Logarithmic Radix Binning and Vectorized Triangle Counting

Oded Green, James Fox, Alex Watkins, Alok Tripathy, Kasimir Gabert,
Euna Kim, Xiaojing An, Kumar Aatish, and David A. Bader

Computational Science and Engineering, Georgia Institute of Technology - USA

***Abstract*— Triangle counting is a building block for numerous graph applications and given the fact that graphs continue to grow in size, its scalability is important. As such, numerous algorithms have been designed for triangle counting - some of which are compute-bound rather than memory bound. Even for compute-bound algorithms, one of the key challenges is the limited control flow available on the processor. This is in-part due to the high dependency between the control flow, input data, and limited utilization of vector instructions. Not surprising, compilers are not always able to detect these data dependencies and vectorize the algorithms. Using the branch-avoiding model we show to remove control flow restrictions by replacing branches with an equivalent set of arithmetic operations. More so, we show how these can be vectorized using Intel's AVX-512 instruction set and that our new vectorized algorithms are $2\times - 5\times$ faster than scalar counterparts. We also present a new load balancing method, Logarithmic Radix Binning (LRB) that ensures that threads and the vector data lanes execute a near equal amount of work at any given time. Altogether, our algorithm outperforms several 2017 HPEC Graph Challenge Champions such as the KOKKOS framework and a GPU based algorithm by anywhere from $1.5\times$ and up to $14\times$.**

I. Introduction

Triangle counting and enumeration is a widely used kernel for numerous applications. These include clustering coefficients, community detection, email spam detection, Jaccard indices, and finding maximal k-trusses. The key building block of triangle counting is adjacency list intersection. Numerous algorithms have been developed for triangle counting and these encapsulate a wide range of programming models: vertex centric, edge centric, gather-apply-scatter (GAS), and linear algebra. Some approaches require the adjacency arrays to be sorted whereas other approaches do not. Most previous algorithms focus on scalar execution as vectorizing these algorithms is typically challenging. In this paper, we show several news algorithm for vectorizing the computational kernels of triangle counting.

We also propose a new load-balancing technique, which we call Logarithmic Radix Binning (LRB), that ensures that all the threads and vector units get a near equal amount of work. LRB improves on previous techniques which only focus on thread level load-balancing and shows how to vectorize at the vector lane granularity. Thus, vector unit utilization.

In this paper, we present several new algorithms for triangle counting. These algorithm uses techniques developed for the branch-avoiding model discussed in Green *et al.* [9], [7]. Specifically, Green *et al.* show that the cost of branch mis-prediction is high for data dependent applications (such as graph algorithms) and that these applications can typically be implemented in an alternative manner that avoids branches entirely. This eliminates mis-prediction and makes execution times more consistent, at the cost of additional operations.

Algorithmic Contributions

- We develop a two-tiered binning mechanism for triangle counting. In the first tier, for each edge we decide on an intersection method that will be applied to that edge. Our current implementation uses two different kernels (though it can be extended to more). Therefore, this tier consist of two bins. For the second tier, we present Logarithmic Radix Binning (LRB). For each intersection kernel, a 2D set of bins is maintained. Each of these bin stores edges with similar computational properties. Thus, our vectorized triangle counting algorithm can grab K edges, where K is the vector width, with similar computational requirements allowing for good load-balancing and utilization at the vector lane granularity. In practice, this offers good scalability.
- We show how to increase the number of control flows in software. For a multi-threaded processor with P physical hardware threads and vector instructions K elements wide, we show how to execute up to $P \cdot K$ concurrent software threads. Where each of these threads is executing a different intersections. Thus, our new algorithm increases the control flow on the processor by a factor of K. For the Intel Knights Landing processor (discussed in Sec. V) used in our experiments, $P = 272$ and $K = 16$, allows us to support up to 4352 concurrent software threads.
- We show two novel vectorized triangle counting algorithms: 1) sorted set intersection, and 2) binary search. The first of these approaches finds common neighbors by scanning across the two sorted adjacency arrays being intersected, similar to a merge. The second approach finds common neighbors by searching for elements of one array in the other. We also show a runtime mechanism for deciding which algorithm should be selected The LRB method is then used to ensure good execution.
- LRB is architecture agnostic and has been extended to the GPU [5]. On the GPU, different bins are executed using a different number of threads, thus ensuring good load-balancing and trading off various overheads.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

Performance Contributions

• We compare our new algorithm against several high performing triangle counting algorithms, including [23], [31], [2], [27]. Several of these were the fastest triangle counting algorithms in the 2017 HPEC Graph Challenge [24]. Our algorithm is faster than all algorithms for a but small number of instances across a wide range of test graphs. This includes outperforming KOKKOS [31] by an average of $4\times$ and as much as $10\times$ and [2] by an average of $2\times$ and as much as $6\times$.

