TriPIM — Exact Triangle Counting on UPMEM Processing-in-Memory for Graph Analytics

Morteza Baradaran
morteza@virginia.edu
University of Virginia
Charlottesville, Virginia, USA

Khyati Kiyawat
vyn9mp@virginia.edu
University of Virginia
Charlottesville, Virginia, USA

Akhil Shekar
as8hu@virginia.edu
University of Virginia
Charlottesville, Virginia, USA

Abdullah T. Mughrabi
atmughra@virginia.edu
University of Virginia
Charlottesville, Virginia, USA

Kevin Skadron
skadron@virginia.edu
University of Virginia
Charlottesville, Virginia, USA

Abstract

Efficient triangle counting remains a fundamental challenge in graph analytics, with applications spanning computational biology, social network analysis, and emerging AI workloads such as Graph Neural Networks (GNNs). As these AI models scale to larger graphs, existing CPU- and GPU-based methods face scalability limits due to memory bottlenecks and limited parallelism. The TriCORE methodology introduced significant improvements with its binary search-driven algorithm, enhancing thread parallelism and memory efficiency on GPUs. This optimization allows TriCORE to outperform existing techniques and handle larger graphs. However, its reliance on multiple graph representations and the inherent limitations of GPU memory capacity can hinder its scalability and practical utility for Exascale graph datasets.

We present TriPIM, a novel architectural solution that combines TriCORE's algorithmic strengths with UPMEM Processing-In-Memory (PIM) technology to overcome graph scalability and memory bandwidth limitations. TriPIM leverages PIM to minimize data movement and accelerate triangle counting computations directly within memory. This integration maintains the algorithmic benefits of TriCORE while extending its applicability to significantly larger graph datasets, overcoming the constraints of traditional CPU- and GPU-based implementations. TriPIM scales triangle counting by partitioning Compressed Sparse Row (CSR) data across multiple UPMEM DIMMs and running TriCORE's binary-search engine independently within each memory partition. In summary, TriPIM offers a scalable and efficient solution for triangle counting in graph analytics, combining the strengths of TriCORE with the benefits of PIM technology. Our evaluation demonstrates TriPIM's superior performance over TriCORE (GPU) and GAP (CPU) benchmarks for Exascale graphs.

Keywords

Triangle Counting, UPMEM, Large-Scale Graph Analytics, Graph Neural Networks, Processing in Memory

1 Introduction

Triangle counting is an essential algorithm for analyzing graph structures, including clustering coefficients [41], k-truss [40], detection of spams [4], link recommendation [37] and social network analysis [6, 8]. In addition to these traditional applications, triangle counting is becoming increasingly important in AI workloads such as graph neural networks (GNNs), community detection, and structural reasoning in large language models. Recent studies highlight that as these models scale to larger graphs, the ability to efficiently count or reason about triangles becomes a computational bottleneck, underscoring the need for fast and scalable triangle counting techniques [7, 16, 23].

Table 1: Triangle Counting Optimization Methods and Their Complexities

Method	Complexity	References
Naïve Methods	$O(n^3)$	[14, 34]
Optimized Edge Iteration	$O(m \cdot d)$	[34]
Vertex Ordering and Intersection	$O(m \cdot \sqrt{m})$	[19]
Adjacency Matrix Multiplication	$O(n^3)$	[14]
Hash-Based Methods	$O(m \cdot d)$	[30]
Exact Algorithms (e.g., Schank and Wagner)	$O(m \cdot d)$	[27]
Approximate Algorithms	$O(m \cdot \log n)$	[36]

Table 2: Symbols

Symbol	Meaning
n	Number of vertices in the graph
m	Number of edges in the graph
d	Average degree of the vertices

In vertex-centric triangle counting, the algorithm examines each pair of connected nodes in the graph to determine if they share a third common neighbor, forming a triangle. This process is typically represented as an intersection between a node's 1-hop neighbors (direct neighbors) and 2-hop neighbors (neighbors of neighbors), resulting in the graph triangle count.

Triangle counting in large graphs presents two significant challenges: computational complexity and efficient data access. To address these, several approaches have been proposed on CPUs, GPUs, and FPGAs [2, 3, 11–13, 15, 21, 32]. In practice, CPU-based methods such as GAP [2, 3] often suffer from cache inefficiencies, while GPU frameworks like TriCORE [12] are constrained by device memory capacity. Although several optimizations have been proposed to

improve memory locality [5, 22, 24–26, 33, 35, 38], these techniques still fall short of overcoming the fundamental scalability barriers. Therefore, addressing these challenges requires methods that not only scale with graph size but also minimize computational overhead and maintain balanced workloads across compute nodes.

