



AMERICAN UNIVERSITY OF BEIRUT

EXPLORING PROCESSING-IN-MEMORY  
ACCELERATION OF BREADTH-FIRST  
SEARCH

by

TAREK GHAITH ABDEL SATER

A thesis

submitted in partial fulfillment of the requirements  
for the degree of Master of Science  
to the Department of Computer Science  
of the Faculty of Arts and Sciences  
at the American University of Beirut

Beirut, Lebanon  
February 2021

AMERICAN UNIVERSITY OF BEIRUT

EXPLORING PROCESSING-IN-MEMORY  
ACCELERATION OF BREADTH-FIRST  
SEARCH

by

TAREK GHAITH ABDEL SATER

Approved by:

---

Dr. Izzat El Hajj, Assistant Professor

Advisor

Computer Science

---

Dr. Haidar Safa, Professor

Member of Committee

Computer Science

---

Dr. Wassim El Hajj, Associate Professor

Member of Committee

Computer Science

Date of thesis defense: November 26, 2020

# AMERICAN UNIVERSITY OF BEIRUT

## THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: Abdel Sater Tarek Ghaith  
Last First Middle

Master's Thesis       Master's Project       Doctoral Dissertation

I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes after: **One \_\_\_ year from the date of submission of my thesis, dissertation or project.**  
**Two \_\_\_ years from the date of submission of my thesis , dissertation or project.**  
**Three \_\_\_ years from the date of submission of my thesis , dissertation or project.**

  
\_\_\_\_\_  
Signature

February 3, 2021

\_\_\_\_\_  
Date

# Acknowledgements

Foremost, I would like show my greatest appreciation to my thesis advisor Dr. Izzat El Hajj for giving me the opportunity to do research and for his guidance throughout this work. He has introduced me to new concepts that seemed complex, yet his explanations always managed to make things very simple and crystal clear. His feedback and suggestions were invaluable to the completion of this work.

I would also like to express my sincere gratitude to the rest of the committee members: Prof. Haidar Safa and Dr. Wassim El Hajj for providing their encouragement and insightful comments throughout the course of my study.

Finally, I would like to thank my family for always standing by me and providing their moral support and warm encouragements. Without them, I would not have had the motivation to work as hard as I possibly could and to the best of my ability.

# An Abstract of the Thesis of

Tarek Ghaith Abdel Sater for Master of Science  
Major: Computer Science

Title: Exploring Processing-In-Memory Acceleration of Breadth-First Search

In the traditional computer architecture, some applications suffer from the “memory wall” problem - workloads that exhibit dominant data movement costs and low data reuse would fully saturate the memory throughput of the system. This leaves the processor waiting on memory for the majority of the computation, thereby incurring significant loss in performance and energy efficiency. Processing-In-Memory (PIM) is an emerging technology that aims to overcome this problem by performing the computation in processors sitting very close to the memory. The massive reduction in latency and increase in total throughput opens up the possibility of accelerating such workloads. Breadth-First Search is a graph application that gets increasingly affected by the memory-wall problem as it scales, and could therefore benefit considerably by taking advantage of this architecture. In this work, we accelerate BFS workloads using UPMEM’s PIM architecture Data Processing Units (DPU). We characterize the performance and scalability of BFS on DPUs and compare with the traditional CPU-based implementation. The objective of this work is to explore PIM acceleration of BFS.

# Contents

<b>Acknowledgements</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>3</b>
2.1 DPU Hardware Architecture . . . . .	3
2.1.1 UPMEM’s PIM-DRAM . . . . .	3
2.1.2 Data Processing Unit . . . . .	4
2.1.3 Technical Specifications . . . . .	5
2.2 DPU Software Architecture . . . . .	6
2.2.1 Application Program Interface . . . . .	6
2.2.2 Programming Model . . . . .	6
2.3 DPUs for Breadth-First Search . . . . .	6
2.3.1 Why DPUs . . . . .	6
2.3.2 Why Breadth-First Search . . . . .	7
2.3.3 Challenges . . . . .	7
2.4 Data Representations of Graphs . . . . .	7
<b>3 BFS Algorithms</b>	<b>11</b>
3.1 Overall Strategy . . . . .	11
3.2 Graph Partitioning Strategies . . . . .	13
3.2.1 Row Partitioning . . . . .	13
3.2.2 Column Partitioning . . . . .	14
3.2.3 2D Partitioning . . . . .	14
3.3 Parallelization Strategies . . . . .	15
3.3.1 Vertex-centric Top-Down BFS . . . . .	15
3.3.2 Vertex-centric Bottom-Up BFS . . . . .	17
3.3.3 Edge-centric BFS . . . . .	17
3.4 The 9 BFS Algorithms . . . . .	19

<b>4</b>	<b>Methodology</b>	<b>20</b>
4.1	Environment . . . . .	20
4.2	Programs . . . . .	20
4.3	Measurements . . . . .	21
4.4	Graphs . . . . .	21
4.4.1	Graph Properties . . . . .	22
4.4.2	Graphs Used for the Strong Scaling Benchmarks . . . . .	22
4.4.3	Graphs Used for the Weak Scaling Benchmarks . . . . .	23
<b>5</b>	<b>Evaluation</b>	<b>25</b>
5.1	Strong Scaling Benchmarks . . . . .	25
5.2	Strong Scaling Analysis . . . . .	37
5.2.1	Host Time Overhead . . . . .	37
5.2.2	BFS DPU Time . . . . .	38
5.2.3	BFS Total Time . . . . .	40
5.3	Weak Scaling Benchmarks . . . . .	41
5.4	Weak Scaling Analysis . . . . .	43
5.4.1	Constant Average Degree Graphs . . . . .	43
5.4.2	Constant Sparsity Graphs . . . . .	43
5.5	CPU BFS Comparison . . . . .	45
<b>6</b>	<b>Related Work</b>	<b>48</b>
6.1	Motivation for Processing-In-Memory . . . . .	48
6.2	Processing-In-Memory Architectures . . . . .	49
6.2.1	DRISA . . . . .	49
6.2.2	Tesseract . . . . .	50
6.2.3	GraphH . . . . .	50
6.2.4	Memristor-based PIM Architectures . . . . .	51
6.2.5	Other Works Using UPMEM Acceleration . . . . .	51
<b>7</b>	<b>Conclusion</b>	<b>52</b>

# Chapter 1

## Introduction

In this thesis, we are concerned with the memory wall issue; a consequence of the ever-increasing processor/memory performance gap [1]. Due to industry trends, the performance of processors is increasing at a much faster rate than the performance of memory. The more cores on a chip, the narrower the memory channel available for each core. If the memory throughput becomes insufficient to provide the processor with data fast enough to continue computation, it would have to wait on memory for the majority of the time. This issue is exacerbated when running workloads that naturally exhibit dominant data movement costs and poor data reuse.

Dominant data movement costs is when the duration of data transfer between the memory and the processor makes up the biggest fraction of the total computation time. In other words, the time it takes to move the data far exceeds the time it takes to process it.

Poor data reuse is a big concern; it happens when the data cached inside the processor is not reused. A processor typically fetches large chunks of data from memory, which are then stored in the cache hierarchy. The expectation is that if data is used once, either the same or neighboring data in the chunk is likely to be requested next, which will be available in the cache. As the processor-cache throughput is extremely large, data movement is not a issue. However, when the nature of the workload does not call for reuse (for example, the data is only processed once, or the data access pattern is highly random), then the caches miss (adding memory-latency), and a new block would have to be fetched from memory. As CPU-Memory channel has limited bandwidth and significant latency, the memory-throughput is rapidly exhausted.

The combination of these issues: limited memory-throughput, dominant data movement costs, and poor data reuse, leaves the memory-throughput saturated and the processor underutilized. This results in severely impacting the energy efficiency and performance of the workload.

Breadth-First Search is one such application that gets increasingly affected by the memory-wall problem as it scales. In BFS, every edge in a graph visited only

once per iteration. Therefore the cached data is rarely reused in subsequent iterations. In addition, marking a vertex as visited is computationally inexpensive, so data movement dominates the performance cost.

Processing-In-Memory architectures aim to overcome the memory-wall issue by physically moving the processing closer to the data. By reducing the distance between the memory and the processor, the memory-latency is reduced and the memory-throughput is massively increased. As a result, data movement is virtually eliminated.

In our work, we will be using the UPMEM’s PIM architecture [2]. This variation of PIM aims to eliminate data movement by embedding processors in the DRAM memory chips. The data would no longer have to travel to the CPU, instead it would be processed in the memory itself. These processors are called Data Processing Units (DPU for short). Each chunk of memory (64MB as of writing) is embedded with a DPU responsible for processing the data within that chunk.

In order to take advantage of this architecture, applications have to be rewritten with DPUs in mind. The programming model changes; an application would now consist of a host program running on the CPU, and a task program that running on each DPU. The host orchestrates and dispatches commands to the DPUs, which process the data in-memory. The developer is responsible for writing both programs as well as ensuring proper DPU orchestration and Host/DPU communication. DPUs cannot directly inter-communicate, so they rely on the Host for synchronization. This unconventional programming model presents a new challenge for software development.

In this work, we create DPU-based implementations of Breadth-First Search. We try 3 graph partitioning strategies and 3 BFS parallelization strategies for a combined total of 9 variations of BFS. We benchmark our BFS-DPU implementations on a diverse set of graphs in order to characterize its performance, strong scaling, and weak scaling behaviour. We compare and contrast the DPU-based approach with a traditional sequential CPU-based approach, so we can determine its effectiveness. Our objective is to explore in-depth the DPU acceleration of BFS.

# Chapter 2

## Background

In this chapter we discuss in-depth the hardware and software architecture of UPMEM’s DPU, the programming model for building DPU-based applications, and how we can use DPUs to accelerate graph-processing workloads.

### 2.1 DPU Hardware Architecture

#### 2.1.1 UPMEM’s PIM-DRAM

Standard DRAM modules (also known as DIMMs) that are found in most computers today consists mainly of a printed-circuit board on which memory chips are mounted. UPMEM’s PIM-DRAM modules (shown in Figure 2.1) build upon the same DRAM technology, by embedding data processors (DPU) inside the memory chips themselves, reducing off-chip data movement and turning the modules into PIM accelerators.

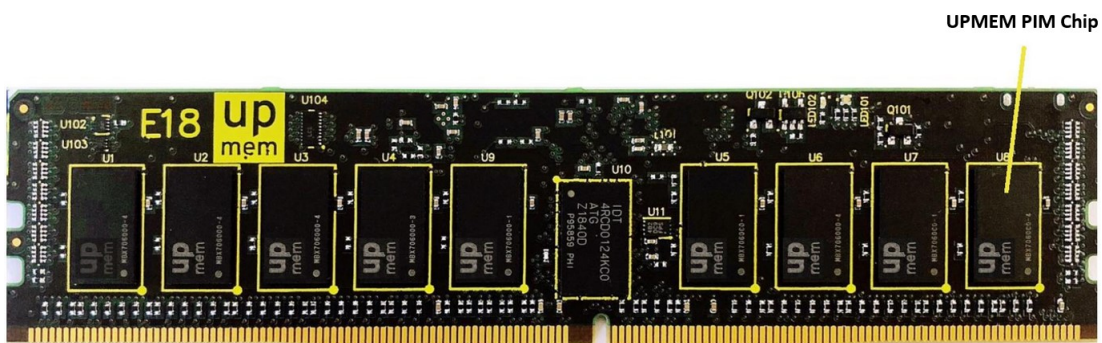


Figure 2.1: Physical view of UPMEM’s 8GB PIM-DRAM Module. Contains 16 PIM chips. Some chips are on the other side [3].

An 8GB UPMEM PIM-DRAM module contains 16 PIM chips. Each PIM chip contains 8 DPUs, and each DPU consists of 64MB of DRAM and a processor.

Therefore a single PIM-DRAM module contains dozens of DPUs, depending on the total memory capacity. For example, an 8 GB module contains 128 DPUs.

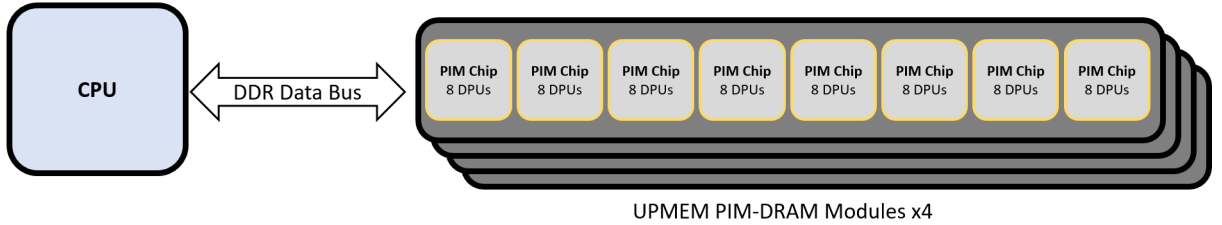


Figure 2.2: Architectural view of a system using four 8GB UPMEM PIM-DRAM modules. There are 16 PIM chips on each module, for a total of 512 DPUs. We perform our benchmarks on a similar system [3].

### 2.1.2 Data Processing Unit

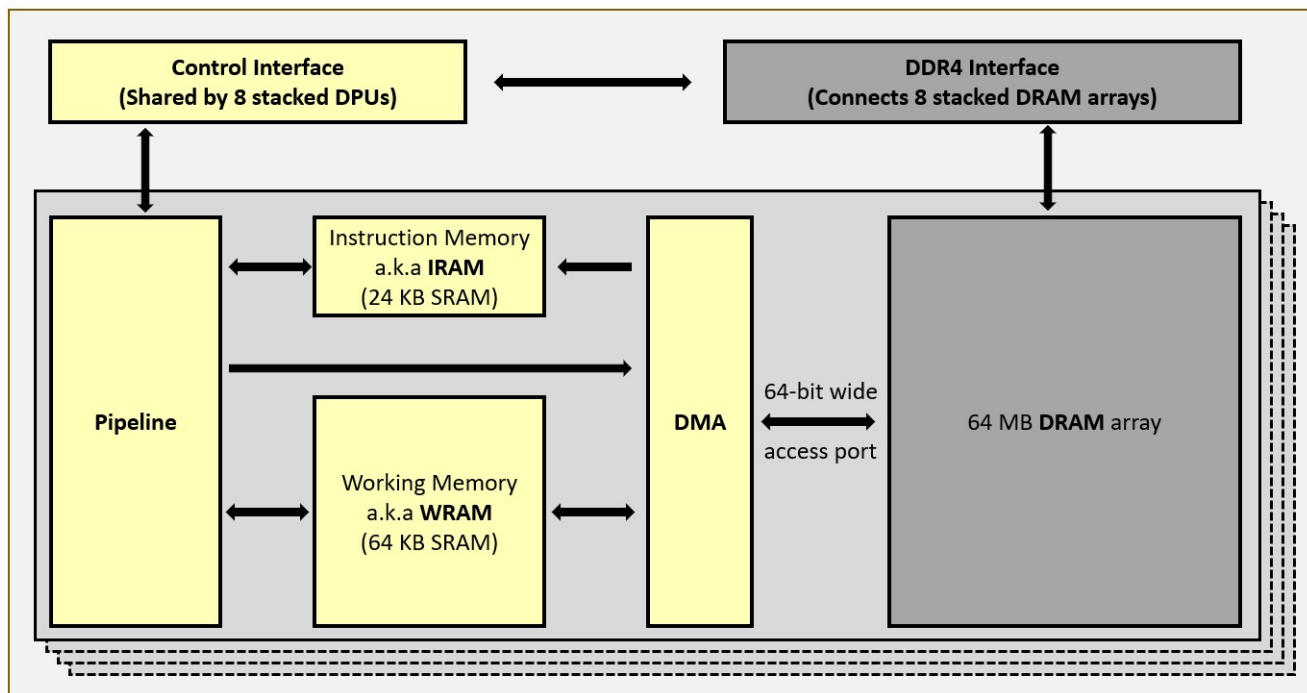


Figure 2.3: Block view of a PIM chip containing 8 stacked DPUs and their internals, along with a Control Interface for Host communication, and a DDR4 Interface for data transfer [3].

