Static Graph Challenge on GPU

Mauro Bisson, Massimiliano Fatica
NVIDIA Corporation
Santa Clara, CA 95050, USA

***Abstract*—This paper presents the details of a CUDA implementation of the Subgraph Isomorphism Graph Challenge, a new effort aimed at driving progress in the graph analytics field. challenge consists of two graph analytics: triangle counting and k-truss. We present our CUDA implementation of the graph triangle counting operation and of the k-truss subgraph decomposition. Both implementations share the same codebase taking advantage of a set intersection operation implemented via bitmaps. The analytics are implemented in four kernels optimized for different types of graphs. At runtime, lightweight heuristics are used to select the kernel to run based on the specific graph taken as input.**

I. Introduction

The use of GPUs in high performance computing, sometimes referred to as *GPU computing*, is now very popular due to the high computational power and high memory bandwidth of these devices coupled with the availability of high level programming languages and tools. In this work, we want to explore the GPU capabilities for big data workloads by implementing the Subgraph Isomorphism Graph Challenge in CUDA.

II. GraphChallenge

A detailed description of the benchmark could be found in [1]. Currently, the Static Graph Challenge is comprised of two problems focusing on different types of subgraphs: triangle counting and k-truss decomposition.

As the name suggests, triangle counting requires to find the exact number of cycles of length three contained in a given graph. This measure is commonly used to compute other metrics in a graph. An important example is the clustering coefficient, used to identify influential entities in a social network or to measure the degree to which the nodes tend to form clusters.

K-truss decomposition is another problem which requires to identify a subgraph of a given input graph. In this case it is necessary to find the maximal subgraph such that each edge is contained in at least $k-2$ triangles belonging to the subgraph. The k-truss decomposition is computationally more expensive (especially for large k) than triangle counting.

Clearly, the two problems are related to each other, since to to find trusses it is necessary to compute edge supports, i.e. the number of triangles to which an edge contributes. For this reason, we first implemented the triangle counting benchmark from scratch and then used the counting kernels as a building block for the k-truss implementation.

In the next sections we describe our CUDA implementation of the triangle counting and the k-truss benchmarks. We chose to store the graph in the Compressed Sparse Row format as it is a widely used data structure for graph processing and because its properties can be effectively exploited for the triangle counting computation. Both codes treat the input graph as undirected.

For what concerns the output, the triangle counting code produces a single number that represent the exact number of triangles found in the input graph. The k-truss code instead outputs the whole subgraph whose edges contribute to at least $k-2$ triangles.

III. Triangle counting

An intuitive way of approaching the triangle counting problem is to consider the matrix-based algorithm, introduced by Azad et al. in [2]. Starting from an adjacency matrix A of an undirected graph G, let L and U be the lower and upper triangular halves of A, not including the diagonal. The multiplication of L by U produces a matrix C whose element c_{ik} counts the number of wedges (i,j,k) (paths of length 2) with a center j, having an index smaller than both i and k. The number of triangles of G can then be computed by summing all the nonzeroes of C that correspond to nonzeroes of A, i.e. the number of all the wedges (i,j,k) that form triangles with the edge $\{i,k\}$, and dividing the result by two (c_{ik} and c_{ki} count the same quantity).

Since A is symmetric, C is also symmetric and thus there is no point in computing it fully. It is sufficient to compute the lower (or upper) half without the diagonal and then sum the nonzeroes selected by element-wise multiplication with L (U). Moreover, since only the nonzeroes selected by L are required for the final sum, the dot products required by $L \times U$ can be restricted exclusively to the nonzeroes of L.

$$N_{tri} = Sum(Lower(L \times U) \ .* \ L) \tag{1}$$

Figure 1 shows an example of triangles counted by using formula 1. Considering the matrix rows as sets of indices, the counting can be expressed in terms of intersections between each nonempty row of L and the columns of U indexed by the nonzeroes in that row. Algorithm 1 shows a pseudocode for this procedure that uses a sparse representation of the matrix L consisting of a set of adjacency lists and that uses the symmetry of A, i.e. $U = L^T$.
This algorithm is essentially the counting version of the $forward$ algorithm for triangle enumeration introduced in [3]. Each triangle $\{i,j,k\}$ is counted exactly once when computing the intersections at line 5. More precisely it accumulates, to the total count, all the triangles i,j,k with $i > k$ and $k > j$.

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

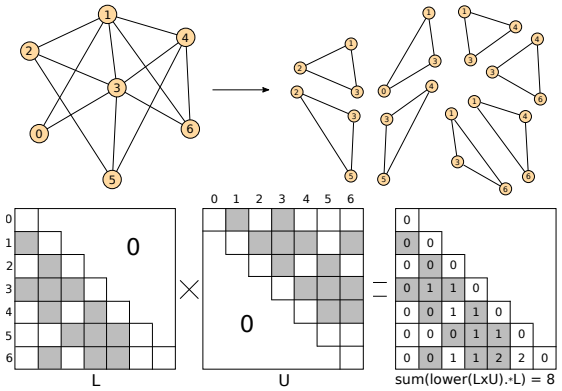

Figure 1. Example of triangle counting via multiplication of the two halves of an adjacency matrix. The sum is restricted to only the grey elements of the original L matrix.

Algorithm 1 Pseudocode for serial triangle counting.

Input: Adjacency lists $Adj_L(v)$, $0 \leq v < n$
Output: Number of triangles

```
1: cnt = 0
2: for each i in 0...n − 1 do
3:     curr_adj = Adj_L(i)
4:     for each k in curr_adj do
5:         cnt += |curr_adj∩Adj_L(k)|
6: return cnt
```

With respect to the example shown in Figure 1, each triangle is counted when the adjacency list of its highest index vertex is processed. More precisely, when it is intersected with the adjacencies of the vertex with the middle value.

