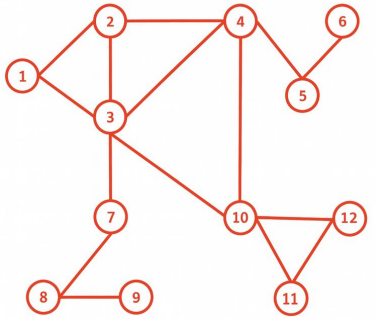
TriPIM: Exact Triangle Counting on UPMEM Processing-in-Memory for Graph Analytics

Morteza Baradaran, Khyati Kiyawat, Akhil Shekar,
Abdullah T. Mughrabi, Kevin Skadron
Email: morteza@virginia.edu

MEMSYS 2025

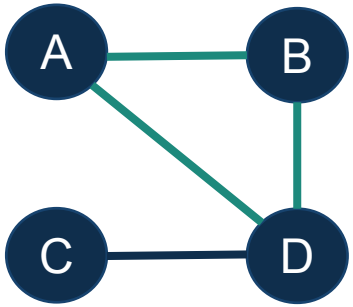
GitHub

Introduction (Overview)



- Triangle counting
  - Counting the triangles (three mutually connected vertices) in a graph.
  - A critical task in graph analytics applications such as social network analysis, link recommendation and emerging AI workloads (like GNNs, structural reasoning in LLMs).
- Existing implementations on CPUs and GPUs
  - Struggle with memory efficiency and scalability limitations for large-scale graphs.
- To overcome these limitations: we propose TriPIM, which integrates
  - Binary search intersection
    - faster triangle counting
  - Load-balanced partitioning of graphs
    - Extensible to significantly larger graphs
  - Uses a real-world Processing-In-Memory (PIM) DRAM technology
    - Minimizing data movement and accelerating computations directly within memory.

Introduction (Triangle Counting)



- Example (naïve intersection, $O(n^3)$):
  - A (u) and B (v) are connected
  - NeighborList (A) = {B, D}
  - NeighborList (B) = {A, D}
  - NeighborList (A) ∩ NeighborList (B) = {D}
    - Count 1 triangle
- Redundancy Issue: The triangle ABD is counted multiple times (once for A, once for B, and once for D) and also for each edge it is counted twice.
  - We should divide the total result by 6.

Naïve intersection TC:

```
Graph g
for u ← 0 to g.num nodes() - 1
  foreach v in g.outNeigh(u)
    it ← g.outNeigh(v).begin() ;
    foreach w in g.outNeigh(u)
      while *it < w
        it + +
        if w == *it  then
          total + + ; // Triangle found
```

Problem Statement

n=number of vertices
m= number of edges
d=degree of a vertex

Triangle counting in large graphs presents two significant challenges:

1- Runtime Complexity

- Naïve approach: 3 nested for loops ($O(n^3)$)
- Ordered intersection, as in GAPBS [3]
  - Sorting vertices
  - Performing ordered intersections of neighbor
  - $O(m^{3/2})$
- Binary Search method for intersection, as in TriCORE [1]
  - Sorting neighbor lists
  - For neighbor lists of nodes *u* and *v*
    - Suppose g.outNeigh(u) < g.outNeigh(v)
    - Look up each node of g.outNeigh(u) in g.outNeigh(v) using binary search
  - Complexity of intersection: O(log(d))

Ordered intersection TC:

```
Graph g
for u ← 0 to g.num nodes() - 1
  foreach v in g.outNeigh(u)
    if v > u then break ;
    it ← g.outNeigh(v).begin() ;
    foreach w in g.outNeigh(u)
      if w > v then break ;
      while *it < w
        it + +
        if w == *it  then
          total + + ; // Triangle found
```