• From a scalability perspective, we show that our algorithm scales to large thread counts. This highlights the fact LRB is frequently able to give each thread a near equal amount of work and ensure good workload balance.

• Our new vectorized algorithms offers $2\times$-$5\times$ performance speedup over their scalar counterparts (which also use the LRB load balancing mechanism).

II. Related Work

The applications in which triangle counting and triangle listing are used is broad. It became an important metric to data scientists with the introduction of clustering coefficients [30]. Other applications for triangle counting are: finding transitivity [18], spam detection in email networks [1], finding tightly knit communities [21], finding k-trusses [4], [28], [10], and evaluating the quality of different community detection algorithms [17], [32]. An extended discussion of triangle counting applications can also be found in [3]. Triangle counting was also a key kernel for the HPEC Graph Challenge [22].

a) Computational Approaches: Given a graph, $G = (V, E)$, where V are the vertices and E are the edges in the graph, the three simplest and mostly widely used approaches for counting triangles in a graph [26]; enumerating over all node triplets $O(|V|^3)$, using linear algebra operations $O(|V|^w)$ (where $w < 2.376$), and adjacency list intersection. Adjacency list intersection can be completed in multiple ways: sorted set intersections, hash tables, or binary searches to look up values. The time complexity of each of these approaches is data dependent, yet upper bounds can be given. Triangle can also be completed using the Gather-Apply-Scatter (GAS) programming models[6], [14]. In this work we focus on the adjacency intersection approaches.

b) Algorithmic Optimization: Numerous computational optimizations can be applied to triangle counting algorithms for *s*tatic graphs to help reduce the overall execution time. For example, Green & Bader [8] present a combinatorial optimization that reduces the number of necessary intersections–offering a better complexity bound. Green *et al.* [12] show a scalable technique for load-balancing the triangle counting on shared-memory systems. Shun & Tangwongsan [23], Polak [20], and Pearce [19] show how to reduce the computational requirements by finding triangles in a directed graph rather than the undirected graph. Leist *et al.* [15], Green *et al.* [13], Wang *et al.* [29], and Fox *et al.*[5] show several different strategies for implementing triangle counting on the GPU.

c) Vector Instruction Sets: Vector instructions have been an integral part of commodity processors in the last twenty years, though the history of Single Instruction Multiple Data (SIMD) programs goes back even further. In SIMD instruction, each datum is placed in a separate lane and each lane executes the same instruction. Intel's AVX-512 ISA can operate on vectors of 512 bits. The AVX-512 instruction set has numerous conditional instructions referred to as masks in the AVX-512 ISA. Our algorithm makes extensive use of these instructions.

III. Logarithm Radix Binning

In this section, we present Logarithm Radix Binning (LRB for short), a method that effectively load balances the intersections across the thread, for both intersection algorithms. This method is efficient and works well for both the scalar and vectorized algorithms. LRB works by placing edges into bins based on the logarithmic value of the estimated amount of work for that edge. For triangle counting, we initially group the edges into two unique bins, one bin for the sorted set intersection and one for the binary search. We will then apply LRB to each to the edges and distribute them over a 2D grid of bins.

a) Initial binning: The intersection of the adjacency arrays can be done in multiple ways. Two popular approaches are 1) sorted set intersections and 2) using binary search. In the sorted set intersection, common elements are found by moving across the two sort adjacency arrays using a merge-like access pattern. Sorted set intersection performs well when the two adjacency arrays are of near equal length– we call this a balanced intersection. However, when one adjacency array is extremely large and the other is small, or the intersection is imbalanced, binary search is more efficient. For binary search, each element in the smaller array is looked up in the larger of the two arrays.