Algorithm 1 can be easily parallelized by exploiting the fact that each adjacency list can be processed independently from the others. This leads to a plain CUDA data-to-threads mapping where each row of the matrix is processed by a group of threads which counts all the triangles having in that row their vertex with highest index.

Almost all of the performance and complexity of the code lies in the implementation of the intersection operation between adjacency lists. In contrast to all other works targeting GPUs that we are aware of, we implemented a bitmap-based intersection. Given two lists, their intersection is performed by first setting the bitmap bits corresponding to the values in the first and then looking for set bits corresponding to each element in the second list. In this way it is possible to avoid multiple scans when the same lists have to be intersected with many others. The main issues when dealing with bitmaps are *(i)* the parallel update of bits (that requires the use of atomic operations, now very efficient on the last generations of GPUs) and *(ii)* the non-optimal bandwidth usage due to the natural, scattered access pattern to which bitmaps are subject.

We implemented Algorithm 1 in CUDA in four kernels that process the adjacency lists using cooperating thread groups of different sizes. Two of them use one block per row, one uses a warp per row and the last uses a single thread per row. We select which kernel to call by using an heuristic based on a simple metric that can be computed with negligible cost. Further details about our triangle counting implementation for CUDA can be found in [4].

IV. K-Truss decomposition

K-Truss decomposition is another problem involving triangles. The truss is a type of subgraph introduced by Cohen in [5] that generalizes the concept of clique. Since a clique is a graph such that every two distinct vertices are adjacent, a clique of order k can be defined as a connected subgraph such that:

1) has exactly k vertices;
2) each couple of connected vertices is also connected with additional $k-2$ vertices.

A k-truss relaxes the condition on the number of vertices, requiring only the second property (thus allowing for a number of vertices $\geq k$). This definition captures subgraphs of high cohesion (each connection between two vertices supported by at least $k-2$ common neighbors) which can also be computed efficiently (in polynomial time).

The objective of the k-truss decomposition problem proposed in the Graph Challenge is to find all the maximal k-trusses in a given graph.

As shown in [5], a simple and intuitive way of performing the decomposition is to prune the graph iteratively by removing, at each iteration, all the edges connecting vertices that share less than $k-2$ neighbors, as shown in Algorithm 2.

Algorithm 2 Pseudocode for k-truss decomposition.

Input: k, $G = (V, E)$ undirected
Output: k-truss decomposition

```
1: while True do
2:     n = |E|
3:     for each e = {u, v} in E do
4:         if (|N(u) ∩ N(v)| < k − 2)  E = E \ {e}
5:     if n == |E| break
6: V = ⋃_{e∈E} e
```

The number of neighbors in common to the ends of an edge is usually called the *support* of that edge. Clearly, the support of an edge represents the number of triangles that contain that edge.

In our implementation we followed a similar approach based on an initial computation of the support function for every edge which is then iteratively refined in order to remove all the edges not belonging to any truss. At each iteration, the function is first used to remove the *weak* edges (those with insufficient support) and then it is updated to take into account the removals. This is done by reducing by one the support of the remaining edges in the broken triangles. The process ends when the support function is greater than or equal $k-2$ for each of the remaining edges.

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

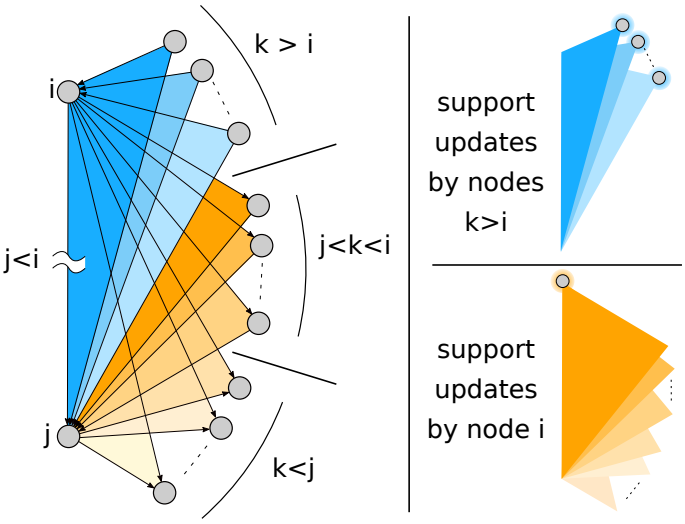

Figure 2. The removal of edge (i, j) breaks triangles formed with vertices k greater than i (and j, in blue) and lesser than i (in orange) thus causing a reduction of the support of the remaining edges by 1. Since we discover triangles starting from their highest index vertex, in order to find the broken triangles and update the supports, it is sufficient to process only the nodes with index $k > i$ and node i. On the right, each triangle is shown with the vertex responsible for the support update of its (remaining) edges.

As in the case of triangle counting, we represent the undirected graph as a directed one with edges connecting vertices with decreasing index (i.e. we use the lower triangular of the adjacency matrix). This allows to halve the number of edges to process thus saving memory and processing time. Moreover, it make possible to avoid explicit management of multiple concurrent discoveries of the same triangle. Since we find triangles by intersecting the adjacency list of a vertex with those of its neighbors (with lower id), each triangle is discovered exactly once, when the adjacency list of its highest-index vertex is processed.

Algorithm 3 shows the pseudocode for our implementation of the k-truss decomposition.

Algorithm 3 Pseudocode for k-truss.