Problem Statement

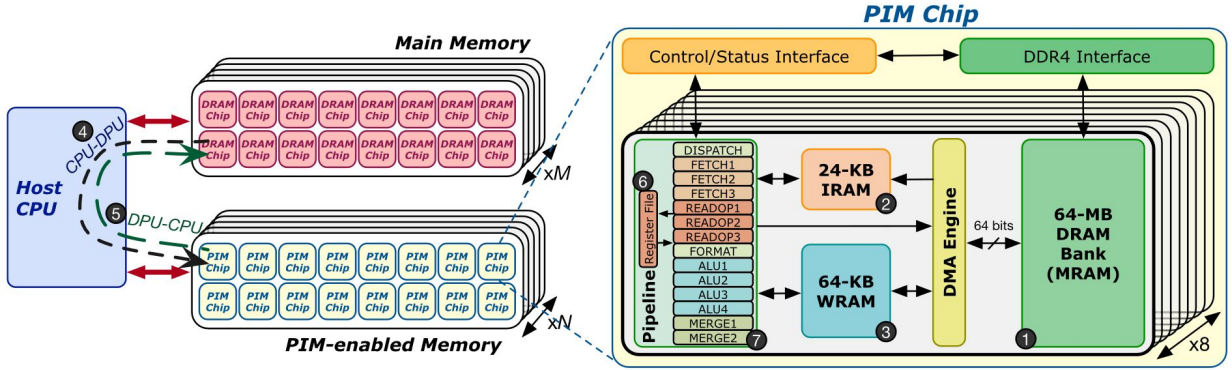
Triangle counting in large graphs presents two significant challenges:

2- Data Movement (For PIM)

- TriCORE: Evenly distributes the computational load across multiple GPU cores
  - Maximizing the utilization of GPU parallel processing capabilities
    - Dividing the graph based on vertex ranges
    - Balancing the number of edges across partitions
    - Can handle edge **data transfers** efficiently via stream buffers
- PIM technologies require data to remain fixed in memory
  - Least data movement between banks (no inter-DPU communications in UPMEM)
- We used TRUST [4] technique
  - An extension to TriCORE
  - Mitigates memory overhead by using a vertex-ordered balanced hash-based partitioning
  - Extra memory resources for building hash maps for each intersect operation.

Background - UPMEM

- Standard DDR4-2400 DIMM containing several PIM chips.
  - Each UPMEM PIM chip has 8 DPUs, each with
    - 64-MB Main memory (MRAM)
    - 24-KB Instruction RAM (IRAM)
    - 64-KB Working RAM (WRAM)
  - The host CPU can access MRAM to transfer input data and retrieve results.
  - No inter-DPU communication → only via the host CPU



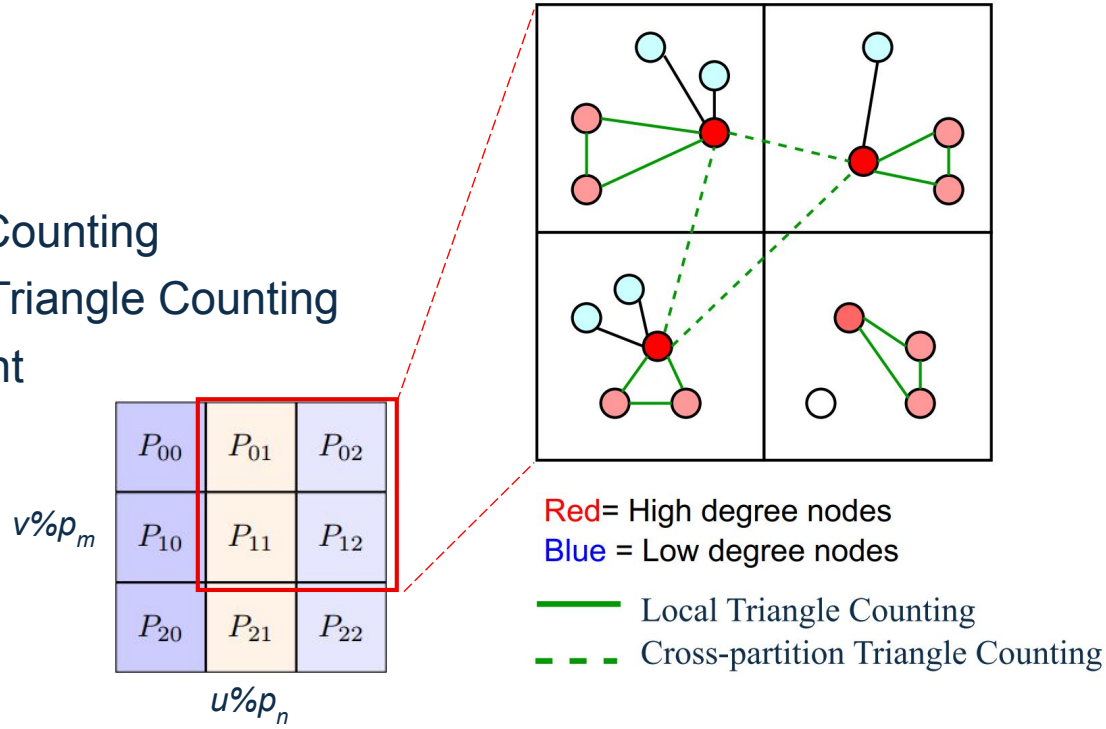
[Gómez-Luna, et. al "Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System,"]

