TRICORE: Parallel Triangle Counting on GPUs

Yang Hu[†] Hang Liu[‡] H. Howie Huang[†]
[†]The George Washington University
[‡]University of Massachusetts Lowell
huyang@gwu.edu,Hang_Liu@uml.edu,howie@gwu.edu

***Abstract*—Exact triangle counting algorithm enumerates the triangles in a graph by identifying the common neighbors of two vertices of each edge. In this work, we present TRICORE, a scalable GPU-based triangle counting system that consists of three major techniques. First, we design a binary search based algorithm that can increase both the thread parallelism and memory performance on Graphics Processing Units (GPUs), both of which are absent from prior work. Second, in contrast to prior attempts which require multiple graph representations, i.e., compressed sparse row (CSR), edge list, and bitmap, to be present in the GPU memory, TRICORE evenly partitions and distributes the partitioned CSR data across all the GPUs, and uses a streaming buffer to load the edge list from the CPU memory on the fly. This design enables TRICORE to process the graphs that are orders of magnitude larger than the GPU memory. Third, we further develop a dynamic workload management technique to balance the workload across GPUs. Our evaluation demonstrates that TRICORE on a single GPU can count the triangles in the billion-edge Twitter graph within 24 seconds, that is, 22× faster than the state-of-the-art CPU project which uses CPUs that are 8× more expensive. When processing big graphs (up to 33.4 billion edges) that are ∼22× larger than the memory size of a single GPU, it achieves 24× speedup when scaling from 1 to 32 GPUs.**

I. INTRODUCTION

Exact triangle counting serves as a building block for an array of graph algorithms such as clustering coefficients [63] and k-truss [58]. Also, exact triangle counting can be easily extended to other triangulation algorithms such as triangle listing, 3-profiles [23], and counting cycles of a specific size [6]. In practice, triangle counting is also used for a wide range of applications, such as, detection of spams [10] and thematic structures [22], link recommendation [55] and social network analysis [18], [20].

Traditional triangle counting algorithm iterates through each edge of the graph and intersects (i.e., compares) the neighbor lists of both source and destination vertices. Once a common neighboring vertex is found, a triangle is enumerated. As such, the theoretical computation complexity of triangle counting is $\mathcal{O}(|E|^{1.5})$, where $|E|$ is the number of edges in the graph of interest [6], [51], [30]. Given real-world graphs can easily go beyond trillion-edge [49], [31], [37], triangle counting is regarded computationally prohibitive [51], [12], [64], [57].

Contemporary GPUs, such as Nvidia Volta V100 [5] that can provide 14 Tera-Floating Points Operation Per Second (TFLOPS) and 900 Gigabytes/s (GB/s) throughput [60], are ideal platforms to accelerate triangle counting algorithm. We observe a plethora of projects [27], [59], [13] have already explored this direction. However, these efforts face severe hindrances to unleash the potential of GPUs. For example, [27] has largely followed a CPU-based approach, i.e., merge-based triangle enumeration. As such, it first needs to pay non-trivial overhead to evenly partition the neighbors of source and destination vertices [26] across a warp of GPU threads. Afterwards, each thread in the warp will encounter strided memory access since the former step schedules consecutive threads to process non-adjacent neighbor list partitions. In addition, all of these projects are restricted to a single GPU and they require multiple formats of the graph to be present in the GPU's memory. Even the GPU with the largest memory available, the Maxwell P100, has only 24GB of memory. Even worse, some of them, e.g., [13], [12], have to reserve considerable GPU memory space for intermediate data structures, further limiting the size of the graphs that can be handled by GPUs.

In this context, scaling triangle counting to distributed memory systems is of vital importance to fully take advantage of GPUs. Unfortunately, triangle counting needs the two-hop neighbors of each vertex in order to enumerate triangles, which places stringent requirements on inter-machine communications and workload balancing. For instance, the most recent Graph Challenge [1] champion (i.e., [46]) which relies on traditional 1-D partition [17], [16] and vertex delegation [47] mechanisms to distribute graphs, experiences non-trivial inter-machine communications and thereby shows poor scalability (4.2× speedup from 16 to 256 machines).

To address these challenges, we have developed a new scalable triangle counting framework – **TRICORE** – which is able to scale our triangle counting to a large number of GPUs and process big graphs that are orders of magnitude larger than GPU memory with up to 24× speedup over the state-of-the-art attempts. It consists of the following major techniques.

First, we design a novel binary search-based triangle counting algorithm for GPUs with rigorous theoretical support. Specifically, our approach uses one neighbor list as the lookup list and the other as the binary search tree. Afterwards, it checks the binary search tree to see if each lookup element exists in the tree. Further, TRICORE caches the first few levels of the binary search tree in the fast GPU shared memory, resulting in another 18% reduction of (expensive) global memory transactions.

This algorithm addresses the warp divergence and inefficient memory access issues, both of which are faced by the conventional merge-based algorithm. Our theoretical analysis shows that in spite of slightly worse time complexity, parallel binary search-based algorithm performs significantly better

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00

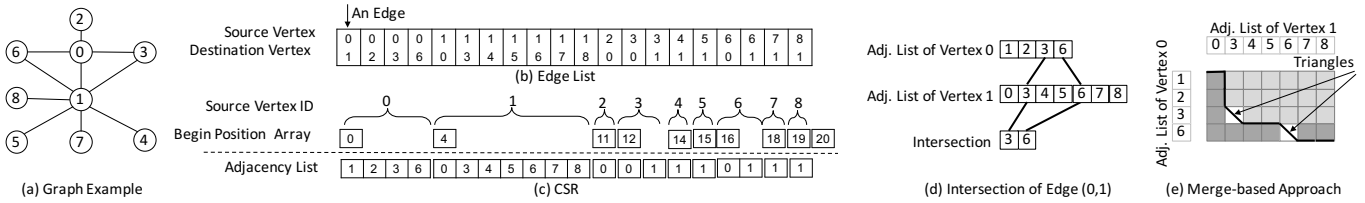


Fig. 1: Various Graph Representations and Hybrid CSR. Our input is a simple undirected graph $G = (V, E)$ and output is the triangle number in this graph. The main structured representations of a graph are shown in Figure 1. Note that the adjacency list format (CSR) is a compressed matrix but also can be seen as half of edge list with a begin position to distinguish the source node of edges. Adjacency array format has similar features but puts neighbor lists in multiple arrays and uses pointers instead of begin position arrays.

than merge-based intersection on GPUs. In short, the new binary search approach reduces the memory transactions by 6×, which makes TRICORE 6× and 2.7× faster than the state-of-the-art GPU implementations [27], [13], respectively, and 2.5× and 1.9× faster than the latest CPU implementations [51], [64], respectively.

Second, we further propose a three-pronged optimization in order to allow TRICORE to accommodate the extremely large graphs with a collection of GPUs in a distributed system. First, TRICORE only stores the CSR format of the graph in GPU memory. To load the edges from CPU memory to GPU memory, TRICORE uses a ring of streaming buffers. In particular, such a process of edge loading is completely overlapped with the counting of triangles. Second, TRICORE adopts an external memory algorithm to partition the CSR format of the graph into communication-free sub-partitions. Third, TRICORE designs a new dynamic scheduling technique to balance workload across a large number of GPUs. Taken together, TRICORE can solve graphs whose sizes are orders of magnitude larger than a single GPU memory space. TRICORE also achieves close to linear scalability from 1 to 32 GPUs.

We evaluate TRICORE in three different settings: a single GPU, multi-GPU on a single machine, and many GPUs across multiple machines. First, our single Nvidia Titan X based TRICORE outperforms the state-of-the-art single GPU and 48-core CPU projects by 2.2× and 8.5×, respectively. Note that the total cost of Titan X GPU and the host CPU is 8× cheaper than the 48-core CPU. Second, TRICORE on two Titan X GPUs can obtain a comparable performance to the latest Graph Challenge Champion [46], which uses 256 machines. This is also 4.4× faster than the existing distributed CPU-based triangle counting (i.e., PDTL) on six machines. Last, we use a cluster of 32 Nvidia K20 GPUs to process the graphs that are up to ∼22× larger than the memory size of a single GPU, and achieve 24× speedup when scaling from 1 to 32 GPUs.

