GraphLink - Parallel Implementation of Graph Algorithms

ABSTRACT

Graph clustering algorithms have applications in diverse fields, including but not limited to social network analysis, machine learning, computer vision, bioinformatics, and many more. Although these algorithms work well for small-sized graphs, but as the graph size increases, the computational complexity of clustering these graphs also increases. Traditional algorithms are often unable to handle such large graphs in a reasonable amount of time. On the other hand, executing multiple processes parallelly can significantly reduce the run time and resolve the issue of high computational cost. Therefore, our developed library 'GraphMaze' utilizes CUDA kernel using GPU to parallelly implement two widely used algorithms - 'k-means' and 'embedding algorithm through deepwalk'. We also evaluated the performance of our library by comparing the parallelized k-means with traditional k-means and received positive results. But, in the case of embedding algorithm, satisfying results were not observed since it already gave good results on the CPU because of the optimized implementation of the 'Gensim' library. Thus, researchers and practitioners in various fields can utilise our newly developed library for efficient and accurate analysis of large graphs and can also be used in other applications.

I. KEYWORDS

Graph Clustering, CUDA, GPU, Parallel Processing, K-means, GraphLink, GraphMaze, Python package.

II. INTRODUCTION

Graph clustering is an important technique used in different fields to identify clusters or communities of nodes in a graph that share common characteristics. For example, they're used to identify groups of individuals with similar interests or behaviours in social network analysis, and in bioinformatics domain, they're used to identify groups of genes that are co-regulated or involved in similar biological pathways.

However, with an increase in the size of graph, clustering becomes computationally expensive, and traditional algorithms utilization power becomes limited due to their inability to handle the workload in a reasonable amount of time. This is where our developed Python library comes into the picture which solves the problem by providing functionalities to execute these graph clustering algorithms parallelly by utilizing CUDA kernel using GPU programming.

Executing large, sparse graphs using traditional algorithms is computationally costly and 'parallelization' can be leveraged to develop a good solution for such a problem. Parallel algorithms distribute the workload across multiple processors/cores and facilitate faster computation of large datasets.

We have developed a library 'GraphMaze' similar to the 'iGraph' library - a popular open source graph analysis library available on Python but containing parallel graph algorithms which will improve the computational performance. We have used C++ and Python programming languages to implement the same. C++ is used to implement the algorithm to be executed on the CUDA kernel and Python is used to implement the frontend part. CUDA is a parallel computing platform provided by NVIDIA which harnesses GPUs for parallel computations.

Our implemented library thus provides the functionality to faster process graphs such as the Facebook graph dataset and enables exploration of new research questions that were previously computationally difficult or unfeasible.

III. METHODS

- **GraphMAZE:**

A library implemented that consists of parallel algorithms and utilizes GPU. With the help of CUDA toolkit, a kernel written in C++ is executed. The library is implemented using python code written to support running CUDA kernel written in C++ to perform the task required to perform the parallelization. One can easily use the library created as a python wheel for own use to perform parallel computing on graphs. Following are the steps to download the library and get started with parallel processing in just three steps:

1. Download the wheel package like any another python package from the source as follows:

   https://github.iu.edu/manaagra/graphlink/blob/master/graphmaze/dist/gmaze-0.1.0-py3-none-any.whl
2. Now install the wheel package in the directory where one wishes to test their graphs using the command: **pip install <package-name>**
3. Once the package is installed, the methods are available to use from the import command. Python code for example is as follows:
   from gmaze import kmeans
   Cluster_labels=kmeans.cluster(points, k=2, max_iter=10)

For now, two algorithms are written using parallel computing where one is giving standard performance and the other is still in the development phase. The implementation of k-means algorithm and word2vec algorithm has been done in parallel fashion. Time complexity of traditional k-means algorithm is Ω(n*d*k*i) where n is the number of nodes in the graph, d is the dimension of the embedding algorithms, k is the number of k-means clusters and i is the number of maximum iterations it takes for the algorithm to converge. After parallel computing the complexity of the new algorithm decreases to Ω(d*k*i) achieving thirty times better speed efficiency on average in comparison to the traditional k-means algorithm. We will further explain how the phenomenon takes place. Let us look at the pseudocode of the new k-means algorithm implemented through CUDA programming.

