UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

UNIVERSITY OF PADOVA

DEPARTMENT OF INFORMATION ENGINEERING

BACHELOR DEGREE IN

COMPUTER ENGINEERING

Approximate triangle counting with vertex coloring on the UPMEM architecture

Supervisor:
Prof. Francesco Silvestri

Candidate:
Lorenzo Asquini

ACADEMIC YEAR 2022-2023

DISSERTATION DATE 17 JULY 2023

Abstract

Due to the need in recent years to analyze an ever-increasing amount of data, the restrictions posed by DRAM memory bandwidth and latency limit the scalability of computing systems and impede faster execution of memory-bound workloads. To address these limitations, in the past decades, a paradigm known as Processing-In-Memory (PIM) has been proposed, which allows for part of the computation to be moved closer to where the data resides.

This thesis presents an implementation of an algorithm for approximate triangle counting in a graph specifically designed for the UPMEM PIM architecture, the first PIM architecture commercialized in real hardware. Through a series of comprehensive tests, the capabilities of the PIM architecture are highlighted, and the potential improvements it offers compared to traditional hardware solutions are demonstrated.

Furthermore, this thesis presents additional improvements that could be made to the proposed algorithm, taking advantage of all the capabilities of the UPMEM PIM architecture.

Sommario

A causa della necessità, negli ultimi anni, di analizzare una quantità sempre crescente di dati, le restrizioni poste dalla larghezza di banda e dalla latenza della memoria DRAM limitano la scalabilità dei sistemi computazionali e impediscono un'esecuzione più rapida di processi memory-bound. Per ovviare a queste limitazioni, negli ultimi decenni è stato proposto un paradigma noto come Processing-In-Memory (PIM), che consente di spostare parte delle operazioni computazionali più vicino a dove risiedono i dati.
Questa tesi presenta un'implementazione di un algoritmo di conteggio approssimato dei triangoli in un grafo specificamente progettato per l'architettura PIM di UPMEM, la prima architettura PIM commercializzata in hardware reale. Attraverso una serie di test, vengono evidenziate le capacità dell'architettura PIM e vengono dimostrati i potenziali miglioramenti che offre rispetto alle soluzioni hardware tradizionali.
Inoltre, questa tesi presenta ulteriori miglioramenti che potrebbero essere apportati all'algoritmo proposto, sfruttando tutte le capacità dell'architettura PIM di UPMEM.

Acknowledgements

This thesis has been one of the most interesting and fulfilling aspects of the last three years at the university. I therefore would like to deeply thank my supervisor, Professor Francesco Silvestri, for providing me with this opportunity and for his guidance throughout the entire journey of writing this thesis.

I would also like to thank the SAFARI Research Group at ETH, and in particular Dr. Juan Gómez Luna, for the help and for granting me access to the necessary hardware that greatly contributed to the realization of this thesis.

Deep and loving thanks to my family for their support and belief in my abilities throughout the years... To Francesco, my unpaid proofreader of many of my works of the past, present, and, possibly, future.

To my dear friends Federico, Federico, Anna, and Tommaso, I am immensely grateful for their support in times of despair and celebration in times of joy.

A final big thank you to all the people that I have encountered during the years who have helped me become who I am now.

Contents

Chapter 1

Introduction

1.1 PIM as a solution for DRAM limitations

Many modern applications, from neural network training to graph computing, need to process very large datasets. While some of these may be computationally bound, many others, as the data volume expands, are being slowed down by the constraints of *Dynamic Random Access Memory* (*DRAM*) technology that are presented below. This issue is also accentuated by the gap between processor and memory performance (the *memory wall*), which has widened during the last few decades, making the processor operate at a significantly faster pace than the memory can provide it with new data [1].

To perform computations on the data that is currently stored in memory, it needs to be moved from the memory to the CPU via the memory bus. Through this off-chip bus, the memory controller, after receiving a request from the CPU, issues different commands to the DRAM module containing the data. The data is then transferred to the CPU cache and eventually to the CPU registers, where the CPU can execute the needed operations on it. This complex process that is needed to transfer data to where the computation will occur adds a lot of latency and consumes a significant amount of energy, in addition to the energy that is already needed to refresh the memory cells and prevent data loss or corruption. For example, data movement accounts for 62% of the total energy consumed by Google consumer workloads on average [2]. In addition, it is difficult for the CPU to send a large number of parallel requests to the memory due to the limited bandwidth between the memory controller and the main memory. Another aspect to consider is that, despite the amount of time and energy needed to transfer the data near the CPU

inside its cache, many times the data is not reused, losing the advantages that usually come from the organization of the memory hierarchy, and making the cache almost useless in some types of workloads [3].

In recent years, workloads have become more and more data-centric, and the costs in terms of latency and energy consumption associated with accessing large amounts of data in memory have become a limiting factor in many applications. In parallel, trying to scale DRAM bandwidth, memory, and capacity has become more difficult. For these reasons, a new paradigm has been proposed. From a processor-centric design of the von Neumann architecture characterized by the limitations described above, the new paradigm moves part of the computation to where the data resides. The so-called *Processing-in-Memory* (*PIM*) has been proposed for decades (e.g. [4]), but only in recent years the technology has evolved enough to allow the computation to be executed directly within the memory, using, for example, 3D-stacked memory dies with DRAM layers and logic layers or general-purpose processing cores integrated in the memory dies. This last approach to developing a PIM architecture is the one that characterizes the UPMEM PIM architecture [5], which combines on the same chip conventional 2D DRAM arrays and general-purpose processing cores, called *DRAM Processing Units* (*DPUs*). This allows the CPU to offload some computation to these DPUs, treating them as co-processors, minimizing the amount of data that needs to be transferred from the memory to the CPU, and taking advantage of the physical proximity of the data to the DPUs to execute memory-heavy applications. [6, 7, 8]

1.2 Enhancing graph processing through PIM architecture

One important field where a PIM architecture may provide significant benefits is graph processing. In traditional systems, big graphs require a large amount of memory to be stored while computation is performed on their elements. However, an increase in the amount of memory inside a system to store bigger datasets does not result in an equal increase in memory performance. This is primarily due to the limitations imposed by the pin count, which sets the upper limit for memory bandwidth and latency regardless of the amount of memory available. In addition, usually applications that need to process graphs do not use the CPU cache in an effective way, increasing the bottleneck caused by the limited bandwidth of the data transfer

between the CPU and the memory. This is caused by the fact that the applications require large amounts of random memory accesses across large memory regions, accessing sections of the graph that are located in different parts of the memory, with limited cache efficiency due to the difficulty of taking advantage of both temporal locality, because the same nodes and edges are rarely used multiple times consecutively, and spatial locality, because nodes and edges are stored in different parts of the memory. This leads to the need for a large amount of bandwidth, which cannot be scaled just by adding more memory capacity; only increasing the number of computing systems would be an effective solution. Making the poor utilization of cache worse is the fact that, usually, for each item to process inside the graph, little computation is needed and new elements are needed very frequently. This puts even more strain on the limited memory bandwidth, resulting in poor utilization of compute resources that are not utilized while waiting for new data to arrive.

For the reasons described above, a PIM architecture would be ideal for graph processing. The bandwidth would significantly increase due to the physical proximity of the processor to the place where the data is stored, and also due to the possibility of increasing the total bandwidth with an increase in memory size. [9, 10]

1.3 Triangle counting in graphs using the UPMEM PIM architecture

One particular graph processing algorithm with many real-world applications is graph triangle counting. Such an algorithm can be used for community detection [11], topic mining [12], detecting sybil accounts and measuring content quality in social networks [13, 14], and spam detection [14].

Different algorithms have been proposed to efficiently count the number of triangles inside a graph (for example, [15] and [16]), but they are limited in performance by the bandwidth of the memory while retrieving the data relative to the neighbors of the vertices inside the graph, although with some improvements that take advantage of parallel computation and that allow using a limited amount of memory.

A new approach is presented in this thesis to count the number of triangles in a graph using the UPMEM PIM architecture. This PIM architecture is designed and developed by UPMEM,

a fabless semiconductor company recognized as the leader in Process-In-Memory technologies [17]. Taking advantage of the possibility of having thousands of DPUs in a single system, each with access to its own memory bank, the counting of the number of triangles inside the given graph is executed in parallel by the DPUs. A subset of the edges of the graph is assigned to each DPU through vertex coloring. Each edge of the graph is colored and is then assigned to one or more DPUs, guaranteeing that each triangle can be counted even if each DPU has access to only a subset of all the edges of the graph, similarly as described in [15]. If the amount of memory available to a DPU is not enough to store all the edges assigned to it, acting similarly as described in [16], edges inside the reservoir of edges specific to a DPU are replaced by new ones, statistically compensating for the lost triangles caused by the removal of some edges, resulting, for this reason, in an approximate count of the number of triangles.
The proposed algorithm is tested on real UPMEM hardware, accessed thanks to the SAFARI Research Group at ETH Zurich, testing its behavior when changing the algorithm's different parameters and showing its performance when counting the number of triangles in real-world graphs.

1.4 Thesis organization

The thesis is organized as follows:

- **Chapter 2** provides an overview of various approaches to achieve Processing-In-Memory (PIM), focusing on the UPMEM PIM architecture and its detailed description.
- **Chapter 3** presents the problem addressed by the algorithm proposed in this thesis, along with a comprehensive explanation of the algorithm's implementation on the UPMEM PIM architecture.
- **Chapter 4** details the different tests conducted to optimize the algorithm's performance by adjusting its parameters. A comparison is made with a standard CPU implementation.
- **Chapter 5** concludes the thesis by discussing the applicability of the proposed algorithm and suggesting potential improvements to address any performance issues encountered during the testing phase.

Chapter 2

The UPMEM architecture

DRAM is the predominant technology used to build main memory, but the trend in recent years of needing to quickly analyze large amounts of data proved that the slow and challenging scaling of DRAM technology is a barrier to fast and cost-effective algorithms. It is difficult to scale, at the same time, capacity, bandwidth, and latency, all characteristics needed to improve the rate at which it is possible to analyze and perform computations on large amounts of data.

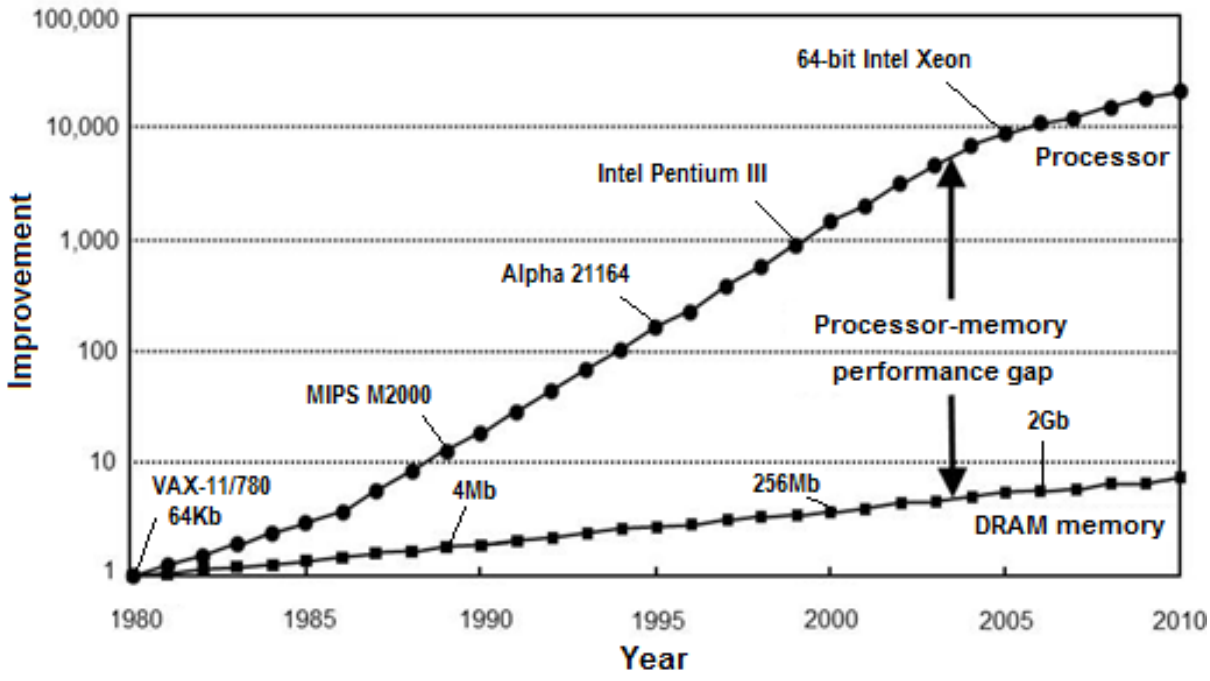


Figure 2.1: (Reproduced from [1])

Yearly improvement of processor and DRAM memory speeds,

for a time period of three decades

In the last decades, DRAM capacity has increased more slowly than computational power, and this processor-memory performance gap is even more present when considering memory bandwidth and memory latency, which remained almost constant during the years (figures 2.1 and 2.2).

