LSTC: Large-Scale Triangle Counting on Single GPU

Kishan Tamboli
Department of CSE
Indian Institute of Technology Bhilai
Bhilai, India
kishant@iitbhilai.ac.in

Vishwesh Jatala
Department of CSE
Indian Institute of Technology Bhilai
Bhilai, India
vishwesh@iitbhilai.ac.in

Abstract

Triangle counting in graphs has applications in a wide range of domains. For many years, researchers have been improving triangle counting performance by exploiting recent architectures, such as multicores and accelerators like GPUs. Improving the performance of triangle counting in GPU has several challenges: (1) GPUs are equipped very low memory, hence real-word large graphs can not be processed using the capacity of single GPU memory, (2) GPUs follow SIMD execution, whereas graphs exhibit irregular data parallelism, and (3) triangle counting incurs huge memory accesses, minimizing the number of global memory accesses is crucial for good performance.

To address these challenges, we propose Large-Scale Triangle Counting (LSTC), which can perform triangle counting on large graphs on a single GPU, even when they exceed the GPU's memory capacity. To achieve this, we first propose a novel workload partitioning scheme that partitions a large graph so that triangles can be counted using a single GPU without communication overhead. The proposed partitioning scheme reduces the number of duplicated edges and vertices across the partitions, which results in minimizing the memory footprint. Further, we propose a triangle-counting algorithm for each partition to improve performance by efficiently exploiting GPU architectural resources.

We evaluated LSTC on a wide range of graph datasets. We achieve not only an average speed up of 2.8× on large datasets when compared it with the state-of-the-art multi-GPU implementations but also require fewer resources for computation.

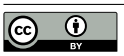
CCS Concepts

• **Computing methodologies** → **Massively parallel algorithms**.

Keywords

Graphics Processing Units, Triangle Counting, Workload Partition

ACM Reference Format:
Kishan Tamboli and Vishwesh Jatala. 2026. LSTC: Large-Scale Triangle Counting on Single GPU. In *Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE '26), May 04–08, 2026, Florence, Italy*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3777884.3797002



ICPE '26, Florence, Italy

ACM ISBN 979-8-4007-2325-4/2026/05
https://doi.org/10.1145/3777884.3797002

1 Introduction

Most of the real-world data can be represented using graph data structures. With a massive growth in graph data, extracting patterns in the graph has become important and has found applications in many domains. For instance, computing the number of triangles in graphs has a wide range of applications, including social network analysis [37], anomaly detection [24], spam filtering [5], etc. Triangle counting is also used in computing graph statistics, such as clustering coefficient [38] and k-truss [7, 34]. Many extensions of triangle counting have also been used in several applications, such as finding cliques [41].

Due to the growing significance of the triangle counting problem, Graph-Challenge [13] competition seeks innovations in improving the performance of triangle counting. Over the last few years, many techniques have been proposed to improve triangle counting performance. Most of the techniques proposed in the literature are majorly classified into three categories: subgraph matching [23], list intersection [19, 29], and matrix multiplication [29, 36]. Out of these, the intersection-based method has become popular and has been implemented using three techniques: merge path, binary search, and hashing [29]. With the rapid growth in graph data, researchers have been exploiting advances in processor architectures to improve performance. For instance, many implementations were proposed on emerging architectures, such as multicores and accelerators. Recently, GPUs, due to their high throughput, have become an attractive platform for triangle counting.

Several techniques [16, 18–20, 29, 36] exploit GPUs to speed up the triangle counting. However, they have limitations in several dimensions. A few techniques [13, 16, 18, 19] propose distributed implementations to support large graphs. However, they suffer from communication overhead, consume significant memory to store the edge list representation of the graph or require duplicating more vertices and edges across partitions. In addition, they require expensive hardware for computation. Many techniques presented in the literature [14, 28, 35, 36] do not support large graphs and do not exploit the GPU hardware resources efficiently to improve the performance on a single GPU.

To address these challenges, we propose Large-Scale Triangle Counting (LSTC), which can perform triangle counting on large graphs using a single GPU, even though they may not fit into the capacity of the single GPU memory. We first propose a workload partition technique that partitions an input graph such that (1) it can support any graph partitioning algorithm that partitions the vertices into disjoint sets, (2) the triangle counting on large graphs can be computed only using a single GPU, (3) it avoids the communication overhead during the triangle counting phase, and (4) it requires fewer number of duplicated edges and vertices across partitions and hence minimizing the number of memory accesses.

Next, we propose an efficient triangle counting algorithm to compute the triangles on each partitioned graph on a single GPU. The algorithm utilizes the GPU resources efficiently by minimizing the number of global memory accesses, managing threads efficiently, and exploiting data locality.

We evaluated LSTC on a wide range of graph datasets and compared it with two state-of-the-art GPU implementations, We achieve an average speedup of 2.8× on large datasets when compared it with the state-of-the-art multi-GPU implementations. Further, our approach achieves an average speedup of 3.49× on the graphs that fit the single GPU, compared to state-of-the-art GPU implementations.

To summarize, this paper makes the following contributions.

(1) We propose a workload partitioning technique to count triangles on very large graphs using a single GPU memory.
(2) We propose a triangle counting algorithm for each partition on the GPU that aims to improve performance by exploiting the GPU resources efficiently.
(3) We experimentally validate our approach on a wide range of graphs. We observe that LSTC not only achieves superior performance when compared with state-of-the-art multi-GPU implementations but also requires fewer resources for computation.

The rest of the paper is organized as follows. We briefly describe the background of GPUs and triangle counting in Section 2. We discuss the related work in Section 3. The details of our approach are described in Section 4. Sections 5 and 6 present the experimental evaluation. We conclude the paper in Section 7.

2 Background

This section briefly discusses background on GPUs and the triangle counting problem on single and distributed machines.

2.1 Graphics Processing Units

A typical NVIDIA GPU consists of a set of streaming multiprocessors (SMs), where each SM maintains several resources, such as registers, ALUs, shared memory, L1 cache, etc. GPUs are also equipped with L2 cache and global memory, which can be accessed from any SM. Several parallel programming interfaces, such as CUDA [25] and OpenCL [26], exist to parallelize programs on GPUs. The portion of code to be parallelized on GPU is invoked with a special function called *kernel*. The kernel is launched with a specified number of thread blocks and a specific number of threads within each thread block.

In GPUs, global memory has long access latency, which is a primary reason for the performance bottleneck in the memory-bounded applications. Programmers can reduce or hide the latency by (1) exploiting shared memory, (2) using massive thread-level parallelism, or (3) using coalesced access to global memory. Unlike global memory, shared memory is a fast storage unit but has a limited size.

2.2 Triangle Counting

The most popular method to compute the triangle counting is list intersection. Consider an undirected graph $G(V, E)$ having V vertices and E edges. Triangle counting can be computed by iterating over each edge ($u \rightarrow v$) of a graph and finding the common neighbors of it, i.e., computing $|N(u) \bigcap N(v)|$, where $N(u)$ and $N(v)$ are the neighborhood of u and v, respectively. If w is a common neighbour of u and v, then uvw forms a triangle. To avoid repeated counting of triangles, the triangle counting algorithms perform a pre-processing step, which converts an undirected graph to a directed graph, using any heuristic.

To count triangles on a graph that do not fit a single machine's memory, the input graph is partitioned among multiple compute nodes/machines and loaded on CPU/GPU memory. Each compute node performs the triangle counting on its own portion of the graph and performs synchronization to count the triangles which are split among the machines. The final triangle count is obtained by aggregating the local triangles computed on each machine.

3 Related Work

With advances in processor technologies and rapid growth in the size of the graphs, researchers have been trying to accelerate triangle counting by exploiting the latest hardware. Many approaches are implemented using shared-memory CPUs, single GPUs, and distributed systems. Recently, the Static Graph Challenge [13] sought innovative solutions to accelerate triangle counting of large graphs using emerging platforms. This section briefly describes several triangle counting techniques implemented on various platforms.

