KTRUSSEXPLORER: Exploring the Design Space of K-truss Decomposition Optimizations on GPUs

Safaa Diab*, Mhd Ghaith Olabi*, Izzat El Hajj
American University of Beirut
{syd04, moo02}@mail.aub.edu, izzat.elhajj@aub.edu.lb

*Equal contribution

***Abstract*—K-truss decomposition is an important method in graph analytics for finding cohesive subgraphs in a graph. Various works have accelerated k-truss decomposition on GPUs and have proposed different optimizations while doing so. The combinations of these optimizations form a large design space. However, most GPU implementations focus on a specific combination or set of combinations in this space.**

This paper surveys the optimizations applied to k-truss decomposition on GPUs, and presents KTRUSSEXPLORER, a framework for exploring the design space formed by the combinations of these optimizations. Our evaluation shows that the best combination highly depends on the graph of choice, and analyses the conditions that make each optimization attractive. Some of the best combinations we find outperform previous Graph Challenge champions on many large graphs.

I. INTRODUCTION

A k-truss is a subgraph of a graph such that every edge in the subgraph participates in at least $k-2$ triangles in the subgraph. K-truss decomposition discovers the maximal k-trusses in a graph for $k \geq 2$. Samsi et al. [11] describe the problem in detail. A popular algorithm for finding a k-truss is to count the triangles that each edge participates in (i.e., an edge's *support*), delete the edges with support less than $k-2$ (i.e., *weak* edges), and iteratively repeat this process until no edges are deleted. Triangles are typically found by intersecting the adjacency lists of an edge's endpoints.

Various works have accelerated k-truss decomposition on GPUs and have proposed different optimizations while doing so. These optimizations include using a directed graph [4], [9], directing edges by degree [4], [9], tiling the adjacency matrix [9], [13], parallelizing list intersection operations [4], [9], removing deleted edges from the graph data structure in between iterations [4], [5], [6], only recounting the support of edges affected by deletions [1], [4], [6], [8], [10], and others. The combinations of these optimizations form a large design space. However, most GPU implementations focus on a specific combination or set of combinations in this space.

To address this issue, we present KTRUSSEXPLORER, a framework for exploring the design space of k-truss decomposition optimizations on GPUs. KTRUSSEXPLORER consists of multiple configurable kernel implementations. The framework is highly parameterized, allowing users to specify any combination of many of the optimizations mentioned previously. Our evaluation shows that the best optimization combination highly depends on the graph of choice. We also perform a quantitative analysis of the conditions that make each optimization attractive. Some of the best combinations we find outperform previous Graph Challenge champions on many large graphs. KTRUSSEXPLORER has been open-sourced to help further advance research on k-truss decomposition optimizations.

II. KTRUSSEXPLORER

This section surveys the literature on optimizations for k-truss decomposition on GPUs while describing how these optimizations are supported in KTRUSSEXPLORER.

A. Edge-Centric and Vertex-Centric Parallelization

One key distinction between parallel implementations is whether they are edge-centric [1], [6], [10], [12] or vertex-centric [3], [4]. Edge-centric implementations assign a thread (or group of threads) to each edge to find the support of that edge. Vertex-centric implementations assign a thread (or group of threads) to each vertex to find the support of that vertex's outgoing edges.

The edge-centric approach makes load balancing easier than the vertex-centric approach where different vertices may have a substantially different number of outgoing edges whose support needs to be computed. On the other hand, the vertex-centric approach works with a CSR representation of the graph whereas the edge-centric approach typically uses more memory to store a COO+CSR representation to look up the endpoints of each edge. Moreover, the vertex-centric approach enables optimizations that target edges that share the same source vertex, such as laying out the source vertex's adjacency list as a bitmap for all its outgoing edges to share [3], [4].

Linear algebraic formulations [5] can also be classified as edge-centric or vertex-centric depending on how the sparse matrix multiplication is implemented. Assigning threads to nonzeros corresponds to an edge-centric implementation, whereas assigning threads to rows corresponds to a vertex-centric implementation.

KTRUSSEXPLORER currently only implements the edge-centric parallelization approach.

B. Graph Directedness

Another key distinction between implementations is the use of undirected graphs [1], [8] or directed graphs [3], [4], [9]. In

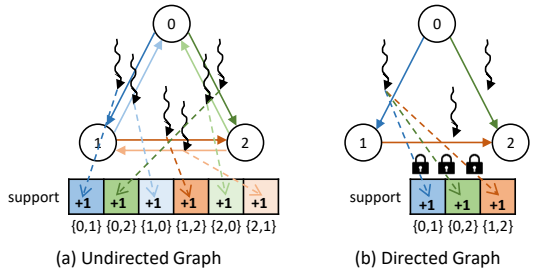


Fig. 1: Discovering Triangles Based on Graph Directedness

linear algebraic terms, using directed graphs is equivalent to operating on triangular matrices [5]. The distinction between using undirected and directed graphs is illustrated in Fig. 1. With an undirected graph, each edge's thread discovers the triangles that the edge participates in independently. However, each triangle is redundantly discovered six times as shown in Fig. 1(a). With a directed graph, each triangle is discovered only once. For example, in Fig. 1(b), only edge $\{0,1\}$'s endpoints have a common neighbor so only that edge's thread will discover a triangle. However, although redundancy is reduced, the thread that discovers the triangle must update the support of all edges involved which requires atomic operations.

Using a directed graph has many advantages. First, fewer edges are stored which reduces memory capacity and bandwidth requirements. Second, triangles are not discovered redundantly which reduces the amount of work. Third, adjacency lists are shorter which makes intersecting them faster.

On the other hand, using undirected graphs also has advantages owing to the fact that each edge's thread finds the edge's support independently. First, no atomic operations are needed to update support values. Second, a thread may stop an intersection operation earlier if certain conditions are met [12]. One such condition is when one of the two adjacency lists is smaller than $k-2$. In this case, it is impossible for the thread's edge to participate in $k-2$ triangles so there is no need to attempt the intersection. Another such condition when $k-2$ triangles are found but the intersection operation is not yet complete. In this case, the thread does not need to continue the operation because it has already established that the edge is not weak. These early stopping conditions cannot be applied with directed graphs because a thread is also responsible for updating the support of other edges, not just the one it owns.

Still, undirected graphs suffer from the redundancy of discovering triangles six times. As an optimization, some works use undirected graphs but only count triangles for one of the edge directions [6], [10], [12] which reduces the redundancy from six to three. In linear algebraic terms, this optimization is equivalent to using the full adjacency matrix as an input, but only computing a triangular matrix as an output, taking advantage of the fact that the output is symmetric. The drawback of this approach is that if an edge has weak support and needs to be deleted, finding the reverse edge to delete it as well can be expensive.

KTRUSSEXPLORER provides an option for selecting between directed and undirected graphs. For undirected graphs, both conditions for stopping the intersection operation early are implemented, and triangles are counted for both directions of each edge (i.e., each triangle is discovered six times).

C. Directing Edges by Degree

A simple approach for converting an undirected graph to a directed one is to keep only the edges from the vertex with the lower index to the vertex with the higher index, i.e., direct edges by index. Another approach is to keep the edges from the vertex with lower degree to the vertex with higher degree, i.e., direct edges by degree. The latter approach is good for keeping adjacency lists short. With undirected graphs, high-degree vertices have long adjacency lists which are expensive to intersect. Directing edges by degree significantly shrinks the adjacency lists of high-degree vertices.

Directing edges by degree has been done in two ways. One way is to simply remove the edges from higher-degree to lower-degree vertices from the graph [6], [9]. Another way is to sort vertices by increasing degree, relabel vertices according to their sort index, and then direct edges by index [4].

KTRUSSEXPLORER provides an option for directing graphs by index or by degree. Directing by degree is currently implemented by sorting, relabelling, then directing by index.