The rest of paper is organized as follows. Section II describes background and related work. Section III presents the main challenges of achieving good performance of triangle counting on single GPU and multiple GPUs, and gives an overview for all solutions in our system. Section IV and V discuss the techniques proposed in this paper. The experiments and results are presented in Section VI. Section VII concludes.

II. BACKGROUND AND RELATED WORK

This section first discusses the preliminary background for TRICORE – GPU architecture, current triangle counting algorithms, and merge-based intersection. Subsequently, we present the landscape of the related work.

A. Graphics Processing Unit

For triangle counting, GPUs offer two advantageous features. First, GPUs have thousands of simplified CUDA (Compute Unified Device Architecture) cores, thus can run a large number of threads. For example, Nvidia K40c GPUs (used in this paper) feature 2,880 CUDA cores and support millions of threads [43]. Second, their memory bandwidth is high, e.g., 288GB/s in K40c vs. 68GB/s on Xeon E5-2683 [3]. Below, we rigorously discuss GPU thread and memory hierarchies.

Thread hierarchy. A GPU contains several streaming multiprocessors (SMX), and each SMX contains hundreds of CUDA cores. The CUDA programming model provides several thread mapping abstractions. Namely, a *thread* uses one core to execute, and 32 consecutive threads form a *warp*. A set of consecutive warps further arrive at a *Cooperative Thread Array* (CTA). All the CTAs together are called a *Grid* which accounts for all the threads on a GPU. In particular, each SMX executes one warp of threads in Single Instruction Multiple Data (SIMD) fashion. Given that, when a warp is executing through a branch, it has to wait even if only part of the threads of the warp take a particular branch, which is termed as *branch divergence*.

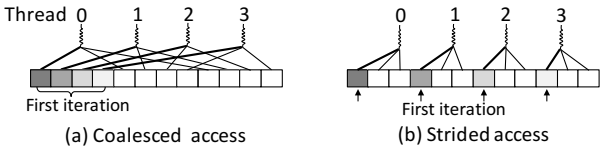


Fig. 2: Memory access patterns.

Memory hierarchy. All the SMXs share the same GPU global memory and L2 cache. Besides, each SMX further owns a private, manually controllable on-chip shared memory that is accessible by all the threads in one CTA. It is worth noting that GPUs contain much smaller caches than CPUs. Using the GPU and CPU used by this paper as an example, each K40c SMX features a 64KB shared memory and shares the 1.5 MB L2 cache with all other SMXs. In contrast, Xeon E2683 CPU equips a 35 MB last level cache (LLC) [3] . As a result, data reuse opportunity is challenging on GPUs especially for data-intensive tasks such as triangle counting. GPU full memory bandwidth can only be achieved via *coalesced memory accesses*, i.e., all threads in the warp access consecutive memory addresses. In contrast, *strided memory accesses* that are caused

by scheduling each thread in a warp to read far apart addresses will result in much lower throughput.

Mathematically, a stride of k will lead to $\frac{k-1}{k}$ throughput degradation. Coalesced access is a special case where $k = 1$. As shown in Figure 2, the stride of Figure 2(a) and 2(b) are 1 and 3. Thus Figure 2(b) experiences $\frac{2}{3}$ throughput reduction.

B. Triangle Counting Algorithm

A graph $G = (E, V)$, where E and V represent the edges and vertices, respectively, can be stored in a variety of representations, such as edge list and CSR. Figure 1(b) and 1(c) exemplify the two formats, respectively. A triangle is defined as a group of three vertices with one edge between each pair of vertices. For example, vertices 0, 1 and 3 from Figure 1(a) form a triangle $\Delta_{0,1,3}$.

Algorithm 1 Vertex-iterator

```
1: G = (E, V)
2: foreach u ∈ V in parallel do
3:     foreach v ∈ N(u) do
4:         count+=Intersection(u, v);
5:     end for
6: end for
```

Algorithm 2 Edge-iterator

```
1: G = (E, V)
2: foreach (u, v) ∈ E in parallel do
3:     count += Intersection(u, v);
4: end for
```

Current triangle counting algorithm, i.e., *compact-forward* [35], [51], [24], [27], [7], consists of two major steps: *orientation* and *computation*. On one hand, orientation step pre-processes the input (undirected) graph – the focus of this work – which reduces half of the edges in order to eliminate redundant computation, e.g., triangle $\Delta_{0,1,3}$ and $\Delta_{0,3,1}$ are actually identical but will be found twice. On the other hand, computation exploits *intersection* to count triangles on the preprocessed graph. For sake of brevity, we use $d(v)$ and $N(v)$ to denote the degree and all neighbors of vertex v, respectively. The $intersection()$ function counts $|N(u) \cap N(v)|$, which is the number of shared neighbors between vertex u and v of an edge (u, v). As shown in Figure 1(a), there exist two shared neighbors 3 and 6 among the neighboring lists of vertex 0 and 1 from the edge $(0, 1)$, which means there are two triangles $\Delta_{0,1,3}$ and $\Delta_{0,1,6}$. Eventually, the sum of all the counts returned by the intersection is the total number of triangles in a graph.

Algorithm 1 and 2 are vertex- and edge-iterator triangle counting algorithms with the only difference being that the former initializes the parallelism from each vertex while the latter from each edge. TRICORE prefers edge-centric stemming from the consideration of workload balancing. In particular, the workload of each edge (u, v) is $d(u) + d(v)$, that is, the summation of out-degrees of the two vertices. In comparison, with vertex-centric, the workload of each vertex v turns out to be $d^2(v) + \Sigma_{u_i \in N(v)} d(u_i)$. Intuitively, these equations imply that the workload imbalance is linear and quadratic to the skewness of degree distribution [19], [27], respectively. However, edge-centric mechanism requires edge list to be present in memory for higher parallelism [27], which limits the graph size that can be accommodated.

Hash-based algorithm [35] exploits hash instead of intersection to count triangles but experiences the same complexity of $O(|E|^{3/2})$ as mainstream intersection-based algorithms. Prior works [35], [51] also show that in experiments the merge/intersection based algorithm is practically better than all others, such as hash-based one. This also explains the reason TRICORE chooses to optimize intersection-based algorithms.

C. Merge-Based Intersection

Merge-based intersection is commonly recognized as the state-of-the-art approach to identifying common neighbors for triangle counting [59]. Its time complexity is $O(d(u) + d(v))$ [24] where $d(u)$ and $d(v)$ are the degrees of vertex u and v, respectively. Figure 1(e) illustrates the merge-based intersection for edge (0, 1). During intersection, it uses two pointers to scan through the neighbor lists of vertex 0 and 1. If the neighbors from both lists are equal, a triangle is found and both pointers are incremented. Otherwise, the pointer for the array with the smaller neighbor ID is increased. In this example, one has to traverse from the top left to the bottom right corner. Any vertical or horizontal moves indicate no matches, while a diagonal move stands for the finding of a triangle, e.g., vertices 3 and 6. Overall, the time complexity of merge based triangle counting algorithm is $\sum_{v \in V} d^2(v)$.