A Data-Processing Unit is a miniature general-purpose RISC processor [3]. It does not run an operating system, only low-level drivers. This keeps DPUs under the total control of the host application running on the main CPU. DPUs can

only access their assigned 64 MB of the main memory (hereby referred to as MRAM), therefore they cannot inter-communicate and would have to rely on the host for communication. Host-DPU communication happens over the regular memory channel.

The Control Interface and DDR4 Interface are shared by all DPUs in a PIM chip. The host uses the Control Interface for sending commands to the DPUs (via an API), and the DDR4 Interface for data transfer.

While the DPU is located near the MRAM, the DPU itself contains a small amount of fast SRAM (typically used for CPU caches). This SRAM is split into the IRAM, and the WRAM.

The IRAM (Instructions RAM) contains the task program running on the DPU, while the WRAM (Working RAM) serves as a data cache (allocations, stack, heap). From the DPU's perspective, the WRAM is the main memory, while the MRAM is a secondary memory that can only be accessed through DMA instructions. It is important to note that data in MRAM persists among multiple DPU executions, as well as any data stored in static memory (i.e. global variables). The programmer should take this into account.

### 2.1.3 Technical Specifications

The following are the relevant technical specifications of a DPU [3]:

- Up to 500 MHz clock rate (in this paper, we are running at 267MHz)
- 32-bit instruction set architecture
- 24 hardware threads available.
- Interleaved pipeline processor.
- 1 instruction/cycle throughput.
- 64-bit wide access port to MRAM.
- 1 GB/s MRAM throughput.
- 88 KB of SRAM, split into
  - 64 KB of WRAM data cache.
  - 24 KB of IRAM instruction cache.

An 8 GB module would have would have a total of 2 TB/s memory throughput with its DPUs. This lends credit the claim that on-chip data movement is virtually eliminated. However, there is still latency associated with memory access. Therefore, a program running on a DPU is typically compute-bound.

The use of multithreading is recommended to help hide the memory access latency. It is recommended to use between 11 and 16 tasklets (threads) to achieve nominal performance [4]. In our implementations, we use 11 tasklets.

## 2.2 DPU Software Architecture

### 2.2.1 Application Program Interface

Data-Processing Units do not run an OS; instead they are managed by the CPU through a C API. UPMEM provides a C SDK [5] for writing DPU-based applications. DPU ranks are mapped by the SDK onto the *sets* abstraction. A set contains one or more DPUs loaded with the same program, and is responsible, as a whole, to run a single workload.

### 2.2.2 Programming Model

In order to leverage the potential of DPUs, one would have to write at least two programs: a “host” program that runs on the CPU, and a “task” program that runs on the DPUs. The host is required to orchestrate and manage the DPUs, while the task performs the bulk of the computation.

A host application typically allocates a DPU set, loads the desired task program into the DPUs, and populates/initializes the MRAM with the data to be processed. The host then launches the DPU set, signaling each DPU to execute the program. The tasks process the data in-memory, and keep the results in MRAM. The host could then collect and aggregate the results from MRAM of each DPU.

A task program loads the data from MRAM into the local WRAM cache for processing. There is a small latency cost associated with memory access. Fortunately, multi-threading using tasklets can be used to negate this cost.

Tasklets are logical threads that are mapped onto the hardware threads. Each tasklet has a separate place in the WRAM to store its execution stack, therefore only global variables are shared among tasklets. Using at least 11 active tasklets allows us to reach the DPU’s nominal performance of 1 instruction per cycle.

## 2.3 DPUs for Breadth-First Search

### 2.3.1 Why DPUs

For our research, we opted for UPMEM’s DPU technology over other PIM implementations for multiple reasons:

1. UPMEM’s PIM-DRAM are available for use and will soon enter mass-production.

2. DPUs are general purpose RISC processors; programming the task themselves is straightforward.
3. The provided C SDK, emulator, and documentation are stable and of good quality.

### 2.3.2 Why Breath-First Search

Breadth-First Search serves as a good subject to study the capabilities of PIM, seeing as:

1. Graph applications are typically data intensive workloads.
2. BFS is a good representative of popular graph-processing workloads.
3. BFS suffers from the memory-wall issue.

If BFS benefits from Processing-In-Memory, then other graph applications may benefit similarly.

### 2.3.3 Challenges

When it comes to graph-processing on DPUs, several challenges will have to be addressed:

1. Data partitioning: splitting the graph vertices and edges effectively over the DPUs.
2. Load balancing: spreading the computation fairly over the DPU tasklets, in a way to maximize the total performance of the algorithm per DPU.
3. Synchronization: as DPUs cannot inter-communicate, problems can manifest such as neighbor-traversal between vertices on different DPUs.

## 2.4 Data Representations of Graphs

### Adjacency Matrix

Graphs are typically represented as an adjacency matrix 2.4: a square matrix where the elements indicate whether pairs of vertices in the graph are adjacent (i.e. linked by an edge). The Row-indexes represent the source-vertices while the Column-indexes represent the neighbors destination-vertices. The value of a cell indicates the presence and weight of an edge from the source-vertex to the destination-vertex.

If there is no edge between two vertices, the value of the corresponding element is 0, otherwise it is the weight of the edge. If the graph is unweighted, the value of an edge is 1. If the graph is undirected, the adjacency matrix is symmetric.

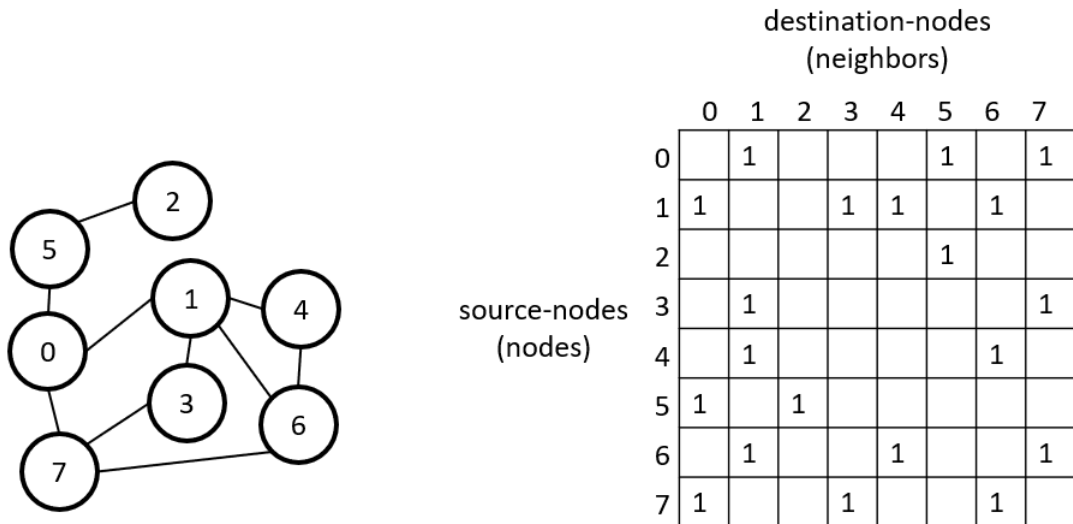


Figure 2.4: Adjacency matrix of a graph of 8 vertices. The zeroes are omitted for better visibility.

An adjacency matrix is typically a sparse matrix; it is stored in memory as an adjacency list, also known as the Coordinate-wise (COO) representation. COO can be further compressed into Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) representations, depending on the BFS algorithm being used.

In this paper, we will be focusing exclusively on undirected, unweighted graphs, with no self-loops. We do not need to store the weight of the edges in our data structures, as each edge has weight 1.

### Coordinate-wise Storage

The COO representation of an adjacency matrix is a data structure composed of the following arrays:

- Row[N]: contains the row-index of the non-zero edges (source-vertices).
- Col[N]: contains the column-index of the non-zero edges (destination-vertices).

Where  $N$  is the number of vertices in the graph. COO is most suitable when iterating over the edges.

rows	0	0	0	1	1	1	1	2	3	3	4	4	5	5	6	6	6	7	7	7
columns	1	5	7	0	3	4	6	5	1	7	1	6	1	2	1	4	7	1	3	6

Figure 2.5: Coordinate-wise representation of the graph.

### Compressed Sparse Row

The CSR representation of an adjacency matrix is a data structure composed of the following arrays:

- RowPtr[N]: contains the row-index range of the non-zero edges (source-vertices).
- Col[N]: contains the column-index of the non-zero edges (destination-vertices).

Where  $N$  is the number of vertices in the graph. CSR is most suitable when iterating over the source-vertices.

rowIdx	0	1	2	3	4	5	6	7	8											
rowPtr	0	3	7	8	10	12	14	17	20											
columns	1	5	7	0	3	4	6	5	1	7	1	6	1	2	1	4	7	1	3	6

Figure 2.6: Compressed Sparse Row representation of the graph.

### Compressed Sparse Column

The CSC representation of an adjacency matrix is the transpose of the CSR representation. The data structure is composed of the following arrays:

- ColPtr[N]: contains the column-index range of the non-zero edges (destination-vertices).
- Row[N]: contains the row-index of the non-zero edges (source-vertices).

Where  $N$  is the number of vertices in the graph. CSC is most suitable when iterating over the destination-vertices.

colIdx	0	1	2	3	4	5	6	7	8											
colPtr	0	3	7	8	10	12	14	17	20											
rows	1	5	7	0	3	4	6	5	1	7	1	6	1	2	1	4	7	1	3	6

Figure 2.7: Compressed Sparse Column representation of the graph.

The CSC and CSR representations are identical in the case of undirected graphs, since the adjacency matrix is square and symmetrical. CSC becomes useful when the matrix is partitioned into non-square sub-matrices, as we will see in the next section.

# Chapter 3

## BFS Algorithms

In this section, we explore the PIM acceleration of Breadth-First Search. We employ 3 different BFS parallelization strategies and 3 different graph partitioning strategies, combined for a total of 9 variations of BFS. In the evaluation section, we benchmark these variations and compare them against each other and against reference CPU and GPU based implementations. All BFS implementation code written for this research is available in our GitHub repository [6].

### 3.1 Overall Strategy

In order to parallelize the BFS computation over hundreds of Data Processing Units, we start by partitioning the graph data (vertices and edges) according to the partitioning strategy, then we convert the partitions into the desired representation (CSR/CSC/COO) according to the parallelization strategy. After that, we populate the MRAM of every DPU with the minimum data it needs to compute its part of the BFS. The initialization steps can be summarized as follows:

1. Load adjacency matrix of graph as COO.
2. Partition COO by the number of DPUs (w.r.t partitioning strategy).
3. Convert COO partitions to CSR or CSC (w.r.t parallelization strategy).
4. Populate the MRAM of each DPU with a partition and the required meta-data.

Regardless of the BFS strategies used, there is a need to overcome the Synchronization challenge: seeing as the DPUs cannot inter-communicate, vertex-traversal on each DPU is inherently incomplete since a DPU only has a fraction of the total vertices in its MRAM. Additionally, if a DPU visits a vertex, that fact may not be known to other DPUs.

We overcome this problem by using the Host (CPU) to maintain the *next frontier* bitarray across all DPUs. The next frontier contains all the vertices that are to be traversed in the next BFS iteration. There is a Host-side “global” next frontier, and a DPU-side “local” next frontier that must be synced each iteration. The Host is therefore in control of the “outer” BFS loop wherein each iteration is an execution of the DPUs on the current BFS level.

Each iteration, every DPU uses its next frontier to update the local visited vertices array. The next frontier is then copied to the current frontier bitarray and cleared. Following the selected parallelization strategy, the DPU traverses vertices in its current frontier and adds vertices to the now clear next frontier. When all DPUs are done, the Host collects, merges, and redistributes the next frontiers to the DPUs. The BFS level is incremented, and the process re-iterates until the “global” next frontier is empty, indicating that all vertices have been traversed. The pseudo-code in Algorithms 1 and 2 demonstrate the synchronization logic executed on the host side and the DPU side respectively.

---

**Algorithm 1** BFS-Host

---

```

1: next_frontier  $\leftarrow$  make_bitarray(total_vertices)
2: next_frontier.set(0) ▷ Visit root vertex
3: level  $\leftarrow$  0
4: while true do
5:   launch_dpuses() ▷ executes Algorithm 2
6:
7:   clear(next_frontier)
8:   frontiers  $\leftarrow$  copy_from_dpuses(“next_frontier”)
9:
10:  next_frontier.aggregate(frontiers)
11:    ▷ Union/Concatenate/Both (Row/Col/2D-partitioning)
12:
13:  if next_frontier is empty then
14:    break
15:  end if
16:
17:  level ++
18:  copy_to_all_dpuses(level, “level”)
19:  copy_to_all_dpuses(next_frontier, “next_frontier”)
20:    ▷ slice to copy depends on partitioning strategy.
21: end while

```

---

The lengths of the DPU-side next frontier and current frontier varies depending on the partitioning strategy; a DPU often only needs a specific slice of the bitarrays each level. Therefore, in practice, there are 3 minor variations of Algorithm 1, one for each partitioning strategy (discussed in the next section).

---

**Algorithm 2** BFS-Init DPU Initialization

---

```
1:  $current\_frontier \leftarrow clone(next\_frontier)$ 
2: for  $chunk$  in  $next\_frontier$  do
3:    $visited\_vertices[chunk] \mid = next\_frontier[chunk]$ 
4:   for  $vertex$  in  $next\_frontier$  do
5:     if  $vertex$  is set then
6:        $vertex\_levels[vertex] = current\_level$   $\triangleright$  Update vertex level if it
       had just been visited.
7:     end if
8:   end for
9: end for
10:  $clear(next\_frontier)$   $\triangleright$  After init we are ready to run BFS-DPU
```

---

## 3.2 Graph Partitioning Strategies

We discuss the strategies for partitioning the graph of  $V$  vertices and  $E$  edges over  $D$  Data Processing Units.

### 3.2.1 Row Partitioning

Row partitioning is where the adjacency matrix is partitioned by row into  $D$  partitions. Each partition is converted to the desired graph representation (COO/CSR/CSC) and sent to the MRAM of a DPU. The length of the next frontier is  $V$ , while the length of the current frontier is  $V/D$ . The *aggregate()* function unions the next frontiers of each DPU into a single global next frontier as shown in Figure 3.1.

		destination-vertices (neighbors)							
		0	1	2	3	4	5	6	7
source-vertices (nodes)	0		1				1		1
	1	1			1	1		1	
	2						1		
	3		1						1
	4		1					1	
	5	1		1					
	6		1			1			1
	7	1			1			1	

$\downarrow$  union next frontiers

Figure 3.1: Row-partitioning of an adjacency matrix over 4 DPUs. The global next frontier of each iteration is the union of the next frontiers of each DPU.

