Bachelor Thesis
(TWIN, Honours)

Graph Challenge: Counting Triangles and Enumerating Trusses in Graphs

Elise Sänger (5665795)

Supervisor:

Prof. Dr. Rob Bisseling

January 7, 2019

Abstract

Counting triangles and enumerating trusses are fundamental graph analysis algorithms with several applications. The 2018 Static Graph Challenge sought to improve the efficiency of these algorithms, so that the time for graph analysis decreases and larger graphs can be analysed. In this thesis we compared different serial algorithms on the CPU for both triangle counting and truss enumeration. The algorithms for triangle counting can be divided into two categories: (1) linear algebra based algorithms using the adjacency matrix of a graph, and (2) graph based algorithms using the edge list and neighbourhoods of a graph. We concluded that the graph based algorithms were faster and could handle larger graphs than the linear algebra based algorithms. For enumerating trusses we concluded that treating 3-trusses as a separate case gave better results because special properties of 3-trusses could be used. We also concluded that the runtime for k-trusses did not strongly depend on k for $k > 3$.

After writing our own algorithms we looked at the champions of the 2018 Graph Challenge, as well as the results for triangle counting of the 2017 Graph Challenge. The focus of the champions was on improving the rate of edges processed per second by improving how the graph data is partitioned for parallel computing and how the graph data is stored in memory. One of the champions did this for linear algebra based algorithms on CPUs by improving the partitioning of the adjacency matrix before doing masked sparse matrix matrix multiplication. Another champion did it for a graph based algorithm on GPUs by improving how the graph partitions are stored in memory before and during processing. The highest rate of edges processed per second achieved for triangle counting in the 2018 Graph Challenge was $4.7 \cdot 10^9$ edges per second, which is approximately a factor 10^4 higher than the rate that we achieved for the same graph. The largest graph processed for triangle counting in the 2017 Graph Challenge has $2.7 \cdot 10^{11}$ edges, which is approximately a factor 10^5 larger than our largest graph. These numbers are not comparable due to differences in available computing power.

Contents

1 Introduction **2**
1.1 The Challenge 2
1.2 Applications 2
1.3 Graphs 3

2 Triangles **5**
2.1 Algorithms 5
2.2 Results 11
2.3 Conclusion 14

3 Trusses **15**
3.1 Algorithms 17
3.2 Results 20
3.3 Conclusion 22

4 Graph Challenge Champions **23**
4.1 The 2017 Graph Challenge 23
4.2 Fast Triangle Counting Using Cilk 24
4.3 High-Performance Triangle Counting on GPUs 24

5 Concluding Remarks **26**

A Python Code for Triangle Counting **27**

B Python Code for Truss Enumeration **32**

1. Introduction

1.1 The Challenge

The amount of data collected and analysed with graph analytics is large and increasing rapidly. There is also an increasing need to analyse evolving graphs in real-time. The structure of graph data can make current graph analytics inefficient. The size of the graphs that can be processed is limited by the size of the main memory and by the amount of processing required [1]. A new graph analytics system needs to be developed to enable the possibility of analysing larger graphs or more complex questions that require more processing. The Graph Challenge has as a goal to make advancements in graph analytics systems. The main focus is to accelerate and expand graph analytics algorithms by improving algorithms and their implementations [2].

The 2018 Graph Challenge consisted of two challenges:

- Static Graph Challenge: Subgraph Isomorphism, and
- Streaming Graph Challenge: Stochastic Block Partition.

This thesis focuses on the Static Graph Challenge. The Subgraph Isomorphism Problem is the computational problem of determining whether a graph G contains a subgraph that is isomorphic to a graph H. This problem is known to be NP-complete, since the subproblem of finding k-cliques is NP-complete [3]. However, there exist subproblems that have polynomial time complexity.

The 2018 Static Graph Challenge focuses on algorithms that find triangles and k-trusses [1]. Triangle counting can be considered as a special case of the subgraph isomorphism problem with a triangle as the subgraph of interest. Algorithms that find triangles do not find general matching subgraphs, but can be a part of algorithms that do. Algorithms that find k-trusses can be used for characterisation of (sub)graphs. If k-truss features of two (sub)graphs are inconsistent this proves that these (sub)graphs are not isomorphic. Consistent k-truss features indicate that additional examination is required to determine if an isomorphism exists. Improving the efficiency of triangle counting and k-truss algorithms may therefore lead to improvements in efficiency of subgraph isomorphism algorithms and was therefore the goal of the 2018 challenge [2].

1.2 Applications

Counting triangles and determining k-trusses of graphs is usually not directly interesting in graph analytics, but they are both fundamental algorithms that may be part of more complex

algorithms used in graph analytics. Triangle counting and k-truss determination are anticipated to be useful for a fast initial analysis of the structure of large graphs and to point out regions of interest for further analysis with slower, more complex algorithms [2].

There exist many important measures of a graph that are based on triangles, for example the clustering coefficient and the transitivity ratio. The clustering coefficient is a measure for the tendency of nodes to cluster together and for how much a graph resembles a so called small-world graph. The transitivity ratio is the probability of three connected nodes to be a triangle [3].

Below are some examples of real world applications of graph analytics where triangle counting or k-trusses can be applied.

- Spam detection: Local triangle counting, where for each vertex the number of triangles it is part of is counted, can be used for detecting web spam. The distribution of the number of triangles for a host is different for nonspam and spam hosts. This is used to detect if spamming is present in large scale web graphs [4].
- Epidemiology: In preventing epidemics a fast identification of the pathogens and their source is important. During an E. coli outbreak in 2011 in Germany it took more than a month to identify the source correctly. This was too late to have a large impact on the epidemic. Improved algorithms can decrease this time significantly to possibly less than 1 day [5]. Genetic sequencing is becoming faster so biome networks are created faster and become larger. This means that improving the algorithms for analysing the biome networks is necessary for faster detection of the source of an epidemic. Triangle counting and k-trusses can be used for fast initial identification of different organisms in the biome network. This is done by "fingerprinting" of clusters in the network and comparing the results to a database of "fingerprints" [2].
- Brain mapping: MRI data of the human brain can be modelled as graphs where the nodes represent regions in the brain and edges represent interactions between the regions. Graph analytics are used to understand the structure and function of the brain and can maybe even help detecting diseases. One of the measures used is the clustering coefficient where triangle counting is important [6].

1.3 Graphs

In this section we give some definitions that are used in this thesis.

Definition. A *directed graph* is an ordered pair $G = (V, E)$ where V is a set of vertices and E a set of ordered pairs of vertices called edges denoted as (x, y). In an *undirected graph* edges are unordered pairs of vertices denoted as $\{x, y\}$. This is equivalent with $(x, y) \in E$ if and only if $(y, x) \in E$.

Definition. A *simple graph* is an undirected graph with no multiple edges or loops $\{i, i\}$.

In this thesis only simple graphs are considered.

Definition. The *adjacency matrix* A of a graph G with n_V vertices is the $n_V \times n_V$-matrix where a_{ij} is the number of edges from i to j. Note that in an undirected graph the adjacency matrix is symmetric.

Definition. In a graph G the *neighbourhood* of a vertex i are all vertices $j \in V(G)$ such that $(i, j) \in E(G)$ and it is denoted as $nbh(i)$.

Definition. A *subgraph* of a graph G is a graph H such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$.

Definition. A simple graph G and a simple graph H are called *isomorphic* if an isomorphism exists between them. An *isomorphism* from a simple graph G to a simple graph H is a bijection $f : V(G) \to V(H)$ such that $\{i, j\} \in E(G)$ if and only if $\{f(i), f(j)\} \in E(H)$.

2. Triangles

In this chapter we will compare algorithms that count the number of triangles in a simple graph. We first define a triangle in a simple graph as follows:
Definition. A triangle is a subgraph T of a simple graph G with vertex set $V(T) = \{a, b, c\} \subseteq V(G)$ and edge set $E(T) = \{\{a, b\}, \{b, c\}, \{c, a\}\} \subseteq E(G)$.

The example graph in Figure 2.1 contains 5 triangles, namely $\{0, 1, 2\}$, $\{0, 1, 3\}$, $\{0, 2, 3\}$, $\{1, 2, 3\}$ and $\{0, 1, 4\}$

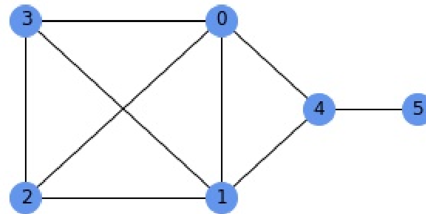

Figure 2.1: This graph contains 5 triangles.

2.1 Algorithms

The algorithms for counting triangles in this section can be divided into two types depending on the input used. The first type uses the adjacency matrix of a graph as input and the second type uses the list of edges of a graph. To be able to compare the algorithms, we use implementations that always use the list of edges as input. This choice is made because graph data is often provided in the form of a list of edges. Such a list of edges has the directed edges in it, so for undirected graphs both (i, j) and (j, i) are in the list if $\{i, j\}$ is an edge of the graph.

Turning a list of edges into the equivalent adjacency matrix is done using Algorithm 1. It follows directly from the definition of the adjacency matrix of a graph. Note that it only works for graphs with no multiple edges.