TriPIM

- TriPIM is an approach that combines
  - Load-balanced hash-based graph partitioning (*inspired by* TRUST)
    - Ensures each partition has an approximately equal number of vertices
    - The vertices are labeled according to their degrees to facilitate load balancing
  - Binary search intersection technique (*inspired by* TriCORE)
  - Parallelizing the triangle counting using a real-world PIM, UPMEM [2]
- Triangle counting in TriPIM happens in two steps:
  - **Local Triangle Counting**: Counting triangles within individual partitions (intra-DPUs)
  - **Cross-Partition Triangle Counting**: Counts triangles that span multiple partitions (inter-DPUs), which are not captured by local counting alone.

TriPIM

- TriPIM Steps
  - 1- Partitioning the graph
  - 2- Transfer partitions to DPUs
  - 3- Local (intra-DPU) Triangle Counting
  - 4- Cross-Partition (inter-DPU)Triangle Counting
  - 5- Collecting the Triangle Count



Methodology

CPU Baseline:

The GAP Benchmark Suite

GPU Baseline:

TriCORE

Graph Datasets: Synthetic graph datasets generated using GAPBS

Kronecker [5]: replicates many real-world network properties

Urand [6]: worst-case scenario for locality, as every vertex has an equal probability of being a neighbor to every other vertex

GPU	
Model	NVIDIA A40
CUDA Cores	10752
Boost Clock Speed	1.74 GHz
Memory	48 GB GDDR6
Memory Bandwidth	696 GB s^{-1}

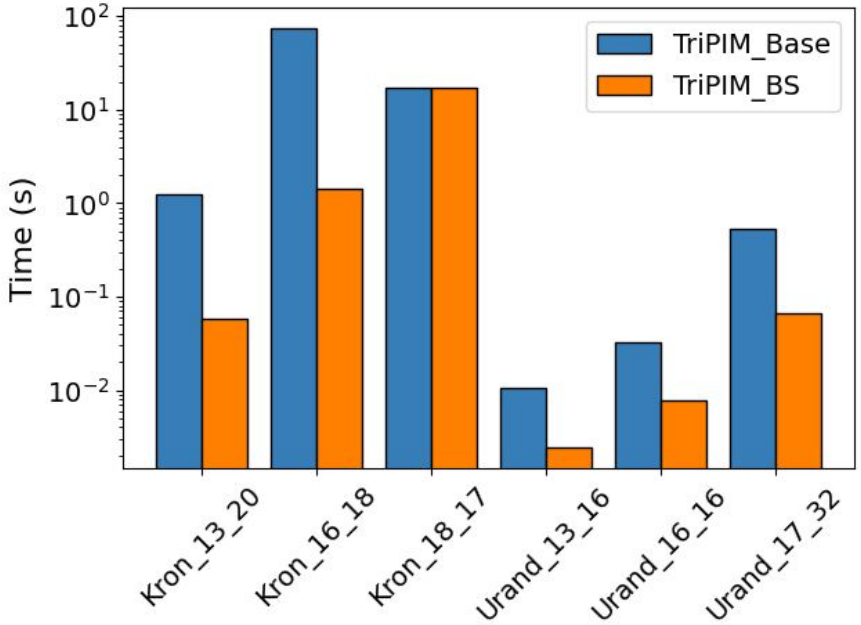
PIM Configuration	
Model	UPMEM PIM
DPUs	2549 used / 2560 total
DPU Frequency	350 MHz
Memory per DPU	64 MB
Total Memory	160 GB
DIMMs	20
Memory Bandwidth	1000 GB s^{-1}

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)

