Parallel Triangle Counting and k-Truss Identification using Graph-centric Methods

Chad Voegele*, Yi-Shan Lu*, Sreepathi Pai† and Keshav Pingali
The University of Texas at Austin
chad@cs.utexas.edu, yishanlu@utexas.edu, sreepai@ices.utexas.edu, pingali@cs.utexas.edu

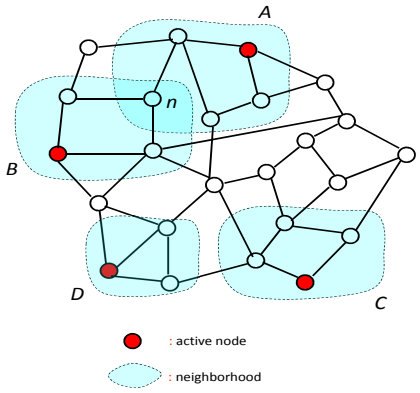

Fig. 1: Operator formulation [10].

***Abstract*—We describe CPU and GPU implementations of parallel triangle-counting and k-truss identification in the Galois and IrGL systems. Both systems are based on a graph-centric abstraction called the operator formulation of algorithms. Depending on the input graph, our implementations are two to three orders of magnitude faster than the reference implementations provided by the IEEE HPEC static graph challenge.**

I. Introduction

This paper describes high-performance CPU and GPU implementations of triangle counting and k-truss identification in graphs. We use a graph-centric programming model called the *operator formulation of algorithms* [10], which has been implemented for CPUs in the Galois system [8] and for GPUs in the IrGL system [9].

A. Operator formulation of algorithms

The operator formulation is a *data-centric* abstraction which presents a *local view* and a *global view* of algorithms, shown pictorially in Fig. 1.

The local view is described by an *operator*, which is a graph update rule applied to an *active node* in the graph (some algorithms have active edges). Each operator application, called an *activity* or *action*, reads and writes a small region of the graph around the active node, called the *neighborhood* of that activity. Fig. 1 shows active nodes as filled dots, and neighborhoods as clouds surrounding active nodes, for a generic algorithm. An active node becomes inactive once the activity is completed. In general, operators can modify the graph structure of the neighborhood by adding and removing nodes and edges (these are called *morph* operators). In most graph analytic applications, operators only update labels on nodes and edges, without changing the graph structure. These are called *label computation* operators.

The global view of a graph algorithm is captured by the location of active nodes and the order in which activities must appear to be performed. Topology-driven algorithms make a number of sweeps over the graph until some convergence criterion is met, e.g., the Bellman-Ford SSSP algorithm. Data-driven algorithms begin with an initial set of active nodes, and other nodes may become active on the fly when activities are executed. They terminate when there are no more active nodes. Dijkstra's SSSP algorithm is a data-driven algorithm. The second dimension of the global view of algorithms is *ordering* [4]. Most graph analytic algorithms are *unordered* algorithms in which activities can be performed in any order without violating program semantics, although some orders may be more efficient than others.

Parallelism can be exploited by processing active nodes in parallel, subject to neighborhood and ordering constraints. The resulting parallelism is called *amorphous* data-parallelism, and it is a generalization of the standard notion of data-parallelism [10].

B. Galois and IrGL systems

The Galois system is an implementation of this data-centric programming model[1]. Application programmers write programs in sequential C++, using certain programming patterns to highlight opportunities for exploiting amorphous data-parallelism. The Galois system provides a library of concurrent data structures, such as parallel graph and work-list implementations, and a runtime system; the data structures and runtime system ensure that each activity appears to execute atomically. The Galois system has been used to implement parallel programs for many problem domains including finite-element simulations, n-body methods, graph analytics, intrusion detection in networks and FPGA tools [6]. The IrGL compiler translates Galois programs into CUDA code, applying a number of GPU-specific optimizations while lowering code to CUDA [9].

*Contributed equally to this paper
†Now at the University of Rochester, sree@cs.rochester.edu
‡ Research supported by NSF grants 1337281, 1406355, and 1618425, and by DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563.

[1]A more detailed description of the implementation of the Galois system can be found in our previous papers such as [8].