To determine which bin to place an edge $(u, v) \in E$ into, we use the following estimations:

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

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

Intersecting (u, v) and (v, u) will find the same triangles, as such only one of these is necessary. For simplicity and performance reasons, we choose the edge (u, v) such that $d_u < d_v$. If $d_u = d_v$, we select the vertex with the smaller id. The intersection method selected is based on the minimum of Eq. (1) and Eq. (2).

b) Finer grain binning: Fig. 1 depicts an edge list with the estimated amount of work for that edge. For each edge, the method with the minimal amount of estimated work is selected using Eq. 1 and Eq. 2. The yellow boxes denote edges that use the sorted set intersection approach and the blue boxes represent edges that use the binary search. The second row represents an edge ordering where the edges are placed into two bins—one for each approach. For the vectorized algorithm, this can lead to significant workload

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

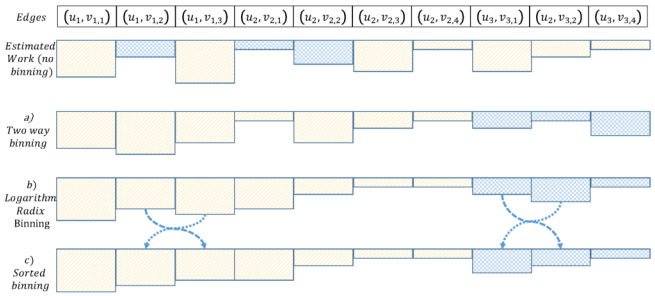


Fig. 1. Example of binning methodologies for a given input graph. The top array depicts an edge list (no binning is applied). The yellow boxes depict edges using the sorted set intersection and the blue boxes are edges using the binary search based approach: a) edges are categorized based on the intersection method, b) edges ordered using LRB, and c) edges are ordered from the largest to smallest (assuming a sorting algorithm is used). In practice LRB offers good load-balancing (see Sec. V for empirical performance). Lastly, the sorted ordering and LRB ordering are different in a number of places and marked by arrows showing the order changes.

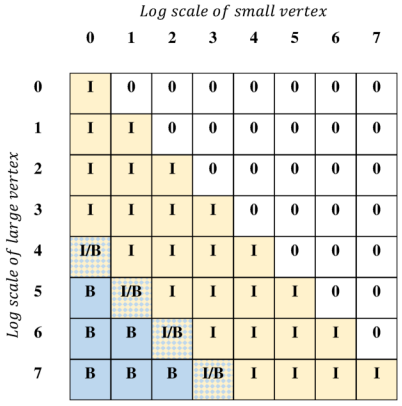


Fig. 2. The estimated binning matrix for sorted set intersection and binary search. Note this figure is conceptual, and exact bin assignments should not be inferred from it. A separate binning matrix is used for each approach.

imbalance within the vector. For both the vectorized and scalar algorithms, this ordering can also lead to thread under-utilization. To ensure a good workload balance, a prefix sum operation can be computed and then the edge list split across the threads such that each thread gets a near equal amount of work—this can be done yet adds significant overhead. Unfortunately, the prefix summation does not resolve the the vector level workload imbalance. The third row depicts the ordering created using LRB. The last row in this figure depicts the edge ordering assuming that the edges have been sorted based on the amount of work. This approach enables processing the heavy edges first. LRB will give similar load-balancing as sorting without the overhead.

c) Logarithm Radix Binning: Given a vertex $v \in V$, that vertex requires $\log(|V|)$ bits to represent its value. In practice, these values are represented by integers of 32 or 64 bits. We denote this width with B. Initially, we create two $B \times B$ matrices: M_s for the sort set intersection and M_b for the binary search approach. The entries of these matrices will be used to count the number of edges of a specific size and their estimated amount of work. Specifically, given an edge (u, v) such that $d_u < d_v$ that edge will be placed in

Algorithm 1: Binning strategy. This binning strategy can be extended to 64 bit integers by replacing 32 with 64.

```
for i = 0 : 1 : 31 do
    for j = 0 : 1 : 31 do
        InterBin[i][j] = 0;
        bBin[i][j] = 0;

forall e = (u, v) ∈ E s.t. d_u < d_v do
    logD_u ← 32 − leadingZeros(d_u);
    logD_v ← 32 − leadingZeros(d_v);
    IntersectWork ← d_u + d_v
    BinWork ← d_u · logD_u
    if BinWork < IntersectWork then
        bBin[logD_u][logD_v] ← bBin[logD_u][logD_v] + 1
    else
        iBin[logD_u][logD_v] ← iBin[logD_u][logD_v] + 1
```

the sub-bin with the following indices:

$$(row, col) = (\lceil log_2(d_v) \rceil, \lceil log_2(d_u) \rceil). \quad (3)$$

As can be seen, the row and the column are logarithms of the vertex degrees. Hence the name **"Logarithm Radix Binning"**. Note the ceiling operation in Eq. 3 ensures that the indices are integer values. In practice, we can find these values using less expensive approach:

$$(row, col) = (B - cntLZ(d_v), B - cntLZ(d_u)), \quad (4)$$

where $cntLZ$ is a function for counting the number of leading zeros. By subtracting the leading zeros from the word width, we can identify the ceiling of the logarithm value. Note the $cntLZ$ function is implemented efficiently on numerous architectures.