D. Tiling

Tiling refers to partitioning of the adjacency matrix into tiles. Tiling has been evaluated in the literature for triangle counting on GPUs [9], [13]. KTRUSSEXPLORER applies tiling to k-truss decomposition on GPUs. We discuss tiling at more length than other optimizations because it has received less attention in the literature and its implementation varies across different works.

An example of how tiling is implemented in KTRUSSEXPLORER is shown in Fig. 2. Fig. 2(b) shows the logical adjacency matrix of the graph in Fig. 2(a) and Fig. 2(c) shows how it is physically stored as a CSR data structure (we use a hybrid COO+CSR but omit the COO part in the figure). Fig. 2(d) shows the same adjacency matrix with tiling applied and Fig. 2(e) shows how it is physically stored as a tiled CSR data structure. The matrix is logically divided into 2D square tiles, with each tile's edges stored contiguously. We refer to the number of tiles in each dimension as the *tiling factor*.

Fig. 3 is used to illustrate the advantages of tiling. In each subfigure, the three matrices represent different logical views of the same physical data structure. The view to the bottom right depicts how threads are assigned to edges. The view to the left depicts how a thread traverses the adjacency list of its edge's source vertex. The view on top depicts how a thread traverses the adjacency list of its edge's destination vertex.

The first advantage of tiling is that it improves data locality. In Fig. 3(a), the first four threads in the grid are assigned to the first four edges of the CSR data structure which all have the same source vertex but different destination vertices. Hence, there is high spatial and temporal locality when accessing the

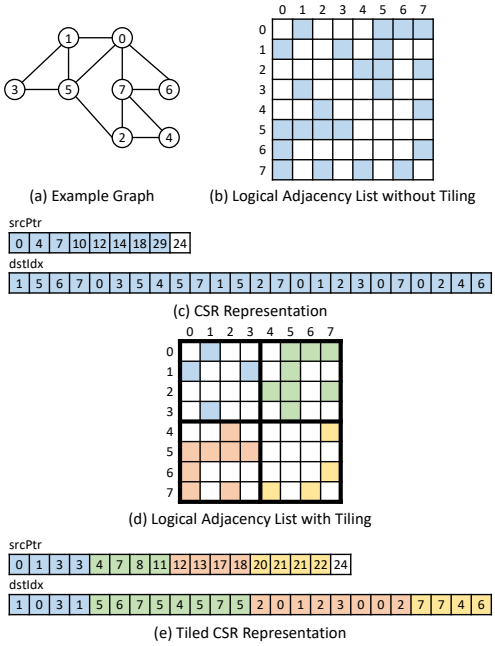


Fig. 2: Example of Tiling an Adjacency Matrix

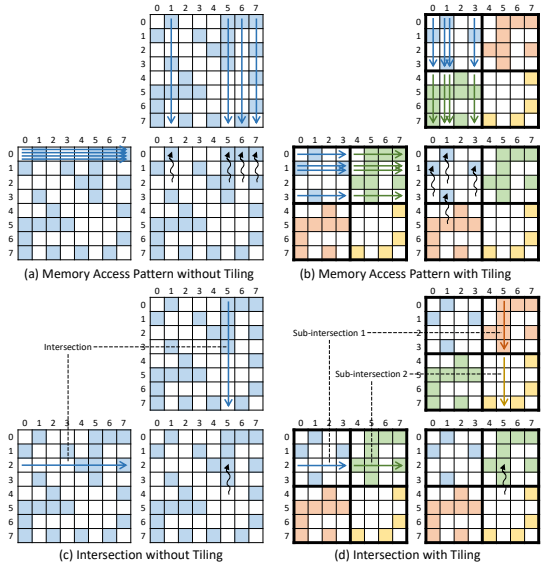


Fig. 3: Impact of Tiling

source vertex adjacency lists, but poor spatial and temporal locality when accessing the destination vertex adjacency lists. In contrast, in Fig. 3(b), the four threads are assigned to the first four edges of a tile. In this case, accesses to both source and destination vertex adjacency lists exhibit good spatial and temporal locality because of tiling.

We also tried loading tiles to shared memory for faster access, however, we found that doing so significantly hurt performance. The reason is that some data in a tile may not be needed so loading that data to shared memory is wasteful. For example, in Fig. 3(b), vertex 2's adjacency list is not needed by any thread. Loading the entire tile to shared memory would result in vertex 2's adjacency list being loaded even though it is not needed. For this reason, we do not use shared memory and rely on the L1 cache for fast access to reused data.

The second advantage of tiling is that it partitions long intersection operations into multiple shorter *sub-intersections*. This partitioning makes intersection operations faster because it allows skipping edges in a sublist if the other sublist is empty or has reached the end. For example, in Fig. 3(c), the thread assigned to edge $\{2, 5\}$ needs multiple steps to intersect the adjacency lists of vertices 2 and 5. In contrast, in Fig. 3(d) with tiling, the intersection is partitioned into two sub-intersections. Sub-intersection 1 takes zero steps because the sublist of vertex 2's adjacency list is empty. Sub-intersection 2 also takes zero steps because the sublist of vertex 5's adjacency list is empty. Therefore, the intersection takes fewer steps overall and loads less data from memory. This affect is similar to that achieved by Intersect-Path [7], except that Intersect-Path finds sublists dynamically rather than partitioning the data structure.

KTRUSSEXPLORER provides an option for specifying whether or not to apply tiling and the tiling factor of choice.

E. Parallelizing Intersections

Various works parallelize individual intersection operations to further extract parallelism from the computation. This parallelization typically divides one of the lists across multiple threads and has each thread find the corresponding edge(s) in the other list via binary search [9] or a bitmap lookup [3], [4].

KTRUSSEXPLORER parallelizes intersections in the context of tiling. Recall that with tiling, each intersection operation is partitioned into multiple sub-intersections. These sub-intersections can be performed in parallel. KTRUSSEXPLORER assigns multiple threads to each intersection operation (up to the tiling factor) and divides the sub-intersections across these threads. Each sub-intersection is performed sequentially by one thread. In the absence of tiling, the whole intersection operation is performed sequentially by one thread.

We refer to the number of threads assigned per intersection as the *parallelization factor*. KTRUSSEXPLORER provides an option for experimenting with different parallelization factors.

F. Removing Deleted Edges Intermediately

When a weak edge is deleted during an iteration, it may be marked as deleted and kept in the graph data structure or it may be removed from the data structure entirely before the next iteration. The advantage of removing edges intermediately is that it shrinks adjacency lists making intersections faster, and reduces the graph's memory footprint. The disadvantage is that removing edges is more expensive than simply marking them as deleted. For this reason, some works do not remove edges in between iterations [9], [10], [12], some remove them every iteration [3], [4], [5], [6], and some remove them only if enough edges have been deleted [1].

Removing deleted edges from the graph data structure can be done via a stream compaction operation that filters out edges marked as deleted. Stream compaction can be done on the entire edge list [1] or on each vertex's adjacency

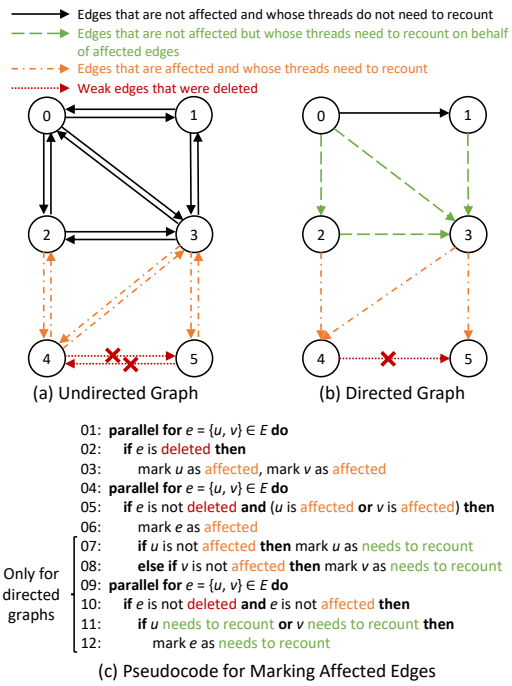


Fig. 4: Recomputing Support Values for Affected Edges

list separately [3], [4], [5]. Compacting the entire edge list is more expensive because the array to be compacted is larger, and because the CSR pointers need to be recomputed after the compaction. On the other hand, compacting each vertex's adjacency list separately requires more metadata to track where each vertex's adjacency list ends.

