Fast Graphlet Transform with CUDA

Assignment for the course of Parallel & Distributed Systems, ECE AUTh
Odysseas Sofikitis, 10130, sodyssea@ece.auth.gr, GitHub Repo

I. Introduction

A) The FGlT

The goal of the assignment is to implement *Fast Graphlet Transform* [1] for undirected graphs using *CUDA*. A sequential and parallel solution (using *OpenCilk*) is already given, based on which we discuss the acceleration achieved using a *GPU*.

We restrict our computations to the first four frequencies, shown in *Table 1*.

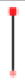
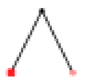
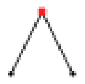
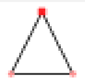
	σ_0	singleton	$\hat{d}_0 = e$
	σ_1	1-path, at an end	$\hat{d}_1 = p_1$
	σ_2	2-path, at an end	$\hat{d}_2 = p_2$
	σ_3	bi-fork, at the root	$\hat{d}_3 = p_1 \odot (p_1 - 1)/2$
	σ_4	3-clique, at any node	$\hat{d}_4 = c_3$

Table 1. Frequencies to be calculated

Vectors shown on the third column are calculated through the auxiliary table of [1]. These do not represent the final frequencies, but are called *raw frequencies* and are transformed to actual usable quantities, *net frequencies*, with some simple calculations.

B) Structure

Input graphs are handled in the "*mtx*" file, converting the *COO* format of *Matrix-Market* to *CSR* for better memory usage and faster calculations. Time needed for this is not accounted for, as it is done in the most optimized way, $O(m + n)$, and does not add any additional time compared to what is already required for reading the file.

The core file of the implementation is "*fglt*". It holds all functions needed for calculations, both in host and device. Although *cuSPARSE API* [2] could be used for the purposes of this assignment, it was not preferred as it occasionally adds severe time overhead and is not ideal to deal with the demanding memory management needed for graphs of higher order.

II. Parallelization

A) Device Memory

Appropriate memory management can be difficult and tests have shown that most of the running time consists of allocations and copying to device memory. Due to this, there was a large effort to achieve an implementation with the minimum pointers needed. Most of the calculations make use only of the rows-index pointer of the *CSR* format. Also, the fact that we handle adjacent matrices helps greatly, as there is no need for allocation of the values of non-zero elements. This significantly reduces memory complexity as well as computational complexity, resulting in an upper bound of $O(m + n)$.

B) Frequencies 0 through 3

The first four frequencies pose more of an observation problem than a programming one. Computational times are reduced through understanding the mathematical substance of the sizes represented. For example, d_1 could be calculated in a brute-force manner, as proposed in the auxiliary table of [1], by calculating the matrix-vector multiplication Ae, where A denotes the adjacent matrix and e a $n \times 1$ vector, filled with 1s. However, the result would be the degree of each vertex, which can be calculated from the row-index pointer of the *CSR* matrix almost instantly. By combining these observations and the auxiliary table, computational time is remarkably reduced.

Moreover, not only the calculation of these frequencies are inherently parallelizable on their own, but some of these can also be calculated simultaneously in the same kernel function call. As a result, d_0 and d_1 are calculated together, while d_2 and d_3, that both make use of p_1, are respectively calculated in the same function.

C) Frequencies 4

Calculating C_3 was the most challenging part of the assignment. The given implementation uses vectors of size $n \times n_p$, where n is the number of vertices and n_p the number of threads used. This way, each thread uses a unique part of the vectors for the calculations needed. Although the proposed way is typically faster, it is practically impossible to be implemented in a *GPU* environment as the number of threads are way more, making the memory complexity approach $O(n^2)$.

In addition, more memory problems can occur by implementing a matrix-matrix multiplication that become apparent by examining the formula of d_4:

$$d_4 = c_3 = (A \odot A^2)e/2 = C_3 e/2$$

If at least a vertex has a degree comparable to the total number of vertices (*hub node* [3]), A^2 can prove to be a dense matrix, despite A being sparse. For the reasons stated above, a new way to calculate d_4 was needed to be improvised.

Note that there is no need to actually store either A^2 or C_3. The only elements a_{ij} of A^2 that are going to be used are the ones for which $A_{ij} \neq 0$, due to the *Hadamard Product* [4]. Respectively, C_3 will be reduced to a vector for which every element c_i will be equal to $\sum_{j=1}^{N} (A \odot A^2)_{ij}$. Considering the above observations, the calculation of d_4 is implemented as following:

a) Each thread *i* is responsible for the calculation of *d[i]*.

b) We iterate through the non-zero elements of row *i*.

c) For every non-zero element a_{ij}, we compute the the corresponding element of A^2.

   i) A^2_{ij} is a linear combination of the row *i* and the column *j*. *A*, being an adjacent matrix of an undirected graph, is symmetric.

   ii) Instead of using column *j*, the multiplication is calculated by using row *j*, taking advantage of the *CSR* format.

   iii) If row *i* and *j* share an element in the same column, *d[i]* is incremented.

d) *d[i]* is divided by 2.

Steps described above result in storing neither the whole A^2 nor C_3. Instead, the sum of each row of C_3 is directly calculated. Finally, knowledge of the symmetry of the graph and sorted non-zero elements while constructing the *CSR* format contribute in significantly reducing unnecessary iterations

III. Results

Results are shown in *Table 2*. All graphs are *matrix-coordinate-pattern* from *SuiteSparse-Collection* [5]. Graphs vary both in size and density. Allocation, copy and free times are proportional to the size of the graph, while calculation time is affected by the density. For example, great-britain has x15 the number of vertices of auto, but only x2.5 more edges. This results in x8 more time for copying results to host but x16 less time for calculation. Because of this, total time stays almost the same for both graphs. Difference in calculation times can be explained by the final frequency results. The most time consuming computation is that of d_4, which is low in great-britain and larger in auto and delaunay, regardless of the total size of the networks.

Network	|V|	|E|	*OpenCilk*	*CUDA 256 threads / block*			
			4 thr.	**Allocation / Copy to device**	**Free / Copy to host**	**Calculation**	**Total**
auto	448K	3.3M	954 ms	5 ms	9 ms	84 ms	98 ms
great-britain_osm	7.7M	8M	939 ms	16 ms	75 ms	5 ms	96 ms
delaunay_n22	4M	12.5M	833 ms	16 ms	42 ms	39 ms	97 ms

Table 2. CUDA and OpenCilk execution times

CUDA implementation performs x10 faster than *OpenCilk*. However, most of the time is used in device memory management, for which nothing can be done to be improved. Thus, *CUDA* would show actual utility for the calculation of more graph frequencies, as allocation time would be the same, while calculation time would grow much slower than that of *OpenCilk*. On the other hand, broader tests on other networks proved that *OpenCilk* performs better on networks with high value of d_4, for the reasons explained previously in this paper.

IV. References

[1] Fast Graphlet Transform

[2] cuSPARSE

[3] Hub Node

[4] Hadamard Product

[5] SuiteSparse Collection