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

In the implementations of triangle-counting and k-truss detection described in this paper, we assume that input graphs are symmetric, have no self-loops and have no duplicated edges. We represent input graphs in compressed sparse row (CSR) format which uses two arrays – one for adjacency lists and another to index into the adjacency list array by node. Instead of removing edges physically, we track edge removals in a separate boolean array. For k-truss, arrays also track node removals and effective degree as edges are removed.

Shao et al. [14] also use a graph-centric approach for k-truss identification in a distributed memory setting. They partition a given graph among hosts with each host responsible for its partition. Their focus is on how to exchange edge removals among hosts efficiently.

C. Algorithms based on linear algebra primitives

Graph algorithms can also be formulated in terms of linear algebra primitives [12]. The basic idea is to represent graphs using their incidence or adjacency matrices, and formulate algorithms using bulk-style operations like sparse matrix-vector or matrix-matrix multiplication. For example, topology-driven/data-driven vertex programs [6] can be formulated using the product of a sparse-matrix and a dense/sparse vector respectively, where the vector represents the labels of active nodes in a given round.

Triangles can be counted in a graph by using an overloaded matrix-matrix multiplication on adjacency and incidence matrices for the graph, as in miniTri [15]. Regular and Hadamard matrix-matrix multiplication are also used to count triangles [2]. A k-truss identification algorithm using regular matrix-matrix multiplication and other matrix operations is demonstrated in Samsi et al. [12].

While vertex programs can be formulated naturally in terms of matrix operations, it is non-trivial to formulate more complex graph algorithms such as triangle-counting and k-truss detection in terms of matrix operations. In addition, our graph-centric implementations rely on certain key optimization such as sorting of edge-lists, early termination of operators, and symmetry-breaking to avoid excess work, as described in later sections. These are difficult to implement in matrix-based formulations, leading to implementations that are orders of magnitude slower than ours.

II. Triangle Counting

Triangle counting can be performed by iterating over the edges of the graph, and for each edge (u, v), checking if nodes u and v have a common neighbor w; if so, nodes u, v, w form a triangle. The common neighbors of nodes u and v can be determined by intersecting the edge lists of u and v. Finding the intersection of sets of size p and q can take time $O(p*q)$, but if the sets are sorted, the intersection can be done in time $O(p+q)$ [13]. To avoid repeated counting of triangles, we can increment the count only for an edge (u, v) and a common neighbor w of u and v where $u<w<v$. Work can be further reduced by *symmetry breaking*: triangles are counted using only those edges (u, v) where the degree of u is lower than the degree of v.

Algorithm 1 Edge list Intersection

Input: U, V: sorted edge lists for nodes u and v
Output: Count of nodes appearing in $U \cap V$

```
1:  procedure INTERSECT(U, V)
2:      i ← 0; j ← 0
3:      while i < |U| and j < |V| do
4:          d ← U[i] − V[j]
5:          if d = 0 then
6:              count++; i++; j++
7:          else if d < 0 then
8:              i++
9:          else if d > 0 then
10:             j++
11:         end if
12:     end while
13:     return count
14: end procedure
```

In terms of the operator formulation, this approach to triangle counting is a topology-driven algorithm in which the active elements are edges. The operator implements edge list intersection.

In a parallel implementation, edges are partitioned between threads. Each thread keeps a local count of triangles for the edges it is responsible for, and these local counts are added at the end.

A. CPU Implementation

Our CPU implementation uses the triangle counting from Galois Lonestar [5]. First, threads cooperatively create a work-list that contains all edges (u, v), where $u<v$.

Threads then claim work from the work list, preferring work generated by themselves. Edge list intersection terminates as soon as one of the two edge lists reaches its end. This enables early termination for the edge operator. In contrast, triangle counting using matrix algebra needs to multiply matrices in full [12], which can be inefficient.

B. GPU Implementation

GPU triangle counting implements the approach from Polak [11] in IrGL [9]. First, a filtering step removes edges that point from nodes of a higher degree to those of lower degree, breaking ties by using node identifiers. The remaining edges are the active edges. Then, an efficient segmented sort from ModernGPU [1] is used to sort the edge lists of each node. Finally, the edge lists of edges remaining from the first step are intersected to determine the count of triangles.

