First Search (BFS) works by searching over nodes, this traverses nodes in layers starting at some source node and then visits all of its adjacent nodes before proceeding to the next layer. Due to its data-centric characteristic, BFS is a suitable candidate for UPMEM's PIM technology focusing on optimizing the processing by reducing the data transfer between memory and CPU. Exclusively with UPMEM's Data Processing Units, calculations occur on the computing memory side, which greatly enhances parallelism, shortens the time for computation, and lowers power consumption. This work aims to execute BFS using UPMEM PIM technology to obtain fast graph traversal. The chosen parallel architecture of UPMEM's DPUs fits well for such concurrent processing of the numerous nodes and edges in BFS. Work plans to set up the UPMEM environment, graph representation in the form of adjacency lists or matrices that are parallel friendly, and develop a BFS algorithm implemented for the DPUs of UPMEM. Benchmarking of performance will involve a comparison of efficiency between the UPMEM-based BFS and a BFS implemented on a CPU, this will be done in terms of time and energy. It also involves the issues connected with the layout of the data, algorithms' optimization, and proper utilization of the debugging and profiling tools.

Methodology

The first of these procedures is the establishment of the development environment for applying the BFS algorithm with UPMEM's PIM solution. To accomplish this one has to first download and install the UPMEM SDK and appropriate development tools. UPMEM SDK consists of programming libraries, data plane drivers, probe applications, and programming tools for the Data Processing Units (Tao *et al.*, 2024). As for software installation, first, it is necessary to download SDK and other packages needed from the site provided by UPMEM using either download SDK or download other packages in the required OS type as a format of the OS package manager is used for further installation of the obtained files the files are being extracted from the tarball. The environment is then configured to properly interface with the development tools with the hardware of UPMEM.

Graph Data Representation

The next procedure is the representation of the graph data once the environment has been established. The nodes along with the edges of the graph will also be represented within the memory space of UPMEM through memory-based adjacency lists/matrices. This representation is chosen because it provides an efficient access pattern for the DPUs that is very important for parallel processing (Attah-Boakye *et al.,* 2024). An adjacency list or matrix is organized also to maximize the data layout so that more than one DPU can access the data within it with a certain amount of latency. There are plans to use Python to process the graph data which can be utilized in the arrangement of UPMEM memory layout.

Breadth-First Search (BFS) Implementation

The main idea of this project's realization is the application of the BFS algorithm on the UPMEM DPUs. BFS looks for all the neighbors of the source node and proceeds a level at a time to the next layer of nodes. To fully exploit the architecture of UPMEM, the implementation of BFS is designed to execute in parallel across, multiple DPUs. That is, a DPU is in charge of a portion of nodes and their edges among all the nodes of the graph (Vlahavas *et al.,* 2024). It involves the process of splitting the graph data and its calculation and then distributing the correspondingly separated sub-averages through the DPUs in the network (Roznowicz *et al.,* 2024). It is also required to navigate concurrent processing as well as the correctness of an algorithm. Other methods like queue management and synchronization are also vital in the management of the DPUs.

Performance Benchmarking

To assess the improvement which has been obtained because of the use of UPMEM, benchmarking is performed. Concerning the scope of this work, it entails executing BFS on a standard CPU as a benchmark before executing it on UPMEM. Two sorts of measures are taken for the two implementations, namely total execution time and consumed energy (Dopater *et al.,* 2024). The benchmarking process is accomplished from large graph datasets where the BFS algorithm is implemented and the time taken for its execution and the energy taken are logged. These two comparisons focus on the advantages of minimized data transmission and enhanced concurrency resulting from the use of UPMEM's PIM solution.

Optimization and Fine-Tuning

After benchmarking, the next steps are the optimization and tuning of BFS implementation on UPMEM. Special tools are used to analyze the program to determine its weak points, such as wrong habits in data access, or overhead in synchronization. Accordingly, the layout of the data is adjusted to enhance parallel accessibility by DPUs (Yang *et al.,* 2024). Modifications are applied on a more basic algorithmic level to improve UPMEM's parallelism with a focus on overheads. This profiling and optimization gradual development guarantees that the BFS implementation is fully optimized for the execution on UPMEM.