Triangle Counting Runtime Complexity $O(|n|^3)$: One effective approach to reducing computational complexity is the vertex ordering method used in the GAP Benchmark Suite (GAPBS) [2, 3]. As presented in Algorithm 1, vertex ordering involves sorting vertices and performing ordered intersections of neighbor lists to count triangles efficiently. The algorithm considers each vertex u, iterates over its neighbors v such that $v > u$, and for each pair (u, v), finds common neighbors w such that $w > v$. By leveraging the sorted order of vertices, this method reduces the need to consider all possible triplets, thereby lowering the computational complexity from $O(|n|^3)$ in the naive approach to $O(m \cdot \sqrt{m})$, where m is the number of edges and n is the number of vertices.

The Binary Search method shown in Algorithm 2, used in TriCORE [12] to effectively address inefficient node lookups by leveraging the sorted nature of neighbor lists to perform quick searches. When processing each edge (u, v), the algorithm performs a binary search on the neighboring list of v to identify intersections with the neighbor list of u. This approach reduces the complexity of finding a common neighbor from $O(d)$ to $O(logd)$, where d is the degree of the vertex.

Optimization Dependent Data Movement From GPU to PIM: The partitioning technique used by TriCORE [12] is highly efficient for GPU scenarios as it evenly distributes the computational load across multiple GPU cores. This ensures that each core has a balanced workload, minimizing idle times and maximizing the utilization of the parallel processing capabilities inherent in GPUs. Dividing the graph based on vertex ranges and balancing the number of edges across partitions facilitates efficient graph processing. However, this partitioning approach faces challenges when implemented for emerging Processing-in-Memory (PIM) technologies [17, 18, 28, 29, 31] like UPMEM [1, 9]. In PIM architectures, data processing relies on the data being stationary and load-balanced within the memory. Unlike TriCORE, which can handle edge data transfers efficiently via stream buffers, PIM technologies require data to remain fixed in memory to exploit the benefits of in-memory processing fully. The dependence on GPU stream buffers and frequent edge data requests inherent in TriCORE's partitioning method can lead to inefficiencies and performance degradation in PIM environments. Therefore, while TriCORE's partitioning technique is optimized for the high-throughput stream buffers and parallel nature of GPUs, it is less suitable for PIM architectures, where stationary and balanced data placement in memory is crucial for optimal performance.

TriPIM — Breaking Through the Scalability Barrier with UPMEM: TRUST [24], an extension to TriCORE, mitigates some

Algorithm 1: GAPBS - TriangleCount

```
Data: Graph g
Result: Total Triangle Count total
1  total ← 0 ;        // Initialize total triangle count
   /* Parallel triangle counting using OpenMP     */
2  #pragma omp parallel for reduction(+ : total)
     schedule(dynamic, 64)
3  for u ← 0 to g.num_nodes() − 1 do
4      foreach v in g.outNeigh(u) do
5          if v > u then
6              break ;        // Ensure v < u for ordered
                counting
7          end
8          it ← g.outNeigh(v).begin() ;      // Initialize
            iterator for v's neighbors
9          foreach w in g.outNeigh(u) do
10             if w > v then
11                 break ;    // Ensure w < v for ordered
                    counting
12             end
13             while ∗it < w do
14                 it++ ; // Advance iterator to find w
15             end
16             if w == ∗it then
17                 total++ ;            // Triangle found
18             end
19         end
20     end
21 end
22 return total ;    // Return the total triangle count
```

Algorithm 2: TriCORE - Binary Search Intersection

```
Data: Array neighborIdxs_m, Long start, Size end, NodeID
      target, Pointer cache_w
Result: Index of target in neighborIdxs_m or −1 if not
        found
1  left ← (start == −1)?0 : start ;     // Initialize left
    boundary
2  right ← end − 1 ;        // Initialize right boundary
3  while left ≤ right do
4      medium ← left + ((right − left) ≫ 1) ; // Calculate
        middle index
5      current ← neighborIdxs_m(medium) ;       // Load
        current value
6      if current == target then
7          return medium ;      // Target found, return
            medium index
8      end
9      if current < target then
10         left ← medium + 1 ;   // Adjust left boundary
11     end
12     else
13         right ← medium − 1 ; // Adjust right boundary
14     end
15 end
16 return −1 ;          // Target not found, return -1
```

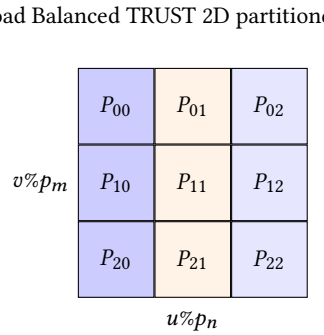

Figure 1: TRUST 2D partitioning for vertex-centric triangle counting. Vertices are labeled according to degree for load-balanced partitioning.

