# Fast and Adaptive List Intersections on the GPU

James Fox, Oded Green, Kasimir Gabert, Xiaojing An, and David A. Bader

Computational Science and Engineering, Georgia Institute of Technology - USA

*Abstract*— **List intersections are ubiquitous and can be found in a wide range of applications, including triangle counting and finding the maximal $k$-truss, both of which are part of the HPEC Static Graph Challenge. For many graph based problems it is necessary to find intersections for a very large number of lists—these lists tend to vary greatly in size and are difficult to efficiently load-balance. Numerous parallel algorithms on list intersections for triangle counting have been proposed, but load-balancing decisions are typically made at a global level. In this paper we present an efficient and adaptive approach to load-balancing at a finer granularity. Our approach assigns a different number of threads for different intersections in order to effectively utilize the resources of the GPU. We show the applicability of our load-balancing method to two different intersection methods, one search-based and one merge-based. Our algorithm outperforms several recent triangle counting algorithms, including recent HPEC Graph Challenge Champions.**

## I. Introduction

List intersections is a widely used kernel for numerous applications. These include triangle counting and enumeration, community detection, missing link prediction, and finding $k$-trusses. The latter application is one of the HPEC GraphChallenge problems, and the authors' prior participation experience [10] emphasized the necessity of a high-performance triangle counting kernel that performs well across a broad range of inputs. This can be challenging as inputs vary significantly in scale, degree distribution, and other graph characteristics. A high performing intersection kernel that ensures good performance in light of these factors is the primary focus and contribution of this work.

A list intersection is formulated as: given an edge $(u, v)$ find the set $adj(u) \cap adj(v)$, where $adj(u)$ indicates adjacency list of vertex $u$. Triangle counting seeks to count all such intersections for every edge in the graph.

The intersection of two sorted lists can be done in multiple ways—the two basic ones addressed in this work are 1) **Sorted Set Intersection** and 2) **Binary Search**. In sorted set intersection, common elements are found by moving across the two sorted adjacency arrays in a merge-like fashion. In binary search, each element in the smaller array is looked up in the larger of the two sorted arrays, with a match indicating a triangle.

Naive parallelization of triangle counting on massively multi-threaded systems, e.g. assigning single intersections per thread, can result in poor performance and scalability. Parallel processing in the context of non-uniform graphs (e.g. power law graphs) can easily lead to underutilization due to load imbalance. This is especially costly for GPUs with thousands of cores and approximately ten times that number of active threads.

Prior work in Green et. al. [17] introduced a method of dividing a Sorted Set Intersection into independent sub-intersections. In this paper we also propose a scheme for dividing up a Binary Search based intersection. Altogether, these sub-intersections can be computed in parallel. We refer to these parallel versions as **Intersect-Path** and **Par-Search** in order to distinguish from their respective sequential versions. Details of Intersect-Path and Par-Search are discussed in later sections.

The above steps allow an arbitrary number of threads to work on a single intersection, opening the door to better load-balancing. However, the number of threads per intersection poses a tuning problem, and furthermore the optimum is graph-dependent. The wide variety of GraphChallenge datasets underscores the need for a generalized load-balancing approach that is input-adaptive, cost-effective, and fast.

### *Summary of contributions*

- We extend Logarithmic Radix Binning [11], an intersection-granularity approach for binning intersections of similar estimated work, to triangle counting on the GPU.
- We present Logarithmic Threads Per Intersection, a new approach for scaling the number of threads per intersection in accordance with the Logarithmic Radix Binning ordering of edges. The collective of Logarithmic Radix Binning and Logarithmic Threads Per Intersection constitutes a new scheme that provides better load-balancing of list intersections on GPU cores.
- We apply our new load-balancing approach to two different list intersection methodologies, Intersect-Path and Par-Search, and present their respective parallel triangle counting implementations on the GPU.
- Finally, we compare our new algorithm against several high performing triangle counting algorithms, including [22], [28], [4], [11]. Several of these were the fastest triangle counting algorithms in the 2017 HPEC Graph Challenge [23]. Our algorithm is faster than all implementations compared against, but for a small number of instances, across a wide range of test graphs.

## II. Related Work

The applications in which triangle counting and enumeration are used is broad. It became an important metric to data scientists with the introduction of clustering coefficients

[27]. Other applications for triangle counting include finding transitivity [16], spam detection in email networks [2], finding tightly knit communities [20], finding $k$-trusses [7], [25], [10], and evaluating the quality of different community detection algorithms [15], [29]. Several triangle counting applications are discussed in detail in [6]. Triangle counting is also a key kernel for the HPEC Graph Challenge [21].

*a) Computational Approaches:* Given a graph $G = (V, E)$, there are several approaches for counting triangles: enumerating over all node triplets $O(|V|^3)$, using linear algebra operations $O(|V|^w)$ (where $w < 2.376$), and adjacency list intersection. One of the first to differentiate between and bound these approaches was Schank *et. al.* [24]. The adjacency list intersection can be completed in multiple ways: sorted set intersections, hash tables, binary searches, and gather-apply-scatter.