KTRUSSEXPLORER provides an option for specifying for how many initial iterations (if any) edges should be removed. This approach is based on the observation that the largest number of edges is removed in the first few iterations and decreases as iterations proceed. Edges are removed via stream compaction of the entire edge list using the in-place stream compaction primitive in Thrust [2]. Compacting each vertex's adjacency list separately is not yet supported.

G. Recomputing Support Values

Deleting a weak edge breaks the triangles that the edge participates in which may affect the support of the edges that share a vertex with the deleted edge. These edges are referred to as *affected* edges. When recomputing support values across iterations, it is sufficient to recompute the support of affected edges and unnecessary to recompute for all edges.

The advantage of recomputing the support of only affected edges is that it eliminates unnecessary work. However, the disadvantage is that it incurs additional overhead for identifying which edges are affected. For this reason, some works recompute the support of all undeleted edges [5], [9], [12] while others recompute the support of only the affected edges [1], [3], [4], [6], [8], [10].

Deciding which edges' threads should recount triangles differs between undirected graphs and directed graphs. This distinction is shown in Fig. 4. In the undirected graph in Fig. 4(a), each thread is solely responsible for computing its edge's support. Therefore, only the affected edges' threads need to recount triangles. However, for directed graphs, an edge may not be affected, but its thread may be responsible for finding a triangle on behalf of an affected edge and updating its support. Therefore, it is not sufficient for only the affected edges' threads to recount. Threads assigned to any edge that shares a vertex with an affected edge must also recount. For example, in Fig. 4(b), edge $\{2,3\}$ is not affected by the deletion of edge $\{4,5\}$. However, its thread must recount because it must notify edge $\{3,4\}$ which is affected that it is part of the triangle $\{2,3,4\}$. The pseudocode for marking affected edges is shown in Fig. 4(c).

Another optimization related to recomputing support values is specific to the case when $k = 3$. When $k = 3$, any edge deleted in the first iteration does not belong to a triangle. Hence, no triangles are broken in the first iteration, so no further iterations are needed to recompute support values [3].

KTRUSSEXPLORER provides an option for specifying whether to recompute support values of all undeleted edges or only affected edges for both directed and undirected graphs. If $k = 3$, only one iteration is performed.

TABLE I: Design Space Explored

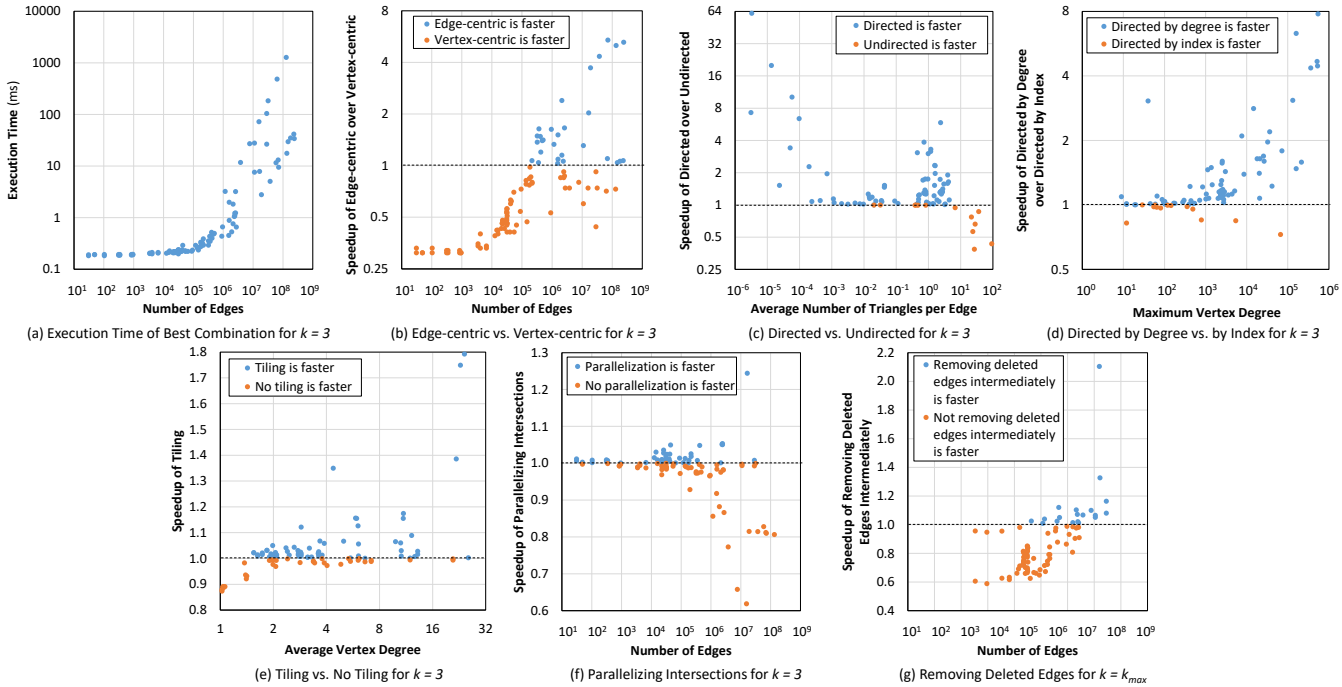
Optimization	Options
Graph Directedness	undirected, directed by index, directed by degree
Tiling	no tiling, tiling with a factor of {2, 4, 8, 16}
Parallelizing Intersections	no parallelization, parallelization with a factor of {2, 4, 8, 16}
Removing Edges Intermediately	no iterations, first {1, 2, 4, 8} iterations, all iterations (edges are also always removed at the end for all combinations)
Recomputing Support Values	all edges, affected edges

III. METHODOLOGY

We evaluate KTRUSSEXPLORER using a Volta V100 GPU with 16GB of device memory coupled with an AMD EPYC 7551P CPU with 15GB of main memory. We evaluate with all the data sets in the graph challenge collection [11] except for Friendster, graph500-scale24-ef16, and graph500-scale25-ef16 due to limited device memory capacity. We report results for $k = 3$, and for $k = k_{max}$ when k_{max} is not 3.

The design space explored is summarized in Table I. For $k = 3$, the space is searched exhaustively for graphs with less than 17 million edges. For larger graphs, only a subset of the combinations are searched based on which combinations did best with the other graphs. For graphs where the best parallelization factor was 16, a parallelization factor of 32 is also attempted. Since $k = 3$ only needs one iteration to converge, edges are removed once at the end and recomputing support values is not relevant.

For each combination, we report the mean of 10 runs after discarding 5 warm up runs. For large graphs, we take the mean of 5 runs with no warm up runs. The time reported includes: counting triangles, marking edges as deleted, data transfer from device to host to check for convergence, removing deleted edges intermediately for relevant combinations, removing deleted edges at the end for all combinations, and marking affected edges for relevant combinations. The time reported



(a) Execution Time of Best Combination for $k = 3$
(b) Edge-centric vs. Vertex-centric for $k = 3$
(c) Directed vs. Undirected for $k = 3$
(d) Directed by Degree vs. by Index for $k = 3$
(e) Tiling vs. No Tiling for $k = 3$
(f) Parallelizing Intersections for $k = 3$
(g) Removing Deleted Edges for $k = k_{max}$

Fig. 5: Evaluation

does not include: allocation and deallocation time, initial and final copy time, and initial and final graph conversion time (COO to CSR, relabelling vertices, undirected to directed).

