Update on k-truss Decomposition on GPU

Mohammad Almasri*, Omer Anjum*, Carl Pearson*, Zaid Qureshi[§], Vikram S. Mailthody*, Rakesh Nagi[‡], Jinjun Xiong[†], and Wen-mei Hwu*

*ECE, [§]CS, [‡]ISE, University of Illinois at Urbana-Champaign, Urbana, IL 61801

[†]Cognitive Computing Systems Research, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 10598

[†]{almasri3, oanjum, pearson, zaidq2, vsm2, nagi}@illinois.edu, jinjun@us.ibm.com, w-hwu@illinois.edu

***Abstract*—In this paper, we present an update to our previous submission on k-truss decomposition from Graph Challenge 2018. For single GPU k-truss implementation, we propose multiple algorithmic optimizations that significantly improve performance by up to 35.2x (6.9x on average) compared to our previous GPU implementation. In addition, we present a scalable multi-GPU implementation in which each GPU handles a different *'k'* value. Compared to our prior multi-GPU implementation, the proposed approach is faster by up to 151.3x (78.8x on average). In case when the edges with only maximal k-truss are sought, incrementing the *'k'* value in each iteration is inefficient particularly for graphs with large maximum k-truss. Thus, we propose binary search for the *'k'* value to find the maximal k-truss. The binary search approach on a single GPU is up to 101.5 (24.3x on average) faster than our 2018 k-truss submission. Lastly, we show that the proposed binary search finds the maximum k-truss for "Twitter" graph dataset having 2.8 billion bidirectional edges in just 16 minutes on a single V100 GPU.**

***Index Terms*—GPU, CUDA, k-truss decomposition, binary search, multi-GPU, multi-node**

I. Introduction

Finding cohesive structures in a large graph is an important data mining technique for information retrieval in a variety of areas such as genomics, social media, cybersecurity, computer networks, public health and many more. Due to the advancements in cloud services that naturally curate and maintain big datasets, these datasets have been growing exponentially. Thus, there is an immediate need for developing innovative algorithms, software and hardware to efficiently process these big datasets in reasonable time.

k-truss decomposition is a graph analysis technique and the focus of this paper. k-truss is a cohesive subgraph in which each edge is part of at least $k-2$ triangles. This subgraph relaxes the concept of clique and can be computed in polynomial time. At the high level, state-of-the-art implementations of k-truss decomposition algorithm [1], [2], [3] for a given value of k are comprised of three major steps: 1) Determine the trussness value to each edge in the graph which fundamentally tells the number of triangles each edge is part of; 2) For a given value of k, remove edges with trussness value less than $k-2$, often called as *"peeling"*; 3) Remove the affected edges until none of the edge in the graph is left with trussness less than $k-2$. If the graph is not empty then the value of k is incremented by one, and the above three steps are repeated until we reach the maximum k-truss value where the graph is empty. However, the incremental approach in which k value is incremented by one after each iteration causes additional computations if only the edges with the maximum k-truss were to be determined. In this case a faster approach is required.

In this paper, we redesigned and improved our peeling-based [1] incremental k-truss decomposition algorithm by enhancing triangle counting, affected edge identification, and stream compaction steps. We also present our multi-GPU implementation that scales better than the 2018 submission. In our prior submission, edges were divided evenly across GPUs such that a single k value is calculated at the end of each iteration. This approach introduced significant slowdown as we scaled the number of GPUs due to a high number of page faults from actively reading and updating the shared lists at the short update and marking the affected edge steps. To address this limitation, we propose to evaluate a different k value in parallel across GPUs at each iteration. Although, the proposed approach introduces load imbalance, the scaling efficiency gain is significantly better than the prior approach primarily due to the reduced number of GPU page faults.

However, the incremental approach is inefficient when only the maximum k-truss is to be found as mentioned earlier. To address this inefficiency, we propose "Binary k-truss decomposition" which achieves a significant reduction in the required number of iterations with further improved performance.

We note that the previous year's champions [3] reported the time for $k = 3$ truss and did not proceed to find the largest truss. While the former depends on the efficiency of triangle counting, there is a fair amount of sophisticated work involved in treating deleted edges as we move to find higher values of k. Thus, a direct comparison between the proposed approach and the approach of the 2018 champions is not possible.