*b) Algorithmic Optimization:* Numerous techniques have be developed to reduce the time complexity for triangle counting. For example, Green & Bader [8] present a combinatorial optimization that reduces the number of necessary intersections, offering a better complexity bound. Shun & Tangwongsan [22], Polak [19], and Pearce [18] show ways to reduce the computational requirements by finding triangles in the directed graph (rather than the undirected graph). Green *et al.* [12] presents a scalable technique for load-balancing triangle counting on shared-memory systems.

*c) GPU Algorithms:* Leist *et al.* [14] showed the first GPU algorithm for triangle counting which used each thread to execute a different intersection. This proves to be inefficient due to a mix of load-balancing related issues. Green *et al.* [13] shows how to parallelize intersections by splitting them into smaller sub-intersections. This work leaves the number of threads for each intersection as a parameter to be configured. This leads to ineffective utilization of the GPU in terms of load-balancing as well as introducing overheads (that in some cases dominate the execution). Our work uses and extends the Intersect-Path algorithm such that a different number of threads is used for each intersection based on the workload characteristics of that intersection. By doing so, our algorithm is in some cases over two orders of magnitude (100×) faster than [13]. Wang *et al.* [26] show several different strategies for implementing triangle counting on the GPU, including matrix multiplication.

*d) Graph Challenge:* Wolf et. al. [28] uses an optimized sparse matrix multiplication formulation, and was a HPEC 2017 Graph Challenge champion. It was implemented on several many-core systems. Bisson et. al. [4], [3], also an HPEC Graph Challenge Champion submission, demonstrated efficient triangle counting algorithms using hash maps on the GPU. A more recent paper by Green *et al.* [11] uses the branch avoiding model [9] and shows how to vectorize triangle counting using Intel's AVX-512 instruction set. This paper also introduces a new method for load-balancing list intersections based on work estimations.

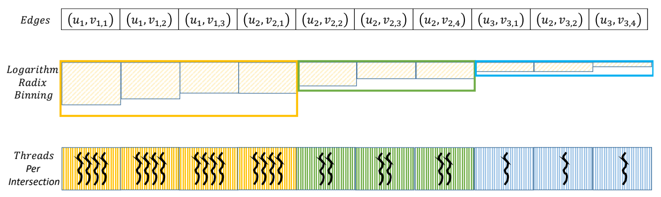

Fig. 1. Example of allocating threads per intersection according to work-estimate bins obtained through Logarithmic Radix Binning. The second row depicts edge list after binning. The third row shows threads per intersection in powers of 2, matching the relative size of each intersection and its bin.

## III. Fine-Grain Load-Balancing on the GPU

In this section we show how we extend the **Logarithmic Radix Binning** (LRB) load-balancing scheme, detailed in Green *et. al.* [11] to the GPU. Our load balancing scheme proceeds in two phases. In the first phase, the edges of the graph are reordered according to an estimated work of their adjacency intersections using LRB. In the second phase, the number of threads assigned to each intersection is determined according to the new edge ordering. We call this second phase **Logarithmic Threads Per Intersection** (Log-TPI).

### A. *Work Estimation of List Intersections*

In the first step we perform the LRB load-balancing scheme. LRB starts off by reordering a list of edges based on the integer ceiling of the *logarithm* of an edge's estimated work. For every edge $(u, v) \in E$, the expected work for the intersection is given by either of

$$IntersectionWork(u, v) = d_u + d_v \tag{1}$$

$$BinaryWork(u, v) = d_u \cdot log(d_v). \tag{2}$$

depending on the intersection method chosen. Green et. al. [11] presents mixing both intersection methods, which is outside the scope of this paper. These edges are then grouped into **bins** based on log-stepped work estimates. The bins are *log-stepped* in that edges are grouped by the nearest power of 2 ceiling of their work estimate.

### B. *Thread Granularity Workload Partitioning*

To ensure good load-balancing, our goal is to ensure that the cores (SMs) and the threads (SPs) get a near equal amount of work when processing intersections.

*a) Logarithmic Threads Per Intersection:* Let the estimated work for some intersection be given by $W$. Intersections of the same bin are processed by $T$ threads each, i.e. each intersection results in $T$ *sub-intersections*. $T$ is a bin-dependent parameter, and chosen such that work is balanced across threads. Suppose $W_b$ is the work ceiling of intersections in bin $b$.

Then we choose $T$ such that

$$\frac{W_b}{T} = C, \tag{3}$$

where $C$ is the desired size of a sub-intersection. $C$ is a constant defined ahead of time, and is explained later.

Since threads in the same bin have similar work estimates, this implies that any thread processing an intersection in bin $b$ is assigned work $\approx C$.

In this way, the number of threads assigned per intersection is scaled with the the size of the intersection. Intersections with larger estimated work will get a larger number of threads than intersections with a smaller amount of work. These concepts are illustrated in Fig. 1 where the intersections for 3 different bins are given a different number of threads—from one thread to four threads per intersection.