3.1 Multicore Implementations

Several approaches [32, 33, 42] exploit various resources of multicore processor architectures to improve performance. For instance, Shun et al. [32] design parallel implementations of merge-based and hash-based intersection by exploiting dynamic multithreading and the memory hierarchy. FESIA [42] leverages SIMD instructions to accelerate set intersection. It filters unmatched elements using bitmaps and uses kernels to compute the intersection. Tangwongsan et al. [33] propose a coordinated bulk-parallel algorithm to compute approximate triangles in streaming graphs on multi-core processors. Their ideas are shown to perform better due to bulk-edge processing and improved cache efficiency.

3.2 Single-GPU Implementations

As with CPUs, several algorithms have been proposed that exploit GPU resources and architecture. An evaluation [36] of various triangle counting algorithms, such as subgraph matching, programmable graph analytics, with a set intersection, and matrix-multiplication, is conducted on GPUs. They observed these techniques perform well in GPUs compared to CPUs due to their high throughput. Moreover, they observe that the graph analytical approach performs better due to efficient workload management. Similarly, many list intersection methods [4] are implemented on GPUs using merge-path [15], binary search [7, 16, 18, 19], and bitmap [6] techniques. Tricore [19] shows that binary search performs better than the merge-path technique due to coalesced access to global memory and cache hits. Recently, TRUST [29] proposes a vertex-centric hashing-based implementation on GPUs. It reorders the vertices based on their degrees to reduce the computation overhead with hashing collisions. Hu et al. [17] propose lightweight graph preprocessing methods to accelerate GPU triangle counting algorithms without affecting

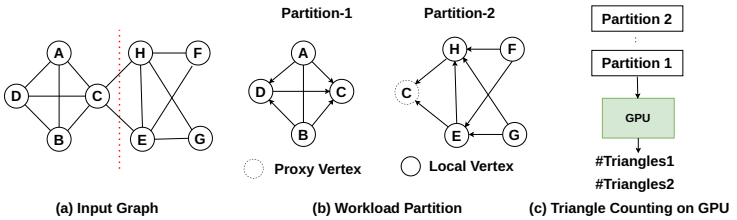


Figure 1: Overview of LSTC

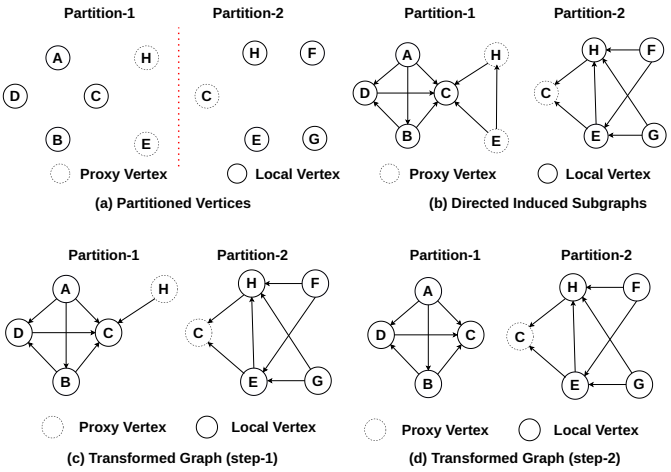


Figure 2: Workload partitioning

their implementations. STMatch [39] and VSGM [21] propose techniques to accelerate subgraph matching using a GPU. Fast triangle counting algorithm [15] exploits the GPU shared memory to reduce the memory accesses; however, they do not support graphs whose neighbours exceed the size of the shared memory. Aljundi et al. [3] exploit GPU shared memory to accelerate the computation of Jaccard weights, which is a metric to compute the similarity of two sets and can be used for triangle counting. Our approach differs from these techniques, as they cannot handle graphs on a single GPU if the input graphs do not fit in the GPU's memory. Moreover, even if the neighbours do not fit the capacity of shared memory, our approach can process them using the available shared memory.

3.3 Distributed Implementations

Due to the growing size of graph data, research on developing efficient distributed triangle-counting algorithms has gained momentum. Recently, many techniques were proposed in the graph challenge [10, 11, 16, 30] for distributed architectures. DistTC [16] supports distributed multi-GPU triangle counting. They extend the outgoing-edge-cut partitioning scheme to avoid communication overhead. In contrast, our approach supports any arbitrary partitioning scheme and utilizes shared memory to reduce global memory accesses. Block-based [40] triangle counting algorithm aims to handle large graphs using heterogeneous architectures. Tric [11] exploits the graph structure to improve the performance of triangle counting in the distributed setting. However, it suffers from the synchronization overhead. Tricore [19] and TC-Stream [20] address large graphs, but they suffer from memory overhead due to their edge list representation. PDTL [12] supports distributed triangle counting, but its focus is on external memory. Trust [29] also supports triangle counting on distributed machines. However, it suffers significantly from vertex reordering. To summarize, many distributed solutions are proposed for triangle counting; however, they suffer from communication overhead or do not exploit the hardware resources efficiently. Moreover, they require huge hardware to support large-scale graphs.

4 Large Scale Triangle Counting

LSTC aims to compute the number of triangles in a large graph using a single GPU, even though it exceeds the GPU's memory capacity. Figure 1 illustrates the overview of LSTC. First, we propose a workload partitioning technique to partition a given input graph (listed in Figure 1(a)) into multiple partitions (shown in Figure 1(b)) such that triangles can be counted on each partition independently,

Algorithm 1 Workload Partitioning

1: **function** WorkloadPartition(Graph $G(V, E)$)
2: $\mathcal{G} = \emptyset$
3: $\{V_1, V_2 \dots V_n\} \leftarrow Partition(G)$
▷ $\bigcup_{i=1}^{n} V_i = V$ and $\forall_{ij}\ V_i \bigcap V_j = \emptyset$, where $i \neq j$
4: **for** i in $\{1..n\}$ **do**
5: $V_i' = PartitionsWithProxy(G, V_i)$
6: $G_{dir} \leftarrow DirectedInducedSubgraph(V_i', G)$
7: $G_{part} \leftarrow TransformGraph(G_{dir})$
8: $\mathcal{G} = \mathcal{G} \bigcup G_{part}$
9: **end for**
10: **Return**$\mathcal{G}$
11: **end Function**

without requiring communication. Moreover, it reduces the number of duplicate edges among the partitions. Then, we propose an efficient triangle counting algorithm to compute the triangles of each partition on a single GPU, as shown in Figure 1(c). Finally, we aggregate the triangle counts obtained from each partition to compute the total number of triangles.

The rest of the section is organized as follows. Section 4.1 describes details of our workload partitioning method, Section 4.2 discuss a method for triangle counting on the partitioned graphs, Section 4.3 describes the proof of correctness, and Section 4.4 explains the triangle counting algorithm, which has been optimized to utilize the GPU resources efficiently.

4.1 Workload Partitioning

4.1.1 ***Workload Partition.*** Our workload partitioning strategy can be integrated with any vertex-based graph partitioning algorithm that divides a graph into disjoint vertex sets. Algorithm 1 outlines the procedure to partition large graphs for triangle counting while minimizing communication and duplication of vertices and edges.

Algorithm 2 Graph Transformation

```
1:  function TransformGraph(Graph G(V, E))
2:  H = { h | h is a proxy and indegree(h) = 0}
3:  if |H| <= 0 then
4:     return G
5:  else
6:     for h in H do
7:        delete h from V
8:        delete h → u from E for each u ∈ V
9:     end for
10:    Let G' (V', E') be the resultant graph
11:    return TransformGraph(G'(V', E'))
12: end if
13: end function
```

We partition the input graph $G(V, E)$ into n disjoint sets $V_1, V_2, \ldots, V_n$, where $\bigcup_{i=1}^{n} V_i = V$ and $V_i \cap V_j = \varnothing$ for $i \neq j$ (Line 3). Any partitioning scheme can be used; in this work, we adopt random partitioning. Each partition holds a set of local vertices and introduces proxy vertices for neighbours that reside in other partitions (Line 5), following prior frameworks [8, 9, 27]. Figure 2(a) illustrates partitioned vertices and their proxies derived from Figure 1(a).

