UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT

ECE508 MANYCORE PARALLEL ALGORITHMS

K-Truss Decomposition

Instructor:
Wen-mei Hwu
Acting Department Head
Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

Submitted By:
Hengzhe Ding, hengzhe2, 2173050668
Yijian Duan, yijiand2, 2177214642
Dong Liu, dongl3, 2173050736

Contents

1 Design Overview

1.1 Background

Graphs are important components in our daily life. From abstract social network to concrete traffic network, graphs have been seen everywhere. Therefore, it is quite useful to find those densely connected sets of vertices and edges, or cohesive subgraphs in those existed graphs.

Although there are already many algorithms related to the computation of cohesive subgrpahs, such as n-clique[1] and k-plex[2], most problems of computing the most cohesive subgraphs are NP-hard problems. Compared to those algorithms, there also exists a polynomial time algorithm for computing K-Truss, which can be applied to deal with the daily problems nowadays.

1.1.1 K-Truss

K-Truss is defined as a subgraph with at least k-2 triangles on every edge, which can measure cohesiveness of community and cluster coefficient, etc. Here the triangle means the cycle of length tree[3]. K-Truss can be applied to the high collaboration subnetwork identification, visualization of large-scale networks, analysis of network connectivity and maximal clique finding[4]. In the implementation of k-truss algorithm, the triangles and trusses can be enumerated in polynomial time, which can be quite useful for today's large network graphs like social networks and traffic networks. It can help us analyze the community network connection using the K-Truss decomposition algorithm. In our implementation, we would like to select K-Truss decomposition as our final project and to take the method of parallel and high performance system to handle the K-Truss decomposition algorithm.

1.1.2 Parallel Programming Basics

K-Truss decomposition is built based upon triangle counting. Triangle counting may need compaction design like SPMV with CSR/ELL/COO/JDS format. The counting process may need intersection strategy to find the common node for triangle or other advanced searching algorithms like binary search. To increase the throughput, techniques like thread coarsening and joint tiling may also be applied during the decomposition. For update, short update and long update can be used for different steps. Besides all above, selecting appropriate data structure is vital like indexing edge instead of nodes for edge centric with edge list as well as dynamic graph structures. All above should ensure the generalization of various scenarios with different data specification.

1.2 Objective

Our goal is to firstly implement a CPU version of K-Truss decomposition and then turn it into CUDA version, where we can apply many optimizations to our kernel code.

To perform K-Truss decomposition, first we need to read in the graph data, and then delete the edges with TC < (K-2) recursively along with update till no edge left in the graph.

In this project, we first read materials and papers about some up-to-date K-Truss algorithms, choose one algorithm and implement it. After the implementation, we made several changes to the code and added some optimizations to see whether the optimizations have a better efficiency and GPU utilization. Finally, according to what we have got in the results, we did analysis on our implementation to find out whether it can be improved in the future.

1.3 Challenges

During our implementation process, there are many obstacles and challenges we need to face. Firstly, most common network nowadays are quite large networks, with millions of vertices and hundred millions of edges, which means the parallel techniques applied to computation of K-Truss decomposition will apparently affect the performance of our algorithms. Secondly, the cost and limitations of computation in parallel algorithms is a significant factor as well since computation resources are limited. It is important for us to design our parallel algorithms when considering the CUDA limitations. Also, data structures are also crucial factors which will influence our design since we need to take the original choice of data structure in the code into consideration.

2 Implementation

2.1 Data Structure

Like triangle counting, we assign each thread to access each edge. Data structure we use is shown in Table 1.

Variable name	**Data structure**	**Size**	**Description**
edgeSrc	int array	# of edges	Graph array, source of each edge, sorted
edgeDst	int array	# of edges	Graph array, destination of each edge
rowPtr	int array	# of nodes + 1	Graph array, row pointer, in CSR format
affected	int array	# of edges	Status array, set 1 if affected, -1 if not
to_delete	int array	# of edges	Status array, set 1 if need to be deleted, -1 if not
e_aff	int array	# of edges	Status array, "middleman" of affected array

Table 1: Data Structure

For the easiness of implementation, we set most of data structure to be array of the same size, number of edges, to achieve this, we made some modification on algorithm mentioned in [5], detailed explanation on our algorithm is in section 2.2.