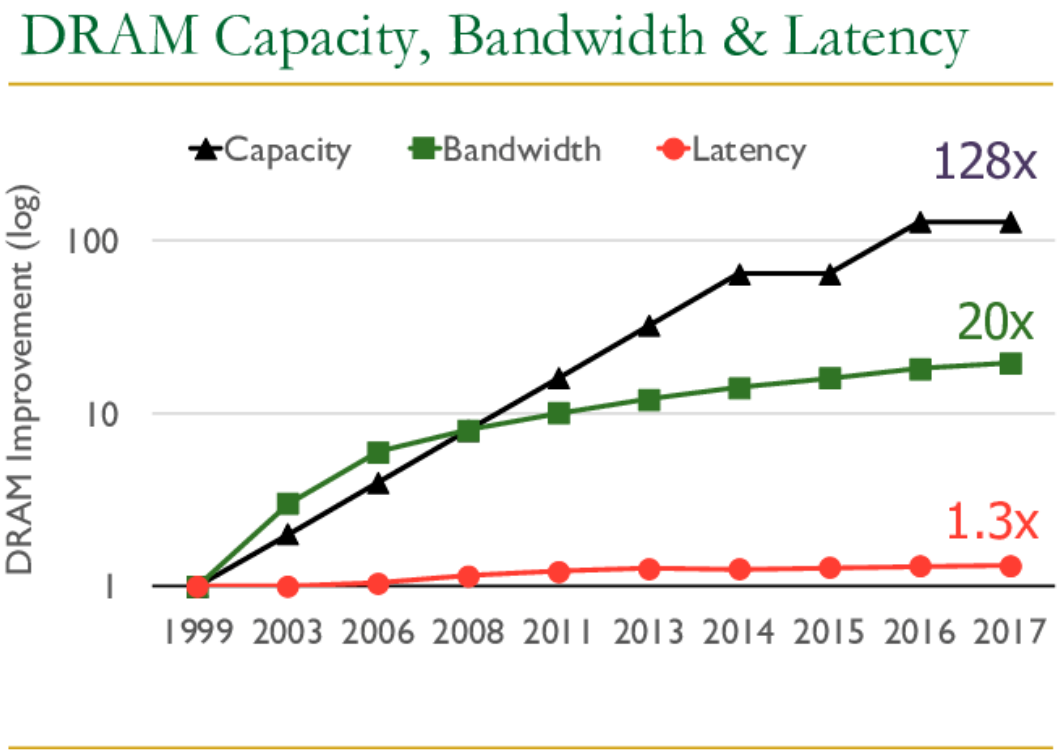


Figure 2.2: (Reproduced from [18])
Scaling of DRAM capacity, bandwidth and latency between 1999 and 2017, normalized to the value in 2017

These slow improvements in DRAM technology, considering also the fact that the main memory uses a lot of energy when not in use (e.g., periodic memory refresh), made it necessary to introduce a new paradigm, that is *Processing-In-Memory* (*PIM*).

2.1 Advancements in PIM architectures

During the last decades, different implementations of architectures that allow Processing-In-Memory have been proposed, but, until recently, the available technology has made it too difficult to manufacture such designs. In addition, in the past years, there were no workloads that were as constrained by memory as today, where the data movement bottleneck is critical to

system cost, energy, and performance.

In recent years, two major innovations have emerged that could lead to a future where Processing-In-Memory can be widely used.

The first innovation that could revolutionize the way we think about the role of memory is *3D-Stacked Memory* [19, 20], which allows to create memory chips that are composed of multiple layers of memory stacked on top of each other. These layers are connected using vertical *through-silicon vias* (*TSVs*), which can be thousands per single 3D-stacked memory chip. The TSVs provide much greater internal bandwidth within the layers compared to the external bandwidth of the memory channel. This technology allows for greater transfer speeds within the chip itself, and it also incorporates a logic layer inside the chip. This layer, usually situated at the bottom of all the others, can embed processing elements and could be used to implement functionalities that interact both with the processor and the DRAM. This approach, although the logic layer could be used to add different computation logics to the DRAM chip, presents some restrictions. The capabilities of this logic layer are limited by hardware area, energy consumption, and thermal dissipation constraints.

The second innovation that could provide PIM capabilities to future systems is the use of byte-addressable *resistive nonvolatile memory* (*NVM*) [21]. Researchers and manufacturers are developing new hardware devices with the goal of storing data at higher densities than those typically available in existing DRAM. Starting from nothing while developing a new technology, it is possible to design it in a way that can be useful not only for storing data. This led to the idea of integrating PIM functionalities inside NVM memory arrays. In particular, one idea that could be successful with this goal would be to use the memory cells themselves to perform logic operations [22]. This approach, however, is limited to some basic logic operations, while more complex ones would require a large area of the memory chip to be allocated to this task, also considering the necessity to operate only on a few bits when executing more precise operations, and not only on entire rows.

These two innovations are examples of two different paths that can lead to Processing-In-Memory. A 3D-stacked design with a logical layer follows the approach of *Processing-Near-Memory*, where PIM logic is integrated close to or inside the memory (e.g. [9]). The PIM logic can be integrated inside the logic layer of memories manufactured with 3D-stacked technology or inside the memory controller, but recent developments in silicon interposers (in-package extremely fast electrical signal conduits that connect directly to the TSVs) could allow for separate

logic chips in the same die package as the 3D-stacked memory that can take advantage of the full bandwidth provided by the TSVs [23].
On the other hand, NVM is an example of the *Processing-Using-Memory* approach, where the properties and operational principles of memory cells are used to perform computation through the interaction between the cells themselves (e.g., [24]).

2.2 The UPMEM PIM architecture

The possible technological paths that can be followed to reach a useful and performing architecture which allows Processing-In-Memory described above are still under development and research, so there is still no commercial implementation of those solutions ready for use.
UPMEM, a fabless semiconductor company recognized as the leader in Process-In-Memory technologies [17], on the other hand, successfully manufactured PIM-enabled memory DIMMs, commercialized in real hardware, that could be integrated directly into current systems. With a different approach than the ones described above, to avoid the limitations mentioned, the UPMEM PIM architecture uses conventional 2D DRAM arrays combined with general-purpose processing cores, called *DRAM Processing Units* (*DPUs*), on the same chip. This integration results in a DRAM silicon area overhead of 30% to 50% [25]. The manufacturing of these chips imposes different challenges that limit the fabrication of fast logic transistors, but these problems are softened by the fact that the DPU cores are relatively deeply pipelined and fine-grained multithreaded.
The UPMEM PIM architecture has different advantages compared to other proposed PIM architectures. It uses mature 2D DRAM design and fabrication, without relying on emerging 3D-stacked memory technologies. The resulting general-purpose DPUs support a variety of computations and data types; the threads can be executed independently; and, through a complete software stack provided by UPMEM, it is possible to program the DPUs through C code [25][26].

2.2.1 System organization

In a system that uses PIM-enabled memory manufactured by UPMEM, there is a host CPU, standard DRAM memory, and PIM-enabled memory, which can reside on one or more memory

channels (as presented in figure 2.3). All the DPUs in the UPMEM modules operate as parallel co-processors for the host CPU.

Inside each UPMEM PIM chip, there are 8 DPUs. Each DPU has exclusive access to a 64-MB DRAM bank, called *Main RAM* (*MRAM*), a 24-KB instruction memory that can store up to 4,096 48-bit encoded instructions, called *Instruction RAM* (*IRAM*) and a 64-KB scratchpad memory, a high-speed memory used to hold small items of data for rapid retrieval, called *Working RAM* (*WRAM*). PIM-enabled memory maps entire 64-bit words inside the MRAM bank, allowing the DPUs to operate on data types of up to 64 bits while having access to only a MRAM bank.

To move instructions from the MRAM bank to the IRAM and data between the MRAM bank and the WRAM, it is possible to use DMA instructions provided by the *Instruction Set Architecture* (*ISA*).

The MRAM is accessible by the host CPU for copying and retrieving data to and from the DPUs. These transfers can happen serially or in parallel and can be used to indirectly transfer data between DPUs because there is no support for direct communication between the DPUs themselves.

The host CPU can allocate as many DPUs as needed, limited by the number of DPUs available in the system. After loading the DPUs' source code, the *kernel*, the CPU host can launch the DPU kernel synchronously, suspending the host CPU thread operations until the DPUs conclude the kernel execution, or asynchronously, where the execution returns immediately to the host CPU thread.

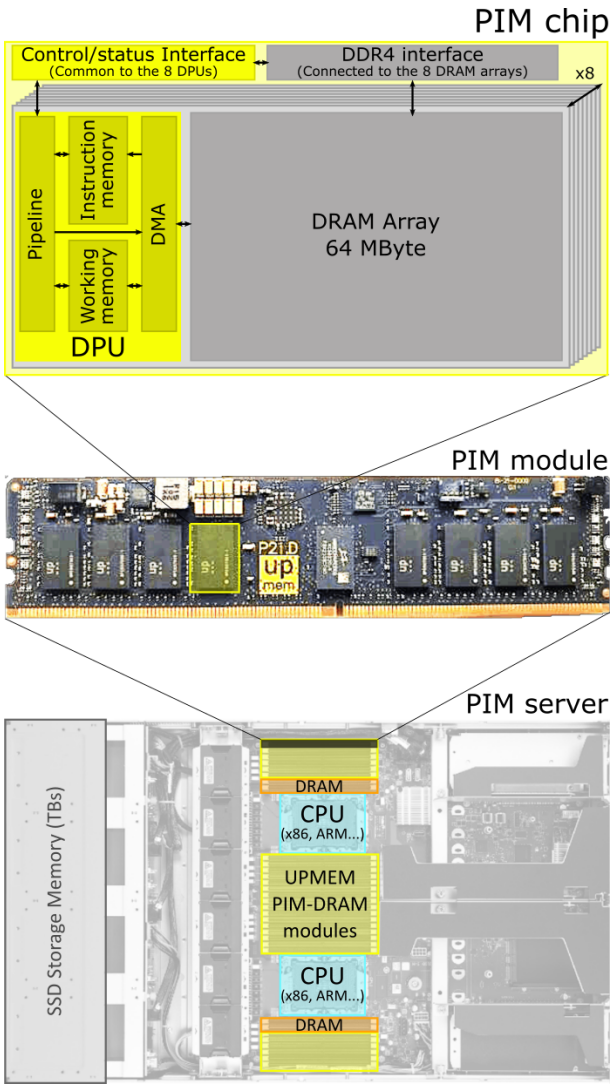


Figure 2.3: (Reproduced from [25])
UPMEM PIM DRAM solution

2.2.2 The DPU architecture

A DPU is a RISC® processor with a specific *Instruction Set Architecture* (*ISA*), composed of 100 48-bit instructions [25]. Each DPU has 24 hardware threads, called *tasklets*, each with its own set of 24 32-bit general-purpose registers. They share the IRAM and the WRAM, where operands are stored. The DPU instruction pipeline is composed of 14 stages. Because only the last three stages of the pipeline can be executed in parallel with the DISPATCH and FETCH stages of the next instruction in the same thread, instructions from the same thread are dispatched 11 cycles apart, requiring at least 11 threads to fully utilize the pipeline in a DPU.
The DPUs provide native support for 32-bit and 64-bit integer additions and subtractions, while 32-bit and 64-bit integer multiplications and divisions, as well as floating point operations, are not natively supported. To compensate for this limitation, the UPMEM runtime library emulates these operations, resulting in significantly reduced throughput. [27]

Chapter 3

Triangle counting in a graph on the UPMEM PIM architecture

Generally, graph analysis workloads suffer from limited memory bandwidth. This is caused by the fact that it is usually necessary to perform a large number of random accesses to memory across large memory regions to fetch a certain element of the graph that may not be located near the element that was just fetched. This leads to limited cache efficiency due to the difficulty of exploiting cache spatial locality, making it necessary to continuously transfer large amounts of data through the memory bus. In addition to this, cache temporal locality cannot be efficiently used due to the fact that, usually, the data about a specific element of a graph is not reused frequently, requiring access to new data for almost each new computation.

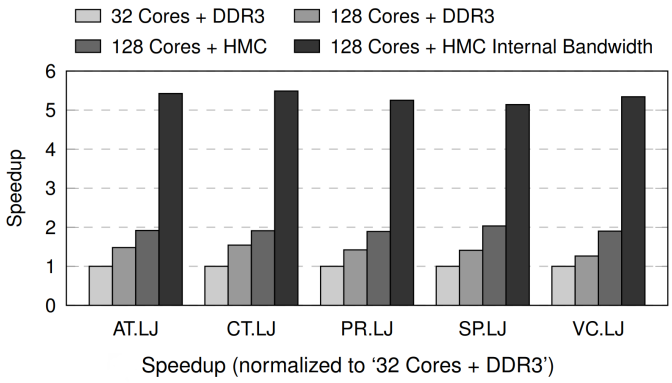


Figure 3.1: (Reproduced from [9])
Performance of large-scale graph processing in conventional systems versus with ideal use of the Hybrid Memory Cube (HMC) internal memory bandwidth

Another aspect to consider is that, in graph analysis workloads, the amount of computation needed per element of the graph is usually quite small, leading to the necessity of a continuous

fetch of data from memory and exacerbating the difference in time and energy needed for the computation and the data transfer.
These aspects make it difficult to scale up this type of workload using conventional architectures, especially considering that simply increasing the number of cores to parallelize the specific analysis of a graph is ineffective in improving performance due to the limited increase in memory bandwidth with an increase in core count (see figure 3.1) [9].

Such problems could be solved using a system that takes advantage of a PIM architecture, which could significantly speed up the processing of graphs. Moving the computation inside the memory, where the object of the computation is stored, would both decrease the memory latency while accessing data and make it possible to increase the total memory bandwidth proportionally to the memory size compared to traditional systems. With an increase in the number of memory chips with processing capabilities that could come from installing more PIM-enabled memory DIMMs, the total amount of available bandwidth also increases because each processor has its own dedicated link to its memory.
Considering the potential benefit of using a PIM architecture to analyze large graphs, in this thesis an implementation of an algorithm for triangle counting in a graph designed to run on the UPMEM PIM architecture is presented and discussed.