To ensure no triangles are missed, each partition constructs an induced subgraph (Line 6) containing all edges between vertices within that partition. This guarantees that if a triangle *ABC* exists in the original graph, it will appear in the induced subgraph containing *A*, *B*, and *C*. The induced graph is undirected and is converted into a directed graph to prevent duplicate triangle counts: for edge u, v, we keep $u \rightarrow v$ if $\deg(u) < \deg(v)$, or if degrees are equal and $u < v$. Figure 2(b) shows the resulting directed subgraphs.

However, this process may duplicate vertices and edges across partitions [16], as seen in Figure 2(b), where vertices *E*, *H*, and *C* and their edges appear in multiple partitions. These duplicates increase memory and computation overhead.

To mitigate this, we transform each induced subgraph to remove redundant vertices and edges without losing triangles. Each partition contains local and proxy vertices; our triangle counting algorithm (Sections 4.2 and 4.4) processes triangles originating from local vertices only. A proxy vertex with zero in-degree cannot contribute to local triangles, since its triangles are already counted in the partition where it is local. Such vertices and their outgoing edges are redundant and can be safely removed (proof in Section 4.3).

Algorithm 2 describes this transformation. Line 2 identifies zero-in-degree proxy vertices, Lines 6-9 remove them and their outgoing edges, and Line 11 repeats the process until no further deletions are possible. Figures 2(c)-(d) illustrate the transformation. For example, triangle *ECH* is detected in Partition-2 where *E* is local; hence, *E* and its edges can be removed from Partition-1. The same applies to *H*, leading to the final optimized subgraph in Figure 2(d).

4.1.2 ***Time Complexity.*** For a given graph $G(V, E)$ and p partitions, the time complexity of the Algorithm 2 is $O(pE)$. The graph transformation algorithm, in the worst case, needs to visit each proxy vertex and its neighbours, resulting in O(E) complexity. Since this process must be repeated for each directed induced subgraph, it incurs $O(pE)$ overhead.

4.1.3 ***Graph Size Limit.*** The workload partition requires to store the original graph *G* in the CSR format in CPU memory. The graph *G* needs to be partitioned into several partitions such that each partition can fit the capacity of the GPU memory. Since we process each partition at once, it is sufficient if the partitioned graph, which has the maximum size, fits the CPU memory along with the original graph. In addition, we require $O(V)$ memory to store the metadata required to compute the directed graph from the undirected graph. Hence, the proposed workload partitioning algorithm can support a large graph *G* as long as CPU RAM can fit the following (a) size of the *G* in the CSR representation, (b) partitioned graph G_p in CSR format such that G_p is the maximum size across all partitioned graphs of *G* where G_p fits GPU memory, and (c) $O(V)$ metadata (required to compute directed graph from undirected graph).

4.2 Triangle Counting on Partitioned Graphs

Once the directed subgraphs are created, we perform the triangle counting on each subgraph at a time on a single GPU. Section 4.1.1 ensures that no triangles are lost during the partition. Hence, the triangle can be identified within the partition without requiring communication. However, a triangle can occur in multiple partitions even after the graph transformation removes some redundant vertices and edges. To ensure that each triangle is counted exactly once, we count a triangle ABC (with edges $A \rightarrow B$, $A \rightarrow C$, and $B \rightarrow C$) in the partition by processing vertex *A* in the partition only if it is a local vertex. Thus, we get the total triangles in the partition by processing the local vertices of the partition. Once we compute the number of triangles in each partition, we can get the total triangle by aggregating the local triangles obtained from each subgraph. We briefly present the proof of correctness below.

4.3 Proof of Correctness

Claim: Every triangle in the input graph is counted exactly once.

Assume that the original graph has a triangle *ABC* with edges $A \rightarrow B$, $B \rightarrow C$, and $A \rightarrow C$.

Case-1: If all the vertices are local to a partition (P_i), then the triangle is identified only by the partition P_i while processing the vertex *A*. It is because all the edges of the triangle are present in P_i. Even if the triangle occurs in another partition, say P_j, it is not counted by the partition as the vertex *A* is a proxy vertex in P_j.

Case-2: If the vertices *A*, *B*, *C* span across three partitions, which are local to P_i, P_j, and P_k, respectively, then triangle is counted only by partition P_i. This is because P_i contains vertex *A* as the local node, and *ABC* is counted while traversing the edge $A \rightarrow B$. Because the induced subgraph correspond to P_i contains all the edges $A \rightarrow B$, $A \rightarrow C$, and $B \rightarrow C$. Algorithm 2 does not remove the vertices of the proxy vertices *B* and *C* in the partition, as the indegree of *B* and *C* is non-zero. Hence, the triangle is indeed identified by the partition P_i. Moreover, the triangle *ABC* can not be identified as any partition P_j as the vertex *A* is a proxy vertex to P_j.

Cases 1 and 2 show that if a triangle is present in the original graph, it is counted exactly once.

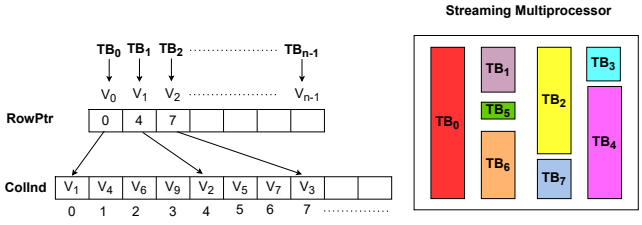


Figure 3: Thread block management

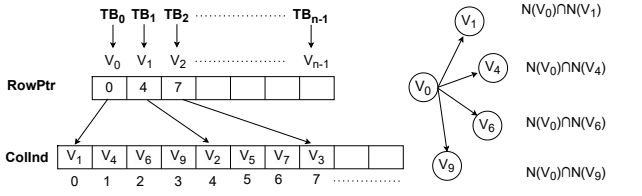


Figure 4: Exploiting shared memory

4.4 Efficient Triangle Counting on Single GPU

Once the partitions are created (as described in Section 4.1.1), we need to compute the number of triangles in each partition efficiently. Section 4.2 ensures that processing the local vertices along with the edges of the partition ensures triangles are identified exactly once. However, efficient triangle in the GPU is still a challenge as (1) the graphs are irregular in nature and GPUs follow SIMD execution, (2) triangle counting incurs a large number of global memory accesses, and (2) suffers from load imbalance, etc. To address these challenges, we propose an efficient triangle algorithm such that (1) it manages the thread blocks for better resource utilization, (2) it minimizes global memory accesses by leveraging shared memory, (3) it exploits data locality through efficient thread management. We describe these techniques below.

4.4.1 ***Thread Block Management.*** Like many graph analytical applications, triangle counting suffers from an irregular vertex-degree distribution. Hence, efficient workload management is crucial for better resource utilization and performance. Most GPU-based triangle counting algorithms [19, 31] use edge list representation to process the input graph to ensure workload balance. However, the edge list representation consumes significant memory and still suffers from load imbalance, as the quantum of work per edge varies. A few techniques use CSR representation to address memory issues and assign a vertex to a thread or a warp result. However, they result in an imbalance in thread workload. Moreover, it suffers from thread (or warp) divergence, causing its thread block to acquire the GPU resources without utilizing them. To overcome this, we assign a thread block TB_i to compute the number of triangles associated with edges whose source vertex is V_i. Figure 3 shows an example of thread block distribution to the vertices (corresponding to RowPtr in CSR representation).

When a thread block finishes processing a vertex, it releases the resources from the streaming multiprocessor, and another thread block that is not yet scheduled can start executing by acquiring those resources. Thus, achieving better resource utilization. Figure 3 illustrates an example of thread block execution in an SM. When TB_3 finishes earlier than other thread blocks, TB_4 gets launched and starts processing V_4. This demonstrates that even though thread blocks can have varying workloads, GPU resources are better utilized.

4.4.2 ***Reducing Global Memory Accesses.*** Each thread block TB_i processes its assigned vertex V_i and computes $N(V_i) \cap N(V_j)$, where $N(V_i)$ and $N(V_j)$ are the neighborhood of V_i and V_j respectively and $V_j \in N(V_i)$. When TB_i computes the intersection corresponding to each edge $V_i \rightarrow V_j$, it frequently accesses elements of $N(V_i)$. To illustrate this, consider an example shown in Figure 4. TB_0 has repeated access to $N(V_0)$ to compute the triangles associated with the edges $V_0 \rightarrow V_1$, $V_0 \rightarrow V_4$, $V_0 \rightarrow V_6$, and $V_0 \rightarrow V_9$.