2.2 Algorithm & Implementation

The pseudo code of the algorithm we used is shown below.

Algorithm 1 k-truss decomposition

Input: $G = (V, E)$
Output: k-truss for $3 \le k \le k_{max}$
1: $k \leftarrow 3$
2: **while** true **do**
3:     Mark all $e \in E$ as “affected"
4:     **while** true **do**
5:         $E_{aff} \leftarrow Select(E,$ “affected" and “valid")
6:         **if** E_{aff} is empty **then**
7:             Break
8:         **end if**
9:         Mark all $e \in E$ as “unaffected"
10:         **for** $e = (u, v) \in E_{aff}$ **do**
11:             $tc \leftarrow |adj(u) \cap adj(v)|$
12:             **if** $tc < k - 2$ **then**
13:                 Mark (u, v) and (v, u) as “delete"
14:                 $W \leftarrow adj(u) \cap adj(v)$
15:                 **for** $w \in W$ **do**
16:                     Mark (u, w), (w, u), (v, w) and (w, v) as “affected"
17:                 **end for**
18:             **end if**
19:         **end for**
20:         **for** $e = (u, v) \in E$ **do**
21:             **if** e labeled “deleted" **then**
22:                 $u \leftarrow -1, v \leftarrow -1$
23:             **end if**
24:         **end for**
25:     **end while**
26:     $countEdge \leftarrow count(E,$ “not deleted")
27:     **if** $countEdge > 0$ **then**
28:         $k \leftarrow k + 1$
29:     **else**
30:         Break
31:     **end if**
32: **end while**

Our algorithm is mainly based on [5]. Instead of using streaming compaction to overwrite the *edgeSrc* and *edgeDst* array (**long update** mentioned in [5]), we only do the *short update* and add some extra conditions in the inner loop to determine whether an edge accessed by a thread is a valid edge in the current sub-graph (line 5). At the end of inner loop, we count the number of valid edges in current sub-graph and decide whether to break the loop (line 27 to 30).

With the removal of streaming compacting, every kernel launched will have the same number of threads accessing data.

As for CUDA implementation, we wrote 6 kernels to help accelerate the algorithm, kernel name and lines of pseudo code it corresponding to is shown in Table. 2.

Kernel name	Line of pseudo code	Description
mark	line 3	mark all edges "affected"
selectAff	line 5	select and mark "affected" edges
markAll	line 9	mark all edges "unaffected"
checkAffectedEdges	line 10 to 19	do triangle counting on all "affected" edges & mark relevant ones "deleted" or "affected"
shortUpdate	line 20 to 24	set -1 to src and dst of "deleted" edges
countEdges	line 26	count number of edges remained in the sub-graph

Table 2: Kernel Specification

3 Performance and Result Analysis

We have experimented 2 datasets with several methods. The simpler one is 8-node graph shown in Figure. 1 and another big dataset is “roadNet-CA_adj“. The parameters are shown in Table. 3.

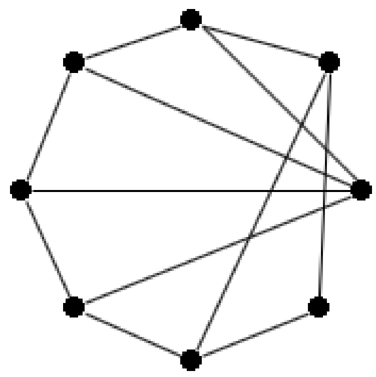

Figure 1: 8-node Network

	# edges	# nodes
8-node	12	8
roadNet-CA	2766607	1965207

Table 3: Datasets Parameters

All tests are completed on *NVIDIA TITAN V*.