IV. Evaluation

Fig. 5(a) shows how the execution time (in *milliseconds*) for the best optimization combination scales with the number of edges for $k = 3$. Details on the best combination found for each graph and the breakdown of the execution time can be found in Table II for $k = 3$ and Table III for $k = k_{max}$. It is clear that there is no best combination for all graphs and that the best combination depends on the graph of choice. In the rest of this section, we quantitatively analyze the conditions that make each design decision or optimization attractive.

A. Edge-centric and Vertex-centric Parallelization

Since we do not currently support a vertex-centric implementation, we compare the edge-centric implementation in KTrussExplorer to the vertex-centric implementation in Bisson & Fatica [4] using the kernel times reported in that work. Bisson & Fatica [4] are also the 2018 Graph Challenge champions. Fig. 5(b) shows the speedup of the edge-centric implementation over the vertex-centric implementation for $k = 3$. It is clear that the vertex-centric implementation is superior for smaller graphs, but as the graphs get larger, the edge-centric implementation becomes competitive on many graphs (up to 5.4× faster). Recall from Section II-A that the advantage of edge-centric parallelization over vertex-centric parallelization is that edge-centric parallelization makes load balancing easier. The importance of load balance becomes more pronounced as the graphs grow in size.

Note however that this comparison has some limitations because different optimizations are applied to each implementation so there may be other factors impacting the performance difference. For this reason, we plan to support a vertex-centric implementation in KTrussExplorer as future work in order to have a more controlled comparison.

B. Graph Directedness

Table II shows that most graphs have better performance when a directed graph is used. However, a significant number of graphs perform better when an undirected graph is used. The preference for undirected graphs is independent of graph size as there are very small as well as very large graphs that do better when the graph is undirected. We observe that the preference for undirected graphs is correlated with the average number of triangles per edge in the graph.

Fig. 5(c) shows how the average number of triangles per edge impacts the speedup of the best combination that uses a directed graph over the best combination that uses an undirected graph. It is clear that when the number of triangles per edge becomes large, undirected graphs perform better. Recall from Section II-B that the advantages of undirected graphs are that they do not need atomic operations to update support values, and that intersection operations can be stopped early as soon as the threshold for an edge not being weak is met. When the number of triangles per edge is high, support values will be updated frequently, intersection operations will be long, and the chances of meeting the threshold early will be high. Hence, avoiding atomic operations and stopping intersections early are features that make undirected graphs attractive.

C. Directing Edges by Degree

Table II shows that the overwhelming majority of cases where a directed graph is preferred, directing edges by degree is the faster option. We observe that the extent to which directing edges by degree is better than by index correlates with the maximum vertex degree in the graph.

Fig. 5(d) shows how the maximum vertex degree impacts the speedup of the best combination that directs edges by degree over the best combination that directs edges by index. It is clear that as the maximum vertex degree increases, the benefit of directing edges by degree increases. Recall from Section II-C that the advantage of directing edges by degree is reducing the size of the adjacency lists for high-degree vertices. Hence, directing the graph by degree is more attractive as the degree of high-degree vertices increases.

D. Tiling

Table II shows that some graphs have better performance when tiling is applied while others are better off without tiling. We observe that the preference for tiling is correlated with the average vertex degree in the graph. Fig. 5(e) shows how the average vertex degree impacts the speedup of the best combination that applies tiling over the best combination that uses does not apply tiling. It is clear that as the average vertex degree increases, the benefit of tiling increases. Recall from Section II-D that one of the advantages of tiling is that it partitions long intersection operations into shorter sub-intersections. As the average vertex degree increases, the intersection operations become longer and partitioning them becomes more important. Hence, tiling is more attractive when the average vertex degree is high.

E. Parallelizing Intersections

Table II shows that some graphs have better performance when individual intersection operations are parallelized while others do not. Fig. 5(f) shows how the size of the graph (number of edges) impacts the speedup of the best combination that parallelizes intersections over the best combination that does not. It is clear that the advantage of parallelizing intersections diminishes for large graphs. Recall from Section II-E that the advantage of parallelizing intersection operations is to extract more parallelism from the computation. For large graphs, there is a sufficient amount of parallelism to fully utilize the device because there are many intersections to perform. Hence, extracting more parallelism becomes less attractive as the graph gets larger.

F. Removing Deleted Edges Intermediately

For $k = 3$, only one iteration is needed for convergence so edges are always removed once at the end. Table II shows that a significant fraction of the computation is spent removing deleted edges at the end. This overhead is particularly high for small graphs, where removing deleted edges at the end accounts for up to 80% of the execution time. We plan to reduce this overhead by further optimizing the stream compaction operation, as well as supporting stream compaction on each vertex's adjacency list separately [3], [4], [5] as opposed to the entire edge list.

For $k = k_{max}$, Table III shows that most graphs have better performance when deleted edges are not removed intermediately, but we expect that reducing the overhead of stream compaction should allow more graphs to show benefit. We observe that the preference for removing deleted edges intermediately is correlated with the number of edges in the graph. Fig. 5(g) shows how the number of edges impacts the speedup of the best combination that removes deleted edges intermediately over the best combination that does not. It is clear that the advantage of removing deleted edges intermediately increases with the number of edges. Recall from Section II-F that the benefit of removing deleted edges is shrinking intersections and reducing the memory footprint, which is more critical as graphs get larger.

G. Recomputing Support Values

For $k = 3$, only one iteration is needed for convergence so recomputing edge support values is not relevant. For $k = k_{max}$, Table III shows that all except a few very large graphs have better performance when the support of all edges is recomputed, not just affected edges. We plan to further optimize the feature of recomputing the support of affected edges. First, we observe that in the initial iterations where many edges are deleted, the number of affected edges is very large. Hence, the overhead of tracking affected edges is not worth the effort it saves. For this reason, we plan to provide the option to track affected edges for only later iterations where the affected edges are few. Second, our current implementation launches threads for all edges and each thread checks if it needs to recount or not. Hence, computational resources are still allocated for threads that do not need to recount and there is high control divergence. Instead, we plan to reduce this divergence by creating a frontier of edges whose threads need to recount and only launching threads for those edges.

V. Conclusion

This paper surveys the optimizations applied to k-truss decomposition on GPUs, and presents KTrussExplorer, a framework for exploring the design space formed by the combinations of these optimizations. The optimizations supported include using a directed graph, directing edges by degree, tiling the adjacency matrix, parallelizing list intersection operations, removing deleted edges from the graph in between iterations, and recomputing support values for only affected edges. Future work includes supporting vertex-centric parallelization, expanding and enhancing the selection of supported optimizations, and leveraging properties of the graph to prune the search space or infer the best combination rather than search the space exhaustively.

Acknowledgments

We would like to thank Amer Mouawad and Jad Ismail for the valuable discussions we had with them. This work is supported by the University Research Board of the American University of Beirut (URB-AUB-103782-25509).

TABLE II: Execution Time (in *milliseconds*) for the Best Configuration for $k = 3$