Compared to our baseline [1], we make the following contributions in the paper:

- An efficient single GPU implementation of k-truss decomposition outperforming the our prior work by up to 35.2x and 151.3x for single GPU and multi-GPU implementations, respectively.
- A scalable multi-GPU implementation of k-truss decomposition.
- A binary k-truss algorithm to search for the largest k-truss which speeds up the computation by 101.5x (24.3x on average) on a single GPU.

The rest of the paper is organized as follows. Section II covers the related work for k-truss decomposition and particularly summarizes the efforts in the graph challenge community. Section III goes over the proposed k-truss algorithm and related optimizations. Section IV provides the implementation details of single GPU and multi-GPU approaches. Results are covered in Section V followed by conclusion in Section VI.

II. Literature Review

In the last couple of years of the Graph Challenge, static k-truss decomposition challenge has seen a number of participants, particularly trying to boost the performance of the k-decomposition algorithm [2], [3], [4], [5], [1], [6], [7]. In [6] instead of calculating the exact triangle support for an edge, triangles are counted until the count is $\geq (k-2)$ which is enough to know if the edge needs to be removed. Further optimization is obtained by removing the nodes with the degree $< (k-1)$, an observation taken from [8]. k-truss decomposition presented in [2] is a very large scale CPU implementation of the graphs, using 128 to 256 compute nodes and uses degree-ordered directed graph. The triangle counting is done once in the beginning and then the count is decremented as the triangles unroll. In [9], k-truss decomposition uses *"peeling"* strategy and evaluates using GPUs. After the removal of an edge the support of affected edges is decremented. Input matrix compaction presented in [3] is an update on what was presented in [9]. The objective of matrix compaction is to reduce the variance in the adjacency list thereby assist in minimizing the load imbalance among thread blocks. In [7], a multi-stage peeling algorithm is proposed which relies on communication between threads using message queues. Doubly-linked lists per truss value are used to insert and delete edges in $\mathcal{O}(1)$ time. Neighbor list intersection is used for triangle counting. In [5], an approximate algorithm is run before the exact algorithm which returns the exact k-truss value. Hardware optimizations are applied for intersecting adjacency lists and triangle count is decremented as the triangles are torn down by the removal of edges. CPU-GPU collaboration strategy presented in [1] offloads some of the work load to CPU and uses delayed stream compaction to physically change the graph until there are no more affected edges. Our work is an update on the method described in [1].

III. k-truss Decomposition

In this section, we explain the algorithmic improvements we made over our prior approach for k-truss decomposition [1]. We will first discuss optimized incremental approach and then discuss the Binary k truss decomposition algorithm.

A. Incremental k-truss Decomposition

The pseudocode for the improved k-truss decomposition algorithm is shown in Algorithm 2. The first step is to initialize three lists named as 'Keep', 'Affected' and 'Reversed' whose sizes are equal to number of edges in the graph. 'Keep' holds a flag for each edge to indicate whether the edge is kept (not deleted). Each element of the 'Affected' list also corresponds to an edge and indicates whether the edge is affected by the deletion of any other edge with which it shares triangles. For any edge (u,v) in an undirected graph, we also need to know the index of edge (v,u) to ensure that both edges are marked as kept or affected. The index of edge (v,u) is kept in the list named as 'Reversed'.

The incremental approach starts with $k=3$, the minimum possible value of k-truss. The outer *while loop* at Line 3 of Algorithm 2 repeats until we reach the maximum k-truss where the graph becomes empty. The inner *while loop* at Line 5 is repeated until no affected edges are found. For any edge in the loop at Line 8 with $u < v$, if it is kept and affected then triangle counting is performed at Line 11. If triangle count is less than $k-2$, both edges (u,v) and (v,u) are deleted at Line 13. Other edges that share the triangle with the deleted edges will be marked as affected as shown in Line 14. All the deleted edges are then counted as in Line 17. After each iteration of the outer *while loop*, stream compaction is performed to physically remove the deleted edges, as shown in Line 29. Compared to our earlier submission we make following algorithmic optimizations:

