Finding the number of triangles in a Graph

Iakovidis Ioannis

September 20, 2019

Contents

1 Code

In this link there is a github repository with the code used. Code

2 Introduction

In this document we will describe GPU (Cuda) implementation for counting the number of triangles in a undirected simple sparse graph. Let Graph $G(V, E)$ where V is the vertex set and E is the edge set, with $N = |V|$ and $M = |E|$. We are given the adjacency matrix A of the graph in a COO format (matrix-markets format). And our the expected output is the number of triangles n_T.

It is true that we can compute the number of triangles given an adjacency matrix with the following computation:

$$n_T = \frac{1}{6} \sum_{i,j} (A^2 \odot A)_{i,j}$$

Where $\odot$ is the Hadamard product ([3] and [4] have theoretical analysis of the problem).

By the algorithm description it is evident that we don't need to compute the whole matrix multiplication. But we just need the elements $A^2_{i,j}$ of A^2 such that $A_{i,j} \neq 0$.

In the following paragraphs we will first simplify the computation to one that both fits our data format and does less operations. Then we will explain a trivial parallelization and then we will do optimization choices.

3 Simplifying The Computation

Because the graph that we work is a simple undirected graph we know that A is symmetric and has diagonal 0 therefore $\exists L$ lower triangular matrix such that $A = L + L^T$ so

$$\begin{aligned} A^2 &= (L + L^T) \cdot (L + L^T) \\ &= L^2 + (L^T)^2 + LL^T + L^T L \\ &= L^2 + (L^2)^T + L^T L + (L^T L)^T. \end{aligned}$$

So because A is symmetric and L^2 is lower triangular.

$$\begin{aligned} \text{sum}_A A^2 &= \text{sum}_A[L^2 + (L^2)^T + L^T L + (L^T L)^T) \cdot (L + L^T)] \\ &= 2 \cdot \text{sum}_A(L^2 + L^T L) \\ &= 2 \cdot \text{sum}_L(L^2 + L^T L) + 2 \cdot \text{sum}_{L^T}(L^2 + L^T L) \\ &= 2 \cdot \text{sum}_L(L^2) + 4 \cdot \text{sum}_L(L^T L) \end{aligned}$$

Where $\text{sum}_A D$ be the sum of the elements of D masked by A $\text{sum}_A D = \sum_{ij} (D \odot A)_{ij}$.

$$\text{Claim } \text{sum}_L(L^2) = \text{sum}_L(L^T L)$$

Proof. Let vertex set $V = \{1, \dots, N\}$.
L is itself an adjacency matrix of a directed graph in the vertex set of A.
For the edge set E' of this graph it is true that $(i, j) \in E' \iff ((i, j) \in E \text{ and } i > j)$.
So L^2 counts the number of paths of length 2 (i, j, k) in the graph of A $(i, j), (j, k) \in E$ and $i > j > k$.
Now with the point-wise multiplication by L we check if $i > k$ so we count all the triangles of G once.
Therefore $n_T = \text{sum}_L(L^2) \implies \text{sum}_L(L^2) = \frac{1}{3}\text{sum}_L(L^2) + \frac{2}{3}\text{sum}_L(L^T L)$ □

So

$$n_T = \mathrm{sum}_L(L^T L)$$

$$(L^T L)_{i,j} = \sum_{k=0}^{n-1} L_{k,i} \cdot L_{k,j}$$

We choose $L^T L$ over LL^T and L^2 because the matrices are given the lower triangular part in column major order (in the matrix market).

4 Parsing The Matrix

In the COO format is difficult reference a column or row.So first of all from the given COO lower triangular sparse matrix we extract :

```
    int *I,*dI; //device and host arrays of the row indices of the non zero elements
    int *J,*dJ; //device and host arrays of the col indices of the non zero elements
    int *col_ptr; //Array of indices to the starts of the columns
```

This is done because the matrix is given in column major order (if in row major order row pointers).

The col_ptr array is of length N and is constructed in the device memory by the kernel:

```
__global__ void findCol_ptr(int *dJ, int nz, int *col);
```

The col_ptr array is initialized with -1. Also we handle neighbouring indices in for obtaining the column lengths with col_ptr[i+1]-col_ptr[i] ,length of i'th column (if non-empty).

5 Trivial Parallelization

Because the number of triangles that we count for each element is independent we can give an elements to each thread. Then perform the $(L^T L)_{i,j}$ for the thread that (i, j) is mapped to, And to finish we can do a sum reduction to the results of each thread. To compute the number of triangles.

To find the numbers of triangles that is associated with each element we need to take a dot product of the i'th and j'th column. For the dot product or intersection algorithm we start two pointers at the start of each column obtained from col_ptr if the pointed row indices are the same we increment a counter or else we increase the smaller one.

6 Performance Considerations

In the above description we have some performance errors that make us not utilize the GPU in the right way. Bellow we describe these errors and give motivation for the second version.

First of all there are multiple loads for every column that can be scattered across SM's. This can be reduced if each column is processed by a single Thread-Block. That way we can also expose more parallelism in our program. Performance gains will become more apparent when we have columns with many elements.