D. Graph Benchmark

Name	Abbr.	Description	$\|V\|$	$\|E\|$	Triangle
Facebook	FB	Facebook user to friend link	96M	620M	3B
Orkut	OR	Orkut online social network	8M	327M	223M
Twitter	TW	Twitter follower network	41M	1.4B	34B
Wikipedia	WK	Links between Wikipedia pages	11M	258M	10B
RMAT	RM	R-mat (scale 22, degree 64)	4M	253M	2.1B
Random	RD	GTgraph: uniform degree	4M	511M	349K
Kronecker1	KR1	Kronecker (scale 22, degree 64)	4M	242M	5.3B
Kronecker2	KR2	Kronecker (scale 25, degree 16)	33M	523M	22B
Gsh-2015	GSH	Web graph	988M	33.2B	1.78T
Kron-30-16	KR3	Kronecker (scale 30, degree 16)	1.07B	17.0B	2.3T
Kron-31-16	KR4	Kronecker (scale 31, degree 16)	2.14B	34.1B	1.07T

TABLE I: Graph specification.

In this work, we use both real datasets and synthetic graphs to evaluate the performance of TRICORE. The detailed descriptions of graph benchmarks are shown in Table I. Briefly, real datasets include the social networks, e.g., Twitter [33], Facebook [25] and Orkut [42], and web graphs such as Wikipedia [32]. In social networks like Facebook and Orkut, each vertex is a user and each edge stands for a friendship between two users. Twitter is a social media graph where each directed edge represents a user follow relationship. In web graphs such as Wikipedia, each vertex is a page and each directed edge represents a hyperlink. We also use three popular synthetic graph generators to generate graphs with various degree distributions. For instance, random distribution [39] ensures all vertices have the same number of out-edges. R-MAT [19] and Kronecker [4] generate power-law degree distribution graphs. To demonstrate that TRICORE can handle large graphs, we also use three large graphs, i.e., GSH [15], [14], KR3 and KR4 that contain tens of billions of edges.

E. Related Work

This section discusses the landscape of related work in three aspects – exact, approximate, and distributed triangle counting.

The former two report the count of triangles with the only difference dwelling on how accurate the count is, while the last one focuses on counting triangles in a distributed environment.

Exact Triangle Counting is the goal of TRICORE. For simplicity, we use triangle counting to denote *exact triangle counting* in this paper. Triangle counting can be implemented using generic graph engine [34], [11], [21], or using linear algebra methods [9], [64], [59]. A representative project can be found in [51], which gains the speedup mainly via edge reduction motivated by another theoretical work [35]. Scaling up to 40 CPU cores, [51] uses a merge-based algorithm and achieves up to 50× speed up over a single threaded implementation.

The most relevant work [27] implements a merge-based algorithm on a single GPU. To enhance the parallelism, this work adopts their prior method [26] to partition two neighbor lists of each edge into 32 balanced and disjoint sections, which are subsequently processed by 32 threads from a warp concurrently. As noted in Section I, this attempt suffers from nontrivial overhead of partitioning the neighbor lists and further strided memory access. To solve this problem, we introduce a binary search based intersection method, and a scalable approach to distribute TRICORE across GPUs.

Approximate Triangle Counting. As a trade-off of precision and time consumption, many works shift to approximate triangle counting, e.g., [54], [44], [48], [50]. The most important technique in this line of research is sampling, that is, how to achieve high accuracy with much smaller sampled graph. Notably, Doulion [56] achieves 99% accuracy by speeding up the process by 130×. While our work delivers better performance with exact counts, it would be interesting to combine our technique with the approximation methods, which we leave as future work.

Distributed Triangle Counting concerns about workload balancing and communication. A few projects have proposed load balancing schemes for triangle counting in distributed environment, whereas they either fail to exactly balance the workload, or end up with extremely high balancing overhead. For example, MapReduce triangle counting work [53] uses rank-by-degree to distribute an equal number of vertices across machines in a round robin fashion. However, this design can merely alleviate, not resolve, the workload imbalance issue. Message Passing Interface (MPI) based triangle counting works [7], [8] use a static workload partitioning which relies on a costly process to estimate the workload for each vertex.

Deploying triangle counting on MapReduce [53], [45], [61] is another form of distributed triangle counting. One notable work [53] proposes to partition the graph into overlapped subgraphs in order to ensure one triangle appears in at least one partition. A follow-up work [45] carefully classifies the type of triangles during partitioning so that each triangle is counted only once. However, all these attempts pay significant overhead to load graph in memory for computation.

In contrast, we identify two essential requisites for a scalable distributed triangle counting: *a balanced and communication free partitioning method, and a runtime load balancing schema*. In particular, we extend the external memory partitioning method to distribute the CSR format of the graph across GPUs. Afterwards, TRICORE uses a well-tuned stream buffer to load the edge list in the GPU memory, which is completely overlapped with computation. Finally, we build an on-the-fly workload balancer to address the workload imbalance issue.

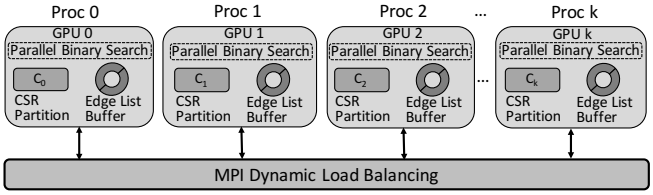


Fig. 3: TRICORE framework.

III. TRICORE OVERVIEW

In this section, we introduce the architecture of TRICORE, a high-performance scalable triangle counting algorithm. To motivate our designs, we evaluate prior GPU-based triangle counting projects [27] and come up with two key observations. First, we identify irregular memory footprint as the major bottleneck for prior projects. In response, we design a novel memory friendly intersection algorithm to overcome this bottleneck. Second, prior endeavors are limited to single GPU thus require graph data to fit in meager GPU memory. Worse still, to retain efficiency, they tend to store multiple formats of the graphs in GPU memory. Consequently, these attempts can only count triangles on graphs of 3 - 4 GBs. In contrast, TRICORE not only stores just one graph format (CSR) in GPU memory, but also partitions this format so that each GPU only accounts for one part of the CSR. Further, the edge list format is streamed to GPU memory at runtime with negligible performance penalty and further facilitates our dynamic workload balancing designs. Figure 3 presents the architecture of TRICORE.

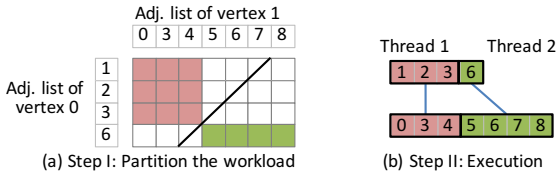


Fig. 4: Merge-based intersection.

Observation #1: Prior intersection algorithms suffer from irregular memory access patterns. We use the merge-based algorithm [27], shown in Figure 4, as an example. Note that we use the same graph as shown in Figure 1. Particularly, it consists of two steps: partitioning the workload (Figure 4(a)) and execution (Figure 4(b)). The first step, which partitions adjacent lists of vertex u and v into two disjoint parts, requires nontrivial $O(\log(|N(u)| + |N(v)|))$ operations. In this case, u and v are vertices 0 and 1, respectively. Afterwards, the execution also suffers from strided memory access patterns. For instance, while thread 1 is checking the 1st elements of those two arrays (i.e., 1 and 0), thread 2 is doing that on the 4th elements of those two arrays (i.e., 6 and 5).

Thereby, consecutive threads which are thread 1 and 2, are accessing non-consecutive addresses. Such a strided memory access pattern will potentially experience orders of magnitudes lower memory throughput [43], [36]. Another state-of-the-art work [13] uses bitmap based algorithm which incurs scattered (random) memory access pattern. A bitmap stores a record using its value as address to support $O(1)$ computational complexity search. However, when the threads in a warp search a list of keys in parallel, they can potentially read different random addresses, which makes the performance even worse than strided memory access.

In this paper, we propose a parallel GPU-aware binary search based intersection algorithm that experiences $\sim$6$\times$ fewer memory traffic comparing to conventional merge based algorithm on GPUs. We further cache critical binary tree nodes in GPU shared memory (L1 cache) to reduce another $\sim$20% expensive global memory transactions. Beyond that, we also provide detailed theoretical proofs to demonstrate why our parallel GPU-aware binary search based intersection is a better fit for GPU architecture.