of its memory overhead by using a vertex-ordered balanced hash-based partitioning, thus eliminating the need for edge stream buffers. This approach simplifies data handling, reduces memory consumption, and minimizes data movement overhead. While TRUST uses less memory for each partition by not requiring the edge list streams, it incorporates hash-based intersection, which requires extra memory resources for building hashmaps for each intersect operation. In this work, we introduce TriPIM as a hybrid approach that combines TRUST-based memory-efficient partitioning and the TriCORE binary search technique, which avoids using hash-based intersections. TriPIM reduces graph storage and intersection overhead using prior triangle counting techniques for PIM. By leveraging the efficient partitioning strategy of TRUST and the fast intersection method of the TriCORE binary search, TriPIM aims to optimize triangle counting in PIM environments, ensuring high performance for the Exascale graph with low memory usage and low computational overhead.

2 Background and Related Work

2.1 UPMEM Architecture

An UPMEM module [1, 9, 39] is a standard DDR4-2400 DIMM containing several PIM chips. Each module's DPUs operate as a parallel coprocessor to the host CPU. Each UPMEM PIM chip has 8 DPUs, each with 64-MB Main memory (MRAM), 24-KB Instruction RAM (IRAM), and 64-KB Working RAM (WRAM).

The host CPU can access MRAM to transfer input data and retrieve results. Data transfers can be parallel across MRAM banks if buffers are the same size; otherwise, they occur serially. DPUs do not communicate directly; inter-DPU communication happens via the host CPU. Concurrent access by the CPU and DPUs to the same MRAM bank is not supported in current UPMEM-based PIM systems.

2.2 Triangle Counting (TC) on CPU

For CPU experiments, we used GAPBS [2, 3], a portable high-performance baseline that only requires a compiler with support for C++11. It uses OpenMP for parallelism but can be compiled without OpenMP to run serially. The details of the benchmark can be found in the specification.

2.3 Triangle Counting (TC) on GPU

TriCORE [12] is a scalable GPU-based triangle counting system with three key innovations. It employs a binary search-based algorithm to boost thread parallelism and memory performance on GPUs. Unlike previous methods requiring multiple graph representations in GPU memory, TriCORE partitions CSR data across GPUs and streams the edge list from CPU memory, enabling it to handle larger graphs. Additionally, it uses dynamic workload management to balance tasks across GPUs. TriCORE can count triangles in the billion-edge Twitter graph in 24 seconds on a single GPU, 22 times faster than the best CPU-based solutions.

3 TriPIM Overview

3.1 Load-Balanced Hash-Based Graph Partitioning

We employ a load-balanced hash-based partitioning method for distributing the graph across multiple partitions. This approach ensures that each partition has an approximately equal number of vertices, thus balancing the computational load across the DPUs. The vertices are labeled according to their degrees to facilitate load balancing, as shown in Figure 1.

3.2 TriPIM Description

3.2.1 Local Triangle Counting. The local triangle counting algorithm runs on the DPUs in parallel, counting triangles within individual partitions. The algorithm iterates over each partition, calculates the index for the current partition, and counts the triangles. Local triangle counting allows us to leverage the DPU's parallel processing capabilities, providing efficient computation within individual partitions. This method ensures that all local triangles are counted accurately before moving on to cross-partition counting.

Algorithm 3: Partitioned - Local Triangle Counting

Data: Partitioned Graph p_g, Parameters p_m, p_n
Result: Total Triangle Count $total$

$p\_g \leftarrow b$.MakePartitionedGraph() ; // Create a partitioned graph
$total \leftarrow 0$; // Initialize total triangle count
$p\_total \leftarrow 0$; // Initialize partition triangle count
/* Local triangle counting in parallel on the DPU */
for $col \leftarrow 0$ **to** $p\_m - 1$ **do**
  for $row \leftarrow 0$ **to** $p\_n - 1$ **do**
    $idx \leftarrow row \cdot p\_m + col$; // Calculate the index for the partition
    $p\_total \leftarrow$ TriangleCount($p\_g[idx]$) ; // Count triangles in the partition (executed in parallel on DPU)
    $total \leftarrow total + p\_total$; // Accumulate the total triangle count
  end
end

3.2.2 Cross-Partition Triangle Counting. The cross-partition triangle counting algorithm identifies and counts triangles that span multiple partitions. This is achieved by iterating over pairs of partitions and counting triangles involving vertices from both. Cross-partition triangle counting is crucial for identifying triangles that span multiple partitions, which are not captured by local counting alone. Running this in parallel on the DPUs, TriPIM ensures that the computation is efficient and scalable.