Graph	Number of Vertices	Number of Edges	Directedness	Tiling (tiling factor)	Parallelizing Intersections (parallelization factor)	Threads per Block	Time (ms)	% Triangle Counting	% Marking Deleted Edges	% Removing Deleted Edges
Theory-3-4-B1k	20	31	undirected	8	8	128	0.191	6.0%	5.0%	77.1%
Theory-3-4-B2k	20	31	undirected	8	2	256	0.183	6.7%	4.5%	78.7%
Theory-4-5-B1k	30	49	undirected	2	no parallelization	128	0.192	6.3%	5.4%	76.7%
Theory-4-5-B2k	30	49	undirected	2	2	128	0.192	6.0%	5.1%	74.4%
Theory-5-9-B1k	60	104	directed by index	8	8	256	0.193	6.9%	5.3%	75.1%
Theory-5-9-B2k	60	104	directed by degree	4	4	1024	0.183	6.0%	4.7%	76.9%
Theory-9-16-B1k	170	313	directed by degree	2	2	128	0.194	6.8%	5.0%	76.6%
Theory-9-16-B2k	170	313	directed by index	4	2	256	0.187	7.0%	4.4%	77.2%
Theory-3-4-5-B1k	120	346	undirected	4	4	256	0.194	6.5%	5.7%	77.0%
Theory-3-4-5-B2k	120	346	directed by degree	2	no parallelization	1024	0.187	7.1%	4.9%	76.3%
Theory-16-25-B1k	442	841	undirected	no tiling	no parallelization	1024	0.194	6.9%	5.4%	73.2%
Theory-16-25-B2k	442	841	directed by degree	2	no parallelization	128	0.188	6.9%	4.9%	76.8%
Theory-4-5-9-B1k	300	940	directed by degree	2	no parallelization	128	0.195	7.5%	4.8%	77.2%
Theory-4-5-9-B2k	300	940	directed by degree	2	2	256	0.191	6.3%	4.7%	73.2%
Theory-5-9-16-B1k	1,020	3,448	directed by degree	no tiling	no parallelization	128	0.206	6.8%	5.7%	74.0%
Theory-5-9-16-B2k	1,020	3,448	directed by degree	no tiling	no parallelization	512	0.202	9.7%	5.3%	71.3%
Theory-25-81-B1k	2,132	4,156	directed by degree	8	no parallelization	512	0.211	7.0%	4.9%	75.8%
Theory-25-81-B2k	2,132	4,156	directed by degree	no tiling	no parallelization	512	0.199	6.7%	4.8%	77.4%
Theory-3-4-5-9-B1k	1,200	6,583	directed by degree	no tiling	no parallelization	128	0.212	7.6%	4.8%	76.1%
Theory-3-4-5-9-B2k	1,200	6,583	directed by degree	16	4	512	0.206	6.0%	3.5%	80.2%
as20000102	6,474	12,572	directed by degree	8	4	512	0.205	7.5%	4.7%	74.4%
ca-GrQc	5,242	14,484	directed by index	8	8	256	0.212	7.7%	4.7%	73.9%
Theory-9-16-25-B1k	4,420	15,988	directed by degree	2	no parallelization	128	0.226	9.1%	4.3%	74.3%
Theory-9-16-25-B2k	4,420	15,988	directed by degree	4	4	512	0.216	7.5%	3.9%	76.9%
p2p-Gnutella08	6,301	20,777	directed by degree	2	2	256	0.208	7.8%	5.8%	72.1%
oregon1_010407	10,729	21,999	directed by degree	8	4	256	0.216	8.9%	4.9%	72.4%
oregon1_010331	10,670	22,002	directed by degree	4	2	512	0.215	8.9%	4.9%	72.9%
oregon1_010414	10,790	22,469	directed by degree	no tiling	no parallelization	256	0.216	9.6%	5.2%	72.9%
oregon1_010428	10,886	22,493	directed by degree	no tiling	no parallelization	1024	0.209	9.9%	4.9%	72.8%
oregon1_010505	10,943	22,607	directed by degree	16	16	256	0.218	8.7%	4.5%	73.6%
oregon1_010512	11,011	22,677	directed by degree	16	16	512	0.217	9.2%	4.6%	72.8%
oregon1_010519	11,051	22,724	directed by degree	2	no parallelization	1024	0.218	9.6%	4.4%	73.0%
oregon1_010421	10,859	22,747	directed by degree	4	4	1024	0.218	10.8%	5.8%	71.5%
oregon1_010526	11,174	23,409	directed by degree	16	16	128	0.219	9.6%	4.4%	73.3%
ca-HepTh	9,877	25,973	directed by index	8	4	256	0.214	8.4%	4.8%	73.2%
p2p-Gnutella09	8,114	26,013	directed by degree	16	2	1024	0.210	8.6%	4.6%	72.5%
oregon2_010407	10,981	30,855	directed by degree	4	no parallelization	512	0.230	12.9%	5.1%	68.8%
oregon2_010505	11,157	30,943	directed by degree	2	no parallelization	512	0.227	11.5%	5.0%	70.8%
Theory-4-5-9-16-B1k	5,100	31,036	directed by degree	2	2	512	0.239	10.1%	4.0%	70.9%
Theory-4-5-9-16-B2k	5,100	31,036	directed by degree	8	2	128	0.226	12.2%	5.2%	70.5%
oregon2_010331	10,900	31,180	directed by degree	2	no parallelization	256	0.224	11.8%	4.3%	70.5%
oregon2_010512	11,260	31,303	directed by degree	4	4	128	0.226	12.8%	4.9%	68.8%
oregon2_010428	11,113	31,434	directed by degree	4	no parallelization	128	0.226	12.1%	4.5%	70.8%
p2p-Gnutella06	8,717	31,525	directed by degree	8	8	128	0.199	7.2%	6.1%	74.9%
oregon2_010421	11,080	31,538	directed by degree	4	2	512	0.216	12.3%	5.0%	70.0%
oregon2_010414	11,019	31,761	directed by degree	2	no parallelization	128	0.214	13.1%	5.5%	67.9%
p2p-Gnutella05	8,846	31,839	directed by degree	4	no parallelization	256	0.218	7.7%	5.0%	70.0%
oregon2_010519	11,375	32,287	directed by degree	no tiling	no parallelization	256	0.235	12.2%	4.9%	70.0%
oregon2_010526	11,461	32,730	directed by degree	8	2	512	0.237	11.6%	5.0%	69.9%
p2p-Gnutella04	10,876	39,994	directed by degree	8	4	256	0.220	7.0%	4.9%	75.6%
Theory-81-256-B1k	21,074	41,809	directed by degree	8	8	512	0.238	9.0%	4.2%	74.7%
Theory-81-256-B2k	21,074	41,809	directed by degree	2	2	1024	0.232	8.5%	4.4%	73.3%
facebook_combined	4,039	44,117	undirected	8	2	512	0.291	22.1%	3.7%	59.8%
as-caida20071105	26,475	53,381	directed by degree	4	no parallelization	256	0.228	11.1%	5.2%	71.1%
p2p-Gnutella25	22,687	54,705	directed by degree	no tiling	no parallelization	256	0.222	8.3%	5.6%	71.8%
p2p-Gnutella24	26,518	65,369	directed by index	16	4	1024	0.222	9.4%	5.2%	73.3%
p2p-Gnutella30	36,682	88,328	directed by index	4	4	512	0.223	7.9%	10.0%	69.3%
ca-CondMat	23,133	93,439	directed by degree	no tiling	no parallelization	512	0.234	12.1%	5.1%	69.9%
ca-HepPh	12,008	118,489	undirected	16	2	512	0.290	23.9%	8.0%	57.7%
Theory-16-25-81-B1k	36,244	137,164	directed by degree	no tiling	no parallelization	512	0.269	16.7%	3.9%	65.5%
Theory-16-25-81-B2k	36,244	137,164	directed by degree	4	2	128	0.255	11.9%	7.1%	66.7%
