numbaMCL: An attempt to parallelize Markov Chain Clustering (MCL) algorithm

Guangyuan(Frank) Li

Introduction

In this project, I attempted to parallelize Markov Chain Clustering (MCL) algorithm using NVIDIA CUDA drive-empowered GPU resources. To be specific, I first implemented the sequential version using the basic python numpy package. I then utilized the python numba package to write three customized kernel functions to parallelize the primary three steps in the original Markov Chain Clustering (MCL) algorithm. Unfortunately, the benchmark result from a dummy testing data (a random generate adjacency matrix with 1024 x 1024 dimensions) shows that GPU version actually takes longer time than CPU version, it is attributed to the overhead of repetitive host-to-device memory allocation which drastically hides the potency of GPU programming. Although trying to find a python-based solution to solve this bottleneck more intelligently by using shared memory and dynamic parallelism, the currently limited functionality of python numba (I will discuss later in the paper) hinders me from completely designing a superior GPU version of MCL. In this paper, I will first introduce Markov Chain Clustering (MCL) algorithm, followed by my design and optimization approach. I will show the time consumed for both CPU version and GPU version testing on Ohio Supercomputer. Finally, I will share the difficulties I encountered when using the python numba package, which is officially recognized by NVIDIA as a CUDA extension and the future work I could possibly work on.

Markov Chain Clustering (MCL) is one type of graph clustering algorithm that has been widely used in a lot of bioinformatics applications[1]. It simulates a random walk process and assumes that if a path starting from node A frequently ends up at node B, then node A and node B share underlying commonality and may form a unique cluster. Concretely, It contains three main steps -- expansion, inflation and convergence checking. The first step has the effect of strengthening strong connections and weakening the lower edges[2]. The convergence checking step determines if the iteration will continue or end up. These three steps also represent the most computationally-demanding steps when the dimension of input data scales up (i.e. millions of data points), running the sequential version of MCL and iteratively checking convergence could be intensive. To ease this process, I attempted to use CUDA kernel function to parallelize the expansion, inflation and convergence checking step, which can ideally achieve a faster performance in the large dataset. The working complexity will be $O(n^2)$ for an input adjacency matrix of dimension N x N, times the rounds of iteration k. The step complexity, instead is O(3) since there are only three sequential steps in general, and also needs to multiply by the needed rounds of iteration k. It suggests that a parallel version of MCL algorithm theoretically can largely reduce the running time. Two similar attempts to parallelize the MCL algorithm[2,3] will be briefly discussed in the last part of the paper.

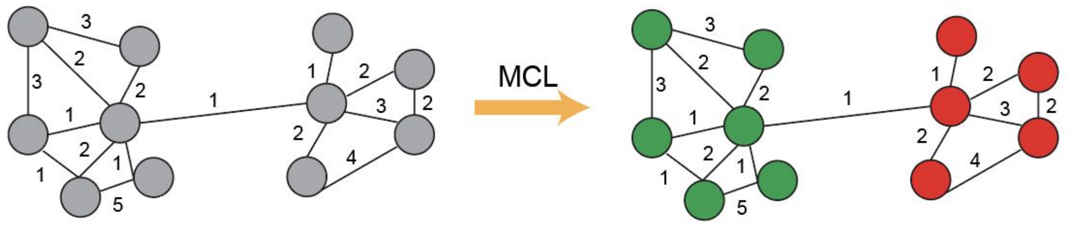
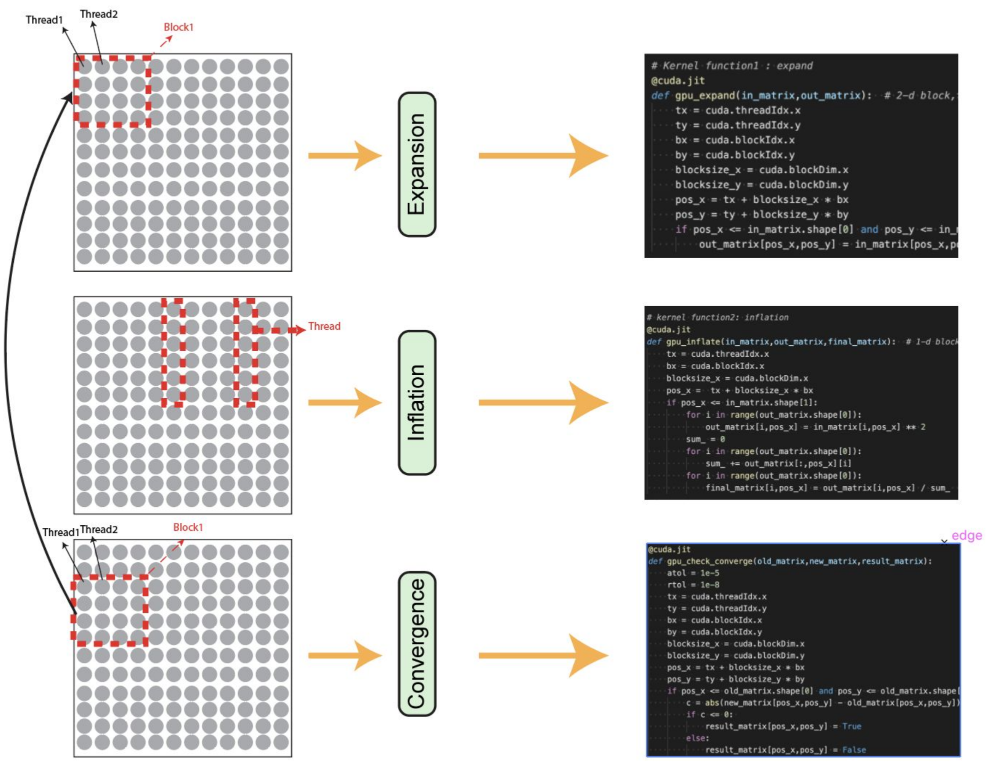


