

Accelerating FGLT in the GPU using CUDA

Tsirakis Orestis, AEM: 9995

Github: <https://github.com/Orestistsira/auth-parallel-and-distributed-systems-ex3>

Brief Summary:

FGLT is a multi-threading library for Fast Graphlet Transform of large, sparse, undirected networks/graphs. The graphlets are used as encoding elements to capture topological connectivity quantitatively and transform a graph $G=(V,E)$ into a $|V| \times 16$ array of graphlet frequencies at all vertices. In this exercise we only calculate and use the first five graphlets ($\sigma_0, \sigma_1, \sigma_2, \sigma_3, \sigma_4$) shown in figure 1, so we have a $|V| \times 5$ array.






Σ_{16}	Graphlet, incidence node	Formula in vector expression
 σ_0	singleton	$\hat{d}_0 = e$
 σ_1	1-path, at an end	$\hat{d}_1 = p_1$
 σ_2	2-path, at an end	$\hat{d}_2 = p_2$
 σ_3	bi-fork, at the root	$\hat{d}_3 = p_1 \odot (p_1 - 1)/2$
 σ_4	3-clique, at any node	$\hat{d}_4 = c_3$

Figure 1: Graphlets

The transformed data array serves multiple types of network analysis: statistical or/and topological measures, comparison, classification, modeling, feature embedding, and dynamic variation, among others. In this report we tried to accelerate the computation of these graphlets, given a sparse matrix A , by passing it to the GPU using CUDA and we will explain our parallelization and streaming choices.

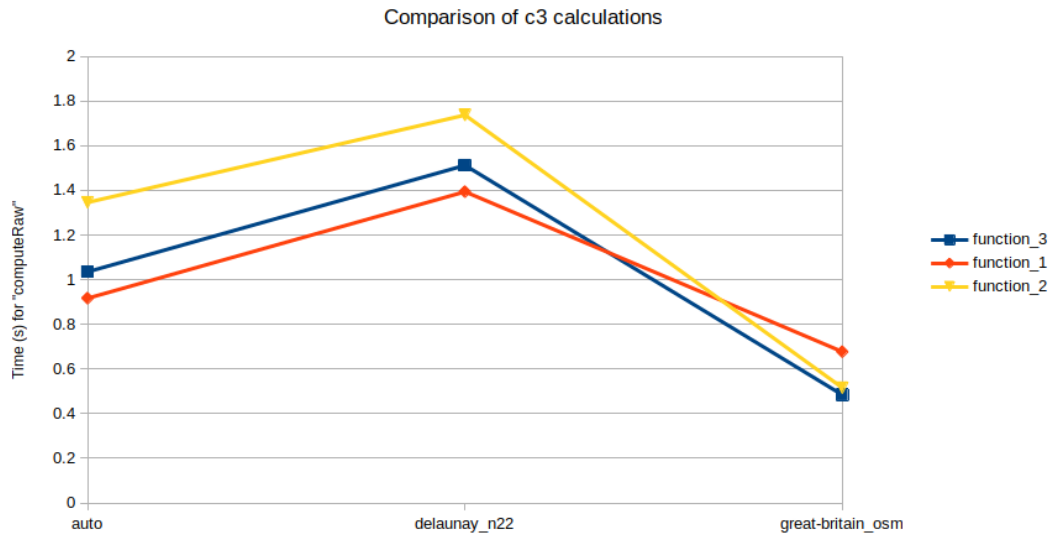
Sequential Implementation:

The sequential implementation of the solution is located in the file **“fglt_sequential.cpp”**. We created this file to show the computations needed as clear as possible and to find the best possible solution that can be parallelized easily on the GPU. After reading the graph file and converting it to a sparse matrix in CSC format (or CSR as the matrix is symmetric) we call the function **“computeRaw”** that calculates all the graphlet frequencies d_i from figure 1.

At first we calculate d_0, d_1, d_3 on a single loop iterating every vertex. For every vertex: $d_0 = 1$, as each one of them is a singleton, so e is a vector of ones with length $|V|$. We calculate d_1 by counting the number of elements in each row, as $p1 = A \times e$ and then we get $d_3 = 1 \odot (1 - 1)/2$ as an element by element multiplication.