- Pseudocode for parallel k-means

1. Initialize centroids randomly
2. Repeat until maximum iterations reached:

   a. Assign each data point to the nearest centroid using a CUDA kernel that runs in parallel on the GPU

   b. Update the centroids based on the new assignments using another CUDA kernel that runs in parallel on the GPU

   c. Copy the updated centroids and cluster assignments from the GPU memory to the CPU memory

   d. Set the previous assignments to the new ones and go back to step 2a

To understand what's going on behind the scene, we can look at how the k means kernel of the above example looks like:

```cpp
#include <cuda_runtime.h>
#include <stdio.h>

#define BLOCK_SIZE 256

extern "C"
    __global__ void kmeans_kernel(
    float* points,
    float* centroids,
    int* cluster_assignments,
    int k, int d, int n) {

    int idx = blockIdx.x *
        blockDim.x +
        threadIdx.x;

    if (idx < n) {
        float min_dist = 10000000000;
        int min_idx = -1;
        for (int i = 0; i < k; i++) {
            float dist = 0.0f;
            for (int j = 0; j < d; j++) {
                float diff = points[idx * d + j] -
                    centroids[i * d + j];
                dist += diff *
                    diff;
            }
            if (dist < min_dist) {
                min_dist = dist;
                min_idx = i;
            }
        }
        cluster_assignments[idx] = min_idx;
    }
}
```

From the above figure, we can conclude that we can find the k-means cluster of each embedded graph node in a parallel way by assigning a thread to each node. The thread then performs the operation of finding the appropriate cluster label by calculating the distance between each cluster and the data point. This is how the parallel computing library works.

IV. EXPERIMENTAL SETUP

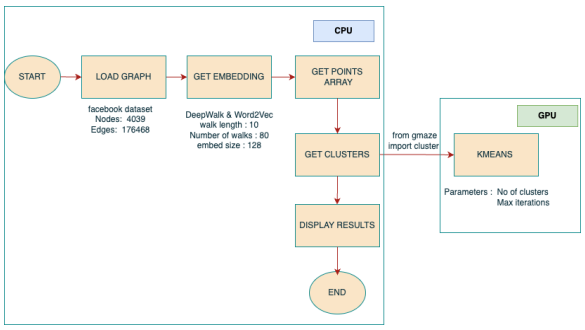
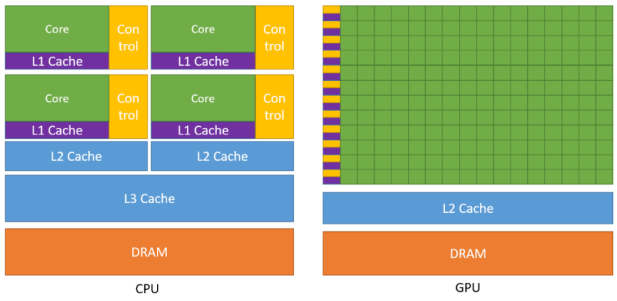


Fig 1 : Implementation Flow chart

In order to evaluate our parallel computing library for the k-means algorithm, we conducted several tests on Big Red 200, which consists of 64 GPU-accelerated nodes. Each node has a single 64-core, 2.0 GHz, 225-watt AMD EPYC 7713 processor, 256 GB of memory, and four NVIDIA A100 GPUs. We utilized the Facebook dataset [2] from https://snap.stanford.edu, which contains 4039 nodes and 176468 edges. We loaded the graph and generated graph embeddings using DeepWalk and Word2Vec algorithms. Then, we created a points array, which was passed as a parameter to the k-means algorithm of our library. As depicted in the flow chart, all of this processing occurred on the CPU, while the execution of the k-means algorithm using our library took place on the GPU. Our developed algorithm is configurable in that the number of clusters and maximum iterations can be passed as parameters to the k-means function of the gmaze library.