3.1 Algorithm description

Given an unweighted, undirected graph $G = (V, E)$, where V represents the set of vertices and E represents the set of edges, the triangle counting problem aims to determine the total number of triangles present in the graph. A triangle is defined as a set of three vertices such that any two of them are connected by an edge of the graph. In other words, a triangle consists of three vertices u, v, and w such that there exist edges (u, v), (v, w), and (w, u) in the graph.
Many different algorithms have been proposed over the years to find the number of triangles in a graph as quickly as possible. Some implementations of this algorithm have been proposed with the goal of providing the exact number of triangles (e.g., [28, 15]), while others only provide an approximation while using less resources and time (e.g., [29, 16]).
In this thesis, it is presented an algorithm that can provide the exact number of triangles inside

the given graph or an approximation, according to the given graph size and the parameters used while executing the algorithm.

The algorithm proposed for the triangle counting problem utilizes both the CPU and the DPUs present in the PIM-enabled memory installed in the computing system. The CPU first retrieves the edges, each consisting of two vertices connected within the graph, from a file. The CPU transmits these edges to the DPUs, and each one will store a different subset of them locally. The DPUs then independently count the triangles in the subgraph assigned to each DPU. Once all DPUs have completed their triangle counting, the results are gathered and combined by the CPU, which ultimately returns the final result.
Distributing the edges among the DPUs presents two challenges. Firstly, the total amount of memory combining the MRAM of all the DPUs is limited, and it may be insufficient to store all the necessary graph data. This limitation may restrict the applicability of the algorithm, particularly for very large graphs. To address this, a technique is employed where some edges are ignored locally within the DPUs, and after statistical correction, an approximate count of triangles is obtained after counting the triangles formed by the remaining edges. This technique may also be used to obtain an approximate triangle count while consuming fewer time and resources, even if the available space is adequate to store all the graph data.
The second challenge involves distributing the edges from the CPU to the DPUs in a manner that ensures all triangles are counted while considering that the graph is partitioned among the allocated DPUs. This distribution technique aims to avoid direct communication between the DPUs themselves while still guaranteeing accurate triangle counting.

When considering the aforesaid case where the algorithm produces an approximate result, the edges inside each DPU are handled similarly as described in [16].
When a new edge is assigned to a DPU (according to the procedure described in section 3.3.2), the edge may be handled in two different ways. Defining the *sample* as a reservoir of edges contained inside the MRAM of a DPU, M the maximum allowed edges inside the sample, and t as the total number of edges assigned to a DPU, including the current edge that is being considered:

- if $t \leq M$, then the new edge assigned to the DPU is inserted inside the sample. If this

inequality is valid when the last edge is assigned to the DPU, then the triangle count provided as the output by the single DPU will be exact;

- if $t > M$, then more edges than the number that are allowed to be stored in the DPU's sample are assigned to the DPU. A biased coin with a heads probability of $\frac{M}{t}$ is tossed. If the outcome is a head, an edge inside the sample is chosen uniformly at random and replaced by the new edge. Otherwise, the new edge is discarded. If this inequality is valid when the last edge is assigned to the DPU, then the triangle count provided as the output by the single DPU will require a statistical correction, as described below.

At the end, when all the triangles τ defined by the edges inside the sample of a single DPU have been counted, a parameter $\xi = \min\left\{\frac{M\cdot(M-1)\cdot(M-2)}{t\cdot(t-1)\cdot(t-2)}; 1\right\}$ is calculated. The purpose of calculating ξ is to assess the completeness of the sample. If the sample was large enough to include all the edges assigned to the DPU, ξ will be equal to 1. However, if the capacity of the sample was insufficient and edges were replaced within the sample, ξ will be lower than 1.
At this point, the approximate triangle count will be defined as $\frac{\tau}{\xi}$ (for a detailed proof and further information, refer to [16]). If the sample contained all the edges, the result would be equal to τ, representing the exact number of triangles. If the sample couldn't accommodate all the edges and some replacements occurred, the result will be greater than τ, indicating an approximate count of the triangles formed by the edges assigned to the DPU.
This design choice was made to not limit the applicability of the algorithm to only graphs with a dimension compatible with the PIM system and to allow for a quicker, even if not exact, result when needed. Because of the way the edges are divided among the DPUs, it may be possible to have an approximate result only from some of them and an exact result from others, which will still lead to a total approximate result given that, at the end, the host application will add up all the DPUs outputs.

To address the challenge of dividing the edges among the DPUs while ensuring that all triangles could be counted, a coloring technique is employed. The technique involves using C different colors to color the vertices of the graph, similar to what is proposed in [15]. The hash function that is used to color always with the same color the generic node with identifier u is:

$$h_C(u) = ((a \cdot u + b) \mod p) \mod C \tag{3.1}$$

where p is a prime number, a is a random integer in $[1, p-1]$, and b is a random integer in $[0, p-1]$.

Each possible triangle that could form inside the graph using this coloring technique is defined by a triplet of colors. By utilizing C colors, it becomes possible to create $\binom{C+2}{3}$ different triplets of colors. It's important to note that for these triplets, the order of the colors does not matter, as two triplets with the same colors in a different order are considered equivalent. When distributing the triplets among the DPUs (according to the procedure described in section 3.3.1), each DPU is assigned a specific set of triplets, which could also be empty. Consequently, the edges contained within the DPU's sample only consist of those compatible with the assigned triplets, which means that all the edges assigned to a DPU contain colors that could form triangles colored with the colors of one of the DPU's triplets. As a result, once each DPU completes the counting of triangles formed by its assigned edges and normalizes the result if necessary (as explained previously), it is guaranteed that all the triangles formed inside the graph could have been considered.

One issue that may arise is that a triangle is counted by two different DPUs. Given, for example, a triangle that, in the graph, has nodes all of the same color A. This triangle should be counted by the DPU which has the triplet $A-A-A$ assigned to it, with all the edges colored $A-A$ inside its sample. Because the edges are considered individually, a DPU with a triplet $A-A-B$ assigned to it will also have all the edges colored $A-A$ inside its sample. This would lead to counting the triangles colored $A-A-A$ inside both DPUs. To prevent the triangle from being counted by multiple DPUs, an additional check is performed. When a triangle is encountered while processing the edges within a DPU's sample, it is counted only if the colors of that triangle match the colors specified in the triplets assigned to that DPU.

3.2 Host application

The host application has the duty to coordinate the work of the different DPUs by sending the necessary data, launching the DPUs' kernel, and retrieving the final results.

The operations performed inside the host application are the following:

- **Receiving parameters:** the host application receives different parameters when run to easily customize the execution of the algorithm;

- **DPUs setup:** the host application loads the kernel inside the DPUs. It then calculates and sends the values that are needed to the DPUs;
- **Sending the graph edges to the DPUs:** the host application reads the file containing all the edges and loads the content in different batches of edges that are sent one by one to the DPUs when full or when the graph has ended. After each batch of edges is sent, the DPUs' kernel is started to allow the processing of that particular batch;
- **Retrieving the results:** the host application launches for the last time the DPUs' kernel and retrieves the counting results from all the DPUs, adding them up.

3.2.1 Receiving parameters

The following parameters are given to the host application at the start:

- **Seed:** the seed that will be used to set the random number generator seed, both for the host and for the DPUs;
- **Maximum edges inside the sample:** the maximum number of edges that can be stored inside the reservoir of each DPU. This can be any positive number lower than the maximum allowed by the system configuration. The maximum allowed value depends on the memory occupied by the batches of edges sent from the host application to the DPUs and on the total available memory inside the DPUs' MRAM. From now on, this value will also be referred to as *sample size*;
- **Number of colors:** the number of colors used to create the triplets assigned to the DPUs;
- **Path to the file containing the graph:** the graph is described with a list of edges. Each line of the file describes an edge, which is composed of two non-negative 32-bit numbers separated by a space character that represent the two nodes connected by the edge itself. Inside the file, there cannot be any duplicate edges.

3.2.2 DPUs setup

The parameters required for coloring the nodes within the DPUs using the aforementioned hash function (equation 3.1) are computed. These parameters will be the same for all the DPUs to maintain consistency in vertex coloring.

The DPUs' kernel is loaded into all the allocated DPUs, and, using a broadcast, the random number generator seed, the maximum number of edges inside the sample, the number of colors used to form the triplets, and the parameters related to the coloring hash function are sent to all the DPUs. Broadcasting is chosen as the optimal method for this data transfer, as highlighted in [27], due to its efficiency in transmitting identical data to multiple DPUs.
Additionally, an individual identification number, ranging from 0 to the total number of DPUs minus one, is sent to each DPU sequentially.

3.2.3 Sending the graph edges to the DPUs

Following the initialization phase, the next step involves transmitting the edges from the host application to the DPUs. Because, as stated in [27, 25], it is better to execute on the DPUs portions of parallel code that are as long as possible, the host application sends large amounts of edges in batches to the DPUs. Each batch is formed by reading a certain number of edges from the given file, and transforming the unordered graph into an ordered one, making sure that the nodes within each edge are arranged in ascending order.
After each batch is sent to the DPUs using a broadcast, the DPUs' kernel is started asynchronously. This allows the host application to prepare the next batch while the current one is processed by the DPUs.

3.2.4 Retrieving the results

Once all the edges from the graph file have been transmitted to the DPUs, the host application waits for the handling of the last batch by the DPUs before starting their kernel once again synchronously. This last execution of the DPUs' kernel involves counting the triangles. Once all DPUs have completed their counting, the host application collects the individual triangle count results from each DPU, adding them together to obtain the final overall triangle count.

3.3 DPU Kernel

The DPUs, coordinated by the host application, execute most of the operations related to the counting of triangles, from the organization of the edges to the actual counting.
The operations performed inside the DPUs' kernel are the following:

1. **One-time variable setup:** different variables required for the multiple executions of the kernel are initialized;

2. **Insertion of edges inside the sample:** as long as the graph has not reached its end, the following operations are performed for each edge in the different batches sent from the host application to the DPUs:

   - Retrieve an edge from the current batch of edges;
   - Color the nodes of the edge using the hash function described above (equation 3.1);
   - Verify if the current edge should be handled by the DPU according to the assigned triplets. If not, the edge is discarded;
   - If the edge is not discarded, it is inserted into the sample if space is available. If the sample is full, an existing edge inside the sample may be replaced based on the result of a biased coin flip.

3. **Counting the triangles:** the edges within the sample are sorted, and the locations of the nodes are determined. The triangles formed by the edges within the sample are then counted. If necessary, the result is statistically corrected.

3.3.1 One-time variable setup

Only when the DPUs' kernel is executed for the first time, it begins by initializing global variables required for each iteration of the kernel. This operation is performed by a single tasklet, but the resulting variables are made available to all the tasklets used.

The random number generator seed provided by the host application is used to set the local random number generator seed for the DPU. Since the DPUs lack a built-in function to generate random numbers, a linear congruential generator (LCG) is employed. [30]

To determine the number of triplets assigned to the specific DPU and to identify them, first of all, the total number of triplets that can be created with the given C colors is calculated using $\binom{C+2}{3}$. Considering a round-robin distribution of the total triplets between the DPUs, the maximum number of triplets that any DPU can have is calculated by taking the ceiling value of the division between the total number of possible triplets and the total number of allocated DPUs. After that, each triplet that can be created using C colors is generated, giving a number to each of these triplets starting from 0. When the number assigned to the triplet leaves a remainder

equal to the DPU ID when divided by the total number of available DPUs, that particular triplet is assigned to the DPU in question.

Once the single tasklet completes the global setup, each tasklet allocates its buffer in the WRAM. This is used to transfer to the WRAM large amounts of bytes when accessing data inside the MRAM. This improves the performance of the execution because, as stated in [27, 25], it is faster to use large DMA transfer sizes when all the accessed data stored in the MRAM is going to be used.

The assignment of triplets to the DPUs may result in unbalanced divisions in terms of the number of edges that each DPU needs to handle within its sample. Given C colors, there are:

- C triplets containing only one color;
- $2 \cdot \binom{C}{2}$ triplets containing two different colors;
- $\binom{C}{3}$ triplets containing three different colors.

Considering E the total number of edges inside the graph and assuming an optimal distribution of triplets between the DPUs where only one triplet is assigned to each DPU, the DPUs with a triplet containing only one color will receive $\frac{E}{C^2}$ edges, the DPUs with a triplet containing two different colors will receive $3 \cdot \frac{E}{C^2}$ edges, and the DPUs with a triplet containing three different colors will receive $6 \cdot \frac{E}{C^2}$ edges. This leads to an imbalance in the number of edges distributed among the DPUs, with the DPUs having triplets containing three different colors handling a larger number of edges compared to the other DPUs.

A better assignment of the triplets to the DPUs may seem necessary, maybe assigning more triplets with only one or two colors to the same DPU. However, this would result in a significant slowdown. Each new edge would need to be checked against a larger number of triplets (as described in section 3.3.2), which becomes time-consuming for larger graphs. Moreover, the benefits that come from having fewer edges to handle inside a DPU, which leads to faster ordering and triangle counting times (as described in section 3.3.3), would be counterbalanced by the need to check more triplets when determining whether a discovered triangle should be counted or not. For these reasons, although not optimal, allowing certain DPUs to handle considerably more edges provides the best performance outcome with this implementation.

3.3.2 Insertion of edges inside the sample

The DPUs kernel is launched by the host application every time a new batch of edges is sent to the DPUs' MRAM. Considering a batch of edges containing a number of edges greater than zero, the batch is processed, dividing the operation equally between the different taklets. Each tasklet divides its buffer allocated in the WRAM into two portions. The first portion is dedicated to retrieving and storing a subset of edges from the current batch for analysis; the other is used to store the edges that need to be sent to the DPU sample.