*b) Log-TPI Implementation Details:* Threads have been referred to conceptually so far, but in reality there are CUDA thread limits and hardware constraints. In practice we never need to exceed the device thread limit, since the largest threads-per-intersection is proportional to the maximum degree of an input. If the number of edges times threads-per-intersection is too large, then we simply iterate over multiple grids of edges. Furthermore, in the case of Sorted Set Intersection, we limit threads-per-intersection at block size (256), beyond which there are diminishing returns.

A CUDA kernel is launched for each bin, with threads per intersection specified by the Log-TPI scheme. Assuming the largest logarithmic bin value is $B = 32$, then there are up to 32 kernel launches. A kernel does not launch if there are no edges for that bin.

Finally, recall that $C$ is the desired sub-intersection size. We found that having $C_{intersect} = 16$ for Sorted Set Intersection and $C_{search} = 8$ for Binary Search works well in practice. Note that these sub-intersection sizes roughly match the size of a cache line, suggesting that some amount of minimum work is needed to amortize memory access cost. In the case of Sorted Set Intersection, Eq. 1 is used for the intersection work estimation. Moreover, the Intersection-Path [13] algorithm has a partitioning phase that introduces non-trivial overheads; using too small of a $C_{intersect}$ can reduce performance as more time may be spent on partitioning an intersection than on actually finding common neighbors. In the case of Binary Search, we found it more effective to use only the $d_u$ term of $BinaryWork(u, v) = d_u \cdot \log(d_v)$. $\log(d_v)$ represents a worst-case upper bound and we believe most searches are less than the upper bound in practice.

### C. Parallel Processing of an Intersection

Here we describe how a single intersection gets divided into sub-intersections and processed in parallel.

Alg. 1 shows the algorithm Intersect-Path for processing a single intersection using $T$ threads. The key step is a parallel partitioning procedure that breaks a pair of sorted lists into pairs of sublists, such that each sublist pair can be independently intersected in a merge-like fashion. We omit the details of the partitioning procedure and refer the reader to [17] and [13].

Alg. 2 shows the algorithm for Par-Search on a single intersection. The partitioning here is rather straightforward—we divide up the *smaller* of the two adjacencies evenly. Each thread has its sublist, and searches into the larger list to look for matches.

**Algorithm 1:** Sorted Set Intersection based list intersections. Given two sorted adjacency arrays, assign sub-intersections to each thread. Compare elements of resulting sub-arrays in merge-like fashion and check for equal values.

```
for $threadIdx \in threads$ do in parallel
    index = $\lfloor \frac{threadIdx}{T} \rfloor$ // $T$ is threads per intersection
    $u, v \leftarrow E[index]$
    $\mathbf{u}_{adj}, \mathbf{v}_{adj} \leftarrow adj(u, G), adj(v, G)$
    $ui, uj, vi, vj \leftarrow findSplit(\mathbf{u}_{adj}, \mathbf{v}_{adj}, threadIdx)$
    $\mathbf{u}_{sub}, \mathbf{v}_{sub} \leftarrow \mathbf{u}_{adj}[ui : uj], \mathbf{v}_{adj}[vi : vj]$
    $k \leftarrow 0, l \leftarrow 0, equalCount \leftarrow 0$
    while True do
        if $k \geq size(\mathbf{u}_{sub})$ or $l \geq size(\mathbf{v}_{sub})$ then
            return $equalCount$
        if $\mathbf{u}_{sub}[k] < \mathbf{v}_{sub}[l]$ then
            $k \leftarrow k + 1$
        else if $\mathbf{u}_{sub}[k] > \mathbf{v}_{sub}[l]$ then
            $l \leftarrow l + 1$
        else
            $k \leftarrow k + 1, l \leftarrow l + 1$
            $equalCount \leftarrow equalCount + 1$
```

**Algorithm 2:** Binary Search based intersections. Given two sorted adjacency arrays, $adj(v)$ and $adj(u)$, each thread is assigned a portion of the searches from the smaller list into the larger list.

```
for $threadIdx \in threads$ do in parallel
    index = $\lfloor \frac{threadIdx}{T} \rfloor$ // $T$ is threads per intersection
    $u, v \leftarrow E[index]$
    $\mathbf{u}_{adj}, \mathbf{v}_{adj} \leftarrow adj(u, G), adj(v, G)$
    $ui, uj \leftarrow findSplit(\mathbf{u}_{adj}.threadIdx)$
    $\mathbf{u}_{sub}, \mathbf{v}_{adj} \leftarrow \mathbf{u}_{adj}[ui : uj], \mathbf{v}_{adj}$
    $equalCount \leftarrow 0$
    for $key \in \mathbf{u}_{sub}$ do
        $low \leftarrow 0, high \leftarrow len(\mathbf{v}_{adj}) - 1$
        while $high \geq low$ do
            $middle \leftarrow (low + high)/2;$
            if $\mathbf{v}_{adj}[middle] = key$ then
                $equalCount \leftarrow equalCount + 1$
                break
            else if $\mathbf{v}_{adj}[middle] < key$ then
                $low \leftarrow middle + 1$
            else
                $high \leftarrow middle - 1$
```

### D. Optimizations

*a) Virtual Bins:* Instead of maintaining separate arrays for each bin, all bins are processed in the context of a single array of edges in memory. A CUDA kernel is launched for each bin, with threads per intersection calibrated accordingly.