Here, $zeros(m, n)$ is an $m \times n$ matrix containing only zeros. We use $max(E)$ to denote the function $\max\{i : \{i, j\} \in E\}$. In the graph data used, the vertices are numbered from 0 to $n_V - 1$ where n_V is the number of vertices. Assuming that we have no isolated vertices we get that $n_V = max(E) + 1$.

Algorithm 1 Implementation to determine the adjacency matrix of a graph.

Input: List of edges E of a graph G.
Result: Adjacency matrix A of G.

$n_V = max(E) + 1$
$A = zeros(n_V, n_V)$
for (i, j) in E **do**
  $a_{ij} = 1$
return A

Algorithm 2 uses the adjacency matrix of a graph to determine the number of triangles in the graph. This algorithm is used as a baseline algorithm for how efficient we can calculate the number of triangles. We are going to prove that the baseline algorithm indeed gives the number of triangles in a graph G.

Algorithm 2 Baseline algorithm using the adjacency matrix of a graph.

Input: Adjacency matrix A of a graph G.
Result: Number of triangles in G.

$B = A^2 \circ A$
$n_T = \frac{1}{6} \sum_{ij} b_{ij}$
return n_T

where $\circ$ denotes element-wise multiplication

Proof. In the adjacency matrix A of a graph G the element a_{ij} is the number of edges from i to j in the graph. Thus, $a_{ik}a_{kj}$ is the number of ways to walk from i to k to j. Summing over all different vertices k gives the number of ways to walk from i to j using exactly two edges. If we multiply this with a_{ij} we get the number of triangles in which (i, j) is an edge. So $b_{ij} = a_{ij} \sum_k a_{ik}a_{kj}$ is the number of such triangles. When summing over all b_{ij} we count each triangle $\{i, j, k\}$ six times, namely once in $b_{ij}, b_{ji}, b_{ik}, b_{ki}, b_{jk}$ and b_{kj} (see Figure 2.2). We conclude that n_T is exactly the number of triangles in the graph G. □

In Algorithm 2 we counted each triangle six times and therefore had to divide by six in the end. It would be more efficient to only count each triangle once. In Algorithm 3 we use that we are only focusing on simple graphs. This means that the adjacency matrix is symmetric and that we only need to count the triangles $\{i, k, j\}$ with $i < k < j$. So the number of triangles in which $\{i, j\}$ is an edge becomes

$$a_{ij} \sum_{i<k<j} a_{ik}a_{kj}.$$

In Algorithm 3 we also use the symmetry of the adjacency matrix to switch the indices so we only use rows. This is done because in NumPy (Python) the default is row-major order, so using rows instead of columns leads to more efficient use of cache for larger graphs. We also use the fact that a_{ij} is either zero or one.

The adjacency matrix of a graph contains mostly zeros so using a sparse matrix uses less memory. In Algorithm 4 we use the list of edges to create a sparse matrix in Dictionary of Keys (DOK) format. Then we convert the sparse matrix into Compressed Sparse Row (CSR) format. This

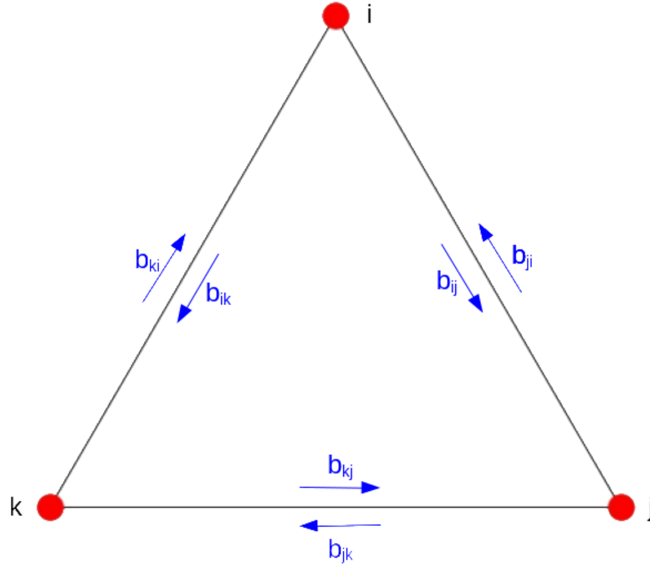

Figure 2.2: In Algorithm 2 each triangle is counted six times, once for every blue arrow.

Algorithm 3 Improved algorithm using the adjacency matrix of a graph. Note that it only works for simple graphs.

Input: Adjacency matrix A of a graph G.
Result: Number of triangles in G.

  $n_T = 0$
  for $i < j$ **do**
    if $a_{ij} \neq 0$ **then**
      $n_T = n_T + \sum_{i<k<j} a_{ik} a_{jk}$
  return n_T

---

is done because sparse matrices are more easily constructed in DOK format but arithmetic operations are more efficient in CSR format [7].

Algorithm 4 Algorithm using sparse matrices.

Input: List of edges E of a graph G.
Result: Number of triangles in G.

  $n_V = \max(E) + 1$
  $A_{DOK} =$dok_matrix(n_V, n_V)
  for (i, j) in E **do**
    $A_{DOK}(i, j) = 1$
  $A_{CSR} =$csr_matrix(A_{DOK})
  $B_{\text{CSR}} = A^2_{\text{CSR}} \circ A_{\text{CSR}}$
  $n_T = \frac{1}{6} \sum_{ij} b_{ij}$
  return n_T

  where $\circ$ denotes element-wise multiplication

The algorithms described above all use the adjacency matrix of a graph. We are now going to look at algorithms that use the neighbourhoods of the vertices of a graph. Turning a list of edges into the neighbourhoods of the vertices is done using Algorithm 5.

Algorithm 5 Implementation to determine the neighbourhoods of vertices of a graph. We call this function EDGES_TO_NBH.

Input: List of edges E of a graph G.
Result: List with neighbourhoods of all vertices of G.

```
n_E = |E|
n_V = max(E) + 1
nbh = [{}] * n_V
for (i, j) in E do
    nbh(i) = nbh(i) ∪ {j}
return nbh
```

Here, $[\{\}] * n_V$ is a list of n_V copies of an empty set. When EDGES_TO_NBH is done the i-th set in nbh is the neighbourhood of vertex i.

Algorithm 6 uses the neighbourhoods of the vertices in a graph to determine the number of triangles in the graph. We are going to prove that this algorithm indeed gives the number of triangles in a simple graph G.

Algorithm 6 Algorithm using the neighbourhoods of vertices in a graph.

Input: List of edges E of a graph G.
Result: Number of triangles in G.

```
nbh = EDGES_TO_NBH(E)
count = 0
for (i, j) in E do
    count = count + |nbh(i) ∩ nbh(j)|
n_T = count/6
return n_T
```

Proof. Let $\{i, j\} \in E$ be an edge of the graph G. A third vertex k forms a triangle with i and j if and only if k is in the neighbourhoods of both i and j. So the number of triangles in which $\{i, j\}$ is an edge is $|nbh(i) \cap nbh(j)|$. If we do this for every edge $(i, j) \in E$ we count each triangle $\{i, j, k\}$ six times, namely once for each of the edges $(i, j), (j, i), (i, k), (k, i), (j, k)$ and (k, j). So n_T is indeed the number of triangles in the graph G. □

In Algorithm 6 we counted each triangle six times and therefore had to divide by six in the end. It would be more efficient to only count each triangle once. In Algorithm 7 we only count the number of triangles the edge (i, j) is in if $i < j$ to only count the edge $\{i, j\}$ once. We also remove an edge $\{i, j\}$ from the graph after we determined the number of triangles it is in. This is done by removing j from the neighbourhood of i and i from the neighbourhood of j.

The time complexity of determining the intersection of two sets A and B is $\mathcal{O}(\min\{|A|, |B|\})$ because in Python sets are hashed [8]. So determining $|nbh(i) \cap nbh(j)|$ is more time consuming if both $nbh(i)$ and $nbh(j)$ are large sets. When we remove an edge $\{i, j\}$ the sizes of both $nbh(i)$ and $nbh(j)$ decrease by one. So if we start with the edges $\{i, j\}$ where $\min\{|nbh(i)|, |nbh(j)|\}$ is small, then by the time we reach the larger ones they have decreased significantly provided we

Algorithm 7 Improved algorithm using the neighbourhoods of vertices in a graph. It deletes the edge after determining the number of triangles that edge is in.

Input: List of edges E of a graph G.
Result: Number of triangles in G.
  nbh =EDGES_TO_NBH(E)
  $n_T = 0$
  for (i, j) in E **do**
    if $i < j$ **then**
      $n_T = n_T + |nbh(i) \cap nbh(j)|$
      $nbh(i) = nbh(i) - \{j\}$
      $nbh(j) = nbh(j) - \{i\}$
  return n_T

can delete edges in $\mathcal{O}(1)$ time. Deleting elements from a set is done in $\mathcal{O}(1)$ time in Python so edges are deleted in $\mathcal{O}(1)$ time [8].

This idea is used in Algorithm 8. In this algorithm we sort the vertices by degree. Then we take the vertex i with the lowest degree. For this vertex i we determine the number of triangles each edge $\{i, j\}$ with $j \in nbh(i)$ is in. Then we take the vertex with the second lowest degree and do this, etc. This algorithm should be faster than Algorithm 7 which uses a more or less random order.

Algorithm 8 Algorithm using the neighbourhoods of vertices in a graph. It sorts the vertices by degree before counting the number of triangles.