### 3.2.2 Column Partitioning

Column partitioning is where the adjacency matrix is partitioned by column into  $D$  partitions. Each partition is converted to the desired graph representation (COO/CSR/CSC) and sent to the MRAM of a DPU. The length of the next frontier is  $V/D$ , while the length of the current frontier is  $V$ . The *aggregate()* function concatenates the next frontiers of each DPU into a single global next frontier as shown in Figure 3.2.

		destination-vertices (neighbors)								
		0	1	2	3	4	5	6	7	
source-vertices (nodes)	0		1					1		1
	1	1			1	1		1		
	2						1			
	3		1						1	
	4		1					1		
	5	1		1						
	6		1			1			1	
	7	1			1			1		

concatenate next frontiers

Figure 3.2: Column-partitioning of an adjacency matrix over 4 DPUs. The global next frontier of each iteration is the concatenation of the next frontiers of each DPU.

### 3.2.3 2D Partitioning

2D partitioning is where the adjacency matrix is partitioned into  $D$  equally sized tiles. Each partition is converted to the desired graph representation (COO/CSR/CSC) and sent to the MRAM of a DPU. The length of the next frontier is  $V/D_{vertical}$ , and the length of the current frontier  $V/D_{horizontal}$ , where  $D_{vertical}$  and  $D_{horizontal}$  are the number vertical and horizontal splits of the adjacency matrix respectively. The *aggregate()* function is a both the union and concatenation of the next frontiers of each DPU as shown in Figure 3.3.

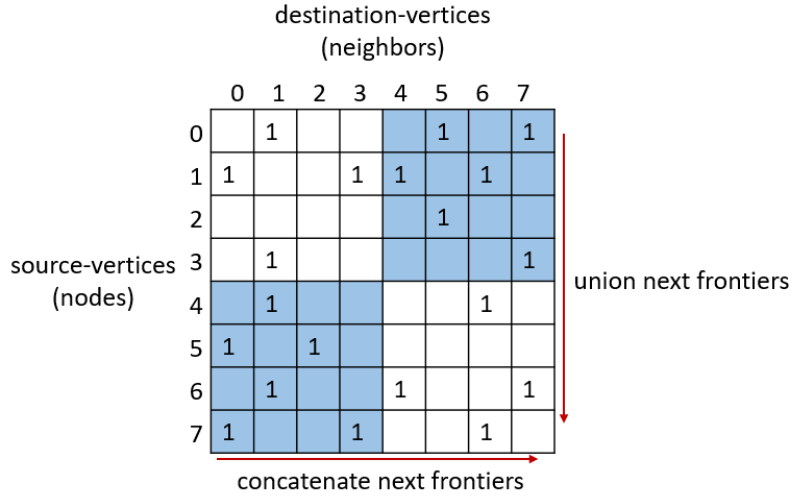


Figure 3.3: 2D-partitioning of an adjacency matrix over 4 DPUs. The global next frontier of each iteration is the concatenation of the next frontiers of each DPU horizontally, unioned vertically.

### 3.3 Parallelization Strategies

We discuss the strategies for parallelizing the BFS computation over the  $T$  tasklets of a DPU. Recall that each tasklet runs a logical thread of execution on top of a hardware thread. In our implementation, we use 11 tasklets per DPU.

#### 3.3.1 Vertex-centric Top-Down BFS

The partitions are required to be stored in MRAM using the CSR representation. We parallelize the computation by assigning a tasklet for each chunk of source-vertices (i.e. rows) as shown in Figure 3.4. The size of a chunk is the number of vertices per DPU divided by  $T$  tasklets.

For each source-vertex in the current frontier, we add all its non-visited neighbors (destination-vertices) to the next frontier. This approach requires a mutex because the next frontier chunk of the neighbor to add might be in use by another tasklet adding a neighbor to the same chunk.

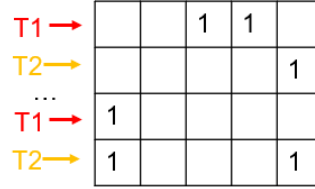


Figure 3.4: Top-Down BFS on a single DPU with 2 tasklets. The rows are interleaved among the tasklets.

---

**Algorithm 3** BFS-DPU Vertex-centric Top-Down

---

```

1: ...                                ▷ Initialization shown in Algorithm 2 (BFS-Init)
2: for chunk in current_frontier do
3:   for vertex in chunk do
4:     if vertex is set then
5:       for each neighbor of vertex do
6:         if neighbor not visited then
7:           mutex_lock(next_frontier_mutex)
8:           add_to_next_frontier(neighbor)
9:           mutex_unlock(next_frontier_mutex)
10:        end if
11:       end for
12:     end if
13:   end for
14: end for

```

---

### 3.3.2 Vertex-centric Bottom-Up BFS

The partitions are required to be stored in MRAM using the CSC representation. We parallelize the computation by assigning a tasklet for each chunk of destination-vertices (i.e. columns). The size of a chunk is the number of vertices per DPU divided by  $T$  tasklets.

For each non-visited source-vertex, if any of its neighbors (destination-vertices) are in the current frontier, we add the vertex to the next frontier. This approach does not require a mutex because the next frontier chunk of the vertex to add will only ever be access by the same tasklet.

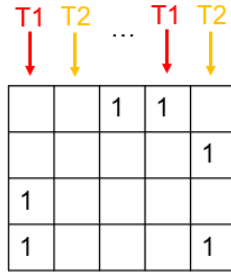


Figure 3.5: Bottom-Up BFS on a single DPU with 2 tasklets. The columns are interleaved among the tasklets.

---

#### Algorithm 4 BFS-DPU Vertex-centric Bottom-Up

---

```

1: ... ▷ Initialization shown in Algorithm 2 (BFS-Init)
2: for vertex in next_frontier do
3:   for vertex in chunk do
4:     if vertex is not visited then
5:       for each neighbor of vertex do
6:         if neighbor is in current_frontier then
7:           add_to_next_frontier(vertex)
8:           break
9:         end if
10:      end for
11:    end if
12:  end for
13: end for

```

---

### 3.3.3 Edge-centric BFS

The partitions are required to be stored in MRAM using the COO representation. We parallelize the computation by assigning a tasklet for each chunk of edges. The size of a chunk is the number of edges per DPU divided by  $T$  tasklets.

For every edge, if its source vertex is in the current frontier and its destination vertex is not visited, add the destination vertex to the next frontier. This approach requires a mutex because the next frontier chunk of the vertex to add might be in use by another tasklet adding a vertex to the same chunk.

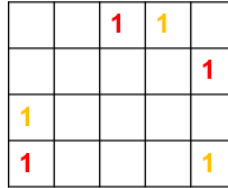


Figure 3.6: Edge-centric BFS on a single DPU with 2 tasklets. The edges are interleaved among the tasklets.

---

**Algorithm 5** BFS-DPU Edge-centric

---

```

1: ... ▷ Initialization shown in Algorithm 2 (BFS-Init)
2: for edge in edges do
3:   source_vertex = vertices[edge]
4:   if vertex is in current_frontier then
5:     destination_vertex = neighbors[edge]
6:     if destination_vertex is not visited then
7:       add_to_next_frontier(destination_vertex)
8:     end if
9:   end if
10: end for

```

---

### 3.4 The 9 BFS Algorithms

We will explore in this paper the 9 combinations of the BFS algorithm as shown in Figure 3.7.

	Row-partitioning	Column-partitioning	2D-partitioning
Top-Down BFS	<b>Top-Row</b>	<b>Top-Col</b>	<b>Top-2D</b>
Bottom-Up BFS	<b>Bot-Row</b>	<b>Bot-Col</b>	<b>Bot-2D</b>
Edge-centric BFS	<b>Edge-Row</b>	<b>Edge-Col</b>	<b>Edge-2D</b>

Figure 3.7: All 9 combinations of the graph partitioning strategies with the BFS parallelization strategies

We treat each as its own unique BFS algorithm. We may refer to these combinations as variations of BFS depending on the context.

# Chapter 4

## Methodology

In this section, we describe our benchmarking methodology.

### 4.1 Environment

We conduct our benchmarks on a machine with the following specs: equipped with a **Intel(R) Xeon(R) Silver 4110 CPU** (8 cores @ 2.1GHz), 64 GB of RAM, and **512 DPUs** running at 267MHz. The machine is running **Debian Linux 10** (kernel version 4.19.0) with **UPMEM SDK 2020.4.0** installed.

### 4.2 Programs

We create 5 programs in total:

- BFS-Host, CPU side of the DPU implementation.
- BFS-DPU, DPU side of the DPU implementation, with 3 variations:
  - BFS-DPU Vertex-centric top-down
  - BFS-DPU Vertex-centric bottom-up
  - BFS-DPU Edge-centric
- BFS-CPU implementation for comparison with the DPU implementation. Uses a sequential, vertex-centric top-down BFS algorithm.

The source code is available in our online repository [6].

Our CPU-targeting code is compiled using GCC version 9.3.0 (with optimization flag `-O3`), this includes BFS-Host and BFS-CPU. BFS-Host uses the Host Library, available in the UPMEM SDK, for communication with BFS-DPU.

Our DPU-target code is compiled using **dpu-upmem-dpurte-clang** (based on clang 10.0.0) (with optimization flag `-O2`), available through the UPMEM

SDK. We have configured all implementations to use **11 Tasklets** and **32 Bytes** DMA-transfer block size.

### 4.3 Measurements

We parse and preload the graph data into memory prior to the benchmarking. We do not measure the time it takes to fetch the results from memory after the BFS algorithm is finished.

We conduct our measurements directly inside our C code, using the *gettimeofday()* function to measure the code blocks of interest.

We measure the following:

- **DPU compute time:** total time spent by the DPUs processing the data. The DPUs are launched in parallel each BFS level; the time spent per level is the time spent by the last DPU that exits, as the others would be waiting for it to finish. Therefore, the DPU compute time is the sum of the time spent by the worst performing DPU each level.
- **Host overhead:** total time spent on the CPU-side of the BFS algorithm. It consists of the time spent transferring data to/from the DPUs plus the time it takes to aggregate the frontiers each BFS level.

For the CPU implementation, we measure the total time spent by the CPU processing the data.

### 4.4 Graphs

All graphs used are undirected, contain no duplicate edges and no edges to self. The edges are counted twice - once for each direction - since we are partitioning the full adjacency matrix. This is in order to ensure proper load balancing across the DPUs, otherwise some DPUs would treat the edges as directed. All graphs are stored as adjacency lists, parsed into COO, partitioned according to the partitioning strategy, and then converted into CSR or CSC depending on the parallelization strategy used.

For our strong scaling benchmarks graphs, we use data sets downloaded from the GraphChallenge[7] Data Sets. For a broad characterization of BFS, we chose a set of diverse graphs, as shown in Figure 4.1. In addition, we generate two very large synthetic graph: one with a high ratio of edges to nodes, and another with a low ratio of edges to nodes. These generated graphs are shown in Figure 4.2

For our weak scaling benchmarks, we generate for every DPU count (8, 16, 32, 64, 128, 256, 512) two graphs: a constant average degree graph, and a constant sparsity graph.

- The **constant average degree graphs** maintain the same ratio of edges to nodes (i.e.  $E/V$ ) at any DPU count: each time we double the number of DPUs, we grow the graph by multiplying the number of edges by 2 and the number of vertices by 2 .
- The **constant sparsity graphs** maintain the same ratio of edges to the maximum possible number of edges (i.e.  $E/V^2$ ) at any DPU count: each time we double the number of DPUs, we grow the graph by multiplying the number of edges by 2 and the number of vertices by  $\sqrt{2}$ .

These graphs are shown in Figures 4.3, 4.4.

All synthetic graphs are generated using PaRMAT[8], with parameters  $a = b = c = d = 0.25$  for a uniform distribution of edges to nodes, in order to properly test the load distribution among the DPUs.

#### 4.4.1 Graph Properties

In order to get a general characterization of each BFS algorithm, we run the benchmarks on a diverse set of graphs. The graphs differ by the following properties:

- Number of edges (E)
- Number of vertices (V)
- Average degree (average number of edges per vertex i.e.  $E/V$ )
- Maximum degree (maximum number of edges for a vertex)
- Sparsity (how close the graph is to the max number of edges i.e.  $E/V^2$ )
- Levels (i.e. layers, max-distance from root vertex)

#### 4.4.2 Graphs Used for the Strong Scaling Benchmarks

For the strong scaling benchmarks, we use the graphs shown in Figures 4.1 and 4.2.

<b>Graph</b>	<b>Vertices</b>	<b>Edges</b>	<b>Avg degree</b>	<b>Max degree</b>	<b>Levels</b>
soc-Epinions1	75879	811480	10.69	3044	8
soc-Slashdot0902	82168	1008460	12.27	2552	8
loc-gowalla	196591	1900654	9.66	14730	9
loc-brightkite	58228	428156	7.35	1134	10
roadNet-CA	1965206	5533214	2.81	12	554
roadNet-TX	1379917	3843320	2.78	12	722
amazon0601	403394	4886816	12.11	2752	17
cit-HepTh	27770	704570	25.37	2468	10

Figure 4.1: Properties of graphs for the strong scaling benchmarks

<b>Graph</b>	<b>Vertices</b>	<b>Edges</b>	<b>Avg degree</b>	<b>Max degree</b>	<b>Levels</b>
gen_high_ratio	3200000	38400000	12	42	9
gen_low_ratio	6000000	16800000	2.8	19	24

Figure 4.2: Properties of synthetic graphs for the strong scaling benchmarks

#### 4.4.3 Graphs Used for the Weak Scaling Benchmarks

For the weak scaling benchmarks, we use the the synthetic graphs shown in Figures 4.3 and 4.4.

<b>Num DPUs</b>	<b>Vertices</b>	<b>Edges</b>	<b>Avg Degree</b>	<b>Max Degree</b>	<b>Levels</b>
8	800000	1200000	1.5	13	3
16	1600000	2400000	1.5	12	6
32	3200000	4800000	1.5	13	54
64	6400000	9600000	1.5	13	55
128	12800000	19200000	1.5	15	57
256	25600000	38400000	1.5	14	57
512	51200000	76800000	1.5	16	58

Figure 4.3: Properties of synthetic graphs with constant avg. degree for weak scaling benchmarks. Each time we double the number of DPUs, we double the number of edges and vertices.

<b>Num DPUs</b>	<b>Vertices</b>	<b>Edges</b>	<b>Avg Degree</b>	<b>Max Degree</b>	<b>Levels</b>
8	800000	1200000	1.50	11	8
16	1131370	2400000	12.00	13	19
32	1600000	4800000	3.00	18	21
64	2262741	9600000	4.24	18	16
128	3200000	19200000	6.00	26	13
256	4525483	38400000	8.48	27	11
512	6400000	76800000	12.00	41	9

Figure 4.4: Properties of synthetic graphs with constant sparsity for weak scaling benchmarks. Each time we double the number of DPUs, we double the number of edges and we multiply the vertices by  $\sqrt{2}$ .

# Chapter 5

## Evaluation

In this section, we present our benchmark data for analysis and evaluation. The benchmarks measure the performance of each variation of the BFS algorithm on the Data-Processing Units. We look into the characterization of each algorithm; the strong scaling and weak scaling of the algorithms with respect to the number of DPUs used. The benchmarks are conducted over a diverse set of graphs.

### 5.1 Strong Scaling Benchmarks

Strong scaling is defined as how performance scales as we increase the number of processors (here DPUs) for the same graph. For the strong scaling benchmarks, we use the graphs shown in Figures 4.1 and 4.2.