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

Results

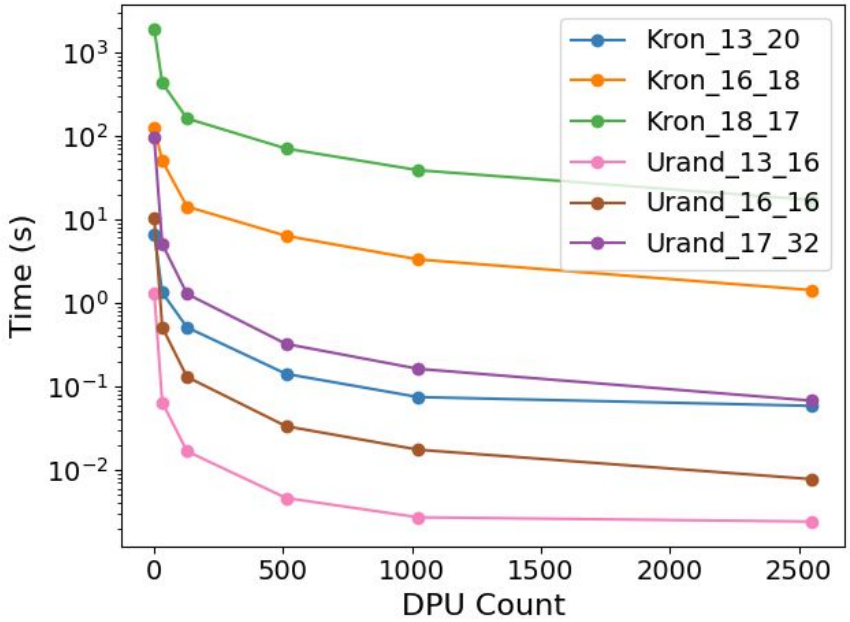
- **TriPIM Binary Search vs. Base**
  - *Base approach:* Compares elements of 1-hop and 2-hop adjacency lists; counts a triangle when elements match.
  - *Binary Search Approach:* Inserts the longer adjacency list into a binary search tree; probes with elements from the shorter list.
  - Binary search achieves speedup of
    - **7.3X** *on average* across all datasets
    - Up to a **51X** in the Kron_16_18



Results

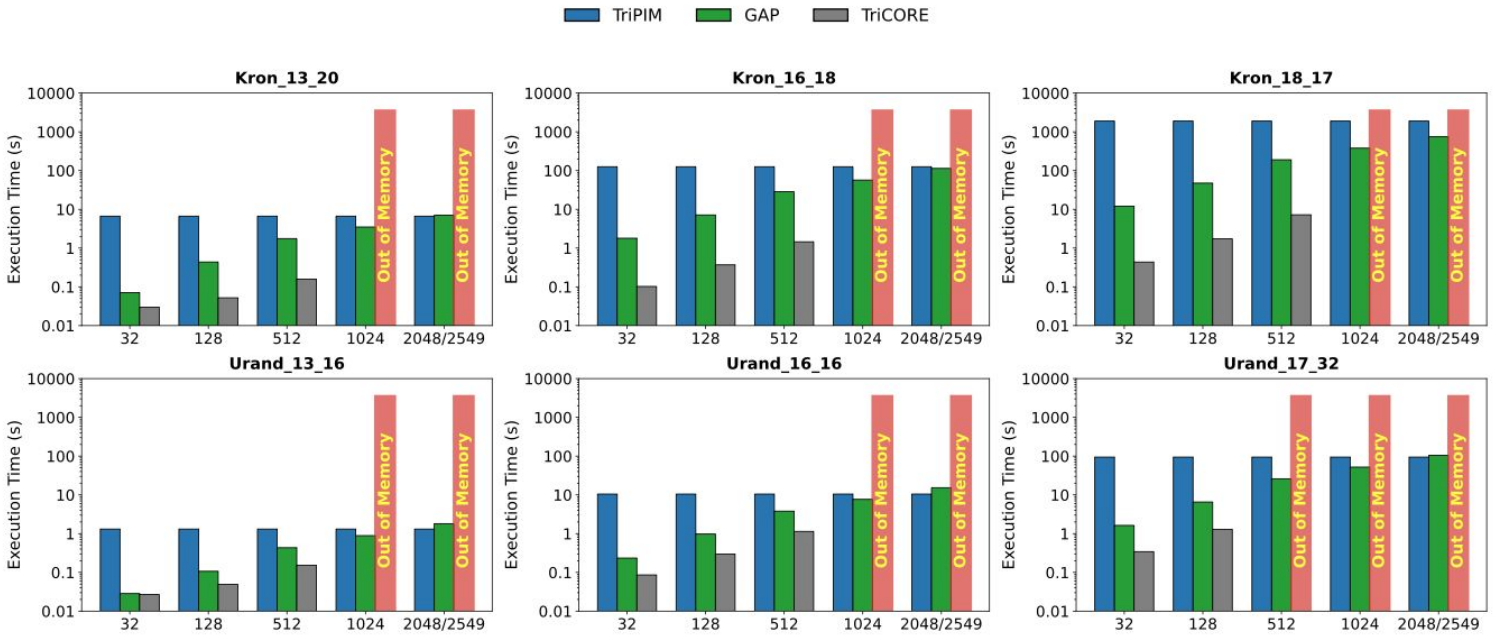
- **Scalability (small graphs):**
  - As the number of DPUs increases, execution time decreases
    - Exponential from 1 to 8 DPUs
    - Low DPU count ⇒ less parallelism, longer runtimes.
    - Each DPU handles more work, increasing runtime