p2p-Gnutella31	62,586	147,892	directed by degree	2	no parallelization	1024	0.236	7.6%	6.8%	72.2%
Theory-5-9-16-25-B1k	26,520	175,873	directed by degree	2	2	128	0.333	26.8%	4.3%	55.4%
Theory-5-9-16-25-B2k	26,520	175,873	directed by degree	no tiling	no parallelization	256	0.273	18.1%	6.3%	63.5%
email-Enron	36,692	183,831	directed by degree	8	2	256	0.316	20.9%	4.5%	60.5%
ca-AstroPh	18,772	198,050	undirected	2	no parallelization	256	0.327	29.7%	3.8%	55.6%
loc-brightkite_edges	58,228	214,078	directed by degree	4	2	128	0.298	19.1%	6.0%	61.6%
Theory-3-4-5-9-16-B1k	20,400	217,255	directed by degree	2	no parallelization	128	0.389	35.6%	3.3%	48.3%
Theory-3-4-5-9-16-B2k	20,400	217,255	directed by degree	16	4	128	0.293	18.2%	5.8%	59.8%
Theory-256-625-B1k	160,882	320,881	directed by degree	no tiling	no parallelization	256	0.356	21.5%	4.9%	63.2%
Theory-256-625-B2k	160,882	320,881	directed by degree	no tiling	no parallelization	128	0.292	15.7%	8.2%	64.3%
cit-HepTh	27,770	352,285	directed by degree	4	4	128	0.435	31.6%	4.6%	50.6%
email-EuAll	265,214	364,481	directed by degree	no tiling	no parallelization	128	0.348	21.2%	8.3%	59.0%
soc-Epinions1	75,879	405,740	directed by degree	no tiling	no parallelization	256	0.521	41.6%	4.9%	44.8%
cit-HepPh	34,546	420,877	directed by degree	16	8	1024	0.400	31.0%	5.5%	49.3%
soc-Slashdot0811	77,360	469,180	directed by degree	no tiling	no parallelization	256	0.479	35.0%	7.0%	49.0%
soc-Slashdot0902	82,168	504,230	directed by degree	no tiling	no parallelization	128	0.495	39.6%	7.6%	43.6%
amazon0302	262,111	899,792	directed by degree	no tiling	no parallelization	256	0.436	25.8%	13.2%	51.3%
loc-gowalla_edges	196,591	950,327	directed by degree	no tiling	no parallelization	512	0.661	44.7%	8.5%	38.9%
flickrEdges	105,938	1,158,474	undirected	16	no parallelization	128	3.218	75.9%	6.0%	16.2%
roadNet-PA	1,088,092	1,541,898	directed by degree	no tiling	no parallelization	512	0.454	19.9%	18.6%	51.7%
Theory-4-5-9-16-25-B1k	132,600	1,582,861	directed by degree	no tiling	no parallelization	512	1.993	75.8%	1.5%	19.9%
Theory-4-5-9-16-25-B2k	132,600	1,582,861	directed by degree	2	no parallelization	256	0.886	53.9%	10.2%	28.3%
roadNet-TX	1,379,917	1,921,660	directed by degree	no tiling	no parallelization	256	0.532	21.3%	20.2%	49.4%
Theory-25-81-256-B1k	547,924	2,132,284	directed by degree	no tiling	no parallelization	128	1.833	69.6%	3.8%	23.2%
Theory-25-81-256-B2k	547,924	2,132,284	directed by degree	2	2	128	0.760	37.3%	15.8%	38.7%
amazon0312	400,727	2,349,869	directed by degree	4	2	512	1.053	42.7%	11.0%	40.3%
amazon0505	410,236	2,439,437	directed by degree	4	2	128	1.092	50.2%	11.8%	32.5%
amazon0601	403,394	2,443,408	directed by degree	4	2	512	1.218	47.6%	11.6%	34.2%
Theory-9-16-25-81-B1k	362,440	2,606,125	directed by degree	no tiling	no parallelization	128	3.192	78.7%	2.1%	17.1%
Theory-9-16-25-81-B2k	362,440	2,606,125	directed by degree	no tiling	no parallelization	128	1.229	58.1%	12.5%	24.1%
roadNet-CA	1,965,206	2,766,607	directed by index	no tiling	no parallelization	512	0.658	21.2%	25.9%	45.3%
graph500-scale18-ef16	174,147	3,800,348	undirected	16	no parallelization	128	11.745	89.1%	2.9%	7.2%
graph500-scale19-ef16	335,318	7,729,675	undirected	32	no parallelization	128	27.188	91.3%	2.6%	5.8%
Theory-3-4-5-9-16-25-B1k	530,400	11,080,030	directed by degree	no tiling	no parallelization	1024	27.837	91.7%	0.5%	7.6%
Theory-3-4-5-9-16-25-B2k	530,400	11,080,030	directed by degree	no tiling	no parallelization	1024	7.606	82.9%	8.7%	7.3%
graph500-scale20-ef16	645,820	15,680,861	undirected	32	no parallelization	128	72.228	93.5%	2.0%	4.3%
cit-Patents	3,774,768	16,518,947	directed by index	8	4	256	7.866	67.7%	12.4%	19.1%
MAWI Graph 1	18,571,154	19,020,160	directed by degree	no tiling	no parallelization	512	2.779	24.7%	37.0%	36.1%
Theory-5-9-16-25-81-B1k	2,174,640	28,667,380	directed by degree	2	2	128	104.308	93.3%	0.5%	6.1%
Theory-5-9-16-25-81-B2k	2,174,640	28,667,380	directed by degree	8	no parallelization	1024	26.609	89.4%	5.7%	4.6%
graph500-scale21-ef16	1,243,072	31,731,650	undirected	32	no parallelization	128	183.804	95.4%	1.4%	3.2%
MAWI Graph 2	35,991,342	37,242,710	directed by degree	no tiling	no parallelization	512	5.035	25.2%	38.6%	35.0%
Graph 5 - V2a	55,042,369	58,608,800	directed by degree	no tiling	no parallelization	256	11.543	50.9%	25.3%	23.1%
graph500-scale22-ef16	2,393,285	64,097,004	undirected	32	no parallelization	128	484.921	96.4%	1.1%	2.4%
Graph 3 - U1a	67,716,231	69,389,281	directed by degree	no tiling	no parallelization	256	13.082	48.7%	26.3%	24.2%
MAWI Graph 3	68,863,315	71,707,480	directed by degree	no tiling	no parallelization	256	9.456	27.1%	37.6%	34.3%
graph500-scale23-ef16	4,606,314	129,250,705	undirected	32	no parallelization	128	1,273.281	97.2%	0.8%	2.0%
MAWI Graph 4	128,568,730	135,117,420	directed by degree	no tiling	no parallelization	256	17.669	28.1%	37.8%	33.6%
Graph 2 - P1a	139,353,211	148,914,992	directed by degree	no tiling	no parallelization	256	29.643	40.4%	16.0%	43.4%
Graph 1 - A2a	170,728,175	180,292,586	directed by degree	no tiling	no parallelization	256	35.029	51.7%	25.5%	22.4%
Graph 4 - V1r	214,005,017	232,705,452	directed by degree	no tiling	no parallelization	256	41.740	48.3%	27.7%	23.8%
MAWI Graph 5	226,196,185	240,023,947	directed by degree	no tiling	no parallelization	256	34.052	29.0%	35.2%	35.6%