Observation #2: Conventional GPU-based projects consume excessive amount of memory space. As detailed, in Section II-B, existing work chooses edge-centric design over vertex-centric one to combat workload imbalance concerns. TRICORE further exploits a finer-grained parallelization, called *warp-edge*, which uses a warp to work on an edge at one time. Because this avoids the warp divergence problem [26], [28], [41], this mechanism shows more promising performance.

However, both edge-centric and warp-edge parallelizations generally require both the edge lists and the adjacency lists (CSR) formats to be present in the GPU memory [27]. We have also investigated the possibility of using CSR format alone to implement edge-centric design. However, with this data format, if one edge in CSR adjacency list is dispatched to a thread, this thread has to calculate the source vertex of the edge, which introduces nontrivial overhead [40]. To avoid this overhead, we need to store the edge list in memory to support edge-centric parallelization.

To further exacerbate the situation, an edge list format takes about twice the storage space of the CSR format. Thus this design, in total, consumes 3$\times$ of the space of a CSR alone design. Another work [13] suffers the same issue and, worse, further requires an additional memory space to store a bitmap data structure. Using Twitter Graph [33] (60M nodes and 1.4B edges) as an example, the bitmap consumes about 2 - 10 GB based upon various configurations, while the CSR format takes about 4GB memory space.

In this work, TRICORE preserves the edge-centric computing model with partitioning the CSR format graph in 2-D vertical manner so that each CSR, i.e., CSR_0, ... CSR_k in Figure 3 can fit in GPU memory. Meanwhile, all GPUs use a streaming buffer to, on-the-fly, load the edge list in GPU memory for intersection. The size of the streaming buffer is tuned so that loading and computing are perfectly overlapped.

IV. TRICORE: PARALLEL BINARY SEARCH-BASED INTERSECTION ON GPU

This section starts with our novel binary search-based intersection algorithm. Afterwards, we detail our optimizations. Eventually, we provide the theoretical analysis to explain why this algorithm bests traditional merge-based intersection.

A. Binary Search-based Intersection

This proposed binary search-based intersection algorithm takes the two neighbor lists of an edge as inputs. In particular, it selects one neighbor list as the lookup list while the other as binary search list. During intersection, it checks whether each entry of the lookup list appears in the binary search list. The basic idea is illustrated in Figure 5. In practice, instead of constructing a real binary search tree, we simply use a sorted array for the binary search list. In each iteration, it compares the lookup element to the middle of the binary search list, such as the root vertex 5 in the binary search tree. While descending, the search-space halves. For instance, the process of lookup vertex 1 checks vertex 5, 3 and 0 iteratively. The detailed code is shown in Algorithm 3.

Algorithm 3 Parallel binary search-based intersection for A∩B

```
1: Assuming |A| < |B| and count = 0;
2: foreach x ∈ A in parallel do
3:     bottom = 0, top = |B| − 1;
4:     while bottom < top − 1 do
5:         mid = (top + bottom) >> 1;
6:         if x < B[mid] then
7:             top = mid;
8:         else if x > B[mid] then
9:             bottom = mid;
10:        else                                   ▷ x == B[mid]
11:            count++; break;
12:        end if
13:    end while
14: end for
```

TRICORE always uses the longer neighbor list as a binary search tree while the shorter one as the lookup list to minimize the time complexity which we will discuss shortly. By using binary search, the time complexity of the intersection function of $e = (u, v)$ is $O(m * \log n)$ where $m = \min\{d(u), d(v)\}$ and $n = \max\{d(u), d(v)\}$. As a result, the overall time complexity $T = \sum_{e \in E} O(m_{(e)} \cdot \log n_{(e)})$. The worst case complexity is reached when the graph is a complete graph, thus with $\sqrt{|E|}$ vertices and $|E|$ edges, then takes $O(|E|^{3/2} \cdot \log \sqrt{|E|})$ time cost when the graph is a clique.

In terms of time complexity, We have to admit that binary search-based intersection is often worse than merge-based designs on CPUs. Specifically, for an arbitrary edge (u, v), we still use the notation $m = \min\{d(u), d(v)\}$, $n = \max\{d(u), d(v)\}$. Merge and binary search-based intersections present $O(m + n)$ and $m \cdot \log(n)$ time complexities, respectively. The following observations are straightforward:

- $m \ll n$: Binary search provides better time complexity.
- $m \sim n$: Merge-based intersection excels.

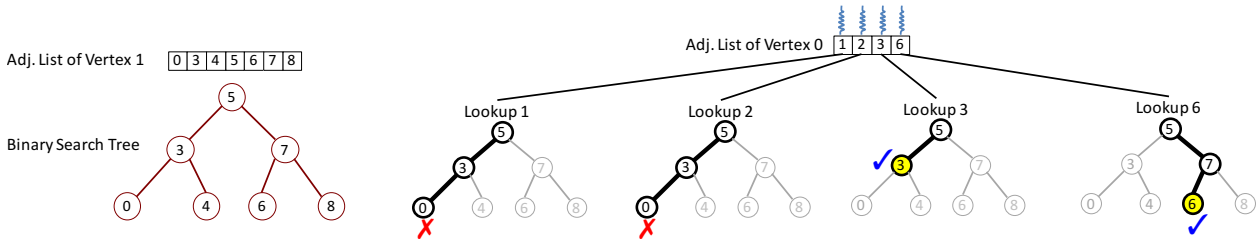


Fig. 5: Binary search-based intersection: one array for the lookup list while the other for the binary search tree. Note, the binary search tree is simply implemented as a sorted array.

Firstly, the orientation step from compact-forward design [35], [51], [24], [27], [7] tends to modify the graphs towards the latter condition. Further, binary search-based design has higher worst case complexity $O(|E|^{1.5} \log |E|)$ when the graph is a clique. Another reason that forces binary search-based algorithm to lose the edge on CPU is its poor cache behavior. That is, merge-based algorithm always reads the two neighbor lists sequentially, assuming one CPU thread works on one edge. In contrast, binary search algorithm tends to jump through the binary search list in a strided manner.

B. Parallel Binary Search on GPU

Despite of those drawbacks on CPUs, binary search-based algorithm shows unique advantages on GPUs largely due to GPU's particular features, such as SIMD architecture, coalesced memory access design, and manually controllable shared memory. This section unveils the parallelization and memory access patterns of our binary search-based intersection algorithm on GPUs.

In practice, a group of $p = 32$ threads, i.e., a warp, is used to work on each query edge $e = (u, v)$ as we have already discussed in Observation #2. An intersection between two neighbor lists with lengths m and n ($m < n$) yields m independent binary search lookups on binary search list with length of n. In each iteration, 32 lookups are executed simultaneously by a warp. This design makes our binary search-based algorithm immediately better than the merge-based counterpart.

Coalesced memory access. Binary search-based intersection introduces more friendly memory access patterns than that of merge-based design in two ways. First, as shown in Figure 5, accessing lookup key arrays is both sequential and consecutive as four adjacent threads will load four consecutive lookup keys. Further, because each lookup key follows the same order to check the binary search tree, accessing the binary search tree array also experiences high cache hit ratio. Along with a shared memory caching technique we will discuss later, the performance counter shows optimized binary search intersection reduces the load transactions by 5× comparing to merge-based approach.

Workload balancing. With the binary search intersection approach, workload imbalance can be tackled much easier – there is no need for partitioning and distributing workload across threads in the warp. Still using the two neighbor lists as an example in Figure 5, we assign a group of four threads to work on four lookup keys in parallel. Our result further shows that majority of the searches often reach the leaf nodes of the binary search tree. Therefore the workloads across threads are almost balanced.

GPU shared memory optimization. As we have already mentioned, each lookup key accesses the same binary search tree repeatedly. One immediate optimization is to cache such a tree in GPU shared memory in order to avoid expensive global memory access. However, due to the size limit of GPU shared memory, fitting the entire tree in the shared memory would introduce low thread occupancy issue and jeopardize the overall performance.