The GPU Devotes More Transistors to Data Processing

The figure presented in reference [3] illustrates why GPUs are able to execute faster than CPUs, which is rooted in their technical design. Unlike CPUs, which are designed to rapidly execute sequential instructions, GPUs allocate more space for data processing. This design feature allows GPUs to avoid the latency associated with accessing cache, which is a bottleneck for CPUs.

To make a comparison, we conducted an experiment in which we ran both the k-means algorithm from our gmaze library and the traditional k-means algorithm from the gensim library on Big Red 200. We used the same Facebook dataset with the same graph embedding for both algorithms and captured the execution time for each.

V. RESULTS AND DISCUSSION

We observed that our parallel implementation of kmeans algorithm on GPU provides comparable results to that of the traditional kmeans algorithm but at significantly higher speeds on average.

In this section we show the speed difference between the traditional kmeans approach and our parallel implementation of kmeans algorithm on GPU so that we get an idea about what makes the parallel implementation better than the traditional kmeans approach. We did some performance comparison tests on different kmeans parameters on the facebook dataset, so that we can compare both the algorithms. We got the following results:

- Performance Comparison - K

First we tried to vary the K value in kmeans algorithm which is known to be a hyperparameter that needs to be provided to the algorithm before running it. K is basically the number of clusters that need to be formed from the data. We tried executing the kmeans algorithm and the parallel kmeans algorithm for different k values of 5,10,15,20,25 on the facebook graph dataset and we got the running times for these k values as illustrated below.

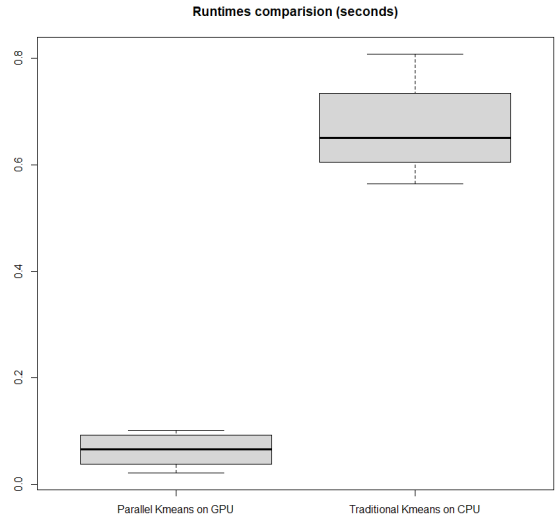


Fig 1

k	Parallel_Kmeans_GPU_runtimes	Traditional_Kmeans
5	0.0225	0.605
10	0.0385	0.565
15	0.0655	0.808
20	0.0900	0.735
25	0.0970	0.651

Fig 2

As we can observe there is a significant difference in run times (approximately 8-12x compared to traditional k means)

- Performance Comparison - Max Iterations

Then we tried to check the performance difference when we varied the number of max iterations, which is the number of iterations the algorithm runs for. We tested the kmeans algorithm and the parallel kmeans algorithm for different maximum iteration values of 5,10,15,25,50 on the facebook graph dataset and we got the following running times for these values.

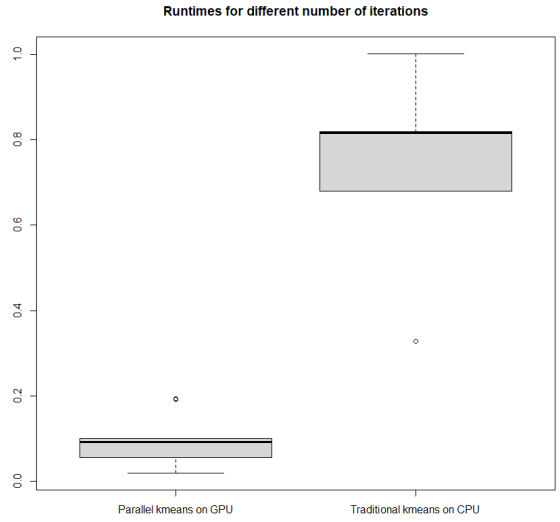


Fig 3

max_iters	Parallel_Kmeans_GPU_runtimes	Traditional_Kmeans
5	0.0200	0.328
10	0.0965	0.819
15	0.0565	0.680
25	0.0950	1.001
50	0.1930	0.816

Fig 4

We can observe a significant difference in run times and once again we can see that the parallel implementation of kmeans is better than the traditional algorithm.