We publish the results of our strong scaling benchmarks on the next pages. The total BFS time includes the DPU computation time plus the Host communication time overhead.

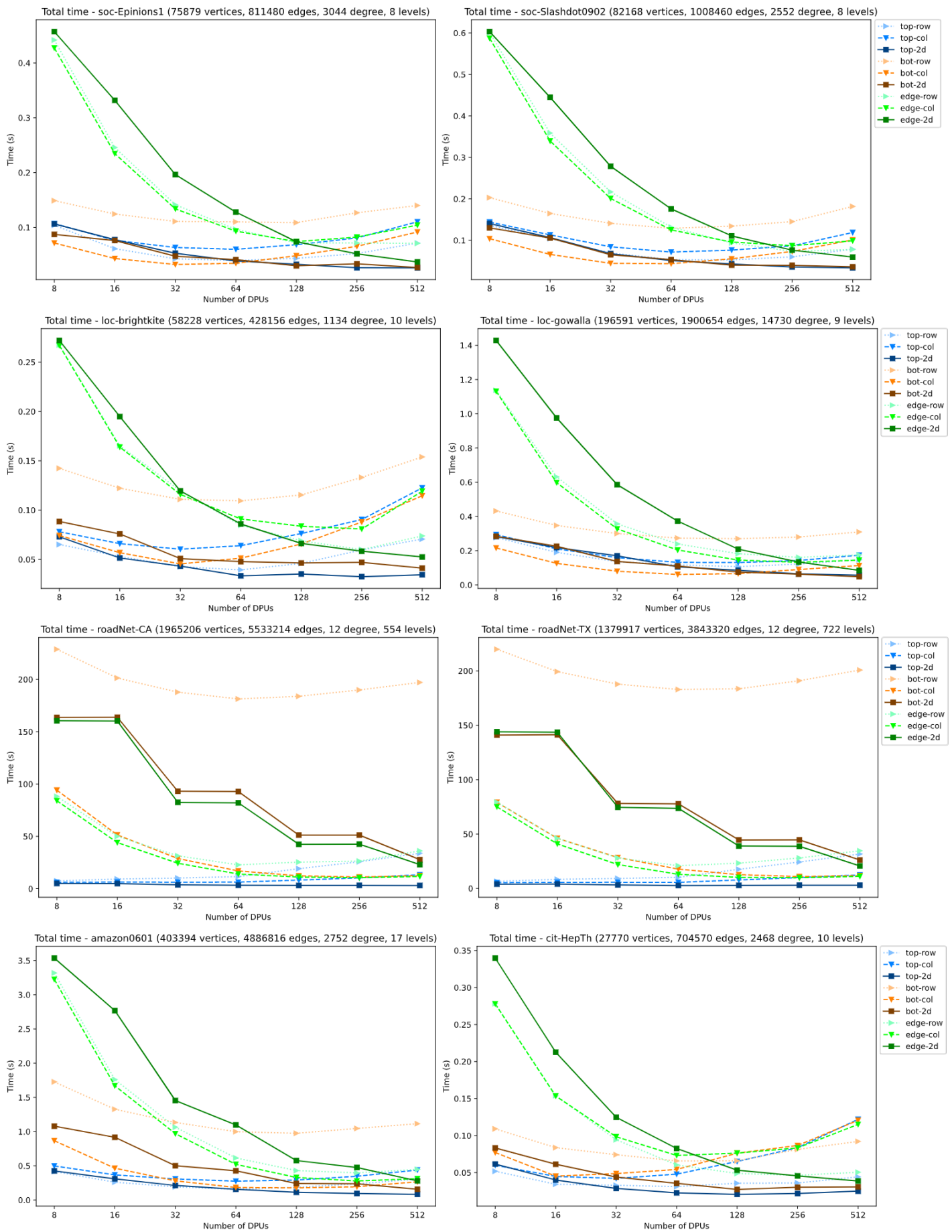


Figure 5.1: BFS Total Time - strong scaling benchmarks

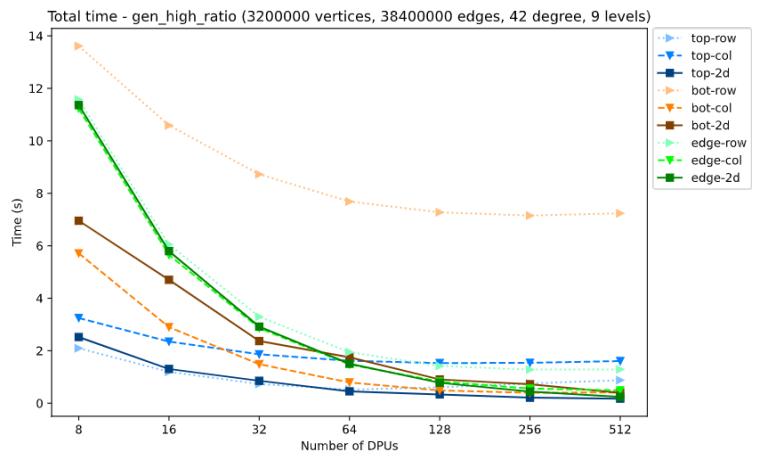
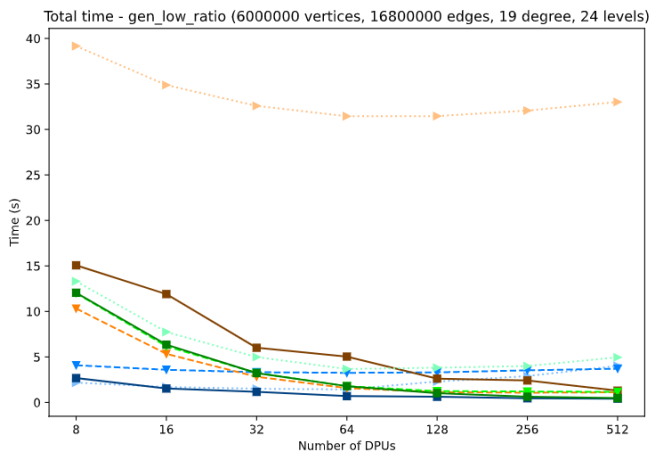