To avoid the use of a separate work-list of edges, the IrGL implementation sorts edges so that active edges precede inactive edges in the edge lists of each node. The computation is then initially parallelized over the nodes, and IrGL's nested parallelism optimization is used to dynamically parallelize execution over edges at runtime.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

III. K-TRUSS COMPUTATION

Our *DirectTruss* algorithm works in rounds. In each round, we compute the number of triangles that an edge e participates in, which we term as the support of that edge e. If the support of e is less than $k-2$, it cannot be part of the k-truss and is removed from the graph. Removing e necessitates recomputing the support of other edges that may have participated in triangles containing e. The algorithm terminates when no edges are removed in a round.

Unlike triangle counting, where symmetry permits only one edge of a triangle to be processed, k-truss identification requires that support be computed for *all* edges that may be part of the same triangle. Counting the support of only one edge would not reveal the support of the other edges of the triangle since they could be part of other triangles. However, although edge list intersection is used for computing the support for an edge, k-truss does not really require the exact count of triangles on an edge—it only needs to know if there are at least $k-2$ triangles containing that edge. Thus, intersection can be terminated as soon as this is determined.

Work can also potentially be reduced by using an observation from Cohen [3]: a k-truss is always a $(k-1)$-core which is a graph where each node has at least $k-1$ neighbors. Computing the $(k-1)$-core can eliminate a large number of nodes and the edges connected to them from consideration, reducing the number of edge list intersections. Computing the k-truss on the resultant graph may be potentially faster. We call this *CoreThenTruss* algorithm. To compute a $(k-1)$-core, we use the *DirectCore* algorithm that removes all nodes v if $deg(v) < k-1$ iteratively in rounds. The *DirectCore* algorithm terminates when no nodes are removed in a round.

Algorithm 2 summarizes the above algorithms. Since both *DirectTruss* and *CoreThenTruss* algorithms need edge list intersection to compute edge support, we sort edge lists for all nodes before actual k-truss computation. We use an array of size $|E|$ to track if an edge is removed.

A. CPU Implementation

We implement both *DirectTruss* and *CoreThenTruss* algorithms in Galois [10]. Since *CoreThenTruss* algorithm is built from *DirectTruss* and *DirectCore* algorithms, we will present the latter two in operator formulation. The node removals in line 16 of Algorithm 2 are skipped, because they are done through removing all their edges by the edge operator shown in line 9 to 13 in Algorithm 2. We report the resulting k-truss edge by edge and keep track of involved nodes during the process, so correctness remains unaffected. For better performance, we consider only edges (u, v), where $u<v$, to halve the work. In this case, removal of edge (u, v) will remove both (u, v) and (v, u).

We reason about correctness of *DirectTruss* parallelization as follows. Consistency is preserved: an edge (u, v), where $u<v$, can only remove (u, v) and (v, u), and the barrier between rounds ensures that edge removals in round r are visible before round $r+1$ begins. Termination upon no edge removal in a round is guaranteed: Since removed edges are never added back to the graph, the remaining edges' supports will never increase as the rounds progress. When *DirectTruss* terminates, each remaining edge has its support $\geq k-2$. Hence, *DirectTruss* computes a k-truss for the graph correctly.

The *DirectCore* algorithm also maps well to the operator formulation. A node operator is indicated by line 24 to 28 in Algorithm 2 to track degree and node removal. Node v removes itself by removing edges (v, n) and (n, v) for each v's neighbor n. The degree check for v can stop once we know that $deg(v) \geq k$ when computing for k-core. This enables early termination of the node operator.

The correctness of our *DirectCore* parallelization can be argued similarly to that for *DirectTruss*. There are only two

Algorithm 2 K-Truss Computation

Input: $G = (V, E)$, an undirected graph; k, the truss number to consider.
Output: All edges belonging to k-truss of G.

```
1:  procedure IsEdgeSupportGeqK(E, e, k)
2:      return |{v|(e.src, v) ∈ E ∧ (e.dst, v) ∈ E}| ≥ k
3:  end procedure
4:  procedure DirectTruss(G, k)
5:      W_next ← E; W_current ← ∅
6:      while W_current ≠ W_next do
7:          W_current ← W_next; W_next ← ∅
8:          for all e ∈ W_current do
9:              if IsEdgeSupportGeqK(E,e,k−2) then
10:                 W_next ← W_next ∪ {e}
11:             else
12:                 E ← E − {e}
13:             end if
14:         end for
15:     end while
16:     V ← {v|v ∈ V ∧ deg(v) > 0}
17:     return G
18: end procedure
19: procedure DirectCore(G, k)
20:     W_next ← V; W_current ← ∅
21:     while W_current ≠ W_next do
22:         W_current ← W_next; W_next ← ∅
23:         for all v ∈ W_current do
24:             if deg(v) < k then
25:                 V ← V − {v}
26:             else
27:                 W_next ← W_next ∪ {v}
28:             end if
29:         end for
30:     end while
31:     return G
32: end procedure
33: procedure CoreThenTruss(G, k)
34:     G′ ← DirectCore(G, k−1)
35:     return DirectTruss(G′,k)
36: end procedure
```