Input: k, $G = (V, E)$ directed
Output: Subset of edges with support $\geq k - 2$

1: S = get_support(V, E)
2: **while** True **do**
3: E_{weak} = $\{e \in E | S(e) < k - 2\}$
4: **if** ($E_{weak} == \emptyset$) **break**
5: **if** $k > 3$ **then**
6: V_{weak} = $\{u | (u, v) \in E_{weak}\}$
7: V_{weak} = $V_{weak} \cup \bigcup_{u \in V_{weak}} N^-(u)$
8: update_support(S, V_{weak}, E, E_{weak})
9: E = $E \setminus E_{weak}$
10: **if** k == 3 **break**
11: **return** E

Initially, we compute the number of triangles that each edge contributes to (line 1) and then the iterative phase starts. The `get_support` routine basically implements a triangle counting procedure that, instead of adding to a global count, increments, for each discovered triangle, the local counters associated to its edges.

At each iteration of the loop, the weak edges are first identified (line 3). If this set is empty then each of the remaining edges belongs to a k-truss and thus the loop is exited (line 4) and the current set is returned as the solution (line 11). Otherwise, if there are edges with support less than $k - 2$ then they must be removed and the supports for the broken triangles need to be updated. However, if k is equal to three then weak edges do not belong to any triangle and thus the support update can be skipped (line 5).

The removal of a weak edge can only affect the support of the edges connected to its ends and so it is desirable to limit the update to the edges "close" to the weak ones. Since our kernels are vertex-based and, for each vertex, they visit all the triangles formed with vertices having lower id, we can restrict the update to the source vertices of the weak edges and to the vertices with edges leading to weak edges (lines 6 and 7). In this way we restrict the update to the smallest subset of vertices that are connected to the edges affected by the removals. We call this the set of weak vertices. Figure 2 depicts the triangles involved in the removal of a weak edge, in the general case, and the vertices that are processed in order to update the supports of the edges of the broken triangles.

The `update_support` routine finds all the triangles by processing the adjacency lists of the weak vertices and, for each one containing a weak edge, decrement by one the supports of its edges (line 8). Finally, the weak edges are removed from the edge set (line 9) and, if k equals three, then no additional iteration is performed. This simple optimization is possible because weak edges in a 3-truss cannot be part of triangles and so their removal cannot break any.

A. CUDA Implementation

Almost all of the complexity of our k-truss implementation resides in the routines used to compute and update the supports. As previously stated, we store the input graph as a CSR containing the lower triangular of the adjacency matrix. Supports are stored in the value array, with one element per nonzero. In addition to the usual CSR arrays, there is an additional array with an element per row that we use to store the length on each adjacency list. So, denoting with *N* the side of the matrix, with *n* the number of nonempty rows and with *NNZ* the number of nonzeroes, we use the following arrays:

- `cols[NNZ]`: nonzeroes in each row;
- `supp[NNZ]`: edges supports;
- `roff[N+1]`: starting offset of each row;
- `rlen[N]`: length of each row;
- `rows[n]`: indices of the nonempty rows;

We use the `rlen` array in order to avoid recomputing the CSR after the removal of weak edges that takes place at the end of each iteration. Instead, we compact the rows containing weak edges and update the corresponding `rlen` entries.

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

Once the arrays are created, they are copied in device memory and then all the computations described in Algorithm 3 are performed by GPU kernels. The most computationally expensive are, by far, the support computation and update kernels and so we'll focus on those in the following.

Similarly to our triangle counting code, we implemented different versions of both kernels, each one using a thread group of different size to process the adjacency lists. We developed a total of four versions: two that use a full block per row, one using warps and one using a single thread per row. All but the thread-based kernel use a bitmap to perform the intersections between adjacency lists. More precisely, each thread group uses a private bitmap that is set with the nonzeroes in the current row, and that is read for each nonzero in the adjacency lists with which it is intersected.

Algorithm 4 CUDA block-based parallel support computation.

Input: cols[NNZ], array of column indices
Input: roff[N+1], array of row offsets
Input: rows[n], array of nonempty row indices
Input: bitmap[**gridDim**.x][], array of bitmaps, one per thread block
Output: Edge supports in supp[NNZ]

```
 1: bmap = bitmap[blockIdx.x]
 2: for (s = blockIdx.x; s < n; s += gridDim.x) do
 3:     i = rows[s]
 4:     io_s = roff[i]
 5:     io_e = io_s + rlen[i]
 6:     for (io = io_s; io < io_e; io += blockDim.x) do
 7:         c = (io+threadIdx.x < io_e) ? cols[io+threadIdx.x] : -1
 8:         if (c > -1) atomic_set(bmap+c)
 9:         for (t = 0; t < blockDim.x; t++) do
10:             j = broadcast(c, t)
11:             if (j == −1) break
12:             cnt = 0
13:             jo_s = roff[j]
14:             jo_e = jo_s + rlen[j]
15:             for (jo = jo_s+threadIdx.x; jo < jo_e; jo += blockDim.x) do
16:                 k = cols[jo]
17:                 if (bmap[k] == 1) then
18:                     cnt += 1
19:                     atomic_add(supp+jo, 1)          ▷ (j, k) supp.
20:                     l = find_off(cols+io_s, k)
21:                     atomic_add(supp+io_s+l, 1)      ▷ (i, k) supp.
22:             atomic_add(supp+io+t, cnt)              ▷ (i, j) supp.
23:     reset_bitmap(bmap)
24: return supp[]
```