- In our earlier implementation, reduction is performed over the 'Affected' list which is an unnecessary repetitive step taking place inside the inner *while loop*. We take a different approach and introduce affected hint flag per thread to indicate if there are any edges a thread has effected. The inner *while loop* uses this flag and continues until there are no affected edges to process, as shown in Line 21 of Algorithm 2.
- While doing triangle counting (Algorithm 3) we keep track of the indices of the first and last intersections of the two adjacency lists. These indices define the range where all the affected edges can be found to avoid traversing full length of adjacency lists during the affected edge determination phase in Line 14 of Algorithm 2.
- To further optimize the affected edge determination phase, we start marking affected edges early during the triangle counting phase as shown in Lines 16-22 of Algorithm 3. In the triangle counting phase, for each edge, when the sum of the remaining elements in one of the two adjacency lists and the triangles we have already counted (TC) falls below $k-2$, we anticipate that the edge will be deleted. Thus, we start to mark all the edges subsequently detected by the intersections as affected. As a result, by the time we perform the affected edge determination phase, line 14 Algorithm 2, some of the affected edges have already been marked by the triangle counting phase. As a result, the affected edge determination phase needs to traverse the list of edges up to the indices at which the triangle counting phase started to mark the edges as affected. With this optimization, we reduce the total number of intersections performed by the triangle counting phase and the affected edge determination phase.
- Stream compaction is an expensive operation that can take more time than the inner *while loop* itself. Thus, we only perform stream compaction when the percentage of deleted edges exceeds some predefined threshold such as 10%.

Algorithm 1 Initialize

```
Input: E, Keep, Affected, Reversed
1: for each e ∈ $E$ do
2:     keep[e] ← TRUE
3:     affected[e] ← FALSE
4:     ep = FindIndex(u,v)
5:     Reversed[e] = ep
6: end for
```

B. Binary k-truss Decomposition

Trusssness of an edge represents the highest k-truss subgraph to which the edge belongs. Incremental enumeration of k values helps decide the trussness value of each edge in the graph. However, when only the edges of the maximal k-truss are required, the incremental approach is inefficient and time-consuming. To address this, we propose binary search based k truss decomposition algorithm.

However, for a binary search based approach to work, knowing the upper bound value of k is a necessity. We approximate the upper bound value for k using Algorithm 6 for a given graph. We note a k-truss with k nodes each with a degree of $k-1$ forms a clique [8]. In addition, it is also the minimum requirement to form a k-truss for the number of nodes and degree per node. *This implies that for the largest possible k-truss one needs to find the largest degree d for which there are at least $d+1$ nodes*. The value of $d+1$ becomes the upper bound value of k.

In Algorithm 6, we use a map with node degree as key and the number of nodes that have this degree as value. We traverse the map from the largest to smallest degree and keep aggregating the number of nodes until we reach the largest degree d for which there are $d+1$ aggregated number of nodes. We use the value of $d+1$ as the upper bound value of k to implement the binary k-truss algorithm. We make the following modifications to Algorithm 2:

- The outer *while loop* evaluates the next k as $k = (k_{min} + k_{upper\_bound})/2$ and runs until the range between k_{min} and k_{upper_bound} cannot be further divided.
- In the binary approach, two successive k values can be far apart. Therefore, before processing the next k value, it is very likely that there are a large number nodes with degree less than or equal to $k-1$. The edges of these nodes are not involved in the k-truss to be evaluated. Marking the edges of those nodes as 'deleted' reduces the total number of edges to be evaluated and improves performance. We do not perform this step for the incremental approach because the number of eliminated edges between successive k values is low and thus the overhead defies the benefit.
- After evaluating for a value of k:

  If the graph is not empty, perform stream compaction and set k_{min} to k.

  If the graph is empty, the evaluated k is larger than the maximal k and, thus, the 'Keep' array is set to true to fall back to the graph as in the previous state. We also set k_{upper_bound} to k.

Algorithm 2 Find k-truss

```
Input: E, NumEdges, Keep, Affected, Reversed
Output: $k$
1: Initialize(...)                                          ▷ Algo. 1
2: k ← 3
3: while TRUE do
4:     MoreAffected ← TRUE
5:     while MoreAffected do
6:         MoreAffected ← FALSE
7:         NumDelE, NumEAffBy ← 0
8:         for each e(u,v) ∈ E do
9:             if Keep[e] AND Affected[e] AND u<v then
10:                Affected[e] ← FALSE
11:                TC,s1,s2,l1,l2,NumEAffBy← IntersectTc(...)  ▷ Algo. 3
12:                if TC < $k-2$ then
13:                    Keep[e], Keep[Reserved[e]] ← FALSE
14:                    NumEAffBy ← IntersectAffect(...)      ▷ Algo. 4
15:                end if
16:                if !Keep[e] then
17:                    NumDelE + +
18:                end if
19:            end if
20:        end for
21:        if NumEAffBy> 0 then
22:            MoreAffected ← TRUE
23:        end if
24:    end while
25:    if NumDelE ==NumEdges then
26:        Break
27:    else
28:        $k$ + +
29:        E, NumEdges ← StreamCompact(...)
30:    end if
31: end while
32: return $k$
```

