Accelerate FGlT with CUDA

Stylianos Andreadis

andreads@ece.auth.gr

https://github.com/AndreadisStel

About FGlT

FGlT is a C/C++ multi-threading library for Fast Graphlet Transform of large, sparse, undirected networks/graphs. The graphlets in dictionary Σ16, shown in Table 1, are used as encoding elements to capture topological connectivity quantitatively and transform a graph G=(V,E) into a |V| x 16 array of graphlet frequencies at all vertices. The 16-element vector at each vertex represents the frequencies of induced subgraphs, incident at the vertex, of the graphlet patterns. 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. The library FGlT is distinguished in the following key aspects. (1) It is based on the fast, sparse, and exact transform formulas, which are of the lowest time and space complexities among known algorithms, and, at the same time, in a ready form for globally streamlined computation in matrix-vector operations. (2) It leverages prevalent multi-core processors, with multi-threaded programming in Cilk, and uses sparse graph computation techniques to deliver high-performance network analysis to individual laptops or desktop computers. (3) It has Python, Julia, and MATLAB interfaces for easy integration with, and extension of, existing network analysis software.

More details in the paper.

Objective of this project

Implement the calculation of σ1, σ2, σ3, σ4 from the Table 1 using NVIDIA's parallel computing platform and programming model CUDA to parallelize FGlT code on the GPU.

Code structure and general ideas

For the purpose of the task, consider as input a sparse matrix A representing a large undirected graph. Graphs are originated from the SuiteSparse Matrix Collection and received in the code as a .mtx file (Matrix Market) in a COO format (rows, columns). To better handle the data and be able to access any point of the matrix A, without a time consuming process, it is better to keep the data on CSR or CSC format (it is the same because of the symmetry of matrix A); coo_to_csr is the function that implements the change of format in the code. A point to notice in that process is the single appearance of each edge at the mtx file in the COO format, which is problematic since we need to be able to look each edge from both sides for the CSR format. To resolve that problem we read the “COO edges” both forwards and backwards in order to get the correct result on CSR.

For the calculation of σ1, σ2, σ3, σ4 Auxiliary formulas from Table 1 are used and the main ideas are presented below:

- σ1 is the vector presenting the number of edges of each node. That is easily obtained by a subtraction of CSR pointers vector.
- σ2 is given by the Ap1 – p1 formula. To avoid a matrix-vector multiplication code we look directly at the number of “children” and sum them up. At the end, we subtract from the result the number of “children” of the “parent” node. (At an undirected graph Child and Parent doesn’t really make much sense, but the terminology is used here to make the formula easily understood) .
- σ3 is self explanatory; just as a note to point out that the symbol $\odot$ refers to the Hadamard product
- σ4 is the most complex graphlet frequency to calculate, because of the A^2 in the formula. To avoid the Matrix-Matrix multiplication, we only calculate the non-zero spots on the original matrix A, because in the $A \odot A^2$ product the zero spots will remain. After that we calculate the half-sum of each row and take the result.

Acceleration with CUDA

In the parallel code, the main idea remains the same, but instead of using ‘for’ loops to read the matrix, we spawn blocks of threads to run in the GPU in parallel and aim for them to have the smallest work load. To achieve that, we converted the formulas’ functions to __global__ functions callable by the GPU and we splitted the memory into device and host memory.

Results

Graphs	Sequential	CUDA	Speedup	Parallelization Speedup
auto	1.18 (1.73)	0.154 (0.65)	x2.66	x7.66
great-britain-osm	0.71 (2.10)	0.128 (1.54)	x1.36	x5.54
delaunay_n22	2.12 (4.14)	0.12 (2.16)	x1.91	x17.6
delaunay_n24	8.6 (16.67)	0.3 (8.5)	x1.96	x28.6
coPapersDBLP	14.10 (16.65)	0.9 (3.17)	x5.25	x15.67
com-Orkut	>12min (>12min)	107 (129)	-	-

Time in seconds, except com-Orkut Sequential

In the table above you can see two different times for each approach(Sequential&Parallel). The first time (e.g. 1.73) represents the total run time of the program and the other time (e.g 1.18) measures the amount of time spent on σ1, σ2, σ3, σ4 calculations. That is the part of the code that we implement in parallel using CUDA.

We can see that the speedup is pretty good even for the total time without the use of really large graphs and larger the graph gets the better the speedup becomes.

The most important thing is that the speedup looks really good, up to x15.67 times faster for the part of the code we pararellize.

Run the code

You can find the code in here: https://github.com/AndreadisStel/FGlTxCUDA/tree/main

To run the Sequential branch use:

- gcc src/serial.c -o bin/serial
- ./bin/serial [MatrixMarket.mtx]

To run the Parallel (main branch) use:

- nvcc src/parallel.cu -o bin/parallel
- ./bin/parallel [MatrixMarket.mtx]