The kernels share most of their structure with the corresponding ones from the triangle counting implementation. Algorithm 4 shows a simplified pseudocode of our block-based kernel for support computation. The bitmaps are assumed to be set equal to zero upon kernel entry for every block. The blocks loop over the nonempty rows until each one is processed (line 6). Rows are read in chunks of length equal to the block size (line 7) and, for each nonzero, the corresponding bitmap bit is set (line 8). Threads that would read past the current line set their column index to -1 instead, and don't write to the bitmap. In the pseudocode, `bmap` is assumed to be a bit array while in the actual code it is a 64-bit integer array so, in order to avoid race conditions due to multiple threads setting bits in the same word, we use an `atomicOr` operation for the writes. The word containing the bit is found by dividing the nonzero by 64 and the bit offset inside the word is computed via a modulus operation. At this point the blocks start to process the adjacency lists corresponding to the nonzeroes in the current chunk (line 9). One at a time, each thread broadcasts its nonzero to the other threads (line 10). The broadcast of a -1 signals that the last chunk of the current row has been processed (line 11). For each nonzero, the adjacency list of the corresponding row j is intersected (in parallel) with row i in order to find the triangles formed by common neighbors. First of all, a local, per-thread counter of the triangles passing through the edge (i, j) is reset (line 12), and the offsets of row j are computed (lines 13 and 14). Then the threads loop over the adjacency list, independently of each other (line 15). Each nonzero k is checked against the bitmap (line 17) in order to identify a triangle containing it. If the k-th bit is set then it means that k was in the adjacency list of i and thus (j, k) closes a triangle with (i, k) and (i, j) (with $i > j > k$). Figure 3 shows the correspondence between a triangle and the nonzeroes in the adjacency lists of its vertices. A triangle $<i, j, k>$ is identified exactly once, in the block handling the adjacency list of i, by the thread that reads the nonzero k in the adjacency list of j. For each triangle, the support of edge (i, j) local to the thread is increased by one (line 18) and then the supports for edges (j, k) and (i, k) are also incremented. Since the supports of every edge can be modified by any thread in the grid, the `supp` array is modified via atomic operations. Moreover, as it shares the size and layout of the nonzero array `cols`, the increments are performed using the same offsets of the nonzeroes. The increment for edge (j, k) is performed immediately, as the thread that read k can use the same offset for the support array (line 19). For what concerns edge (i, k), on the other hand, the thread not necessarily knows its offset as the k-th bit may have been set in line 8 by another thread (in the current iteration) or even during the read of a previous chunk from row i. For this reason, the offset of k in the adjacency list of i must be retrieved. This is done by the `find_off` call (line 20). This function can be implemented in many ways, for example by either a linear or a binary search on row i. However, since it is executed for every triangle it has a sensible impact on the overall running time of the kernel and thus we optimized it so that it is executed in $O(1)$ time (we'll describe this optimization later). Once the offset is found, the support of (i, k) is incremented (line 21). After the processing of row j, each thread atomically adds to the support of (i, j) the number of triangles it found (line 22) and then the block starts the processing of the next next nonzero of the current chunk. After all the nonzeros in the last chunk of i have been processed, the block resets the bitmap (line 23) and begins the processing of the next row.

At the beginning of the support refinement phase (line 3 of Algorithm 3), weak edges are marked by setting their support to -1. Then the `update_support` kernel is called passing a nonempty rows array containing the weak vertices identified in lines 6 and 7. For each triangle having at least one edge with negative support, the positive support of the other edges

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

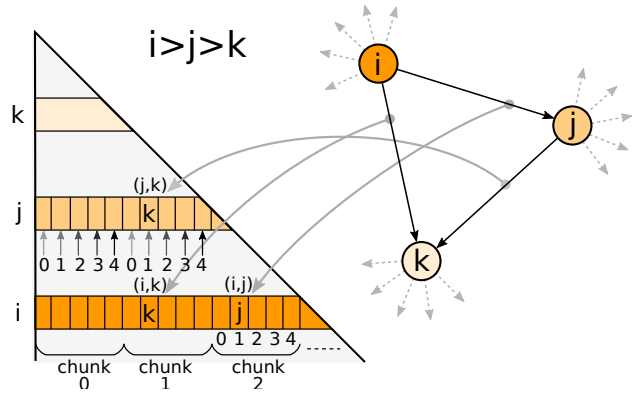

Figure 3. The edges of triangle <i, j, k> are located in the adjacency lists of vertices i and j. Since we store only directed edges leading to vertices with lower id, no edge is present in the adjacency list of k. More precisely, two edges are in the list of the vertex with highest id, i, and the third is in list of the middle valued vertex, j. The triangle is found by the block that processes row i. During the processing of the third chunk, at the second broadcast, thread 1 transmit nonzero j. At this point, the bits in the bitmap that correspond to the nonzeroes in chunks 0, 1, and 2 are all set. During the intersection of row j with row i, thread 1 reads nonzero k and finds that the corresponding bitmap bit is set. Thus, it increases by one: *(i)* its local counter for triangles found from i, *(ii)* the support of (i, j) and, *(iii)* the support of (j, k).

(if any) is reduced by one. This is done by performing explicit checks on the supports before the atomic operations used for the decrements. The update is performed with a kernel similar to that described by Algorithm 4.

In the end, the removal of weak edges at the end of the iteration is performed by a kernel using a warp per row in the CSR to compact the nonzero and support arrays by removing the entries with support equal to -1.

For what concerns the kernel launch configuration, using a bitmap per thread group limits the number of groups that can be launched by the amount of available memory. So we launch the largest possible grid with size smaller than the number of rows and have the blocks cycle through the adjacency lists.

V. Heuristic

In general, with CSRs having a number of nonzeroes per row greater than the number of treads per block, it is usually better to use the block-based kernels. This is true for both triangle counting and k-truss and is mainly due to a higher cache hit rate when accessing the bitmaps. When the row length reduces below a certain value, the waste due to unused threads in the block begins to limit the performance and thus a warp-based kernel should be used. With very short lengths, however, even using a warps results in a large waste of computational resources and so it is necessary to resort to thread-based processing. In practice, row lengths can vary sensibly and the performance also depends on the topology of the datasets. An accurate performance model for each kernel would be quite complex to develop and it may also have a computational cost comparable to that of the computing kernels. We believe that the design of such heuristic to be outside the scope of this work, requiring the analysis of a wide range of graphs, each having specific topology and properties.