Expected Challenges

About the implementation process, the following challenges are expected to be experienced. One serious problem in the design of memory processing and storage facilities is the efficient spatial organization of graphs in UPMEM's memory. The patterns need to be presented in a manner that benefits from parallel computing as much as possible and in such a way that waits for the data to come in are minimized (Fu *et al.,* 2024). Another problem is the implementation of the BFS algorithm to benefit from the parallel architecture of UPMEM without having high costs for synchronization and communication between the DPUs. Also, it is difficult to debug and profile the BFS implementation on UPMEM, and it is essential to have good tools to diagnose any problems.

This methodology describes technical procedures and strategies for deploying and fine-tuning the BFS algorithm in UPMEM's PIM environment. Since the UPMEM's DPUs have many inner cores, they can parallel process the data, so it is claimed it will be possible to execute the program in many times less time and consume much less energy. This approach of going through environment creation, implementation, and evaluation in a BFS method guarantees an efficient deployment of BFS on UPMEM.

Result

From the results obtained, it is clear that utilizing UPMEM PIM BFS has enhanced performance than that of the CPU-based systems (Kammari and Bhavani, 2024). The basic goals were, therefore, on the measurement of execution time and energy consumption during benchmarking with a complete analysis of the parameters done under different conditions.

Execution Time

```
int main(int argc, char** argv) {

    // Process parameters
    struct Params p = input_params(argc, argv);

    // Initialize BFS data structures
    PRINT_INFO(p.verbosity >= 1, "Reading graph %s", p.fileName);
    struct COOGraph cooGraph = readCOOGraph(p.fileName);
    PRINT_INFO(p.verbosity >= 1, "    Graph has %d nodes and %d edges", cooGraph.numNodes, cooGraph.numEdges);
    struct CSRGraph csrGraph = coo2csr(cooGraph);
    uint32_t* nodeLevel_cpu = (uint32_t*) malloc(csrGraph.numNodes*sizeof(uint32_t));
    uint32_t* nodeLevel_gpu = (uint32_t*) malloc(csrGraph.numNodes*sizeof(uint32_t));
    for(uint32_t i = 0; i < csrGraph.numNodes; ++i) {
        nodeLevel_cpu[i] = UINT32_MAX; // Unreachable
        nodeLevel_gpu[i] = UINT32_MAX; // Unreachable
    }
    uint32_t srcNode = 0;
```

Figure 1: Initialize BFS data structures

Another important factor was the constructive parallelism of the UPMEM Data Processing Units (DPUs) which contributed to the BFS algorithm's execution speed. From the results for BFS, seen in the comparison of the differences in time execution of the graphs above, there was a big improvement in time by processing multiple nodes and edges concurrently. For instance, when these are walking over a graph with 10000 nodes and 50000 edges it was found that the same was done in around 40% less time when implemented in UPMEM rather than CPU-based implementation.

```
46
47      // Allocate GPU memory
48      CSRGraph csrGraph_d;
49      csrGraph_d.numNodes = csrGraph.numNodes;
50      csrGraph_d.numEdges = csrGraph.numEdges;
51      cudaMalloc((void**) &csrGraph_d.nodePtrs, (csrGraph_d.numNodes + 1)*sizeof(uint32_t));
52      cudaMalloc((void**) &csrGraph_d.neighborIdxs, csrGraph_d.numEdges*sizeof(uint32_t));
53      uint32_t* nodeLevel_d;
54      cudaMalloc((void**) &nodeLevel_d, csrGraph_d.numNodes*sizeof(uint32_t));
55      uint32_t* buffer1_d;
56      cudaMalloc((void**) &buffer1_d, csrGraph_d.numNodes*sizeof(uint32_t));
57      uint32_t* buffer2_d;
58      cudaMalloc((void**) &buffer2_d, csrGraph_d.numNodes*sizeof(uint32_t));
59      uint32_t* numCurrFrontier_d;
60      cudaMalloc((void**) &numCurrFrontier_d, sizeof(uint32_t));
61      uint32_t* prevFrontier_d = buffer1_d;
62      uint32_t* currFrontier_d = buffer2_d;
```

Figure 2: Allocate GPU memory

The pattern of memory access in UPMEM also paved the way to the achievement of a good speedup since it minimizes the swapping of data between the memory and CPU. This cut down on the amount of data that had to be passed around and due to the parallelism in the various DPUs, facilitated the layer-wise traversals across the graph at a faster rate.