Algorithm 3 List Intersection for Triangle Counting

```
Input: u,v, RowPointers
Output: TC,s1,s2,l1,l2,NumEAffBy
1: $u_{ptr}$ ← RowPointer[u] , $v_{ptr}$ ← RowPointer[v]
2: $u_{end}$ ← RowPointer[u+1] , $v_{end}$ ← RowPointer[v+1]
3: FirstIntersect ← TRUE
4: while TC < $k-2$ AND $u_{ptr}, v_{ptr} < u_{end}, v_{end}$ do
5:     w1 = Zd[$u_{ptr}$]
6:     w2 = Zd[$v_{ptr}$]
7:     if w1=w2 then
8:         if keep[w1] AND keep[w2] then
9:             TC++
10:            if FirstIntersect then
11:                s1 ← $u_{ptr}$
12:                s2 ← $v_{ptr}$
13:                FirstIntersect ← FALSE
14:                nu, nv ← $u_{end} - u_{ptr}, v_{end} - v_{ptr}$
15:                if (nu AND nv ≥ $k-2-$TC) then
16:                    l1 ← $u_{ptr}+1$
17:                    l2 ← $v_{ptr}+1$
18:                else
19:                    NumEAffBy ← Affect(...)              ▷ Algo. 5
20:                end if
21:            end if
22:    else if w1<w2 then
23:        ++$u_{ptr}$
24:    else
25:        ++$v_{ptr}$
26:    end if
27:    end if
28: end while
```

IV. GPU Implementation

We use unified memory to store the graph and the auxiliary lists which are 'Keep', 'Affected' and 'Reversed' in both single

Algorithm 4 List Intersection for Affected Edges Determination

```
Input:  s1,s2,l1,l2
Output: NumEAffBy
 1: $u_{ptr}, v_{ptr}, u_{end}, v_{end} \leftarrow s1, s2, l1, l2$
 2: while $u_{ptr}, v_{ptr} < u_{end}, v_{end}$ do
 3:     w1 = Zd[$u_{ptr}$]
 4:     w2 = Zd[$v_{ptr}$]
 5:     if w1=w2 then
 6:         NumEAffBy ← Affect(...)                ▷ Algo. 5
 7:     else if w1<w2 then
 8:         $++u_{ptr}$
 9:     else
10:         $++v_{ptr}$
11:     end if
12: end while
```

Algorithm 5 Affected Edges Determination

```
Input:  $u_{ptr}, v_{ptr}$, Keep, Affected, Reversed, NumAffE
Output: NumAffE
 1: NumAff ← NumAffE
 2: y1 ← Reserved[$u_{ptr}$]
 3: y2 ← Reserved[$v_{ptr}$]
 4: if !Affected[$u_{ptr}$] then
 5:     !Affected[$u_{ptr}$] ← $TRUE$
 6:     NumAffE $++$
 7: end if
 8: if !Affected[$v_{ptr}$] then
 9:     Affected[$v_{ptr}$] ← $TRUE$
10:     NumAffE $++$
11: end if
12: if !Affected[y1] then
13:     Affected[y1] ← $TRUE$
14:     NumAffE $++$
15: end if
16: if !Affected[y2] then
17:     Affected[y2] ← $TRUE$
18:     NumAffE $++$
19: end if
20: return NumAffE
```

Algorithm 6 Approximating initial k upper bound(k_{ub})

```
Input:  DegMap #DegMap=map⟨degree, NumberOfNodes⟩
 1: TotalNodes = 0
 2: for each (degree, NumberOfNodes) ∈ DegMap do
 3:     CurrentDegree ← degree+1
 4:     TotalNodes ← TotalNodes + NumberOfNodes
 5:     if TotalNodes > CurrentDegree then
 6:         $k_{ub}$ = CurrentDegree
 7:         break
 8:     end if
 9: end for
10: return $k_{ub}$
```