Each tasklet is responsible for handling a portion of the batch of edges, and the operations executed are as follows:

1. If the portion of the WRAM buffer which stores edges from the batch is empty or if all previously transferred edges have been processed, new edges are transferred from the batch in the MRAM to the WRAM buffer;

2. For each edge, the tasklet proceeds to color its nodes using the hash function described in the equation 3.1, and the resulting colors are ordered;

3. The tasklet checks whether the edge fits any of the triplets assigned to the DPU. Because both the colors of the triplets and of the edge are ordered, this check is performed efficiently by considering all the possible combinations directly. If the edge does not match any of the triplets assigned to the DPU, it is ignored;

4. If the edge is not ignored, the counter for the total number of edges (t) assigned to the DPU is incremented. If there is sufficient space in the DPU sample for additional edges, the current edge is stored in the portion of the tasklet's WRAM buffer used to temporarily store the edges that need to be transferred to the sample in the MRAM. This portion of the buffer is transferred to the MRAM when it becomes full or when all edges assigned to the tasklet from the current batch have been analyzed.

   If there is not enough space for a new edge in the sample (so $t > M$), the tasklet transfers its buffer of edges designated to be stored in the sample to the MRAM. It then waits for the other tasklets to do the same, ensuring that an edge replacement can only occur once all edges currently in local buffers have been transferred to the MRAM. When the sample is considered complete, it is possible to determine if it is necessary to replace one edge inside the sample through a biased coin toss, as described in section 3.1. If an edge needs

to be replaced, it is chosen uniformly at random inside the sample and replaced with the currently processed edge.

When all these operations are performed by a tasklet, it waits for the others by using a barrier. If the size of the batch sent by the host application is smaller than the maximum allowed, it indicates that the graph has reached its end. If the number of edges in the graph is multiple of the maximum number of edges allowed inside the batch, the second-to-last batch sent will be completely full, and the last one will be empty, containing only the information about the number of edges inside that batch, which will be zero.
When the graph has reached its end, a flag is set to `true` globally within the DPU kernel. This flag serves as an indicator to skip the batch handling process during the next kernel launch and proceed directly to the operations required for counting the triangles.
From now on, all the steps described in this section will also be referred to as *sample creation*.

3.3.3 Counting the triangles

Ordering the edges inside the sample

Once the sample has been created inside each DPU, the next step is to start counting the triangles.
First of all, it is necessary to order the edges within the sample, which significantly speeds up the subsequent triangle counting operation. Since all the edges in the sample belong to a directed graph and the nodes within each edge are ordered such that the first node is always smaller than the second node, an efficient ordering of the edges can be achieved using quicksort. In this case, a generic edge (u, v) is considered smaller than another generic edge (z, w) if either $(u < z)$ or $(u = z$ and $v < w)$. The case where two edges are equal does not need to be handled, as the graph does not contain duplicate edges if input graph requirements are met.
Due to the limited space available inside the program stack, an iterative implementation of quicksort is performed with the pivot chosen at random. One tasklet is responsible for creating partitions of the sample that need to be sorted by each tasklet in order to distribute the sorting operation among all available tasklets. These partitions are created recursively, similarly to how the partitioning step in quicksort works. When a pivot edge is chosen at random, the sample is virtually divided into two. All the edges greater than the pivot are moved into the right side

of the sample, while the edges smaller than the pivot are moved to the left side of the sample. At this point, if the two sections were to be sorted independently, the final array would be fully ordered. Utilizing this concept, partitioning is recursively performed on both the left and the right sections until a pair of indexes marking the section of sample that a tasklet needs to order are created for each tasklet.
At this point, each tasklet performs the iterative quicksort on its section of the sample.

Determining the information relative to the position of the edges inside the sample

To speed up the process of counting the triangles, given the now-ordered sample, one tasklet creates a data structure that enables quick location of the edges related to a unique node. After the ordering step, all the edges starting with the same node are close to each other, with the second node on each edge being ordered. This allows for the creation of a data structure that is used to efficiently find the starting point of the section of edges associated with a particular node. At the same time, it is possible to determine how many edges start with that node, which is also the number of neighbors of the particular node that have a greater indentifier.
Given a node, the information relative to the index inside the sample of the first edge where the node is the first of the two nodes, and the information relative to the number of edges starting with that particular node (so the number of neighbors of that node with a higher identifier) are determined. These information will also be referred to as *unique node information*.
During this operation, the buffer inside the WRAM is used. It is divided into two parts: one is used to store the edges of the sample that need to be traversed, and the other contains all the unique node information already determined that needs to be transferred to the MRAM.
The memory footprint of this data structure limits the maximum allowed sample size. Each edge consists of a pair of 32-bit integers, and each element of the data structure described above occupies 16 bytes. In extreme cases where each edge introduces a new unique node, each edge occupies an overall $8 + 16 = 24$ bytes. Given the amount of memory available inside the MRAM after considering the space occupied by the batch of edges and some other possible variables, the remaining space in bytes should be divided by 24 to determine the maximum allowed number of edges inside the sample to accommodate even the extreme case graphs.

Counting the triangles

After these preliminary steps, the next phase involves counting the triangles formed by the edges

within the sample. The sample is evenly divided among the tasklets. The counting procedure is as follows:

1. For each generic edge (u, v) in the part of the sample assigned to a tasklet, binary search is used to retrieve the unique node information of the two nodes (u and v). This provides information about the starting index in the sample of the section relative to the second node (v) and the number of neighbors with higher identifiers for both nodes (u and v). If the second node (v) has no neighbors, the current edge is ignored.

2. Once both the indexes in the sample of the sections of edges related to the two nodes are determined, the second nodes of each edge within the sections are considered. Considering the generic edge starting with u (u, w) and the generic edge starting with v (v, z), the nodes w and z are compared. Leveraging the fact that the edges are ordered, if $w < z$, the next edge starting with u is considered. Similarly, if $w > z$, the next edge starting with v is considered. If $w = z$, it indicates that nodes u and v share a common neighbor, forming a triangle. In this case, the three nodes (u, v, $w = z$) are colored, and it is checked whether the triangle should be counted by the DPU (as described in section 3.3.2). If the triangle is counted or ignored, the next edges for both u and v are considered. This process continues until either u or v has no more neighbors with higher identifiers. The procedure is then repeated for the next edge in the part of the sample assigned to the tasklet until all edges have been processed.

At the end, when all the tasklets have counted the triangles inside the part of the sample assigned to them, all the counts are summed up and normalized if necessary (see section 3.1).

Chapter 4

Algorithm test results

After developing the algorithm, it is essential to optimize its performance by finding the best parameters and identifying its strengths and weaknesses, with the goal of improving it even further. In this section, the results of various tests are presented.
The tests were conducted on a dedicated machine provided by UPMEM (identified as *upmem-cloud6*), accessed via the SAFARI research group at ETH:

PIM-enabled Memory						DRAM Memory	
DIMM Codename	Number of DIMMs	Ranks/ DIMM	Total DPUs	DPU Frequency	Total Memory	Number of DIMMs	Totale Memory
P21	20	2	2519	350MHz	157.4375 GB	4	256 GB

CPU				
CPU Processor	CPU Frequency	Sockets	Mem. controllers/ Socket	Channels/ Mem.Controllers
Intel Xeon Silver 4215 [31]	2.50 GHz	2	2	3

The PIM-enabled memory DIMMs should have 128 DPUs per DIMM, but in the actual system, only 2519 out of the total 2560 DPUs were available. This does not impact the functionality of the system or the correctness of the test results.
The tests were conducted using the UPMEM SDK version 2023.1.0 [32].
The graphs used in the tests refer to Orkut [33] and Twitter [34], two well-known and widely used social networks. All the duplicate and invalid edges were removed before performing the tests.

Graph	Number of nodes	Number of edges	Number of triangles
Orkut [33]	3,072,441	117,185,083	627,584,181
Twitter [34]	81,306	134,229	13,082,506

Each test focuses on varying a single parameter while keeping the others constant, allowing for a detailed understanding of the impact of each parameter on the algorithm's execution time and efficiency. For a better understanding of the connection between some parameters, future research may explore simultaneous parameter variations, but this would significantly increase the number of tests required.

For each change in parameters, each test was executed 10 times, and the average results were calculated. Due to the fact that all DPUs were synced, the averaged times relate to the slowest DPU in each test iteration.

In the evaluation of the algorithm's performance, three distinct timings were measured, each representing a different aspect of the execution:

- **Setup time:** time required by the host application to allocate the necessary DPUs, load the kernel onto them, and provide them with the initial arguments;
- **Sample Creation Time:** time taken by the host application to read all the edges of the graph and transmit them in batches to the DPUs. During this phase, each edge is analyzed and potentially inserted into the sample within each DPU. The sample creation time concludes when the last DPU has processed the final edge of the graph;
- **Triangle Count Time:** time required to sort the sample, determine the unique node locations, and perform the triangle counting operation by each DPU. The triangle count time concludes when all DPUs have completed their triangle counting tasks and the host application has retrieved and aggregated the results from each DPU.

4.1 Changing the number of colors used

The first parameter under consideration is the number of colors to utilize in relation to the available DPUs. As described in section 3.3.1, to limit the number of checks per edge, it is

advisable to use a number of colors that allows for exactly one triplet to be assigned to each DPU. This configuration optimally utilizes all allocated DPUs, minimizes the number of edges per DPU, and minimizes the number of comparisons required to determine whether an edge should be processed during the sample creation process or a triangle should be counted during the counting process.

The following test results show, given a fixed amount of DPUs, how the performance changes when changing the number of colors used. The following are the specific parameters used for these tests:

Number of DPUs	Number of tasklets	Buffer in WRAM/ tasklet	Maximum edges in sample	Number of colors	Edges in batch	Seed	Dataset
2300 (2519)	1	2048 B	2,621,440	18, 21, 22, 23, 24, 25, 29	131,072	Random	Orkut[33]

The batch size is fixed at 1 MB, which allows for a maximum of 131,072 edges in each batch. With at least 60 MB of free space in the MRAM, the maximum sample size, as discussed in section 3.3.3, is set to $\frac{60\text{ MB}}{24\text{ B}} = 2,621,440$ edges. This ensures that edge replacement within the sample is not required for the tests conducted in this section. These parameters are chosen to try to isolate only the number of colors as the main reason behind changes in computation time.

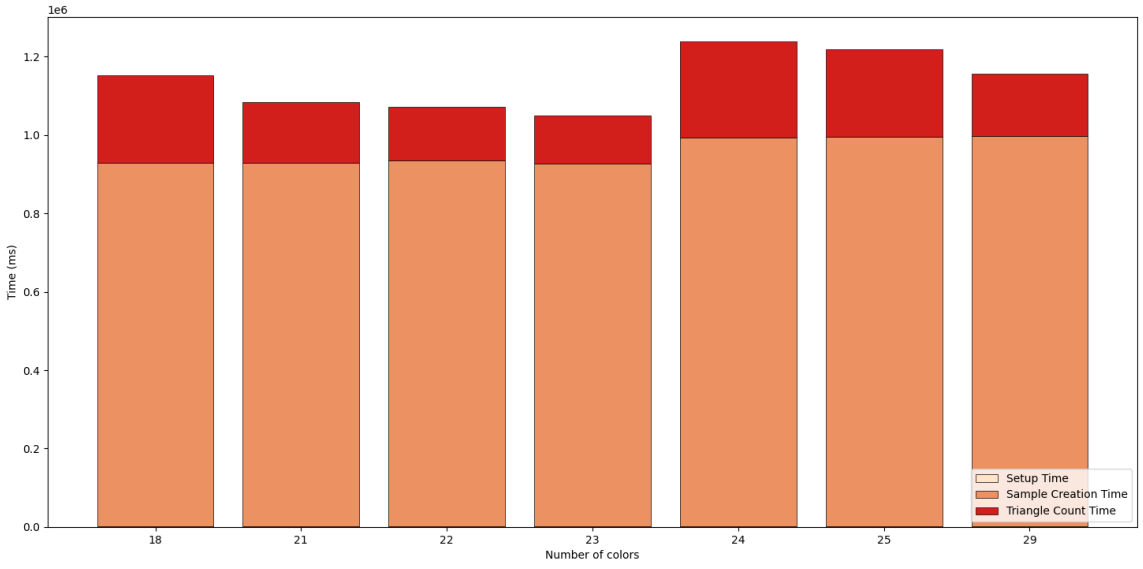


Figure 4.1: Averaged execution time while varying the number of colors used while using 2300 DPUs

With 2300 DPUs allocated for this test, using 23 colors ensures that each DPU is assigned

one triplet; using fewer than 23 colors leads to assigning one triplet to only some of the DPUs, $\binom{C+2}{3}$ to be precise, and none to others, resulting in wasted DPUs. Referring to figure 4.1, the results show that the time required to create the sample is similar when using different numbers of colors smaller or equal to 23, because, in all these cases, the number of triplets needed to be checked to determine if each edge should be considered or not is one in all the working DPUs. The time needed to count the triangles decreases as the number of colors increases. This is due to the fact that DPUs with a triplet with three different colors, which have the maximum amount of edges inside their sample, around $6 \cdot \frac{E}{C^2}$ edges, dictate the running time (considering that the maximum sample size in these tests is enough to contain all the edges assigned to the DPU). For this reason, when using fewer colors, there are in general more edges inside each sample, so the time it takes to order the edges and count the triangles is greater.
When using more than 23 colors, some DPUs will be assigned more than one triplet. In particular, with 29 colors, 4495 triplets are generated, making nearly all DPUs have two triplets assigned to them. The time required to create the sample increases compared to previous cases due to the larger number of triplets that need to be checked when deciding if an edge should be considered or not. Since the total running time is determined by the slowest DPU (due to synchronization among all DPUs), the time to create the sample remains similar whether only a few DPUs or almost all DPUs have two triplets. The time required to count the triangles also increases because, although a larger number of colors reduces the number of edges per triplet, it is likely that a DPU has two triplets with three different colors. This results in an estimated number of edges to handle of $12 \cdot \frac{E}{C^2}$. The doubling of edges is generally not offset by the increase in the number of colors used.