TABLE III: Execution Time (in *milliseconds*) for the Best Configuration for $k = k_{max}$

Graph	Number of Vertices	Number of Edges	k_{max}	Number of Iterations	Directedness	Tiling (tiling factor)	Parallelizing Intersections (parallelization factor)	Removing Edges Intermediately	Recomputing Support Values	Threads per Block	Time (ms)	% Triangle Counting	% Marking Deleted Edges	% Removing Deleted Edges
Theory-3-4-5-B1k	120	346	6	3	directed by degree	2	no parallelization	no iterations	all edges	256	0.239	12.5%	9.7%	48.0%
Theory-3-4-5-B2k	120	346	4	2	directed by degree	8	8	no iterations	all edges	512	0.218	11.2%	8.6%	60.1%
Theory-4-5-9-B1k	300	940	7	3	directed by degree	2	2	no iterations	all edges	1024	0.234	15.6%	11.1%	52.1%
Theory-4-5-9-B2k	300	940	4	2	directed by degree	no tiling	no parallelization	no iterations	all edges	256	0.220	11.5%	8.8%	62.7%
Theory-5-9-16-B1k	1,020	3,448	8	3	directed by degree	16	8	no iterations	all edges	256	0.279	14.5%	9.9%	52.4%
Theory-5-9-16-B2k	1,020	3,448	4	2	directed by degree	no tiling	no parallelization	no iterations	all edges	256	0.233	12.6%	8.3%	60.6%
Theory-3-4-5-9-B1k	1,200	6,583	10	3	directed by degree	4	no parallelization	no iterations	all edges	256	0.263	16.9%	9.8%	50.6%
Theory-3-4-5-9-B2k	1,200	6,583	5	3	directed by degree	16	2	no iterations	all edges	1024	0.272	15.4%	11.0%	49.7%
as20000102	6,474	12,572	10	4	directed by degree	8	2	no iterations	all edges	1024	0.335	17.9%	12.1%	46.0%
ca-GrQc	5,242	14,484	44	3	directed by degree	2	no parallelization	no iterations	all edges	256	0.345	21.7%	10.3%	48.7%
Theory-9-16-25-B1k	4,420	15,988	12	3	directed by degree	4	no parallelization	no iterations	all edges	128	0.312	16.9%	9.7%	52.1%
Theory-9-16-25-B2k	4,420	15,988	4	2	directed by degree	4	no parallelization	no iterations	all edges	512	0.257	12.5%	7.5%	63.2%
p2p-Gnutella08	6,301	20,777	5	7	directed by degree	8	8	no iterations	all edges	128	0.438	20.6%	14.1%	34.2%
oregon1_010407	10,729	21,999	14	5	directed by degree	8	4	no iterations	all edges	1024	0.398	20.8%	12.9%	40.3%
oregon1_010331	10,670	22,002	16	8	directed by degree	8	no parallelization	no iterations	all edges	512	0.573	27.0%	14.3%	28.1%
oregon1_010414	10,790	22,469	15	5	directed by degree	4	4	no iterations	all edges	1024	0.422	22.0%	13.5%	38.6%
oregon1_010428	10,886	22,493	15	6	directed by degree	4	2	no iterations	all edges	1024	0.445	21.3%	13.6%	36.0%
oregon1_010505	10,943	22,607	14	8	directed by degree	16	4	no iterations	all edges	1024	0.550	23.9%	14.6%	32.1%
oregon1_010512	11,011	22,677	15	5	directed by degree	2	no parallelization	no iterations	all edges	256	0.404	21.3%	15.7%	38.3%
oregon1_010519	11,051	22,724	15	7	directed by degree	no tiling	no parallelization	no iterations	all edges	512	0.514	23.9%	15.0%	31.2%
oregon1_010421	10,859	22,747	15	6	directed by degree	no tiling	no parallelization	no iterations	all edges	256	0.461	24.3%	13.6%	35.6%
oregon1_010526	11,174	23,409	14	7	directed by degree	2	no parallelization	no iterations	all edges	1024	0.502	28.1%	14.0%	30.0%
ca-HepTh	9,877	25,973	32	3	directed by degree	8	4	no iterations	all edges	128	0.336	18.0%	9.9%	50.3%
p2p-Gnutella09	8,114	26,013	5	6	directed by degree	4	no parallelization	no iterations	all edges	1024	0.409	19.8%	14.3%	36.6%
oregon2_010407	10,981	30,855	24	8	directed by degree	8	2	no iterations	all edges	128	0.605	34.6%	13.1%	25.7%
oregon2_010505	11,157	30,943	21	10	directed by degree	4	no parallelization	no iterations	all edges	256	0.700	31.8%	14.1%	22.5%
Theory-4-5-9-16-B1k	5,100	31,036	12	3	directed by degree	no tiling	no parallelization	no iterations	all edges	512	0.347	22.1%	9.2%	45.4%
Theory-4-5-9-16-B2k	5,100	31,036	5	3	directed by degree	4	4	no iterations	all edges	512	0.308	18.7%	10.3%	49.6%
oregon2_010331	10,900	31,180	25	12	directed by degree	16	2	no iterations	all edges	512	0.833	34.3%	15.3%	21.2%
oregon2_010512	11,260	31,303	21	12	directed by degree	no tiling	no parallelization	no iterations	all edges	128	0.788	36.3%	14.6%	18.1%
oregon2_010428	11,113	31,434	21	11	directed by degree	4	no parallelization	no iterations	all edges	512	0.762	33.7%	14.0%	19.3%
p2p-Gnutella06	8,717	31,525	4	5	directed by degree	16	8	no iterations	all edges	1024	0.377	17.6%	12.2%	39.6%
oregon2_010421	11,080	31,538	22	11	directed by degree	no tiling	no parallelization	no iterations	all edges	512	0.732	35.1%	15.3%	21.9%
oregon2_010414	11,019	31,761	24	11	directed by degree	2	no parallelization	no iterations	all edges	256	0.731	36.9%	14.7%	20.2%
p2p-Gnutella05	8,846	31,839	4	5	directed by degree	no tiling	no parallelization	no iterations	all edges	512	0.343	17.7%	13.6%	43.1%
oregon2_010519	11,375	32,287	24	7	directed by degree	4	no parallelization	no iterations	all edges	256	0.546	30.6%	13.5%	29.2%
oregon2_010526	11,461	32,730	25	7	directed by degree	no tiling	no parallelization	no iterations	all edges	512	0.516	29.9%	13.1%	31.8%
p2p-Gnutella04	10,876	39,994	4	3	directed by degree	no tiling	no parallelization	no iterations	all edges	128	0.271	14.6%	11.2%	53.2%
facebook_combined	4,039	44,117	97	13	directed by degree	no tiling	no parallelization	first 1 iterations	all edges	256	1.595	50.5%	8.7%	23.6%
as-caida20071105	26,475	53,381	16	9	directed by degree	2	2	no iterations	all edges	512	0.598	28.6%	13.9%	25.1%
p2p-Gnutella25	22,687	54,705	4	4	directed by degree	no tiling	no parallelization	no iterations	all edges	1024	0.323	15.8%	14.1%	44.8%
p2p-Gnutella24	26,518	65,369	4	4	directed by degree	2	no parallelization	no iterations	all edges	512	0.330	16.2%	11.8%	46.7%
p2p-Gnutella30	36,682	88,328	4	3	directed by degree	2	no parallelization	no iterations	all edges	512	0.259	14.3%	13.6%	48.7%
ca-CondMat	23,133	93,439	26	3	directed by degree	2	2	no iterations	all edges	1024	0.345	19.0%	10.4%	50.5%
ca-HepPh	12,008	118,489	239	3	directed by degree	8	4	all 3 iteration	all edges	128	1.081	49.5%	4.1%	37.5%
Theory-16-25-81-B1k	36,244	137,164	19	3	directed by degree	no tiling	no parallelization	no iterations	all edges	512	0.401	28.0%	9.9%	42.4%
Theory-16-25-81-B2k	36,244	137,164	4	2	directed by degree	2	no parallelization	all 2 iteration	all edges	128	0.301	14.2%	8.6%	57.4%
p2p-Gnutella31	62,586	147,892	4	3	directed by degree	2	2	no iterations	all edges	512	0.358	15.2%	11.3%	51.6%
Theory-5-9-16-25-B1k	26,520	175,873	19	3	directed by degree	4	2	no iterations	all edges	128	0.429	33.3%	9.3%	40.4%
Theory-5-9-16-25-B2k	26,520	175,873	5	3	directed by degree	2	2	no iterations	all edges	256	0.381	22.8%	10.9%	46.3%
email-Enron	36,692	183,831	22	20	directed by degree	2	2	no iterations	all edges	256	1.759	50.2%	14.0%	11.6%
ca-AstroPh	18,772	198,050	57	4	directed by degree	no tiling	no parallelization	no iterations	all edges	256	0.564	33.2%	10.0%	37.5%
loc-brightkite_edges	58,228	214,078	43	10	directed by degree	4	2	no iterations	all edges	256	0.909	42.7%	12.5%	20.2%
Theory-3-4-5-9-16-B1k	20,400	217,255	19	4	directed by degree	4	no parallelization	no iterations	all edges	128	0.635	41.2%	9.8%	31.4%
Theory-3-4-5-9-16-B2k	20,400	217,255	6	3	directed by degree	no tiling	no parallelization	no iterations	all edges	256	0.418	29.2%	10.3%	42.6%
cit-HepTh	27,770	352,285	30	12	directed by degree	no tiling	no parallelization	no iterations	all edges	1024	1.104	41.3%	13.7%	20.2%
email-EuAll	265,214	364,481	20	14	directed by degree	2	2	no iterations	all edges	1024	1.675	53.6%	12.1%	14.7%
soc-Epinions1	75,879	405,740	33	26	directed by degree	no tiling	no parallelization	first 2 iterations	all edges	1024	2.995	49.7%	10.6%	19.7%
cit-HepPh	34,546	420,877	25	8	directed by degree	16	4	no iterations	all edges	512	0.991	40.4%	13.8%	24.6%
soc-Slashdot0811	77,360	469,180	35	23	directed by degree	4	4	first 1 iterations	all edges	256	2.241	49.7%	11.6%	16.4%
soc-Slashdot0902	82,168	504,230	36	16	directed by degree	no tiling	no parallelization	first 1 iterations	all edges	512	1.769	47.6%	10.7%	20.8%
amazon0302	262,111	899,792	7	4	directed by degree	no tiling	no parallelization	no iterations	all edges	512	0.652	28.6%	18.4%	35.4%
loc-gowalla_edges	196,591	950,327	29	10	directed by degree	2	no parallelization	no iterations	all edges	512	1.371	46.4%	13.3%	14.6%
flickrEdges	105,938	1,158,474	574	4	directed by degree	2	no parallelization	no iterations	all edges	128	18.554	93.0%	2.2%	2.9%
roadNet-PA	1,088,092	1,541,898	4	4	directed by degree	2	no parallelization	no iterations	all edges	256	0.682	21.8%	19.6%	41.8%
Theory-4-5-9-16-25-B1k	132,600	1,582,861	28	4	directed by degree	2	2	first 1 iterations	all edges	512	2.492	72.3%	5.5%	17.1%
Theory-4-5-9-16-25-B2k	132,600	1,582,861	6	3	directed by degree	4	no parallelization	no iterations	all edges	256	1.117	55.6%	13.2%	21.6%
roadNet-TX	1,379,917	1,921,660	4	4	directed by degree	no tiling	no parallelization	no iterations	all edges	1024	0.776	27.6%	23.6%	32.7%
Theory-25-81-256-B1k	547,924	2,132,284	28	3	directed by degree	2	no parallelization	no iterations	all edges	128	2.157	72.7%	9.8%	11.9%
Theory-25-81-256-B2k	547,924	2,132,284	4	2	directed by degree	2	2	all 2 iteration	all edges	128	0.836	39.5%	17.3%	33.1%
amazon0312	400,727	2,349,869	11	6	directed by degree	2	no parallelization	first 1 iterations	all edges	512	1.572	41.5%	14.3%	31.5%
amazon0505	410,236	2,439,437	11	6	directed by degree	4	no parallelization	no iterations	all edges	256	1.629	51.1%	21.6%	16.8%
amazon0601	403,394	2,443,408	11	6	directed by degree	4	2	first 1 iterations	all edges	128	1.643	42.4%	16.4%	29.7%
Theory-9-16-25-81-B1k	362,440	2,606,125	28	3	directed by degree	no tiling	no parallelization	all 3 iteration	all edges	128	3.267	77.4%	5.4%	13.7%
Theory-9-16-25-81-B2k	362,440	2,606,125	5	3	directed by degree	2	no parallelization	no iterations	all edges	256	1.480	60.6%	14.3%	17.7%
roadNet-CA	1,965,206	2,766,607	4	4	directed by degree	no tiling	no parallelization	no iterations	all edges	512	1.001	30.9%	25.7%	30.3%
graph500-scale18-ef16	174,147	3,800,348	159	51	directed by degree	16	no parallelization	first 1 iterations	all edges	512	71.229	95.5%	1.5%	0.9%
graph500-scale19-ef16	335,318	7,729,675	213	29	directed by degree	16	4	all 29 iteration	all edges	1024	110.339	90.3%	0.7%	8.2%
Theory-3-4-5-9-16-25-B1k	530,400	11,080,030	62	4	directed by degree	2	no parallelization	all 4 iteration	all edges	1024	26.377	93.4%	2.5%	3.5%
Theory-3-4-5-9-16-25-B2k	530,400	11,080,030	7	3	directed by degree	no tiling	no parallelization	first 1 iterations	all edges	512	7.795	81.0%	8.4%	9.0%
graph500-scale20-ef16	645,820	15,680,861	284	134	directed by degree	8	4	first 8 iterations	affected edges	1024	372.465	96.7%	0.7%	0.5%
cit-Patents	3,774,768	16,518,947	36	22	directed by degree	16	4	first 1 iterations	all edges	512	13.136	66.4%	8.5%	20.7%
Theory-5-9-16-25-81-B1k	2,174,640	28,667,380	84	4	directed by degree	no tiling	no parallelization	all 4 iteration	all edges	128	83.136	96.4%	1.8%	1.7%
Theory-5-9-16-25-81-B2k	2,174,640	28,667,380	6	3	directed by degree	no tiling	no parallelization	all 3 iteration	all edges	256	25.825	89.4%	5.7%	4.5%
graph500-scale21-ef16	1,243,072	31,731,650	373	25	undirected	32	no parallelization	first 2 iterations	affected edges	256	2,997.922	99.5%	0.3%	0.2%
graph500-scale22-ef16	2,393,285	64,097,004	485	28	undirected	32	no parallelization	first 4 iterations	all edges	256	9,208.547	99.8%	0.1%	0.1%
graph500-scale23-ef16	4,606,314	129,250,705	625	74	undirected	32	no parallelization	first 2 iterations	affected edges	256	27,002.074	99.8%	0.1%	0.1%