*b) Directed Graph:* In order to count the number of unique triangles in a graph and cut out extraneous work, we convert the undirected graph into a directed graph as a preprocessing step. Numerous approaches and heuristics exist in the literature (see Section II). We use the degree-ordered-directed (referred to as DOD) approach from Pearce

[18].

### E. Overheads Complexity Analysis

*a) Logarithmic Radix Binning:* The total time complexity is $O(|E|+B^2)$, where $B$ is a value chosen to contain the log of the largest degree in the graph. Typically $B \in 32, 64$ and $B^2$ is much smaller than number of edges in the graph, see [11] for details.

*b) Parallelizing Intersections:* Parallelizing Sorted Set Intersection across intersections, namely the $findSplit$ operation in Alg. 1, incurs an $O(\log(\max_v |adj(v)|))$ overhead, corresponding to the log of the size of the largest adjacency (see [17] for details). This overhead is the worst case overhead and in practice is smaller when using vertices with smaller adjacency arrays. Parallelizing Binary Search incurs constant time overhead, as its $findSplit$ operation requires a simple division over the smaller adjacency.

*c) Storage Complexity Analysis:* We use CSR (compressed sparse row) to represent the original graph. For triangle counting, CSR requires two arrays, one for the offsets of size $O(|V|)$ and one for the indices (edges) which is of size $O(|E|)$. The binning technique used by LRB stores the edges in a different order, which requires an additional array of size $O(|E|)$.

## IV. Performance Analysis

### A. Experimental Setup

*a) System:* The experiments presented in this paper were carried out on a NVIDIA V100 GPU. The V100 is a Volta based GPU with 80 SMs and 16GB HBM2 memory. We independently execute two different triangle counting methods: 1) Intersect-Path and 2) Par-Search. Where applicable, these are denoted as “IP” and “BSearch” in the plots. Our CUDA code is compiled with the NVIDIA Compiler (`nvcc`) using CUDA version 9.1. The host compiler is `gcc`, version 6.4.0.

*b) Software and Libraries:* The algorithms presented were implemented using Hornet [5] as the underlying graph data structure. Hornet was configured to maintain sorted adjacency lists and this preprocessing time is not included in our results.

*c) Inputs:* The algorithms are tested using real world graphs and networks taken from the HPEC Graph Challenge [23] (see Table I). By default, all graphs are treated as undirected, and duplicate edges removed. As mentioned earlier, we utilize the optimization of finding triangles in the directed graph (where only half the edges exist) [18].

### B. Performance of Adaptive Approach

In this section we detail experiments and results that highlight different aspects and contributions of our implementations. In the interest of space, the figures shown include only a subset of the graphs we used from the Graph Challenge. Table II reports a full set of our results, including raw times and triangle counts.

### C. Comparison with State of the Art

For consistency, we refer to our Par-Search version of triangle counting (unless noted otherwise), as it was faster than our Intersect-Path implementation in most cases. We compare the performance of our new algorithm with several of the fastest algorithms and implementations from last year’s HPEC Graph Challenge. Two of these, KokkosKernels [28] and Bisson et. al. [4], were designated as competition champions and we compare against times reported in those papers. We also compare with [11], a 2018 Graph Challenge paper. This approach combines Logarithmic Radix Binning load-balancing with vectorization on the Intel Knight’s Landing. Finally, we include [22] as reference to a high-performing implementation prior to the Graph Challenge. For all but a small number of cases, our algorithm is faster than best results from last year’s Graph Challenge. On the larger graphs, our speedup is in the 5 – 10× range, and up to 20 – 30×. Our performance is comparable with that of [11], and is faster in some instances while slightly slower in others. Fig. 2 depicts the speedup of our algorithm over these aforementioned algorithms and shows the systems they ran on.

*a) Edges Per Second:* Fig. 3 depicts the number of edges processed per second for Sorted Set Intersection and Binary Search methods with Logarithmic Radix Binning. This metric serves to normalize our performance based on graph size. Edges per second is measured as the number of edges in the input graph divided by the total execution time, load-balancing overhead included. For most graphs we are able to process over 100 million edges per second, and in some cases as high as billions of edges per second. The lowest rates come from graphs smaller than 1 million edges.

*b) Adaptive Threads Per Intersection:* Fig. 4 compares the performance of our adaptive Logarithmic Threads Per Intersection approach, which allocates threads based on the expected amount of work, over static threads-per-intersection configurations which uses a constant number of threads (1, 32, and 256) for Par-Search. These correspond to processing single intersections with a thread, warp, or thread block.

For all the configurations, we use the LRB reordered lists for consistency. We do not consider these times in our analysis, as our focus is on the *processing* time of load-balanced list-intersections. In all instances except one (*email-EuAll*), Log-TPI performs at least as well (and often much better) than any of the static configurations. In some cases the adaptive Logarithmic Threads Per Intersection is over 20× faster. Furthermore, the best-performing static thread count varies by graph and is not known ahead of time. This underscores the benefits of an adaptive approach that is able to optimize load-balancing at the intersection level.