Another question that may arise is whether it is better to utilize all available DPUs with more colors or not use them at all when there are more DPUs in the system than the number required to assign exactly one triplet to each DPU.
Referring to figure 4.2, as expected from the previous tests, using a number of colors that result in assigning more than one triplet to at least some DPUs leads to worse results. In addition, although the difference is small, it can be observed that allocating more DPUs than strictly required also leads to marginally worse performance. This can be attributed to the overhead of broadcasting the batch of edges to a larger number of DPUs.
Based on the results of these tests, for future tests, the configuration of 2300 DPUs with 23

colors will be used.

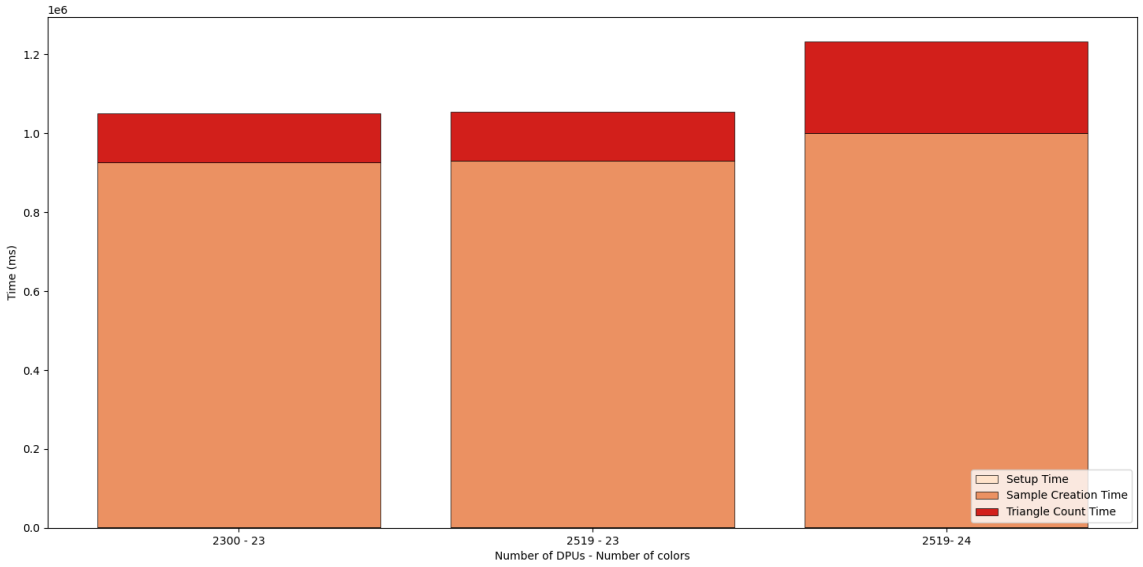


Figure 4.2: Averaged execution time while varying the number of colors used while using 2519 DPUs

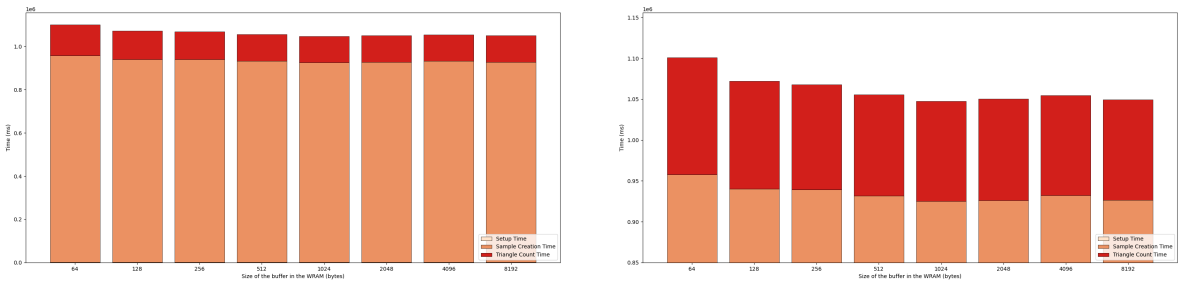
4.2 Changing the size of the buffer inside the WRAM

As described in section 3.3.1, to speed up the access time to data stored in the MRAM, a buffer within the WRAM is utilized by every tasklet. This buffer allows for temporary storage of data fetched from the MRAM or that needs to be sent to the MRAM. Using this buffer leads to using larger DMA transfers, improving access times. The following are the specific parameters used for these tests:

Number of DPUs	Number of tasklets	Buffer in WRAM/ tasklet	Maximum edges in sample	Number of colors	Edges in batch	Seed	Dataset
2300	1	64 - 128 - 256 - 512 - 1024 - 2048 - 4096 - 8192 B	2,621,440	23	131,072	Random	Orkut[33]

As described above, the batch size is set to 1 MB, and the space available inside the MRAM is considered to be 60 MB.

Figure 4.3 shows that, as expected, the execution time generally decreases with an increase in the WRAM buffer size. This is caused by the fact that larger DMA transfers allow for fewer accesses to the MRAM, resulting in faster computation times for both the sample creation phase



(a) (b) Detail

Figure 4.3: Averaged execution time while varying the size of the WRAM buffer for each tasklet

and the triangle counting phase. The larger the buffer size within the WRAM, the less frequently it becomes necessary to access the MRAM. However, this is true until the buffer size is 2048 bytes, or, as described previously, until the transfers have a maximum size of 1024 bytes, due to the fact that the entire buffer is divided into two sections to store different types of data at various stages of the algorithm. The decrease in performance observed with buffer sizes of 4096 bytes and 8192 bytes may be attributed to the increased latency associated with larger transfers between the MRAM and WRAM, as documented in [27]. Further tests are required to determine if this is the actual cause of the decrease in performance or if it is caused by the algorithm implementation.

Given that the results when using a buffer in the WRAM of 1024 bytes and 2048 bytes are virtually the same (the difference may be caused by chance and by the limited number of tests performed), for consistency with the first tests, a WRAM buffer of 2048 bytes will be used going forward.

4.3 Changing the number of tasklets

One way to achieve better performance when using the DPUs is to use more than just one tasklet, trying to fully saturate the pipeline. The following are the specific parameters used for these tests:

Number of DPUs	Number of tasklets	Buffer in WRAM/ tasklet	Maximum edges in sample	Number of colors	Edges in batch	Seed	Dataset
2300	From 1 to 21	2048 B	2,621,440	23	131,072	Random	Orkut[33]

Although a DPU can utilize up to 24 tasklets, due to the fact that each tasklet has its own buffer of 2048 bytes inside the WRAM, using more than 21 tasklets would result in a crash due to the insufficient amount of memory available inside the WRAM.
As described above, the batch size is set to 1 MB, and the space available inside the MRAM is considered to be 60 MB.

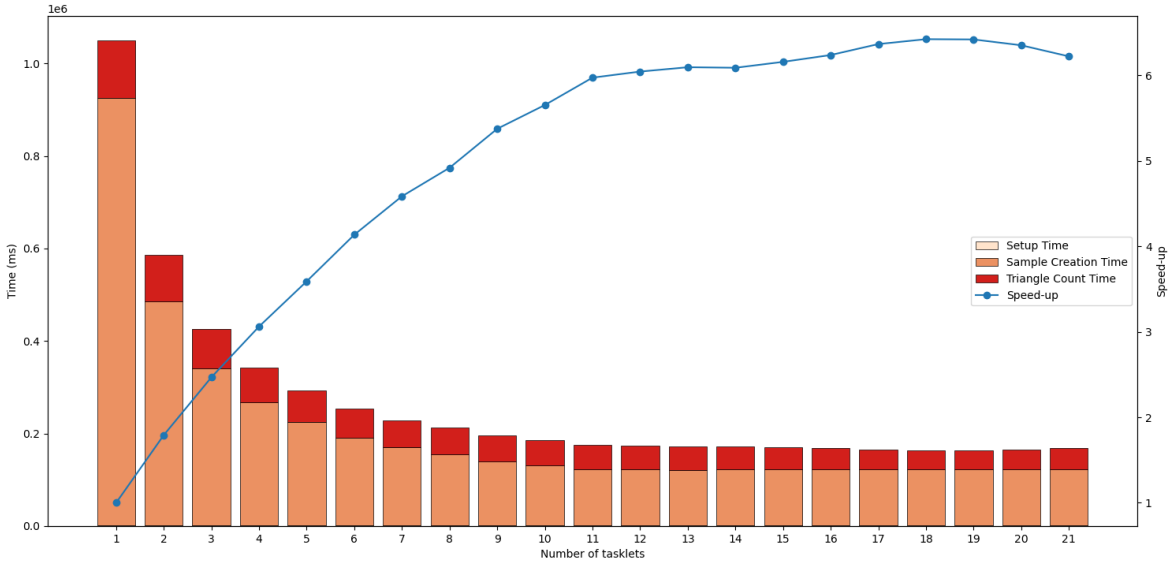
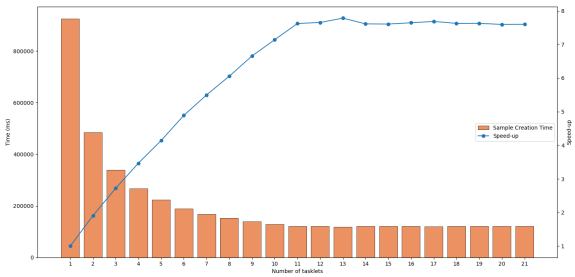
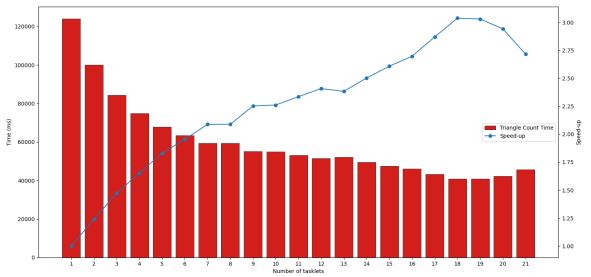


Figure 4.4: Averaged execution time and relative speed-up while varying the number of tasklets used



(a) Sample creation



(b) Triangle count

Figure 4.5: Averaged execution time and relative speed-up while varying the number of tasklets used

Referring to figures 4.4 and 4.5, the results of the tests indicate that using multiple tasklets can significantly improve the performance of the algorithm, especially in the sample creation phase. The speed-up gradually increases as the number of tasklets increases, reaching a plateau

when using 11 tasklets, beyond which the performance gains become smaller. During the sample creation phase, the speed-up initially shows a linear trend, with a maximum speed-up of approximately 7.62 compared to the time taken when using only one tasklet, achieved when using 11 tasklets.
In the triangle counting phase, the speed-up is not as significant, reaching a maximum of only 3.04 when using 18 tasklets. This is probably caused by the suboptimal parallelization in the counting triangle phase implementation, specifically in the division of the sample for ordering, because the current implementation of the algorithm may lead to an unequal division in terms of the number of edges to be sorted between the tasklets. For this reason, the benefits increase well past the 11 tasklet mark, maybe due to the better chance of achieving equal division. When using even more than 18 tasklets, the performance drops because the increased overhead outweighs the benefits of having more threads.
Although the performance increase goes well past 11 tasklets for the triangle counting phase, due to the small speed-up that that phase provides and the fact that the sample creation phase accounts for the majority of the execution time, the total speed-up of the algorithm execution is dominated by the sample creation phase's speed-up. As a result, the performance increases almost linearly until 11 tasklets are used, with only a small improvement when using more tasklets.
In general, it is possible to see that using more tasklets leads to an appreciable performance increase when considering the algorithm execution as a whole.
Based on the results of these tests, for future tests, 18 tasklets will be used, which leads to a speed-up of around 6.42 compared to the time taken when using only one tasklet.

4.4 Change in number of edges inside the batch

The previous test results have shown that a significant part of the total computation time is determined by the operations regarding the creation of the sample. One way to decrease this time would be to increase the uninterrupted execution time of the DPUs' kernel. As stated in [27, 25], it is better to execute sections of the kernel that are as long as possible. For this reason, transferring a larger number of edges from the host application to the DPUs each time should decrease the computation time.

The following are the specific parameters used for these tests:

Number of DPUs	Number of tasklets	Buffer in WRAM/ tasklet	Maximum edges in sample	Number of colors	Edges in batch	Seed	Dataset
2300	18	2048 B	1,572,864	23	65,536 - 131,072 - 262,144 - 524,288 - 1,048,576 - 2,097,152 - 3,145,728	Random	Orkut[33]

With the increase in batch size, it becomes necessary to adjust the maximum number of edges that can be stored within the sample due to the limited space available inside the MRAM. Considering that the biggest batch size tested is 24 MB, it is possible to assume that at least 36 MB will be free inside the MRAM. With those 36 MB, it is possible to store $\frac{36MB}{24B} = 1,572,864$ edges inside the sample, which is bigger than the maximum expected amount of edges per DPU, which is $6 \cdot \frac{E}{C^2} = 6 \cdot \frac{117,185,083}{23^2} \approx 1,329,131$, also considering the possibility that this theoretical limit is surpassed.