*** high DPU counts are crucial to maintain TriPIM performance.**



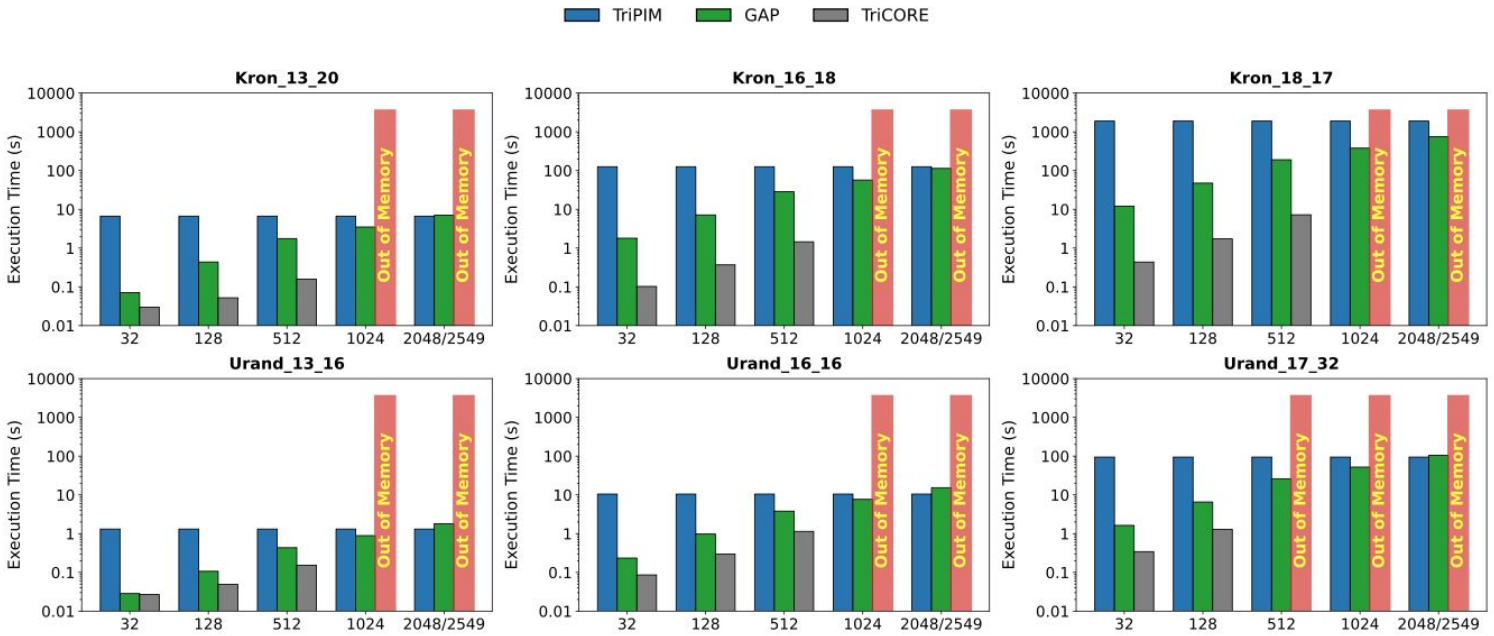
Results

- X-axis: factor by which base graph size is multiplied
- Measured only triangle counting kernel time (excluded setup, loading, and result collection).