GPU and multi-GPU implementation. The graph we use is undirected and stored in COO format with an additional row pointer array adopted from standard CSR format. We call this hybrid format COO+CSR. For both implementations, we use a single CUDA kernel, we call it the core kernel, to perform the *for loop*, lines 8-20 in Algorithm 2. A thread is assigned to an edge to perform triangle counting and edge affecting steps. The core kernel writes to 'Keep' and 'Affected' lists and outputs two values: the number of deleted edges and a hint to indicate if there are any affected edges. For triangle counting step, we use the standard two-pointer intersection algorithm, with the additional optimizations explained in Section III to improve affecting edges step. To count the deleted edges, we perform reduction operation at the block and then at the grid levels. The core kernel repeats until there is no more affected edges. To perform stream compaction on source and destination lists, we use device-wide partition routine using Nvidia CUB [10] to separate kept and deleted edges. Then, we launch two other kernels to: a) rebuild row pointer list and mark all edges as kept and not affected, and b) rebuild 'Reversed' list.

Multi-GPU Specific Considerations: In the 2018 multi-GPU implementation, edges were divided evenly across GPUs such that a single k was evaluated at the end of each iteration. All the auxiliary data structures and the graph are stored in the unified memory. By default, sharing unified memory pages between GPUs causes many page faults due to page migration among GPUs and CPU. Moreover, page faults occur even if the shared pages are read-only. This makes the multi-GPU solution very slow and infeasible.

To address the above limitation, we use CUDA unified memory hints. Since the graph data structure and 'Reversed' lists are read only, we set the *cudaMemAdviseSetReadMostly* hint for these two data structures prior to the core kernel executions such that the pages are duplicated avoiding GPU page faults. This significantly improves the core kernel performance. However, stream compaction physically changes the graph and 'Reversed' lists. Thus, updating these lists while the read-mostly hint is set can still generate many page faults. To repress this, before the stream compaction step, we unset the read-mostly hint using *cudaMemAdviseUnsetReadMostly* for these data structures and set it again after the execution of the stream compaction step.

Although the unified memory hints reduces the number of page faults, it cannot remove the page faults that occurs from the 'Affected' list. This list is randomly read and updated by multiple GPUs throughout the core kernel execution. Memory hints are useless for this list as maintaining coherent view of the data is still needed across GPUs. One possible solution is to privatize the Affected list such that each GPU has its own copy. However, GPUs would need to merge and distribute their results among them in every iteration of the inner *while loop*, line 5 of Algorithm 2. This becomes an expensive solution contributing to significant increase in memory traffic between GPUs and CPU.

To overcome aforementioned limitation, we propose to privatize 'Keep' and 'Affected' lists and assign each GPU with a consecutive k value to execute the outer loop of Algorithm 2. Compared to the program execution in the single GPU case, multi-GPU approach executes the incremental k-trusss decomposition algorithm in strides as shown in Figure 1.

Each box in the Figure 1 represents an iteration of the k-truss decomposition algorithm. Let's assume 'S' as the number of edges in the input graph and 'd', a constant (for simplicity) the number of deleted edges in each iteration step. In the case of single-GPU implementation, in each iteration of k value, a new graph of size equal to the $S - d$ is provided as input to check if the sub-graph is still a truss.

However, in case of multi-GPU implementation, each GPUs concurrently execute the core kernel for consecutive k values

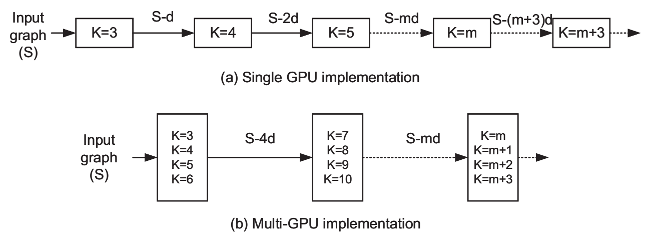

Figure 1. An example of the single and multi-GPU incremental approach timeline.

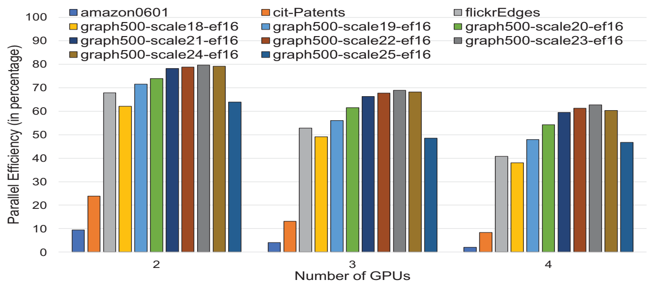

Figure 2. Parallel efficiency of incremental multi-GPU k-truss decomposition algorithm compared with single GPU implementation.

on the same input graph in each stride. At the end of each stride, all GPUs synchronize at line 25 present in Algorithm 2 to ensure there are no more affected edges found by any other GPU.

We perform stream compaction using the 'Keep' of the largest k value in current stride in case all the GPUs have non-empty graphs and feed the resulting graph as input graph to the next stride. Otherwise, k_{max} is found and we exit the outer *while loop*, lines 3-31 of Algorithm 2. We linearly scan across the GPUs to determine the k_{max} value. The proposed approach completely avoids any sharing of 'Affected' and 'Keep' lists between the GPUs and removes the page fault overhead thus providing higher performance. It is important to mention here that since each GPU has a different k value to process, load imbalance is known to happen. Other GPUs has to wait for the GPU that evaluates the largest k value at each iteration. However, we were able to solve a fundamental limitation to scaling out with this approach, which is the privatization of 'Keep' and 'Affected' lists.

Processing Large Graphs: Processing large graphs such as "Twitter Follower", even with the binary-based k-truss decomposition, is still a challenge on its own. To efficiently process large graphs, we add an additional optimization to our binary-based k-truss decomposition algorithm with the observation that the initial steps of k computation consumes large amount of memory and compute. Our empirical study of graphs shows that the maximum k value is always larger than 5% of k_{upper_bound}. Thus, before the first iteration of the binary approach, we prune the nodes along with their edges having out-degree less than or equal to 5% of k_{upper_bound}. It significantly helps to reduce the number edges to be processed by the core kernel resulting in further performance improvement. After the first iteration of k-truss, we set k_{min} to the current k value and execute the binary approach as explained in Section III-B. If the graph is empty graph after the first iteration of k-truss implies that our estimation of the first k has exceeded the maximum k value. In this case, we fall back to the original copy of the graph.

V. Results and Discussion

In this section, we discuss how our optimized approach compares with our prior submission. First, we will discuss the performance improvements of the incremental k-truss decomposition approach. Second, we will discuss how our optimized incremental approach scales with multi-GPU implementation. Third, we evaluate the binary-based k-truss decomposition approach and show that it is significantly faster when only the maximum trussness is sought. Lastly, to measure the benefit of the binary-based k-truss decomposition approach, we use a large graph dataset "Twitter" having 2.8 billion edges and show that maximum trussness can be found in just 16 minutes using a single V100 GPU.

Evaluation Platform and metrics: We use a node from the NCSA HAL cluster to evaluate the proposed approaches [11]. The node has two sockets, each having 20-core IBM POWER9 CPU@2.4 GHz. Each CPU hosts 2 NVIDIA V100 GPUs with 16 GB HBM2. There is 72TB RAID array, NFS-mounted via IB EDR. The machine has a DRAM size of 256GB and has NVLink bandwidth of 150 GB/s. We use CUDA version 10.1 to run the experiments. The system described is very similar to the one we used previous year except the last year submission used CUDA version 9.2 and had 512GB of DRAM memory.

We use the evaluation metrics provided by the Graph Challenge to represent our results and they include total number of edges, total number of nodes, and edges (bidirectional) per second. Table I presents the results for our single GPU implementation using the incremental and binary k-truss approaches. The measurements include the time taken by all the steps executed after reading the graph on CPU.

Single-GPU Incremental approach: From Table I, the proposed incremental approach outperforms our baseline for all the graphs by up to 35.2x (6.9x on average). This performance improvement can be attributed to the removal of unnecessary reduction step that was occurring in the inner *while loop*, optimized triangle counting as described in Section III-A and minimizing the number of stream compaction steps. These performance optimizations have enabled us to execute graphs that has more than a billion edges using a single V100 GPU with the help of unified memory in a reasonable time which was infeasible in 2018 submission.

Multi-GPU incremental approach: Next, we discuss the strong scaling of the incremental approach. From Figure 2, we note that the performance improves as the number of GPUs increases for the graphs with high maximum k-truss. The maximum achieved speedup is up to 2.5x with 4 GPUs compared with the single-GPU implementation. However, the parallel scaling efficiency is still sub-optimal compared to the theoretical efficiency due to the overhead associated with the multi-GPU implementation such as multiple kernel

TABLE I

SINGLE GPU IMPLEMENTATION RESULTS FOR THE PROPOSED INCREMENTAL AND BINARY COMPARED TO OUR 2018 SUBMISSION.