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

differences. First, the node operator applied on node v checks for $deg(v) \geq k$ in k-core computation. Second, if neighboring nodes v and n both get removed in a round, they can mark edges (v, n) and (n, v) as removed concurrently, since an edge is removed no matter how many times it is marked.

Our implementations work as in Gauss–Seidel iterative algorithms. If an edge is removed once the edge or one of its endpoints deems so, the other nodes or edges may see the edge removal in the same round. Therefore, other edge removals may happen earlier, which speeds up the convergence of both *DirectTruss* and *DirectCore* algorithms. Matrix-based approaches, on the other hand, usually perform edge removals separately [12], as in Jacobi iterative algorithms.

B. GPU Implementation

We implement the iterative *CoreThenTruss* algorithm on GPU making several modifications to our approach from triangle counting to improve performance.

First, we choose to work directly on edges, instead of on nodes. This flattens the parallelism completely with the cost amortized over multiple iterations. A separate array tracks the degree of each node. This is decremented every time a node's edge is removed for lack of support. Another array tracks if an edge is valid which is used to ignore edges when computing the intersection of edge lists.

Valid edges are tracked at all times on an IrGL worklist. Our GPU implementation begins by iteratively removing all edges whose end points have a degree less than $k-1$ from the worklist. It then computes the support of remaining edges, removing edges that lack support immediately.

However, unlike the CPU, we interleave computing the support of each edge with removing edges whose end points have a degree less than $k-1$. Since removing edges by examining their end points is cheaper than removing edges by computing support, this interleaving strategy may be faster.

IV. Results

We use the GraphChallenge input graphs from SNAP [7] as well as the synthetic datasets based on Graph500. We augment this dataset with very large "community" datasets [7]. Apart from three road networks, all inputs are power-law graphs (Table I). Our GPU experiments used a Pascal-based NVIDIA GTX 1080 with 8GB of memory while our CPU experiments used a Broadwell-EP Xeon E5-2650 v4 running at 2.2GHz with a 30MB LLC and 192GB RAM. Our machine contains two processors with 12 cores each, therefore we present results for 1, 12 and 24 threads.

GPU code was compiled using NVCC 8.0. CPU code used GCC 4.9. The serial baseline for triangle counting is mini-Tri [15] implemented in C++. We compare to the reference serial Julia implementation of k-truss run using Julia 0.60.[2]

CPU Energy statistics are gathered using the Intel RAPL counters available through the Linux powercap interface on our Broadwell-EP processor. The `nvprof` systemwide profiling mode is used to sample GPU power statistics which are integrated over the entire run to obtain energy. We measure energy for complete executions, and not just for computation. When reporting energy for the GPU, we exclude CPU energy for the host part of the program.

Memory usage is measured for the GPU using the `cudaMemGetInfo` interface, once at the beginning of the program and again immediately after the computation ends, but before deallocation. Memory usage for CPU is collected from Galois's internal memory allocator which tracks OS memory allocations during program runs. For miniTri, glibc's `malloc_stats` is used to report the total in use size. Julia's `@time` macro is used to track memory allocated.

Our runtimes include end-to-end calculation time after the graph is loaded and before the results are printed. All results were verified by comparing to the benchmark code when possible and by checking that the output satisfied the triangle and k-truss properties. Some results are missing because all benchmarks were limited to a maximum of 4800 seconds or

[2]The reference Python version produced incorrect results for $k > 3$

TABLE I: Datasets used in experiments. Size is in bytes.