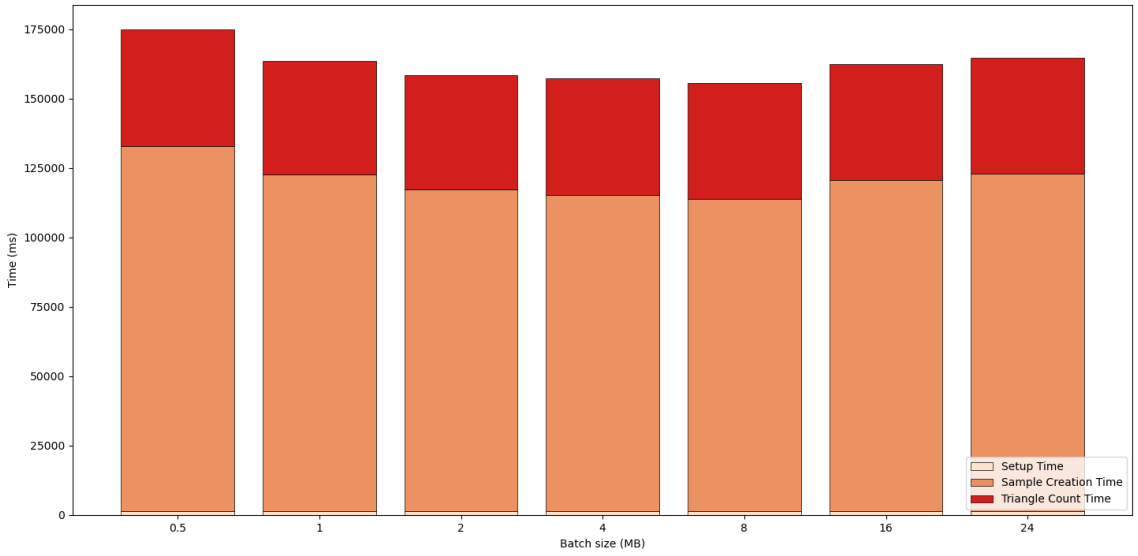


Figure 4.6: Averaged execution time while varying the number of edges inside the batch sent from the host application to the DPUs

Referring to figure 4.6, it is clearly visible that, while the time necessary to count the triangles remains almost the same for all the values of the batch size, the time taken to create the sample decreases with larger batch sizes, up to a certain point. This improvement can be attributed to the longer uninterrupted execution of the DPUs' kernel, as less frequent data transfers from the host application result in reduced overhead.

While it may be reasonable to think that larger batch sizes would always yield better performance, the test results show otherwise, with a significant increase in computational time when using batches of 16 or 24 MB. This is likely caused by the amount of cache available to the host CPU. As it is possible to see in the data sheet of the host CPU [31], only 11 MB of cache are available. This means that it is possible to efficiently store a maximum of 11 MB of data for the transfers to the DPUs. The broadcast communication used to transfer data from the host application to the DPUs takes advantage of the temporal locality of the data inside the cache hierarchy to increase the transfer bandwidth [27]. When the amount of data to transfer to the DPUs is larger than the amount of space available inside the host CPU's cache, multiple transfers between the RAM and CPU cache are required for each broadcast to the DPUs, making the process less efficient. Further experimentation may be necessary to determine the optimal batch size that maximizes performance given the host CPU's cache size.
For the following tests, a batch of 8 MB will be used. There is a maximum amount of free memory inside the MRAM used to store the data regarding the sample of at least 54 MB, which is enough to store 2,359,296 edges and enough to guarantee that no edge replacement will be necessary while using the Orkut graph, 2300 DPUs, and 23 colors.

4.5 Change in maximum edges allowed inside the sample

One characteristic of the algorithm presented in this thesis is that it is possible to have both the exact number of triangles inside the graph, if the graph is not too large, or an extimation of that number. To test the change in performance and the accuracy of the resulting number of triangles, different values for the maximum number of edges allowed inside the sample of the DPUs are tested. Considering the theoretical maximum number of edges that a DPU could have to handle while using 23 colors, which is $6 \cdot \frac{E}{C^2} = 6 \cdot \frac{117{,}185{,}083}{23^2} \approx 1,329,131$, the values tested represent, other than the control, around 75%, 50%, 25%, 10%, 5% and 1% of that value. The following are the specific parameters used for these tests:

Number of DPUs	Number of tasklets	Buffer in WRAM/ tasklet	Maximum edges in sample	Number of colors	Edges in batch	Seed	Dataset
2300	18	2048 B	2,359,296 - 975,000 - 650,000 - 325,000 - 130,000 - 65,000 - 13,000	23	1,048,576	Random	Orkut[33]

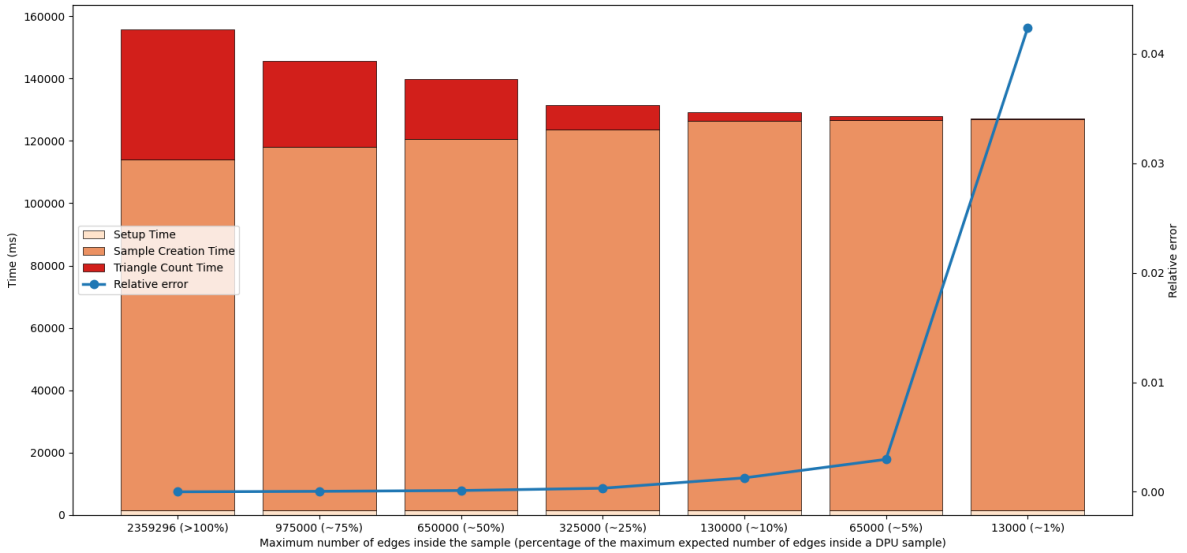


Figure 4.7: Averaged execution time and relative error of the resulting triangle counted while varying the maximum number of edges allowed inside the sample of each DPU

Referring to figure 4.7, the time required to count the triangles decreases as the maximum sample size decreases, while the time needed to create the sample increases. The increase in sample creation time is primarily due to the higher number of edge replacements that occur when the number of edges exceeds the maximum allowed within a DPU's sample. Each edge replacement inside the sample, considering that a random index inside the sample is chosen, cannot take advantage of the WRAM buffer used previously. For this reason, all the needed accesses to the MRAM happen individually, increasing the total time needed for this operation. The time needed to count the triangles decreases because the number of edges to sort and traverse for triangle counting decreases proportionally to the maximum allowed number of edges within the sample.

The relative error is calculated by averaging the relative error from the 10 tests conducted for each sample size. It is possible to see that the error increases more rapidly as fewer edges are allowed inside the sample. This is primarily due to the fact that initially only the DPUs with a triplet of three different colors are affected. Only when the maximum sample size reaches a value lower than $3 \cdot \frac{E}{C^2} = 3 \cdot \frac{117185083}{23^2} \approx 664566$ also the DPUs with a triplet with two different colors affected, and, lastly, when the values become lower than $\frac{E}{C^2} = \frac{117185083}{23^2} \approx 221522$ all the DPUs are affected. When the maximum allowed number of edges inside the sample becomes too low, even the statistical correction cannot provide an accurate enough result, although the

relative error remains lower than 5% in the configuration tested.

It is possible to assess that the significant decrease in the operations regarding the triangle counting makes it justifiable to have an approximation in the triangle count. The substantial time needed to create the sample, however, diminishes the benefits of a non-accurate result.

4.6 Change in the number of DPUs used

The following tests have the goal to see the change in computational time and accuracy of the resulting triangles counted when using a different amount of DPUs. As found out in the first round of tests (see test 4.1), the optimal number of DPUs given C colors is the amount that allows for exactly one triplet per DPU. In the following tests, the number of DPUs allocated corresponds to the number of colors that allow for exactly one triplet per DPU. The following are the specific parameters used for these tests:

Number of DPUs	Number of tasklets	Buffer in WRAM/ tasklet	Maximum edges in sample	Number of colors	Edges in batch	Seed	Dataset
286 - 455 - 680 - 969 - 1330 - 1771 - 2300	18	2048 B	2,359,296	11 - 13 - 15 - 17 - 19 - 21 - 23	1,048,576	Random	Orkut[33]

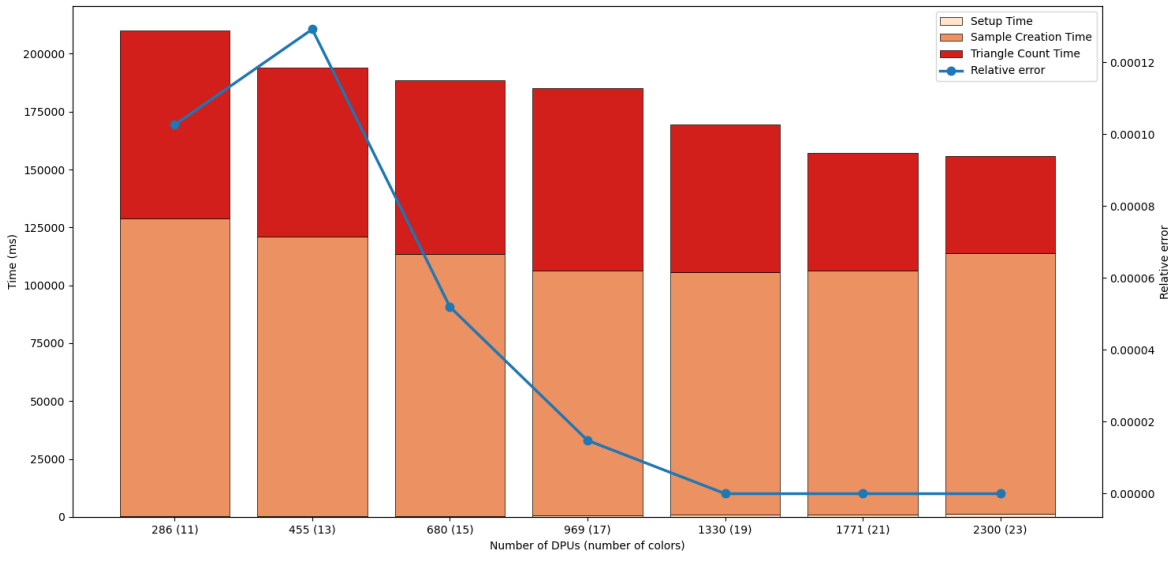


Figure 4.8: Averaged execution time and relative error of the resulting triangle counted while varying the number of DPUs used (Orkut graph)

Referring to figure 4.8, when using a small number of colors, there is a possibility of filling

up the available space within the sample of edges in each DPU. This can lead to a loss of accuracy in the triangle count estimation, although the impact is relatively small for the graph used for the tests, with a relative error much smaller than 1%.
It is possible to see that the execution time decreases with an increase in the number of DPUs used. One reason for this behavior is that, when the number of colors is insufficient and the sample is filled up, edge replacements become necessary, leading to longer sample creation times. This phenomenon is attenuated by the fact that the number of transfers required to send each batch of edges from the host application to the DPUs decreases, leaving fewer DPUs that need to receive the batches of edges. This overhead in sending a batch of edges to more DPUs is visible when using 19, 21, and 23 colors. In these cases, the sample of edges inside the DPUs is never filled, and no edge replacement happens. For this reason, the main cause of an increase in sample creation time is the number of DPUs that need to receive the batch of edges from the host application. In addition, the increase in DPUs used also increases the time it takes the host application for the initial setup, although the setup time is negligible considering the total computation time.
The main advantage of using more colors lies in the reduced number of edges per DPU, minimizing the likelihood of filling up the sample. With fewer colors, the sorting and triangle counting operations must be performed using samples that may be full for some DPUs, resulting in longer computation times. By increasing the number of colors, the triangle counting time decreases due to the smaller number of edges in the DPUs' samples.
In the case of larger graphs, it is generally more beneficial to utilize a higher number of DPUs and colors to avoid filling up the sample and to reduce the number of edges considered during the triangle counting phase by each DPU. However, with smaller graphs, this may not be the case.

In order to gain insights into the impact of graph size on the overall execution time, a smaller graph is analyzed for the following tests, where the time needed for the setup of the DPUs and the overhead associated with sending edge batches to multiple DPUs may become more significant relative to the overall time. For the following tests, a graph related to Twitter [34] is used.