Instead, we cache the top levels of the tree in the shared memory as they are the most frequently accessed ones. For example in Figure 5, the root node of the binary search tree which stores vertex 5 is accessed by all lookups. In general, elements from levels that are closer to root are more likely to be accessed repeatedly. In particular, in level k, the possibility of each element being accessed is $\frac{1}{2^{k-1}}$ because the total number of elements in level k is 2^{k-1} and only one element will be accessed by one lookup key at each level. We thus cache the first k levels of the tree in GPU shared memory. The binary search-based intersection now consists of two phases. First, all threads search on the cached array in shared memory until a match is found. Otherwise, they continue to do so on a subrange in the global memory.

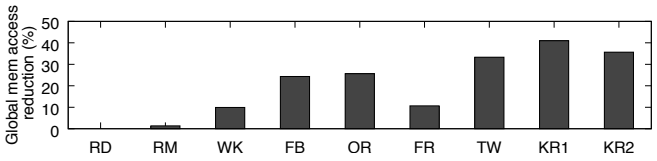


Fig. 6: Percentage of GPU global memory access transactions reduced by shared memory optimization.

Figure 6 demonstrates the global memory access reduction (i.e., $gld\_transactions$) introduced by the shared memory optimization. The caching technique brings averagely 18% memory transaction reduction with the maximum from KR1 to be 40.2%. We also notice a slight increase (i.e., 2.4%) for the RD synthetic graph. The reason lies in that this optimization works well for the graphs with a considerable number of high degree vertices, especially the real graphs with skewed distribution. Our analysis finds that for this small degree graph, the number of memory transactions needed for preloading the first few levels of the tree into shared memory

is much larger than what is required for eventual lookups. Obviously, the caching technique would not beneficial in this case. Fortunately, many real world datasets follow the power-law degree distribution, different from this synthetic RD graph with uniform (small) degree distribution.

C. Memory Complexity Analysis

This section analyzes the memory complexities of both binary search and merge-based algorithms to demystify why the former one always outperforms the latter one on GPUs. Note triangle counting operations are mainly memory intensive, thereby memory complexity can closely indicate its performance. For the reader's convenience, as shown in Table II, we again mention that for edge (u, v), we assume $m = \min\{d(u), d(v)\}, n = \max\{d(u), d(v)\}$, and we use a group of p threads to work on this edge in both algorithms. Each transaction can load as many as B data records. The best case to access N consecutive data entries takes N/B transactions while the worst case takes N transactions. Note we assume a linear relationship between B and p, i.e., $B = c{\cdot}p$. Specifically, to align with GPU convention, we use a warp with $p = 32$ as a group, and $B = 16$ as half-warp coalesced memory read.

Notation	Description
(u, v)	An edge
$d(u)$, $d(v)$	Degree of the two vertices
m, n	The smaller and larger degrees of $d(u)$ and $d(v)$
p	Number of threads for finer-grained parallelization
B	Size of each memory transaction
s_1, s_2	The factors of strided access
C	Number of nodes cached in shared memory
k	Levels of cached binary tree

TABLE II: The notations used in this paper.

Table III summarizes the time and memory transaction complexities of merge- and binary search- based intersection methods. In particular, the merge-based algorithm suffers from strided memory access with the strided factors $s_1 = \min\{B, m/p\}$ and $s_2 = \min\{B, n/p\}$ (s_1 and s_2 are the stride lengths of accesses on both neighbor lists) which causes s_1 and s_2 times more transactions correspondingly. Partitioning takes $p \cdot \log n$ random memory accesses. Thus, the algorithm takes $s_1 m/B + s_2 n/B + p \cdot \log m$ memory transactions.

Algorithm	#Operations	#Memory transactions
Merge-based	$m + n + p \cdot \log n$	$s_1 m/B + s_2 n/B + p \cdot \log m$
Binary search	$m \cdot \log n$	$m/B + m(\log n - k) + C$

TABLE III: #Operations and #memory transactions of merge- and binary search- based intersection algorithms.

Binary search algorithm generates three possible memory traffics: the shorter neighbor list, the longer neighbor list, and caching. For the shorter neighbor list, it uses coalesced memory access pattern which takes m/B transactions. For the longer one, it costs $m \cdot \log n$ random accesses in the worst case. Assuming the shared memory can cache C elements, the first $k = \log C$ levels of the binary search tree can be accessed from the global memory only once, leading to $m \cdot k$ transactions reduced with p overhead instead. Taken together, binary search-based algorithm takes $m/B + m(\log n - k) + C$ memory transactions in the worst case.

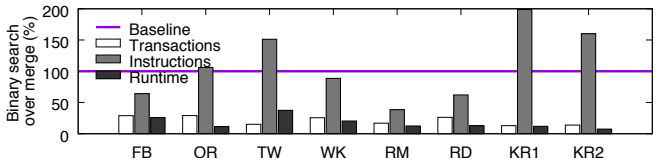


Fig. 7: Binary search-based algorithm over merge-based algorithm in terms of #memory transactions, #instructions and runtime.

For any intersection (u, v), since m and n, the degrees of both vertices, can decide the memory costs, we explore three cases of them to evaluate the advantages and disadvantages of both algorithms. Here, a common threshold θ is exploited to classify edges by the degrees of its vertices.

1. Both m and n are small numbers. In general, if m and n are smaller than θ, the stride factor s_1 and s_2 are also small. In this case, the overhead of partitioning the intersections from merge-based algorithm takes the majority of the runtime. In the binary search algorithm, most of the memory transactions to the binary search tree can be cached by the shared memory with the average cost on caching reducing to a small number since all C reads can be done by a small number of coalesced memory reads. In summary, binary search-based algorithm will excel in this case.

2. m is small number while n is large number. When $d(u) \ll d(v)$, one of the two vertices has large degree while the other has small degree. As we discussed, binary search will yield much better complexity than merge-based intersection. Thus, binary search takes fewer operations and instructions than merge-based algorithm on these graphs.

3. Both m and n are large numbers. This case takes the majority of total time cost on a collection of real-world graphs from this work. In a nutshell, binary search-based intersection still provides surprisingly better performance even it suffers from higher computation complexity (i.e. loads more number of data elements). The reason is that binary search intersection provides more efficient global memory access on both neighbor lists. In addition, since the lookup lists are also sorted, threads of a warp at any iteration are more likely to search in the same range of the binary search tree, leading to low thread divergence and largely coalesced memory access. Thus, the average memory access on binary search tree can be approximately reduced to $1/p$ of its original transaction counts. Overall, this case, which dominates the total time consumption, yields the highest memory efficiency benefits.

As shown in Figure 7, we also profile the binary search and merge -based implementations on GPU for the real datasets. Using merge-based algorithm as baseline, binary search, on average, executes merely 10% more instructions, partly due to the caching mechanism. However, binary search only conducts $\sim \frac{1}{5}$ of the total global memory transactions from merge-based algorithm, resulting in 5.9× speedup.

V. Scalable TriCore

A. Big Graph Challenge

Triangle counting algorithm has $O(|E|^{1.5})$ computational complexity, which can be time consuming when graph size

increases. Scaling to multiple GPUs, TRICORE faces two more challenges when distributing computations. First, the proposed TRICORE (Section III) requires both edge list and CSR formats of the entire graph to be present in the memory of every participating GPU, which restricts the size of the graph that can be addressed. Second, workload imbalance also surfaces while distributing computations across GPUs. In particular, this imbalance issue is distinct from conventional ones that appear in BFS [36], [38], [31] and PageRank [62].

This section addresses the aforementioned challenges in order to extend TRICORE to big graphs. First, for edge list, we propose to stream it into the GPU while counting triangles. The key of this design is to minimize the overhead of streaming in edge list. Besides, when CSR format of the graph cannot fit in GPU memory, we further adopt an I/O-efficient, external memory triangle counting framework [29] to partition the CSR across multiple GPUs. And the key of this design is to avoid communications across various CSR partitions. Lastly, we propose a dynamic workload management component to balance workload across GPUs.