d) Prefix Summation For $B \times B$ Bins: After counting the number of instances of each edge type, two prefix sum arrays are computed - one for each LRB. Initially, the prefix sum array for M_s is computed and is followed by a prefix sum for M_b. The values in the prefix sum array for M_b will start where M_s finishes. These prefix sum arrays will point to the starting points (locations) of each edge size in the reordered edge list. These are the bins. The relatively small size of these arrays, B^2 elements (1024 elements for 32 bit vertices), means that this phase is computationally inexpensive as the sequential time for the prefix is $O(B^2)$. Now in a second sweep across the edges, each edge is placed in a bin. For a parallel execution, the values of the parallel prefix sum array can be incremented using atomic instructions - this does not ensure a deterministic ordering of the edges within the bin.

Fig. 2 depicts the combined LRB matrix for the two kernels. This matrix is for conceptual illustration only and to highlight that balanced intersections are placed close to the main diagonal and the imbalanced intersections are placed away from the main diagonal. Lastly, the two approaches use mostly different entries in the binning matrix, this is a direct result of how the work is estimated in Eq. 1 and Eq. 2.

e) Parallel Execution: Given the reordered edge list, a simple parallelization scheme is used to split the edges based on this edge's index modulo with respect to P. For example thread 0 will receive the edges $0, P, 2 \cdot P, ...$, thread 1 will receive the edges $1, P + 1, 2 \cdot P + 1, ...$ and so forth. LRB does not ensure the perfect ordering; however, on average

Algorithm 2: Branch-avoiding with conditional instructions. Variants of the branch-based and branch-avoiding algorithms can be found in [7].

$Tri - Counting - Branch - Avoiding - Conditionals()$
$a_i \leftarrow 0; b_i \leftarrow 0; count \leftarrow 0;$
while $(a_i < |A|$ *and* $b_i < |B|)$ **do**
  $CMP(A[a_i], B[b_i]));$
  $CADDEQ(count); //Conditional - ADD\ if\ (A[a_i] = B[b_i])$
  $CADDLEQ(a_i); //Conditional - ADD\ if\ (A[a_i] \leq B[b_i])$
  $CADDGEQ(b_i); //Conditional - ADD\ if\ (A[a_i] \geq B[b_i])$

Algorithm 3: Vectorized sorted intersection kernel

```
while (cond) {
    AVec = _mm512_i32gather_epi32(indexA, EArr, 4);
    BVec = _mm512_i32gather_epi32(indexB, EArr, 4);
    cmpLeVec = _mm512_mask_cmple_epi32_mask(cond,AVec, BVec);
    cmpGeVec = _mm512_mask_cmpge_epi32_mask(cond,AVec, BVec);
    cmpEqVec = _mm512_mask_cmpeq_epi32_mask(cond,AVec, BVec);
    tris     = _mm512_mask_add_epi32(tris   , cmpEqVec, tris, mione32);
    indexA   = _mm512_mask_add_epi32(indexA, cmpLeVec, indexA, mione32);
    indexB   = _mm512_mask_add_epi32(indexB, cmpGeVec, indexB, mione32);
    cond = _mm512_mask_cmpgt_epi32_mask(cond,indexAStop, indexA);
    cond = _mm512_mask_cmpgt_epi32_mask(cond,indexBStop, indexB);
}
```

the threads get an equal amount of work. This will become in Section V where the scaling of the algorithm is almost perfectly linear.

f) Time Complexity Analysis: Phase 1: finding the proper bin takes $O(|E|)$ steps for evaluating E edges. Phase 2: $O(B^2)$ to compute the prefix matrices. Phase 3) an additional $O(|E|)$ steps to reorder the edge list. Phase 1 and Phase 3 are embarrassingly parallel and easily split across the P threads. Phase 2 can also be done in parallel, however, the cost of the prefix operation on arrays of B^2 is relatively small in comparison to the other two phases and sequential implementation is enough. Recall that $B \in 32, 64$ and that $B^2 << |E|$. The total time complexity is $O(|E| + B^2) = O(|E|)$.

g) Storage Complexity Analysis: We use CSR (compressed sparse row) to represent the original graph and assume that the adjacency arrays are sorted. 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. We used an array of size $O(|E|)$ to store these edges. While this new edge list does not increase the theoretical upper-bound; from a practical perspective, it does double the memory consumption. The edges in the new reordered edge-lists determine the order in which the edges will be intersected. However, the intersection process itself uses the sorted adjacency arrays in the CSR graph.