For those reasons, we looked for simple heuristics based on inexpensive metrics that could select the best kernel specifically for the datasets considered in this work, those specified in the Graph Challenge[1] and the large Twitter graph[2] from Kwak et al. [6].

By analyzing the timings of all the kernels for both challenges on each graph, we devised the following heuristic that is able to select the best kernel for 93% of the datasets in case of triangle counting and for 90% in case of k-truss:

Algorithm 5 Kernel selection heuristic.

Input: $G = (V, E)$ directed
Output: Kernel type
1: avg_deg = |E|/|V|
2: max_deg = $d^+(G)$
3: **if** (max_deg < 13.0) **return** TB kernel
4: **if** (avg_deg < 8.0) **return** WB kernel
5: **if** (|V| < 1024*128) **return** BB-SHMEM kernel
6: **return** BB-GLMEM kernel

The constants have been found by a purely empirical process. The thread-based kernel is called for graphs with a maximum degree (maximum length of CSR rows) of 12 because this is the threshold up to which that variant of the support kernel can perform a processing completely in the registers using no more than 32 registers per thread. If the maximum row length is greater than 12 but the average length is less than 8.0 then the warp-based version is used. Otherwise, in case the average length is higher, we use the block-based version. For these graphs, we found that 128K is the threshold on the number of vertices below which the version storing the bitmaps in shared memory delivers higher performance than the one using global memory.

VI. Results

We ran our triangle counting and k-truss implementations on the SNAP and Synthetic datasets specified by the Static Graph Challenge and on the large Twitter graph from [6]. For each algorithm, we report three timings (Fig. 4):

1) **Kernel time**: wall clock time elapsed between the first and the last CUDA kernel called. For triangle counting it only includes the counting kernel and the CUB reduction while for k-truss it includes the entire implementation of Algorithm 3.
2) **Process time**: this timing includes all the computations from the moment the CSR is ready in GPU memory. It includes: the heuristic, allocations, initializations and deallocations of both the bitmaps and the temporary results arrays, and everything timed in 1.
3) **Total time**: this timing starts right after the dataset has been read from file to device memory. It includes the

[1] Available at http://graphchallenge.mit.edu/data-sets#PartitionDatasets
[2] https://an.kaist.ac.kr/traces/WWW2010.html

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

removal of self-loops, of multiple edges, the creation of the CSR and everything timed in 2.

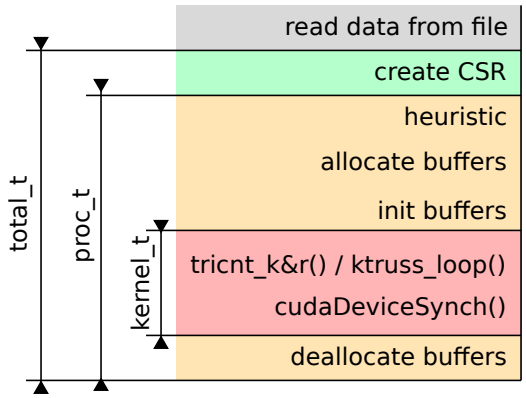

Figure 4. Reported times.

Table I summarizes the results we obtained on a Titan X Pascal (equipped with 12GB of global memory). For each graph, we report the number of vertices and edges read from file, the processing kernel selected by the heuristic, and the timings for both triangle counting and k-truss. The Friendster graph from the SNAP dataset is not present in the table as it does not fit in the memory available on the Titan X Pascal. The table is summarized in Figure 6, where the traversed edges per seconds are plotted with the graphs sorted by the kernel performances. Since most of the graphs are processed in fractions of a second, the creation of the CSR and the memory allocation/deallocation have a significant effect on performances. With larger graphs (like the Graph500 ones), the overhead of these operations becomes negligible.

For k-truss, we also performed a series of runs to measure the impact of increasing the target support parameter, k, on the total running time. We computed the decompositions varying k, between 3 and 100, of the largest datasets processed with each kernel type and of the largest dataset of all, the Twitter graph. The total runtimes are plotted in Figure 5. From this plot, it is clear that there is a notable increase in processing time going from k equal 3 (the current value in the Static Graph Challenge) to larger values, but then the increase becomes very moderate.

VII. Conclusions

Building on a very fast triangle counting implementation (as shown in [4], this is the fastest triangle counting implementation available, outperforming the current state-of-the-art works by a factor ranging from 2x to 11x), we were able to build a very fast and flexible k-truss computation. Since we use a standard CSR storage format for the graph and we do not need any specific pre-processing, the algorithms presented could be easily added to other codes. This work shows that GPUs perform very well in this challenge and represents additional evidence of the relevance of GPU computing for graph processing [7].