Graph	n	m	kmax	Newell (2018 submission) Edges/s	HAL Incremental (2019 Submission) Edges/s	Speedup vs 2018 submission	HAL Binary (2019 submission) Edges/s	Speedup vs 2018 submission
as20000102	6,474	25,144	10	463,223	3,024,380	6.5	4,740,010	10.2
ca-GrQc	5,242	28,968	44	202,471	2,326,810	11.5	11,687,000	57.7
oregon1_10331	10,670	44,004	16	453,213	2,433,110	5.4	10,251,700	22.6
ca-HepTh	9,877	51,946	32	407,249	14,326,700	35.2	24,671,100	60.6
p2p-Gnutella04	10,876	79,988	4	7,140,476	23,577,000	3.3	42,050,500	5.9
as-caida20071105	26,475	106,762	16	670,506	4,544,100	6.8	8,293,890	12.4
facebook_combined	4,039	176,468	97	159,387	1,390,120	8.7	9,514,610	59.7
ca-CondMat	23,133	186,878	26	1,416,010	8,272,540	5.8	62,425,700	44.1
ca-HepPh	12,008	236,978	239	263,878	3,299,680	12.5	26,789,300	101.5
email-Enron	36,692	367,662	22	963,079	8,394,100	8.7	19,156,400	19.9
ca-AstroPh	18,772	396,100	57	862,452	6,119,610	7.1	69,090,800	80.1
loc-brightkite_edges	58,228	428,156	43	889,012	9,116,940	10.3	37,786,600	42.5
cit-HepTh	27,770	704,570	30	1,159,954	9,674,120	8.3	20,808,800	17.9
email-EuAll	265,214	728,962	20	2,623,830	27,098,600	10.3	26,499,100	10.1
soc-Epinions1	75,879	811,480	33	1,181,446	8,412,780	7.1	16,258,600	13.8
cit-HepPh	34,546	841,754	25	1,905,473	21,494,900	11.3	45,908,000	24.1
soc-Slashdot0811	77,360	938,360	35	1,907,923	18,399,200	9.6	34,133,000	17.9
soc-Slashdot0902	82,168	1,008,460	36	1,979,478	18,638,700	9.4	33,649,200	17.0
amazon0302	262,111	1,799,584	7	33,140,027	178,984,000	5.4	175,205,000	5.3
roadNet-PA	1,088,092	3,083,796	4	93,106,578	282,585,000	3.0	243,933,000	2.6
roadNet-TX	1,379,917	3,843,320	4	98,403,250	290,285,000	2.9	242,104,000	2.5
flickrEdges	105,938	4,633,896	574	641,408	1,854,380	2.9	34,883,300	54.4
amazon0312	400,727	4,699,738	11	26,760,310	131,725,000	4.9	196,273,000	7.3
amazon0505	410,236	4,878,874	11	25,936,061	122,850,000	4.7	179,852,000	6.9
amazon0601	403,394	4,886,816	11	28,369,638	131,233,000	4.6	188,468,000	6.6
roadNet-CA	1,965,206	5,533,214	4	115,721,301	295,564,000	2.6	252,762,000	2.2
cit-Patents	3,774,768	33,037,894	36	41,798,278	161,032,000	3.9	187,762,000	4.5
graph500-scale18-ef16	174,147	7,600,696	159	1,102,217	2,419,760	2.2	18,769,600	17.0
graph500-scale19-ef16	335,318	15,459,350	213	969,474	2,042,130	2.1	13,279,800	13.7
graph500-scale20-ef16	645,820	31,361,722	284	751,343	1,669,410	2.2	11,850,600	15.8
graph500-scale21-ef16	1,243,072	63,463,300	373	476,772	1,143,670	2.4	4,725,110	9.9
graph500-scale22-ef16	2,393,285	128,194,008	485	296,406	814,933	2.7	4,303,600	14.5
graph500-scale23-ef16	4,606,314	258,501,410	625	187,628	586,391	3.1	3,651,540	19.5
graph500-scale24-ef16	8,860,451	520,523,686	791	-	387,454	-	3,063,770	-
graph500-scale25-ef16	17,043,781	1,046,934,896	996	-	244,705	-	902,447	-

synchronization, kernel launch overheads and many others. For the 'cit-Patents' and 'Amazon0601' graphs, we observe a significant slow down on the execution time as we move from single GPU to four GPUs. The core kernel execution for these two graphs in the single GPU is quiet fast compared to the rest of the graphs as shown in Table I. This implies as we move to two or more GPUs the overhead from multi-GPU stream compaction becomes significant compared to the core kernel execution and is the primary limiter for the scaling. Stream compaction of the graph requires frequent movement of the data across GPUs. However, for the other graphs, the time taken by the core kernel is still larger than the multi-GPU stream compaction step. Thus, the scaling efficiency is comparatively better.