IV. Vectorized Algorithms

In this section we present our new branch-avoiding and vectorized triangle counting algorithms. Alg. 2 depicts the sorted list intersection using the branch-avoiding programming model [9], [7]. Note, the control flow for the branch-avoiding algorithms is largely **independent** of input values–this is a key enabling factor for our vectorized approach. See Green *et al.* [9], [7] for additional discussion on the branch-avoiding programming model.

Algorithm 4: Vectorized binary search kernel

```
while (cond){
    sumVec = _mm512_add_epi32(low,high);
    middle =  _mm512_maskz_srl_epi32 (cond,sumVec,oneShifter);
    vals   = _mm512_mask_i32gather_epi32(vals, cond, middle, EArr, 4);
    cmpEqVec = _mm512_mask_cmpeq_epi32_mask(cond,vals,keys);
    cmpLtVec = _mm512_mask_cmplt_epi32_mask(cond,vals,keys);
    cmpGtVec = _mm512_mask_cmpgt_epi32_mask(cond,vals,keys);
    tris     = _mm512_mask_add_epi32(tris ,cmpEqVec,tris ,mione32);
    low      = _mm512_mask_add_epi32(low, cmpLtVec, middle, mione32);
    high     = _mm512_mask_add_epi32(high, cmpGtVec, middle, miMone32);.
    cond     = _mm512_mask_cmpge_epi32_mask(cond & ~cmpEqVec,high, low);
}
```

Alg. 2 depicts a branch-avoiding algorithm for list intersection—this algorithm uses conditional instructions. Such instructions do not always exist in all architectures and are in fact designed for a single control flow systems where a single instruction is executed based on hardware flags (zero-flag, carry-flag, and overflow-flag). As such, a naïve vector implementation might be constrained to a single set of these flags or would a require a single control flow. We show how to overcome this hardware constraint using the AVX-512 instruction set. Specifically we show 1) how to increase the number of software control flows and 2) how to control the execution of each lane using masks (even though we do not have enough hardware flags).

Our experience with conditional instructions is that the compiler is not able to figure out how to use them. We strongly differentiate between *compare* and *branch* instructions. Compare instructions are used for evaluating different values whereas a branch typically uses compare output to decide on the next sequence of executable instructions.

a) Vectorized Intersections: We started off by describing how to increase the control flow. The vectorized triangle counting can be implemented in a variety of ways using the branch avoiding model. In the first approach the different lanes work together on the same intersection (consisting of two arrays). This approach was shown to be effective on the GPU's SIMT programming model [11], [13]; however, this approach is significantly more challenging to implement for vector instructions. In the second approach, each lane in the vector unit is responsible for a different intersection. Thus, each vector unit requires $2 \cdot K$ different adjacency arrays for K different intersections.

We choose the second of the approaches–each lane is responsible for a different intersection. This also removes the overhead of the partitioning scheme found in[13]. Thus, the maximal number of concurrent intersections (software threads) that can be executed on a system is $Concurrent = P \cdot K$.

The branch-avoiding algorithm found in Alg. 2 depicts an initial "recipe" for implementing a vectorized triangle counting algorithm. Note that the number of data dependent branches has been significantly reduced, yet there still remains one condition in the control flow that is data dependent–the WHILE loop's condition which checks bounds of the two respective arrays. To vectorize the algorithm, this condition also needs to be vectorized and this is by no means trivial as this condition is responsible for the entire WHILE loop. The vectorized version of the algorithm

TABLE I

NETWORKS USED IN OUR EXPERIMENTS.

Name	$\|V\|$	$\|E\|$	Name	$\|V\|$	$\|E\|$
amazon0312	400,727	3,200,440	g500-s23-ef16	4606314	129250705
amazon0505	410,236	3,356,440	g500-s24-ef16	8860450	260261843
amazon0601	403,394	3,387,388	g500-s25-ef16	17043780	523467448
cit-HepTh	27770	352285	soc-Epinions1	75879	405740
cit-Patents	3774768	16518947	soc-LiveJournal1	4847571	68993773
email-EuAll	265214	364481	soc-Slashdot0811	77360	469180
g500-s21-ef16	1243072	31731650	soc-Slashdot0902	82168	504230
g500-s22-ef16	2393285	64097004			

ensures that certain data lanes will be ignored if the bounds of the indices for that lane are exceeded—each of the K lanes is responsible for managing its own bounds. Similar restrictions exist for the binary search based intersection (Alg. 4).