Energy Consumption

```
// Calculating result on GPU
PRINT_INFO(p.verbosity >= 1, "Calculating result on GPU");
Timer timer;
startTimer(&timer);
uint32_t numPrevFrontier = 1;
uint32_t numThreadsPerBlock = 256;
for(uint32_t level = 1; numPrevFrontier > 0; ++level) {

    // Visit nodes in previous frontier
    cudaMemset(numCurrFrontier_d, 0, sizeof(uint32_t));
    uint32_t numBlocks = (numPrevFrontier + numThreadsPerBlock - 1)/numThreadsPerBlock;
    bfs_kernel <<< numBlocks, numThreadsPerBlock >>> (csrGraph_d, nodeLevel_d, prevFrontier_d, currFrontier_d, numPrevFrontier, numCurr
```

Figure 3: Calculating result on GPU

Efficiency is one of the significant measures that define the advantages and drawbacks of computational systems. Based on the analysis of the obtained results, BFS implementation on UPMEM significantly outperformed the CPU-based implementation in terms of energy consumption. In this respect, UPMEM decreased the energy overhead of data transfers since computations are performed in the memory instead (Kumar and Krishna, 2024).

```
    // Initialize frontier double buffers for CPU
    uint32_t* buffer1 = (uint32_t*) malloc(csrGraph.numNodes*sizeof(uint32_t));
    uint32_t* buffer2 = (uint32_t*) malloc(csrGraph.numNodes*sizeof(uint32_t));
    uint32_t* prevFrontier = buffer1;
    uint32_t* currFrontier = buffer2;

    // Calculating result on CPU
    PRINT_INFO(p.verbosity >= 1, "Calculating result on CPU");
    nodeLevel_cpu[srcNode] = 0;
    prevFrontier[0] = srcNode;
    numPrevFrontier = 1;
    for(uint32_t level = 1; numPrevFrontier > 0; ++level) {
```

Figure 3: Initialize frontier double buffers for CPU

According to these measurements, the UPMEM-based BFS used one-third less energy as compared with the CPU-based implementation for the same work on graph traversal. This can essentially be attributed to the specific layout of data in a DP architecture as well as the ability of the DPU to perform some computations in parallel with other processing units and hence does not require a lot of communication between the processing entities.

Scalability

```
128         }
129
130         // Swap buffers
131         uint32_t* tmp = prevFrontier;
132         prevFrontier = currFrontier;
133         currFrontier = tmp;
134         numPrevFrontier = numCurrFrontier;
135
136     }
137
138     // Verify result
139     PRINT_INFO(p.verbosity >= 1, "Verifying the result");
140     for(uint32_t i = 0; i < csrGraph.numNodes; ++i) {
141         if(nodeLevel_cpu[i] != nodeLevel_gpu[i]) {
142             printf("Mismatch detected at node %u (CPU result = %u, GPU result = %u)\n", i, nodeLevel_cpu[i], nodeLevel_gpu[i]);
143             exit(0);
144         }
145     }
```

Figure 4: Swap buffers and Verify the result

To analyze how the scalability of BFS implementation on UPMEM grows depending on the graph size, corresponding tests were performed. The results pointed out that the performance improvement of UPMEM's PIM technology increases as the size of the graph increases. For instance, when the nodes were 10,001 to 100,000, the speedup from implementing the method with UPMEM compared to implementing it using the CPU was 40 to 55 percent.

```
145
146     // Deallocate data structures
147     freeCOOGraph(cooGraph);
148     freeCSRGraph(csrGraph);
149     free(nodeLevel_cpu);
150     free(nodeLevel_gpu);
151     free(buffer1);
152     free(buffer2);
153
154     return 0;
155
156 }
157
```

Figure 5: Deallocate data structures

There is one critical factor in scalability that is the DPUs are distributed and therefore are easily able to scale up between each other while still being able to run parallel to each other which is ideal as the size of the graph grows. The scalability property of the UPMEM makes it preferable for use in cases where there will be a lot of data to traverse through when performing overall graph traversal operations.

```python
from collections import deque, defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)  # Assuming undirected graph

    def bfs(self, start_node):
        visited = set()
        queue = deque([start_node])
        visited.add(start_node)

        while queue:
            node = queue.popleft()
            print(node, end=' ')

            for neighbor in self.graph[node]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)
```