Input: List of edges E of a graph G.
Result: Number of triangles in G.
  n_V =max$(E) + 1$
  nbh =EDGES_TO_NBH(E)
  p =sorted(range(n_V), key = lambda idx : $|nbh(idx)|$)
  $n_T = 0$
  for i in p **do**
    for j in $nbh(i)$ **do**
      $n_T = n_T + |nbh(i) \cap nbh(j)|$
      $nbh(i) = nbh(i) - \{j\}$
      $nbh(j) = nbh(j) - \{i\}$
  return n_T

  where the sorting function used is the degree of the vertex.

We are only dealing with simple graphs so if we have the edge (i, j) we also have the edge (j, i). This means we can delete all edges (i, j) with $i > j$ from the list of edges without losing any information. This is done in Algorithm 9. We use a mask with all the indices of the edges (i, j) with $i > j$ so we can delete them all at the same time because this is more efficient than deleting the edges one by one.

Note that now k is only in the neighbourhood of i if (i, k) is an edge of the graph and $k > i$. Let (i, j) be an edge with $i < j$. Now $|nbh(i) \cap nbh(j)|$ is the number of triangles $\{i, j, k\}$ with $i < j < k$. This means that we only count each triangle once so n_T is indeed the number of triangles of the graph.

Algorithm 9 Algorithm using the neighbourhoods of vertices in a graph. Of every edge $\{i, j\}$ it deletes (i, j) in the list of edges if $i > j$ so only (j, i) is left in the list of edges.

Input: List of edges E of a graph G.
Result: Number of triangles in G.
  n_V =max$(E) + 1$
  $n_E = |E|$
  $mask = []$
  for $i \in$ range(n_E) **do**
    if $E(i, 0) > E(i, 1)$ **then**
      $mask$.append(i)
  E =delete$(E, mask)$
  $n_E = |E|$
  nbh =EDGES_TO_NBH(E)
  $n_T = 0$
  for (i, j) in E **do**
    $n_T = n_T + |nbh(i) \cap nbh(j)|$
  return n_T

In Algorithm 10 we combine Algorithms 8 and 9. First all edges (i, j) with $i > j$ are deleted from the list of edges such as done in Algorithm 9. Then the vertices are sorted by degree such as done in Algorithm 8. Combining these improvements should make this algorithm faster than both Algorithms 8 and 9.

Algorithm 10 Algorithm using the neighbourhoods of vertices in a graph. Of every edge $\{i, j\}$ it deletes (i, j) in the list of edges if $i > j$ so only (j, i) is left in the list of edges.

Input: List of edges E of a graph G.
Result: Number of triangles in G.
  n_V =max$(E) + 1$
  $n_E = |E|$
  $mask = []$
  for $i \in$ range(n_E) **do**
    if $E(i, 0) > E(i, 1)$ **then**
      $mask$.append(i)
  E =delete$(E, mask)$
  $n_E = |E|$
  nbh =EDGES_TO_NBH(E)
  p =sorted(range(n_V), key = lambda idx : $|nbh(idx)|$)
  $n_T = 0$
  for i in p **do**
    for j in $nbh(i)$ **do**
      $n_T = n_T + |nbh(i) \cap nbh(j)|$
  return n_T

  where the sorting function used is the degree of the vertex.

Algorithm 11 is divided in three parts where the first and last part can be recognized from Algorithm 9. The second part is based on the idea described above that sorting the edges should make the algorithm more efficient. We sort the edges $\{i, j\}$ by $\min\{|nbh(i)|, |nbh(j)|\}$ which we

call the edge degree. The implementation is based on the counting sort algorithm. Here, *deg* is the list with edges and their degrees and E_s is the sorted list of edges.

Algorithm 11 Algorithm using the neighbourhoods of vertices in a graph. It sorts the edges by minimum degree of the endpoints before counting the number of triangles.

Input: List of edges E of a graph G.
Result: Number of triangles in G.

```
n_V =max(E) + 1
n_E = |E|
mask = []
for k ∈ range(n_E) do
    i, j = E(k)
    if i > j then
        mask.append(k)
E =delete(E, mask)
n_E = |E|
nbh =EDGES_TO_NBH(E)

deg = zeros(n_E, 3)
for k ∈ range(n_E) do
    i, j = E(k)
    deg(k) = (i, j, min(|nbh(i)|, |nbh(j)|))

{Below a counting sort is done on the edge degree.}
count = [0] ∗ n_V
for a in deg(:, 2) do
    count(a)+=1
running_count = [0] ∗ n_V
for i ∈ range(1, n_V) do
    running_count(i) = running_count(i − 1) + count(i)
E_s =zeros((n_E, 3))
for a ∈ range(n_E) do
    idx = running_count(deg(a, 2)) − 1
    E_s(idx) = deg(a)
    running_count(deg(a, 2))-=1

n_T = 0
for (i, j) in E_s do
    n_T = n_T + |nbh(i) ∩ nbh(j)|
return n_T
```

2.2 Results

The implementations are written in Spyder 3.2.8 using Python 3.6.5. The processor used is Intel® Core™ i5-7200U CPU @ 2.50 GHz with 4 cores of which one is used for the computations. The size of the main memory is 8 GB and the size of the cache is 3 MB.

The graph data used for counting triangles is taken from the Stanford Large Network Dataset Collection [9]. The characteristics of the graphs can be found in Table 2.1. Here n_V is the number of vertices, n_E is the number of edges, $d_{avg} = 2n_E/n_V$ the average degree of the vertices and n_T the number of triangles in the graph. These graphs are all simple graphs, i.e. the graphs are undirected and do not contain loops or multiple edges.

Table 2.1: Characteristics of the graphs that were used.

Graph	n_V	n_E	d_{avg}	n_T	Description
0edges	348	2519	14.5	10740	Social circles from Facebook
tvshow	3892	17262	8.87	29092	Gemsec Facebook dataset
athletes	13866	86858	12.5	140537	Gemsec Facebook dataset
new_sites	27917	206259	14.8	131010	Gemsec Facebook dataset
Brightkite_edges	58228	214078	7.35	494728	Friendship network of Brightkite users
Gowalla_edges	196591	950327	9.67	2273138	Friendship network of Gowalla users
roadNet-PA	1090920	1541898	2.83	67150	Road network of Pennsylvania
roadNet-CA	1971281	2766607	2.81	120676	Road network of California

In Table 2.2 the runtimes for the different algorithms using the adjacency matrix of a graph can be found. The entries marked with * gave a memory error because the adjacency matrix did not fit into main memory. Sparse matrices use less memory than arrays and therefore we do not have this problem for Algorithm 4.

Table 2.2: Runtimes for the different algorithms using the adjacency matrix of the graph.

graph	runtime A2 (s)	runtime A3 (s)	runtime A4 (s)
0edges	0.0375	0.0266	0.0339
tvshow	40.9	2.01	0.188
athletes	$1.84 \cdot 10^3$	24.5	1.04
new_sites	$1.59 \cdot 10^4$	98.3	2.70
Brightkite_edges	*	*	2.65
Gowalla_edges	*	*	21.6
roadNet-PA	*	*	17.0
roadNet-CA	*	*	30.4

We notice that Algorithms 3 and 4 are always faster than the baseline algorithm (Algorithm 2) which is what we expected. We also see that Algorithm 4 is faster than Algorithm 3 for all graphs except the smallest graph `0edges`. Using sparse matrices was expected to be faster because they require less memory space than the corresponding array and they also perform fewer operations. For small matrices this difference is relatively small. Together with the extra conversion step, and therefore computation time, needed this could explain why Algorithm 3 is faster than Algorithm 4 for the graph `0edges`.

In Table 2.3 the runtimes for the different algorithms using the neighbourhoods of the vertices of a graph can be found. Each time the average of 10 runs is taken. For each graph the fastest algorithm is marked by the runtime in bold.

We notice that Algorithm 10 is the fastest algorithm for all graphs except for `roadNet-CA` where Algorithm 9 is the fastest one. We expected that sorting the vertices by degree would be faster

Table 2.3: Runtimes for the different algorithms using the neighbourhoods of the vertices of the graph.

graph	runtime A6 (s)	runtime A7 (s)	runtime A8 (s)	runtime A9 (s)	runtime A10 (s)	runtime A11 (s)
0edges	0.0145	0.00992	0.00835	0.00662	**0.00573**	0.0122
tvshow	0.0924	0.0660	0.0585	0.0424	**0.0384**	0.0794
athletes	0.502	0.393	0.356	0.221	**0.204**	0.415
new_sites	1.39	1.06	0.973	0.547	**0.510**	1.01
Brightkite_edges	1.16	0.911	0.816	0.601	**0.578**	1.10
Gowalla_edges	9.99	8.36	7.91	6.65	**6.55**	8.90
roadNet-PA	5.05	5.50	6.21	4.13	**3.92**	7.44
roadNet-CA	9.23	9.86	9.62	**6.91**	7.02	13.5

than a random order. Therefore we expected Algorithm 8 to be faster than Algorithm 7 and Algorithm 10 to be faster than Algorithm 9. We see that both hold except for the graphs `roadNet-PA` and `roadNet-CA`. These graphs both have a much lower average vertex degree ($d_{avg} = 2.8$) than the other graphs. Therefore sorting the vertices by degree leads to a smaller improvement in runtime. The sorting process also costs time and apparently it costs more time than the improvement is. This could explain why sorting is not profitable for graphs with low average vertex degree.