Alg. 3 and Alg. 4 depict the vectorized code for the sorted set intersection approach and for the binary search approach, respectively. These algorithms show close-to-real vector code instructions (using Intel's AVX-512 instructions set) rather than pseudo-code. This allows highlighting:

• The vectorized algorithms require gathering the elements for K intersections (using $2 \cdot K$ arrays) instead of just two arrays for a scalar execution. The introduction of efficient gather instructions has greatly simplified the process of collecting elements from random locations in memory.

• The AVX-512 instruction set introduces masked-vector instructions. These masks enable operating on a subset of the vectors lanes and updating the counters for each lane. Masked instructions are not conditional instructions. Specifically, the masked instructions are always executed across all the lanes; however, some data lanes might not be updated based on the value of the mask. Another key difference is that conditional instructions were designed for a single control flow (one per thread) whereas the masked operations allow a vector-wide control flow (for multiple control flows). This distinction is the reason that a single conditional instruction is replaced with two masked instructions. For example, the CADDEQ operations is replaced with vector CMPEQ instruction followed by a masked add vector operation. While this obviously incurs a performance penalty, it also enables increasing the scalability of the algorithm across the vector.

V. PERFORMANCE ANALYSIS

a) Experiment System: The experiments presented in this paper are primarily executed on an Intel Xeon Phi 7250 processor with 96GB of DRAM memory (102.4 GB/s peak bandwidth). This processor is part of the Xeon Knights Landing series of processors. In addition to the main memory, the processor has an additional 16GB of MCDRAM high bandwidth memory (400 GB/s peak bandwidth) which is used used as our primary memory - if the graph fits into main memory the lower latency DRAM memory is not utilized. The Intel Xeon Phi 7250 has 68 cores with 272 threads (4-way SMP). These cores run at a 1.3 GHz clock and share a 32MB L2 cache. Given these system parameters and using our new algorithms, we are able to execute up to 4352 concurrent intersections [1]. We also provide results for a dual Intel Xeon 8160 Skylake processor, with 48 cores (96 threads with hyper-threading), 32 MB LLC, and 192GB of DDR4-2400 memory. All code, on both systems, is compiled with the Intel Compiler (icc) (version 2017).

TABLE II

DIFFERENT PARALLEL VARIATIONS OF OUR TRIANGLE COUNTING ALGORITHMS. † DENOTES OUR FASTEST IMPLEMENTATION.

Algorithm name	Description
Mixed-EdgeList	Simple algorithm that selects intersection method based on edge properties.
lrb-scalar	Scalar implementation of our LRB load-balancing.
lrb-scalar-dod	Scalar implementation that includes the direction optimized graph.
lrb-vector	Vectorized (branch-avoiding) implementation of our LRB load-balancing.
lrb-vector-dod †	Vectorized (branch-avoiding) implementation including the direction optimized graph.

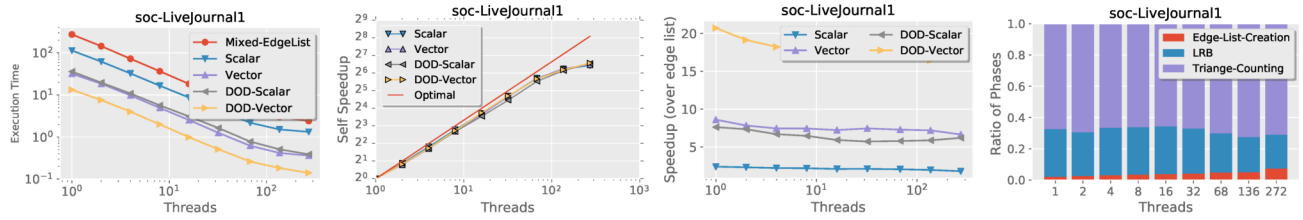
b) Inputs: The algorithms are tested using real world graphs and networks taken from SNAP[16] and the HPEC Graph Challenge [24], Table I. By default, all graphs are treated as undirected. Directed graphs are transposed and duplicate edges created in this phase are removed. Our algorithm can also utilize the optimization of finding triangles in a directed graph (where only half the edges exist). This concept is used in [23], [20], [19] and is referred to as the **DOD** graph in [19]—which is the terminology we use in this paper.

c) LRB Analysis: Our algorithm implementation incorporates multiple optimizations. To capture the benefits of each of these optimization, we execute our algorithm with several different optimizations. Table II describes the various optimizations we use.