Since the CSR representation of an input graph is stored in global memory, each thread block accesses the neighbours from global memory. Since global memory has huge access latency, the performance of the application degrades significantly. Few techniques, such as [15], exploit shared memory to store the neighbours; however, since the capacity of shared memory is limited, it cannot support graphs with a large number of neighbours. For instance, the NVIDIA V100 GPU has up to 96 KB of shared memory, whereas the global memory has 16 GB. To support vertices that have neighbourhood size exceeding the size of shared memory, we tile neighbourhood vertices. Figure 5 shows an example of neighbourhood tiling, where the neighbourhood of V_0 is tiled such that each tile can fit into the shared memory of GPU. Each thread block TB_i first loads a tile of $N(V_i)$ into shared memory and completes all the neighbourhood intersections associated with the tile before processing the next tile. This ensures each tile gets the maximum benefit from shared memory and avoids redundant transfers of tiles from global memory to shared memory.

4.4.3 ***Improving Data Locality with Efficient Thread Management.*** Work distribution to the threads within a thread block has an impact of the performance. Consider a workload distribution that assigns contiguous neighbours of a vertex to contiguous threads as shown in Figure 6(a). Thread block TB_0 processes V_0 and visits 4 neighbors V_1, V_4, V_6, V_9 to compute their respective neighborhood intersections. This is achieved by assigning successive threads to successive neighbours of V_0. Such a distribution results in a poor locality and an uncoalesced memory access pattern, as neighbours of V_1 and V_4 can be far apart in global memory. It also leads to poor workload imbalance among the threads, since the neighbourhood sizes of each vertex differ.

To improve the data locality, we assign the threads to access the neighbourhood elements of a vertex in a round-robin fashion. Figure 6(b) shows an example of our threads distribution scheme. Here, successive threads T_0 to T_3 access successive elements of $N(V_1)$ in the global memory, exploiting data locality and coalesced memory. Each thread in the thread block TB_i searches its assigned element in $N(V_i)$ using the binary search. If the element is found, the number of triangles are incremented.

Algorithm 3 shows our triangle counting algorithm on the GPU, combining the optimizations discussed in Section 4.4.1, 4.4.2, and 4.4.3. A vertex is processed by the thread block only if it is a local vertex (as shown in condition at Line 11). At Line 14, a thread block (id is denoted as *bid*) stores a tile of the neighbourhood of the vertex (*bid*) into shared memory (*nebSrc*). Subsequently, the

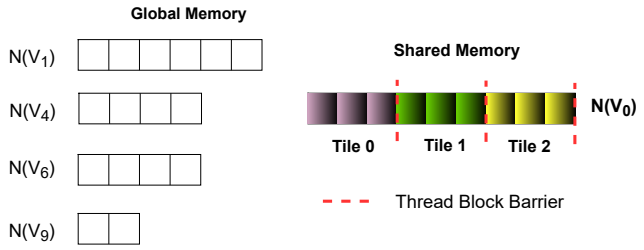


Figure 5: Neighbourhood tiling

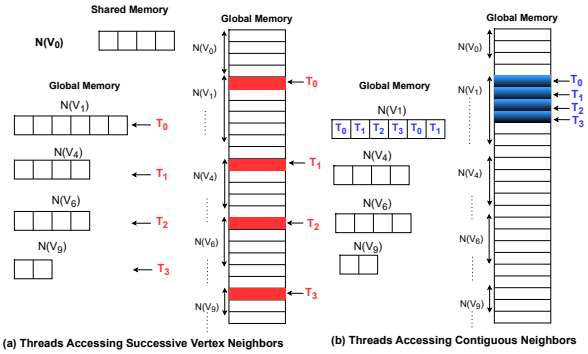


Figure 6: Workload distribution

threads of the thread block process the elements of $N(V_j)$ where V_j is neighbour of $N(bid)$ in the round robin fashion, as shown in Line 16 to Line 26. Each thread performs a binary search at Line 23; if the element is found in the source neighbour list (*nebSrc*), then the triangle count is incremented at Line 24.

5 Experimental Setup

We evaluated our approach on a wide range of input graphs. Table 1 shows the input graphs along with their properties used for evaluation. We grouped the input graphs into medium and large datasets. We use medium size graphs to evaluate the performance of triangle counting on single-GPU without using workload partitioning. Similarly, we use large graphs to evaluate the performance of triangle counting when used with workload partitioning. Table 1 reports the number of the graphs after removing the self-loops and isolated vertices.

We evaluated our approach on two different hardware configurations. The first hardware consists of a cluster having multiple compute nodes each with 2 NVIDIA V100 GPUs with 16 GB RAM. For LSTC evaluation, we used only a single GPU throughout the paper. However, to compare our solution with other state-of-the-art distributed implementations, we are required to run their applications on multiple GPUs. The second hardware unit consists of a machine equipped with a single NVIDIA RTX A6000 GPU with 48GB of memory. We use this hardware to verify the effectiveness of our implementation of larger datasets when without using workload partition.

We compile our implementation using CUDA 11.7 [25] and GCC 8.3. We report the kernel execution time of GPU triangle counting. We compare our implementation with Tricore [19] and Trust [29]

Algorithm 3 Large Scale Triangle Counting

```
1:  function LSTC_Kernel(Graph G)
2:  bid ← blockIdx.x          ▷ Block bid processes the vertex bid
3:  tid ← threadIdx.x
4:  triangles ← 0
5:  TILE_WIDTH ← blockDim.x
6:  __shared__ nebSrc[TILE_WIDTH]
7:  start ← G.RowPtr[bid]
8:  end ← G.RowPtr[bid + 1] − 1
9:  n_tiles ← end − start/N
10: N ← blockDim.x            ▷ Number of threads in a thread block
11: if isLocalNode(bid) then
12:   for i in {0..n_tiles − 1} do
13:     index ← i ∗ TILE_WIDTH + tid
14:     nebSrc[tid] ← G.ColInd[start + index]
15:     __syncthreads()
16:     for dst in {start..end} do
17:       base ← G.ColInd[dst]
18:       nebDstStart ← G.RowPtr[base]
19:       nebDstEnd ← G.RowPtr[base + 1] − 1
20:       for k in {nebDstStart..nebDstEnd − 1} do
21:         index ← nebDstStart + k ∗ N + tid
22:         key ← G.ColInd[index]
23:         result ← BinarySearch(nebSrc, key)
24:         triangles ← triangles + result
25:       end for
26:     end for
27:   end for
28:   s_sum[tid] = triangles;
29:   __syncthreads()
30:   if tid==0 then
31:     Aggregate s_sum to g_sum[bid]
32:   end if
33: end if
34: end function
```

frameworks. Tricore is a triangle counting implementation based on the binary search, and Trust is an optimized implementation based on hashing. We report results on an average of 3 runs.

6 Experimental Results

This section analyzes the effectiveness of our proposed techniques. We first analyze the performance of triangle counting on large graphs when using workload partitioning. Subsequently, we evaluate the GPU triangle counting algorithm (discussed in Section 4.4) on medium size graphs without using workload partitioning. This evaluation helps to analyze the effectiveness of optimizations of triangle counting on GPU.

