Parallel And Distributed Systems
Assignment 4

Konstantinos Chatziantoniou 8941

1/10/2019

Code is available on github.
"https://github.com/KonstantinosChatziantoniou/GraphTrianglesCounting"

1 Problem Statement

Implement a CUDA algorithm to calculate the triangles of a sparse undirected graph. The number of triangles is given using the adjacency matrix A with the formula $\frac{1}{6}\sum_{ij}[(A \cdot A) \odot A]$. The matrix is given in COO format and since it is symmetric (undirected graph), only the half is given in the dataset.

2 First Glance

At first glance, the triangles counting problem corresponds to a sparse matrix multiplication, only at the non zero positions of A and without storing the whole result but rather summing it. To solve this problem we have to transform the COO to CSR and CSC format and load them to the GPU. We need both CSR and CSC to make the most out of the GPU batch reading. The amount of memory that will be used is $2 \cdot (rows + non-zero-elements) * sizeof(int)$. Also we more code will be written to use the symmetric part that is not given by the dataset.

3 The epiphany

While working on the previous idea, I noticed that $\sum_{ij}[(U \cdot U^T) \odot U]$ and $\sum_{ij}[(U \cdot U) \odot U]$ yielded the same result.

Some quick Math

The adjacency matrix A can be written as the sum of its upper U and lower L triangular matrix. Since the graph is undirected, A is *symmetric* $U = L^T$. Also, the diagonal of A is zero, so $A = U + U^T$. This is useful, because the input data provide only the upper triangle of the adjacency matrix as COO and can be easily transformed to both CSR and CSC.
Now the mulitplication $A \cdot A$ can be expanded to

$$A \cdot A = U \cdot U + U \cdot U^T + U^T \cdot U + U^T \cdot U^T.$$

This expansion also allows as to prove that the product of $A \cdot A$ is symmetrical.

$$\begin{aligned}(A \cdot A)^T &= (U \cdot U + U \cdot U^T + U^T \cdot U + U^T \cdot U^T)^T = \\ &= (U \cdot U)^T + (U \cdot U^T)^T + (U^T \cdot U)^T + (U^T \cdot U^T)^T = \\ &= U^T \cdot U^T + U^T \cdot U + U \cdot U^T + U \cdot U = A \cdot A\end{aligned}$$

Now, the initial product is expanded to 4 products.
First, we observe that 1 of the products is the transpose of the other.

$$(U \cdot U)^T = U^T \cdot U^T$$

Next we further examine those 2 products. The product $U \cdot U$ is an upper triangle matrix as product of 2 triangle matrices.

Proof. The elements of U are u_{ij} with $u_{ij} = 0$ for $i < j$ and the elements of the product $C = U \cdot U$ are c_{ij}.

$$\text{So} \quad U \cdot U = \sum_i^n \sum_j^n \sum_k^n u_{ik} u_{kj}$$

We have the following cases for k if $i < j$:

if $k < i < j$ then $u_{kj} = 0$

if $i < k < j$ then $u_{ik} = u_{kj} = 0$

if $i < j < k$ then $u_{ik} = 0$

Therefore $c_{ij} = 0$ for $i < k$ and so U is upper triangular and U^T is lower triangular. □

Also the other 2 products are symmetrical

$$\begin{aligned}(U^T U)^T &= (U)^T (U^T)^T = U^T U \\ \text{and } (U U^T)^T &= (U^T)^T (U)^T = U U^T\end{aligned}$$

Before we continue we will reform the equation that gives the total number of the triangles of an undirected graph:

$$\begin{aligned}n &= \frac{1}{6} \sum_{ij} [(A \cdot A) \odot A] \\ \sum_{ij} [(A \cdot A) \odot A] &= \sum_{ij} [(U + U^T)(U + U^T) \odot (U + U^T) = \\ &= \sum_{ij} UU \odot U + \sum_{ij} U^T U^T \odot U + \sum_{ij} UU^T \odot U + \sum_{ij} U^T U \odot U \\ &+ \sum_{ij} UU \odot U^T + \sum_{ij} U^T U^T \odot U^T + \sum_{ij} UU^T \odot U^T + \sum_{ij} U^T U \odot U^T\end{aligned}$$

Since UU is upper triangular and U^T is lower triangular, the elementwise product is the zero matrix: $(UU) \odot U^T = \mathbf{0}$. The same applies to $(U^T U^T) \odot U$.

So

$$\sum_{ij}[(A \cdot A) \odot A] = \sum_{ij} UU \odot U + \sum_{ij} UU^T \odot U + \sum_{ij} U^T U \odot U + \sum_{ij} U^T U^T \odot U^T + \sum_{ij} UU^T \odot U^T + \sum_{ij} U^T U \odot U^T$$

Hadamard and Trace Properties

Below are listed the properties that will help us reduce the amount of calculations.