Also there is no usage of shared memory. Every load is done from global memory.

An other performance hazard is that our column dot product (intersection algorithm) is divergent between threads of the same warp.

7 Use of Shared Memory

For the second version we as said above each column is processed by a single Thread-Block, performs the task to find the triangles from the elements of this column. So in each Thread-Block we store the row indices of a column in the shares memory. Then each thread counts the triangles for a point in this column. And then we reduce the Thread-Block.

Also because the columns may not have enough element we load multiple columns so we can launch a large number of threads without having them idle. The following kernel (pseudocode) describe this procedure.

```
__global__ void computeCol(int* dI,int* dJ,int nz,int* col ,int* out, int N,int k) {
    int s=0;
    int j;
    extern __shared__ int nt [];
    __shared__ int blockCol[sharedsize ];
    int tid=threadIdx.x;
    for(int i=blockIdx.x;i<N/k+1;i+=gridDim.x){
    /*Find the lengths of k consecutive columns */
     lengths(col ,k,N,nz);
      /*Load the columns*/
        colStart=b;
        for( j=tid;j<a;j+=blockDim.x)
        {
                blockCol[j]=dI[j+colStart];
        }
          __syncthreads();
     /*Search each element of the columns*/
     for( j=tid;j<a;j+=blockDim.x)
        {
      /*Find the appropriate length in the blockCol array b*/
        s=s+ComputeIntersection(blockCol,col,b ,nz, dI, j);
     }
    }
    /*Sum reduce the results of the threads*/
      Reduce(nt,s);
        if(tid<32){
        out[blockIdx.x]+=nt[0];}
}
```

Ending we reduce the results of the blocks to find the number of triangles.

The advantages of this computation is that we load each column only once and is handled by one block. The disadvantages is that the have additional serial code to find and manage the lengths of the non empty columns that are processed by the thread-block.

With all that said we expect that we expect the method to be beneficial when the average number of non-zeros per column is larger. And there may be a threashold to which algorithm we should use.

8 Bitmap

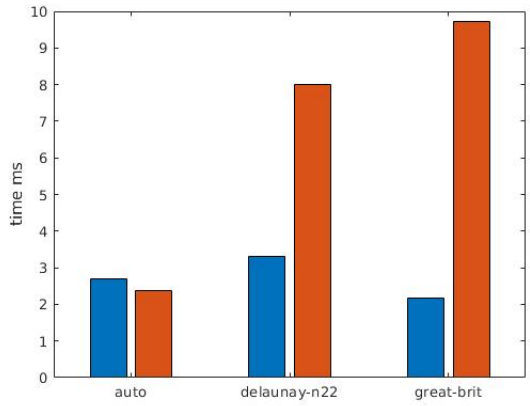
We also implement a bitmap implementation of the algorithm. As described in [2] a bitarray stored in global memory used to perform the intersections.

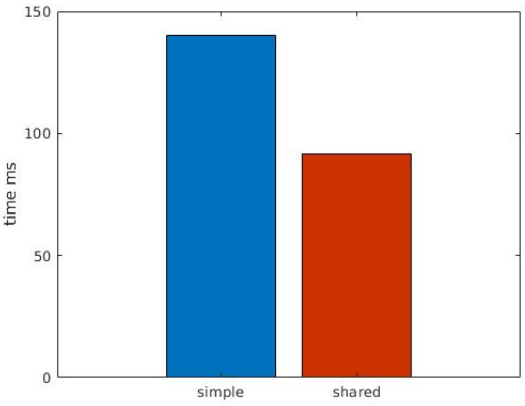
9 Results and Optimization Choices

We tested our implementations on a number of graphs auto,great-britan,delaunay-n22. These graphs have average number 7.3,3 and 1.5 of non-zeros per column. And for a more dense graph that has average 36.9 non-zeros per column coPapers-citeseer.

We used a NVDIA Tesla T4 compute capability 7.5. First we choose 512 threads and 256 blocks for the simple implementation. And for the second implementation we use 128 threads 1024 blocks and we load 27 columns to the shared memory implementation on the first 3 graphs. For the more dense graph we load 1 column in the shared memory (max number of non-zeros in a column is 1093).

Below we see the bar charts of the experiments.





We have so we see that the shared memory implementation works well in the more dense graphs while the threshold is around 7 nnz per column (auto graph). For the exact results see the table below.

Results of simple and shared implementation on a NVDIA Tesla T4				
Graph	nnz/column	Time of simple	Time of shared	Speedup
delaunay-n22	1.5	3.32 ms	8 ms	0.41
great-britan	3	2.17 ms	9.72 ms	0.22
auto	7.3	2.69 ms	2.38 ms	1.13
coPapers-citeseer	36.9	140.23 ms	91.47 ms	1.53

10 Bibliography

1. Guide To Optimizing Parallel Reductions in CUDA. Mark Harris
2. High Performance Exact Triangle Counting on GPUs,Mauro Bisson
3. Notes on theoretical analysis
4. Counting Triangles and the Curse of the Last Reducer