Algorithm 4: Partitioned - Cross Triangle Counting

Data: Partitioned Graph p_g, Parameters p_m, p_n
Result: Total Triangle Count $total$

```
p_g ← b.MakePartitionedGraph() ;        // Create a partitioned graph
total ← 0 ;      // Initialize total triangle count
p_total ← 0 ;      // Initialize partition triangle count
/* Cross triangle counting in parallel on the DPU                    */
for col ← 0 to p_m − 1 do
    for row1 ← 0 to p_n − 1 do
        idx1 ← row1 · p_m + col ;      // Calculate the index for the first partition
        for row2 ← row1 + 1 to p_n − 1 do
            idx2 ← row2 · p_m + col ;  // Calculate the index for the second partition
            p_total ← CrossTriangleCount(p_g[idx1], p_g[idx2]) ;
            // Count cross-partition triangles (executed in parallel on DPU)
            total ← total + p_total ;  // Accumulate the total triangle count
        end
    end
end
```

4 Evaluation and Results

4.1 Methodology

We utilized the system configurations detailed in Table 3 for our evaluation. We used TriCORE as the GPU baseline and the GAP Benchmark Suite (GAP) as the CPU baseline. Additionally, we implemented the binary search and GAP triangle counting algorithms with TRUST partitioning on UPMEM. The implementation of the **TriPIM** framework is available at: https://github.com/UVA-LavaLab/TriPIM.

4.1.1 CPU Baseline. The GAP Benchmark Suite (GAP) was used as the baseline for CPU-based triangle counting. GAP provides a set of graph algorithms optimized for multi-core CPUs, allowing us to measure triangle counting performance on traditional CPU architectures.

4.1.2 GPU Baseline. TriCORE was used as the baseline for GPU-based triangle counting. It uses a binary search method to find neighboring triangles and takes full advantage of the parallel processing capabilities of GPUs to achieve high throughput.

4.1.3 UPMEM PIM. In order to take full advantage of UPMEM's processing-in-memory (PIM) technology, we have incorporated both the binary search technique and GAP's triangle counting algorithms with TRUST partitioning. This approach aims to combine the memory-efficient partitioning of TRUST with the TriCORE intersection method of binary search.

Table 3: System environment for TriPIM evaluation

GPU	
Model	NVIDIA A40
CUDA Cores	10752
Boost Clock Speed	1.74 GHz
Memory	48 GB GDDR6
Memory Bandwidth	696 GB s^{-1}
PIM Configuration	
Model	UPMEM PIM
DPUs	2549 used / 2560 total
DPU Frequency	350 MHz
Memory per DPU	64 MB
Total Memory	160 GB
DIMMs	20
Memory Bandwidth	1000 GB s^{-1}
Host CPU	
Model‖Cores‖Threads	Intel Xeon Silver 4216‖16‖32
Clock Speed	2.10 GHz
Memory	256 GB
OS	Ubuntu 22.04 LTS
L1‖L2‖L3 Cache	512 kB (8-way)‖16 MB (16-way)‖22 MB (11-way)

Table 4: Graph workloads used in evaluation

Graph	#Vertices	#Edges	Degree	Size (MB)	Triangle Count
Kron_13_20	8,191	246,888	20	1	1,744,952
Kron_16_18	65,536	2,026,386	18	8	19,672,632
Kron_18_17	262,143	8,057,720	17	32	93,521,523
Urand_13_16	8,192	261,556	16	1	5,559
Urand_16_16	65,536	1,965,534	16	8	4,467
Urand_17_32	131,072	8,386,472	32	32	43,363

4.2 System Environment

4.2.1 CPU System. The evaluations for both the GPU and CPU were conducted on a host CPU with an Intel Xeon Silver 4216 processor. This CPU is equipped with 16 cores and 32 threads, operates at a clock speed of 2.10 GHz, and has 256 GB of memory. The operating system used for the evaluations was Ubuntu 22.04 LTS.

4.2.2 GPU System. For the GPU evaluation, we utilized the NVIDIA A40. This GPU is equipped with 10,752 CUDA cores, providing extensive parallel processing capabilities. The boost clock speed reaches 1.74 GHz. The A40 includes 48 GB of GDDR6 memory, offering a memory bandwidth of 696 GB/s.