1 $\sum_{ij} A \odot B = tr(B^T A)$

2 $tr(AB) = tr(BA)$

3 $tr(A^T B) = tr(AB^T)$

4 $tr(A^T B) = tr(B^T A)$

Applying the properties

Applying the first property:

$$\begin{aligned}\sum_{ij}[(A \cdot A) \odot A] &= \sum_{ij} UU \odot U + \sum_{ij} UU^T \odot U + \sum_{ij} U^T U \odot U \\ &+ \sum_{ij} U^T U^T \odot U^T + \sum_{ij} UU^T \odot U^T + \sum_{ij} U^T U \odot U^T \\ &= tr(U^T UU) + tr(U^T UU^T) + tr(U^T U^T U) + tr(UU^T U^T) + tr(UUU^T) + tr(UU^T U)\end{aligned}$$

Applying the second,third and forth properties:

$$\begin{aligned} & & x &= tr(U^T UU) \\ (3), A^T = U^T U, B = U & & &= tr(U^T UU^T) \\ (4), A^T = U^T, B = UU & & &= tr(U^T U^T U) \\ (3), A^T = U^T, B = UU & & &= tr(UU^T U^T) \\ (2), A^T = U^T, B = UU & & &= tr(UUU^T) \\ (2), A^T = U^T U, B = U & & &= tr(UU^T U)\end{aligned}$$

Therefore,

$$\sum_{ij}[(A \cdot A) \odot A] = 6x = 6tr(UU^T U) \text{ and}$$
$$n = x = tr(UU^T U)$$

4 Implementation

We will not use the last relation as is ($n = tr(UU^TU)$) because it requires 2 multiplications. We will use the equivalent $n = \sum_{ij} UU^T \odot U$.

The GPU memory will be loaded with the CSR format of the data. Since we have UU^T the elements are in the right order for batch reading. The effect of $\odot U$ is loading only the column that the row has non zero element.The kernel's number of **blocks** will be the number of *rows* of the matrix A and the number of **threads** will be the *max number of non zero elements per row*. Each block will do a row part of the multiplication.

Steps

CPU

1 Read the data from the text file (COO).

2 Compress the row vector and free the uncompressed (CSR). At the same time the max non zero elements per row is calculated.

3 Do the memory allocation and data loading to the GPU.

4 Launch the kernel.

5 Read the results for the sum of each row and do the final summation.

GPU KERNEL

1 Batch read the row that is assigned to the block and store it to shared memory. (main row)

2 Batch read all the rows that the main row's elements are pointing at and store them to the shared memory.

3 Each thread does the multiplication and summation between their row and the main row. Because the A matrix is boolean, this ends up being a search for the common elements between 2 sorted series.

4 Each thread stores the result in shared memory.

5 The thread with id 0, does the rest of summation and stores the result that corresponds to the main row to device memory.

Comments

- For the largest dataset (delunay), reading the data takes 1-2 secs. This cannot be parallelized and was left as is.
- For the largest dataset (delunay), converting the data to CSR takes 20ms. This is low enough and doesn't need to be parallelized.
- In order to reduce the conditional blocks in the kernel, I did the following modification.

```
if(a == per_row || b == per_col){
                    break;
                }

                if(current_row[a] == current_col[b]){
                    sum++;
                    a++;
                    b++;
                    continue;
                }

                if(current_row[a] > current_col[b]){
                    b++;
                    continue;
                }

                if(current_row[a] < current_col[b]){
                    a++;
                    continue;
                }
```

Listing 1: With Conditional Blocks

```
int b1 = current_row[a] == sh_cols[id][b];
int b2 = current_row[a] > sh_cols[id][b];
int b3 = current_row[a] < sh_cols[id][b];

a = a + b1 + b3;
b = b + b1 + b2;
sum = sum + b1;
```

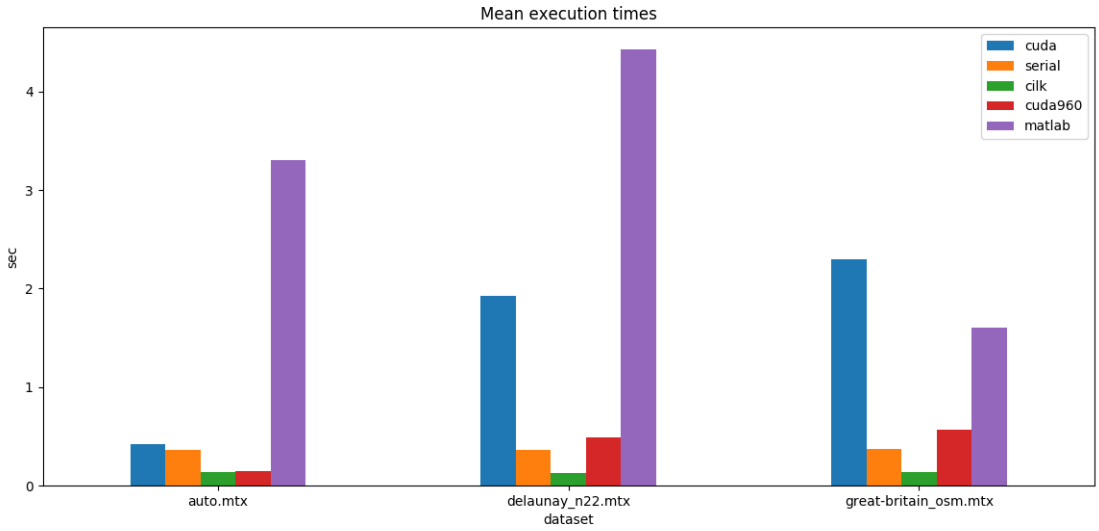
Listing 2: Without Conditional Blocks

5 Results

Hardware Specs

- CPU i7 7500U (2 cores 4 threads @2.5Hz 3.4Hz boost)
- GPU GT 940M 2gb
- GPU GTX 960 4gb

Graphs



Comments

Firstly, we will examine the algorithm without the parallelization. The CPU version achieves a steady 0.36ms for all three datasets. It's x10 faster on average from the matlab implementation that was given with the problem.

