HiPC Programming Contest - 2021

GPU Track

Challenge: A High-Performance Efficient Implementation of K-Cliques Using GPUs

Motivation:

Detecting patterns with specific properties has been a great interest in the graph research community for several years. In particular, clique finding (definition is listed below) has applications in several domains, such as graph mining [1], social networking [2], and human brain networks [3]. Since many of these applications need to handle real-world graphs, developing an efficient implementation is a challenge.

Graphics processing units (GPUs) have become popular platforms for developing graph analytical applications for many years due to their massive thread-level parallelism. Despite the computational advantages provided by the GPUs, writing efficient parallel applications on GPUs remains a great challenge. It is especially more difficult for developing irregular applications like pattern detections in graphs as they (1) exhibit unpredictable memory-access patterns, (2) generate non-uniform workload, and (3) suffer from GPU memory constraints as they require to process real-world graphs. To get a deep insight into understanding these challenges, in this programming context, you will be designing an efficient implementation for computing the number of cliques in a graph using GPUs.

Definition:
Consider an undirected graph G = (V, E). A k-clique is a subgraph G'(V', E') of G, such that |V'| = k and G' is complete.

Challenge:
For a given undirected graph G (V, E) and k, the challenge is to compute the number k-cliques in the graph efficiently using a GPU.

Example:
The undirected graph shown in Figure-1 has six 3-cliques (abc, def, deg, dfg, efg, and hij) and one 4-clique (defg).

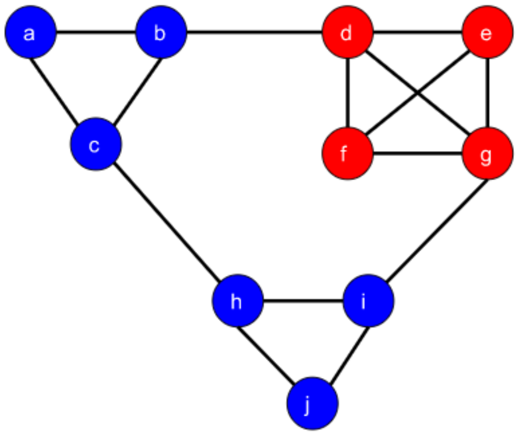

Figure-1: An example of an undirected graph with cliques.

Infrastructure:

1. Language: C/C++ and CUDA
2. You require a CUDA-capable GPU Device. If you do not have a personal GPU device, you can set up a cloud-based NVIDIA GPU and run CUDA programs through the Google Colab platform.
3. The instructions to set up Google Lab and run CUDA programs are given in the link.

Input:

1. File path to the input graph (G), which is symmetric, has no self-loops, and has no duplicated edges. The input graph is specified as the list of edges; a single space separates the source and destination vertices of each edge. You can find some of the input datasets in the below graph challenge website. Note that the edges in these graphs are separated by a tab space instead of a single space.
   http://graphchallenge.mit.edu/data-sets
2. Parameter k

Output:

1. Prints the number of k-cliques in G.
2. Prints the execution time of the application. Note that execution time should **exclude** the time (in milliseconds) for (a) reading the input graph from the file (b) loading it in the CPU memory, and (c) copying it from CPU to GPU memory.

Example: Assume that a graph shown in Figure-2 is named as **graph.txt** and saved in the **/home** directory.

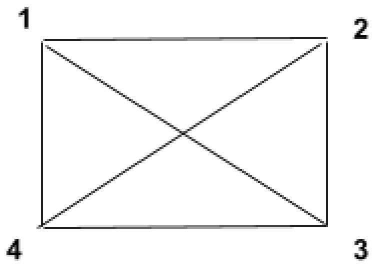

Figure-2: An example for input graphs.

The contents of **graph.txt** are:

```
1 2
1 3
1 4
2 1
2 3
2 4
3 1
3 2
3 4
4 1
4 2
4 3
```

Input-1:
/home/graph.txt
4 //Denotes the clique size

Output-1:
1
Execution time: 2 ms

Input-2:
/home/graph.txt
3 //Denotes the clique size

Output-2:
4
Execution time: 5 ms

Submission Instructions:

1. Submit your source code in the form of git-hub repository
2. The repository should contain a readme with the following information
    a. Required GCC and NVCC (with versions)
    b. Any other software with instructions or references to set it up
    c. Compilation instructions
    d. Execution instructions
3. The code should run on a Linux machine
4. Submissions that do not compile/run will be disqualified
5. A technical report containing two (2) single-spaced double-column pages using 10-point size font on 8.5×11 inch pages (IEEE conference style), **excluding references**. The report should contain a summary of your contributions, with the following sections: (1) Abstract, (2) Introduction, (3) Approach/Methodology, (4) Experimental results, (5) Conclusion, (6) References.

Note: Submission portal will be enabled soon.

References:

1. Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15). Association for Computing Machinery, New York, NY, USA, 425–440. DOI:https://doi.org/10.1145/2815400.2815410
2. M. Naim, F. Manne, M. Halappanavar and A. Tumeo, "Community Detection on the GPU," 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Orlando, FL, 2017
3. D. Meunier, R. Lambiotte, A. Fornito, K. D. Ersche, and E. T. Bullmore. Hierarchical modularity in human brain functional networks. Frontiers in Neuroinformatics, 37(3), 2010

4. Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in Sparse Real-World Graphs*. In Proceedings of the 2018 World Wide Web Conference (WWW ’18). International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 589–598. DOI:https://doi.org/10.1145/3178876.3186125
5. Irene Finocchi, Marco Finocchi, and Emanuele G Fusco. 2015. Clique counting in mapreduce: Algorithms and experiments. Journal of Experimental Algorithmics (JEA) 20 (2015), 1–7.
6. GeeksForGeeks, “Sparse Matrix Representations | Set 3 (CSR)”, https://www.geeksforgeeks.org/sparse-matrix-representations-set-3-csr/
7. “Sparse Matrix”, https://en.wikipedia.org/wiki/Sparse_matrix#Compressed_sparse_row_(CSR,%20_CRS_or_Yale_format)
8. CUDA Programming Guide: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html