Figure 5.2: BFS Total Time - strong scaling benchmarks over synthetic graphs

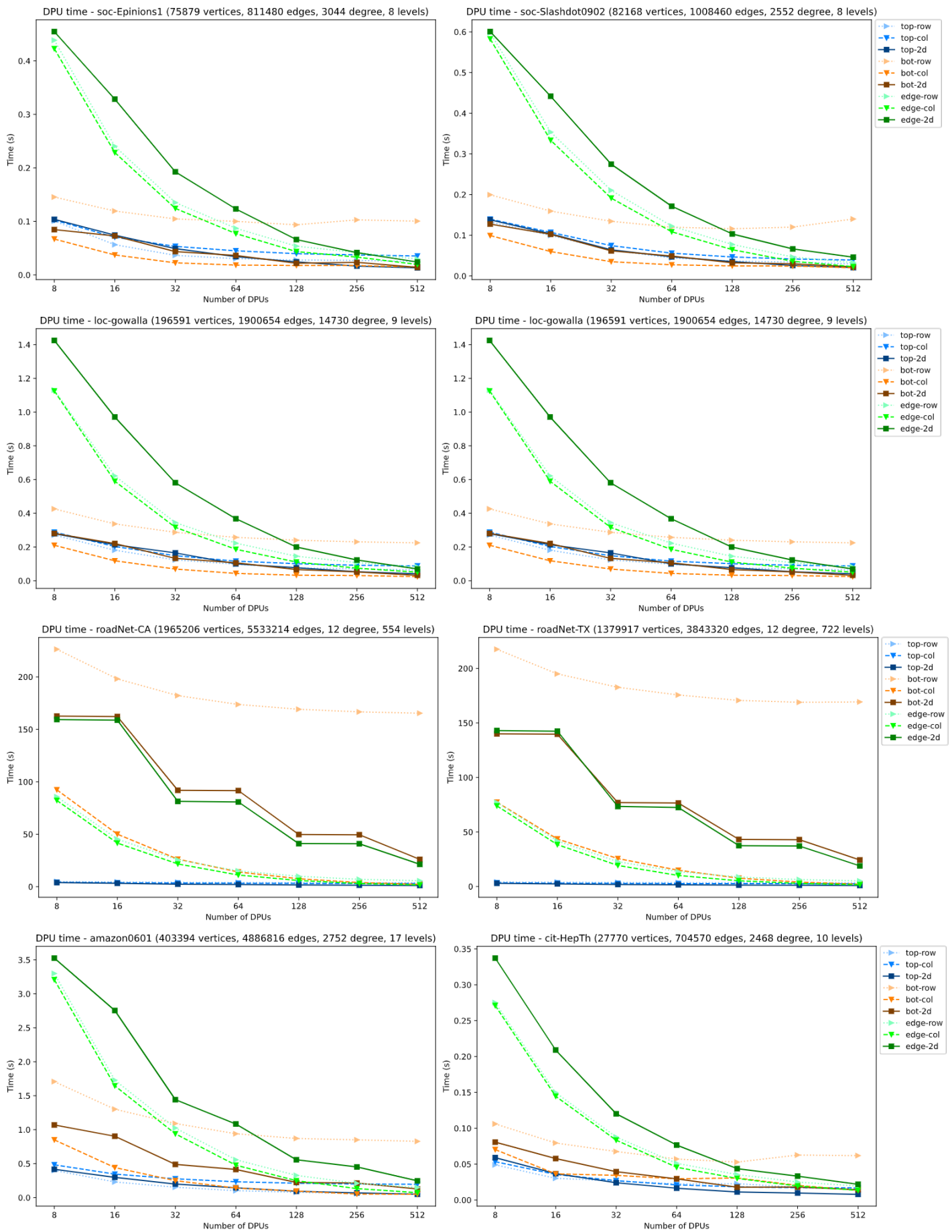


Figure 5.3: BFS DPU Time - strong scaling benchmarks

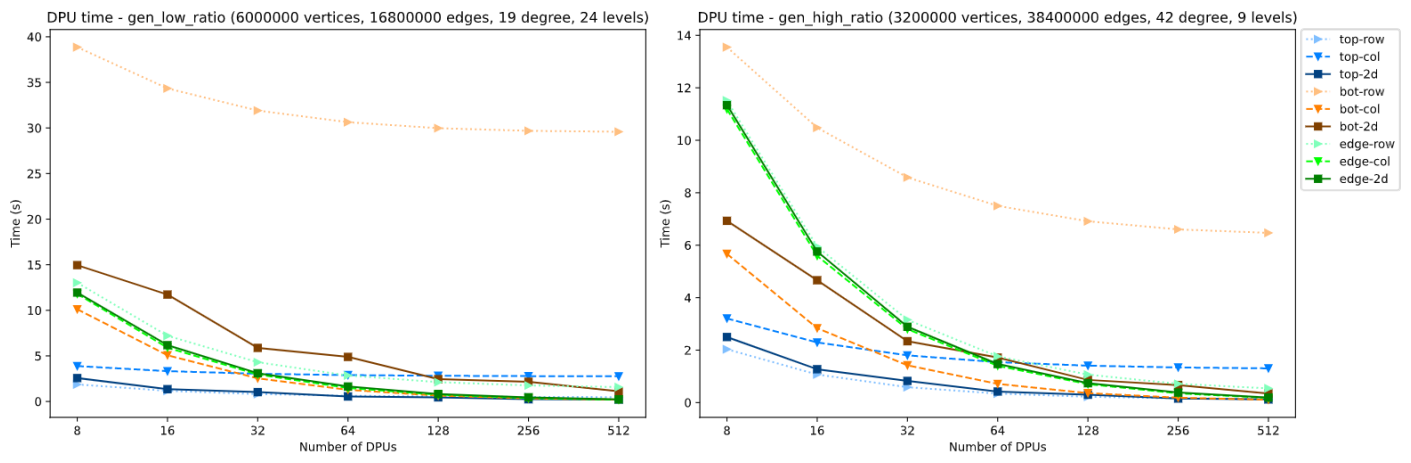


Figure 5.4: BFS DPU Time - strong scaling benchmarks over synthetic graphs

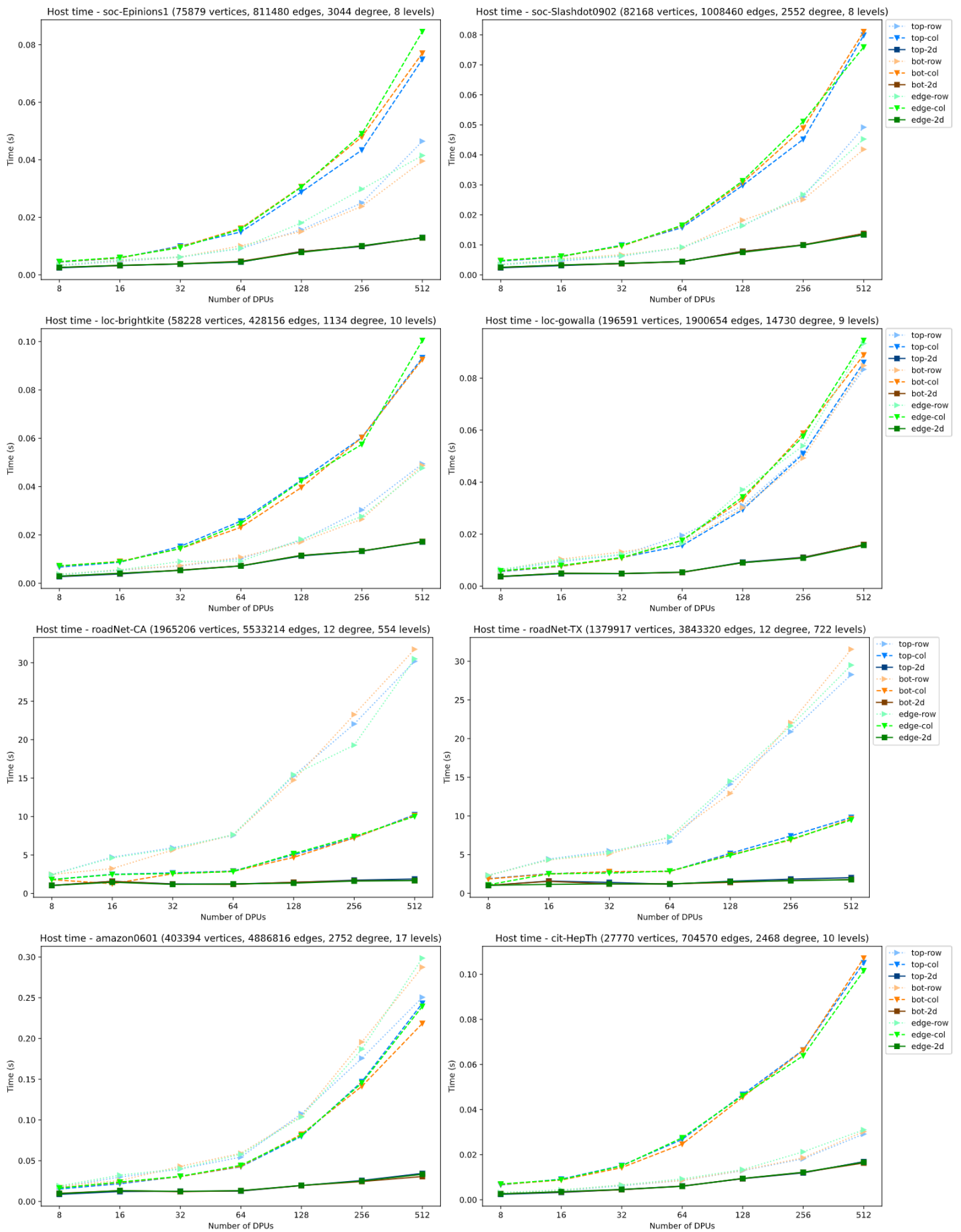


Figure 5.5: BFS Host Time - strong scaling benchmarks overview

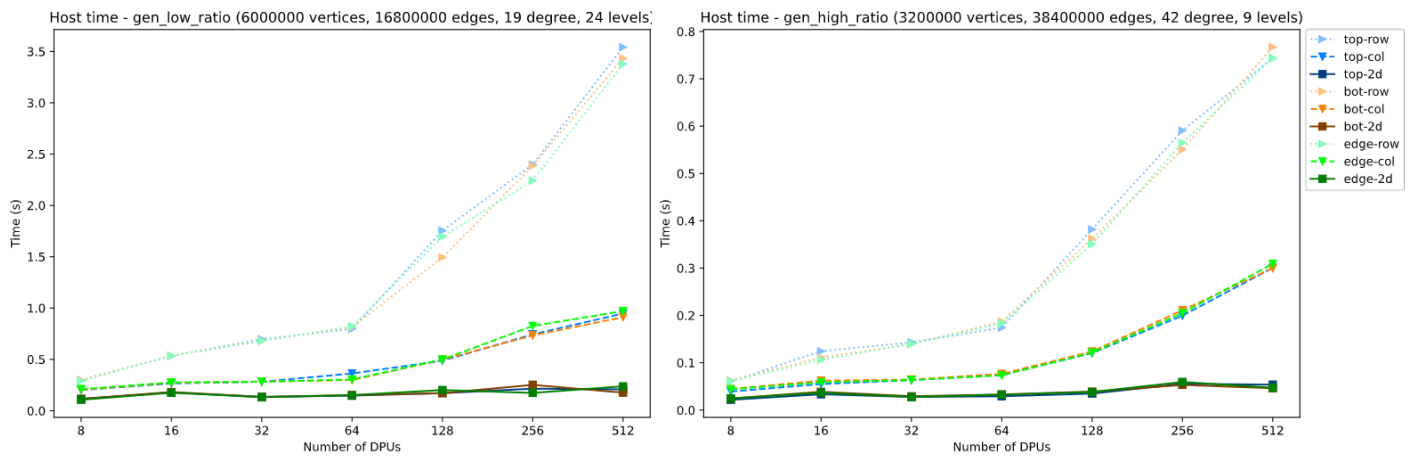


Figure 5.6: BFS Host Time - strong scaling benchmarks over synthetic graphs

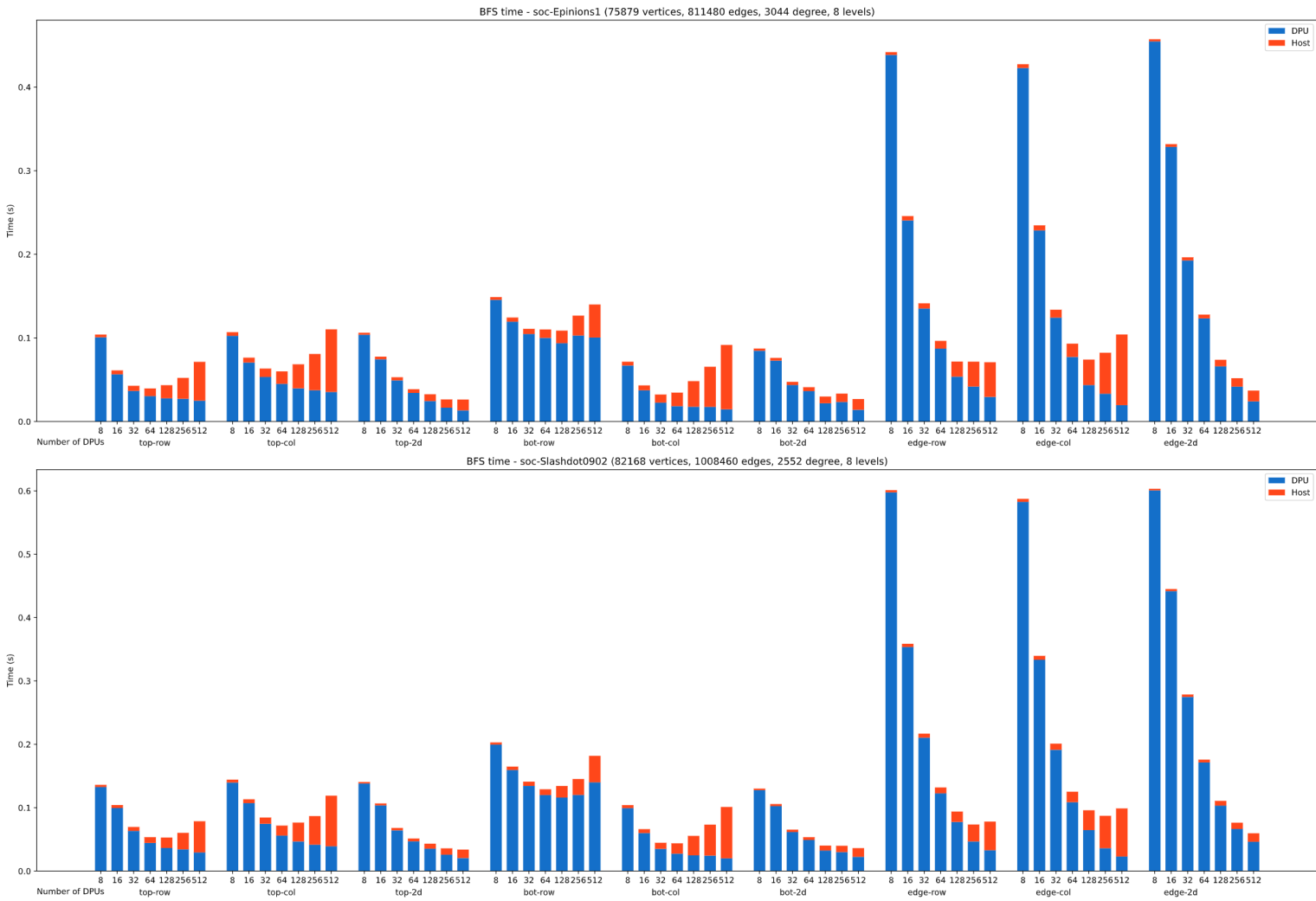
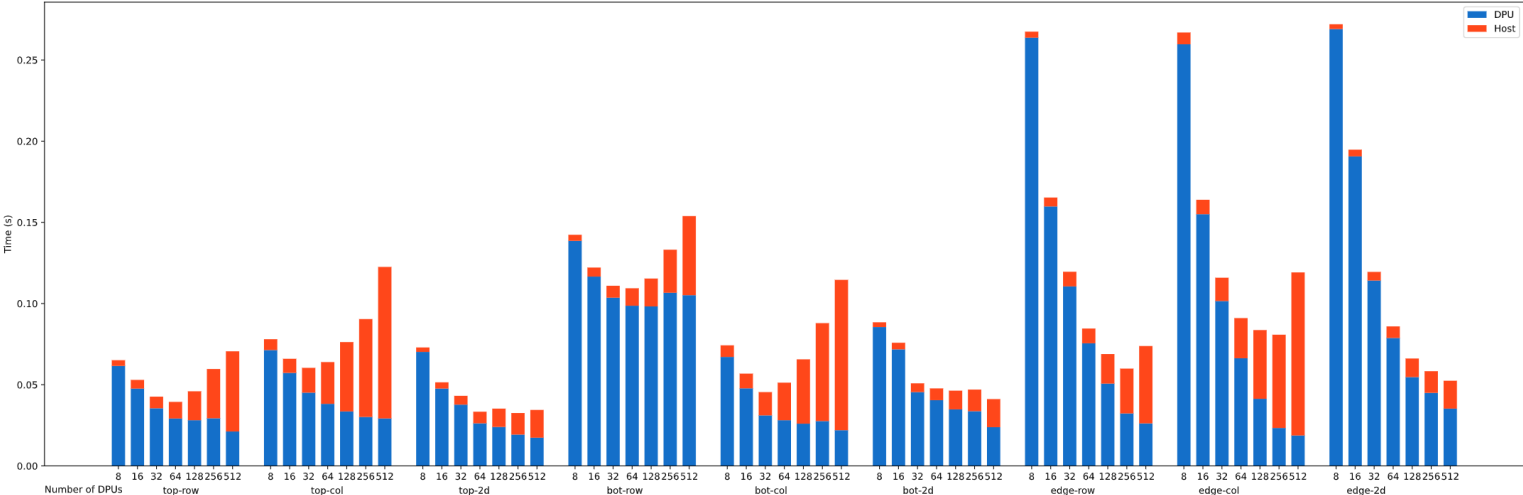


Figure 5.7: BFS strong scaling benchmarks over social media graphs

BFS time - loc-brightkite (58228 vertices, 428156 edges, 1134 degree, 10 levels)



BFS time - loc-gowalla (196591 vertices, 1900654 edges, 14730 degree, 9 levels)