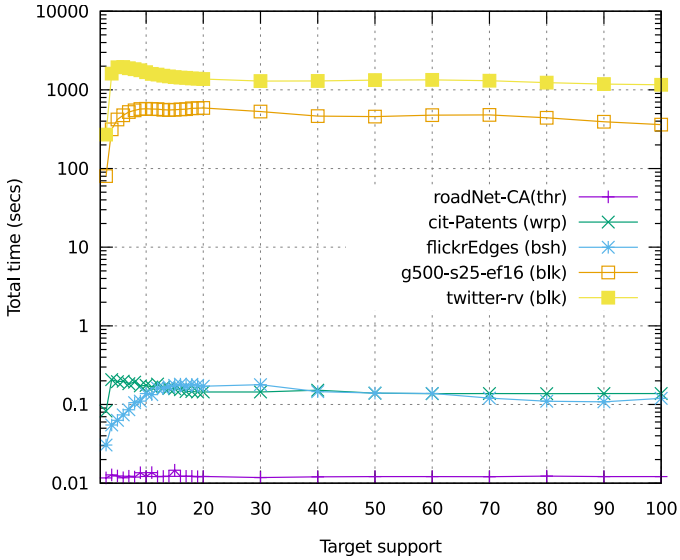

Figure 5. Total k-truss running time for the largest graphs processed with each kernel type (from Table I), measured with target support ranging from 3 to 100.

References

[1] "Static Graph Challenge: Subgraph Isomorphism", S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, J. Kepner, *IEEE High Performance Extreme Computing Conference (HPEC)*, 2017

[2] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, ser. IPDPSW '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 804–811. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2015.75

[3] T. Schank and D. Wagner, *Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 606–609. [Online]. Available: http://dx.doi.org/10.1007/11427186_54

[4] M. Bisson; M. Fatica, "High Performance Exact Triangle Counting on GPUs", in *IEEE Transactions on Parallel and Distributed Systems*, 2017, doi: 10.1109/TPDS.2017.2735405.

[5] J. Cohen, "Trusses: Cohesive subgraphs for social network Analysis" *National Security Agency Technical Report*, 2008.

[6] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW '10: Proceedings of the 19th international conference on World wide web*. New York, NY, USA: ACM, 2010, pp. 591–600.

[7] M. Bisson, E. Phillips and M. Fatica, "A CUDA implementation of the Pagerank Pipeline benchmark", *2016 IEEE High Performance Extreme Computing Conference (HPEC)*