The following are the specific parameters used for these tests:

Number of DPUs	Number of tasklets	Buffer in WRAM/ tasklet	Maximum edges in sample	Number of colors	Edges in batch	Seed	Dataset
286 - 455 - 680 - 969 - 1330 - 1771 - 2300	18	2048 B	2,359,296	11 - 13 - 15 - 17 - 19 - 21 - 23	1,048,576	Random	Twitter[34]

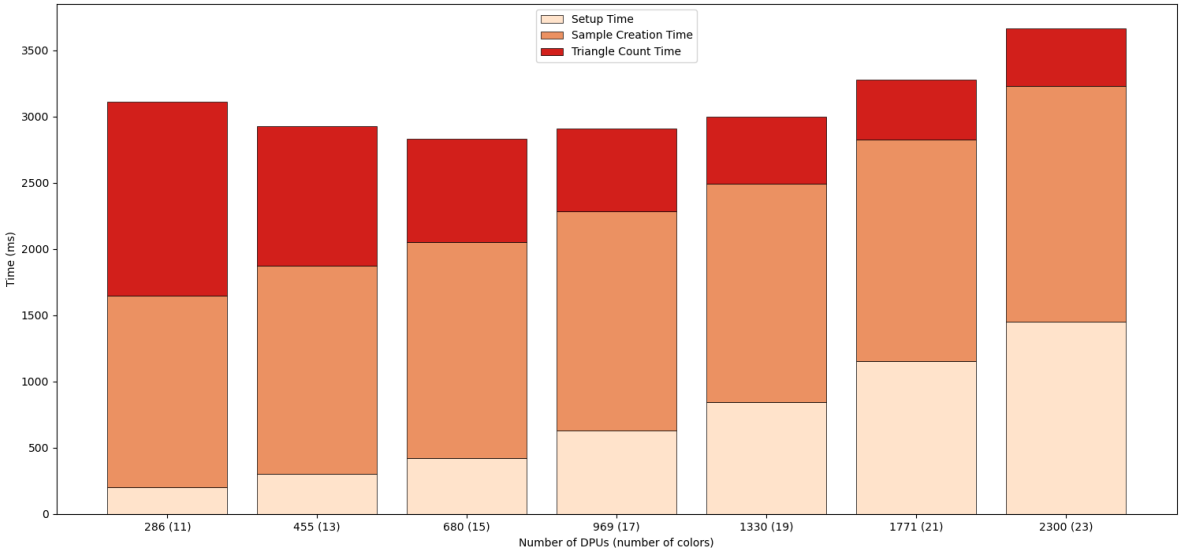


Figure 4.9: Averaged execution time while varying the number of DPUs used (Twitter graph)

Referring to figure 4.9, the results clearly demonstrate that using more DPUs does not always yield better performance. As always, the time required for triangle counting operations decreases with an increase in the number of colors used due to the lower number of edges per DPU. However, the overall benefits of using more DPUs diminish as the number of DPUs increases.

The time needed for the sample creation phase increases slightly with an increase in DPUs due to the additional time needed to distribute each edge batch to a larger number of DPUs.

The most significant difference arises from the setup time. As the number of DPUs increases, the time required to allocate the DPUs, load the kernel into the DPUs, and send starting arguments to each DPU also increases. With smaller graphs, this initial overhead becomes very significant, making the benefits of having fewer edges to consider in the triangle counting phase per DPU not an overall advantage. For these reasons, for the tested configurations and the specific characteristics of the graph, it is more favorable to use 15 colors and 680 DPUs.

4.7 Regular CPU implementation

To state the effectiveness of using a system with PIM-enabled memory, it is possible to compare the results to an implementation of the same algorithm that uses only a regular CPU in a system configured, as described previously, with the possibility to use up to 32 cores 4.
The algorithm implementation is adapted from the version that uses the DPUs. There is a host process that starts different threads that will occupy a core of the system each. Every thread reads autonomously from the file containing the edges of the graph without needing to have them sent from the host process. The edges relevant to each thread are stored in a sample, which, due to the size of the DRAM memory available in the system and the small number of threads used, can accommodate a significant number of edges. Consequently, the likelihood of edge replacement within the sample is reduced. This could lead to an unfair advantage for the CPU implementation when considering graphs whose size leads to the necessity of edge replacement in the DPU implementation but not in the regular CPU one. However, it's important to note that such scenarios do not occur in the tests discussed in this section.
The algorithm implementation for the CPU version is not as refined and optimized as the PIM implementation. However, even with this caveat, it is possible to qualitatively observe the advantages of using a PIM-enabled system, particularly considering that the algorithm is specifically designed to leverage the benefits of the PIM architecture. The following are the specific parameters used for these tests:

Number of Threads	Number of colors	Seed	Dataset
1, 4, 10, 20, 35	1, 2, 3, 4, 5	Random	Orkut[33]

The reference time for the execution using DPUs is $155,788.51$ ms while using the following parameters:

Number of DPUs	Number of tasklets	Buffer in WRAM/ tasklet	Maximum edges in sample	Number of colors	Edges in batch	Seed	Dataset
2300	18	2048 B	2,359,296	23	1,048,576	Random	Orkut[33]

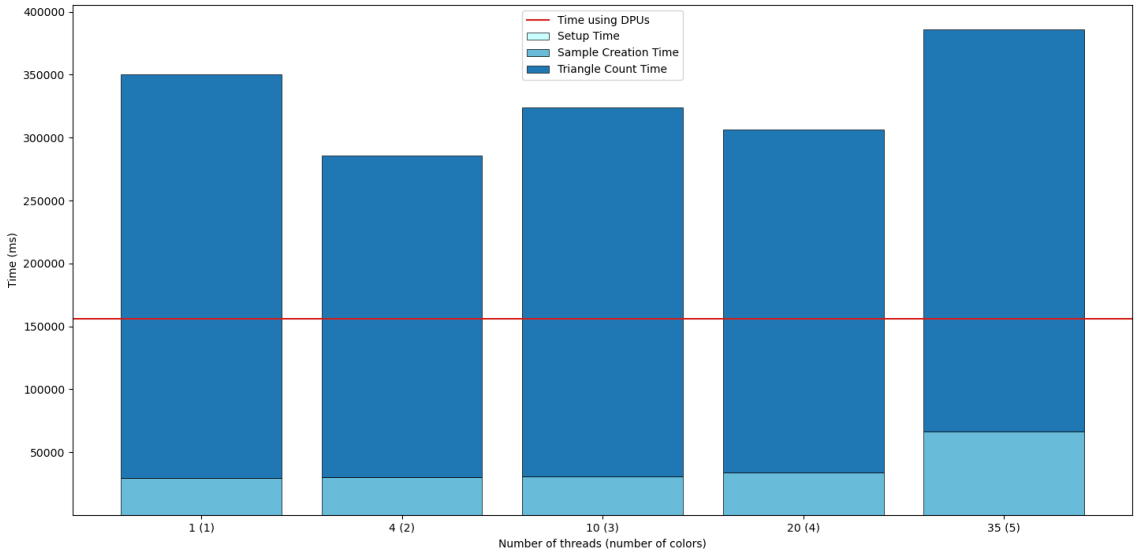


Figure 4.10: Averaged execution time while varying the number of threads used compared to the execution time while using DPUs (Orkut graph)

Referring to figure 4.10, it is possible to see that the time needed to count the triangles in the graph using the CPU is much higher compared to the time needed while using the DPUs. While using the CPU, however, it is possible to see that the time required to create the sample is relatively small in proportion to the overall execution time. This is because each thread can directly access the graph data stored in memory without relying on a host process to transmit the data. In addition, since the threads are initiated simultaneously and considering that all the threads need to access the same edges almost at the same time, the data relative to the edges of the graph at any point in time is likely stored in the cache, and it is possible for the threads to take advantage of the cache's temporal locality. When the number of threads exceeds the available CPU cores, however, the sample creation time increases due to the necessity of context switching, where some threads are paused to allow others to execute their operations.
The dominant factor influencing the total execution time is the time required for counting the triangles. The algorithm proposed in this thesis, which aims to leverage the fast access of the DPUs to their memory banks, involves frequent retrieval of edges from the sample during the triangle counting phase. When using a CPU, accessing edges located in different memory regions becomes a bottleneck, as minimal computation is performed for each retrieved edge. This results in a continuous need to access new edges, making memory bandwidth and latency the limiting factors during this phase of the algorithm. From the limited testing performed, it is not

possible to see a clear trend in execution time when varying the number of threads. This may be attributed to the diverse utilization of the cache during the triangle counting process. Further testing may be needed to understand the behavior of this algorithm in the CPU implementation, maybe considering also a set of tests using the same number of samples as used in the tests of the DPU implementation.

While with bigger graphs the cache size is the limiting factor in the CPU implementation, with smaller graphs it may be possible to see that, with enough space inside the cache to store almost all the edges needed by the threads, it is possible to have faster execution times. In addition, the CPU implementation benefits from reduced setup time as there is no need to allocate DPUs or load kernels, and the absence of data transfer overhead may contribute to improved performance.
To assess the behavior of the CPU implementation while using a smaller graph, a graph related to Twitter [34] is used. The following are the specific parameters used for these tests:

Number of Threads	Number of colors	Seed	Dataset
1, 4, 10, 20, 35	1, 2, 3, 4, 5	Random	Twitter[34]

The reference time for the execution using DPUs is $2,828.72$ ms while using the following parameters:

Number of DPUs	Number of tasklets	Buffer in WRAM/ tasklet	Maximum edges in sample	Number of colors	Edges in batch	Seed	Dataset
680	18	2048 B	2,359,296	15	1,048,576	Random	Orkut[33]

As observed while analyzing the Orkut graph, referring to figure 4.11, most of the execution time is taken by counting the triangles. However, while using fewer threads, it is possible to see that the ability to store most of the needed data inside the cache significantly speeds up the execution time. When the cache is shared among many threads, it becomes more difficult to take advantage of the temporal locality, considering that each thread loads into the cache the edges that it needs, replacing the edges that other threads may need in the future.

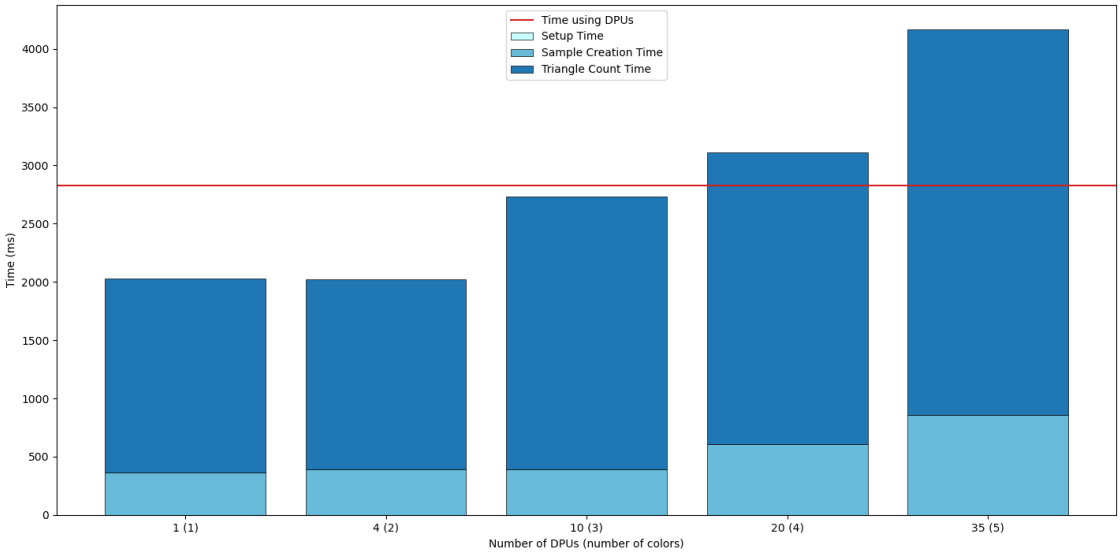


Figure 4.11: Averaged execution time while varying the number of threads used compared to the execution time while using DPUs (Twitter graph)

Based on the analysis conducted, it can be concluded that if the CPU cache can be fully utilized, taking advantage of its fast access times, the CPU implementation tends to be faster due to the reduced overhead in creating the samples and setting up the DPUs. However, as the size of the analyzed graphs increases, the cache becomes too small for the needs of the execution, and the performance drops, limited by the memory bandwidth and latency.

Chapter 5

Conclusions

This thesis proposes an implementation of a triangle counting algorithm specifically designed for an UPMEM PIM-enabled system. The algorithm's operations have been thoroughly explained, highlighting the advantages that arise from using a PIM architecture. Through various tests, optimal parameters have been determined to achieve the best performance. Furthermore, the execution time of the algorithm run on the PIM-enabled system has been compared to an implementation of the same algorithm on a traditional architecture, illustrating the benefits derived from the PIM implementation.
The algorithm presented in this thesis serves as a demonstration of how a PIM-enabled system can significantly accelerate workloads that are slowed down by the limitations of DRAM memory. However, there is still room for improvement in efficiently utilizing the available resources, and further research is required to improve the algorithm's performance.

5.1 Future work

This initial implementation of the algorithm presents several opportunities for improvement.

5.1.1 Improving the sample creation time

As the test results clearly show, a large part of the computation time is needed to create the sample inside the DPUs. The main problem lies in the fact that all the edges are transferred from the host application to all the DPUs, where every edge is considered even if it is not compatible with the triplets assigned to a specific DPU. While using a sequential reader [35] could improve

the edge retrieval process from the batch in the MRAM, it does not address the underlying issue of transferring a large volume of data from the host application to the DPUs.

It would be ideal to find an efficient way to send to the DPUs a smaller number of edges to analyze, guaranteeing at the same time that a specific DPU would always have access to all the edges it should handle. One solution could involve creating smaller batches of edges specific to each DPU within the host application. As seen in the tests performed while using the CPU implementation of the algorithm (referring to the test 4.7), a CPU is fast when retrieving and considering the edges of a graph stored in a file. Leveraging the faster processing capabilities of the CPU, the host application, utilizing multiple threads if necessary, could generate batches tailored to each DPU, containing only the edges that a specific DPU needs to consider. By doing so, the amount of work required by the DPUs during the sample creation phase could be significantly reduced, as they would only need to perform operations if edge replacements within the sample were necessary. Otherwise, the host application could directly send the batches to the appropriate location within the MRAM, eliminating the need for any additional processing inside the DPUs.