For the σ_2 graphlet: $d_2 = A \times p1 - p1$ we calculate the matrix-vector multiplication products using another loop, as we need to calculate the $p1$ vector first.

At last, to calculate σ_4 by $d_4 = (A \odot A^2) \times e / 2$. We have 3 functions that calculate this graphlet: 1. **“compute_d4”** which is copied by the original FGLT implementation for finding d_4 , 2. **“compute_d4_2”** and 3. **“compute_c3”**. Functions (2) and (3) are our implementations of FRED G. GUSTAVSON’s algorithm for sparse matrix-sparse matrix multiplication [1]. Function (2) is very similar to the algorithm described by the paper and modified to only keep the elements of the multiplication result we need and (3) is simplified so that we make the computations faster and without any need for extra arrays, making it way easier to parallelize the function on the GPU.



We observe that the functions (1) and (3) give us the best computation times with the same results, and taking into consideration that the parallelization of (3) will be much easier, we chose this function for our parallel implementation and from now on the sequential times will be with this implementation.

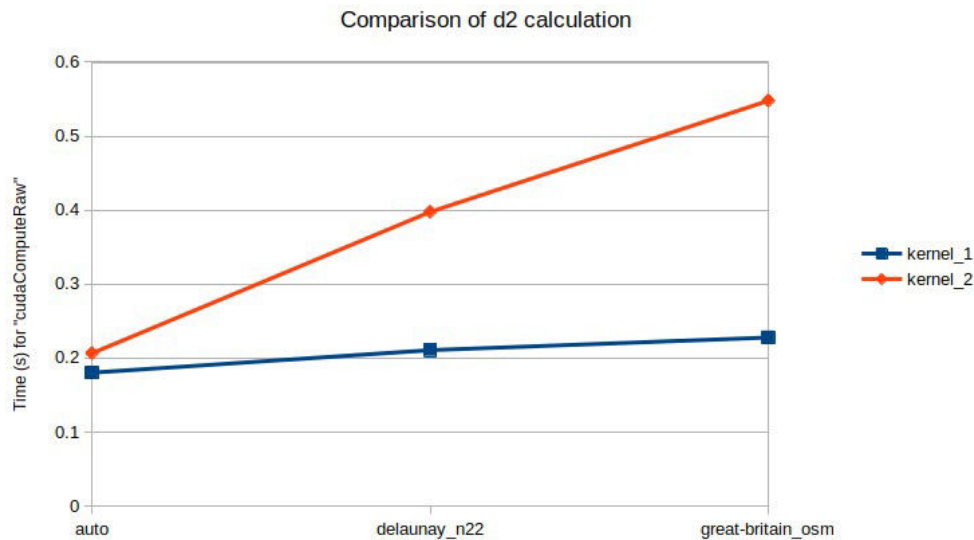
Parallel Implementation:

The parallel CUDA implementation of the solution is located in the file **“fglt_cuda.cu”**. This is the important part of the exercise, as the goal is to accelerate the computation of these graphlets. Similarly to the sequential implementation, after reading the graph file and converting it to a sparse matrix in CSC format (or CSR as the matrix is symmetric) we call the function **“cudaComputeRaw”** that calculates all the graphlet frequencies d_i from figure 1.

At first, to calculate d_0, d_1, d_3 we use a simple kernel **“compute_d0_d1_d3_Kernel”**, in which each thread calculates the frequencies for one vertex, using way we implemented it sequentially, after we transfer all the required data to the GPU.

Then for d_2 we came up with 2 kernels that compute the desired frequencies: 1. **“compute_d2_Kernel”** and 2. **“compute_d2_vector_Kernel”**. The kernel (1) is a simple sparse matrix-vector multiplication kernel that each thread calculates the dot product of one matrix row

with the vector. The kernel (2) unlike the previous implementation, which uses one thread per matrix row, requires coordination among threads within the same warp, as more than one thread works on the same matrix row. We use reduction to exchange data between registers on the same warp to avoid shared memory which is much slower [2].



Despite the use of warps and reduction the computation time does not reduce, and for some graphs it is even more than 2 times higher. So we decided to keep the first kernel, as it gives better results for these graphs and from now on the parallel times will be with this implementation.