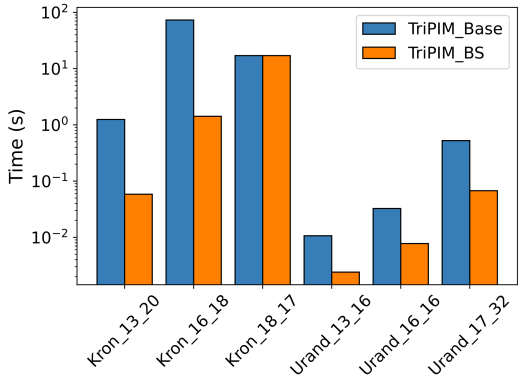

Figure 2: Comparing Binary Search vs. Base Intersection

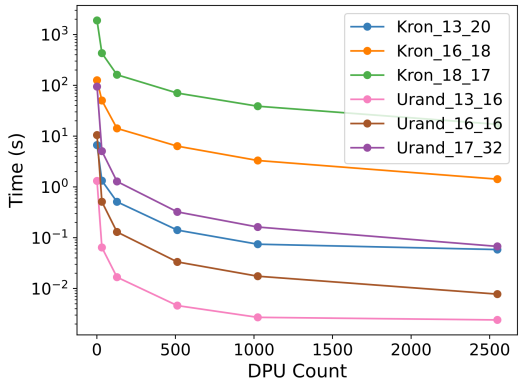

Figure 3: TriPIM Scalability with UPMEM

4.2.3 UPMEM System. In our evaluation of the UPMEM PIM, we used a configuration with 2,560 DRAM Processing Units (DPUs). Each DPU runs at a frequency of 350 MHz and has 64 MB of memory, giving a total memory capacity of 160 GB. The UPMEM architecture offers a memory bandwidth of 1,000 GB/s, allowing for rapid data processing directly within the memory.

4.3 Graph Datasets

In our evaluation, we utilized a diverse set of synthetic graph datasets generated using GAPBS Kronecker and Urand models to assess the performance of TriPIM as listed in Table 4.

Kronecker graphs [20], generated using the GAPBS Kronecker synthetic graph generator, are a key part of our research. With parameters A=0.57, B=0.19, C=0.19, and D=0.05, as used in the Graph 500 benchmark, this model replicates many real-world network properties. The Kronecker model's scalability allows for the generation of large graphs from smaller seed graphs through the Kronecker product, creating synthetic graphs that closely resemble actual networks in various domains.

Urand graphs are generated using the Erdős–Rényi model (Uniform Random) [10]. This model creates graphs by randomly connecting vertices with edges, with each edge having an equal probability of being present. It represents the worst-case scenario for locality, as every vertex has an equal probability of being a neighbor to every other vertex. The uniform random nature of Urand graphs makes them valuable as a baseline for benchmarking, as they lack the structured properties found in real-world graphs, thus allowing for a clear contrast with Kronecker graphs.

4.4 TriPIM Binary Search vs. Base

In this section, we compare two intersection methods: Base and Binary Search (BS). In the Base approach, the inner loop of triangle counting compares the elements of two adjacency lists (i.e., 1-hop and 2-hop), and a triangle is counted whenever the elements are equal. In the Binary Search approach, the longer adjacency list is added to the binary search tree, and the elements in the shorter adjacency list serve as keys for probing the binary search algorithm. As illustrated in Figure 2, the Binary Search algorithm "TriPIM_BS" significantly improves runtime compared to the Base approach "TriPIM_Base", achieving on average 7.3× across all datasets and up to a 51× speedup in the Kron_16_18 experiment.

4.5 TriPIM Scalability

Figure 3 illustrates the scalability of TriPIM for small graphs across different DPU counts. As the number of DPUs increases, execution time drops sharply until the count reaches eight, after which the decrease becomes more gradual. The steep drop in execution time at lower DPU counts reflects reduced parallelism, as each DPU must process a larger share of the graph, leading to longer runtimes. This underscores the importance of maintaining high DPU counts to preserve TriPIM performance, even for small graphs.

4.6 Exascale Graph Processing

In this section, we evaluate TriPIM against GAP (CPU baseline) and TriCORE (GPU baseline) across varying graph sizes (Figure 4). The x-axis values indicate the factor by which the base graph size (*Size*) is multiplied. The evaluation starts from *Size*×32 and increases by a factor of four at each step up to *Size*×2048 for CPU and GPU baselines, while TriPIM can scale further to *Size*×2549, corresponding to the total number of available DPUs. For all three systems, we report only the execution time of the triangle counting kernel; initial setup, graph loading, and result gathering time are excluded. In TriPIM, increasing the graph size simply loads more partitions onto more DPUs, with each DPU processing the same partition size (*Size*). As a result, TriPIM's kernel time remains nearly constant as the total graph size increases. This setup enables direct comparison from small to extremely large graphs.