[0] TITAN V

GPU UUID	GPU-7e599931-52b7-b725-3a9...
▼Duration	
Session	1.55168 s (1,551,682,830 ns)
▼Attributes	
Compute Capability	7.0
▼Maximums	
Threads per Block	1024
Threads per Multiprocessor	2048
Shared Memory per Block	48 KiB
Shared Memory per Multiprocessor	96 KiB
Registers per Block	65536
Registers per Multiprocessor	65536
Grid Dimensions	[2147483647, 65535, 65535]
Block Dimensions	[1024, 1024, 64]
Warps per Multiprocessor	64
Blocks per Multiprocessor	32
Half Precision FLOP/s	29.798 TeraFLOP/s
Single Precision FLOP/s	14.899 TeraFLOP/s
Double Precision FLOP/s	7.45 TeraFLOP/s

[0] TITAN V

▼Multiprocessor	
Multiprocessors	80
Clock Rate	1.455 GHz
Concurrent Kernel	true
Max IPC	4
Threads per Warp	32
▼Memory	
Global Memory Bandwidth	652.8 GB/s
Global Memory Size	11.752 GiB
Constant Memory Size	64 KiB
L2 Cache Size	4.5 MiB
Memcpy Engines	7
▼PCIe	
Generation	3
Link Rate	8 Gbit/s
Link Width	16

Figure 2: NVIDIA TITAN V Properties

3.1 Result

The result we obtained from the implementation specified above is shown in Table. 4 5.

triangle counts	0	1
# edges	2	10
percentage	16.67%	83.33%
k-truss	2-truss	3-truss

Table 4: K-Truss Decomposition Results for 8-node Graph

The 8-node graph is 3-truss with low cohesiveness and network connectivity.

triangle counts	0	1	2
# edges	2406797	359558	252
percentage	86.99%	13.00%	0.01%
k-truss	2-truss	3-truss	4-truss

Table 5: K-Truss Decomposition Results for roadNet-CA

The road network in California is 4-truss meaning the cities in California are regularly linked by roads with common cohesiveness. Compared with the social network, road network's cohesiveness is relatively low because it is impossible to construct numerous roads among a large set of cities. Too many roads will cause waste of transportation capacity and high cost.

3.2 Performance

We focused our experiments on 3 different implementations:

1. **CPU**:
   The original sequential implementation. Variables except *edgeSrc*, *edgeDst*, and *rowPtr* are all pangolin::Vector<int> type.
2. **Primary GPU**:
   The first GPU version with only 2 kernels we thought critical: **markAll** and **checkAffectedEdges**

3. **Optimized GPU**:
   The final GPU version just as the specification of Implementation section above.

4. **(Compacted GPU)**:
   Our original design of compacted GPU algorithm is to implemented an adjusted use **long update** with streaming compaction. First, **scan** kernel to locate the non-deleted edge indices. Then perform **long update** for *edgeSrc*, *edgeDst*. Finally generate *rowPtr* by aggregation [6].

   Due to large overhead of scan, memory update, and *rowPtr* aggregation, we did not test it with the datasets.

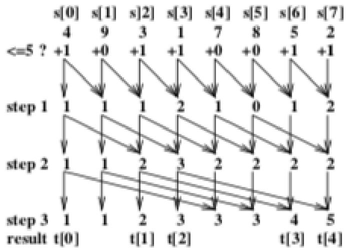

Figure 3: Scan [6]

3.2.1 Time Consumption

The time consumption averaged from 5 tests in case of random error, shown in Table. 6 and Figure. 4, 5. Due to unbalanced scales of two datasets with large difference in size, the figures are demonstrated separately.

	CPU	**Primary GPU**	**Optimized GPU**
8-node [ms]	1.264608	1.729024	1.93104
roadNet-CA [ms]	74785.1016	790.46582	16.978945

Table 6: Time Consumption

As shown in the Figure 4, 5, when the size of dataset is very small, the cost of serialization is small and overhead of parallelism is large so that the more optimized method consumes more time. Time performance of **CPU** is about 2 times better than **Optimized GPU** for 8-node graph with 12 edges.