Figure1 Schema of Markov Chain Clustering (MCL) algorithm. The value on each edge represents the strength of node-node connection, MCL eventually partitions the network into two isolated communities (green dots and red dots).

<u>Design and optimization approaches</u>

In order to parallelize the MCL algorithm, I decided to separate the flow into three steps corresponding to expansion, inflation and check convergence. The operation of expansion essentially takes the power of e (e is an user-defined expansion parameter, here I chose e=2) of each element in the matrix. I employed 2D blocks and threads such that each thread computed the power of each element, this is achieved through gpu_expand kernel function. The resultant matrix will be sent back to the host. Next, I employed a 1D block and thread to implement the inflation step due to the fact that a column-wise normalization is needed in this step, therefore per-element wise fashion is not capable of communicating with each other and access shared memory to fulfil the goal of normalization. A check convergence step was conducted to check if the original matrix (prior to expansion and inflation) is convergent to resultant matrix (after the expansion and inflation), this was implemented in a per-element fashion such that a boolean value was returned indicating if a certain element is convergent or not. The final boolean matrix will transfer to host and determine if convergence has reached, if not we will repeat the process. Otherwise, the MCL algorithm is completed.



```
# Kernel function1 : expand
@cuda.jit
def gpu_expand(in_matrix,out_matrix):  # 2-d block,
    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bx = cuda.blockIdx.x
    by = cuda.blockIdx.y
    blocksize_x = cuda.blockDim.x
    blocksize_y = cuda.blockDim.y
    pos_x = tx + blocksize_x * bx
    pos_y = ty + blocksize_y * by
    if pos_x <= in_matrix.shape[0] and pos_y <= in_
        out_matrix[pos_x,pos_y] = in_matrix[pos_x,p
```

```
# kernel function2: inflation
@cuda.jit
def gpu_inflate(in_matrix,out_matrix,final_matrix):  # 1-d block
    tx = cuda.threadIdx.x
    bx = cuda.blockIdx.x
    blocksize_x = cuda.blockDim.x
    pos_x =  tx + blocksize_x * bx
    if pos_x <= in_matrix.shape[1]:
        for i in range(out_matrix.shape[0]):
            out_matrix[i,pos_x] = in_matrix[i,pos_x] ** 2
        sum_ = 0
        for i in range(out_matrix.shape[0]):
            sum_ += out_matrix[:,pos_x][i]
        for i in range(out_matrix.shape[0]):
            final_matrix[i,pos_x] = out_matrix[i,pos_x] / sum_
```

```
@cuda.jit
def gpu_check_converge(old_matrix,new_matrix,result_matrix):
    atol = 1e-5
    rtol = 1e-8
    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bx = cuda.blockIdx.x
    by = cuda.blockIdx.y
    blocksize_x = cuda.blockDim.x
    blocksize_y = cuda.blockDim.y
    pos_x = tx + blocksize_x * bx
    pos_y = ty + blocksize_y * by
    if pos_x <= old_matrix.shape[0] and pos_y <= old_matrix.shape[
        c = abs(new_matrix[pos_x,pos_y] - old_matrix[pos_x,pos_y])
        if c <= 0:
            result_matrix[pos_x,pos_y] = True
        else:
            result_matrix[pos_x,pos_y] = False
```