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

Dataset	#verts	#edges	KType	Running times on Titan X Pascal (secs)						
				Triangle counting				K-Truss (k=3)		
				Triangles	Kernel	Process	Total	Kernel	Process	Total
roadNet-PA_adj	1,088,092	1,541,898	thr	67,150	0.00014	0.00070	0.00797	0.00067	0.00072	0.00993
roadNet-TX_adj	1,379,917	1,921,660	thr	82,869	0.00015	0.00072	0.00886	0.00073	0.00079	0.01095
roadNet-CA_adj	1,965,206	2,766,607	thr	120,676	0.00021	0.00078	0.01023	0.00105	0.00112	0.01182
as20000102_adj	6,474	12,572	wrp	6,584	0.00014	0.00110	0.00254	0.00050	0.00152	0.00407
ca-GrQc_adj†	5,242	14,484	wrp	48,260	0.00006	0.00102	0.00245	0.00014	0.00107	0.00330
p2p-Gnutella08_adj	6,301	20,777	wrp	2,383	0.00006	0.00100	0.00294	0.00014	0.00111	0.00384
oregon1_010407_adj	10,729	21,999	wrp	15,834	0.00028	0.00123	0.00319	0.00102	0.00201	0.00492
oregon1_010331_adj	10,670	22,002	wrp	17,144	0.00028	0.00122	0.00313	0.00103	0.00196	0.00469
oregon1_010414_adj	10,790	22,469	wrp	18,237	0.00029	0.00124	0.00368	0.00103	0.00197	0.00527
oregon1_010428_adj	10,886	22,493	wrp	17,645	0.00029	0.00126	0.00369	0.00105	0.00199	0.00527
oregon1_010505_adj	10,943	22,607	wrp	17,597	0.00031	0.00148	0.00436	0.00104	0.00197	0.00525
oregon1_010512_adj	11,011	22,677	wrp	17,598	0.00030	0.00124	0.00366	0.00105	0.00217	0.00602
oregon1_010519_adj	11,051	22,724	wrp	17,677	0.00029	0.00124	0.00366	0.00105	0.00199	0.00529
oregon1_010421_adj	10,859	22,747	wrp	19,108	0.00029	0.00123	0.00364	0.00105	0.00199	0.00526
oregon1_010526_adj	11,174	23,409	wrp	19,894	0.00029	0.00124	0.00367	0.00105	0.00204	0.00532
ca-HepTh_adj†	9,877	25,973	wrp	28,339	0.00006	0.00101	0.00342	0.00015	0.00110	0.00437
p2p-Gnutella09_adj	8,114	26,013	wrp	2,354	0.00007	0.00102	0.00343	0.00017	0.00164	0.00494
oregon2_010407_adj	10,981	30,855	wrp	78,138	0.00030	0.00133	0.00343	0.00106	0.00201	0.00481
oregon2_010505_adj	11,157	30,943	wrp	72,182	0.00032	0.00128	0.00339	0.00105	0.00198	0.00477
oregon2_010331_adj	10,900	31,180	wrp	82,856	0.00032	0.00209	0.00557	0.00105	0.00200	0.00479
oregon2_010512_adj	11,260	31,303	wrp	72,866	0.00031	0.00127	0.00321	0.00108	0.00203	0.00493
oregon2_010428_adj	11,113	31,434	wrp	78,000	0.00030	0.00121	0.00314	0.00107	0.00201	0.00483
p2p-Gnutella06_adj	8,717	31,525	wrp	1,142	0.00006	0.00100	0.00293	0.00015	0.00161	0.00444
oregon2_010421_adj	11,080	31,538	wrp	82,129	0.00031	0.00124	0.00317	0.00108	0.00306	0.00873
oregon2_010414_adj	11,019	31,761	wrp	88,905	0.00030	0.00124	0.00319	0.00105	0.00198	0.00478
p2p-Gnutella05_adj	8,846	31,839	wrp	1,112	0.00006	0.00099	0.00293	0.00015	0.00161	0.00440
oregon2_010519_adj	11,375	32,287	wrp	83,709	0.00032	0.00126	0.00273	0.00111	0.00201	0.00486
oregon2_010526_adj	11,461	32,730	wrp	89,541	0.00032	0.00128	0.00279	0.00111	0.00208	0.00504
p2p-Gnutella04_adj	10,876	39,994	wrp	934	0.00006	0.00100	0.00247	0.00015	0.00109	0.00394
as-caida20071105_adj	26,475	53,381	wrp	36,365	0.00037	0.00131	0.00278	0.00130	0.00280	0.00566
p2p-Gnutella25_adj	22,687	54,705	wrp	806	0.00007	0.00102	0.00250	0.00019	0.00166	0.00407
p2p-Gnutella24_adj	26,518	65,369	wrp	986	0.00010	0.00106	0.00254	0.00034	0.00181	0.00423
p2p-Gnutella30_adj	36,682	88,328	wrp	1,590	0.00008	0.00157	0.00354	0.00025	0.00173	0.00504
ca-CondMat_adj	23,133	93,439	wrp	173,361	0.00013	0.00162	0.00361	0.00033	0.00183	0.00522
p2p-Gnutella31_adj	62,586	147,892	wrp	2,024	0.00011	0.00107	0.00421	0.00037	0.00186	0.00636
email-Enron_adj†	36,692	183,831	wrp	727,044	0.00064	0.00233	0.00662	0.00122	0.00271	0.00778
loc-brightkite_edges_adj†	58,228	214,078	wrp	494,728	0.00055	0.00151	0.00471	0.00138	0.00288	0.00791
email-EuAll_adj	265,214	364,481	wrp	267,313	0.00138	0.00302	0.00554	0.00304	0.00513	0.01043
soc-Epinions1_adj*†	75,879	405,740	wrp	1,624,481	0.00164	0.00261	0.00562	0.00360	0.00514	0.00893
soc-Slashdot0811_adj†	77,360	469,180	wrp	551,724	0.00134	0.00231	0.00607	0.00326	0.00478	0.01088
soc-Slashdot0902_adj	82,168	504,230	wrp	602,592	0.00144	0.00292	0.00628	0.00325	0.00491	0.01163
amazon0302_adj	262,111	899,792	wrp	717,719	0.00049	0.00161	0.00743	0.00202	0.00376	0.01188
loc-gowalla_edges_adj	196,591	950,327	wrp	2,273,138	0.00357	0.00464	0.00848	0.00998	0.01177	0.01976
amazon0312_adj	400,727	2,349,869	wrp	3,686,467	0.00230	0.00352	0.01260	0.00760	0.00937	0.02045
amazon0505_adj	410,236	2,439,437	wrp	3,951,063	0.00246	0.00412	0.01324	0.00839	0.01016	0.02129
amazon0601_adj	403,394	2,443,408	wrp	3,986,507	0.00233	0.00404	0.01325	0.00809	0.00987	0.02104
cit-Patents_adj	3,774,768	16,518,947	wrp	7,515,023	0.02509	0.02864	0.05120	0.05281	0.05687	0.08311
facebook_combined_adj	4,039	88,234	bsh	1,612,010	0.00024	0.00089	0.00248	0.00038	0.00042	0.00274
ca-HepPh_adj	12,008	118,489	bsh	3,358,499	0.00042	0.00044	0.00296	0.00066	0.00125	0.00528
ca-AstroPh_adj*	18,772	198,050	bsh	1,351,441	0.00049	0.00051	0.00418	0.00080	0.00139	0.00636
cit-HepTh_adj*	27,770	352,285	bsh	1,478,735	0.00110	0.00112	0.00581	0.00177	0.00236	0.00826
cit-HepPh_adj*	34,546	420,877	bsh	1,276,868	0.00081	0.00084	0.00457	0.00136	0.00194	0.00787
flickrEdges_adj	105,938	2,316,948	bsh	107,987,357	0.01020	0.01022	0.01768	0.02143	0.02204	0.03046
graph500-scale18-ef16_adj	174,147	3,800,348	blk	82,287,285	0.03978	0.05027	0.05966	0.10601	0.11769	0.12870
graph500-scale19-ef16_adj	335,318	7,729,675	blk	186,288,972	0.09777	0.13531	0.15047	0.24779	0.28195	0.29859
graph500-scale20-ef16_adj	645,820	15,680,861	blk	419,349,784	0.26153	0.31769	0.34117	0.56761	0.60441	0.62928
graph500-scale21-ef16_adj	1,243,072	31,731,650	blk	935,100,883	0.75220	0.80441	0.84301	1.62747	1.66195	1.70387
graph500-scale22-ef16_adj	2,393,285	64,097,004	blk	2,067,392,370	1.89169	1.94334	2.01990	3.83588	3.87000	3.94891
graph500-scale23-ef16_adj	4,606,314	129,250,705	blk	4,549,133,002	4.86372	4.91430	5.06730	9.26852	9.30181	9.45599
graph500-scale24-ef16_adj	8,860,450	260,261,843	blk	9,936,161,560	14.7876	14.8371	15.1397	26.4003	26.4351	26.7380
graph500-scale25-ef16_adj	17,043,780	523,467,448	blk	21,575,375,802	37.8414	37.8871	56.2378	63.6145	63.6427	78.0165
twitter_rv.net	61,578,415	1,468,365,182	blk	34,824,916,864	147.441	147.462	196.080	213.662	213.685	260.993

*graphs for which the heuristic chooses a suboptimal kernel for triangle counting.
†graphs for which the heuristic chooses a suboptimal kernel for k-truss decomposition.