We expected Algorithm 7 to be faster than Algorithm 6 because it counts every triangle once instead of six times. Looking at Table 2.3 we notice that this is correct for all graphs except `roadNet-PA` and `roadNet-CA`.

We expected Algorithm 9 to be faster than Algorithm 7, and therefore expected Algorithm 10 to be faster than Algorithm 8, because we use half the number of edges. Looking at Table 2.3 we notice that this is correct for all graphs used.

We expected Algorithm 11 to be faster than Algorithm 9 because we expected that sorting the edges by edge degree would be faster than a random order. In Table 2.3 we see that this is never the case for the graphs we used. The time complexity of counting sort used on a list of length n is $\mathcal{O}(n)$. This means that sorting the edges by edge degree is $\mathcal{O}(n_E)$. The time complexity of counting the triangles as done in Algorithm 9 is $\mathcal{O}(d_{avg} \cdot n_E)$. Usually d_{avg} is relatively small, so this means that sorting the edges is relatively expensive and, apparently, more expensive than the improvement that we get from the sorting.

Sorting the vertices by vertex degree the way it is done in Algorithm 10 has time complexity $\mathcal{O}(n_V \log(n_V))$. This is typically smaller than $\mathcal{O}(n_E)$ for real-life problems, so sorting the vertices is faster than sorting the edges. Sorting the vertices can also be done by a counting sort with time complexity $\mathcal{O}(n_V)$ to make the sorting even faster. We defined the edge degree of an edge $\{i, j\}$ by $\min\{|nbh(i)|, |nbh(j)|\}$ and sorted the edges by this. We count the triangles by starting with the edge with lowest edge degree, and then continue with the edge with second lowest edge degree, etc. When sorting the vertices i by vertex degree we sort them by $|nbh(i)|$. We count the triangles by starting with the edges adjacent to the vertex with lowest degree, and then continue with the edges adjacent to the vertex with second lowest degree, etc. These two ways of sorting lead to the same order of going through the edges upon permutation of edges with the same edge degree. Therefore, the two ways of sorting are equivalent and give the same improvement in runtime. Sorting the edges costs more time than sorting the vertices so we expect Algorithm 11 to be slower than Algorithm 10. This is consistent with the results from Table 2.3.

When we compare Table 2.2 with Table 2.3 we notice that all the algorithms using the neighbourhoods of the vertices of a graph are faster than all the algorithms using the adjacency matrix of a graph. This could be because less memory space is used and fewer operations are performed.

2.3 Conclusion

When looking at the algorithms that use the adjacency matrix of a graph to determine the number of triangles we conclude that using sparse matrices makes it possible to analyse larger graphs than when using arrays. We also conclude that using sparse matrices is also faster except for small graphs.

When looking at the algorithms that use the neighbourhoods of the vertices of a graph we conclude that it is faster to use only the edges (i, j) with $i < j$ than all the edges of the graph. We also conclude that, if the average vertex degree of the graph is not too small, it is faster to sort the vertices by vertex degree before counting the number of triangles.

When comparing all algorithms together we conclude that the algorithms using the neighbourhoods of the vertices of a graph are faster than the algorithms using the adjacency matrix of a graph.

3. Trusses

In this chapter we will compare algorithms that enumerate the maximal k-trusses of a graph. We first define a k-truss as follows:

Definition. Let $k \geq 3$. A k-truss is a non-trivial, connected graph such that each edge is part of at least $k-2$ different triangles.

Here, non-trivial excludes the graph consisting of only a single vertex. A maximal k-truss of a graph G is a k-truss that is not a proper subgraph of another k-truss.

Note that a maximal k-truss of a graph is a subgraph of a maximal $(k-1)$-truss of the graph. So if we want to compute the k-trusses and l-trusses with $k < l$ of a graph G it is faster to compute a maximal k-truss T_k of G and then a maximal l-truss T_l of T_k than computing the l-trusses of G because T_k is smaller than G. The truss T_l is also a maximal l-truss of G so this works.

In Figure 3.1 a graph together with its 3-truss and 4-truss is given. This graph does not contain any k-trusses for $k > 4$.

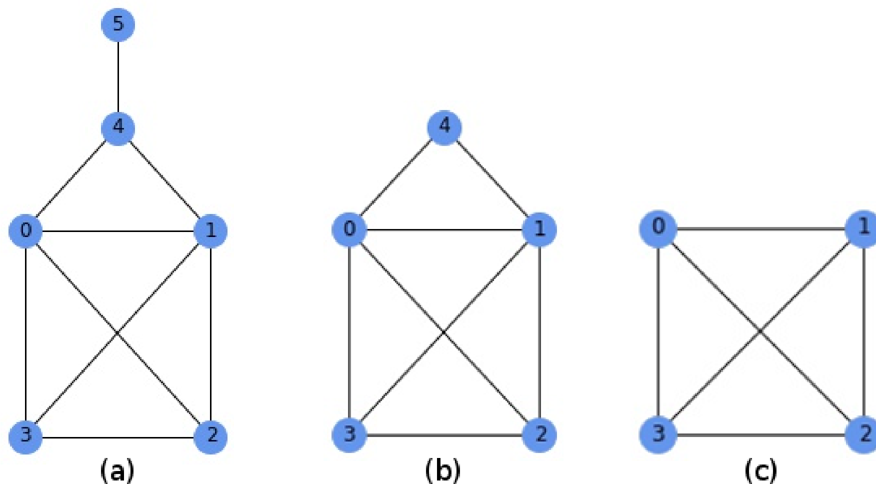

Figure 3.1: The graph (a) together with its maximal 3-truss (b) and its maximal 4-truss (c).

The first introduction of a truss is by Cohen in [10]. Cohen was looking for a generalisation of the clique of order k that is stricter than a $(k-1)$-core. A clique of order k is a subgraph with k vertices that are all adjacent to each other. Large cliques are not very likely to exist in real-life problems and are expensive to compute. A $(k-1)$-core is a maximal subgraph in which every

vertex is adjacent to at least $k-1$ other vertices in the subgraph. Computing the $(k-1)$-core of a graph is possible in polynomial time, but $(k-1)$-cores are not necessarily sets of high cohesion.

We are going to prove that a clique of order k is always a k-truss.

Proof. A clique of order k is a subgraph in which each edge joins two vertices. These two vertices have exactly $k-2$ common neighbours. So every edge is part of at least $k-2$ triangles. Hence, a clique of order k is a k-truss. □

Note that a k-truss does not necessarily need to contain a clique of order k. An example of a 4-truss that does not contain a clique of order 4 can be found in Figure 3.2.

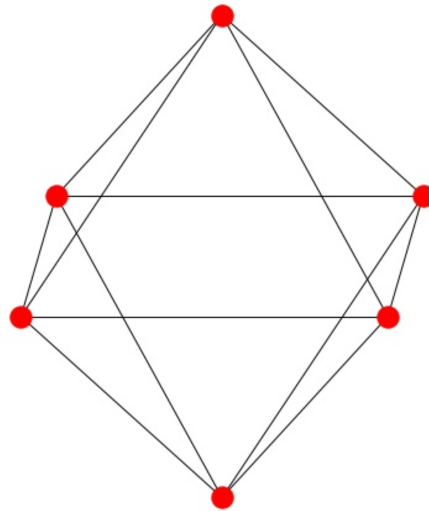

Figure 3.2: The octahedron is a 4-truss because each edge is part of two triangles. There are no four vertices that are all adjacent to each other, so the octahedron does not contain a clique of order 4.

We are going to prove that each k-truss of G is a subgraph of a $(k-1)$-core of G.

Proof. Let T be a k-truss of G. Trusses contain no isolated vertices, so every vertex v in T has a neighbour w. By the definition of a k-truss, the edge $\{v, w\}$ must be in at least $k-2$ triangles, so v and w share at least $k-2$ neighbours. These neighbours together with w make at least $k-1$ vertices in T adjacent to v. So every vertex of T is adjacent to at least $k-1$ vertices in T. Thus T is a $(k-1)$-core of G except possibly for maximality. The maximality of cores ensures that all of T is in the same $(k-1)$-core of G. Hence, each k-truss of G is a subgraph of a $(k-1)$-core of G. □

Note that a $(k-1)$-core of a graph G does not necessarily need to contain a k-truss. An example of a 3-core that does not contain a 4-truss can be found in Figure 3.3.

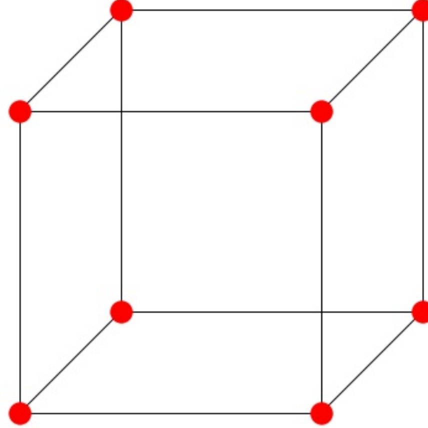

Figure 3.3: The cube is a 3-core because each vertex has three neighbours. The cube does not contain any triangles, so the cube does not contain any trusses.

3.1 Algorithms

We start by looking at the case $k = 3$, i.e. finding the maximal 3-trusses of a graph. Algorithm 12 gives the maximal 3-trusses of a graph G. It determines for each edge if it is part of a triangle. If the edge is not part of a triangle it is removed. We are going to prove that this algorithm indeed gives the 3-trusses of a simple graph G.