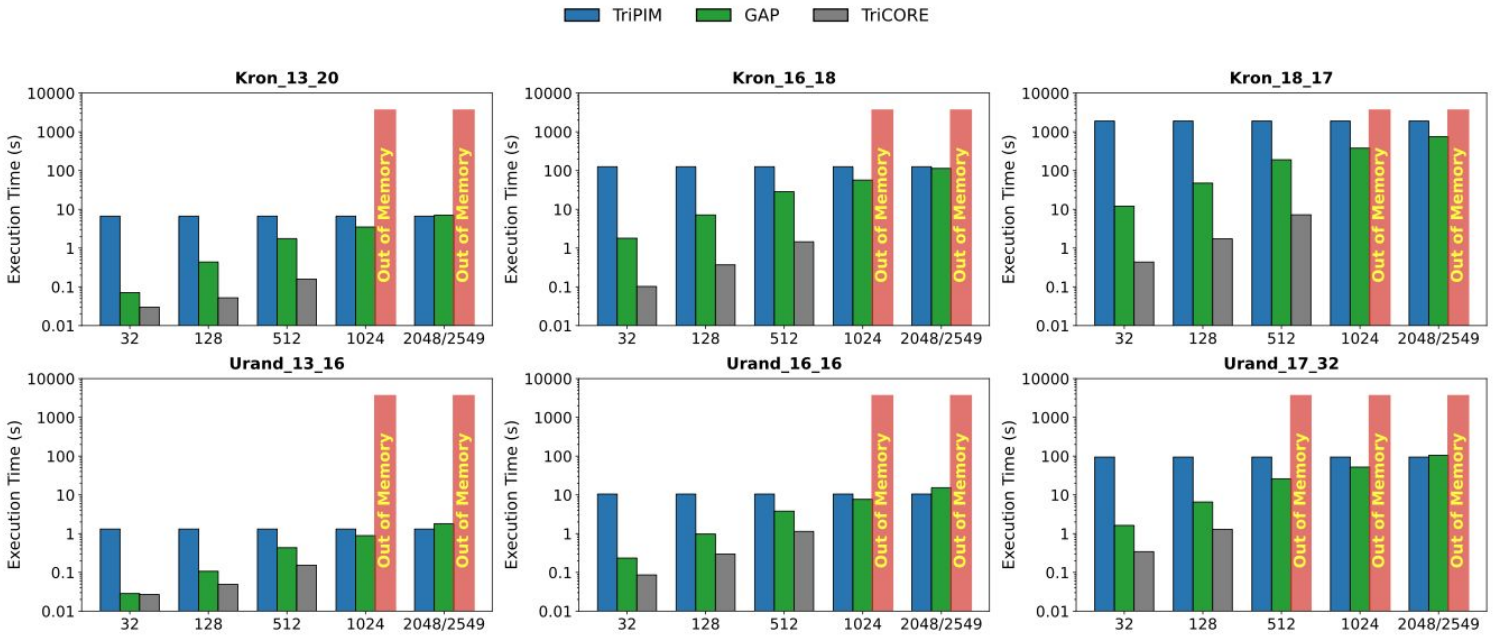
Results

- TriCORE fastest on small graphs (Size*32, *128) but fails beyond GPU memory limits.
- GAP > TriPIM for small graphs (better cache + higher CPU freq).