When size of dataset exceeds certain threshold, the cost of overhead is neutralized by the optimization of parallelism. As size keeps growing larger, the time cost of **CPU** will grow at the rate of $O(n^2)$. The threshold of number of edges is about 100, exceeding which will make **Optimized GPU** a better choice.

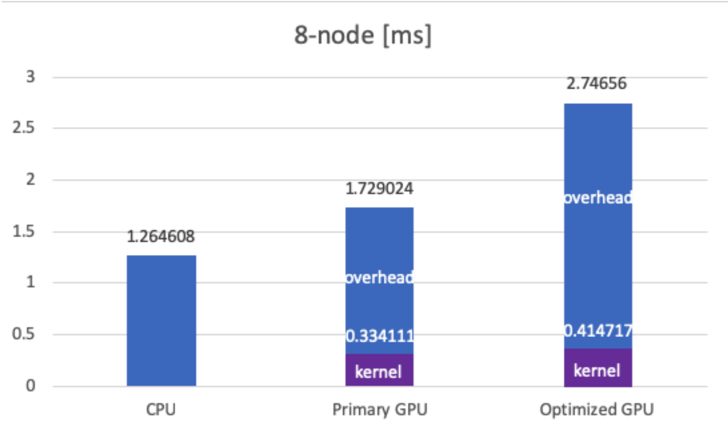

Figure 4: Time Consumption for 8-node Graph

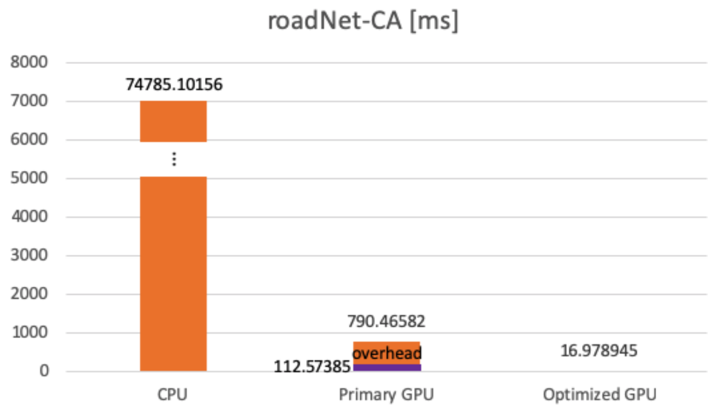

Figure 5: Time Consumption for roadNet-CA

3.2.2 *NVIDIA Visual Profiler*

We only look into the *CUDA* part of **Primary GPU** and **Optimized GPU** algorithms' *nvprof* files including **MemCpy(DtoD)** and **Kernel computation** for roadNet-CA dataset as shown in Figure. 8, 9 and Table. 7.

	Primary GPU	**Optimized GPU**
MemCpy(DtoD) invocations	27	3
MemCpy(DtoD) time	63.04606	8.46917
Total Bytes [MB]	469.139	51.127
Avg. throughput [GB/s]	7.441	6.155
Kernel invocations	12	33
Kernel computation time [ms]	112.57385	4.52873

Table 7: *nvprof* General Properties

Notice that the reason **MemCpy** is called from device to device is that the dataset is input and read into managed memory which is "accessible from all CPUs and GPUs in the system as a single, coherent memory image with a common address space" [7]. Therefore, the *view*

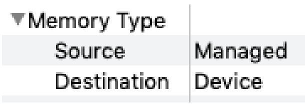

Figure 6: MemCpy

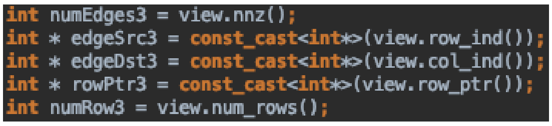

Figure 7: roadNet-CA Input

of roadNet-CA dataset we feed into main function is stored in managed memory rather than host global memory as shown in Figure. 6, 7.