*c) LRB overhead:* Fig. 5 shows ratio of Logarithmic Radix Binning-related overhead vs. triangle counting computation. LRB overhead is relatively high for many of the smaller graphs. Overhead is also relatively high for graphs with very low average degree. This is not surprising, since tiny graphs and/or low-degree graphs already have inherent load-balance. Such graphs could also be much more

TABLE I

NETWORKS USED IN OUR EXPERIMENTS. $|E|$ REFERS TO NUMBER OF DIRECTED EDGES FROM SOURCE, PRIOR TO ANY PREPROCESSING.

| Name | $\vert V\vert$ | $\vert E\vert$ | Name | $\vert V\vert$ | $\vert E\vert$ | Name | $\vert V\vert$ | $\vert E\vert$ | Name | $\vert V\vert$ | $\vert E\vert$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| amazon0312 | 400,727 | 3,200,440 | soc-Slashdot0811 | 77360 | 469180 | g500-s24-ef16 | 8860450 | 260261843 | Theory-5-9-16-25-B1k | 26520 | 351745 |
| amazon0505 | 410,236 | 3,356,440 | soc-Slashdot0902 | 82168 | 504230 | g500-s25-ef16 | 17043780 | 523467448 | Theory-9-16-25-81-B1k | 362440 | 5212249 |
| amazon0601 | 403,394 | 3,387,388 | cit-Patents | 3774768 | 16518947 | roadNet-CA | 1965206 | 2766607 | Theory-25-81-256-B1k | 547924 | 4264567 |
| ca-HepPh | 12008 | 118489 | email-EuAll | 265214 | 364481 | roadNet-PA | 1088092 | 1541898 | Theory-3-4-5-9-16-25-B2k | 530400 | 22160059 |
| ca-HepTh | 9877 | 25973 | g500-s21-ef16 | 1243072 | 31731650 | roadNet-TX | 1379917 | 1921660 | Theory-5-9-16-25-81-B1k | 2174640 | 57334759 |
| cit-HepPh | 34546 | 420877 | g500-s22-ef16 | 2393285 | 64097004 | soc-Epinions1 | 75879 | 405740 | | | |
| cit-HepTh | 27770 | 352285 | g500-s23-ef16 | 4606314 | 129250705 | soc-LiveJournal1 | 4847571 | 68993773 | | | |

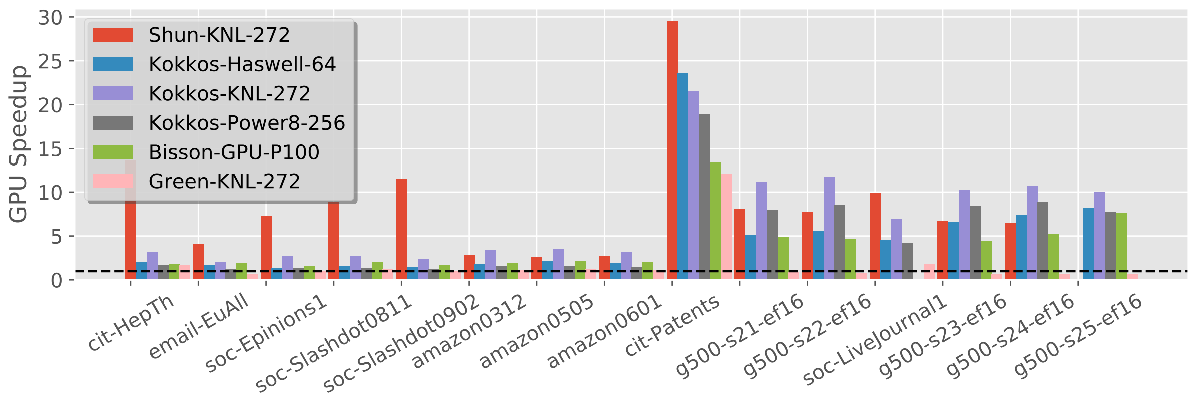

Fig. 2. Speedup of our load-balanced Par-Search algorithm over past implementations. Graphs are sorted in increasing order of edges. There are several missing bars for graphs where time was not reported for an implementation. Black line indicates speedup of 1.

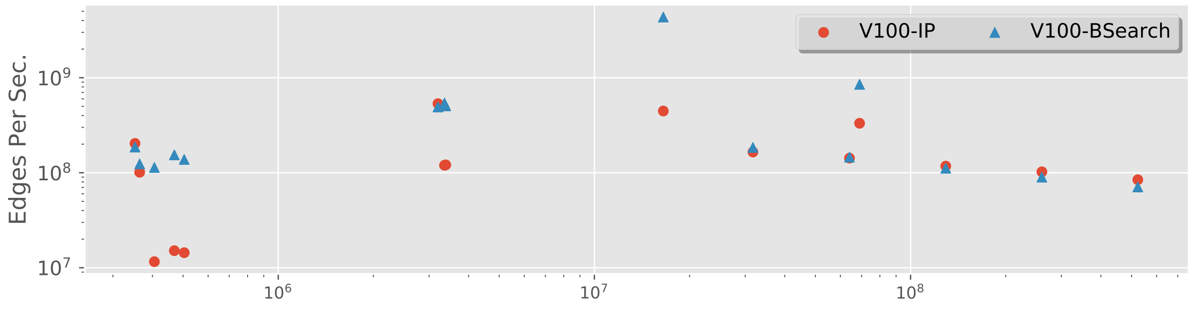

Fig. 3. Number of edges processed per second vs. graph size in number of edges, for load-balanced Intersect-Path and Par-Search algorithms.

