Parallelized algorithm for counting k-cliques

Darsh Kaushik
Department of Computer Science and Engineering
National Institute of Technology Silchar
Assam, India
darsh.kaushik@gmail.com

Rishu Kumar
Department of Computer Science and Engineering
National Institute of Technology Silchar
Assam, India
rishu110067@gmail.com

***Abstract*—With imminent advancements in GPU architecture and parallel programming APIs, high-performance parallel implementations of algorithms have made a significant impact towards efficiently solving parallelizable problems. One such problem, having similar potential, is finding the total number of k-cliques in a graph, which has prominent applications in data mining and social networking amongst other fields. But despite the recent advancements in computational advantage provided by GPUs, it is usually not a straightforward task to write parallel implementations. In this paper, through our participation in the HiPC Programming Contest (GPU track), we present our parallelized implementation for counting k-cliques in a graph. We explore the advantages of multithreading using CUDA kernels and techniques like Dynamic Parallelism to tackle the recursive nature in the problem.**

***Index Terms*—Parallel programming, k-clique counting, GPU, CUDA, HPC**

I. Introduction

A clique in an undirected graph is a subset of vertices such that any two vertices in that subset are adjacent two each other. So effectively, a k-clique is a complete subgraph having 'k' vertices. Cliques were initially used by sociologists to measure social cohesiveness before the advent of computers [1].

Due to computational bottlenecks in computing the exact number of k-cliques in a graph, there have been studies [2], [3] devising methods for approximating this value, such as techniques based on technique based on Turan's theorem. Others, however, developed algorithms for computing the exact count of k-cliques like in [4] for the MapReduce framework. Similarly the algorithm proposed in [5].

In the coming sections we too propose our methodology for computing the exact number of k-cliques given 'k' and the edge list of a graph on GPU. Our source code can be found in the 'Final Source Codes" directory of our github repository[1]. Two versions of parallelized GPU codes have been found, each having its advantages for different kinds of inputs.

II. Methodology

First, we started by creating a non parallel optimized solution on the CPU, then parallelized it for GPU and during the process we explored various techniques for optimization.

[1]https://github.com/darshkaushik/HiPC-Programming-Contest-2021

A. Non-parallelized implementation for CPU

Our non-parallel CPU implementation 'optimizedCPUsolution' is inspired but not completely similar to any of the other works cited before.

First observation is the fact that only those vertices can make k-cliques which have degree equal to more than k - 1 would be eligible to be a part of a k-clique. Let's call these vertices 'important vertices' and store them in the list 'imp' of length 'imp_size' which is the total number of important vertices. The important vertices are stored in ascending order of their vertex IDs so that later we could leverage this fact for finding intersection of lists using binary search.

After finding the list of 'important vertices' then we pick a vertex from this list one by one, denoting the first vertex (vertex with the smallest vertex ID) of the k-clique. Then we find the intersection of 'imp' and the adjacency list of the root vertex. This intersection list has the vertices that can be chosen as the second vertex (second-smallest vertex ID) of the k-cliques so this intersection list represents the options that we have for the second vertex to choose from as the third vertex of the k-clique. We repeat the process above to find the third vertex of the k-clique.

We recursively pick ith vertex from the current intersection list one by one, then find the intersection of the adjacency list of the chosen vertex with the current intersection list resulting in a new intersection list which represents the options that we have for the (i+1)th vertex of the k-clique. We repeat this process of picking vertices and finding intersection lists until we pick the k-th vertex. When we pick the k-th vertex, which signifies that we have successfully found a k-clique, we increment a globally kept variable 'cnt' by 1. An optimization that we chose to do here is that instead of incrementing 'cnt' by 1 whenever we pick the k-th vertex, we can increment 'cnt' by the size of the intersection list after picking the (k-1)th vertex of the k-clique.

To list all the steps involved in 'optimizedCPUsolution':

- As the number of vertices is not given in the input, we calculated it using the largest vertex ID present in the edge.
- We used a 'map' to remove the duplicate edges thus allowing the solution to also work for inputs where an edge can be present multiple times and where the vertices

are indexed from 0 instead of 1.

- To avoid counting the same k-clique again we made the graph directed where each edge is directed from the smaller vertex to the larger vertex.
- We made a recursive function to simulate this recursive process of picking a vertex and finding an intersection list.

B. *Parallelized implementation for GPU*

We have provided two parallelized solutions. 'GPUsolution1' is very similar to the optimized CPU solution. In this solution we parallelized :

- Degree and 'v_size' calculation in 'degree' kernel, each thread incrementing the degrees contributed by each edge.
- Adjacency list computation in 'adj' kernel, each thread adding the larger vertex ID to the adjacency list of smaller vertex ID both of which are end vertices of an edge.

We tried to use Dynamic Parallism by calling child kernels inside parent kernels to parallelize the recursive function, but it didn't give good performance.

'GPUsolution2' on the other hand is not very similar to its non-parallel counterpart. This implementation uses the binary encoded adjacency list, 'G_linear', instead of the CSR representation used in 'GPUsolution2' and 'optimizedCPU. In this solution we have parallelized :

- degree calculation in the 'degree' kernel, each thread incrementing the degrees contributed by each edge
- iterative version of 'find' function in 'find_iterative' kernel, each thread choosing one root vertex from important vertices 'imp' and traversing the recursion subtree of finding intersections of root with binary encoded adjacency list of other vertices but in an iterative manner.
  In other words, each thread will find the total number of cliques which have the root vertex as the smallest vertex ID in that clique.