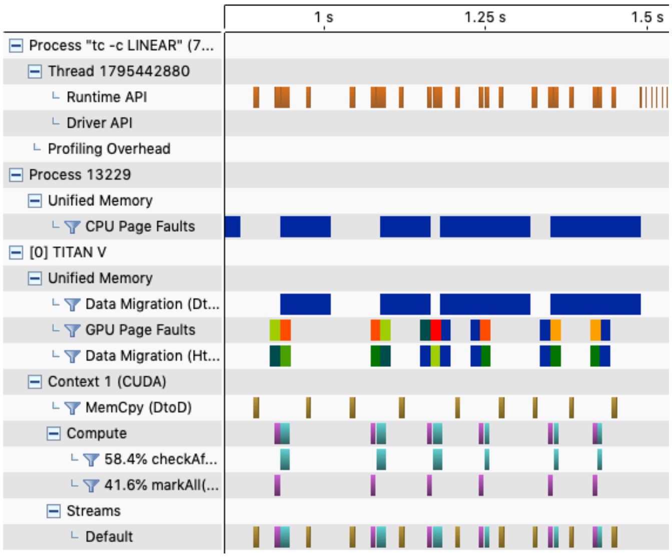

Figure 8: Primary GPU *nvprof*

As indicated in Table. 7, in **Primary GPU**, due to the integration of CPU loops and launched kernels, the variables need to be copied from device to host and vice versa repeatedly. This causes unnecessary memory transmission and large global memory access latency as well. Though using managed memory that is accessible by both device and host, the cost and overhead is still high when network size is large.

Optimized GPU has much less computation time due to high level of parallelism for large dataset with all the 6 kernels invoked 33 times in total unrolling all the rest of sequential code in **Primary GPU**.

- **Kernel Computation Detail**

The details for each kernel during computation is shown in Table. 8, 9.

Primary GPU	importance	invocations	duration [ms]
checkAffectedEdges	58.40%	6	65.76294
markAll	41.60%	6	46.81091

Table 8: Primary GPU Kernel Computation Details

For **Primary GPU**, the importance of the two kernels are very close. However, the **markAll** kernel is one kind of the simplest kernels that setting all elements to one given value indepen-

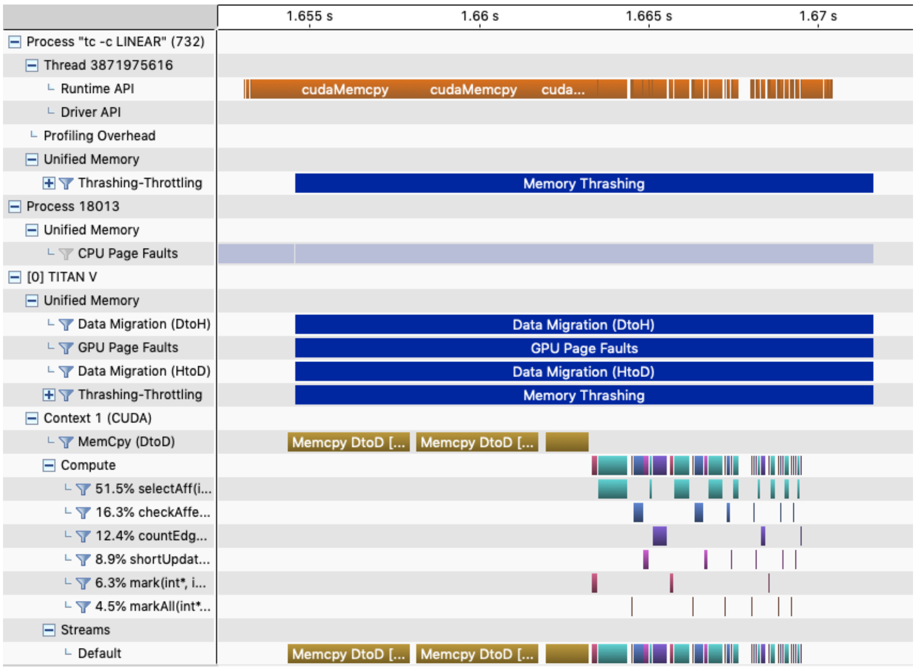

Figure 9: Optimized GPU *nvprof*

dently. Based upon this comparison, we can say our **checkAffectedEdges** is highly optimized in **Primary GPU** with **triangle_counts_set** function and linear research check algorithm.