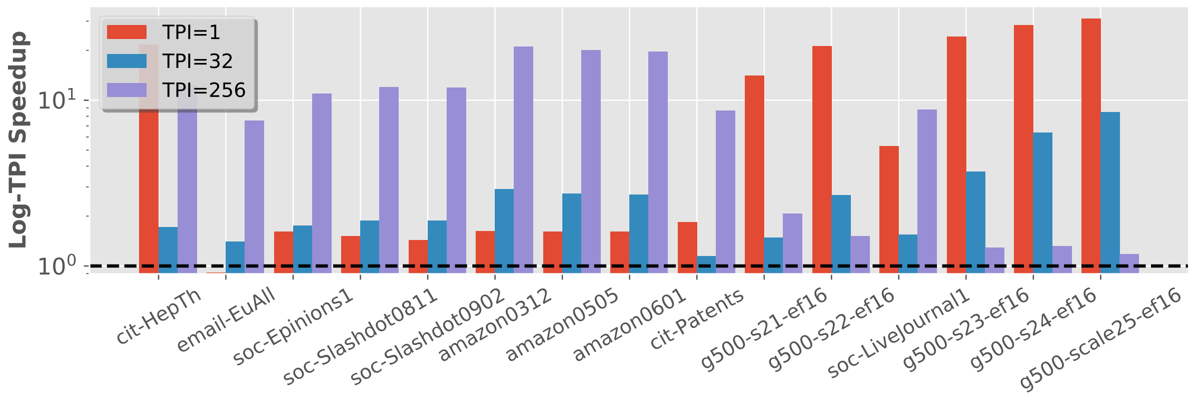

Fig. 4. Speedup (log-scale) of Log-TPI approach over three static threads-per-intersection configurations. Comparison is for the Binary Search-based approach. Speedups are based on *processing* times of intersections, and does not include LRB overhead. Graphs are sorted in increasing order of edges.

memory-bound than compute-bound. However, the overhead gap closes for larger graphs with higher average degree. On graphs with many millions of edges, the binning overhead is negligible and the processing of list intersections dominates.

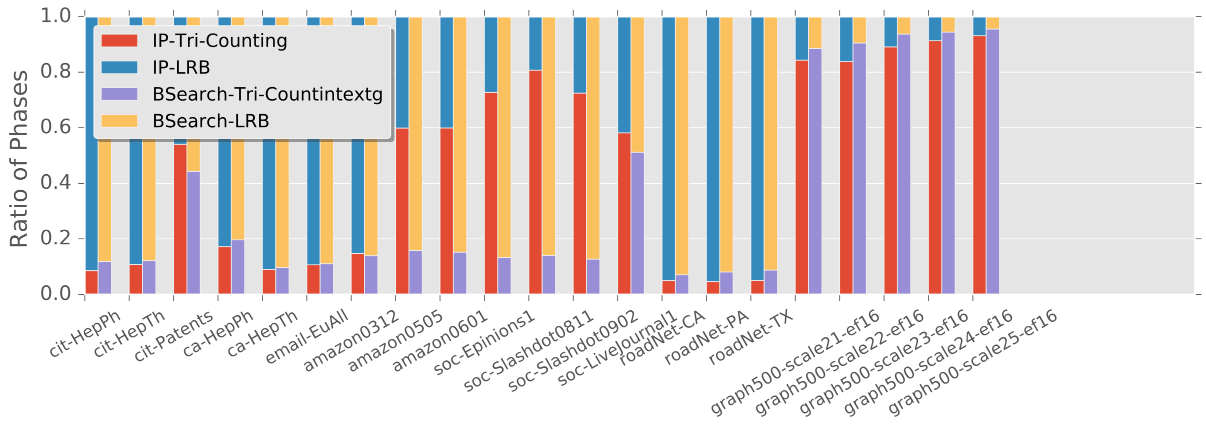

Fig. 5. Utilization ratio of LRB load-balancing overhead to computation time for triangle counting kernels. Breakdown is shown for both Sorted Set Intersection and Binary Search based algorithms.