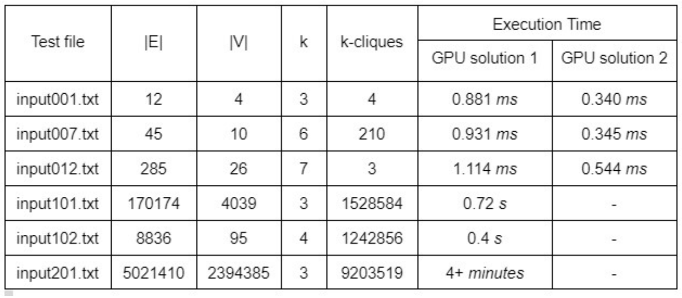
The 'GPUsolution2' works only for small inputs and is much faster than 'GPUsolution1' as can be seen in the next section (for larger inputs the kernels terminated themselves mid-execution due to reasons we were unable to comprehend and debug in spite of extensive testing). The 'GPUsolution1' on the other hand works for test cases of any size specified in the constraints of the HiPC 2021 GPU track, but its execution time is not as fast due to heavy usage of the atomic function 'atomicAdd' and extensive copying of data between the host and device.

III. Experimental results

In figure 1 we have a comparision of the execution times of our 2 GPU solutions.

IV. Conclusion

We present an optimised CPU solution for k-clique counting and two GPU solutions built over the optimised CPU solution. The first GPU solution parallelized some part of the optimised



Test file	|E|	|V|	k	k-cliques	Execution Time	
					GPU solution 1	GPU solution 2
input001.txt	12	4	3	4	0.881 *ms*	0.340 *ms*
input007.txt	45	10	6	210	0.931 *ms*	0.345 *ms*
input012.txt	285	26	7	3	1.114 *ms*	0.544 *ms*
input101.txt	170174	4039	3	1528584	0.72 *s*	-
input102.txt	8836	95	4	1242856	0.4 *s*	-
input201.txt	5021410	2394385	3	9203519	4+ *minutes*	-

Fig. 1. The test files are present in the Development / Test_Files in the github repository.

CPU solution. In the second GPU solution, we took the parallelization further ahead by parallelizing the 'find' function by making it iterative but due to some reason it worked only for small test inputs and was not able to run on the larger inputs. As we didn't have access to A100 GPU when we finished our GPU solution 2, we were only able to test it on colab where it didn't work well. Theoretically it should work faster but we think that due to the limitations of the GPU on colab it didn't work well but may work well for GPUs with high compute capability. We also explored different techniques such as dynamic parallelism to parallelize our find function but due to limitations of dynamic parallelism our implementation using dynamic parallelism didn't work well. Overall, we were able to come up with interesting ideas to solve the problem but the lack of resources and implementation techniques was a major obstacle.

Acknowledgement

The authors would like to thank the organisers of HiPC Programming Contest (GPU track) for conducting the contest, providing access to A100 GPUs and also organizing the tutorial sessions.

References

[1] Stanley Wasserman and Katherine Faust. 1994. Social network analysis: Methods and applications. Vol. 8. Cambridge university press.

[2] Shweta Jain and C Seshadhri. 2017. A Fast and Provable Method for Estimating Clique Counts Using Turán's Theorem. In Proceedings of the 26th International Conference on World Wide Web. International World Wide Web Conferences Steering Committee, 441–449.

[3] Michael Mitzenmacher, Jakub Pachocki, Richard Peng, Charalampos Tsourakakis, and Shen Chen Xu. 2015. Scalable large near-clique detection in large-scale networks via sampling. In Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, 815–824.

[4] 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

[5] 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

Compilation and Execution Instructions

Required gcc version: 8.1 or above
Required nvcc version: 11.2 or 9.2

Follow these steps to run the solution on a GPU device:

1. Clone or download this Repository.
2. The final source codes are present in the `Final Source Codes` folder. It has three solutions: optimised CPU solution, GPU solution 1 and GPU solution 2.
3. To run the solutions, create a repository named `test` (it can be named something else as well). Change the working directory to `test`.
4. Create a `solution.cu` file and a `input.txt` file. Copy the solution to be executed in the `solution.cu` and the input in the `input.txt`. Make sure that the input format is correct.
5. Execute the `solution.cu` file using the command below. The output will be printed on the command-line screen.

   ```
   $nvcc solution.cu
   $./a.out
   ```

   If this doesn't work use:

   ```
   $nvcc -arch=sm_35 -rdc=true solution.cu
   $./a.out
   ```

   Here `arch=sm_35` denotes the compute capability of the gpu. For compute capability of 6.0 use `sm_60`, similarly for 8.5 use `sm_85`.

Input Format

The first line of input should contain two integers - the first one for the number of edges in the graph and the second one for the value of k for which the number of k-cliques needs to be found.

Next lines contain two integers each which represents an edge.

Output Format

The number of k-cliques found along with the execution time is printed.