Optimized GPU	importance	invocations	duration [us]
selectAff	50.30%	9	2275.69
checkAffectedEdges	16.80%	6	0.761179
countEdges	12.80%	3	0.578427
shortUpdate	9.20%	6	0.416028
mark	6.40%	3	0.291517
markAll	4.50%	6	0.205885

Table 9: Optimized GPU Kernel Computation Details

For **Optimized GPU**, the most significant kernel is **selectAff**. Main reason is the atomic operation under large size of array. The size of array is not changed during the process because we do not use streaming compaction but flag the deleted the edge in the array. Therefore, such kernel will perform poorly even compared with sequential code in extreme situation with high atomic operation latency.

- **GPU Usage**

Indicated from Table. 10.

1. **Kernel / Memcpy Efficiency**:
   Primary GPU's efficiency is higher meaning that unit time of **MemCpy** contributes more computation. However, both **MemCpy** time and kernel computation time for **Primary GPU** is much longer than **Optimized GPU**. The efficiency of **Optimized GPU** is low because the computation time is much lower. If the dataset is even larger, then

	Primary GPU	**Optimized GPU**
Kernel / Memcpy Efficiency	112.57/63.05 = 1.785	4.53/8.47 = 0.535
Memcpy / Kernel Overlap	0%	0%
Kernel Concurrency	0%	0%
Compute Utilization	112.57/1551.68 = 7.3%	4.53/771.77 = 0.6%

Table 10: GPU Usage

efficiency of **Optimized GPU** will increase.

2. **Compute Utilization**:
   The API calling cost and profiling overhead of both methods are close. **Primary GPU**'s is higher simple because it takes longer computation time with less parallelism. In other words, this will also benefit **Primary GPU** for small dataset.

3. **Memcpy / Kernel Overlap & Compute Utilization**:
   The results conform with our algorithm design because we avoid using any overlap and concurrent **MemCpy** or kernel to accelerate the performance due to loosing control of the data synchronization barrier.

3.3 Analysis

Some details have already been discussed in previous sections.

1. **CPU**
   Since variables except *edgeSrc*, *edgeDst*, and *rowPtr* are all pangolin::Vector<int> type, by calling *push_back* in some variables, the memory size is slightly smaller. Perform slightly better for very small network with less than about 100 nodes and significantly increase time cost when network growing larger and more densely connected.

2. **Primary GPU**
   This GPU algorithm version is in between **CPU** and **Optimized GPU** version since it only implements 2 kernels. The overhead is smaller than **Optimized GPU** and the level of parallelism is higher than **CPU**.

   Repeat **MemCpy** due to integration of CPU loops and launched kernels, causes unnecessary memory transmission and large global memory access latency.

   Highly optimized **checkAffectedEdges** with high level of parallelism has good time performance.

3. **Optimized GPU**
   This GPU version uses most kernels and has largest overhead. However, when the dataset size is large enough, the overhead is neglected and parallelism takes advantage.

4. **Compacted GPU**
   The overhead of *scan* kernel for compaction we tested is about 40ms based on **Optimized GPU** version in roadNet-CA dataset. After we implemented this, we decided to

not continue trying the following *edgeSrc*, *edgeDst* **long update** and *rowPtr* generation since the latter steps will cause more memory cost and overhead as well.