Algorithm 12 Algorithm to find the maximal 3-trusses of a graph.

Input: List of edges E of a graph G.
Result: List of edges E of the maximal 3-trusses of G.

```
n_E = |E|
nbh = EDGES_TO_NBH(E)
mask = []
for j ∈ range(n_E) do
    a, b = E(j)
    if |nbh(a) ∩ nbh(b)| = 1 then
        mask.append(j)
E = delete(E, mask)
return E
```

Proof. For each edge $\{a, b\}$ of the graph G the number of common neighbours of a and b is determined by $|nbh(a) \cap nbh(b)|$. If this number is less than one then $\{a, b\}$ is not part of a triangle and if this number is at least one then $\{a, b\}$ is part of a triangle. We use $mask$ to store the edges that are not part of a triangle. After we checked all the edges we remove the edges in $mask$ from the graph. Now the graph only contains the edges that were part of a triangle in the original graph. If the edge $\{a, b\}$ was part of the triangle $\{a, b, c\}$ in the original graph, then the edges $\{b, c\}$ and $\{c, a\}$ are also part of the triangle $\{a, b, c\}$. So none of the edges $\{a, b\}$, $\{b, c\}$ and $\{c, a\}$ are removed from the graph. Hence, the triangle $\{a, b, c\}$ is still part of the graph, so $\{a, b\}$ is still part of a triangle. We conclude that all the edges that are left in the graph after

removing the edges in *mask* are part of at least one triangle, so we have a 3-truss left. The 3-truss is also maximal in G because all the edges that are removed are not part of a triangle in G and can therefore not be part of a maximal 3-truss of G. □

Algorithm 13 gives the maximal k-trusses of a simple graph G for a given value of k. We are going to prove that this algorithm indeed gives the maximal k-trusses of a simple graph G.

Algorithm 13 Algorithm to find the maximal k-trusses of a graph.

Input: List of edges E of a graph G and the value for k.
Result: List of edges E of the maximal k-trusses of G.

```
n_E = |E|
i = 1 {any value > 0 works}
while n_E > 0 and i > 0 do
    nbh = EDGES_TO_NBH(E)
    mask = []
    for j ∈ range(n_E) do
        a, b = E(j)
        if |nbh(a) ∩ nbh(b)| < k − 2 then
            mask.append(j)
    i = |mask|
    E = delete(E, mask)
    n_E = |E|
return E
```

Proof. In each iteration Algorithm 13 removes all the edges that are not part of at least $k-2$ triangles. This is repeated until no more edges are removed or until there are no edges left. If there are no edges left the graph does not contain a k-truss. In each iteration *mask* is used to store the edges that are going to be removed in that iteration. Therefore we use $i = |mask|$ to determine if any edges are removed. If no more edges are removed, each edge that is left is part of at least $k-2$ triangles. So each edge that is left can be part of a maximal k-truss of G. Only edges that are part of less than $k-2$ triangles are removed, so all edges that are removed cannot be part of a maximal k-truss of G. Hence, when the algorithm terminates exactly the edges that are part of at least $k-2$ triangles are left. So the algorithm does indeed give the maximal k-trusses of a simple graph G. □

In Algorithm 13 every edge is checked in each iteration even though none of the adjacent edges was removed. This is not necessary and it would be more efficient to only check an edge $\{a, b\}$ again if an edge is removed in one of the triangles that $\{a, b\}$ is part of. This idea leads to Algorithm 14.

Here, *active* is the set of edges that still need to be checked if they are part of at least $k-2$ triangles, and *passive* is the set of edges that are checked and are part of at least $k-2$ different triangles. At the initialisation of the algorithm *active* contains all edges of the graph G, and *passive* is an empty set. When *active* becomes empty all edges are checked and the edges of *passive* together form the maximal k-trusses of the graph G.

As long as *active* is nonempty, an edge $\{a, b\}$ is taken from *active* with the function *active*.pop(). The function *active*.pop() returns a random element from *active* and removes it from *active* at the same time. It is a random element because the set *active* has no ordering, so there is for

Algorithm 14 Improved algorithm to find the maximal k-trusses of a graph.

Input: List of edges E of a graph G and the value for k.
Result: List of edges of the maximal k-trusses of G.

$nbh = $ EDGES_TO_NBH(E)
$passive = \{\}$
$active = E$
while $|active| > 0$ **do**
  $\{a, b\} = active.\text{pop}()$
  if $|nbh(a) \cap nbh(b)| < k - 2$ **then**
    $nbh(a) = nbh(a) - \{b\}$
    $nbh(b) = nbh(b) - \{a\}$
    for $c \in nbh(a) \cap nbh(b)$ **do**
      $active = active \cup \{\{a, c\}, \{b, c\}\}$
      $passive = passive - \{\{a, c\}, \{b, c\}\}$
  else
    $passive = passive \cup \{\{a, b\}\}$
return $passive$

example no first or last element. If the edge $\{a, b\}$ is part of less than $k - 2$ triangles, i.e. $|nbh(a) \cap nbh(b)| < k - 2$, then the edge $\{a, b\}$ is removed from the graph. All the edges $\{a, c\}$ and $\{b, c\}$ with the vertex c a neighbour of both a and b need to be checked again, because the number of triangles they are in is decreased, so they are added to *active* and removed from *passive*. If the edge $\{a, b\}$ is part of at least $k - 2$ triangles; it is added to *passive*.

If in Algorithm 14 an edge $\{a, b\}$ is removed from the graph it can happen that an edge $\{a, c\}$ is moved from *passive* to *active* even though it is still part of at least $k-2$ triangles. When it is the turn of the edge $\{a, c\}$ to be checked it is moved back to *passive*. This makes for unnecessarily moving around of edges which costs unnecessary computing time. In Algorithm 15 we prevent this from happening by checking if the edge $\{a, c\}$ is part of at least $k - 2$ triangles. If it is not it is moved from *passive* to *active*. At the initialisation we only put the edges that are part of less than $k - 2$ triangles in *active* and the others in *passive*. So in Algorithm 15 *active* is the queue of edges that are going to be removed from the graph.

In Algorithm 15 we use a counter $C(\{a, b\})$ for each edge $\{a, b\}$ which represents the number of triangles the edge is a part of. This is done because then we do not have to calculate $|nbh(a) \cap nbh(b)|$ as often as before. Calculating $|nbh(a) \cap nbh(b)|$ has time complexity $\mathcal{O}(\min(|nbh(a)|, |nbh(b)|)$, while reading off or changing the value of $C(\{a, b\})$ has time complexity $\mathcal{O}(1)$ because $C(\{a, b\})$ is stored as a dictionary [8]. This, together with less moving around of edges, should therefore make Algorithm 15 faster than Algorithm 14.

Algorithm 15 Further improved algorithm to find the maximal k-trusses of a graph.

Input: List of edges E of a graph G and the value for k.
Result: List of edges of the maximal k-trusses of G.

```
nbh = EDGES_TO_NBH(E)
n_E = |E|
C = {} {C is an empty dictionary}
passive = {} {passive is an empty set}
active = {} {active is an empty set}
for i ∈ range(n_E) do
   {a, b} = edges(i)
   C({a, b}) = |nbh(a) ∩ nbh(b)|
   if C({a, b}) < k − 2 then
      active = active ∪ {{a, b}}
   else
      passive = passive ∪ {{a, b}}
while |active| > 0 do
   {a, b} = active.pop()
   nbh(a) = nbh(a) − {b}
   nbh(b) = nbh(b) − {a}
   for c ∈ nbh(a) ∩ nbh(b) do
      C({a, c}) = C({a, c}) − 1
      C({b, c}) = C({b, c}) − 1
      if C({a, c}) < k − 2 then
         active = active ∪ {{a, c}}
         passive = passive − {{a, c}}
      if C({b, c}) < k − 2 then
         active = active ∪ {{b, c}}
         passive = passive − {{b, c}}
return passive
```

3.2 Results

The graph data used for enumerating trusses is the same data as used for counting triangles. So the characteristics of the graphs used can again be found in Table 2.1. We do not use the road network graphs because they are typically near planar and do not contain a lot of triangles and therefore contain small and few trusses.

In Table 3.1 the runtimes for the different algorithms for determining the maximal 3-trusses of a graph can be found. Each time the average of 10 runs is taken. Note that Algorithm 12 can only be used to determine the 3-trusses of a graph. The other algorithms can be used to determine the k-trusses of a graph for all $k \geq 3$. We notice that Algorithm 14 is the fastest algorithm for the three smallest graphs, and that Algorithm 12 is the fastest for the other graphs. We expected Algorithm 12 to be the fastest algorithm for 3-trusses because it uses that we only need one iteration for 3-trusses. We also notice that Algorithm 14 is always faster than Algorithm 13, which we expected. It is also always faster than Algorithm 15, which we expect the other way around.