6.1 Triangle Counting with Workload Partitioning

6.1.1 ***Comparison with state-of-the-art.*** Table 2 compares the performance of LSTC with two state-of-the-art implementations, Tricore and TRUST, on large datasets. Since the datasets are huge, Tricore and TRUST require multiple GPUs to perform the computation. Tricore and TRUST require 2 GPUs for processing webbase, uk-2005, and it-2004 datasets, whereas LSTC, using our workload partitioning technique, will help in processing them on single GPU

Table 1: Graph datasets and their key properties

	Dataset	*No. of Vertices*	*No. of Edges*	*No. of Triangles*
Medium Dataset	roadNet-CA (RN-CA)	1,965,206	2,766,607	120,676
	cit-Patents (cit)	3,774,768	16,518,947	7,515,023
	graph500-scale21 (scale21)	1,243,073	31,731,650	935,100,883
	graph500-scale22 (scale22)	2,396,657	64,097,004	2,067,392,370
	graph500-scale23 (scale23)	4,606,314	129,250,705	4,549,133,002
Large Dataset	Wikipedia-link (wikipedia)	13,593,032	334,591,525	13,540,543,134
	Webbase-2001 (webbase)	115,554,441	854,809,761	12,262,060,053
	uk-2005 (uk2005)	39,454,463	783,027,125	21,779,366,056
	it-2004 (it2004)	41,290,648	1,027,474,947	48,374,551,054
	twitter	41,652,230	1,202,513,046	34,824,916,864
	friendster	65,608,366	1,806,067,135	4,173,724,142

Table 2: Comparing total time (workload partition and kernel execution time in seconds) of "LSTC", "Tricore" and "Trust" on large datasets. LSTC is executed on a single GPU whereas Tricore and TRUST are evaluated on 2 GPUs for webbase, uk-2005, and it-2004 and 4 GPUs for twitter and friendster. For TRUST, the reorder time listed in shown in brackets. For LSTC, graph transformation overhead is listed in brackets.

	Trust			*Tricore*			*LSTC*		
Dataset	***Workload Partition***	***kernel (reorder)***	***Total Time***	***Workload Partition***	***kernel***	***Total Time***	***Workload Partition (overhead)***	***kernel***	***Total Time***
webbase	550.43	51.22 (51.15)	601.65	117.75	3.26	121.02	32.50 (6.46)	1.43	**33.94**
it-2004	245.62	30.85 (30.75)	276.47	98.26	5.18	103.45	74.79 (8.40)	5.18	**79.98**
twitter	347.49	56.50 (56.41)	403.99	182.21	7.29	189.51	78.22 (3.38)	16.84	**95.07**
uk-2005	234.36	29.07 (29.01)	263.44	191.56	3.13	194.70	31.10 (1.06)	1.45	**32.56**
frienster	655.32	123.45 (123.36)	778.77	424.60	6.81	431.42	122.54 (10.04)	6.19	**128.73**

memory. Similarly, friendster and twitter being very large require 4 GPUs for evaluation for Tricore and TRUST, but LSTC can process them on single GPU memory.

The table reports both workload partitioning time and the GPU kernel execution time for all the approaches. The table shows that LSTC outperforms Trust and Tricore for all the large datasets for both workload partition and kernel execution times. We achieve an average speed up 2.8× when compared to Tricore and 6.62× when compared to TRUST on the total execution time. The results clearly demonstrate that LSTC not only requires fewer GPUs for processing but also performs better than the state-of-the-art techniques.

Both Tricore and TRUST perform workload partition to process the large graphs, but our workload partitioning method partitions the graphs such that triangle counting on each partitioned graph can be computed independently without incurring any communication overhead. Twitter does not perform better on kernel execution when compared to tricore, because twitter has relatively dense connections compared to the rest of the graphs, which results in duplicating several edges across partitions even after removing many redundant proxy vertices using Algorithm 2. Table 2 also shows that LSTC performs better than TRUST for all input graphs for both workload partition and kernel execution. TRUST reorders the vertices to improve the performance of triangle counting, which consumes a significant amount of time. To quantify this, we report the reorder time (in brackets) for each dataset in the table.

To summarize, LSTC not only achieves superior performance when compared with state-of-the-art implementations but also requires fewer resources for computation. Thus making it overall effective.

6.1.2 *Effectiveness of Optimizations.*

Reduction in the number of redundant vertices and edges: Table 3 and 4 shows effectiveness of our graph transformation approach described in Algorithm 2. Table 3 reports the number of vertices in the induced subgraph (denoted as V_{ind}) and the number of vertices in the transformed graph (denoted as V_{res}) after applying Algorithm 2. Similarly, Table 4 reports the number of edges in the induced subgraph (denoted as E_{ind}) and the number of edges in the transformed graph (denoted as E_{ind}) after applying Algorithm 2. The results are reported for each of the partitions. In addition, we report the impact on memory footprints corresponding to the number of vertices and edges.

These results indicate our approach is effective in removing the redundant vertices and edges for all partitions across all datasets. For instance, consider the twitter dataset; the total number of vertices across all partitions has decreased from 147.769 million to 85.928 million after transformation. That indicates a significant reduction of about 41.87%. Similarly, the number of edges reduced from 4.74 billion to 3.87 billion, representing a reduction of around 18.25%. The corresponding memory footprint decreased from 36.2 GB to 29.59 GB. We can observe similar trends in other datasets as well. These results demonstrate the effectiveness of our approach in

Table 3: Number vertices (in Millions) before and after removing redundant from the induced subgraph. Memory in GB.

Dataset	*Partition 0*		*Partition 1*		*Partition 2*		*Partition 3*		*Total*		Memory footprint	
	$\|V_{ind}\|$	$\|V_{res}\|$	$\|V_{ind}\|$	$\|V_{res}\|$	$\|V_{ind}\|$	$\|V_{res}\|$	$\|V_{ind}\|$	$\|V_{res}\|$	$\|V_{ind}\|$	$\|V_{res}\|$	$\|V_{ind}\|$	$\|V_{res}\|$
webbase	87.8	55.8	87.7	55.8	87.7	55.8	87.6	55.8	350.8	223.3	2.6	1.7
uk-2005	34.6	21.8	34.6	21.8	34.6	21.8	34.6	21.8	138.4	87.3	1.0	0.7
it-2004	35.5	22.5	35.5	22.5	35.5	22.5	35.5	22.5	142.0	89.9	1.1	0.7
friendster	52.6	38.6	52.6	38.6	52.6	38.6	52.6	38.6	210.3	154.2	1.6	1.1
twitter	36.9	21.5	36.9	21.5	37.0	21.5	37.0	21.5	147.8	85.9	1.1	0.6

Table 4: Number edges (in Millions) before and after removing redundant from the induced subgraph. Memory in GB

Dataset	*Partition 0*		*Partition 1*		*Partition 2*		*Partition 3*		*Total*		Memory footprint	
	$\|E_{ind}\|$	$\|E_{res}\|$	$\|E_{ind}\|$	$\|E_{res}\|$	$\|E_{ind}\|$	$\|E_{res}\|$	$\|E_{ind}\|$	$\|E_{res}\|$	$\|E_{ind}\|$	$\|E_{res}\|$	$\|E_{ind}\|$	$\|E_{res}\|$
webbase	797.9	514.2	797.5	514.3	797.3	514.3	796.2	514.2	3189.8	2057.0	23.8	15.3
uk-2005	771.6	496.7	771.4	496.5	771.5	496.6	771.5	496.8	3085.2	1986.5	23.0	14.8
it-2004	1014.4	747.2	1014.5	747.1	1014.2	748.3	1014.3	748.2	4057.4	2992.4	30.2	22.3
friendster	1780.9	1690.9	1780.9	1690.9	1780.9	1690.9	1780.9	1690.9	7123.6	6763.8	53.1	50.4
twitter	1185.8	969.5	1186.3	969.8	1186.7	969.9	1186.6	970.0M	4745.3	3879.1	35.4	28.9

removing redundant vertices and edges, leading to a more efficient use of resources while maintaining accurate triangle counting.

Impact of Graph Transformation To further show the strength of graph transformation and support of large graphs, Table 5 shows the results of LSTC with and without graph transformation for two additional data sets *uk-2007* [1] and *gsh-host* [2] on V100 GPU.

Note that these graphs can not be processed using the capacity of single GPU memory without the workload partition. Also Trust and Tricore frameworks failed to run on these graphs on the single GPU memory. However, LSTC can compute the triangles with single GPU memory, whose results are shown for 4 and 8 partitions. However, when the graph transformation is disabled with LSTC, the resultant partitions suffers from the redundant vertices and edges for both 4 and 8 partitions. For instance, the maximum memory across all the 4 partitions for uk-2007 is 22.2GB, which is more than the available GPU memory capacity. Hence, uk-2007 failed to run on the single GPU memory when the graph transformation is disabled. On increasing the number partitions to 8, uk-2007 still requires 19.69GB memory for one of the created partitions. Moreover, we observe that even after increasing the number of partitions to 16, the larger partition incurs memory more than the size of GPU memory. Hence, without graph transformation these the datasets failed to run even using 16 partitions.