B. Memory Consumption Reduction

This section describes our designs to reduce memory consumption for edge list and CSR formats, respectively.

Edge list space consumption reduction. The good news is that TRICORE does not need the entire edge lists in memory to do triangle counting since different query edges are independent. Therefore, we store the CSR format of the graph in the GPU and exploit a small buffer in GPU memory to stream the edge list in GPU during the computation. Clearly, the size of the streaming buffer is an important parameter: we do not want this size to be too large in order to saturate the GPU computation power because that will drain the small GPU memory space as well. Fortunately, as shown in Figure 8, TRICORE can achieve peak performance with merely a 4 MB streaming buffer on both K40c and Titan X GPUs for a variety of graphs. Further, this peak performance sustains from 4 MB to hundreds of MB – indicating a wide range of options for streaming buffer sizes. As such, we use a trivial 16MB GPU memory buffer and an equal sized pinned memory from the CPU to stream the edge list from CPU to GPU, which enables TRICORE to accommodate graphs that are 3× of the size from existing projects [27], [13]. Edge lists are divided into $|E|/b$ chunks and each of them has the same number of edges. With multiple GPUs, each GPU keeps the same CSR in its memory and stream in different edge chunks for triangle counting.

In summary, this technique allows us to handle the edge list with a small buffer, which is important to enable GPU-based triangle counting on large graphs. Meanwhile, it incurs almost no adversarial performance penalties.

CSR space consumption reduction. We further adopt a 2-D partitioning scheme that is used by external memory triangle counting algorithm [29] to enable TRICORE for graphs whose CSR cannot fit in GPU memory. In simple terms, this design partitions graph by both source and destination vertices of each

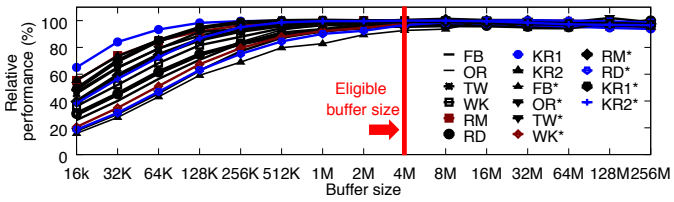


Fig. 8: Relative performance with respect to various buffer sizes. The legend with '*' means the result is on Titan X, otherwise it is on K40c. For example, TW and TW* mean the results of TW on K40c and Titan X, respectively.

edge. The global triangle counting task is divided into sub-tasks, each sub-task can be seen as a local triangle counting task on a subgraph which can fit in any certain memory size. In specific, each sub-task needs to load two CSR partitions and one edge list partition to GPU memory, which is read from a bigger secondary storage such as disk, Solid-State Drive (SSD), or CPU memory. This differs from external memory triangle counting efforts in the sense that the GPU triangle counting kernel cannot actively invoke memory copy from CPU memory to GPU through GPU threads. Instead, all GPU memory copy operations have to be done before the computation on GPUs. Therefore, our algorithm scans through a collection of edges from the edge list to decide the to be loaded CSR partitions. Further, all sub-tasks are independent to each other, thus can be distributed to multiple GPUs.

C. In Memory Multi-GPU Implementation

The aforementioned distributed implementation can tackle big graphs. However, if the graph can fit in multi-GPUs of the same machine, using a distributed system is not necessary. Note, equipping multiple GPUs on a single machine is a very popular case. This section, as a complement, introduces another method to balance workload for the single-node multi-GPU triangle counting.

In this case, we use a work stealing scheduling method to balance the workload across participating devices. Each stealing takes a chunk of edge list which has the same size with the streaming buffer. The method starts with static work distribution and enables workload stealing when any GPU finishes its job and workload imbalance surfaces. As shown in Figure 9, each GPU is initially assigned 3 edge list chunks to work with. Once GPU 1 and 3 finish their tasks, they steal task from other unfinished GPUs, e.g., chunk 6 from GPU 2 to GPU 1, also GPU 3 steals two tasks from GPU 4.

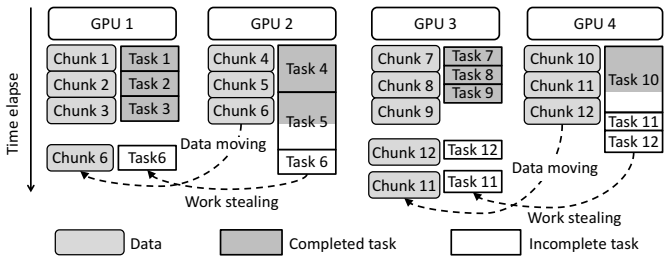


Fig. 9: Dynamic balancing. When a GPU finished its tasks, it finds unfinished tasks on other GPUs and launch in the reversed order.

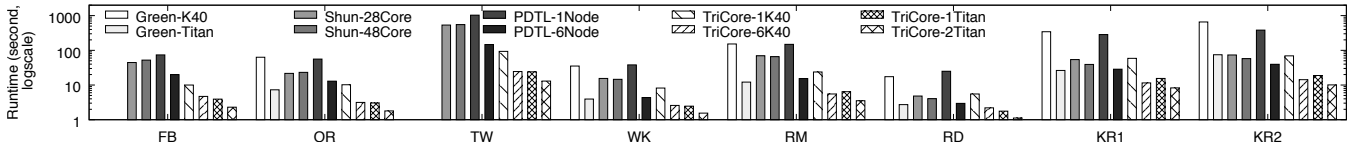


Fig. 10: The runtime of TRICORE and three related projects: Green [27] (on single K40c and single Titan X GPU), Shun [51](on 28-core CPU and 48-core CPU) and PDTL [24] (on single node and 6 nodes of 16-core CPU). TRICORE-1K and TRICORE-6K are our results on one and six K40c GPUs respectively. Similarly, TRICORE-1Titan X and TRICORE-2Titan X are our results on one and two Titan X GPUs respectively. All these GPU results are on a single machine.

D. External GPU Memory Distributed Implementation

We further discuss the distributed system implementation which relies on Message Passing Interface (MPI) [52] to manage GPUs and process extremely large graphs. Assuming we have N GPUs in a distributed system, our design needs two kinds of process, namely, GPU-process and scheduler-process. On one hand, we use each GPU-process to manage one GPU, with a total of N GPU-processes. On the other hand, we set one scheduler-process to coordinate all GPU-processes and make all GPUs busy. In detail, every time when the scheduler-process receives idle signal from one GPU-process, it immediately sends next sub-task ID to this GPU so that this GPU will work on the assigned sub-task.

Since each sub-task can have varied amount of workloads, the main challenge of this distributed system is still workload imbalance. Note that partition will not introduce communications, thus communication will not be a problem. As the number of sub-tasks is much larger than the number of GPUs, we can thus intelligently schedule the sub-tasks across GPUs to resolve the imbalance issue. Assuming each node has a similar amount of GPUs, Peripheral Component Interconnect-express (PCI-e) bandwidth, and secondary storage I/O bandwidth, our distributed settings can achieve close to ideal scalability.

VI. EXPERIMENTS

TRICORE is implemented in 2,000 lines of C++ and CUDA code. We use CUDA 7.5 toolkit, including nvcc and nvprof, and GCC 4.8.5 and OpenMP 3.1 to compile the source with compilation flag set to $-O3$. In terms of storage format, we use uint32 and uint64 to represent vertex ID and begin position, respectively.