- Performance Comparison - Embedding Size

Next we tried to check the impact of the embedding size on the performance, as the embedding size/dimensionality would play an important role in determining how long the algorithm runs for. We tested the kmeans algorithm and the parallel kmeans algorithm for different embedding size values of 64,128,256,512 on the facebook graph dataset and we got the following running times for these values.

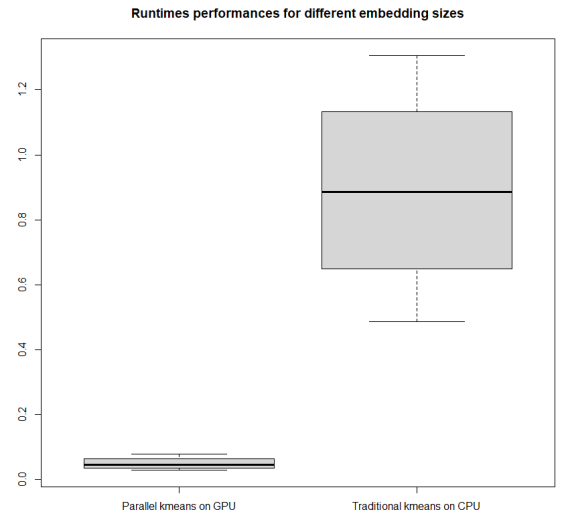


Fig 5

embedding_size	Parallel_Kmeans_GPU_runtimes	Traditional_Kmeans
64	0.0295	0.486
128	0.0390	0.810
256	0.0520	0.960
512	0.0770	1.306

Fig 6

Here we can observe a clear trend of increasing run time for increasing embedding sizes in both the algorithms. But if we observe the boxplot we can see that the parallel implementation performs better than the traditional algorithm on varying embedding sizes.

VI. CONCLUSION

In this project, we implemented a parallel kmeans algorithm which would be able to leverage the power of GPU and be able to deliver comparable results to that of the traditional kmeans algorithms but in much lesser time. We observed a speed increase of upto 30x compared to the traditional kmeans algorithm. We believe that our algorithm can have applications in big data analysis, clustering gene expression data and targeted advising domains due to the better time performance offered.

VII. FUTURE WORK

We would like to try out our parallel kmeans algorithm on much larger datasets in the future and try to optimize the algorithm better and look for options on further optimizing the algorithm. We also plan on implementing parallel GPU versions of several other graph algorithms in the future and include it in our graphMaze package. We would also be looking forward to try and put our package on the pypi repository so that it would be publicly available, after adding more algorithms to our existing graphMaze repository.

VIII. TEAM MEMBER CONTRIBUTIONS

[1] Mana Agrawal - Developed the graphmaze library to generate parallel computing algorithms for graphs like k-means on embedded nodes

[2] Diksha Singh - Worked on parts of GraphMaze algorithm and cluster visualization on the Facebook graph dataset.

[3] Vaibhavi Patel - Implemented the code to load,embed graph and compare execution time of executing kmeans algorithm of gmaze library and traditional kmeans.

[4] Sri Narasimha Nihanth Vuddanti - Worked on developing and performing the performance analysis tests of the algorithms on Facebook dataset,visualization of results and recording the results and observations of the experiments performed.

IX. REFERENCES

[1] https://kb.iu.edu/d/brcc

[2] https://snap.stanford.edu/data/ego-Facebook.html

[3]https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[4] https://igraph.org/