Binary k-truss decomposition: Next let us dive into binary k-truss decomposition approach where the user only needs to the know the maximum truss in a input graph. Since the binary approach allows to skip computing several k values, we obtain significant speed up, up to 101.5x (24.3x on average) with the single GPU implementation compared to our 2018 baseline as shown in Table I. Comparing to our proposed incremental approach, the binary approach provides a speed up of 18.8x (4.1x on average). Considering the graphs with high maximum k-truss, such as 50 and above, the binary implementation is 7.8x and 38.6x faster on average than our proposed incremental and the baseline implementations, respectively. However, the proposed incremental approach is faster than the binary approach when the maximum k value is very small such as roadNet graphs.

To show the benefit of binary k-truss decomposition approach, we process ”Twitter Follower” graph having 41 million vertices, 1.4 billion edges (2.8 billion bidirectional edges), and 34 billion triangles [12]. Incremental approach could not complete calculating the maximum k-truss value in a feasible time (hours). However, the binary approach processed the graph to find its maximum k-truss just less than 16 minutes on a single NVIDIA V100 GPU. This clearly shows the benefit of binary approach if the user is only interested in knowing the maximum trussness of the graph.

VI. EXTRA CONTRIBUTIONS TO THE COMMUNITY

Our k-truss decomposition code is in the Pangolin library, currently available at https://github.com/c3sr/pangolin

VII. CONCLUSION AND FUTURE WORK

In this paper, we presented multiple algorithmic optimizations for k-truss decomposition. We also introduced binary k-truss decomposition to find the maximum k-truss. Compared to our last year's submission, we achieved an increase in speed up to 35.2x and 151.3x on single and multi-GPUs, respectively. For the binary approach, we were able to increase the speed up to 101.5x on a single GPU. Moreover, we showed how the binary search approach can be leveraged to process "Twitter Follower Graph" on a single GPU in just under 16 minutes to find maximum k-truss. Our next step to improve the performance is to perform graph partitioning as currently the scaling efficiency is still limited by data transfer overhead across GPUs.

Acknowledgments

This work is supported by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizons Network. This work utilizes resources supported by NSF-MRI program, grant #1725729, and University of Illinois at Urbana-Champaign. This work is partly supported by the Center for Applications Driving Architectures (ADA) and Center for Research on Intelligent Storage and Processing-in-memory (CRISP), JUMP Centers with prime award coming from Semiconductor Research Corporation (SRC).

References

[1] V. S. Mailthody, K. Date, Z. Qureshi, C. Pearson, R. Nagi, J. Xiong, and W. Hwu, "Collaborative (cpu + gpu) algorithms for triangle counting and truss decomposition," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, Sep. 2018, pp. 1–7.

[2] R. Pearce and G. Sanders, "K-truss decomposition for scale-free graphs at scale in distributed memory," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, Sep. 2018, pp. 1–6.

[3] M. Bisson and M. Fatica, "Update on static graph challenge on gpu," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, Sep. 2018, pp. 1–8.

[4] T. M. Low, D. G. Spampinato, A. Kutuluru, U. Sridhar, D. T. Popovici, F. Franchetti, and S. McMillan, "Linear algebraic formulation of edge-centric k-truss algorithms with adjacency matrices," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, Sep. 2018, pp. 1–7.

[5] A. Conte, D. De Sensi, R. Grossi, A. Marino, and L. Versari, "Discovering$k$-trusses in large-scale networks," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, Sep. 2018, pp. 1–6.

[6] C. Voegele, Y. Lu, S. Pai, and K. Pingali, "Parallel triangle counting and k-truss identification using graph-centric methods," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017, pp. 1–7.

[7] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini, and G. Karypis, "Truss decomposition on shared-memory parallel systems," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017, pp. 1–6.

[8] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National Security Agency Technical Report*, p. 16, 2008.

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

[10] D. Merrill, "Cub," 2016.

[11] Mar 2019. [Online]. Available: https://wiki.ncsa.illinois.edu/display/ISL20/HALcluster

[12] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 591–600.