In Table 3.2, Table 3.3 and Table 3.4 the runtimes for the different algorithms for determining

Table 3.1: Runtimes for the different algorithms for determining the maximal 3-trusses of a graph.

graph	runtime A12 (s)	runtime A13 (s)	runtime A14 (s)	runtime A15 (s)
0edges	0.0187	0.0376	**0.0145**	0.0273
tvshow	0.126	0.225	**0.116**	0.201
athletes	0.668	1.21	**0.661**	1.12
new_sites	**1.69**	3.13	1.89	2.99
Brightkite_edges	**1.52**	2.59	1.66	2.89
Gowalla_edges	**11.4**	20.9	12.3	18.2

respectively the maximal 4-trusses, 5-trusses and 6-trusses of a graph can be found. Each time the average of 10 runs is taken. We notice that for the 4-trusses, 5-trusses and 6-trusses Algorithm 14 is the fastest algorithm, followed by Algorithm 15, and Algorithm 13 is the slowest one. We expected Algorithm 15 to be faster than Algorithm 14. The reason that this is not the case is probably due to the implementation of the algorithms.

We also notice that the runtimes for the 4-trusses, 5-trusses and 6-trusses are close to each other for all the algorithms. The runtimes for the 3-trusses are also close to the runtimes of the other trusses for Algorithms 14 and 15. This indicates that the time complexity of algorithms for enumerating the k-trusses of a graph is not strongly dependent on k.

Table 3.2: Runtimes for the different algorithms for determining the maximal 4-trusses of a graph.

graph	runtime A13 (s)	runtime A14 (s)	runtime A15 (s)
0edges	0.0704	**0.0176**	0.0284
tvshow	0.495	**0.123**	0.209
athletes	3.37	**0.734**	1.20
new_sites	12.3	**2.02**	3.33
Brightkite_edges	8.85	**1.81**	3.08
Gowalla_edges	66.5	**12.9**	18.4

Table 3.3: Runtimes for the different algorithms for determining the maximal 5-trusses of a graph.

graph	runtime A13 (s)	runtime A14 (s)	runtime A15 (s)
0edges	0.0671	**0.0184**	0.0278
tvshow	0.546	**0.129**	0.218
athletes	3.25	**0.819**	1.30
new_sites	11.2	**2.18**	3.45
Brightkite_edges	9.19	**1.96**	3.25
Gowalla_edges	62.5	**13.8**	19.4

Comparing our algorithms with Cohen's algorithms in [10] we see that our Algorithm 13 uses the same idea as Cohen's proof that trusses can be computed in polynomial time. He shows that the time complexity of this method is bounded above by $\mathcal{O}(n_V n_E^2 + n_V)$ and therefore trusses

Table 3.4: Runtimes for the different algorithms for determining the maximal 6-trusses of a graph.

graph	runtime A13 (s)	runtime A14 (s)	runtime A15 (s)
0edges	0.109	**0.0173**	0.0293
tvshow	0.898	**0.127**	0.225
athletes	3.84	**0.906**	1.45
new_sites	11.7	**2.34**	3.71
Brightkite_edges	8.82	**2.09**	3.43
Gowalla_edges	66.2	**14.7**	10.4

can be computed in polynomial time. We also see that our Algorithm 15 uses the same ideas as Cohen's algorithm `removeUnsupportedEdges`(G, j). However, we do not use the idea used by Cohen to reduce the graph to its $(k-1)$-cores first and then use that to determine the k-trusses of the graph.

3.3 Conclusion

When looking at the algorithms to enumerate the maximal 3-trusses of a graph we conclude that the fastest algorithm is the one that only works for 3-trusses because it takes into account that iteration is not needed and that it is enough to determine if an edge is part of a triangle or not.

When looking at the algorithms to enumerate the maximal k-trusses of a graph for general $k \geq 3$ we conclude that it is faster to use "active" and "passive" edges than checking all the edges during each iteration. We also conclude that the runtime for enumerating the k-trusses of a graph is not strongly dependent on k for $k > 3$.

4. Graph Challenge Champions

The champions of the 2018 Graph Challenge are [11]:

- *Fast Triangle Counting Using Cilk* - Abdurrahman Yasar, Sivasankaran Rajamanickam, Michael Wolf, Jonathan Berry (Sandia), Umit V. Catalyurek (Georgia Tech),
- *High-Performance Triangle Counting on GPUs* - Yang Hu (GWU), Hang Liu (UMass Lowell), H. Howie Huang (GWU),
- *Update on Static Graph Challenge on GPU* - Mauro Bisson, Massimiliano Fatica (Nvidia), and
- *K-truss decomposition for Scale-Free Graphs at Scale in Distributed Memory* - Roger Pearce, Geoffrey Sanders (LLNL).

Before reviewing the results and innovations of the 2018 Champions,[1] we first take a look at the results of the 2017 Graph Challenge.

4.1 The 2017 Graph Challenge

The 2017 Graph Challenge consisted of the same challenges as the 2018 Graph Challenge. The organisers of the Graph Challenge compare the submissions of the 2017 Graph Challenge in [12]. They focus on triangle counting because it got the most submissions. The 2017 Graph Challenge had 22 submissions of which 10 focused on triangle counting, 3 on k-truss enumeration and 4 on both.

An important measure for comparing the results of triangle counting is the rate of edges processed and is given by

$$\text{Rate} = \frac{n_E}{T_{tri}}$$

where n_E is the number of edges and T_{tri} the triangle counting runtime. For each submission there are two points of interest: (1) the highest rate (and the corresponding number of edges), and (2) the largest graph processed (and the corresponding rate). The champions all had a high rate on large graphs. Comparing the submissions the organisers conclude that the state-of-the-art of graph analysis at that time was of the order 10^8 edges processed per second for graphs with 10^8 edges. The largest graph processed has $2.7 \cdot 10^{11}$ edges. As comparison, we achieved a rate of $4 \cdot 10^5$ edges per second for the largest graph `roadNet-CA` with $3 \cdot 10^6$ edges.

[1] At the time of writing (November 2018) not all papers of the Champions have yet been published. Therefore we only take a look at the ones that have been published.

Another way of comparing the submissions is by fitting a model to the data of each submission and then comparing these models. The model used is

$$T_{tri} = (n_E/n_1)^\beta$$

where n_1 and β are the parameters of the model. The parameter n_1 can be interpreted as the number of edges that takes 1 second to process. Submissions with larger n_1 and smaller β perform better. Most submissions have $\beta = 4/3$ for large values of n_E, so the state-of-the-art model is proportional to $n_E^{4/3}$ for large values of n_E.

4.2 Fast Triangle Counting Using Cilk

The authors of *Fast Triangle Counting Using Cilk* submitted to the 2017 Graph Challenge as well and became champions [11]. They submitted a linear algebra-based triangle counting implementation called KKTri. They used sparse matrix-matrix multiplication followed by element-wise matrix multiplication:

$$D = (L \cdot L) \circ L$$

where $\circ$ is the element-wise multiplication. Here, L is the lower triangle part of the adjacency matrix of the graph. They found three optimizations that are all used again in this year's implementation: (1) performing masked sparse matrix-matrix multiplication, where the right most L is used as mask, (2) using data compression on the middle matrix, and (3) ordering the graph before constructing the adjacency matrix [13].

For the 2018 Graph Challenge the authors improved their implementation by load-balancing using Cilk, an efficient, task-stealing, multi-threaded runtime. The new, improved algorithm is therefore called KKTri-Cilk. Another improvement made is changing the parallelisation strategy. KKTri used a simple way of partitioning by partitioning the matrix evenly into partitions with a fixed number of rows. KKTri-Cilk also creates partitions of rows, but not into a constant number of rows per partition. Instead it creates partitions such that the number of non-zero elements in each partition is approximately the same.

To demonstrate that KKTri-Cilk is an improvement they compare it to KKTri and a graph based implementation called TCM. KKTri-Cilk shows a speed-up of 5 times to 12 times compared to KKTri, and 3 times to 7 times compared to TCM. The highest rate of edges processed achieved is $1.14 \cdot 10^9$ which is about 2 times the best rate in the 2017 Graph Challenge.

4.3 High-Performance Triangle Counting on GPUs

The authors of *High-Performance Triangle Counting on GPUs* observed that modern GPU servers have multiple powerful GPUs on a single machine. They also observe that those GPU servers also have large enough CPU memories to hold large graphs. TriCore, the state-of-the-art GPU-based triangle counting project, stores the entire graph in the secondary memory to get no communication between different GPUs. The authors show that therefore the major limiting factor in processing speed becomes the disk bandwidth and not the processing speed of the GPUs.

Their solution is to buffer the graph in the CPU memory so that moving the data to the GPU becomes faster (see Figure 4.1). The servers are split into workload balancing groups to make

it possible to solve workload imbalance within the group. Each workload balancing group is responsible for one partition set. This makes it possible to process graphs that are larger than the CPU memory without using the slow disk memory.

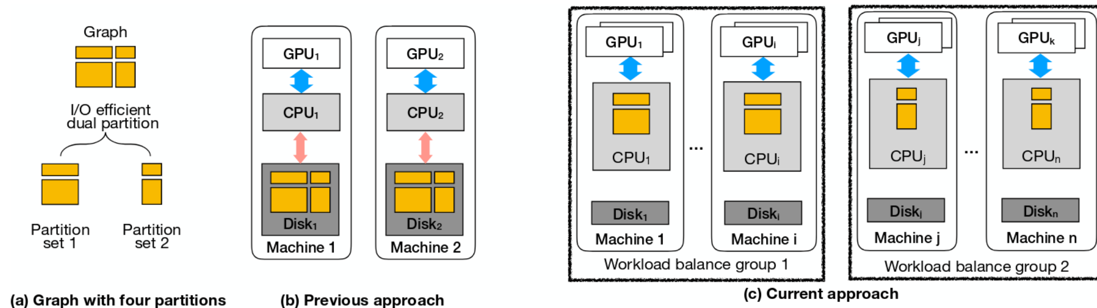

Figure 4.1: (a) A graph with four partitions is grouped into two partition sets. (b) TriCore stores the entire graph on the disk for each GPU. (c) The new approach stores each partition set in CPU memory. Each machine holds a partition set instead of a single partition to make workload stealing easier [14].