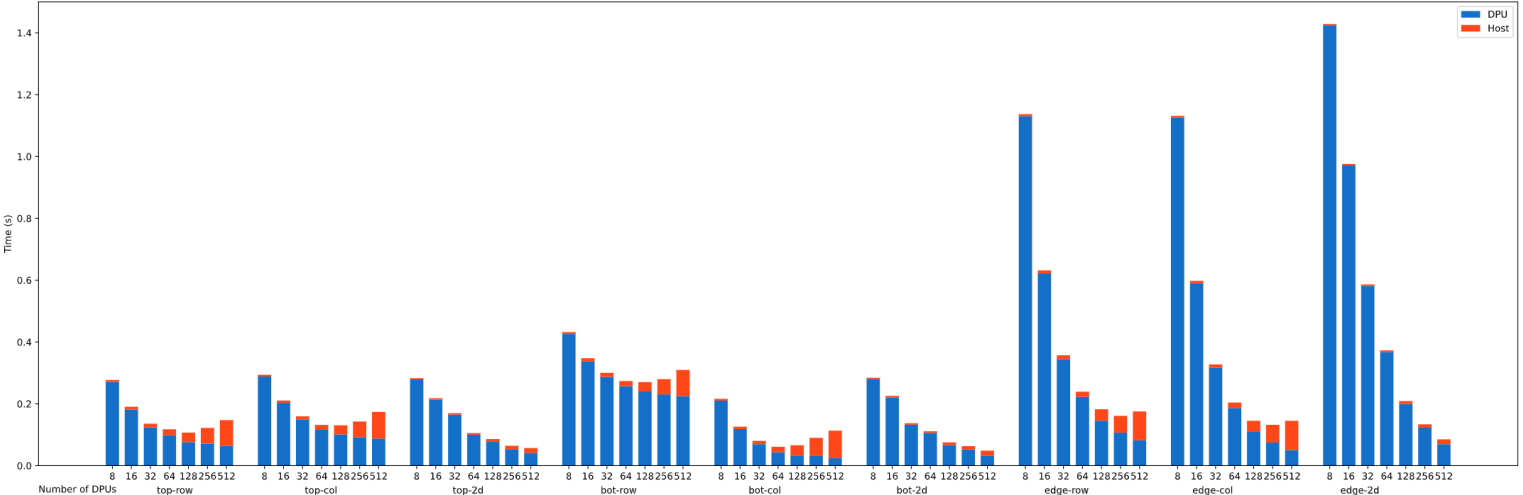


Figure 5.8: BFS strong scaling benchmarks over geo-location graphs

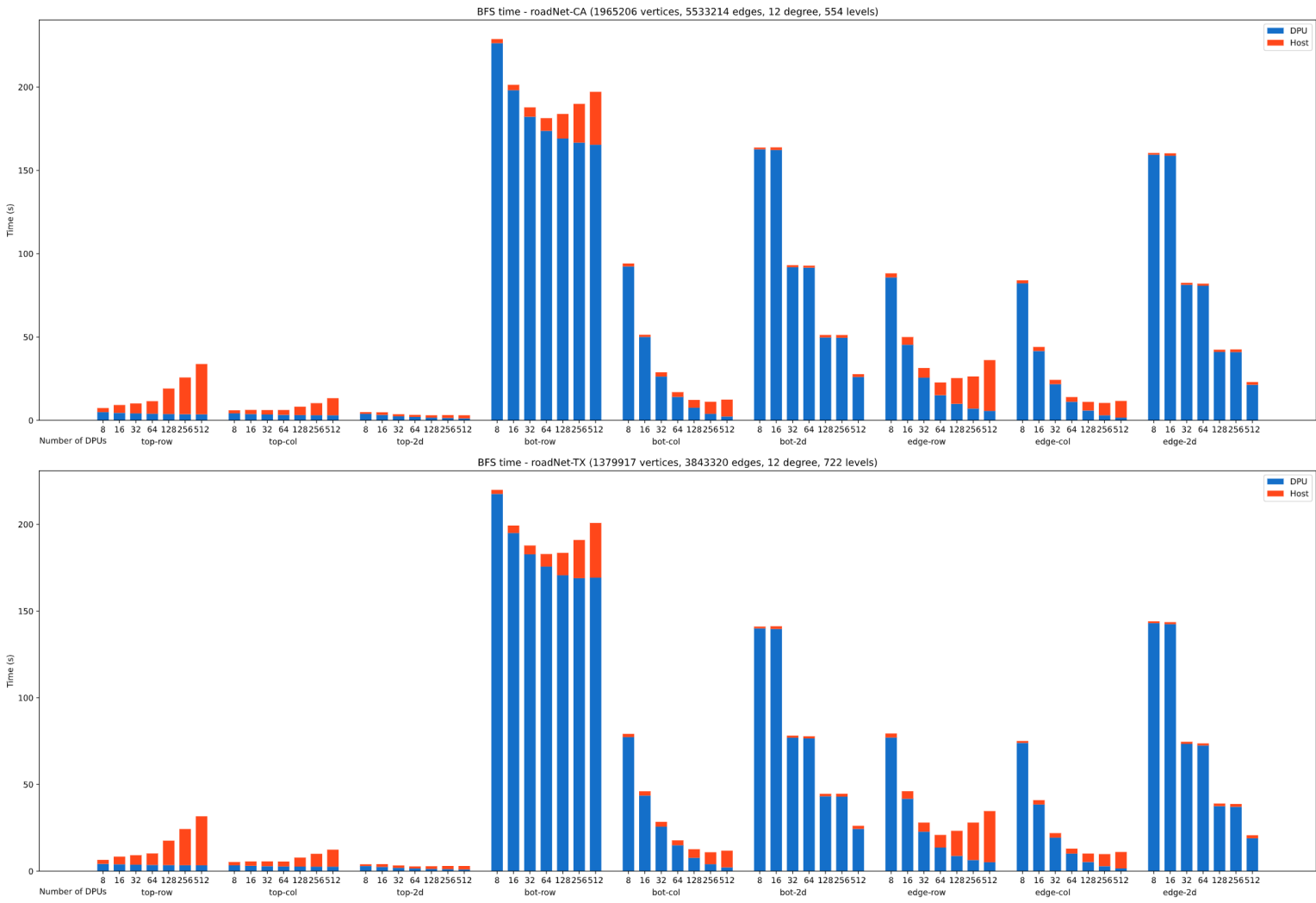


Figure 5.9: BFS strong scaling benchmarks over roadnet graphs

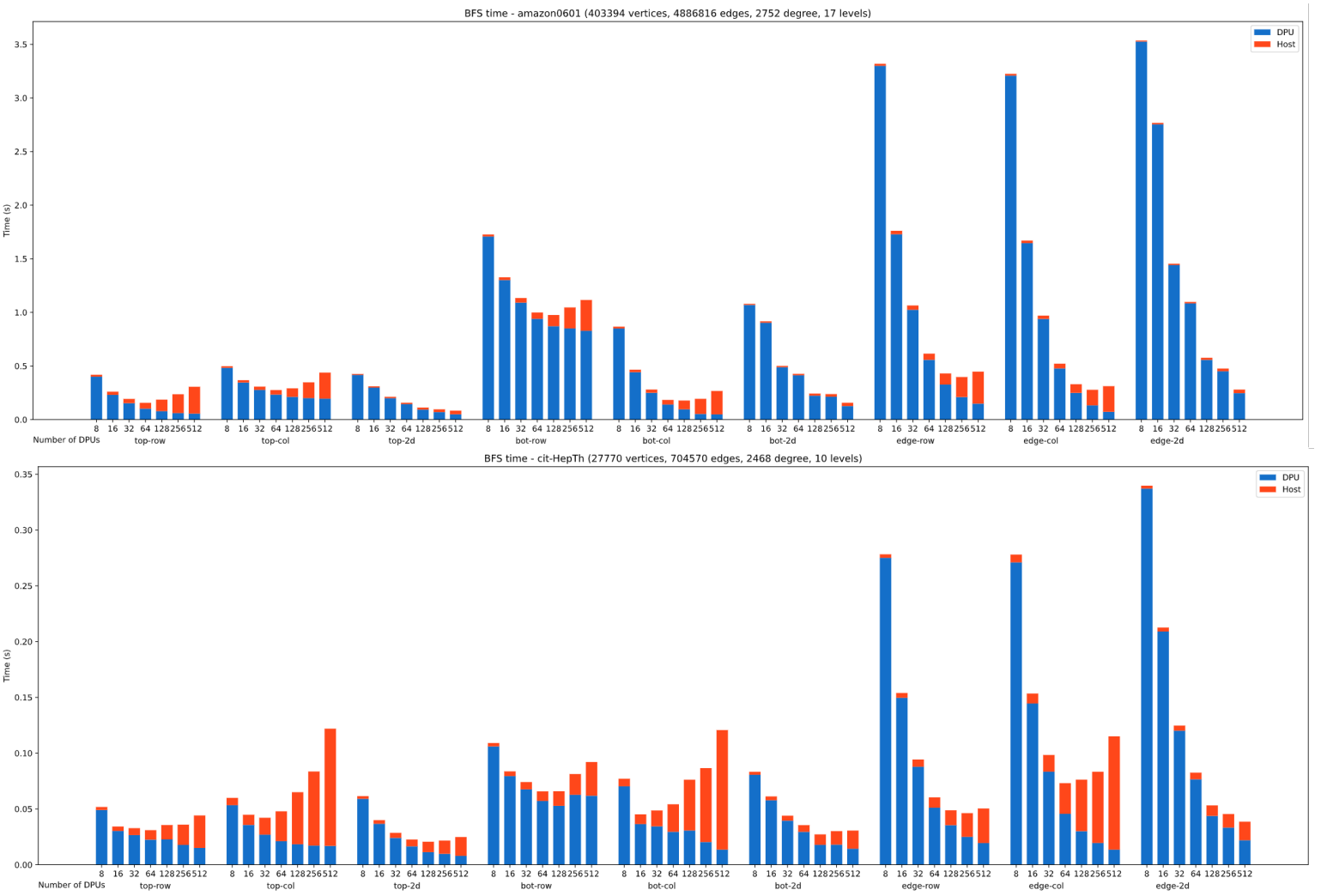


Figure 5.10: BFS strong scaling benchmarks over miscellaneous graphs

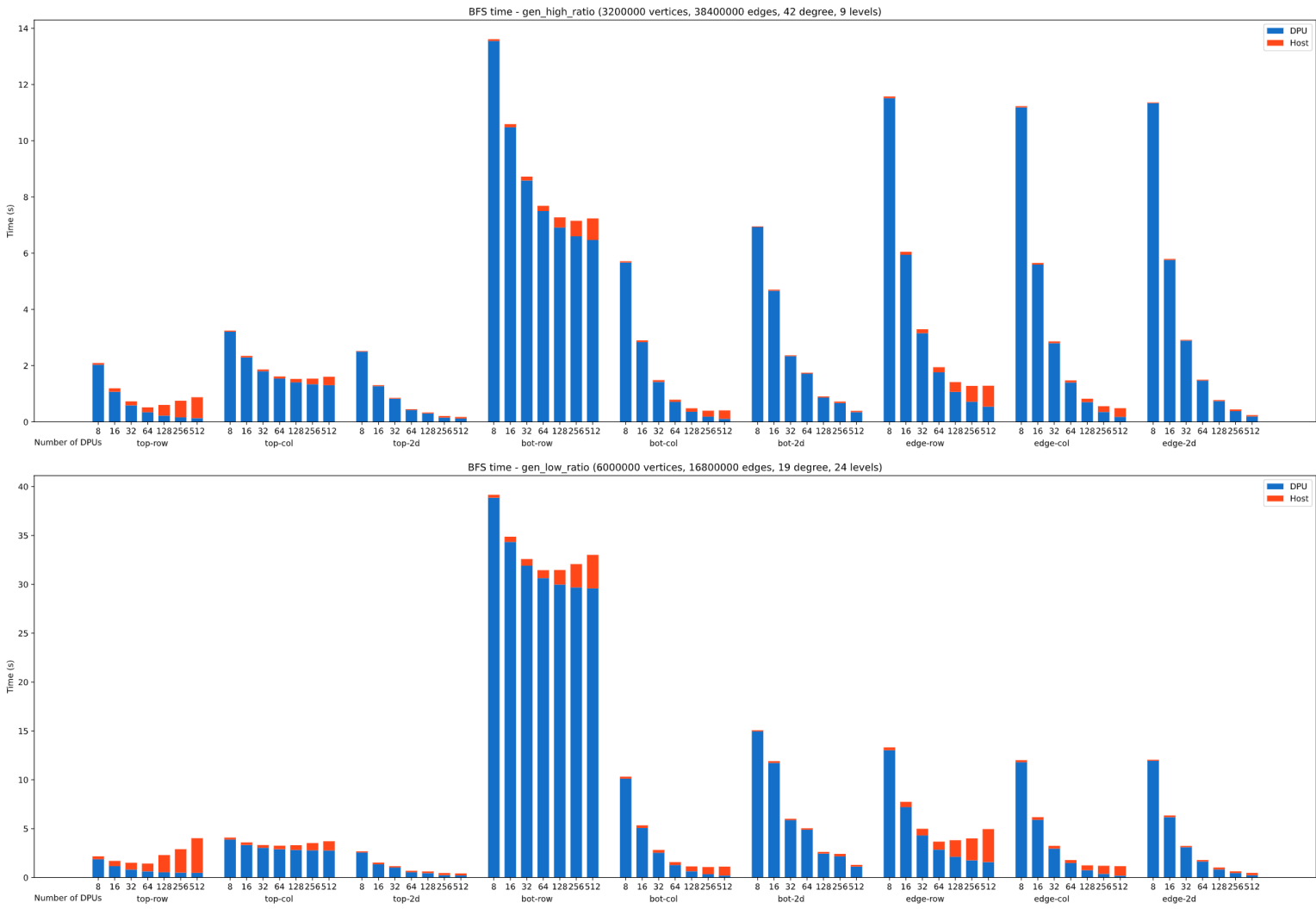


Figure 5.11: BFS strong scaling benchmarks over synthetic graphs

## 5.2 Strong Scaling Analysis

In this section we characterize and analyze each BFS variation across the various benchmarks.

### 5.2.1 Host Time Overhead

The Host time - time spent in CPU host code - is synchronization overhead. The code is shown in Algorithm 1 (BFS-Host) in section 3.1. We inspect the benchmarks in Figures 5.5, 5.6.

We notice that all algorithms with the same partitioning scheme (Row/Col/2D) have almost exactly the same host time and host strong scaling profile. This suggests that the host overhead is completely independent of the BFS algorithm used (Top-Down/Bottom-Up/Edge); instead it is completely dependent on the partitioning scheme. From the Host's perspective, synchronization is agnostic to the algorithm running on the DPUs.

For both Row and Column partitioning, the host overhead increases rapidly as the number of DPUs increases. Row-partitioning is slightly faster than Column-partitioning, except for in the roadnet and synthetic graphs. Meanwhile, 2D partitioning vastly outperforms both, increasing slowly with the number of DPUs.

**Explanation:** For each level in the BFS, the host copies and aggregates (union and/or concatenation) the next frontiers from each DPU into a single next frontier, then re-broadcasts it to the DPUs. As we increase the number of DPUs, the host transfers more data per level. Combined with limited memory throughput, this incurs a large data movement cost. Therefore, it is expected that the host time increases with the number of DPUs.

Depending on the partition scheme, the lengths of the frontiers being copied from/to the DPUs can vary. In Row-partitioning, we copy from each DPU a next frontier with the length of total number of columns ( $V$ ). In Column-partitioning, we copy from each DPU a next frontier with the length of partitioned columns ( $V/Num\_DPUs$ ). This means that Column-partitioning copies much less data from DPUs per level than Row-partitioning. For very large graphs, this performance difference is huge, which explains why it outperforms Row-partitioning in roadnets and synthetic graphs (both of which are very large). In our implementation, concatenation is a bit slower than union; for this reason Row-partitioning is a generally faster in smaller graphs.

The question of why 2D partitioning is so much better remains. Similar to Column-partitioning, the next frontiers being copied from the DPUs is small ( $V/Num\_Horizontal\_DPUs$ ). However, unlike in Row and Column partitioning, we only need to broadcast back to each DPU a slice of the next frontier and a slice

of the current frontier ( $V/Num\_Horizontal\_DPUs + V/Num\_Vertical\_DPUs$ ) rather than a matrix-wide next frontier ( $V$ ). This results in much less data being transferred.

**In summary:**

- Host time is only dependent on the partitioning strategy.
- Host time scales negatively; the overhead increases with the number of DPUs.
- 2D-partitioning is by far the best performing partitioning scheme.
- Column-partitioning scales better than Row-partitioning in very large graphs (like roadnets), while the latter performs slightly better in smaller graphs.

### 5.2.2 BFS DPU Time

The DPU time is the total time spent executing BFS code in the DPUs (as a whole, not individually). A DPU runs BFS-Init (Algorithm 2 in section 3.1) followed by the BFS-DPU algorithm (Algorithms 3, 4, or 5 in section 3.2) each level. We inspect the benchmarks in Figures 5.3, 5.4.

We notice that the DPU time is dependent on both the parallelization strategy and the partitioning strategy. Overall, the DPU time of all BFS variations scale positively with the number of DPUs. However, for all variations, we observe a point of diminishing returns (typically 256 DPUs), after which we see very little gain in performance by adding more DPUs. However, the performance gap between the variations shrink considerably at high DPU count.

**Top-Down:** All 3 Top-Down BFS variations have very similar performance and scaling profiles. Top-Down-2D performs slightly better than Top-Down-Col, while Top-Down-Row is noticeably worse than both (most notably in the synthetic graphs). In fact, Top-Down-2D is consistently one of best performers out of all variations in DPU time.

**Bottom-Up:** We notice that Bottom-Up-Row is consistently the worst performer out of all BFS variations, especially in large graphs such as roadnets. Bottom-Up-Col is one of the best performers in DPU time out of all variations, faring better than Top-Down-2D in social media and geo-location graphs, but worse in roadnet and synthetic graphs. For DPU count above 128, the performance difference tends to be negligible. Bottom-Up-2D, while performing slightly worse than Bottom-Up-Col, performs similarly at high DPU count. However, we notice that in the roadnet graphs it performs much worse.

**Edge:** All Edge-centric variations have roughly the same performance and scaling profile, with Edge-2D being slightly behind. The Edge variations perform worse than all other variations (except Bottom-Up-Row). However, they scale extremely well with the number of DPUs, to the point that the performance gap becomes very small at very high DPU counts. Except for Edge-2D in roadnet graphs, where it performs much worse (similar to Bottom-Up-2D).

**Explanation:** In Top-Down, each DPU is checking its assigned rows (source-vertices) for neighbors (destination-vertices). This means that Row-partitioning should work well for Top-Down, as each DPU only iterates over a partition of rows. However, since each DPU has all the columns of the adjacency matrix, the neighbors in each DPU can be any vertex in the graph. Therefore, when a DPU executes the BFS-Init code at the beginning of each iteration/level in order to update the current frontier and the visited-vertices bitarrays, it has to iterate over all columns, which adds to the DPU compute time. For this reason, Top-Down-Row performs worse than Top-Down-Col, which performs worse than Top-Down-2D because it has to iterate over all the rows in the matrix. Roadnet graphs are quite sparse; therefore the bulk of the DPU compute time is spent in BFS-Init rather than in Top-Down BFS. Since BFS-Init is independent of the graph sparsity, the DPU time in roadnets is relatively constant.

Bottom-Up BFS performs best when the max degree of a graph is high. For this reason, we see this variation perform better than Top-Down in social media and geo-location graphs, yet worse in roadnet and synthetic graphs. Bottom-Up-Row performs badly because each DPU needs to iterate over all the columns for every source-vertex in the frontier, for both BFS-Init and the BFS algorithm (as opposed to just BFS-Init in Top-Down-Row).

Edge-centric BFS are the worst performing variations because, by iterating over the edges, we are loading the edges from MRAM in large blocks, while the “visited”, “current frontier”, and “next frontier” bitarrays are randomly accessed. On the other hand, the vertex-centric approach iterates over the bitarrays by definition (as they either the length of the columns or rows), and would therefore load most data in sequential blocks.

**In summary:**

- Top-Down-2D and Bottom-Up-Col have the best DPU time performance, with the latter being better for graphs with high max degree.
- Bottom-Up-Row is the worst performing variation.
- Edge-centric BFS perform typically worse than Vertex-centric BFS, due to inducing more random memory access.

- At very high DPU count, most variations perform similarly.

### 5.2.3 BFS Total Time

In this section, we compare and contrast the overall performance and strong scaling profile of all BFS variations. We draw our conclusion from the analysis of the Host Overhead and the DPU Time, and from the Figures 5.1, 5.2, 5.7, 5.8, 5.9, 5.10, 5.11.

#### Key Observations:

- For both social media and geo-location graphs, Bottom-Up-2D at 512s DPUs performs the best. Top-Down-2D performs almost as good.
- For roadnet graphs, Top-Down-2D at 512 DPUs perform much better than anything else, however we see a point of diminishing returns after 64 DPUs, as the increasing Host Overhead catches up to the decreasing DPU time.
- For the amazon0601 graph, we see Top-Down-2D at 512 DPUs performing the best.
- For cit-HepTh, a relatively small graph, Top-Down-2D at 128 DPUs performs the best, as the Host overhead outgrows the DPU time.
- For the synthetic graphs, we once again see Top-Down-2D at 512 DPUs outperform all other variations, however Edge-2D comes in a very close second place.