Graph Name	$\|V\|$	$\|E\|$	Size
amazon*	262111–410236	899792–2443408	16M-41M
as20000102	6474	12572	248K
as-caida20071105	26475	53381	1.1M
ca-AstroPh	18772	198050	3.2M
ca-CondMat	23133	93439	1.7M
ca-GrQc	5242	14484	268K
ca-HepPh	12008	118489	2.0M
ca-HepTh	9877	25973	484K
cit-HepPh	34546	420877	6.7M
cit-HepTh	27770	352285	5.6M
cit-Patents	3774768	16518947	281M
com-amazon	548552	925872	19M
com-dblp	425957	1049866	20M
com-friendster	124836180	1806067135	28G
com-lj	4036538	34681189	560M
com-orkut	3072627	117185083	1.8G
com-youtube	1157828	2987624	55M
email-Enron	36692	183831	3.1M
email-EuAll	265214	364481	7.6M
facebook_combined	4039	88234	1.4M
flickrEdges	105938	2316948	37M
graph500-scale18-ef16	174147	3800348	60M
graph500-scale19-ef16	335318	7729675	121M
graph500-scale20-ef16	645820	15680861	245M
graph500-scale21-ef16	1243072	31731650	494M
graph500-scale22-ef16	2393285	64097004	997M
graph500-scale23-ef16	4606314	129250705	2.0G
graph500-scale24-ef16	8860450	260261843	4.0G
loc-brightkite_edges	58228	214078	3.8M
loc-gowalla_edges	196591	950327	17M
oregon1*	10670–11174	21999–23409	428K-456K
oregon2*	10900–11461	30855–32730	568K-604K
p2p-Gnutella0*	6301–10876	20777–39994	376K-712K
p2p-Gnutella2*	22687–26518	54705–65369	1.1M-1.3M
p2p-Gnutella30	36682	88328	1.7M
p2p-Gnutella31	62586	147892	2.8M
roadNet-CA	1965206	2766607	58M
roadNet-PA	1088092	1541898	32M
roadNet-TX	1379917	1921660	40M
soc-Epinions1	75879	405740	6.8M
soc-Slashdot0811	77360	469180	7.8M
soc-Slashdot0902	82168	504230	8.4M

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

because the graphs did not fit into GPU memory.

In our results, we report edge rate (edges processed per second), edge rate per energy (edges/second/Joule), and memory usage (bytes) for all benchmarks. Rate is calculated as number of (undirected) edges in the graph divided by the runtime of the computation. In all the figures, input graphs are ordered by increasing number of edges.

All CPU metrics are reported as *cpu-N* with N being one of 1, 12 or 24 threads. By default, our GPU metrics (*gpu*) include time for data transfer and GPU memory allocation since our implementations currently use the blocking versions of these APIs which may consume significant time for small graphs. We also present results that exclude time for data transfers and memory allocations as *gpu-nomem*. Metrics for the reference implementations are reported as *miniTri* and *julia*.

A. Results for Triangle Counting

Fig. 2 shows the edge processing rate (edges/second) for triangle counting on all our input graphs. Across all inputs, our implementations are 19x (*cpu-01*) to 22081x (*gpu*) faster than *miniTri*. Among our implementations, *cpu-12* is fastest for smaller inputs (up to p2p-gnutella04) but is outperformed by *cpu-24* for the rest of the inputs. The single-threaded *cpu-01* is only competitive for very small inputs. The GPU (*gpu*) only outperforms the CPU for inputs larger than cit-HepTh, with rates up to 8x better than the CPU.

If data transfer time is ignored, the GPU (*gpu-nomem*) outperforms all the other variants on all the inputs. Since reading the graph from disk usually takes much longer than transferring it to GPU, techniques such asynchronous memory transfers to the GPU should be used to hide data transfer latency if data transfer times are significant.

For our implementations, the processing rates depend on the number of edges in the input graph. It is relatively constant regardless of the number of threads until the input has more than 50K edges. At this point, the multi-threaded versions can deliver up to 10x the rate of *cpu-01*. This indicates that the amount of parallelism is limited by the input size, and explains why *cpu-12* has better processing rates than *cpu-24* for small inputs. Surprisingly, the processing rates drop sharply below that of the small inputs for large inputs with more than 3M edges. This is particularly noticeable in the graph500 synthetic inputs, but is also visible in the large community inputs. Since the performance drops across devices, it is likely to be a characteristic of the input graph, but we do not understand this behavior yet.

Fig. 3 presents the edge processing rate (edges/second) per unit energy (Joule). All our implementations again deliver 3.85x to 121534900x edge processing rates for a single unit of energy compared to *miniTri*. On this performance per energy metric, our GPU implementation outperforms all our CPU variants – it provides 10x the processing rate per unit energy for small inputs and can be up to 100x faster for the same energy on larger inputs.