Figure2 Diagram of optimization strategy. Three main steps, expansion, inflation and check convergence were written as separate kernel functions. For expansion and convergence check, each thread is responsible for each element in the matrix. For inflation, each thread calculates each column of the matrix.

The expected performance improvement ought to be observed in these three computation steps (expansion, inflation and check convergence), however it is also expected that the frequent host-to-device data transfer and memory relocation will impair the overall performance improvement. There are several technical challenges in implementing parallelized MCL in Python numba packages. First, numba doesn't support dynamic parallelism yet such that all the kernel functions have to be invoked in the host, which necessitates the host-to-device data transfer. Second, numba doesn't support texture memory yet, which could potentially allow me to further optimize the inflation step in a way to map each block to every column. As a result, all the threads residing in the block can cooperate to compute the sum and perform normalization. Third, while numba made fantastic progress to make CUDA-compatible, the supported data type and supported built-in python function is still very limited, for instance, string type is not supported by numba and only scalar function is able to use in kernel functions.

Application performance Analysis and project results

I implemented and tested the performance of sequential CPU version and parallel GPU version on Ohio Supercomputer Owen cluster. A Jupyter notebook application was launched and CUDA10.2.89 along with Python3.6 were utilized for implementation. The ipynb notebook and code base is freely available at (https://github.com/frankligy/numbaMCL). I used jupyter built-in magic function %%timeit which automatically ran the code block subjected to test multiple times and returned the mean running time along with the standard deviations. As mentioned at the beginning, the sequential version is faster than GPU version during the testing phase that CPU version consumed 172ms (std=3.42) to perform 4 iterations while GPU version consumed 244ms (std=2.36), as shown in Figure3.

The possible reason for the inferiority of GPU version can be attributed to the frequent host-to-device data transfer. In order to overcome this bottleneck, I think the path to go along is to avoid host device communication by implementing these three steps all in GPU once and done. In 2010, Bustamam et al. used CUDA C library to parallelize the MCL algorithm and observe performance improvement in three large datasets[2]. In 2018, Azad et al. from Lawrence national laboratory achieved a high-performance parallel version of MCL based on MPI and OpenMP[3]. Also, the inferior performance can also due to the characteristics of the testing input data which is only 1024 x 1024 in dimension. The numba GPU version might still become superior when the size of input matrix keeps increasing and eventually outperforms its CPU counterpart.