TABLE II

GRAPHS USED IN OUR EXPERIMENTS, WITH REPORTED TIMES (IN SECONDS) AND TRIANGLE COUNTS FOR BOTH SORTED SET INTERSECTION AND SEARCH-BASED IMPLEMENTATIONS. PROCESSING REFERS STRICTLY TO THE KERNELS LAUNCHED FOR TRIANGLE COUNTING, WHILE TOTAL INCLUDES ALL OVERHEAD ASSOCIATED WITH OUR APPROACH. GRAPH LOADING, HOST-TO-DEVICE TRANSFER, AND GRAPH PREPROCESSING TIMES ARE NOT INCLUDED.

| Name | Intersect-Path Processing | Intersect-Path Total | Par-Search Processing | Par-Search Total | Triangles |
|---|---|---|---|---|---|
| amazon0312 | 0.0009 | 0.0065 | 0.0009 | 0.0065 | 3,686,467 |
| amazon0505 | 0.0168 | 0.0280 | 0.0010 | 0.0062 | 3,951,063 |
| amazon0601 | 0.0168 | 0.0280 | 0.0010 | 0.0067 | 3,986,507 |
| cit-HepPh | 0.0004 | 0.0047 | 0.0003 | 0.0028 | 1,276,868 |
| cit-HepTh | 0.0004 | 0.0033 | 0.0003 | 0.0028 | 1,478,735 |
| ca-HepPh | 0.0004 | 0.0022 | 0.0004 | 0.0023 | 3,358,499 |
| ca-HepTh | 0.0002 | 0.0017 | 0.0002 | 0.0019 | 28,339 |
| cit-Patents | 0.0200 | 0.0370 | 0.0167 | 0.0377 | 7,515,023 |
| email-EuAll | 0.0003 | 0.0031 | 0.0003 | 0.0029 | 267,313 |
| soc-Epinions1 | 0.0253 | 0.0348 | 0.0005 | 0.0036 | 162,4481 |
| soc-LiveJournal1 | 0.1214 | 0.2087 | 0.0414 | 0.0810 | 285,730,264 |
| soc-Slashdot0811 | 0.0253 | 0.0313 | 0.0004 | 0.0030 | 551,724 |
| soc-Slashdot0902 | 0.0253 | 0.0349 | 0.0005 | 0.0036 | 602,592 |
| roadNet-CA | 0.35225 | 7.00006 | 0.0005 | 0.0069 | 120,676 |
| roadNet-PA | 0.24985 | 5.38931 | 0.0003 | 0.0039 | 67,150 |
| roadNet-TX | 0.28262 | 5.58182 | 0.0004 | 0.0043 | 82,869 |
| g500-s21-ef16 | 0.1621 | 0.1922 | 0.1528 | 0.1726 | 935,100,883 |
| g500-s22-ef16 | 0.3769 | 0.4498 | 0.3978 | 0.4395 | 2,067,392,370 |
| g500-s23-ef16 | 0.9986 | 1.121 | 1.085 | 1.158 | 4,549,133,002 |
| g500-s24-ef16 | 2.323 | 2.543 | 2.734 | 2.896 | 9,936,161,560 |
| g500-s25-ef16 | 5.775 | 6.200 | 7.050 | 7.382 | 21,575,375,802 |
| Theory-5-9-16 -25-B1k | 0.0008 | 0.0039 | 0.0005 | 0.0036 | 264,799 |
| Theory-9-16-25 -81-B1k | 0.0094 | 0.0152 | 0.0067 | 0.0125 | 4,059,175 |
| Theory-25-81 -256-B1k | 0.0136 | 0.0189 | 0.0072 | 0.0135 | 2,102,761 |
| Theory-3-4-5-9 -16-25-B2k | 0.0258 | 0.0439 | 0.0132 | 0.0296 | 350 |
| Theory-5-9-16 -25-81-B1k | 0.1900 | 0.2174 | 0.1471 | 0.1743 | 66,758,995 |

These results show that Logarithmic Radix Binning is a small penalty to pay for more efficient triangle counting over larger graphs, but there is room for improvement on smaller graphs.

## V. CONCLUSIONS

Triangle counting is known to present load-balancing issues for massively multi-threaded systems. These challenges are amplified on modern GPU systems, which requires tens of thousands of threads for full system utilization. In this paper we introduce an adaptive load-balancing approach ensuring that runtime threads are effectively utilized for list intersections. We combine Logarithmic Radix Binning, a load-balancing approach that bins edges based on their intersection work estimation, with Logarithmic Threads Per Intersection for dynamically assigning work to threads on the GPU. Log-TPI scales the allocation of threads to intersections based on work-estimations, and enables faster processing of list intersections than in any static configuration. We successfully applied our adaptive approach on two different list intersection schemes, to produce two high-performing parallel triangle solutions based on Sorted Set Intersection and Binary Search, respectively.

In the context of the GraphChallenge, our new algorithms outperformed several state-of-the-art high performing and optimized triangle counting algorithms, including HPEC Graph Challenge champions. We believe this approach is extensible as a new kernel for list intersection applications at large. In particular, for the $k$-truss problem, we believe these new results would greatly improve the performance of the authors' previous work [10].

## ACKNOWLEDGMENTS

Funding was provided in part by the Defense Advanced Research Projects Agency (DARPA) under Contract Number FA8750-17-C-0086. This work was partially funded by the Doctoral Studies Program at Sandia National Laboratories for the U.S. Department of Energy's NNSA under contract DE-NA0003525. The content of the information in this document does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. The authors also acknowledge the FutureSystems Center [1] at Indiana University for providing HPC resources that have contributed to the research results reported within this paper.