TRICORE is rigorously studied with the following configurations: A high-end server with Intel Xeon E5-2683 28-core processors, another high-end server with four Intel Xeon E7-8857v2 3.0 GHz 12-core processors and 2 TB of RAM, a server with dual Intel Xeon E5-2620 6-core CPU and six NVIDIA Kepler K40c GPUs and 128 GB of RAM, an Alienware desktop installing Intel i7-8700 CPU and two NVIDIA Titan X GPUs and 16 GB of RAM, as well as a cluster with each CPU node equipping two Intel Xeon E5-2650v2 8-core processors with 128 GB of RAM and each GPU node equipping two Intel Xeon E5-2620 6-core processors with NVIDIA K20 GPU and FDR InfiniBand network interconnect. Particularly, K40, K20 and Titan X contain 2,880, 2,496 and 3,584 CUDA cores, respectively, while E5-2683, E5-2650v2, E5-2620 and i7-8700 CPUs come with 28, 12, 8 and 6 cores respectively. With hyper-threading, all CPUs can support hardware threads up to $2\times$ the number of physical cores. The prices of all CPUs and GPUs used for triangle counting are listed in Table IV. We find that TriCore has a much lower total cost (including GPU and host CPU), while delivering better performance than many CPU alternatives.

Processors	Price ($)
Dual socket Intel Xeon E5-2650v2 (16-core, 8-core/socket)	900
Dual socket Intel Xeon E5-2683 (28-core, 14-core/socket)	3,800
Quad socket Intel Xeon E7-8857v2 (48-core, 12-core/socket)	12,000
Tesla K20c GPU + Intel Xeon E5-2620 host CPU	800 + 130
Tesla K40c GPU + Intel Xeon E5-2620 host CPU	1,256 + 130
Titan X Pascal GPU + Intel i7-8700 CPU	1,200 + 300

TABLE IV: Costs of GPUs and CPUs.

Orientation. All results of TRICORE and related works [27], [51], [24] use oriented graphs as input. We briefly report the impacts of orientation for the best implementations of CPU and GPU: merge-based algorithm on CPU, and binary search-based algorithm on GPU. For all datasets the rank-by-degree orientation brings averagely $53\times$ speedup for merge-based algorithm on CPU, and $10\times$ speedup for TriCore on GPU.

A. TRICORE *Performance*

This section studies the performance comparison of TRICORE and three state-of-the-art triangle counting projects including the ones both on CPUs and GPUs. In specific, these three projects include one single GPU work – Green et al. [27], and a single-node work – Shun et al. [51] and distributed project – PDTL [24]. In terms of testbed, TRICORE is evaluated with up to six and two K40c and Titan X GPUs, respectively. Shun runs on both 28-core and 48-core high end servers. PDTL is examined with up to six 16-core servers.

As shown in Figure 10, for a single K40c case, TRICORE achieves $3.1\times$ to $9.5\times$ of (minimum on RD and maximum on KR2) and $5.9\times$ average speedup over Green et al on a single K40c GPU. The results show that TRICORE can bring considerable speedup on big graphs and graphs with skewed distribution, although the benefit is smaller on small graphs such as RD and WK. TRICORE on a single K40c GPU achieves $0.87\times$ to $5.7\times$ (minimum on RD and maximum on TW) and on average $2.5\times$ speedup over Shun et al. This is also $2.4\times$ to $12.7\times$ (minimum on TW and maximum on FB) and on average $6.5\times$ faster than PDTL.

For six K40c on a single machine, the speedups over the three related project (Green, Shun and PDTL) increase to $24\times$, $9\times$ and $23\times$ correspondingly. TRICORE on six K40c GPUs achieves $1.3\times$ to $6\times$ (minimum on RD and maximum on TW)

and on average 2.6× speedup over PDTL on 6 nodes each with 16-core CPU.

TRICORE achieves 1.6× to 4× of (minimum on RD and maximum on KR2) and 2.2× average speedup over Green on a single Titan X GPU. We notice that the speedup of TRICORE over Green et al. declines from K40c to more powerful Titan X GPUs. The reason is that Green et al. can only tackle small graphs, for which TRICORE already achieves good performance on a K40c GPU. Therefore, more powerful Titan X GPU cannot add too much benefits to TriCore for these small graphs. Figure 10 also suggests that TRICORE achieves higher speedups over Green et al. when the graph is larger. Besides, TRICORE on a single Titan X GPU achieves 2.7× to 22× (minimum on RD and maximum on TW) and on average 8.5× speedup over Shun on 48-core. TRICORE on two Titan X GPUs achieves 2.6× to 11× (minimum on RD and maximum on TW) and averagely 4.4× speedup over PDTL on six nodes.

B. Comparisons with Graph Challenge Champions

This section compares TRICORE against all the champions of graph challenge 2017 [1] which centers around two problem, triangle counting and k-truss. We compare TRICORE against all those projects focusing on triangle counting [12], [64], [46]. Briefly, TCKK [64] exploits linear algebra kernels to count triangles on a single machine, which installs E5-2698 v3 intel CPU processor with 32-core and 512 gigabytes memory, with 64 threads. Another work [12] (also referred as Nv) uses one Nvidia Titan X pascal GPU (host CPU and RAM are not specified), more details can be found in [13]. The third work, Pearce et al. [46], exploits 256 machines, each of which equips 24-core Intel E5-2695 v2 CPUs with 128 GB memory. Note, since these projects are not open source, we directly cite their numbers from their papers and all datasets are readily available from [2]. Figure 11 presents the performance of TRICORE, Nv and TCKK on four Graph 500 datasets from scale 22 to 25, and Twitter. TRICORE outperforms all champions, e.g., 2.7× and 1.9× faster than Nv and TCKK, respectively.

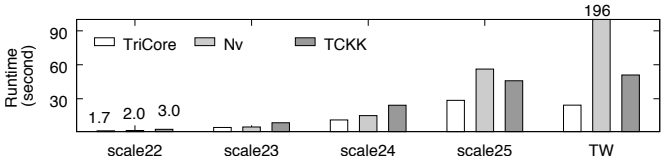


Fig. 11: TRICORE vs Graph Challenge champions. Note, similarly to Nv, the time of TRICORE is on one Titan X GPU.

To compare against the champion in distributed system [46], we use two merely GPUs. As shown in Table V, [46] uses 256 compute nodes to finish Twitter in 11.2 seconds while we use two GPUs to finish the same graph within roughly comparable time, i.e., 13 seconds. Besides, our scalability from one to two GPUs is 1.84×.

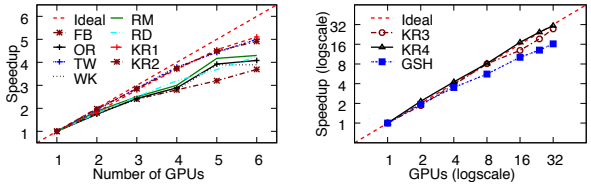
Project	TRICORE	TRICORE-2GPU	Pearce et al. [46]
Runtime (second)	24	13	11.2
Hardware	1 Titan X	2 Titan X	256 nodes

TABLE V: Runtime (second) of TRICORE vs 2017 Graph Challenge champions on Twitter graph.

C. Scalability

Figure 12(a) studies the scalability of TRICORE across multiple GPUs on a single machine on all middle sized graphs. Briefly, 6 GPUs bring, on average, 4.4× speedup across those graphs with 3.7× and 4.9× from KR2 and KR1 as the minimum and maximum speedup. In short, the runtime decreases by 1.8× as the number of GPU doubles.

Figure 12(b) further evaluates TRICORE's capability of processing extremely large graphs with a collection (i.e., 1 - 32) of K20 GPUs in a cluster. Note, since the MPI version uses dynamic scheduling and thus each node decides which partition to load on the fly, the runtime includes the time cost of loading data from disk. In particular, TRICORE yields 24× speedup on average across these three large graphs with 16× and 30× of GSH and KR4 as the minimum and maximum speedups. This is a speedup of 1.9× as the number of GPU doubles. It is worthy of noting that the sizes of these three datasets in CSR format are 131GB, 75GB and 143GB, which are over 15× larger than the memory size of each K20 GPU which has 5GB memory.