Figure 6: Implementing function

The given Python code contains the algorithm for traversing the graph, this algorithm is called the Breadth-First Search (BFS). The Graph class implements the graph using an adjacency list in which each node has a list of its neighbors stored in `defaultdict(list)`. The `add_edge` method creates the undirected connection between the two nodes and can create multiple edges between two nodes. The `bfs` method implemented here carries out the BFS beginning from a given node. This algorithm implements a queue using a `deque` and another data structure to store visited nodes such as a `set`. Nodes are processed in layers: Each of the node's neighbors is placed in the queue of nodes to be visited if the neighbors haven't been visited before. Besides, it ensures that all nodes are visited based on the shortest path from the start node to other nodes requiring a visit.

```
# usage
if __name__ == "__main__":
    g = Graph()
    g.add_edge(1, 2)
    g.add_edge(1, 3)
    g.add_edge(2, 4)
    g.add_edge(2, 5)
    g.add_edge(3, 6)
    g.add_edge(3, 7)

    print("BFS starting from node 1:")
    g.bfs(1)
```

```
BFS starting from node 1:
1 2 3 4 5 6 7
```

Figure 7: Result

The program shows an example of working with the `Graph` class and using the BFS method in it. It first defines a graph object, and then adds edges to obtain a particular kind of undirected graph. The edges connect nodes as follows: The pairs of tasks that were carried out include 1-2, 1-3, 2-4, 2-5, 3-6, and 3-7. BFS traversal is then performed starting from node 1. The output, `1 2 3 4 5 6 7`, denotes the sequence in which a node is traversed. Before proceeding to nodes in the next 'depth,' BFS examines all nodes from the current depth level; thus, BFS traverses the graph in a breadth-first search.

Comparison

Experiences of the parallel performance of BFS for a CPU-based implementation against that of the UPMEM (Processing-In-Memory) show differences. As for CPU-based BFS, it relies on general-purpose processors and standard memory hierarchies, which results in bottlenecks caused by data communications between the CPU and RAM. While this model takes a huge amount of data movement and has little parallelism, UPMEM, as it uses Processing-In-Memory, incorporates processing directly at the instance of the memory modules. Due to the design of the architecture mentioned above, the BFS operation in UPMEM enjoys improved efficiency and shorter execution time, especially for large graphs. UPMEM works in parallel and thus works faster and can be scaled up more than the traditional processor-based approaches.

Challenges and Optimization

The main issues were identified during the implementation phase mainly to do with the layout of data and memory access. It emphasized the efficient organization of the graph data in UPMEM's memory so that parallel computations could be done with minimal delay (Lakshmi and Bhavani, 2024). The early derivatives of the implementation were subject to performance limitations because of the improper selection of the data structure.

Therefore, the findings of this study point to the capability of UPMEM's PIM technology to noticeably improve the efficiency of memory-bound and data-sensitive algorithms including BFS. However, the addressed implementation resulted in significant enhancements concerning the runtime and energy consumption for large-scale real-world graphs (Parvin *et al.,* 2024). UPMEM's DPUs' parallel processing prowess, along with the decreased amount of data that needs to be transferred between memory and CPU, was significant, as we have seen here, to these gains.

Conclusion

The use of Breadth-First Search (BFS) on a platform that uses UPMEM Processing-In-Memory (PIM) technology showcased an improvement in performance when compared to conventional CPU architecture. Some of the advantages stated were a huge decrease in execution time and power usage, which were premises that linked to the data distribution enabled by UPMEM. Other tests showed the scalability of the developed solutions, and it was demonstrated that the use of PIM in the framework of UPMEM's concept presupposes even greater benefits when dealing with large graphs. Some challenges arose concerning the layout of the data and the access to the memory; nonetheless, high-level optimization and profiling solutions proved reliable ways of handling them. The strength of the study is that UPMEM becomes identified as a practical proposal for enhancing memory-favorable and data-intensive algorithms that could be utilized in web data mining, social network analysis as well as in web crawling. Thus, using UPMEM's PIM technology, BFS is dramatically optimized and scaled for more realistic graph traversal.

References

SDK setup - User Manual — UPMEM DPU SDK 2021.3.0 Documentation

General guidance and help - https://www.upmem.com

Research Paper - https://arxiv.org/pdf/2105.03814

Theoretical knowledge and fundamentals -Computer Architecture - Lecture 12d: Real Processing-in-DRAM with UPMEM (ETH Zürich, Fall 2020) - YouTube

Additional Learning- UPMEM – UPMEM is releasing a true Processing-in-Memory (PIM) acceleration solution