From our analysis and these key observations, we draw our final conclusion for the strong scaling performance of the BFS algorithms:

#### Conclusion:

- 2D-partitioning is, without a doubt, the best graph partitioning strategy to reduce host overhead.
- Top-Down-2D is overall the best performing BFS algorithm, while Bottom-Up-2D performs better in high degree graphs.
- **Strong scaling profile:** For a fixed sized graph, BFS scales with the number of DPUs up to a point of diminishing returns. After this point, the host overhead starts to overtake the reduction in DPU time and could lead to negative scaling. **Therefore, depending on the graph properties, there exists an optimal number of DPUs to use that may not be the maximum available.**

## 5.3 Weak Scaling Benchmarks

Weak scaling is defined as how the performance scales as we increase the number of DPUs and the size of the graph proportionally. For the weak scaling benchmarks, we use the synthetic graphs shown in Figures 4.3 and 4.4.

We publish the results of our weak scaling benchmarks on the next pages. The total BFS time includes the DPU computation time plus the Host communication time overhead.

Note:

1. Some benchmarks failed to run for certain algorithms due to having a huge

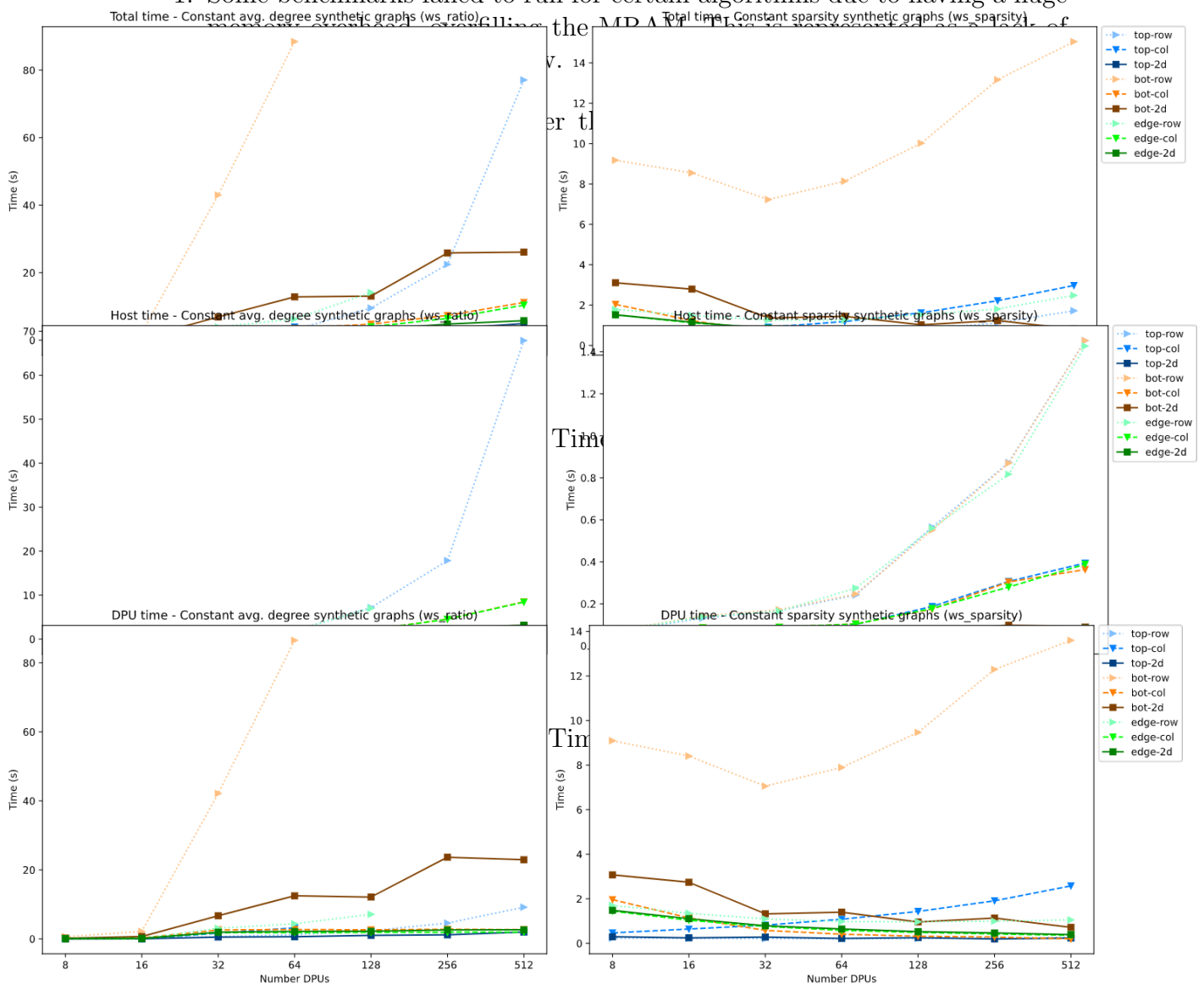


Figure 5.14: BFS DPU Time - weak scaling benchmarks.

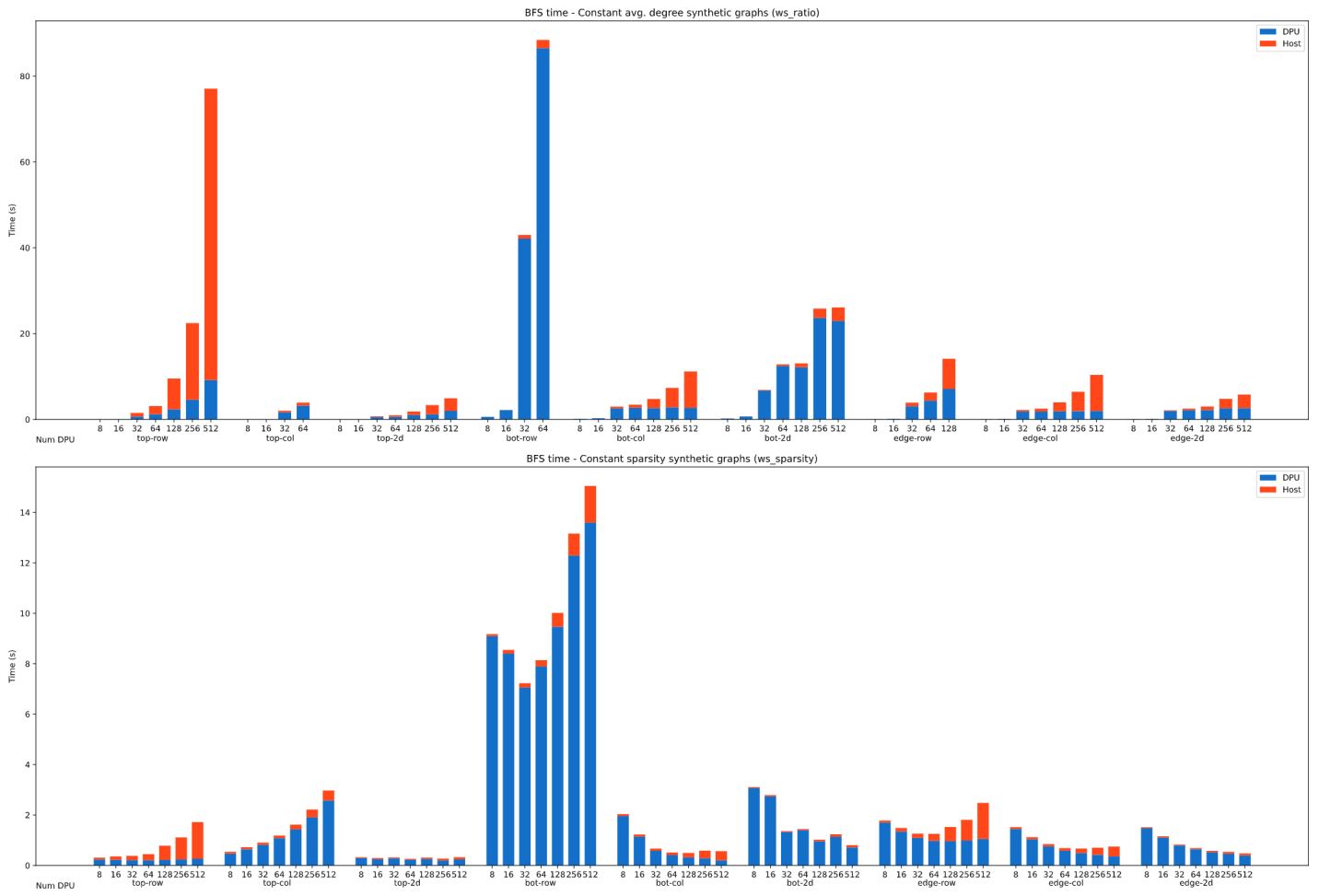


Figure 5.15: BFS Total Time - weak scaling benchmarks breakdown.

## 5.4 Weak Scaling Analysis

We inspect and analyze Figures 5.12, 5.13, 5.14, 5.15.

### 5.4.1 Constant Average Degree Graphs

In these graphs, we maintain the average number of edges per node as we increase the DPU count (i.e.  $E/V$ ).

1. We notice that none of the algorithms scale positively.
2. Using CSR representation on a column-partitioned matrix is incredibly memory-storage inefficient. If the graph is too big, it may not fit in the DPU's 64MB memory, which what happens in **Top-Down-Col** (MRAM overflow).
3. The same thing occurs in **Bottom-Up-Row**, where using CSC representation on a row-partitioned matrix is incredibly memory-storage inefficient.
4. **Top-Down-Row** scales very negatively, both in DPU Time and in Host Time. As we are adding more vertices to the graph, the synchronization cost becomes too high in 1D-partitioning. This is also apparent in **Bottom-Up-Col**.
5. Top-Down-2D is much better than all other variations, but still scales negatively.

In conclusion, doubling the number of DPUs is not enough to match the computation needed for BFS on graphs of double the size.

### 5.4.2 Constant Sparsity Graphs

In these graphs, we maintain the sparsity of edges in the graph as we increase the DPU count. We remind that the sparsity of a graph is ratio of number of edges with respect to the maximum possible number of edges (i.e.  $E/V^2$ ). Here, we see much more useful results than in the previous graph.

**Host Time Overhead:** Row-partitioning, is the worst scaling partitioning strategy. Column-partitioning scales significantly better, but the overhead still grows fast with the graph size. Meanwhile 2D-partitioning sees excellent weak scaling.

**Explanation:** As we are doubling the number of vertices, the size of the “next frontier“ doubles. This especially affects Row-partitioning, which copies the full sized frontier twice (once from and once to) each DPU every level. Column-partitioning is less affected, as it only needs to copy from the DPUs a partitioned

“next frontier”, but still needs to broadcast a full sized frontier to every DPU. 2D-partitioning only needs to copy and broadcast a slice of the frontier to each DPU, which is why it scales best.

**DPU Time:** We see a similar pattern as in the strong scaling benchmarks, with Top-Down-2D exhibiting very good weak scaling. However, others have good DPU-Time weak scaling as well: Top-Down-Row, Bottom-Up-Col, Bottom-Up-2D, Edge-Col, and Edge-2D. The remaining do not scale. These are for the same reasons as outlined in the strong scaling benchmarks.

**Conclusion:**

- 2D-partitioning is the best scaling partitioning strategy. Column-partitioning can be a problem at extremely large graph sizes while Row-partitioning is completely impractical.
- Top-Down-2D, Bottom-Up-Col, Bottom-Up-2D and Edge-2D perform the best in that order.
- These algorithms show good weak scaling when we double the number of DPUs each time we double the number of edges, while maintaining sparsity.

## 5.5 CPU BFS Comparison

In this section we compare the performance of our DPU BFS to a traditional CPU implementation. We are using a **sequential** CPU version as a baseline performance indicator. Typically, one could use multi-threading or GPUs to accelerate BFS, however we leave that for future work.

Graph	CPU Alg.	CPU Time (s)	DPU Config.	DPU Time (s)
amazon0601	Top-Down	0.000352	top-down-2d-512	0.083584
cit-HepTh	Top-Down	2.3e-05	top-down-2d-128	0.020614
loc-brightkite_edges	Top-Down	4.9e-05	top-down-2d-256	0.032551
loc-gowalla_edges	Top-Down	0.00017	bot-2d-512	0.0485
soc-Epinions1	Top-Down	6.4e-05	top-down-2d-512	0.026274
soc-Slashdot0902	Top-Down	6.9e-05	top-down-2d-512	0.033823
roadNet-CA	Top-Down	0.001705	top-down-2d-512	3.043304
roadNet-TX	Top-Down	0.0012	top-down-2d-64	2.73583
gen_low_ratio	Top-Down	0.662935	top-down-2d-512	0.421283
gen_high_ratio	Top-Down	0.363123	top-down-2d-512	0.174864

Figure 5.16: Best BFS algorithm on CPU and DPU for the strong scaling graphs

Graph (Num DPUs)	CPU Alg.	CPU Time (s)	DPU Alg	DPU Time (s)
gen_ratio (8)	Top-Down	0.001374	top-down-2d	0.003089
gen_ratio (16)	Top-Down	0.005432	top-down-2d	0.011102
gen_ratio (32)	Top-Down	0.398098	top-down-2d	0.720172
gen_ratio (64)	Top-Down	0.872666	top-down-2d	0.963179
gen_ratio (128)	Top-Down	1.840153	top-down-2d	1.842433
gen_ratio (256)	Top-Down	3.793615	top-down-2d	3.34165
gen_ratio (512)	Top-Down	7.648169	top-down-2d	4.920437

Figure 5.17: Best BFS algorithm on CPU and DPU for high avg. degree graphs

Graph (Num DPUs)	CPU Alg.	CPU Time (s)	DPU Alg	DPU Time (s)
gen_sparsity (8)	Top-Down	0.001396	top-down-2d	0.326541
gen_sparsity (16)	Top-Down	0.097232	top-down-2d	0.295448
gen_sparsity (32)	Top-Down	0.135142	top-down-2d	0.317287
gen_sparsity (64)	Top-Down	0.191678	top-down-2d	0.26074
gen_sparsity (128)	Top-Down	0.320034	top-down-2d	0.310222
gen_sparsity (256)	Top-Down	0.549793	top-down-2d	0.270553
gen_sparsity (512)	Top-Down	1.020419	top-down-2d	0.32477