(a). Single node. (b). Multiple nodes.

Fig. 12: Scalability of counting Twitter on GPUs installed in (a) single machine, and (b) distributed system.

VII. CONCLUSION

This work presents TRICORE, a scalable triangle counting system on GPUs that is able to compute exact triangle counts on big graphs. TRICORE makes two main contributions, namely, a fast GPU-aware triangle counting algorithm and a scalable framework to process very large graph on many GPUs. As such, TRICORE can process graphs which are orders of magnitude larger than GPU memory sizes and achieve great scalability. Our evaluation of TRICORE on a number of graphs shows that TRICORE can greatly outperform existing endeavors including most recent champions in Graph Challenge.

ACKNOWLEDGMENT

We thank the anonymous reviewers and Hans-Edward Hoene for their helpful suggestions and gratefully acknowledge the NVIDIA Corporation for the donation of the Titan Xp GPUs. This work was supported in part by National Science Foundation CAREER award 1350766 and grants 1618706 and 1717774.

REFERENCES

[1] DARPA HIVE GraphChallenge, https://graphchallenge.mit.edu/darpa-hive.

[2] Graph Challenge Datasets, http://graphchallenge.mit.edu/data-sets.

[3] Intel Xeon E5 2683 v3 Processor, https://ark.intel.com/products/81055/Intel-Xeon-Processor-E5-2683-v3-35M-Cache.

[4] Kronecker: Graph 500 Generator, https://graph500.org/?page_id=12#sec-3.

[5] NVIDIA TESLA V100 GPU ACCELERATOR, http://www.nvidia.com/content/pdf/volta-datasheet.pdf.

[6] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 1997.

[7] S. Arifuzzaman, M. Khan, and M. Marathe. Patric: A parallel algorithm for counting triangles in massive networks. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, 2013.

[8] S. Arifuzzaman, M. Khan, and M. Marathe. A fast parallel algorithm for counting triangles in graphs using dynamic load balancing. In *Big Data*. IEEE, 2015.

[9] Ariful Azad, Aydin Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 804–811. IEEE, 2015.

[10] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *SIGKDD*. ACM, 2008.

[11] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 93–104. ACM, 2017.

[12] Mauro Bisson and Massimiliano Fatica. Static graph challenge on gpu. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–8. IEEE, 2017.

[13] Mauro Bisson and Massimilliano Fatica. High performance exact triangle counting on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[14] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[15] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[16] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *SC*. ACM, 2011.

[17] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.

[18] R. Burt. Structural holes and good ideas1. *American journal of sociology*, 2004.

[19] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, 2004.

[20] J. Coleman. Social capital in the creation of human capital. *American journal of sociology*, 1988.

[21] Disa Mhembere Da Zheng, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 45–58, 2015.

[22] J. Eckmann and E. Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proceedings of the national academy of sciences*, 2002.

[23] E. Elenberg, K. Shanmugam, M. Borokhovich, and A. Dimakis. Beyond triangles: A distributed framework for estimating 3-profiles of large graphs. In *SIGKDD*. ACM, 2015.

[24] I. Giechaskiel, G. Panagopoulos, and E. Yoneki. Pdtl: Parallel and distributed triangle listing for massive graphs. In *ICPP*. IEEE, 2015.

[25] Minas Gjoka, Maciej Kurant, Carter T Butts, and Athina Markopoulou. Practical recommendations on crawling online social networks. *IEEE Journal on Selected Areas in Communications*, 29(9):1872–1892, 2011.

[26] O. Green, R. McColl, and D. Bader. Gpu merge path: a gpu merging algorithm. In *Proceedings of the 26th ICS*, 2012.

[27] O. Green, P. Yalamanchili, and L. Munguía. Fast triangle counting on the gpu. In *Proceedings of the Fourth Workshop on Irregular Applications: Architectures and Algorithms*, 2014.

[28] S. Hong, S. Kim, T. Oguntebi, and K. Olukotun. Accelerating cuda graph algorithms at maximum warp. In *ACM SIGPLAN Notices*, 2011.

[29] Yang Hu, Pradeep Kumar, Guy Swope, and H. Howie Huang. Trix: Triangle counting at extreme scale. Technical report, Department of Electrical and Computer Engineering, The George Washington University, 2017.

[30] Edward Kao, Vijay Gadepally, Michael Hurley, Michael Jones, Jeremy Kepner, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Siddharth Samsi, William Song, et al. Streaming graph challenge: Stochastic block partition. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–12. IEEE, 2017.

[31] Pradeep Kumar and H Howie Huang. G-store: high-performance graph store for trillion-edge processing. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 830–841. IEEE, 2016.

[32] J. Kunegis. Konect: the koblenz network collection. In *International conference on World Wide Web companion*. International World Wide Web Conferences Steering Committee, 2013.

[33] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. ACM, 2010.

[34] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, 2012.

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

[36] H. Liu and H. Huang. Enterprise: breadth-first graph traversal on gpus. In *SC*, 2015.

[37] Hang Liu and H Howie Huang. Graphene: Fine-grained io management for graph computing. In *FAST*, pages 285–300, 2017.

[38] Hang Liu, H Howie Huang, and Yang Hu. ibfs: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*, pages 403–416. ACM, 2016.

[39] K. Madduri and D. Bader. Gtgraph: A suite of synthetic random graph generators, 2012.

[40] Duane Merrill and Michael Garland. Merge-based parallel sparse matrix-vector multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 58. IEEE Press, 2016.

[41] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.

[42] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 29–42. ACM, 2007.

[43] CUDA Nvidia. Programming guide, 2008.

[44] R. Pagh and C. Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Information Processing Letters*, 2012.

[45] H. Park and C. Chung. An efficient mapreduce algorithm for counting triangles in a very large graph. In *International conference on Conference on information & knowledge management*, 2013.

[46] Roger Pearce. Triangle counting for scale-free graphs at scale in distributed memory. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–4. IEEE, 2017.

[47] Roger Pearce, Maya Gokhale, and Nancy M Amato. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 549–559. IEEE, 2014.

[48] M. Rahman and M. Al Hasan. Approximate triangle counting algorithms on multi-cores. In *BigData*, 2013.

[49] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment*, 11(4), 2017.

[50] C Seshadhri, Ali Pinar, and Tamara G Kolda. Wedge sampling for computing clustering coefficients and triangle counts on large graphs.

Statistical Analysis and Data Mining: The ASA Data Science Journal, 7(4):294–307, 2014.

[51] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *Proceedings of the IEEE ICDE*, 2015.

[52] Marc Snir, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *MPI–the Complete Reference: the MPI core*, volume 1. MIT press, 1998.

[53] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *International conference on World wide web*, 2011.

[54] C. Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *ICDM*, 2008.

[55] C. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Network Analysis and Mining*, 2011.

[56] C. Tsourakakis, U Kang, G. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *SIGKDD*. ACM, 2009.

[57] Chad Voegele, Yi-Shan Lu, Sreepathi Pai, and Keshav Pingali. Parallel triangle counting and k-truss identification using graph-centric methods. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–7. IEEE, 2017.

[58] Jia Wang and James Cheng. Truss decomposition in massive networks. *Proceedings of the VLDB Endowment*, 5(9):812–823, 2012.

[59] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D Owens. A comparative study on exact triangle counting algorithms on the gpu. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, pages 1–8. ACM, 2016.

[60] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 41–53. ACM, 2018.

[61] W. Wang, Y. Gu, Z. Wang, and G. Yu. Parallel triangle counting over large graphs. In *Database Systems for Advanced Applications*, 2013.

[62] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 11. ACM, 2016.

[63] D. Watts and S. Strogatz. Collective dynamics of 'small-world'networks. *nature*, 1998.

[64] Michael M Wolf, Mehmet Deveci, Jonathan W Berry, Simon D Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with kokkoskernels. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–7. IEEE, 2017.