We observe the similar behavior for gsh-host as well. Note that for gsh-host we synthetically set the V100 GPU memory limit to 12 GB to demonstrate the effectiveness of graph transformation. LSTC can compute the triangles on gsh-host using 4 partitions, whereas it can not be processed with 4 partitions if the graph transformation is disabled, because one of the created partition requires larger than 12GB memory [1] and hence it can not processed by the single GPU memory. This demonstrates the graph transformation is effective in reducing the redundant vertices and edges, enables to process the large graphs using single GPU memory capacity.

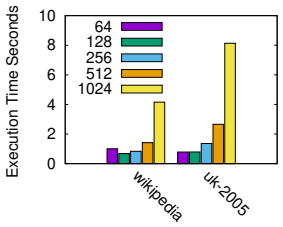


Figure 7: Impact of neighborhood vertices in shared memory

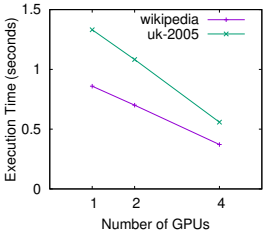


Figure 8: Scalability on the cluster. Each node in the machine is equipped with 2 GPUs.

6.1.3 ***Scalability analysis.*** Figure 8 shows the scalability results[3] to demonstrate that our approach can be made applicable to multi-GPU or multi-node settings. The figure shows the scaling results by varying the number of GPUs from 1 to 4 for uk-2005 and wikipedia datasets. From the results, we observe our approach scales well, i.e, it achieves 2.31× speedup for wikipedia on 4 GPUs when compared to 1 GPU. Similarly, it achieves 2.38× speedup for uk-2005 on 4 GPUs when compared to 1 GPU. This is primarily because our approach does not require any communication overhead while computing the triangles on each partition, except for the final aggregation of the local triangle counts computed for each partition.

[1] It requires 11.57GB for the application processing and additional memory for GPU system monitoring.

[2] We synthetically set the V100 GPU memory limit to 12 GB for gsh-host to demonstrate the effectiveness of graph transformation

[3] We extended our implementation for the multi-node settings using MPI.

Table 5: Effectiveness of graph transformation on V100 GPU. M denotes million. '-' means failed due to out of memory. Both Trust and Tricore frameworks failed to run due to out of memory.

		LSTC				LSTC Without Graph Trasnsform			
Dataset		**Max memory (GB)**	**Data Transfer (s)**	**Kernel Execution (s)**	**Kernel+ Data Transfer Time (s)**	**Max memory (GB)**	**Data Transfer (s)**	**Kernel Execution (s)**	**Kernel+ Data Transfer Time (s)**
uk-2007 (50GB)	**P = 4**	15.08	8.72	27.00	35.72	22.20	-	-	-
$\|V\|$ = **105M** $\|E\|$ = **6,603M**	**P = 8**	12.21	10.84	27.25	38.10	19.69	-	-	-
gsh-host (23GB)[2]	**P = 4**	9.27	4.57	129.02	133.59	11.57	-	-	-
$\|V\|$ = **68M** $\|E\|$ = **3,005M**	**P = 8**	4.66	8.01	129.89	137.90	5.29	7.80	131.04	138.84

Table 6: Comparing LSTC and Tricore with different partitioning schemes.

	Tricore			*LSTC (Contiguous)*			*LSTC (METIS)*		
Dataset	***Workload Partition***	***Kernel***	***Total Time***	***Workload Partition***	***Kernel***	***Total Time***	***Workload Partition (part time)***	***Kernel***	***Total Time***
webbase	117.754	3.269	121.023	13.457	1.538	14.996	421.971 (405.835)	1.369	423.341
it-2004	98.265	5.188	103.453	18.908	5.273	24.181	334.385 (320.389)	5.331	339.717
twitter	182.216	7.296	189.513	56.303	16.718	73.022	2384.611 (2323.378)	16.856	2401.468
uk-2005	191.569	3.133	194.703	13.061	1.495	14.557	270.185 (258.985)	1.415	271.600
Frienster	424.608	6.814	431.422	79.198	6.289	85.487	4286.986 (4193.304)	6.253	4293.239

6.1.4 ***LSTC performance using different partition schemes.*** Table 6 demonstrate that LSTC can be applied to various other partition schemes, we integrated LSTC with two partitioning schemes, i.e., METIS [22] and contiguous partitioning schemes. For the contiguous partitioning scheme, we assign an equal chunk of contiguous vertices to the partitions. Table 6 compares the performance Tricore with LSTC (Contiguous) and LSTC (METIS). Note that for LSTC, we used two partitions but executed them on a single GPU, whereas Tricore is executed on 2 GPUs. The table reports the results of workload partition and kernel execution time for all the datasets. From the results we can observe that LSTC (Contiguous) outperforms Tricore. However, LSTC (METIS) performs very poor this is primarily because METIS consumes significant time in partitioning the vertices (shown in brackets in the figure) of the graph into desired number of partitions. In addition, LSTC outperforms Tricore in terms of kernel execution for most of the datasets, which indicates that our GPU optimizations are effective in improving the overall performance.

6.2 Analyzing Triangle Counting Without Partitioning

This section analyzes the performance of the optimizations (as discussed in Section 4.4) for computing the triangle counting algorithm on a single GPU. We use medium datasets, which fit the capacity of a single GPU for analysis. Hence, workload partitioning is not required for these datasets. This helps us to clearly show the effect of the optimizations without combining the benefits of the workload partitioning technique.

6.2.1 ***Comparison with state-of-the-art.*** Table 9 compares the performance of LSTC with that of Tricore and TRUST frameworks on medium-sized graphs. The table also includes the results on two large graphs, i.e., wikipedia and uk-2005, to further show the effectiveness of our triangle counting algorithm.

From the results, it is evident that LSTC outperforms Tricore for all datasets and achieves an average speedup of 3.5× (Geometric Mean) when compared to Tricore. Even though both Tricore and LSTC employs the binary search algorithm, LSTC further exploits the shared memory for storing neighborhood vertices. It improves the data locality by analyzing the memory access patterns. Furthermore, it minimizes the number of global memory accesses by avoiding atomic operations. These optimizations collectively help minimize the number of global memory accesses, thus improving overall performance.

These results demonstrate that LSTC performs better than state-of-the-art binary search implementation as well has hashing based implementation.

6.2.2 ***Analyzing the Impact of Shared Memory in GPUs.*** Figure 9 shows the benefits of shared memory in LSTC. The figure compares the performance of LSTC on various datasets with and without using shared memory. The results demonstrate that LSTC with shared memory significantly improves performance for most datasets (except for *RN-CA*). This is because all the threads in a thread block access the same neighbours (as discussed in Section 4.4) for computing the neighbourhood intersection. Since global memory has high latency overhead compared to shared memory and shared memory can be accessed by all threads in a block, storing the frequently accessed neighbours in shared memory achieves a

Table 7: Impact of Shared Memory Optimizations on GPU Efficiency

Metrics	uk-2005		wikipedia	
	Without shared memroy	With shared memory	Without shared memroy	With shared memory
No. of Elapsed Cycles	24540235733	2565681897	37397684678	1519322324
Instructions Per Cycle (IPC)	0.22	1.64	0.12	2.13
No. of Stall Cycles Per Instruction	210.58	27.82	404.7	21.8
L1 Cache Hit Rate (%)	66.15	87.42	48.08	51.87
Achieved Occupancy (%)	93.27	97.06	61.06	98.42
Active Warps Per SM	44.77	46.59	29.31	47.24

Table 8: Analyzing thread block occupancy

Shared memory required for neighbors (Bytes) (A)	Other shared memory (Bytes) (B)	Total shared memory per block (Bytes) (C)	Block limit due to shared memory (D)	Block limit due to registers (E)	Block limit due to max thread size (F)	Maximum no. of blocks can reside on SM (G)	Thread block occupancy minimum of D, E, F, G (H)
uk-2005							
256	1544	1800	18	25	24	16	16
512	2056	2568	12	12	12	16	12
1024	3080	4104	15	6	6	16	6
2048	5126	7174	9	3	3	16	3
4096	9228	13324	1	1	1	16	1
wikipedia							
256	1544	1800	18	25	24	16	16
512	2056	2568	12	12	12	16	12
1024	3080	4104	15	6	6	16	6
2048	5126	7174	4	3	3	16	3
4096	9228	13324	1	1	1	16	1

Table 9: Comparing execution time (in seconds) of "LSTC", "Tricore" and "Trust" on a single GPU for medium datasets. For TRUST, the reorder time is listed in brackets.