## REFERENCES

[1] "Futuresystems: Digital science center, school of informatics and computing, indiana university." https://portal.futuresystems.org/.

[2] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient Semi-streaming Algorithms for Local Triangle Counting in Massive Graphs," in *14th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, 2008, pp. 16–24.

[3] M. Bisson and M. Fatica, “High performance exact triangle counting on gpus,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3501–3510, 2017.

[4] M. Bisson and M. Fatica, “Static graph challenge on gpu,” in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–8.

[5] F. Busato, O. Green, N. Bombieri, and D. Bader, “Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs,” in *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2018.

[6] S. Chu and J. Cheng, “Triangle listing in massive networks and its applications,” in *Proceedings of the 17th ACM SIGKDD Int’l Conf. on Knowledge Discovery and Data Mining*, 2011, pp. 672–680.

[7] J. Cohen, “Trusses: Cohesive Subgraphs for Social Network Analysis,” *National Security Agency Technical Report*, p. 16, 2008.

[8] O. Green and D. Bader, “Faster Clustering Coefficients Using Vertex Covers,” in *5th ASE/IEEE International Conference on Social Computing*, ser. SocialCom, 2013.

[9] O. Green, M. Dukhan, and R. Vuduc, “Branch-Avoiding Graph Algorithms,” in *27th ACM on Symposium on Parallelism in Algorithms and Architectures*, 2015, pp. 212–223.

[10] O. Green, J. Fox, E. Kim, F. Busato, N. Bombieri, K. Lakhotia, S. Zhou, S. Singapura, H. Zeng, R. Kannan, V. Prasanna, and D. Bader, “Quickly Finding a Truss in a Haystack,” in *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2017.

[11] O. Green, J. Fox, A. Tripathy, K. Gabert, E. Kim, X. An, K. Aatish, and D. Bader, “Logarithmic Radix Binning and Vectorized Triangle Counting,” in *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2018.

[12] O. Green, L. Munguia, and D. Bader, “Load Balanced Clustering Coefficients,” in *ACM Workshop on Parallel Programming for Analytics Applications (PPAA)*, Feb. 2014.

[13] O. Green, P. Yalamanchili, and L. Munguía, “Fast Triangle Counting on the GPU,” in *IEEE Fourth Workshop on Irregular Applications: Architectures and Algorithms*, 2014, pp. 1–8.

[14] A. Leist, K. Hawick, D. Playne, and N. S. Albany, “GPGPU and Multi-Core Architectures for Computing Clustering Coefficients of Irregular Graphs,” in *Int’l Conf. on Scientific Computing (CSC’11)*, 2011.

[15] J. Leskovec, K. J. Lang, and M. Mahoney, “Empirical comparison of algorithms for network community detection,” in *Proceedings of the 19th Int’l Conf. on World Wide Web*. ACM, 2010, pp. 631–640.

[16] M. E. Newman, D. J. Watts, and S. H. Strogatz, “Random Graph Models of Social Networks,” *Proceedings of the National Academy of Sciences*, vol. 99, no. suppl 1, pp. 2566–2572, 2002.

[17] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk, “Merge path - parallel merging made simple,” in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, may 2012.

[18] R. Pearce, “Triangle counting for scale-free graphs at scale in distributed memory,” in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–4.

[19] A. Polak, “Counting triangles in large graphs on GPU,” *arXiv preprint arXiv:1503.00576*, 2015.

[20] A. Prat-Pérez, D. Dominguez-Sal, J. M. Brunat, and J.-L. Larriba-Pey, “Shaping Communities out of Triangles,” in *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, ser. CIKM ’12, 2012, pp. 1677–1681.

[21] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, “Static Graph Challenge: Subgraph Isomorphism,” in *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2017.

[22] J. Shun and K. Tangwongsan, “Multicore Triangle Computations Without Tuning,” in *IEEE Int’l Conf. on Data Engineering (ICDE)*, 2015.

[23] S. Siddharth, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, “Static graph challenge: Subgraph isomorphism,” in *IEEE Proc. High Performance Embedded Computing Workshop (HPEC)*, Waltham, MA, 2017.

[24] T. Schank and D. Wagner, “Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study,” in *Experimental & Efficient Algorithms*. Springer, 2005, pp. 606–609.

[25] J. Wang and J. Cheng, “Truss Decomposition in Massive Networks,” *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 812–823, 2012.

[26] L. Wang, Y. Wang, C. Yang, and J. D. Owens, “A comparative study on exact triangle counting algorithms on the gpu,” in *Proceedings of the ACM Workshop on High Performance Graph Processing*. ACM, 2016, pp. 1–8.

[27] D. J. Watts and S. H. Strogatz, “Collective Dynamics of ‘Small-World’ Networks,” *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[28] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, “Fast linear algebra-based triangle counting with kokkoskernels,” in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–7.

[29] J. Yang and J. Leskovec, “Defining and evaluating network communities based on ground-truth,” in *Data Mining (ICDM), 2012 IEEE 12th International Conference on*. IEEE, 2012, pp. 745–754.