4 Discussion

1. In the beginning of our experiments, we tried streaming compaction to further optimized our algorithm. However, after we implemented **scan** kernel to locate each non-deleted edge index in previous k-truss, we found both time overhead and memory are worse than only counting the non-deleted edge. This streaming compaction design therefore is definitely not optimized.
2. Since we have 2 infinite **while(true)** loop till break, developing the algorithm needs very careful exit catch in case of burning out the hardware capacity.

4.1 Possible Improvements

1. In addition to linear search, using binary search in finding affected edges may reduce the complexity to $O(\log n)$.
2. The **selectAff** kernel is mainly limited by atomic addition with very high latency with large dataset. Instead of using atomic operation, maybe prefix-sum scan would perform better.
3. Our design of streaming compaction does not contribute to the algorithm positively. However, compaction does save memory space in some ways as its purpose. Maybe a better algorithm design could be further developed.
4. The datase of roadNet-CA contains 2,766,607 edges and 1,965,207 nodes. However, an overwhelming database may exceed capacity of *NVIDIA TITAN V*'s on-chip memory. Therefore, either global memory access will be called or further distributed parallel algorithm needs to be developed.

5 Conclusion

In conclusion, we successfully designed and realized our K-Truss decomposition algorithm based on pseudocode from [5] with correct decomposition result. The correctness of the algorithm enable us to further deal with any problem relating to networks in cohesiveness and connectivity like road network or social network.

We also managed to optimized our algorithm based on our knowledge and techniques in parallel programming including unrolling, SPMV, privatization, linear search etc. Based upon roadNet-CA database with 2,766,607 edges and 1,965,207 nodes, our algorithm has reasonable decomposition performance within 20ms.

However, our algorithm does not fully utilize the capacity of *NVIDIA TITAN V*. Some issues like binary search could be further considered.

6 Resources

6.1 GitHub Repo

For detailed development history, see `https://github.com/nickbigeye/UIUC_SP19_ECE508_K-Truss-Decomposition`.

6.2 Reading materials

In the beginning of the project, we read some famous papers about K-Truss decomposition including [8], [9], [10], [11], and [4].

References

[1] R Duncan Luce. Connectivity and generalized cliques in sociometric group structure. *Psychometrika*, 15(2):169–190, 1950.

[2] Stephen B Seidman and Brian L Foster. A graph-theoretic generalization of the clique concept. *Journal of Mathematical sociology*, 6(1):139–154, 1978.

[3] Sitao Huang, Mohamed El-Hadedy, Cong Hao, Qin Li, Vikram S Mailthody, Ketan Date, Jinjun Xiong, Deming Chen, Rakesh Nagi, and Wen-mei Hwu. Triangle counting and truss decomposition using fpga. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[4] H. Kabir and K. Madduri. Shared-memory graph truss decomposition. *arXiv preprint-arXiv:1707.02000*, 2017.

[5] K. Feng R. Nagi J. Xiong N. S. Kim K., Date and Hwu W. M. Collaborative (cpu+ gpu) algorithms for triangle counting and truss decomposition on the minsky architecture: Static graph challenge: Subgraph isomorphism. *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[6] J. Sang V. Rego and C. Yu. A fast hybrid approach for stream compaction on gpus. *2016 Fourth International Symposium on Computing and Networking (CANDAR)*, pages 476–482, 2016.

[7] NVIDIA. Cuda toolkit documentation - 2.4. heterogeneous programming. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2019.

[8] J. Wang and J. Chen. Truss decomposition in massive networks. *Proc. VLDB Endow.*, 5(9):812–823, May 2012.

[9] C. K. Chou P. L. Chen and M. S. Che. Distributed algorithms for k-truss decomposition. *Proc. Int'l. Conf. on Big Data (Big Data)*, pages 471–480, 2014.

[10] R. A. Rossi and N. K. Ahme. The network data repository with interactive graph analytics and visualization. *AAAI Conference on Artificial Intelligence,*, page 4292–4293, 2015.

[11] C. Seshadhri A. E. Sariyuce and A. Pina. Parallel local algorithms for core, truss, and nucleus decompositions. *arXiv preprint arXiv:1704.00386*, 2017.