Finally, Fig. 4 details the memory usage in bytes for all the implementations. Our GPU implementation uses the

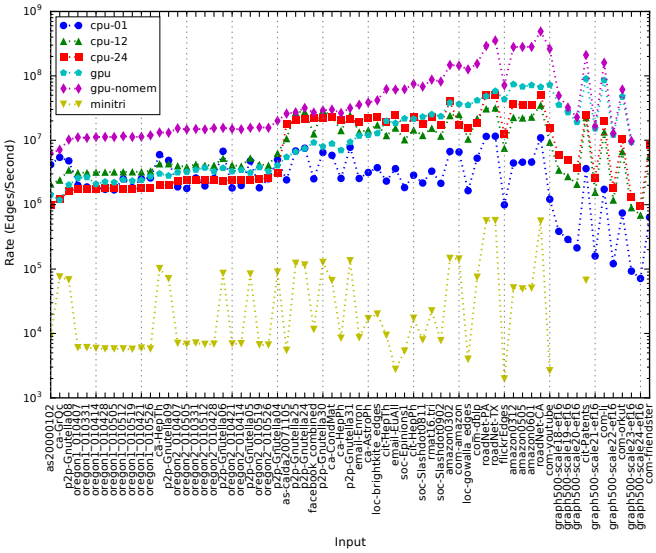

Fig. 2: Triangle Edge Rate (edges/s). Higher is Better.

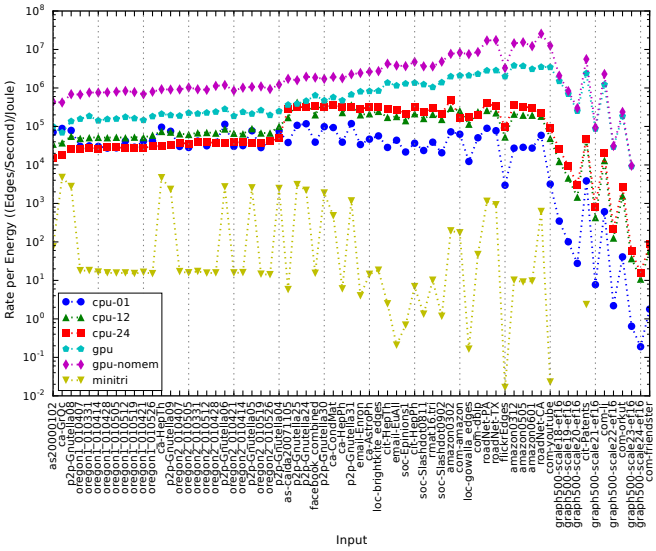

Fig. 3: Triangle Rate per Unit Energy (edges/s/J). Higher is Better.

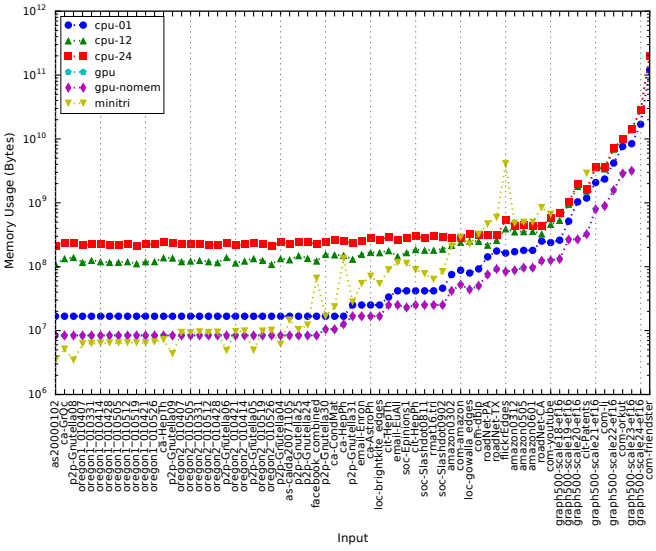

Fig. 4: Triangle Memory (Bytes). Lower is Better.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

least memory among all our implementations. All our CPU implementations suffer a constant memory overhead per thread for small graphs, thus *cpu-24* consumes twice the memory of *cpu-12*. Depending on the device, input graph size becomes the dominant factor for memory consumption around the p2p-gnutella30 input. Unlike other implementations that only count triangles, *miniTri* needs to store the actual triangles in its result matrix [15]. Since the number of triangles is much larger than edges for the largest inputs, *miniTri* uses the most memory for the largest inputs.