REFERENCES

[1] M. Almasri, O. Anjum, C. Pearson, Z. Qureshi, V. S. Mailthody, R. Nagi, J. Xiong, and W.-m. Hwu, "Update on k-truss decomposition on gpu," in *2019 IEEE High Performance Extreme Computing Conference*. IEEE, 2019.

[2] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," in *GPU computing gems Jade edition*. Elsevier, 2012.

[3] M. Bisson and M. Fatica, "Static graph challenge on gpu," in *2017 IEEE High Performance Extreme Computing Conference*. IEEE, 2017.

[4] ——, "Update on static graph challenge on gpu," in *2018 IEEE High Performance extreme Computing Conference*. IEEE, 2018.

[5] M. Blanco, T. M. Low, and K. Kim, "Exploration of fine-grained parallelism for load balancing eager k-truss on gpu and cpu," in *2019 IEEE High Performance Extreme Computing Conference*. IEEE, 2019.

[6] K. Date, K. Feng, R. Nagi, J. Xiong, N. S. Kim, and W.-M. Hwu, "Collaborative (cpu+ gpu) algorithms for triangle counting and truss decomposition on the minsky architecture: Static graph challenge: Subgraph isomorphism," in *2017 IEEE High Performance Extreme Computing Conference*. IEEE, 2017.

[7] J. Fox, O. Green, K. Gabert, X. An, and D. A. Bader, "Fast and adaptive list intersections on the gpu," in *2018 IEEE High Performance extreme Computing Conference*. IEEE, 2018.

[8] O. Green, J. Fox, E. Kim, F. Busato, N. Bombieri, K. Lakhotia, S. Zhou, S. Singapura, H. Zeng, R. Kannan *et al.*, "Quickly finding a truss in a haystack," in *2017 IEEE High Performance Extreme Computing Conference*. IEEE, 2017.

[9] Y. Hu, P. Kumar, G. Swope, and H. H. Huang, "Trix: Triangle counting at extreme scale," in *2017 IEEE High Performance Extreme Computing Conference*. IEEE, 2017.

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

[11] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song *et al.*, "Static graph challenge: Subgraph isomorphism," in *2017 IEEE High Performance Extreme Computing Conference*. IEEE, 2017.

[12] C. Voegele, Y.-S. 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*. IEEE, 2017.

[13] A. Yasar, S. Rajamanickam, J. Berry, M. Wolf, J. Young, and U. V. Catalyürek, "Linear algebra-based triangle counting via fine-grained tasking on heterogeneous environments," in *IEEE*, 2019.