Dataset	*Trust*	*Tricore*	*LSTC*
RN-CA	0.514 (0.513)	0.480	**0.005**
cit	1.633 (1.621)	0.622	**0.034**
scale21	0.853 (0.655)	0.839	**0.204**
scale22	2.054 (1.498)	1.200	**0.485**
scale23	4.646 (3.235)	2.131	**1.310**
wikipedia	12.072 (11.603)	2.442	**0.961**
uk-2005	29.319 (29.019)	3.631	**1.534**

significant reduction in memory access latency. The exception is for *RN-CA* as each vertex in the graph has very few neighbours (maximum 4); as a result, transferring the data from global memory to shared memory outweighs the benefits of shared memory.

To quantify the benefits of shared memory further, we show the reduction in global memory accesses with and without using the shared memory as shown in Figure 10. We collected the number of memory accesses by using the NVIDIA Nsight Compute Unit (NCU) tool [25]. The results also show a significant reduction in the number of global memory accesses compared when using the shared memory for most of the datasets. For instance, scale21, scale22 and scale23 achieve a 94.54%, 94.76%, and 93.90% reduction in global memory access, respectively. The similar behavior can be observed on Wikipedia and uk-2005. However, *RN-CA* suffers from the benefits as each has very few neighbours and hence can not get the benefit of global memory access coalescing.

These results demonstrate that shared memory is effective in minimizing global memory access and improving computational efficiency.

6.2.3 ***Impact of Shared Memory Optimizations on GPU Efficiency.*** Table 7 reports various metrics obtained from the NVIDIA profiler to measure the impact of shared memory optimizations on GPU efficiency. From the table, we can observe that with shared memory optimizations proposed in the paper, the kernel execution takes fewer elapsed cycles and achieves a higher number of instructions executed per cycle for both uk-2005 and wikipedia datasets. This is because our optimizations result in a few number of global memory accesses, which results in a significant reduction of warp stall cycles (reported as No. of Stall Cycles Per Instruction in the table). In addition, with a reduction in the number of global memory accesses, the L1 cache gets less polluted and benefits from increased L1 hit rates, i.e., 87.42% for our approach compared to 66.15% without shared memory optimizations for uk-2005. As a result, the warps in the SM complete their memory accesses soon

and become available for execution. Consequently, our approach benefits from a more number of active warps and effective occupancy.

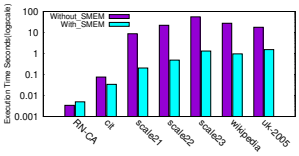


Figure 9: Analyzing impact of shared memory in GPU

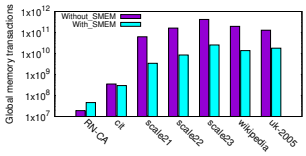


Figure 10: Reduction in global memory accesses

6.2.4 ***Impact on SM Occupancy***. Shared memory significantly impacts performance; however, the choice of shared memory size limits the number of resident thread blocks (thread block occupancy) of the SM in GPU. Which in turn can affect the performance of LSTC. To illustrate this, Figure 7 shows impact of varying shared memory size by varying the number of vertices in the shared memory on LSTC for two large datasets *wikipedia* and *uk-2005*. The results show that the performance of LSTC improves when the neighborhood vertices in shared memory increases from 64 to 128. However, increasing the neighborhood vertices in shared memory beyond 128 degrades the performance.

To understand the variation in performance, we report the thread block occupancy (Column H) for various shared memory sizes in Table 8. We require shared memory to store the neighborhood vertices (in Column A). Shared memory is also required to store local triangle counts per thread and to store system driver data as needed by GPU. This is denoted as other shared memory (in Column B). The total shared memory usage per thread block is denoted in Column C. The number of thread blocks that can be launched as per the total shared memory usage is listed in Column D. However, the thread block occupancy in the SM also depends on the register usage, thread block size, and maximum number of thread blocks that can fit the SM. Hence, thread block occupancy is the minimum number of block limits obtained due to Columns D, E, F, and G.

From the Table 8, we observe that an increase in the shared memory size per thread block reduces the thread block occupancy as each thread block consumes more shared memory. An increase in the shared memory size, in other words, a decrease in the thread block occupancy, should degrade performance, as in the case of an increase in neighborhood size from 128 to 1024 (i.e., shared memory from 512 bytes to 4096 bytes). However, the performance improves even when neighborhood vertices increases from 64 to 128 (i.e., shared memory from 256 bytes to 512 bytes). In these graphs, we observe that most of the vertices have a degree less than 128. Storing all the neighbors in shared memory results in improved performance as all of them can be stored in the shared memory without any overhead. In addition, very few vertices have degrees more than 256; consequently, assigning more shared memory size per thread block not only underutilizes the shared memory but also reduces thread block occupancy and performance.

Table 10: Execution time (in seconds) of "LSTC", "Tricore" and "Trust" on large datasets in A6000 GPU.

Dataset	*Trust*	*Tricore*	*LSTC*
wikipedia	11.939 (11.603)	1.456	**0.694**
webbase	51.409 (51.152)	2.452	**0.763**
uk-2005	29.224 (29.019)	2.368	**0.782**
it-2004	31.230 (30.750)	4.196	**2.469**
friendster	125.813 (123.365)	**3.42**	3.682
twitter	60.167 (56.415)	10.912	**9.716**

6.2.5 ***Comparison of larger datasets on A6000 GPU***. To further show the effectiveness of the triangle counting algorithm without workload partition, we execute the large datasets using the A6000 GPU. Table 10 compares the kernel execution time for 3 approaches LSTC, Tricore and Trust on all large datasets. Since the datasets fit the capacity of a single GPU memory, we do not apply any workload partition for all approaches. The results show that LSTC outperforms Tricore and Trust in most datasets. LSTC achieves an average speedup of 1.82× and a maximum up to 3.2× when compared to Tricore. For friendster the results are comparable to that Tricore. As discussed, Trust spends significant time in reordering the vertices, which consume significant time (shown in brackets). These results demonstrate the overall effectiveness of the proposed optimizations in improving the performance of triangle counting on a single GPU.

7 Conclusion

This paper presents LSTC, an efficient implementation of triangle counting on a single GPU. The key is to (1) process larger graphs on a single GPU even though they do not fit the capacity of single GPU memory and (2) improve performance by efficiently utilizing the GPU resources. To this we first propose an efficient workload partition technique, which helps in computing the triangles across partitions without any communication overhead and reduces the number of redundant vertices and edges across the partitions. Further, we propose several optimizations to efficiently utilize GPU resources to achieve overall improvement. We evaluated our approach on several wide-range data sets and compared it with state-of-the-art triangle counting implementations. We observe that LSTC not only achieves superior performance when compared with state-of-the-art multi-GPU implementations but also requires fewer resources for computation.

Acknowledgments

The authors acknowledge the funding support received from the Department of Science and Technology (DST), Government of India, through the Science and Engineering Research Board (SERB) under the Start-up Research Grant (SRG/2021/001134). The authors also acknowledge the computational resources provided by PARAM Shakti supercomputer at the Indian Institute of Technology, Kharagpur, under the National Supercomputing Mission (NSM), Government of India. In addition, the authors acknowledge the supercomputing facilities provided by the Orion Cluster at the Indian Institute of Technology, Bhilai.

References

[1] 2007. UK-2007. http://konect.cc/networks/.

[2] 2015. GSH-Host. https://law.di.unimi.it/webdata/gsh-2015-host/.