B. Results for K-Truss Computation

Fig. 5 shows the edge processing rate (edges/second). In general, our implementations are at least 66x faster than *julia* and can be up to 34811x faster. K-truss in *julia* slows down for graphs with more than 150K edges.

Like for triangles, the CPU *DirectTruss* implementations start out at around 1M edges/second. This increases to 20M edges/second for the larger inputs before rates reduce sharply for the largest inputs with more than 3M edges. The performance of the GPU implementations closely matches the better of *cpu-12* or *cpu-24* for most of the graphs, but is slower than the CPU for the graph500 synthetic graphs. Again, if data transfer times did not matter, the *gpu-nomem* implementation would outperform the CPU implementations.

The CPU *CoreThenTruss* implementation is 2x faster than *DirectTruss* for graphs larger than com-youtube but 2x slower for all other graphs, so it is not presented.

Fig. 6 presents the edge processing rate (edges/second) per unit energy (Joule). Our CPU implementation deliver 14257x (geomean) the processing rate for the same amount of energy compared to *julia* while our GPU implementations deliver 203798x (geomean). Our GPU implementation is also 10x faster than our CPU implementation for the same amount of energy for graphs of up to 3M edges except for Graph500 graphs, where the poor performance also leads to a poor rate per energy.

Fig. 7 shows memory usage in bytes. The *julia* implementation consumes memory rapaciously, utilizing tens to hundreds of gigabytes even when then are only four graphs that are larger than a gigabyte (see Table I). Julia is a managed language and its garbage collector is unable to efficiently utilize memory. In contrast, all our implementations use manual memory management. Memory usage for GPU k-truss is significantly higher than that for GPU triangles since it uses additional auxiliary structures to track active edges, node degrees, mirror edges, etc. The GPU consumes more memory than the CPU for inputs having more than 2M edges.

V. Conclusion

Our use of graph-centric methods for triangle counting and k-truss identification permits several optimizations that are difficult when using matrix algebra techniques. Our implementations, both on the CPU and GPU, therefore deliver multiple orders of magnitude improvement across all metrics – rate, rate per energy and memory usage – when compared to the reference GraphChallenge code.

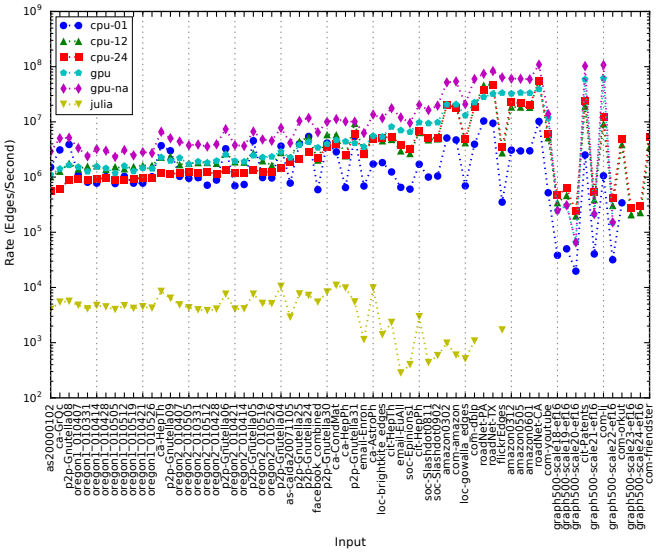

Fig. 5: K-Truss Edge Rate (edges/s). Higher is better.

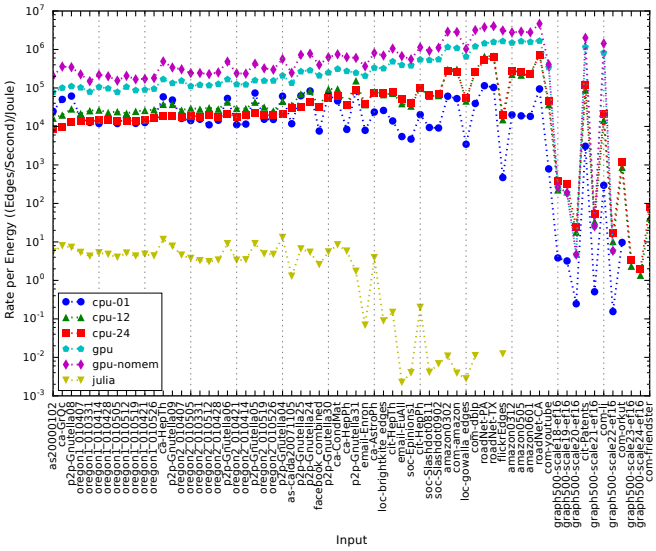

Fig. 6: K-Truss Rate per Unit Energy (edges/s/J). Higher is better.