Across all datasets, TriCORE achieves the lowest execution time for small graphs (e.g., *Size*×32, *Size*×128), outperforming both TriPIM and GAP. However, it fails to execute beyond certain graph sizes due to GPU memory limitations, as indicated by the vertical red "Out of Memory" markers in Figure 4. GAP is faster than TriPIM at smaller graph sizes, benefiting from efficient CPU execution when the graph fits comfortably in the cache hierarchy. Moreover, for small graphs, TriPIM underperforms partly because the setup overhead and characteristics of UPMEM's architecture may not offer a significant advantage over traditional CPU or GPU processing. For large graphs (e.g., *Size*×2048 for CPU/GPU baselines and

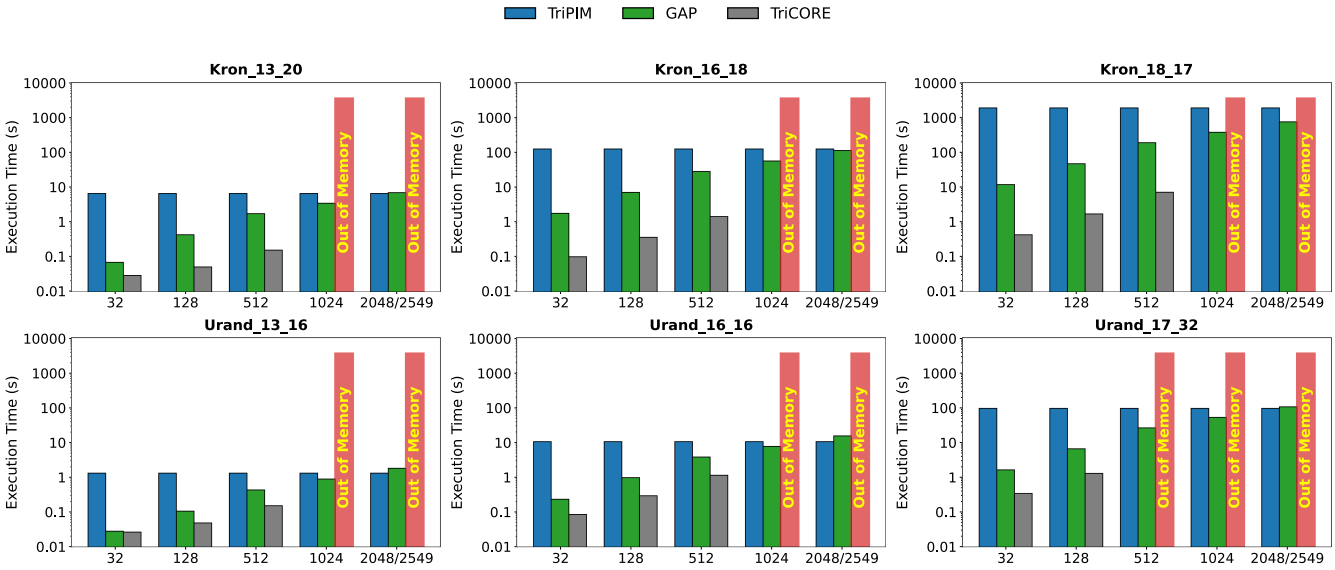

Figure 4: Performance comparison of TriPIM with TriCORE (GPU baseline) and GAP (CPU baseline) across six graph datasets. The x-axis represents increasing graph sizes, from *Size*×32 to *Size*×2048 for CPU and GPU baselines, and up to *Size*×2549 for TriPIM, where *Size* denotes the base graph size. "Out of Memory" markers indicate graph sizes where TriCORE could not execute due to GPU memory limitations.

Size×2549 for TriPIM), GAP's execution time rises rapidly, whereas TriPIM maintains nearly constant performance. Scaling becomes increasingly challenging for CPU and GPU due to hardware constraints—CPUs suffer from more cache misses, while GPUs are limited by their memory capacity (e.g., 48 GB for the A40). At large scales, TriPIM not only surpasses GAP in most cases but can also process graphs that exceed GPU memory capacity.

5 Conclusions and Future Work

Efficient triangle counting remains a main challenge in graph analytics, underpinning applications in computational biology, social networks, and emerging AI workloads such as graph neural networks and community detection. While CPU and GPU approaches, including TriCORE, provide strong performance for small to medium graphs, they face scalability barriers due to cache inefficiencies and limited device memory.

TriPIM integrates the algorithmic advantages of TriCORE with UPMEM's Processing-In-Memory (PIM) architecture to address these challenges. By minimizing data movement and distributing workloads across thousands of DPUs, TriPIM enables scalable triangle counting on datasets that exceed the capacity of conventional processors. Although TriPIM is slower for small graphs due to setup overhead, balanced partitioning allows it to roughly match or surpass CPU/GPU performance at larger graphs. A key strength is its ability to execute beyond GPU memory limits, demonstrating clear scalability advantages for massive graphs.

While these results highlight the advantage of using PIM-based architectures, the comparison is not normalized across hardware cost, memory capacity, or power consumption. Tradeoffs under cost and power constraints, identifying new PIM features, and generalizing these methods to broader graph analytics workloads, are interesting areas for future work.

6 Acknowledgement

This work was supported in part by PRISM, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] 2018. *Introduction to UPMEM PIM. Processing-in-memory (PIM) on DRAM Accelerator (White Paper).* Technical Report. UPMEM, Grenoble, France.
[2] S. Beamer. 2015. *GAP benchmark suite reference code v0.6.* https://github.com/sbeamer/gapbs
[3] S. Beamer, K. Asanovic, and D. A. Patterson. 2015. The GAP benchmark suite. *arXiv:1508.03619* (Aug. 2015).
[4] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. 2008. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Las Vegas, Nevada, USA) *(KDD '08)*. Association for Computing Machinery, New York, NY, USA, 16–24. doi:10.1145/1401890.1401898
[5] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. 2008. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Las Vegas, Nevada, USA) *(KDD '08)*. Association for Computing Machinery, New York, NY, USA, 16–24. doi:10.1145/1401890.1401898
[6] R. Burt. 2004. Structural holes and good ideas. *Amer. J. Sociology* (2004).
[7] Z. Chen, L. Chen, S. Villar, and J. Bruna. 2020. Can graph neural networks count substructures?. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 871, 13 pages.
[8] J. Coleman. 1988. Social capital in the creation of human capital. *Amer. J. Sociology* (1988).
[9] F. Devaux. 2019. The true processing in memory accelerator. In *IEEE Hot Chips 31 Symposium (HCS)*. 1–24.

[10] P. Erdős and A. Rényi. 1959. On Random Graphs I. *Publicationes Mathematicae* 6 (1959), 290–297.

[11] L. Hu, L. Zou, and Y. Liu. 2021. Accelerating Triangle Counting on GPU. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 736–748. doi:10.1145/3448016.3452815

[12] Y. Hu, H. Liu, and H. H. Huang. 2018. TriCore: Parallel Triangle Counting on GPUs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 171–182. doi:10.1109/SC.2018.00017

[13] S. Huang, M. El-Hadedy, C. Hao, Q. Li, V. Mailthody, K. Date, J. Xiong, D. Chen, R. Nagi, and W. Hwu. 2018. Triangle Counting and Truss Decomposition using FPGA. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–7. doi:10.1109/HPEC.2018.8547536

[14] A. Itai and M. Rodeh. 1977. Finding a minimum circuit in a graph *(STOC '77)*. Association for Computing Machinery, New York, NY, USA, 1–10.

[15] O. Jaiyeoba, A. Mughrabi, M. Baradaran, B. Gul, and K. Skadron. 2024. Swift: A Multi-FPGA Framework for Scaling Up Accelerated Graph Analytics. In *2024 International Conference on Field Programmable Technology (ICFPT)*. 1–10. doi:10.1109/ICFPT64416.2024.11113456

[16] C. Kanatsoulis and A. Ribeiro. 2024. Counting graph substructures with graph neural networks. In *International Conference on Learning Representations*.

[17] J. Kim, S. Kang, S. Lee, H. Kim, W. Song, Y. Ro, S. Lee, D. Wang, H. Shin, B. Phuah, J. Choi, J. So, Y. Cho, J. Song, J. Choi, J. Cho, K. Sohn, Y. Sohn, K. Park, and N. Kim. 2021. Aquabolt-XL: Samsung HBM2-PIM with in-memory processing for ML accelerators and beyond. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–26. doi:10.1109/HCS52781.2021.9567191

[18] Y. Kwon, V. Kornijcuk, N. Kim, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, G. Kim, B. An, J. Kim, J. Lee, I. Kim, J. Park, C. Park, Y. Song, B. Yang, H. Lee, S. Kim, D. Kwon, S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, M. Lee, M. Shin, M. Shin, J. Cha, C. Jung, K. Chang, C. Jeong, E. Lim, I. Park, J. Chun, and Sk Hynix. 2022. System Architecture and Software Stack for GDDR6-AiM. In *2022 IEEE Hot Chips 34 Symposium (HCS)*. 1–25. doi:10.1109/HCS55958.2022.9895629

[19] M. Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science* 407 (2008), 458–473.

[20] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. 2010. Kronecker Graphs: An Approach to Modeling Networks. *J. Mach. Learn. Res.* 11 (March 2010), 985–1042.

[21] J. Liang, M. Rajashekar, X. Tian, and Z. Fang. 2024. HiTC: High-Performance Triangle Counting on HBM-Equipped FPGAs Using HLS. In *2024 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. 1–6. doi:10.1109/PACRIM61180.2024.10690214

[22] A. T. Mughrabi, M. Baradaran, A. Samara, and K. Skadron. 2024. ECG: Expressing Locality and Prefetching for Optimal Caching in Graph Structures. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 520–525. doi:10.1109/IPDPSW63119.2024.00105

[23] L. H. Nguyen and Y. Yan. 2024. Evaluating the structural awareness of large language models on graphs: Can they count substructures?. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*.

[24] S. Pandey, Z. Wang, S. Zhong, C. Tian, B. Zheng, X. Li, L. Li, A. Hoisie, C. Ding, D. Li, and H. Liu. 2021. TRUST: Triangle Counting Reloaded on GPUs. *arXiv:2103.08053* (2021).

[25] H. Park, F. Silvestri, R. Pagh, C. Chung, S. Myaeng, and U. Kang. 2018. Enumerating Trillion Subgraphs On Distributed Systems. *ACM Trans. Knowl. Discov. Data* 12, 6, Article 71 (Oct. 2018), 30 pages. doi:10.1145/3237191

[26] K. Ravichandran, A. Subramaniasivam, P.S. Aishwarya, and N.S. Kumar. 2023. Chapter Eight - Fast exact triangle counting in large graphs using SIMD acceleration. In *Principles of Big Graph: In-depth Insight*, Ripon Patgiri, Ganesh Chandra Deka, and Anupam Biswas (Eds.). Advances in Computers, Vol. 128. Elsevier, 233–250. doi:10.1016/bs.adcom.2021.10.003

[27] T. Schank and D. Wagner. 2005. Finding, counting and listing all triangles in large graphs, an experimental study. In *International Workshop on Experimental and Efficient Algorithms*. 606–609.

[28] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. 2017. Ambit: in-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 273–287. doi:10.1145/3123939.3124544

[29] A. Shekar, M. Baradaran, S. Tajdari, and K. Skadron. 2023. HashMem: PIM-based Hashmap Accelerator. doi:arXiv:2306.17721[cs.AR]

[30] J. Shun and K. Tangwongsan. 2015. Multicore triangle computations without tuning. In *2015 IEEE 31st International Conference on Data Engineering*. 149–160. doi:10.1109/ICDE.2015.7113280

[31] F. Siddique, D. Guo, Z. Fan, M. Gholamrezaei, M. Baradaran, A. Ahmed, H. Abbot, K. Durrer, K. Nandagopal, E. Ermovick, K. Kiyawat, B. Gul, A. Mughrabi, A. Venkat, and K. Skadron. 2024. Architectural Modeling and Benchmarking for Digital DRAM PIM. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*. 247–261. doi:10.1109/IISWC63097.2024.00030

[32] G. Singh, M. Alser, D. Senol Cali, D. Diamantopoulos, J. Gómez-Luna, H. Corporaal, and O. Mutlu. 2021. FPGA-Based Near-Memory Acceleration of Modern Data-Intensive Applications. *IEEE Micro* PP (06 2021), 1–1. doi:10.1109/MM.2021.3088396

[33] L. De Stefani, A. Epasto, M. Riondato, and E. Upfal. 2017. TRIÈST: Counting Local and Global Triangles in Fully Dynamic Streams with Fixed Memory Size. *ACM Trans. Knowl. Discov. Data* 11, 4, Article 43 (June 2017), 50 pages. doi:10.1145/3059194

[34] S. Suri and S. Vassilvitskii. 2011. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web* (Hyderabad, India) *(WWW '11)*. Association for Computing Machinery, New York, NY, USA, 607–614. doi:10.1145/1963405.1963491

[35] A. Sarah Tom, N. Sundaram, N. K. Ahmed, S. Smith, S. Eyerman, M. Kodiyath, I. Hur, F. Petrini, and G. Karypis. 2017. Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. doi:10.1109/HPEC.2017.8091054

[36] C. E. Tsourakakis. 2008. Fast Counting of Triangles in Large Real Networks without Counting: Algorithms and Laws. In *2008 Eighth IEEE International Conference on Data Mining*. 608–617. doi:10.1109/ICDM.2008.72

[37] C. E. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos. 2011. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Network Analysis and Mining* 1 (2011), 75–81. doi:10.1007/s13278-010-0001-9

[38] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. 2009. DOULION: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Paris, France) *(KDD '09)*. Association for Computing Machinery, New York, NY, USA, 837–846. doi:10.1145/1557019.1557111

[39] UPMEM. 2020. *UPMEM Website*. https://www.upmem.com

[40] J. Wang and J. Cheng. 2012. Truss decomposition in massive networks. *Proc. VLDB Endow.* 5, 9 (May 2012), 812–823. doi:10.14778/2311906.2311909

[41] D. J. Watts and S. H. Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* 393 (1998), 440–442. doi:10.1038/30918