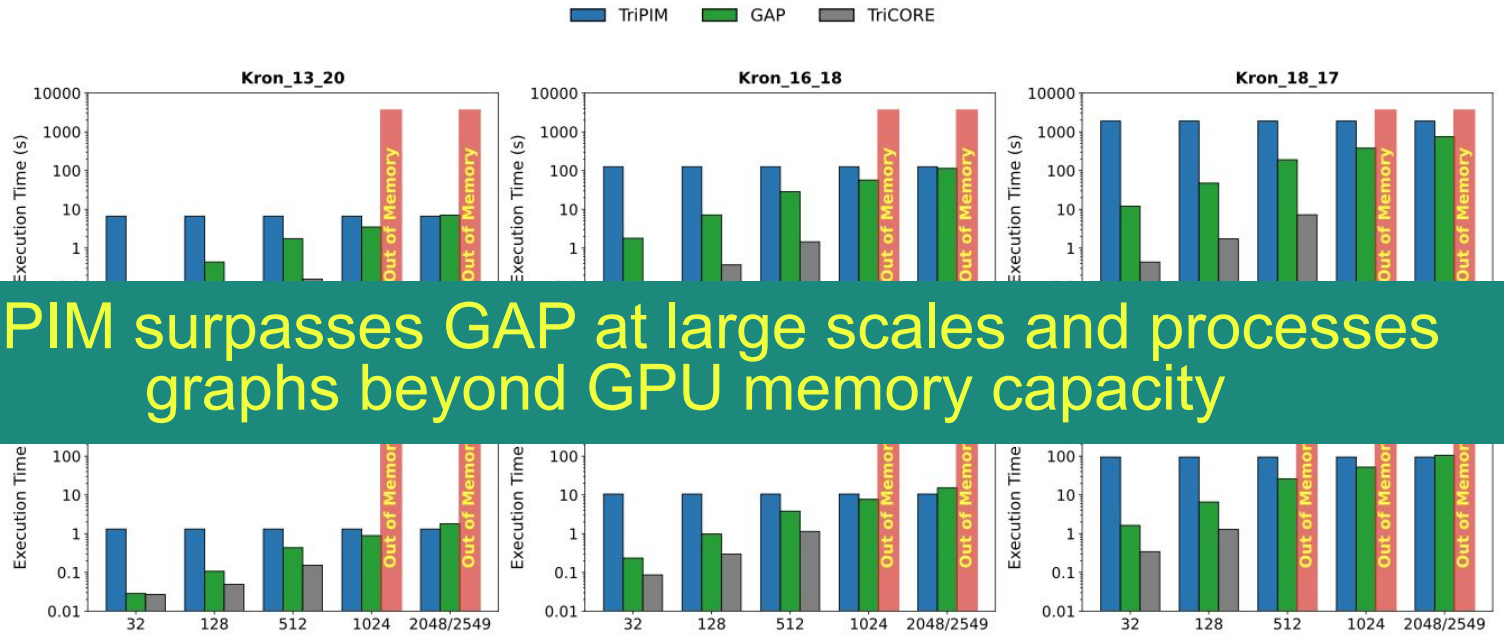
Results

- TriPIM constant runtime as graphs grow
  - CPUs suffer from cache misses
  - GPUs hit memory limits.



Results

- TriPIM constant runtime as graphs grow
  - CPUs suffer from cache misses
  - GPUs hit memory limits.



TriPIM surpasses GAP at large scales and processes graphs beyond GPU memory capacity

Conclusions

- Triangle counting is vital for graph analytics in social networks and AI, but CPU/GPU methods struggle to scale due to cache inefficiency and limited memory.
- TriPIM integrates TriCORE's binary search algorithm with UPMEM PIM to overcome these scalability barriers.
- Minimizes data movement, leverages thousands of DPUs, and keeps runtime nearly constant as graphs grow.
- Outperforms CPU/GPU on very large graphs and processes datasets beyond GPU memory capacity, though less effective on small graphs due to lower DPU frequency.

Future Directions

- Employing other intersection methods for the 1-hop and 2-hop intersection
- Using other graph partitioning methods such as bbTC to distributing nodes in a more balanced way to DPUs
- Explore tradeoffs under cost and power constraints.
- Evaluate TriPIM on other emerging PIM architectures to identify the most beneficial features for triangle counting.
- Explore the generalization of this approach to other applications, i.e., other graph algorithms.
- Investigate further optimizations for handling small graphs more efficiently within the PIM architecture.
- Experimenting with real-world graphs, such as Twitter

References

[1] Y. Hu, H. Liu, and H. Huang, ”TriCORE: Parallel Triangle Counting on GPUs,” *SC18: International Conference for High-Performance Computing, Networking, Storage and Analysis*, 2018, pp. 171-182.

[2] UPMEM. (2020). UPMEM Website. [Online]. Available: https://www.upmem.com

[3] S. Beamer, K. Asanovic and D. A. Patterson, ”The GAP benchmark suite,” arXiv:1508.03619, August 2015.

[4] S. Pandey, Z. Wang, S. Zhong, C. Tian, B. Zheng, X. Li, L. Li, A. Hoisie, C. Ding, D. Li, H. Liu, ”TRUST: Triangle Counting Reloaded on GPUs ,” arXiv:2103.08053.

[5] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, C. Faloutsos, and Z. Ghahramani, ”Kronecker Graphs: An Approach to Modeling Networks,” Journal of Machine Learning Research, vol. 11, pp. 985-1042, 2010.

[6] P. Erdos and A. R ´ enyi, ”On Random Graphs I,” Publicationes Mathematicae, vol. 6, pp. 290-297, 1959.

Q & A

GitHub