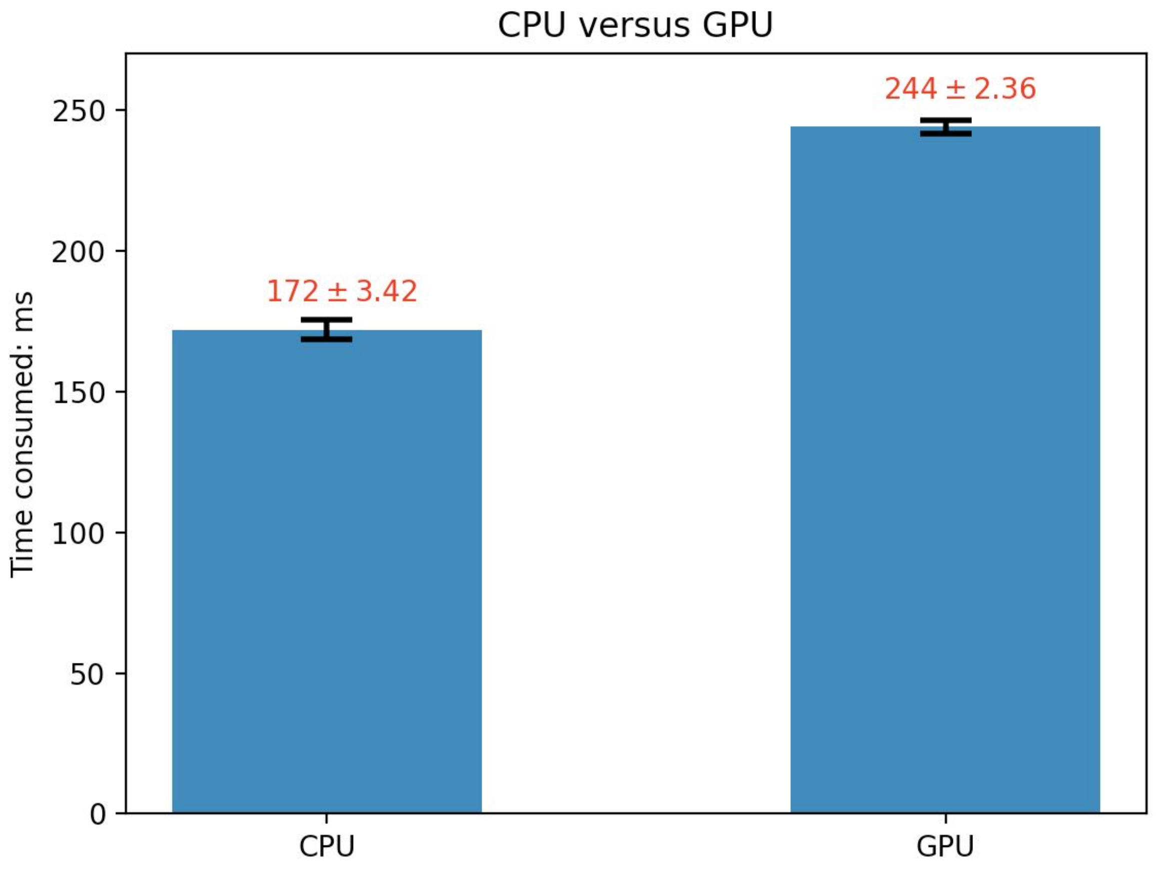


Figure3 Comparison of performance between sequential CPU version and parallel GPU version. The average time spent along with standard deviation was shown as right on top of each bar chart.

Self-assessment

This project was conducted by myself, I am a second-year PhD student in College of Medicine and I am a python programmer for almost all my daily tasks. I tried to implement the MCL algorithm in CUDA C but my lack of C-centric experience hinders me from successfully implementing the algorithm. The Python numba package provides most of CUDA functionality but there are still several voids (i.e. dynamic parallelism, texture memory). More importantly, the limited support for python primitive data type (no array function, no string support) are a bigger issue when I was researching and trying to implement the algorithm. The limitation of my final project is: (1) the sub-optimal data transfer strategies impair the GPU version's overall performance. (2) the lack of suitable testing dataset which might provide a more comprehensive comparison between GPU and CPU version.

Despite the drawbacks, there are still merits I can see from my endeavors. First, this is the first numba implementation of MCL algorithm from what I am aware of. The Numba package suffers from limited examples and tutorials so this implementation could serve as a real example for people who are interested in using Numba to parallelize their existing program. Also, I discussed the origin of the GPU's inferior performance and explained the current limitation of CUDA python which can serve as a rubric for people who have to decide which programming language to use. The codes and jupyter notebook are freely available at (https://github.com/frankligy/numbaMCL).

Reference

1. DePasquale EAK, Schnell DJ, Van Camp P-J, Valiente-Alandí Í, Blaxall BC, Grimes HL, et al. DoubletDecon: Deconvoluting Doublets from Single-Cell RNA-Sequencing Data. Cell Rep. 2019;29: 1718–1727.e8.

2. Bustamam A, Burrage K, Hamilton NA. A GPU Implementation of Fast Parallel Markov Clustering in Bioinformatics Using EllPACK-R Sparse Data Format. 2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies. 2010. doi:10.1109/act.2010.10

3. Azad A, Pavlopoulos GA, Ouzounis CA, Kyrpides NC, Buluç A. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. Nucleic Acids Res. 2018;46: e33.