Fig. 3 depicts various performance characteristics of our new algorithms and the various optimizations for the *soc-LiveJournal1* graph - similar results were seen for other graphs. Note the abscissa is log scale for all the sub-figures. Fig. 3 (a) depicts the execution time as a function of the number of threads and Fig. 3 (b) depicts the speedup for each of these configurations in comparison with a sequential execution of a specific algorithm. For all these configurations, the parallel scalability is near linear all the way up to 68 threads which is the number of physical cores on the KNL system used in our experiments. While there is some performance improvement beyond 68 threads, the scaling it is not linear. This is a well known artifact of multiple threads per core when resources are shared. Yet, it also shows that LRB is successful as a load-balancing mechanism.

Fig. 3 (c) highlights the contributions of the different optimizations of our algorithm. For each thread count, all algorithms are normalized against the "Mixed edge-list" implementation for a given thread count. The typical speedup of going from the scalar execution to the vectorized execution increases performance by roughly $2.5\times$ for both the regular graph as well the the DOD graph. For other graphs, the vectorization increased performance by as much as $5\times$. Applying all these optimizations together greatly improves performance over an already optimized algorithm (that selects an ideal intersection kernel for each edge). Specifically, for *soc-LiveJournal* this improves performance by an average

[1] We note that parallelism may be limited in practice by the number of vector units. To the best of our knowledge 4 threads (single core) share 2 VPUs [25].



(a) Execution time (b) Self Speedup (c) Speedup contributions (d) Phases breakdown

Fig. 3. Performance characteristics for the *soc-LiveJournal1* graph.

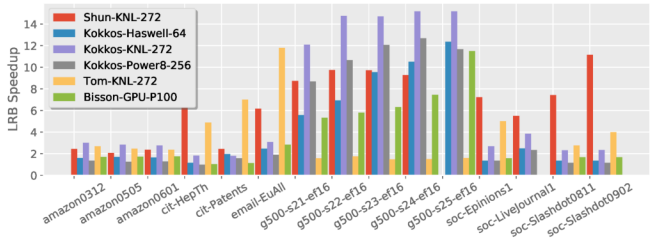


Fig. 4. Speedup of our new algorithm over past implementations. Missing bars reflect times not reported by past implementations.

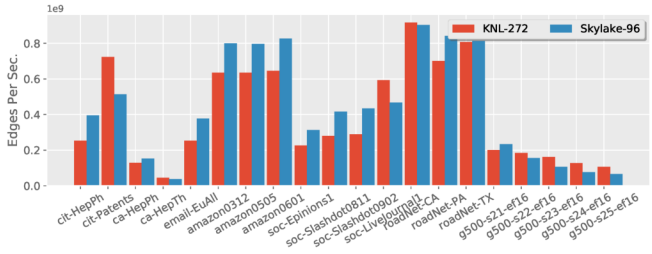


Fig. 5. Number of edges processed per second (in hundreds of millions).

of 20× for different thread counts. For other graphs, we saw speedups that were as high 62×.

Fig. 3 (d) depicts the ratio of time spent in the three phase of the algorithm: 1) edge list creation, 2) LRB, and 3) triangle counting. For most graphs and thread counts, a majority of time is spent in the triangle counting phase.

d) Comparison with other high performance implementation: We also compare the performance of our new algorithms with several of the fastest algorithms and implementations from last year's HPEC Graph Challenge. Two of these, KOKKOS [31] and [2], were designated as Champion of the competition and we compare against the times reported in those papers. We also compare against [23] and [27]. For these two algorithms, we used open-source software and report times on our KNL system. Of all these algorithms, only KOKKOS uses vector instructions explicitly - this is due to its SpMV nature and the fact that the other algorithms are unable to utilize the SIMD instructions directly. Fig. 4 depicts the speedup of our algorithm over these aforementioned algorithms. The ordinate represents the speedup of our algorithm over each framework. Note, that for all but a small number of cases our algorithm is faster. For handful of instances our algorithm is around 10% slower than the other frameworks. A more detailed comparison can be found in Table III.

TABLE III

PERFORMANCE COMPARISON OF OUR NEW ALGORITHM WITH SEVERAL RECENT OPTIMIZATION ALGORITHMS.

Algorithm name	Description
KOKKOS [31]	The experiments in [31] report the execution of the KOKKOS for three different processors: a IBM Power 8 system with 256 threads, an Intel Haswell system with 64 threads, an Intel KNL system with 272 threads. On average our algorithm is 2× faster than KOKKOS-Power8-256, 1.5× faster than KOKKOS-Haswell-64, 2.3× faster than KOKKOS-KNL-272. Our algorithm outperformed KOKKOS by as much as 4× on KNL.
Bisson-GPU-P100 [2]	Our new algorithm outperforms that of [2] which used a hashing based intersection on an NVIDIA P100 GPU. In many cases our algorithm was 2× faster than [2] and was as much as 4× for some inputs.
Shun-Hawell-272 [23]	Our new algorithm also outperforms the highly optimized algorithm by Shun and Tangwongsan [23] by an average factor of 5× and in numerous cases is over 8× faster.
Tom-KNL-272 [27]	Our new algorithms outperforms the algorithm of Tom *et al.* [27] by roughly 2×−4× in most cases. The times reported for Tom *et al.* [27] includes only their triangle counting phase and *does not* include their preprocessing phase which includes sorting the adjacency arrays and creating the edge ordering. In contrast, our execution times include our own preprocessing time (that does not use sorting). As such our algorithm could be potentially faster than Tom *et al.*.

e) Edges Per Second: Lastly, Fig. 5 depicts the number of edges processed per second for both the KNL and Skylake processors. Edges per second is measured as the number of edges in the input graph divided by the execution time. For most graphs we are able to process over 200 million edges per second and are able to process as much as 900 million edges per second.

VI. CONCLUSIONS

In this paper we showed a scalable vectorized algorithm for triangle counting that use the branch avoiding model. Specifically our two key contributions are as follows. The first of these is a novel non-SpMV programming model that enables using vector instructions for implementing graph algorithms. Using this vectorized branch-avoiding model we show how to execute 4352 concurrent threads (which we refer to as software control flows) on a system that only supports 272 hardware threads. We showed two different variants of adjacency array intersections using this model: sorted set intersection and binary search. Specifically, we utilize Intel's AVX-512 ISA which gives wide vectors (16 elements of 32 bits). Our second contribution is a novel load-balancing technique, Logarithm Radix Binning (LRB), that ensures that the threads and their respective vector units are utilized in an efficient manner. We believe that LRB might be applicable to additional graph analytics and plan on investigating this.

From a practical perspective, our new algorithm outperformed several high performing and optimized triangle counting algorithms - including several HPEC Graph Challenge champions. On average our algorithm out performed

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

KOKKOS, a SpMV based implementation that uses vector instructions, HPEC Graph Challenge Champion by an average of $2.5\times$. Our new algorithm is also upto $4\times$ faster than the fastest algorithm for the GPU (running an NVIDIA P100 GPU). There are numerous instances where our new algorithm is also $5 \times -10\times$ faster than these algorithm.

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. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration 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 acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

REFERENCES

[1] 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.

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

[3] 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.

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

[5] J. Fox, O. Green, K. Gabert, X. An, and D. Bader, "Fast and Adaptive List Intersections on the GPU," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2018.

[6] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," in *OSDI*, vol. 12, 2012.

[7] O. Green, "When Merging and Branch Predictors Collide," in *IEEE Fourth Workshop on Irregular Applications: Architectures and Algorithms*, 2014, pp. 33–40.

[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, R. McColl, and D. Bader., "GPU Merge Path: A GPU Merging Algorithm," in *26th ACM International Conference on Supercomputing*, 2012, pp. 331–340.

[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] F. Khorasani, K. Vora, R. Gupta, and L. Bhuyan, "CuSha: Vertex-Centric Graph Processing on GPUs," in *23rd ACM Int'l Symp. on High-Performance Parallel and Distributed Computing (HPDC)*, 2014, pp. 239–252.

[15] 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.

[16] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data.

[17] 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.

[18] 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.

[19] 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.

[20] A. Polak, "Counting triangles in large graphs on GPU," *arXiv preprint arXiv:1503.00576*, 2015.

[21] 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.

[22] 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.

[23] J. Shun and K. Tangwongsan, "Multicore Triangle Computations Without Tuning," in *IEEE Int'l Conf. on Data Engineering (ICDE)*, 2015.

[24] 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.

[25] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," *Ieee micro*, vol. 36, no. 2, pp. 34–46, 2016.

[26] 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.

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

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

[29] 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.

[30] D. J. Watts and S. H. Strogatz, "Collective Dynamics of 'Small-World' Networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[31] 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.

[32] 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.