The authors show that their improvements lead to faster triangle counting. The highest rate of edges processed per second obtained is $4.7 \cdot 10^9$ for the `roadNet-CA` graph. This is approximately 10^4 times higher than our rate for the same graph. As far as the authors know, their implementation is the first GPU-based triangle counting that can process graphs with more than 10^{10} edges. It is also the first triangle counting project to achieve a rate of more than 10^8 edges per second for graphs with more than 10^{10} edges.

5. Concluding Remarks

We first looked at different algorithms for triangle counting. We divided the algorithms into two categories: (1) linear algebra based algorithms using the adjacency matrix of a graph, and (2) graph based algorithms using the neighbourhoods of a graph. When looking at the linear algebra based algorithms we concluded that using sparse matrices made it possible to analyse larger graphs than when using arrays. We also conluded that using sparse matrices was faster for large graphs. When looking at the graph based algorithms we concluded that it was faster to use only the edges (i, j) with $i < j$ than all the edges of the graph. We also concluded that sorting the vertices by vertex degree before counting the triangles was faster if the average vertex degree was not to small. When comparing the linear algebra based algorithms with the graph based algorithms we concluded that the graph based algorithms were faster than the linear algebra based algorithms.

After that we looked at enumerating maximal k-trusses. We treated the maximal 3-trusses as a special case, so we could take into account special properties of the 3-truss. We concluded that the algorithm that used the special properties was faster than the algorithms for general k-trusses when enumerating maximal 3-trusses. When looking at the algorithms for general k-trusses we concluded that it was faster to use "active" and "passive" edges than checking all the edges during each iteration. We also concluded that the runtime for k-trusses did not strongly depend on k for $k > 3$.

In the last part we looked at the champions of the 2018 Graph Challenge, as well as the results for triangle counting of the 2017 Graph Challenge. The largest graph processed for triangle counting in the 2017 Graph Challenge has $2.7 \cdot 10^{11}$ edges. Although incomparable, the largest graph that we processed has $3 \cdot 10^6$ edges. When reading the available papers of the 2018 champions we conclude that they focused on improving the triangle counting algorithms by improving how the graph data was partitioned before using parallel computing, and by how the graph data was stored in memory. One of the 2018 champions did this for linear algebra based algorithms on CPUs by improving the partitioning of the adjacency matrix before doing masked sparse matrix matrix multiplication. Another 2018 champion did it for a graph based algorithm on GPUs by improving how the graph partitions are stored in memory before and during processing. The highest rate of edges processed per second achieved for triangle counting in the 2018 Graph Challenge was $4.7 \cdot 10^9$ edges per second. Although incomparable, we achieved a processing rate of $4 \cdot 10^5$ edges per second for the same graph.

Our results cannot be compared to the results of the champions due to differences in available computing power. The most important difference is the hardware that was used, for example different processors and memory. Other differences can be the software used and the implementation of the algorithms. To be able to correctly compare the algorithms of the champions and our algorithms, the same hardware and software needs to be used.

A. Python Code for Triangle Counting

```
import numpy as np
import scipy.sparse
import time
import csv

def tsv_to_edgelist(file):
    matrix=[]
    with open(file) as tsvfile:
        reader=csv.reader(tsvfile,delimiter='\t')
        for row in reader:
            row_int=[]
            for i in range(len(row)):
                row_int.append(int(row[i]))
            matrix.append(row_int)
    return np.array(matrix)

def edges_to_adjacency(edges):
    n_ver=np.amax(edges)+1
    A=np.zeros((n_ver,n_ver),dtype=int)
    for i in range(len(edges)):
        A[edges[i,0],edges[i,1]]=1
    return A

def edges_to_dok(edges):
    n_ver=np.amax(edges)+1
    A_dok=scipy.sparse.dok_matrix((n_ver,n_ver),dtype=int)
    for i in range(len(edges)):
        A_dok[edges[i,0],edges[i,1]]=1
    return A_dok

def edges_to_nbh(edges):
    n_edges=len(edges)
    n_ver=np.amax(edges)+1
    nbh=[set()]*n_ver
    for i in range(n_edges):
        nbh[edges[i,0]]=nbh[edges[i,0]]|{edges[i,1]}
    return nbh
```

```python
#%% Adjacency matrix based algorithms

def n_triangles_baseline(edges):
    A=edges_to_adjacency(edges)
    B=A @ A * A
    count=B.sum()//6
    return count

def n_triangles_matrix(edges):
    A=edges_to_adjacency(edges)
    n_rows=len(A)
    count=0
    for i in range(n_rows):
        for j in range(i+1,n_rows):
            if A[i,j]!=0:
                count+=np.sum(A[i,i+1:j]*A[j,i+1:j])
    return count

def n_triangles_sparse(edges):
    A_dok=edges_to_dok(edges)
    A_sparse=scipy.sparse.csr_matrix(A_dok)
    B=A_sparse.multiply(A_sparse*A_sparse)
    count=B.sum()//6
    return count

#%% Neighbourhood based algorithms

def n_triangles_nbh(edges):
    nbh=edges_to_nbh(edges)
    n_edges=len(edges)
    count=0
    for k in range(n_edges):
        i=edges[k,0]
        j=edges[k,1]
        count+=len(nbh[i]&nbh[j])
    return count//6

def n_triangles_nbh_improved(edges):
    nbh=edges_to_nbh(edges)
    n_edges=len(edges)
    count=0
    for k in range(n_edges):
        i=edges[k,0]
        j=edges[k,1]
        if i<j:
            count+=len(nbh[i]&nbh[j])
            nbh[i] -= {j}
            nbh[j] -= {i}
```

```
    return count

def n_triangles_nbh_improved2(edges):
    n_ver=np.amax(edges)+1
    nbh=edges_to_nbh(edges)
    permutation=sorted(range(n_ver),key=lambda idx: len(nbh[idx]))
    count=0
    for i in permutation:
        nbhi=nbh[i]-set()
        for j in nbhi:
            count+=len(nbh[i] & nbh[j])
            nbh[i] -= {j}
            nbh[j] -= {i}
    return count

def n_triangles_nbh_improved3(edges):
    n_edges=len(edges)
    mask=[]
    for i in range(n_edges):
        if edges[i,0]>edges[i,1]:
            mask.append(i)
    edges=np.delete(edges,mask,0)
    n_edges=len(edges)
    nbh=edges_to_nbh(edges)
    count=0
    for k in range(n_edges):
        i=edges[k,0]
        j=edges[k,1]
        count+=len(nbh[i]&nbh[j])
    return count

def n_triangles_nbh_improved4(edges):
    n_ver=np.amax(edges)+1
    n_edges=len(edges)
    mask=[]
    for i in range(n_edges):
        if edges[i,0]>edges[i,1]:
            mask.append(i)
    edges=np.delete(edges,mask,0)
    n_edges=len(edges)
    nbh=edges_to_nbh(edges)
    permutation=sorted(range(n_ver),key=lambda idx: len(nbh[idx]))
    count=0
    for i in permutation:
        for j in nbh[i]:
            count+=len(nbh[i] & nbh[j])
    return count

def n_triangles_nbh_improved5(edges):
```

```
    n_ver=np.amax(edges)+1
    n_edges=len(edges)
    mask=[]
    for i in range(n_edges):
        if edges[i,0]>edges[i,1]:
            mask.append(i)
    edges=np.delete(edges,mask,0)
    n_edges=len(edges)
    nbh=edges_to_nbh(edges)
    edge_degrees=np.zeros((n_edges,3),dtype='int')
    for k in range(n_edges):
        i=edges[k,0]
        j=edges[k,1]
        edge_degrees[k]=[i,j,min(len(nbh[i]),len(nbh[j]))]
    count=[0]*n_ver
    for a in edge_degrees[:,2]:
        count[a]+=1
    running_count=[0]*n_ver
    for i in range(1,n_ver):
        running_count[i]=running_count[i-1]+count[i]
    sorted_edges=np.zeros((n_edges,3),dtype='int')
    for a in range(n_edges):
        deg=edge_degrees[a,2]
        idx=running_count[deg]-1
        sorted_edges[idx]=edge_degrees[a]
        running_count[deg]-=1
    n_tri=0
    for k in range(n_edges):
        i=sorted_edges[k,0]
        j=sorted_edges[k,1]
        n_tri+=len(nbh[i]&nbh[j])
    return n_tri

#%% Testing the algorithms

test_edges=['0edges.tsv','tvshow.tsv','athletes.tsv','new_sites.tsv',
'Brightkite_edges.txt','Gowalla_edges.txt','roadNet-PA.txt','roadNet-CA.txt']
n_ver=np.zeros(len(test_edges),dtype=int)
n_edges=np.zeros(len(test_edges),dtype=int)
d=np.zeros(len(test_edges))
n_triangles=np.zeros(len(test_edges),dtype=int)
runtimes=np.zeros((len(test_edges),3))
for i in range(len(test_edges)):
    edges=tsv_to_edgelist(test_edges[i])
    n_ver[i]=np.amax(edges)+1
    n_edges[i]=len(edges)/2
    d[i]=2*n_edges[i]/n_ver[i]
    start=time.time()
    n_triangles_sparse(edges)
```

```
    runtimes[i,2]=time.time()-start
for i in range(4):
    edges=tsv_to_edgelist(test_edges[i])
    start=time.time()
    n_triangles[i]=n_triangles_baseline(edges)
    runtimes[i,0]=time.time()-start
    start=time.time()
    n_triangles_matrix(edges)
    runtimes[i,1]=time.time()-start

print(n_ver,n_edges,d,n_triangles)
print(runtimes)

runtimes=np.zeros((len(test_edges),6))
x=10
for i in range(len(test_edges)):
    edges=tsv_to_edgelist(test_edges[i])
    start=time.time()
    for _ in range(x):
        n_triangles_nbh(edges)
    runtimes[i,0]=(time.time()-start)/x
    start=time.time()
    for _ in range(x):
        n_triangles_nbh_improved(edges)
    runtimes[i,1]=(time.time()-start)/x
    start=time.time()
    for _ in range(x):
        n_triangles_nbh_improved2(edges)
    runtimes[i,2]=(time.time()-start)/x
    start=time.time()
    for _ in range(x):
        n_triangles_nbh_improved3(edges)
    runtimes[i,3]=(time.time()-start)/x
    start=time.time()
    for _ in range(x):
        n_triangles_nbh_improved4(edges)
    runtimes[i,4]=(time.time()-start)/x
    start=time.time()
    for _ in range(x):
        n_triangles_nbh_improved5(edges)
    runtimes[i,5]=(time.time()-start)/x

print(runtimes)
```