By using a simple CILK_FOR around the first for loop we achieved x3 speed increase on a 2 core/ 4 threaded CPU.

The CUDA implementation (benchmarked with GTX 960) of the same algorithm, is faster than the matlab implementation by a factor of 7.6 on average, is approximately as fast as the CPU version and always falls behind from the cilk-parallel implementation.

In defense of the cuda implementation, both GPUs are not suited for computational activities. GTX 960 has 16 SMMs opposed to the Tesla V100 with 84 SMMs and 5 times the CUDA cores.

Now we will compare the performance of the CUDA implementation between the datasets. auto 0.15ms < delaunay 0.49 < great-britain 0.57

A logical assumption would be the smaller the dataset the faster the execution. While the auto dataset is the smallest(3.3M nnz), delaunay has more non zero elements (12.5M nnz) than the great-britain (8.1M nnz).

The second metric we will examine is the sparsity.
auto $1.6e^{-5}$ < delaunay $7.15e^{-7}$ < great-britain $1.3e^{-7}$ This seems more correlated with the performance results. More sparse matrix means less non zero elements per row. Since there is one thread for every non zero element, there number of threads is less than the maximum wrap size. This is confirmed by looking the metrics from nvprof. Although the sm_efficiency(Multiprocessor Activity) is at 99% for all the datasets, the wrap execution efficiency varies between 11% and 40%. This is confirmed by looking the max non zero elements per row

- great-britain max nnz per row $= 7 \Rightarrow$ maximum wrap efficiency $= 7/32 = 21\%$, achieved $= 11\%$
- delaunay max nnz per row $= 17 \Rightarrow$ maximum wrap efficiency $= 17/32 = 53\%$, achieved $= 37\%$

6 Problems and Further Improvement

- The number of threads was configured with only the 3 datasets in mind. The number of threads per block is equal with the max non zero elements of each dataset. So this implementation cannot run for datasets with more than 256 non zero elements in a row.
- The size of the shared memory array is staticly allocated 64x64. So the implementation cannot run for more than 64 non zero elements in one row. The total allocated shared memory is 4KB, so we can allocate up to 120*120 to not exceed the 64KB shared memory limit
- In order to increase the wrap efficiency we could assign more than one row to each block.

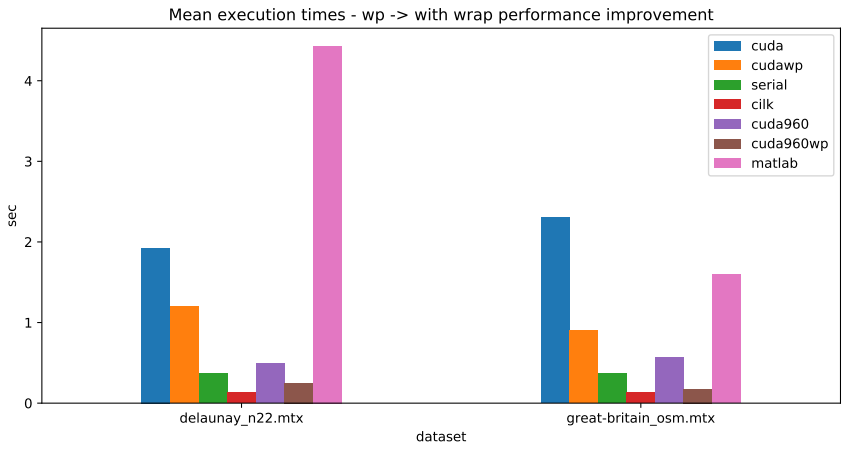
7 Buzzer Beater

The last moment, the improvement stated in the 3rd bullet point above has been implemented. Now each block processes more than one row, with a maximum of 8. (The limit of 64 threads, of max 8 rows per block and the size of shared memory where increased but yielded worst result for these datasets).

Each block processes $group_rows = max(8, div(64, max_n nz_per_row)$ and the number of blocks is $rows/group\_rows$. The wrap efficiency improved:

- delaunay $37\% \rightarrow 53.6\%$. 3 rows per block
- great britain $11\% \rightarrow 55.7\%$ 8 rows per block

and so did the execution times:



I didn't manage to improve the 'auto' dataset because the max non zero elements are close to the wrap size. If we added more threads to the kernel, the first wrap would be filled but the second wouldn't, thus there would be no improvment in the overall efficiency.

8 Verification

The number of triangles the programs returned were checked against the result from the matlab implementation for the 3 datasets.