[3] Amro Alabsi Aljundi, Taha Atahan Akyildiz, and Kamer Kaya. 2022. Degree-Aware Kernels for Computing Jaccard Weights on GPUs. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 897–907. doi:10.1109/IPDPS53621.2022.00092

[4] Naiyong Ao, Fan Zhang, Di Wu, Douglas S. Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. 2011. Efficient Parallel Lists Intersection and Index Compression Algorithms Using Graphics Processing Units. *Proc. VLDB Endow.* 4, 8 (may 2011), 470–481. doi:10.14778/2002974.2002975

[5] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides 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] Mauro Bisson and Massimiliano Fatica. 2017. High Performance Exact Triangle Counting on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 28, 12 (2017), 3501–3510. doi:10.1109/TPDS.2017.2735405

[7] Ketan Date, Keven Feng, Rakesh Nagi, Jinjun Xiong, Nam Sung Kim, and Wen-Mei Hwu. 2017. Collaborative (CPU + GPU) algorithms for triangle counting and truss decomposition on the Minsky architecture: Static graph challenge: Subgraph isomorphism. In *HPEC*. 1–7. doi:10.1109/HPEC.2017.8091042

[8] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *PLDI*.

[9] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Vishwesh Jatala, V. Krishna Nandivada, Marc Snir, and Keshav Pingali. 2019. Gluon-Async: A Bulk-Asynchronous System for Distributed and Heterogeneous Graph Analytics. In *PACT*.

[10] Sayan Ghosh. 2022. Improved Distributed-memory Triangle Counting by Exploiting the Graph Structure. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. doi:10.1109/HPEC55821.2022.9926376

[11] Sayan Ghosh and Mahantesh Halappanavar. 2020. TriC: Distributed-memory Triangle Counting by Exploiting the Graph Structure. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. doi:10.1109/HPEC43674.2020.9286167

[12] Ilias Giechaskiel, George Panagopoulos, and Eiko Yoneki. 2015. PDTL: Parallel and Distributed Triangle Listing for Massive Graphs. In *2015 44th International Conference on Parallel Processing*. 370–379. doi:10.1109/ICPP.2015.46

[13] graphchallenge 2023. Graph Challenge. https://graphchallenge.mit.edu/

[14] Chuangyi Gui, Long Zheng, Pengcheng Yao, Xiaofei Liao, and Hai Jin. 2019. Fast Triangle Counting on GPU. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. doi:10.1109/HPEC.2019.8916216

[15] Chuangyi Gui, Long Zheng, Pengcheng Yao, Xiaofei Liao, and Hai Jin. 2019. Fast Triangle Counting on GPU. In *HPEC*. 1–7. doi:10.1109/HPEC.2019.8916216

[16] Loc Hoang, Vishwesh Jatala, Xuhao Chen, Udit Agarwal, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2019. DistTC: High Performance Distributed Triangle Counting. In *HPEC*. 1–7. doi:10.1109/HPEC.2019.8916438

[17] Lin Hu, Lei Zou, and Yu 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

[18] Yang Hu, Pradeep Kumar, Guy Swope, and H. Howie Huang. 2017. TriX: Triangle counting at extreme scale. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. doi:10.1109/HPEC.2017.8091036

[19] Yang Hu, Hang Liu, and H. Howie 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

[20] Jianqiang Huang, Haojie Wang, Xiang Fei, Xiaoying Wang, and Wenguang Chen. 2022. $TC-Stream$TC-Stream: Large-Scale Graph Triangle Counting on a Single Machine Using GPUs. *IEEE Transactions on Parallel and Distributed Systems* 33, 11 (2022), 3067–3078.

[21] Guanxian Jiang, Qihui Zhou, Tatiana Jin, Boyang Li, Yunjian Zhao, Yichao Li, and James Cheng. 2022. VSGM: view-based GPU-accelerated subgraph matching on large graphs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 52, 15 pages.

[22] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392. arXiv:https://doi.org/10.1137/S1064827595287997 doi:10.1137/S1064827595287997

[23] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable Subgraph Enumeration in MapReduce. *Proc. VLDB Endow.* 8, 10 (jun 2015), 974–985. doi:10.14778/2794367.2794368

[24] Caleb C. Noble and Diane J. Cook. 2003. Graph-Based Anomaly Detection. Association for Computing Machinery, New York, NY, USA.

[25] NVIDIA. [n. d.]. CUDA Technology. http://www.nvidia.com/CUDA/.

[26] OpenCL. [n. d.]. Open Computing Language. https://www.khronos.org/opencl/.

[27] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D. Owens. 2017. Multi-GPU Graph Analytics. In *IPDPS*.

[28] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. 2019. H-INDEX: Hash-Indexing for Parallel Triangle Counting on GPUs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. doi:10.1109/HPEC.2019.8916492

[29] Santosh Pandey, Zhibin Wang, Sheng Zhong, Chen Tian, Bolong Zheng, Xiaoye Li, Lingda Li, Adolfy Hoisie, Caiwen Ding, Dong Li, and Hang Liu. 2021. Trust: Triangle Counting Reloaded on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 32, 11 (2021), 2646–2660. doi:10.1109/TPDS.2021.3064892

[30] Roger Pearce. 2017. Triangle counting for scale-free graphs at scale in distributed memory. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–4. doi:10.1109/HPEC.2017.8091051

[31] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph Processing on GPUs: A Survey. *ACM Comput. Surv.* (2018).

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

[33] Kanat Tangwongsan, A. Pavan, and Srikanta Tirthapura. 2013. Parallel Triangle Counting in Massive Streaming Graphs. In *Proceedings of the 22nd ACM International Conference on Information &; Knowledge Management* (San Francisco, California, USA) *(CIKM '13)*. Association for Computing Machinery, New York, NY, USA, 781–786. doi:10.1145/2505515.2505741

[34] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *Proc. VLDB Endow.* 5, 9 (may 2012), 812–823. doi:10.14778/2311906.2311909

[35] Leyuan Wang and John D. Owens. 2019. Fast BFS-Based Triangle Counting on GPUs. In *HPEC*. 1–6. doi:10.1109/HPEC.2019.8916434

[36] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D. Owens. 2016. A Comparative Study on Exact Triangle Counting Algorithms on the GPU. In *Proceedings of the ACM Workshop on High Performance Graph Processing, HPGP@HPDC 2016, Kyoto, Japan, May 31, 2016*, Toyotaro Suzumura, Dario Garcia-Gasulla, and Miyuru Dayarathna (Eds.). ACM, 1–8. doi:10.1145/2915516.2915521

[37] S. Wasserman and K. Faust. 1994. *Social Network Analysis: Methods and Applications*. Cambridge University Press.

[38] Duncan J. Watts and Steven H. Strogatz. 2006. *Collective dynamics of 'small-world' networks*. Princeton University Press, Princeton, 301–303. doi:doi:10.1515/9781400841356.301

[39] Yihua Wei and Peng Jiang. 2022. STMatch: Accelerating Graph Pattern Matching on GPU with Stack-Based Loop Optimizations. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022*, Felix Wolf, Sameer Shende, Candace Culhane, Sadaf R. Alam, and Heike Jagode (Eds.). IEEE, 53:1–53:13. doi:10.1109/SC41404.2022.00058

[40] Abdurrahman Yaşar, Sivasankaran Rajamanickam, Jonathan W. Berry, and Ümit V. Çatalyürek. 2022. A Block-Based Triangle Counting Algorithm on Heterogeneous Environments. *IEEE Transactions on Parallel and Distributed Systems* 33, 2 (2022), 444–458. doi:10.1109/TPDS.2021.3093240

[41] Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongzhi Chen, and Guoren Wang. 2022. Lightning Fast and Space Efficient K-Clique Counting. In *Proceedings of the ACM Web Conference 2022* (Virtual Event, Lyon, France) *(WWW '22)*. Association for Computing Machinery, New York, NY, USA, 1191–1202. doi:10.1145/3485447.3512167

[42] Jiyuan Zhang, Yi Lu, Daniele G. Spampinato, and Franz Franchetti. 2020. FESIA: A Fast and SIMD-Efficient Set Intersection Approach on Modern CPUs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1465–1476. doi:10.1109/ICDE48307.2020.00130