B. Python Code for Truss Enumeration

```
import numpy as np
import csv
import time

def tsv_to_edgelist(file):
    matrix=[]
    with open(file) as tsvfile:
        reader=csv.reader(tsvfile,delimiter='\t')
        for row in reader:
            row_int=[]
            for i in range(len(row)):
                row_int.append(int(row[i]))
            matrix.append(row_int)
    return np.array(matrix)

def edges_to_nbh(edges):
    n_edges=len(edges)
    n_ver=np.amax(edges)+1
    nbh=[set()]*n_ver
    for i in range(n_edges):
        nbh[edges[i,0]]=nbh[edges[i,0]]|{edges[i,1]}
    return nbh

#%% Truss enumeration algorithms

def truss_3(edges):
    nbh=edges_to_nbh(edges)
    n_edges=len(edges)
    mask=[]
    for j in range(n_edges):
        a,b=edges[j]
        if len(nbh[a]&nbh[b])==0:
            mask.append(j)
    edges=np.delete(edges,mask,0)
    return edges

def k_truss(edges,k):
```

```
    n_edges=len(edges)
    i=len(edges)
    while n_edges>0 and i>0:
        nbh=edges_to_nbh(edges)
        mask=[]
        for j in range(n_edges):
            a,b=edges[j]
            if len(nbh[a]&nbh[b])<k-2:
                mask.append(j)
        i=len(mask)
        edges=np.delete(edges,mask,0)
        n_edges=len(edges)
    return edges

def k_truss_improved(edges,k):
    nbh=edges_to_nbh(edges)
    passive=set()
    active=set()
    for i in range(len(edges)):
        active |= {frozenset(edges[i].tolist())}
    while len(active)>0:
        a,b=active.pop()
        if len(nbh[a]&nbh[b])<k-2:
            nbh[a] -= {b}
            nbh[b] -= {a}
            for c in nbh[a]&nbh[b]:
                active |= {frozenset([a,c]),frozenset([b,c])}
                passive -= {frozenset([a,c]),frozenset([b,c])}
        else:
            passive |= {frozenset([a,b])}
    return passive

def k_truss_improved2(edges,k):
    nbh=edges_to_nbh(edges)
    n_edges=len(edges)
    C={}
    passive=set()
    active=set()
    for i in range(n_edges):
        m,n=edges[i]
        C[frozenset([m,n])]=len(nbh[m]&nbh[n])
        if C[frozenset([m,n])]<k-2:
            active |= {frozenset([m,n])}
        else:
            passive |= {frozenset([m,n])}
    while len(active)>0:
        a,b=active.pop()
        nbh[a] -= {b}
        nbh[b] -= {a}
```

```
        for c in nbh[a]&nbh[b]:
            C[frozenset([a,c])]-=1
            C[frozenset([b,c])]-=1
            if C[frozenset([a,c])]<k-2:
                active |= {frozenset([a,c])}
                passive -= {frozenset([a,c])}
            if C[frozenset([b,c])]<k-2:
                active |= {frozenset([b,c])}
                passive -= {frozenset([b,c])}
    return passive

#%% Testing truss algorithms

test_edges=['0edges.tsv','tvshow.tsv','athletes.tsv','new_sites.tsv',
'Brightkite_edges.txt','Gowalla_edges.txt']
runtimes=np.zeros((len(test_edges),4))
x=10
k=3
for i in range(len(test_edges)):
    edges=tsv_to_edgelist(test_edges[i])
    start=time.time()
    for _ in range(x):
        truss_3(edges)
    runtimes[i,0]=(time.time()-start)/x
    start=time.time()
    for _ in range(x):
        k_truss(edges,k)
    runtimes[i,1]=(time.time()-start)/x
    start=time.time()
    for _ in range(x):
        k_truss_improved(edges,k)
    runtimes[i,2]=(time.time()-start)/x
    start=time.time()
    for _ in range(x):
        k_truss_improved2(edges,k)
    runtimes[i,3]=(time.time()-start)/x

print(runtimes)

runtimes=np.zeros((len(test_edges),3))
x=10
k=4
for i in range(len(test_edges)):
    edges=tsv_to_edgelist(test_edges[i])
    start=time.time()
    for _ in range(x):
        k_truss(edges,k)
    runtimes[i,0]=(time.time()-start)/x
    start=time.time()
```

```
    for _ in range(x):
        k_truss_improved(edges,k)
    runtimes[i,1]=(time.time()-start)/x
    start=time.time()
    for _ in range(x):
        k_truss_improved2(edges,k)
    runtimes[i,2]=(time.time()-start)/x

print(runtimes)

runtimes=np.zeros((len(test_edges),3))
x=10
k=5
for i in range(len(test_edges)):
    edges=tsv_to_edgelist(test_edges[i])
    start=time.time()
    for _ in range(x):
        k_truss(edges,k)
    runtimes[i,0]=(time.time()-start)/x
    start=time.time()
    for _ in range(x):
        k_truss_improved(edges,k)
    runtimes[i,1]=(time.time()-start)/x
    start=time.time()
    for _ in range(x):
        k_truss_improved2(edges,k)
    runtimes[i,2]=(time.time()-start)/x

print(runtimes)

runtimes=np.zeros((len(test_edges),3))
x=10
k=6
for i in range(len(test_edges)):
    edges=tsv_to_edgelist(test_edges[i])
    start=time.time()
    for _ in range(x):
        k_truss(edges,k)
    runtimes[i,0]=(time.time()-start)/x
    start=time.time()
    for _ in range(x):
        k_truss_improved(edges,k)
    runtimes[i,1]=(time.time()-start)/x
    start=time.time()
    for _ in range(x):
        k_truss_improved2(edges,k)
    runtimes[i,2]=(time.time()-start)/x

print(runtimes)
```

Bibliography

[1] 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. *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, 2017.

[2] Scenarios. `http://graphchallenge.mit.edu/scenarios`, May 2017. Accessed: 17 November 2018.

[3] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D. Owens. A Comparative Study on Exact Triangle Counting Algorithms on the GPU. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, pages 1–8. ACM, 2016.

[4] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient Algorithms for Large-Scale Local Triangle Counting. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 4(3):13, 2010.

[5] Stephanie Dodson, Darrell O Ricke, and Jeremy Kepner. Genetic Sequence Matching Using D4M Big Data Approaches. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6. IEEE, 2014.

[6] Alex Fornito. Graph Theoretic Analysis of Human Brain Networks. In *fMRI Techniques and Protocols*, pages 283–314. Springer, 2016.

[7] Sparse matrices (scipy.sparse). `https://docs.scipy.org/doc/scipy/reference/sparse.html`, May 2018. Accessed: 16 September 2018.

[8] TimeComplexity. `https://wiki.python.org/moin/TimeComplexity`, June 2017. Accessed: 12 October 2018.

[9] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data`, June 2014. Accessed: 30 August 2018.

[10] Jonathan Cohen. Trusses: Cohesive Subgraphs for Social Network Analysis. *National Security Agency Technical Report*, 16, 2008.

[11] Graph Challenge Champions. `http://graphchallenge.mit.edu/champions`, September 2018. Accessed: 5 November 2018.

[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. GraphChallenge.org: Raising the Bar on Graph Analytic Performance. *arXiv preprint arXiv:1805.09675*, 2018.

[13] Abdurrahman Yaşar, Sivasankaran Rajamanickam, Michael Wolf, Jonathan Berry, and Ümit V. Çatalyürek. Fast Triangle Counting Using Cilk.

[14] Yang Hu, Hang Liu, and H. Howie Huang. High-Performance Triangle Counting on GPUs.