Table I

PERFORMANCE MEASURES OF OUR GPU IMPLEMENTATIONS OF TRIANGLE COUNTING AND K-TRUSS ON DIFFERENT INPUT GRAPHS. FOR EACH GRAPH ARE REPORTED, FROM LEFT TO RIGHT: THE NUMBER OF VERTICES AND EDGES, THE TYPE OF KERNEL SELECTED BY THE HEURISTIC, THREE TIMINGS FOR TRIANGLE COUNTING, AND THREE TIMINGS FOR K-TRUSS. FOR BOTH ALGORITHMS ARE REPORTED: *(i)* THE KERNEL TIMING, WHICH INCLUDES ALL THE KERNEL CALLS REQUIRED BY THE ALGORITHM (FOR K-TRUSS IT COINCIDES WITH ALGORITHM 3); *(ii)* THE PROCESS TIMING, WHICH INCLUDES THE KERNEL SELECTION PROCESS, THE ALLOCATION, INITIALIZATION AND DEALLOCATION OF BOTH THE BITMAPS AND THE TEMPORARY RESULTS ARRAYS, AND THE KERNEL TIMING; *(iii)* THE TOTAL TIME, WHICH INCLUDES THE REMOVAL OF SELF-LOOPS, THE CREATION OF THE CSR AND THE PROCESSING TIME. THE LAST LINE CONTAINS THE DATA FOR THE HUGE TWITTER GRAPH. DATASETS MARKED WITH AN ASTERISK OR A DAGGER ARE THOSE FOR WHICH THE HEURISTIC CHOSE A SUBOPTIMAL KERNEL FOR, RESPECTIVELY, THE TRIANGLE COUNTING OR THE K-TRUSS ALGORITHM.

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

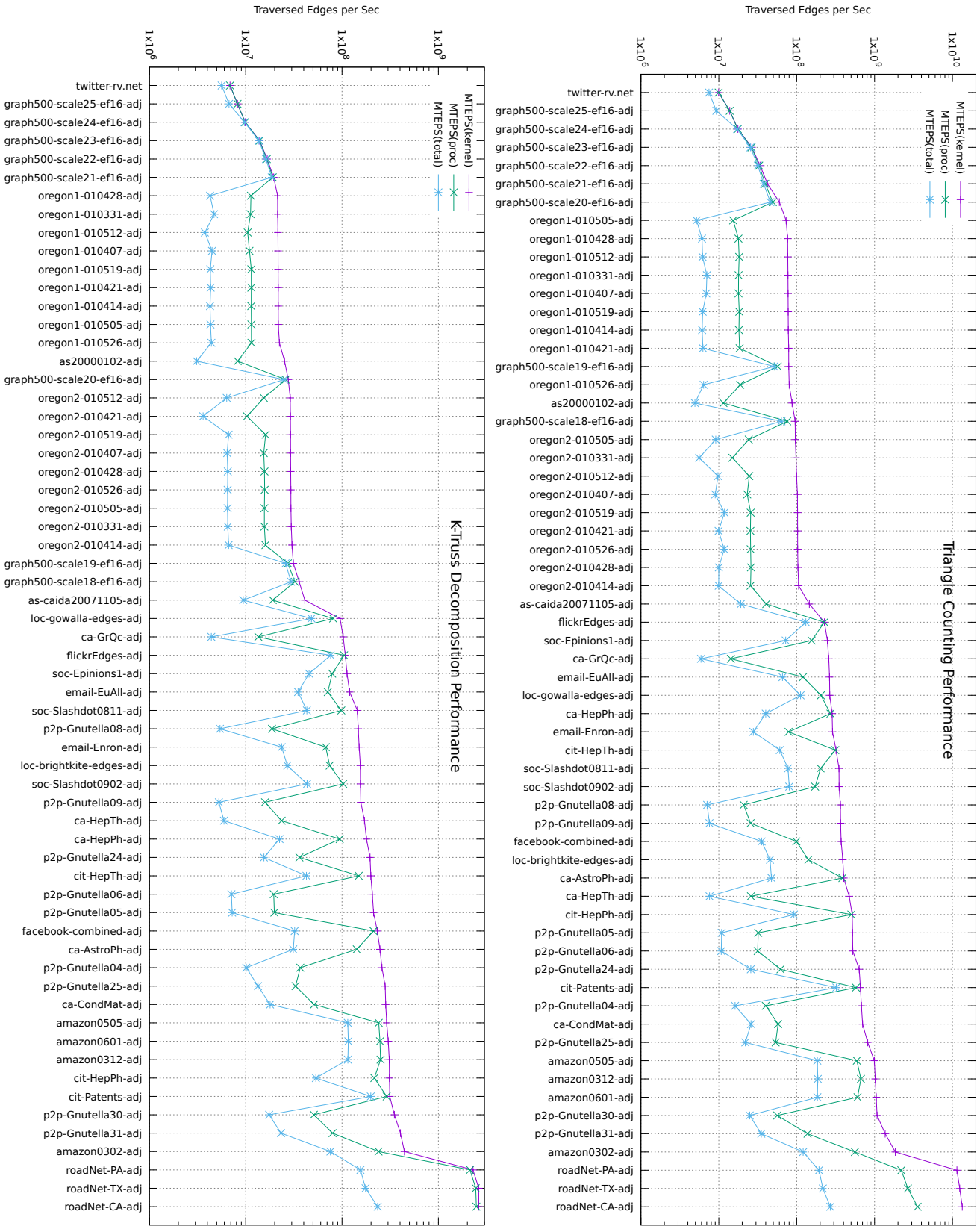

Figure 6. Traversed edges per second for triangle counting and k-truss (k=3) computations for all the graphs included in the Graph Challenge.

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