Figure 5.18: Best BFS algorithm on CPU and DPU for low avg. degree graphs

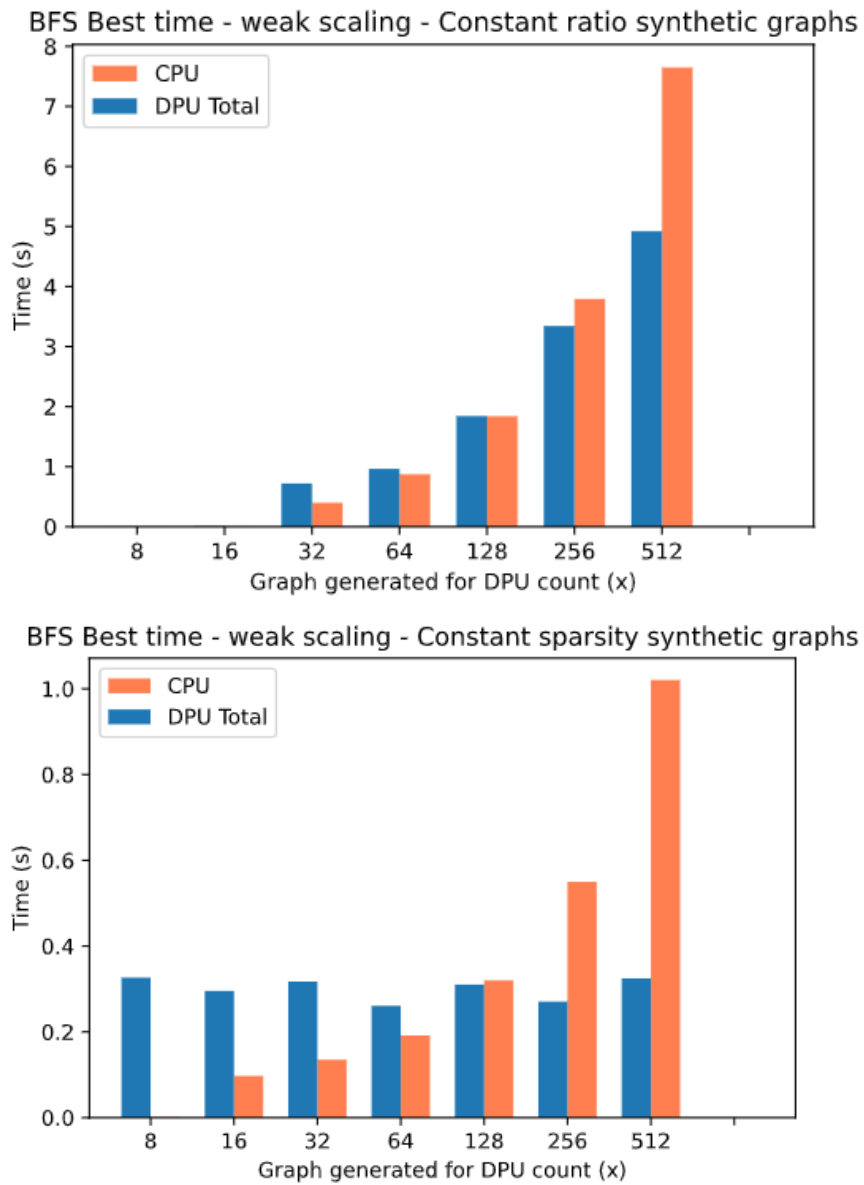


Figure 5.19: Comparing CPU and DPU BFS for constant sparsity graphs

For the graphs used in the strong-scaling benchmarks, the best CPU implementation outperforms our best DPU implementation, usually by several orders of magnitude (as seen in Figure 5.16). This indicates that these graphs are small enough to fully or partially fit in the CPU cache, leading to a blazing fast performance and negating the DPUs' advantages. However, the huge generated graphs `gen_low_ratio` and `gen_high_ratio` (which do not fit in a CPU cache) manage to come close the CPU version, even beating in the case of `gen_high_ratio`.

For the synthetic graphs used in the weak-scaling benchmarks, the results are far more positive (see Figures 5.18, 5.17, 5.19).

In the constant average degree (ratio) graphs, we notice that, while CPU BFS outperforms DPU BFS for the smaller graphs, the DPU BFS performs better as the graph size increases.

In the constant sparsity graphs, the performance of BFS DPU remains consistent as the graph size increases. As the graph grows very large, BFS CPU eventually becomes slower than BFS DPU.

**In summary:**

- BFS CPU is better for small graphs that can fit in the CPU cache (as that would eliminate data movement costs).
- BFS DPU is better for huge graphs. The bigger the graphs, the more likely the advantages of DPUs come into play.
- BFS DPU scales very well on graphs that grow with constant sparsity.

# Chapter 6

## Related Work

### 6.1 Motivation for Processing-In-Memory

Processing-In-Memory was shown to be quite effective for some modern consumer workloads. Google researchers sought out to investigate the effectiveness of PIM at reducing the performance and energy costs of some workloads in the interest of improving battery life in consumer devices. [9]. For their tests, they used a low-end laptop modified to use 3D-stacked memory embedded with a custom ARM processor.

By analyzing the data movement costs of common Google consumer workloads such as Chrome, TensorFlow, and video playback, they found that data movement contributes to as much as 62.7% of the workloads' total energy consumption. This was mostly due to functions and procedures that are good candidates to be executed on the low-powered PIM logic. Candidate functions are memory intensive functions where the data movement energy costs represents the largest component of their total energy consumption.

After offloading these functions to the PIM processor, their analysis showed that PIM can reduce total system energy usage by an average of 55% across the workloads and execution time by an average of 54%.

While the performance and energy savings of PIM on consumer devices is impressive, it is exciting to study how well these savings may translate to HPC-oriented highly data-intensive workloads. In this work, we intended to study the effectiveness and feasibility of PIM acceleration of common graph-processing workloads.

## 6.2 Processing-In-Memory Architectures

The fundamental idea behind PIM is to do the processing and the data-to-be-processed in the same (or near) physical location. In a hardware sense, this translates to physically reducing the distance between the processor and the memory as much as possible, so much so that both practically reside on the same package. This eliminates the data access latency, allowing the processor to spend more time on compute and less time waiting for the data to be fetched.

There have been several approaches to PIM architectures. Some approaches are application-specific; aimed at designing dedicated accelerators and frameworks that address a narrow scope of workloads as efficiently as possible. Other approaches are generic, aimed at creating more flexible frameworks that address a large scope of workloads exhibiting similar data access patterns. The intention is to make PIM more universal, thereby reducing production costs and making market adoption far more likely, similar to how the adoption of GPUs disrupted the HPC industry.

In the following sections, we discuss several PIM architectures, some of which are generic while others are specialized towards graph-processing. We compare them to one another and to our architecture of choice, the DPU.

Name	Memory Type	Compute Granularity	Generic
DPU	DRAM	1 DPU per 64MB	Yes
DRISA	DRAM (modified)	Ops per memory cell	Yes
Tesseract	Hybrid Memory Cube	128 cores per HMC module	No
GraphH	Hybrid Memory Cube + SRAM buffers	multiple cores per HMC module	No

Table 6.1: Overview of existing PIM architectures

### 6.2.1 DRISA

DRISA stands for DRAM-based Reconfigurable In-Situ Accelerator architecture [10]. In this architecture, every memory bitline is able to perform bitwise Boolean operations. By using combinations of these operations, DRISA can be configured to compute various functions on the data. In other words, the compute is happening at the level of the memory cell itself.

Massive parallelism can be achieved by simultaneously activating multiple rows. This completely eliminates data movement costs. Their experiments show that DRISA can achieve 8.8x better performance and 1.2x better energy efficiency compared to GPUs performing the same workload.

However, this PIM approach suffers from many challenges that might hamper market adoption. Firstly, the highly unconventional architecture may not allow it to take full advantage of established manufacturing technologies. The required

innovation makes it less likely to benefit from economies of scale. Secondly, building a software API around a non-Von Neuman architecture could prove challenging for both producers and consumers of the API. Thirdly, it has the limitation of being unsuitable for floating-point operations.

In comparison, DPUs are simple RISC processors embedded in DRAM and are already in the process of being mass produced. DPUs have a conventional C API which any experienced programmer can use, significantly reducing the friction of writing workable software for PIM. Lastly, DPUs are capable of floating-point operations, though at less performance than integer operations.

These drawbacks make DRISA only suitable for use in dedicated accelerators and FPGAs.

### 6.2.2 Tesseract

Recent advancements in 3D-stacked memory such as Hybrid Memory Cube has attracted more attention to processing data in-memory. In 3D-stacked architectures, logic and memory layers can be stacked in a single package. Tesseract [11] exploits this technology for graph-processing. Tesseract is a programmable accelerator for graph-processing. It features multiple in-order cores in 3D-stacked memory, hardware prefetchers specialized for graph-processing, and a programming interface to exploit this design. This removes the need to load the whole graph into the main processor. Tesseract cores can inter-communicate via Message Passing, allowing for faster neighbor traversal in large graphs. Inter-core communication makes locking and synchronization of critical importance despite the added complexity. Using this architecture, they managed to improve a performance by a factor of ten and reduce energy costs by 87%.

One downside with Tesseract is that it exhibits excessive crosscube communications as the cube interconnects have much less bandwidth than the aggregated local bandwidth of the Hybrid Memory Cube [12]. Another downside is the reliance on expensive 3D-stacked memory, instead of the ubiquitous DRAM. Unfortunately Micron is moving away from Hybrid Memory Cube [13], which could drive costs up even further.

DPUs in comparison, rely on cheap DRAM and employ a much simpler programming model. Instead of DPUs inter-communicating, the CPU will be responsible for synchronization and aggregation of data. This could make neighbor traversal on DPUs potentially less efficient however.

### 6.2.3 GraphH

Similar to Tesseract, GraphH [14] is a graph-processing centric approach to PIM based on Hybrid Memory Cubes. While Tesseract could exhibit local bandwidth degradation due to excessive inter-communication, GraphH solves this by integrating an SRAM-based on-chip vertex buffer. In addition, global bandwidth

is improved by introducing configurable double-mesh connections. GraphH also better balances workloads across cores and eliminates conflicts by introducing specialized partitioning and scheduling methods. Further optimizations are introduced in the form of methods to increase data reuse and reduce synchronization overhead.

Experiments showed improvements of two orders of magnitude over DDR-based graph processing solutions.

Overall, GraphH seems like a direct upgrade to the Tesseract architecture, introducing many optimizations that eliminate most of the later’s internal inefficiencies. However as more SRAM is now in the picture, the costs associated with such architectures could prove to be too high.

## 6.2.4 Memristor-based PIM Architectures

Instead of embedding processors inside the memory chips such as in UPMEM, some PIM architectures are based on memristor circuits. Memristors are circuits that can perform matrix-vector multiplications, allowing for PIM architecture. This technology is commonly referred to as Resistive random-access memory (ReRAM).

Some notable ReRAM-based PIM architectures: GraphR[15] for acceleration of graph processing — ISAAC[16], PRIME[17], and PUMA[18] for acceleration machine learning inference (i.e. deep learning) — PipeLayer[19] and PANTHER[20] for acceleration of machine learning training.

## 6.2.5 Other Works Using UPMEM Acceleration

UPMEM can be used to accelerate different kinds of workloads. Most notable works include DNA mapping [21], variant calling [22], and the acceleration of the k-means algorithm [23]. As the technology is still in its infancy, we expect to see more works that make good use of PIM in the future.

# Chapter 7

## Conclusion

We developed and evaluated nine variations of the Breadth-First Search algorithm to run on Data Processing Units, UPMEM’s Processing-In-Memory architecture, with the goal of overcoming the “memory-wall” problem.

These nine variations of BFS are the combinations of three graph partitioning strategies (Row-partitioning, Column-partitioning, and 2D-partitioning) and three BFS parallelization strategies (Top-Down, Bottom-Up, Edge-centric).

We found that, in order to maximize performance on DPUs, the host synchronization overhead must be minimized. 2D-partitioning of graphs proves to be the best partitioning strategy at keeping this overhead low. By combining 2D-partitioning with Top-Down BFS, we get the overall best performing BFS variation.

From our strong scaling benchmarks, we determined that BFS on DPUs scales best for very large graphs. However, for each graph, there exists an optimal number of DPUs to use; otherwise the host synchronization overhead grows larger than the performance improvement and we run into the memory-wall again.

From our weak scaling benchmarks, we determined that BFS on DPUs shows the best weak scaling when, as the graph grows, we add DPUs such that the average sparsity of the graph per DPU is maintained.

We compared our DPU-based BFS implementation to a sequential CPU-based implementation and found that, for small sized graphs, it is not worth using BFS on DPUs; specifically when a significant portion of the graph can fit in a CPU’s cache. Otherwise, as the graph size grows, BFS on DPUs becomes increasingly viable.

Using Processing-In-Memory acceleration for BFS leads to significant performance benefits in big data. As this cost-effective technology matures, and as we start seeing thousands of DPUs in a single system, the benefits could be immense.

# Bibliography

- [1] S. A. McKee and R. W. Wisniewski, *Memory Wall*, pp. 1110–1116. Boston, MA: Springer US, 2011.
- [2] “UPMEM Official Website.” <https://www.upmem.com/technology/>.
- [3] UPMEM, “[Video] UPMEM Presenting its true Processing-in-Memory solution @Hot Chips 2019.” <https://www.upmem.com/video-upmem-presenting-its-true-processing-in-memory-solution-hot-chips-2019/>.
- [4] “Coding tips and recommended practices – UPMEM DPU SDK 2020.2.1 Documentation.” [https://sdk.upmem.com/2020.2.1/fff\\_CodingTips.html#multi-threaded-programs-are-more-efficient-than-single-threaded](https://sdk.upmem.com/2020.2.1/fff_CodingTips.html#multi-threaded-programs-are-more-efficient-than-single-threaded).
- [5] “UPMEM DPU SDK.” <https://sdk.upmem.com/>.
- [6] TarekAS, “BFS Processing-In-Memory on DPUs code repository.” <https://github.com/TarekAS/bfs-pim-research>, 2020.
- [7] “MIT GraphChallenge.” <https://graphchallenge.mit.edu/data-sets/>.
- [8] F. Khorasani, R. Gupta, and L. N. Bhuyan, “Scalable SIMD-Efficient Graph Processing on GPUs,” in *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT ’15, pp. 39–50, 2015.
- [9] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and et al., “Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’18, (New York, NY, USA), p. 316–331, Association for Computing Machinery, 2018.
- [10] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, “DRISA: A DRAM-Based Reconfigurable In-Situ Accelerator,” in *Proceedings of the*

- 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-50 '17*, (New York, NY, USA), p. 288–301, Association for Computing Machinery, 2017.
- [11] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A scalable processing-in-memory accelerator for parallel graph processing,” in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 105–117, June 2015.
- [12] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, “GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 544–557, 2018.
- [13] <https://www.micron.com/about/blog/2018/august/micron-announces-shift-in-high-performance-memory-roadmap-strategy>.
- [14] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, “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.
- [15] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, “GraphR: Accelerating Graph Processing Using ReRAM,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 531–543, 2018.
- [16] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. W. Strachan, M. Hu, S. Williams, and V. Srikumar, “ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars,” *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 14–26, 06 2016.
- [17] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, “PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory,” *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 27–39, 06 2016.
- [18] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy, and D. S. Milojevic, “PUMA: A Programmable Ultra-Efficient Memristor-Based Accelerator for Machine Learning Inference,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19*, (New York, NY, USA), p. 715–731, Association for Computing Machinery, 2019.

- [19] L. Song, X. Qian, H. Li, and Y. Chen, “PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 541–552, 2017.
- [20] A. Ankit, I. Hajj, S. r. Chalamalasetti, S. Agarwal, M. Marinella, M. Foltin, J. P. Strachan, D. Milojicic, W.-m. Hwu, and K. Roy, “PANTHER: A Programmable Architecture for Neural Network Training Harnessing Energy-efficient ReRAM,” *IEEE Transactions on Computers*, vol. PP, pp. 1–1, 05 2020.
- [21] D. Lavenier, J. Roy, and D. Furodet, “DNA mapping using Processor-in-Memory architecture,” in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pp. 1429–1435, 2016.
- [22] “Variant Calling Parallelization on Processor-in-Memory Architecture, author=Lavenier, Dominique and Cimadomo, Remy and Jodin, Romaric,” in *2020 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pp. 204–207, IEEE, 2020.
- [23] S. Bihel, L.-A. Daniel, F. De Moor, and B. Thomas, “Evaluating a Processing-in-Memory Architecture with the k-means Algorithm,” 2017.