This approach, however, may present some problems. First of all, the data transfers would need to be sequential, considering that each batch would be specific to a single DPU, and a broadcast data transfer would not be possible. Furthermore, to have a good performing transfer of data from the host application to the DPUs, the data should fit inside the CPU cache. Considering the large number of DPUs that could be involved, the batch size for each DPU would be quite small given the limited size of the CPU's cache. This would lead to many small and frequent transfers from the host application to the DPUs, which would result in poor performance. A partial solution to this problem would be to use batch sizes proportional to the number of edges statistically received by a DPU given the triplet assigned to it, allowing for an equal number of transfers needed for each DPU. Another consideration is that implementing this improvement may not fully utilize the DPUs while the CPU prepares the batches, as they would remain idle during this phase, potentially leading to underutilization of resources.

If this improvement was to be implemented, it would be necessary to find the optimal batch size by considering the trade-off between larger batches that use the CPU cache less efficiently but require fewer transfers and smaller batches that fit within the cache but necessitate more frequent transfers. As seen in the test where different sizes of batches have been used (referring to the test 4.4), it is possible to expect that using more space for the different batches than the

space available inside the cache would lead to worse results. However, not knowing how the system would perform with this possible improvement implemented, further experimentation and testing are necessary to determine the optimal batch size and address potential performance trade-offs.

An alternative approach involves a balanced distribution of work between the host application and the DPUs by dividing the sample creation process more equally.
In the current implementation, most of the work while creating the sample is performed by the DPUs, and in the possible improvement proposed above, most of the work while creating the sample would be performed by the host CPU, but it may be possible to divide the work more equally between the CPU and the DPUs. As in the above-proposed improvement, the CPU, utilizing multiple threads if necessary, could create specific batches of edges, which would be sent to multiple DPUs. The batches could be organized based on certain characteristics, such as the color of the nodes, making it possible to create batches with edges that are colored in some specific way and that are suitable for more than one DPU. Then, each DPU would receive only the batch type relevant to the type of edges it needs to handle. It would therefore be necessary to find a way to efficiently divide the edges into these specific batches in such a way that each DPU could receive only one type of batch while guaranteeing that each DPU has access to all the necessary edges.
Similar to the previous improvement, the limited cache size could be a possible problem, and experimentation would be required to determine the optimal batch size and the best strategy to form the aforementioned types of batches. However, one advantage of this approach is that some computation could be offloaded to the DPUs, allowing the CPU to asynchronously create the batches. This asynchronous operation enables different stages of sample creation to occur simultaneously, and the goal of exploring this possible improvement would be to synchronize the creation of new batches with the handling of previous batches by the DPUs.

5.1.2 Improving the triangle counting time

The operations related to the triangle counting phase could greatly benefit from various improvements, for example, using the buffers in the WRAM more extensively and more efficiently. As seen in the test performed while changing the number of tasklets used (referring to the test

4.3), it would also be necessary to better scale the performance with the number of tasklets used.

One area for improvement lies in the quicksort implementation. Currently, the sample is divided somewhat randomly among the tasklets, which could result in significant variations in the number of edges that each tasklet needs to sort, possibly leading to some tasklets completing their sorting phase much sooner than others. It would also be possible to greatly improve the performance of the quicksort using the WRAM buffer. The buffer could be used during the partitioning phase, where multiple single accesses to the MRAM could be replaced by less frequent ones. In addition, it would be ideal, if the remaining edges to sort fit the WRAM buffer, to sort them directly inside the WRAM, copying the sorted result into the MRAM.

Another aspect to consider is that in the current implementation, finding the unique node locations is performed by a single tasklet. However, dividing this task among multiple tasklets could potentially enhance performance. Furthermore, given that in this phase the sample is read edge by edge, employing a sequential reader, which is faster than manual transfers from the MRAM to the WRAM, would be ideal.

Finally, even the triangle counting itself could be improved by utilizing a sequential reader instead of manual transfers of edges from the MRAM to the WRAM, and it may be possible to find a better way to divide the sample between the different tasklets during this phase.

Further experimentation and testing are necessary to determine the possible improvements that could be made to speed up the triangle counting operations.

5.1.3 Parallel execution on different systems

Thanks to the way triplets are divided among DPUs, it would be possible to divide the workload even across different PIM-enabled systems. By configuring multiple systems with a setup similar to the one used for the tests presented previously (see chapter 4), it would be feasible to divide the triangle counting task among them. This could be achieved by employing a supervisor that executes the algorithm on the systems and adds up the results.

Given that a triplet is assigned to a DPU according to its ID, it would be feasible to use the DPUs of different systems with incremental IDs. For instance, if two machines were available, each equipped with 2519 DPUs, the first machine could have DPUs with IDs ranging from 0 to 2518, while the second machine would have DPUs with IDs from 2519 to 5037. This approach

would enable the utilization of a greater number of colors, effectively reducing the number of edges that each DPU would need to consider. This reduction in edges would lead to a decrease in triangle counting time, as each DPU would have fewer edges to evaluate.

Both systems would color the nodes with all the available colors, but the host application of a specific system could directly ignore any edges that are not handled by any DPU within its system.

However, scaling would not be ideal. For example, if a total of 5038 DPUs would be distributed between two systems, it would be possible to utilize 30 colors, resulting in 4960 triplets and an equivalent number of DPUs actually utilized. Considering the way edges are distributed among the DPUs, when comparing 30 colors to the original 23 colors, the theoretical speed-up for the triangle counting phase would be $\frac{30^2}{23^2} \approx 1.7$ compared to a single system. Similarly, with three systems and a total of 7557 DPUs, utilizing 34 colors and 7140 triplets, would yield a theoretical speed-up for the triangle counting phase of approximately $\frac{34^2}{23^2} \approx 2.19$ compared to a single system.

The scaling limitations are due to the nature of the binomial coefficient operator, resulting in diminishing returns as the number of systems and DPUs increases. Additionally, there would be some overhead involved due to the necessity of a coordinator to initiate the systems and collect the results.

Further experimentation and testing are necessary to determine the possible improvements that could come from using multiple PIM-enabled systems.

Bibliography

[1] Danijela Efnusheva, Ana Cholakoska, and Aristotel Tentov. "A Survey of Different Approaches for Overcoming the Processor - Memory Bottleneck". In: *International Journal of Information Technology and Computer Science* 9 (2017-04), p. 151. DOI: 10.5121/ijcsit.2017.9214.

[2] Amirali Boroumand 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. Williamsburg, VA, USA: Association for Computing Machinery, 2018, pp. 316–331. ISBN: 9781450349116. DOI: 10.1145/3173162.3173177. URL: https://doi.org/10.1145/3173162.3173177.

[3] Michael Ferdman et al. "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware". In: 2012. URL: http://infoscience.epfl.ch/record/173764.

[4] M. Gokhale, B. Holmes, and K. Iobst. "Processing in memory: the Terasys massively parallel PIM array". In: *Computer* 28.4 (1995), pp. 23–31. DOI: 10.1109/2.375174.

[5] *UPMEM Technology*. URL: https://www.upmem.com/technology/.

[6] Onur Mutlu et al. "Processing data where it makes sense: Enabling in-memory computation". In: *Microprocessors and Microsystems* 67 (2019), pp. 28–41. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2019.01.009. URL: https://www.sciencedirect.com/science/article/pii/S0141933118302291.

[7] S. Ghose et al. "Processing-in-memory: A workload-driven perspective". In: *IBM Journal of Research and Development* 63.6 (2019), 3:1–3:19. DOI: 10.1147/JRD.2019.2934048.

[8] Saugata Ghose et al. “Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions”. In: *CoRR* abs/1802.00320 (2018). arXiv: 1802.00320. URL: http://arxiv.org/abs/1802.00320.

[9] Junwhan Ahn et al. “A scalable processing-in-memory accelerator for parallel graph processing”. In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 2015, pp. 105–117. DOI: 10.1145/2749469.2750386.

[10] Lifeng Nai et al. “GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks”. In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017, pp. 457–468. DOI: 10.1109/HPCA.2017.54.

[11] Jonathan W. Berry et al. “Tolerating the community detection resolution limit with edge weighting”. In: *Physical Review E* 83.5 (2011-05). DOI: 10.1103/physreve.83.056119. URL: https://doi.org/10.1103%5C%2Fphysreve.83.056119.

[12] Jean-Pierre Eckmann and Elisha Moses. “Curvature of co-links uncovers hidden thematic layers in the World Wide Web”. In: *Proceedings of the National Academy of Sciences* 99.9 (2002), pp. 5825–5829. DOI: 10.1073/pnas.032093399. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.032093399. URL: https://www.pnas.org/doi/abs/10.1073/pnas.032093399.

[13] Zhi Yang et al. “Uncovering Social Network Sybils in the Wild”. In: *ACM Transactions on Knowledge Discovery from Data* 8 (2011-06). DOI: 10.1145/2068816.2068841.

[14] Luca Becchetti et al. “Efficient Semi-Streaming Algorithms for Local Triangle Counting in Massive Graphs”. In: 2008-08. DOI: 10.1145/1401890.1401898.

[15] Ha-Myung Park et al. “MapReduce Triangle Enumeration With Guarantees”. In: *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. CIKM ’14. Shanghai, China: Association for Computing Machinery, 2014, pp. 1739–1748. ISBN: 9781450325981. DOI: 10.1145/2661829.2662017. URL: https://doi.org/10.1145/2661829.2662017.

[16] Lorenzo De Stefani et al. “TRIÈST: Counting Local and Global Triangles in Fully-dynamic Streams with Fixed Memory Size”. In: *CoRR* abs/1602.07424 (2016). arXiv: 1602.07424. URL: http://arxiv.org/abs/1602.07424.

[17] *Company – UPMEM*. URL: https://www.upmem.com/company/.

[18] K. K. Chang. *Understanding and Improving the Latency of DRAM-Based Memory Systems*. URL: https://people.inf.ethz.ch/omutlu/pub/understanding-latency-variation-in-DRAM-chips_kevinchang_sigmetrics16-talk.pdf.

[19] JEDEC Solid State Technology Association, 2015.

[20] Donghyuk Lee et al. "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost". In: *ACM Trans. Archit. Code Optim.* 12.4 (2016-01). ISSN: 1544-3566. DOI: 10.1145/2832911. URL: https://doi.org/10.1145/2832911.

[21] Jalil Boukhobza and Pierre Olivier. "10 - Emerging Non-volatile Memories". In: *Flash Memory Integration*. Ed. by Jalil Boukhobza and Pierre Olivier. Elsevier, 2017, pp. 203–224. ISBN: 978-1-78548-124-6. DOI: https://doi.org/10.1016/B978-1-78548-124-6.50014-1. URL: https://www.sciencedirect.com/science/article/pii/B9781785481246500141.

[22] Shuangchen Li et al. "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories". In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)* (2016), pp. 1–6.

[23] Gabriel H. Loh et al. "Interconnect-Memory Challenges for Multi-Chip, Silicon Interposer Systems". In: *Proceedings of the 2015 International Symposium on Memory Systems*. MEMSYS '15. Washington DC, DC, USA: Association for Computing Machinery, 2015, pp. 3–10. ISBN: 9781450336048. DOI: 10.1145/2818950.2818951. URL: https://doi.org/10.1145/2818950.2818951.

[24] Eero Lehtonen et al. "Recursive Algorithms in Memristive Logic Arrays". In: *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on* 5 (2015-06), pp. 279–292. DOI: 10.1109/JETCAS.2015.2435531.

[25] UPMEM. *UPMEM Processing In-Memory (PIM), Ultra-efficient acceleration for data-intensive applications (Tech Paper)*. 2022.

[26] *Developer – UPMEM*. URL: https://www.upmem.com/developer/.

[27] Juan Gómez-Luna et al. "Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System". In: *IEEE Access* 10 (2022), pp. 52565–52608. DOI: 10.1109/ACCESS.2022.3174101.

[28] N. Alon, R. Yuster, and U. Zwick. “Finding and Counting Given Length Cycles”. English (US). In: *Algorithmica* 17.3 (1997-03), pp. 209–223. ISSN: 0178-4617. DOI: 10.1007/BF02523189.

[29] Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. “Reductions in streaming algorithms, with an application to counting triangles in graphs”. In: *ACM-SIAM Symposium on Discrete Algorithms*. 2002.

[30] T. E. Hull and A. R. Dobell. “Mixed Congruential Random Number Generators for Binary Machines”. In: *J. ACM* 11.1 (1964-01), pp. 31–40. ISSN: 0004-5411. DOI: 10.1145/321203.321208. URL: https://doi.org/10.1145/321203.321208.

[31] *Intel® Xeon® Silver 4215 Processor*. URL: https://ark.intel.com/content/www/it/it/ark/products/193389/intel-xeon-silver-4215-processor-11m-cache-2-50-ghz.html.

[32] *UPMEM DPU SDK*. URL: https://sdk.upmem.com/.

[33] Jaewon Yang and Jure Leskovec. *Defining and Evaluating Network Communities based on Ground-truth*. 2012. arXiv: 1205.6233 [cs.SI].

[34] Jure Leskovec and Julian Mcauley. “Learning to Discover Social Circles in Ego Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/7a614fd06c325499f1680b9896beedeb-Paper.pdf.

[35] *Sequential readers on the UPMEM architecture*. URL: https://sdk.upmem.com/2023.1.0/031_DPURuntimeService_Memory.html#sequential-readers.