At last, for d4 we use the kernel **“compute_c3_Kernel”** in which each thread executes the code inside the loop from **“compute_c3”** of the sequential implementation, as the frequencies for every vertex are independent with each other.

At the end of each kernel we gather the data needed from the device and we store it on the host, before printing the final results.

Correctness:

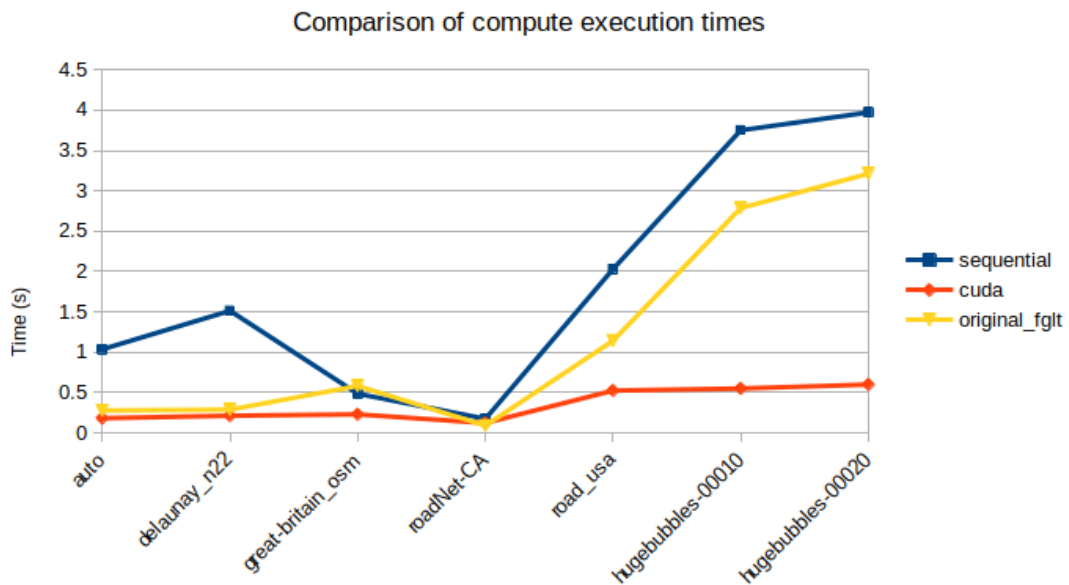
All the graphlets calculated by our project are the same that the original fglt library gives us for the testing graphs: (*auto.mtx*, *delaunay_n22.mtx*, *great-britain_osm.mtx*). We also check that both the sequential and parallel implementations return the same results by comparing the output files, using the **“tester.cpp”** file (more on README.md).

Results:

To compare our CUDA execution times with the times from the multi-threaded original fglt library, we deleted all the unrelated code of the graphlet computations that we do not calculate. These modified files are on the **“fglt_restricted”** folder on the repository of the exercise and they replaced the ones on the fglt original program to run the tests. The original fglt times are computed using OpenCilk with 12 threads and we also include the execution times of our sequential implementation.

Parallel and Distributed Systems

	Sequential	CUDA	Original FGLT (multi-threaded)
auto.mtx	1.0350s	0.1804s	0.2756s
delaunay_n22.mtx	1.5122s	0.2108s	0.2865s
great-britain_osm.mtx	0.4837s	0.2279s	0.5806s
roadNet-CA.mtx	0.1719s	0.1203s	0.0882s
road_usa.mtx	2.0312s	0.5227s	1.1416s
hugebubbles-00010.mtx	3.7515s	0.5487s	3.2149s
hugebubbles-00020.mtx	3.9738s	0.5983s	2.7872s



*All the graphs used are undirected and symmetrical.

We observe that the execution times from the original fglT library are actually accelerated by our CUDA implementation and for some graphs (road_usa, hugebubbles) the time reduction is even more noticeable.

**All tests are with AMD Ryzen 5 3600 6-Core Processor and NVIDIA GeForce GTX 1660 Ti
Driver Version: 525.60.13, CUDA Version: 12.0**

References:

[1]:

Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition FRED G. GUSTAVSON IBM T. J. Watson Research Center

[2]:

<https://medium.com/analytics-vidhya/sparse-matrix-vector-multiplication-with-cuda-42d191878e8f>