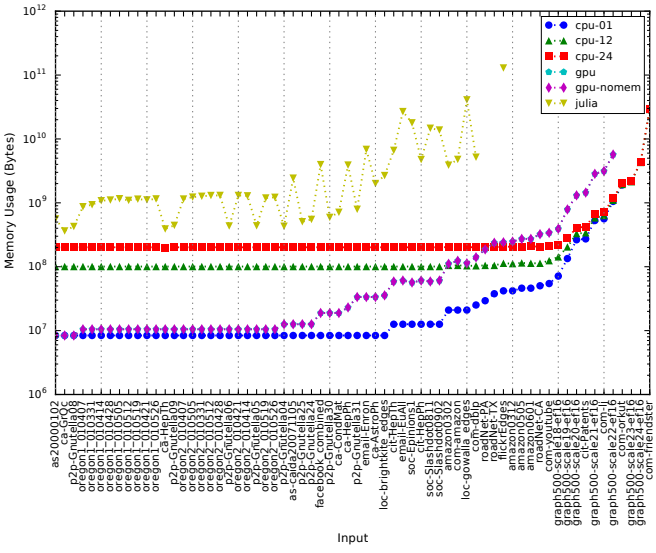

Fig. 7: K-Truss Memory (Bytes). Lower is better.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

References

[1] Sean Baxter. Moderngpu 1.0. https://github.com/moderngpu/moderngpu, 2015.

[2] Paul Burkhardt. Graphing trillions of triangles. *Information Visualization*, 16(3):157–166, 2016. doi: 10.1177/1473871616666393.

[3] Jonathan Cohen. Trusses: Cohesive subgraphs for social network analysis. In *National Security Agency Technical Report*, 2008.

[4] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. Ordered vs unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 3–12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0119-0. doi: http://doi.acm.org/10.1145/1941553.1941557. URL http://iss.ices.utexas.edu/Publications/Papers/ppopp016s-hassaan.pdf.

[5] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009. URL http://iss.ices.utexas.edu/Publications/Papers/ispass2009.pdf.

[6] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel graph analytics. *Commun. ACM*, 59(5):78–87, April 2016. ISSN 0001-0782. doi: 10.1145/2901919. URL http://doi.acm.org/10.1145/2901919.

[7] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[8] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522739. URL http://doi.acm.org/10.1145/2517349.2522739.

[9] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *OOPSLA 2016*, pages 1–19, 2016. doi: 10.1145/2983990.2984015.

[10] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, Muhammad Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *PLDI 2011*, pages 12–25, 2011. doi: 10.1145/1993498.1993501.

[11] Adam Polak. Counting triangles in large graphs on GPU. In *IPDPS Workshops 2016*, pages 740–746, 2016. doi: 10.1109/IPDPSW.2016.108.

[12] Siddharth Samsi, Vijay Gadepally, Michael Hurley, Michael Jones, Edward Kao, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Steven Smith, William Song, Diane Staheli, and Jeremy Kepner. Static graph challenge: Subgraph isomorphism. In *IEEE HPEC*, 2017.

[13] Thomas Schank. *Algorithmic Aspects of Triangle-Based Network Analysis*. PhD thesis, Universität Karlsruhe, 2007.

[14] Yingxia Shao, Lei Chen, and Bin Cui. Efficient cohesive subgraphs detection in parallel. In *SIGMOD 2014*, pages 613–624, 2014. doi: 10.1145/2588555.2593665.

[15] Michael M. Wolf, Jonathan W. Berry, and Dylan T. Stark. A task-based linear algebra building blocks approach for scalable graph analytics. In *HPEC 2015*, pages 1–6, 2015. doi: 10.1109/HPEC.2015.7322450.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE