Efficient Planning of an Optimal Public Transport Grid Using CUDA

Omar Al Jaroudi
oaa20@mail.aub.edu

Mohamad Abou harb
maa299@mail.aub.edu

Department of Electrical and Computer Engineering

American University of Beirut

***Abstract* – Public transport is an effective way to decrease traffic congestion and its negative consequences. However, people should be presented with an efficient and convenient system to incentivize them to give up using privately owned vehicles. In this paper we present a way to plan an efficient public transport system, namely buses, in the city of Beirut, Lebanon. This is accomplished using k-means clustering to locate the bus stops, and Floyd-Warshall algorithm to calculate the all-pair shortest paths between them, both of which were implemented in parallel, using CUDA. By applying our program to Beirut, we achieved reasonable execution times, whereas the serial version of the code did not terminate. The next step which we contemplated is to use the Ant Colonization Optimization algorithm (ACO) to simulate optimal trips for the buses on this newly designed grid.**

***Keywords* – Public transport, CUDA, k-means, Floyd-Warshall, ACO**

I. INTRODUCTION

Lebanon has been notably suffering from the constantly growing number of cars, whose negative effects is aggravated by the lack of long-term official policies to address the problem. Lebanon traffic congestion has reportedly cost Lebanon around $242 million in 2017, which was 0.5% of the country's GDP [1]. This is calculated as a combination of the fuel costs paid for cars, the value of time lost in traffic jams, and the cost of alleviating the environmental damages done due to the excess carbon emissions of cars.

Such problems have become characteristic of modern urban areas. In this case, Beirut, the capital and urban center of Lebanon, has been hit harder than most, with more than 300 thousand cars entering Beirut daily back in 2016, which is more than 16% of the total cars officially registered in Lebanon [2,3].

Therefore, Lebanon, and Beirut in specific, can greatly benefit from an efficient public transportation system, to decrease the number of privately-owned cars, thereby decreasing traffic jams and consequent costs. To that end, we present a program that can generate an optimal grid for public bus transportation, and successfully simulated this grid for the city of Beirut.

Our implementation is in the spirit of [4], which accomplishes this task for the city of West Jakarta, albeit with notable differences. Nevertheless, the approach discussed in this paper can be summarized as:

1) Preprocess map data of Beirut including roads, intersections, and buildings.
2) Cluster this data to find optimal locations for bus stops.
3) Calculate shortest distance between chosen stops.
4) Simulate optimal trips for give buses along the designed grid.

In this paper, we will explain our implementation of steps 1-3, and comment on step 4 as the possible next step of our work. Our code was written in Python with PyCuda and is available on GitHub[1]. The device used to run CUDA was a GeForce GTX 1050.

II. PREPROCESSING DATA

To retrieve data about Beirut road and building infrastructure we relied on OpenStreetMap[2](OSM), which is an open-source effort that provides free access to a world map. However, instead of manually retrieving separate pieces of data, we used OSMnx[3], which is a Python library that automates API calls to OSM. This library is also built on top of several intermediate others to facilitate visualizing and processing graph data.

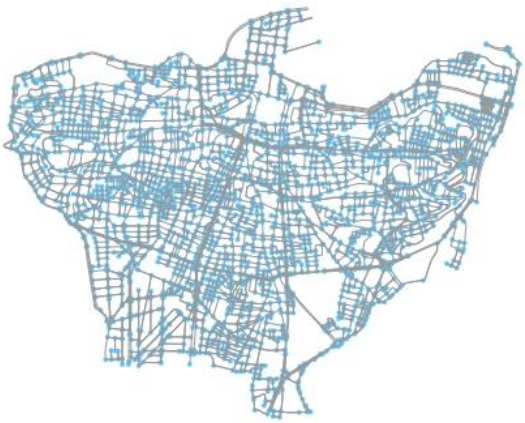
To begin with, our primary intuition was that a successful public transport grid is characterized by being accessible to a maximal portion of the population. To be able to work for this objective, we needed data about the population distribution in Beirut (e.g. dot density maps). To that end, we followed the approach of [4], wherein we considered the buildings as a good representative for population. We therefore generated a map of all buildings in Beirut, which OSM reports as over 13

[1] https://github.com/TwoShock/Optimal-Bus-Route-for-Beirut

[2] https://www.openstreetmap.org

[3] https://osmnx.readthedocs.io/en/stable/#

thousand. Each building was represented as a set of nodes constituting a polygon. In our case, we were more interested in a single point representation, so we reduced each building to the centroid of its constituting nodes.



(a)

(b)

Fig.1 Map of Beirut's (a) road network (b) buildings, retrieved with OSMnx

We also generated a map of the roads and their intersections for Beirut. OSMnx conveniently offered ample data about this map, such as the direction of traffic for a street, its length, as well as geographic coordinates for street intersections (i.e. latitude and longitude). The two maps discussed above can be seen in Fig. 1.

Finally, we stored these maps as byte-stream data on our local machine to avoid issuing unnecessary API calls while debugging, which would only slow down the work.

III. LOCATING BUS STOPS

A. Methodology

The first actual step in our design is to find the optimal location of bus stops in Beirut. We first decided on the number of stops. A recurring concept in the literature was to position a bus stop within a given distance from a passenger in any walking direction [5]. While this distance varied anywhere between 400 and 1000 meters, we chose it to be 500 meters for our testing purposes. Considering each bus stop as a center of a circle with radius 500m and noting the area of Beirut as around 19.8 km^2, a total of 26 bus stops were needed to cover the whole city.

To find the best position of these stops, we used k-means clustering, with the buildings' centroids discussed in Section 2 serving as the nodes being clustered. However, instead of using a serial implementation of the algorithm, we designed a parallel version of it using CUDA.

Given a collection of nodes, k-means clustering works by designating "k" clusters, each having a centroid. Then, to assign a node to a cluster, it should have a minimum distance to the cluster's centroid. After assigning all nodes to clusters, the centroids of these clusters can be calculated again.

This iterative process eventually converges when the new centroids and cluster assignments stop changing with respect to a certain threshold [6]. In this case, k is chosen as 26. An auxiliary array was used to assign a cluster ID to each node. Naturally, the ID was an integer from 1 to 26.

The design includes 3 kernel functions that are invoked in each iteration. The first of these kernels is responsible for clustering, and each thread handles a single building. That is each thread calculates the distance of its assigned building to the 26 existing centroids, and finally adds the building to the closest cluster. In our case, we used Euclidean distance, noting that each building node had latitude and longitude coordinates. This kernel was invoked with the following configuration:

- dimBlock = (32,1,1)
- dimGrid = (ceil(N/32), 1) where N is the total number of nodes

The second kernel is responsible for re-calculating the centroids of the clusters. Shared memory was incorporated in this kernel. The coordinates of *dimBlock.x* nodes, as well as their cluster ID, was loaded into shared memory. Then, the coordinates of each cluster's nodes are added and stored in local memory. Finally, these local sums are written to global memory using atomic operations. This kernel was launched with the same configuration as above.

As for the third kernel, it simply divides the sums calculated above by the number of nodes in each cluster. That is each thread handles 1 cluster by dividing the sum that was calculated in kernel #2 by the number of nodes that were in that cluster. It is noted here that an auxiliary array of length 26 was used to maintain the size of each cluster. This kernel was invoked with the following configuration:

- dimBlock = (32,1,1)
- dimGrid = (ceil(K/32),1) where K is the number of clusters

With this division, the coordinates of the corrected centroid for each cluster is now obtained, marking the end of a single iteration of the whole algorithm.

B. Results

The design discussed above was tested for accuracy and execution time versus its serial counterpart. For that purpose, the version of k-means implemented in scikit-learn[4] was used as the benchmark. It is worth noting here that one unexpected disadvantage encountered with the parallel implementation is the fact that it didn't support large data type containers; a 32-bit representation was used, so any number that didn't fit in 32 bits was truncated before being passed to device memory. As the size of data and the cluster count grows, this inaccuracy becomes more pronounced. However, due to the scale of the data in this study, it didn't really affect our final result.

TABLE 1. SPEEDUP OF PARALLEL K-MEANS IMPLEMENTATION FOR DIFFERENT K & N

K \ N	10^3	10^4	10^5	10^6
10	0.79	4.38	11.62	16.9
100	6.18	18.95	59.59	N/A
500	8.84	24.99	75.31	N/A

In terms of execution time, the two versions were tested with uniformly distributed inputs of different sizes, and their performance was measured in terms of speedup achieved by the parallel version. Referring to Table 1, it can be observed that the parallel algorithm discussed above achieves a speedup greater than 1 in all but one case. Some cells have been marked as *not available* (N/A), which is because the serial version of the code did not terminate. Fig. 2 is a graph showing the speedup achieved by the parallel implementation versus the size of the dataset N (up to 10^5), for different values of cluster size K.

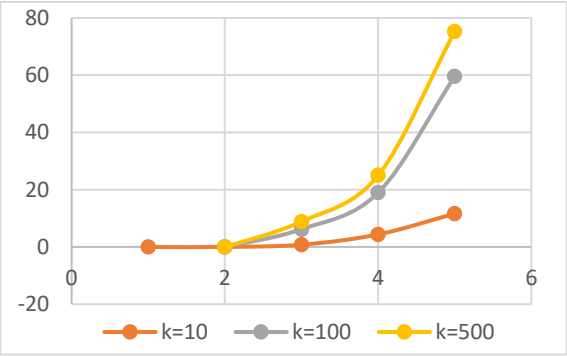


Fig.2 Speedup versus $\log_{10}(N)$ for different values of K

For the case of Beirut, which had 13317 nodes, and 26 clusters, the recorded speedup was 3.11, and the finalized cluster centroids, which will serve as the optimal bus stops, are seen in Fig. 3.

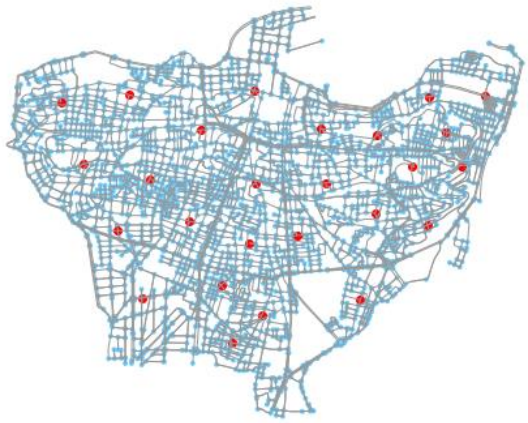


Fig. 3 Recommended bus stops (red)

IV. ALL PAIR SHORTEST PATHS

A. Methodology

After generating the bus stop locations, it is advantageous to have the shortest paths between them. This is modeled as a graph problem, with the bus stops being the nodes denoted by set V, and the streets connecting them are the edges, denoted as set E. This graph is directed since every edge has a direction, and the weight of this edge is the length of the street.

To generate the shortest paths, we used Floyd-Warshall algorithm, which takes as input the graph as an adjacency matrix and returns the optimal path to take for each node i to get to node j.

This algorithm works by choosing each vertex one by one to be an intermediate vertex and updating the adjacency matrix. The parallel version uses shared memory to store current bests per block and each thread updates the adjacency matrix and path vector of the current element it is working on given the cached best data. The kernel was launched with the following configurations:

- dimBlock = (32,1,1)
- dimGrid = ceil(N/32,1) where N is the dimension of the square matrix

B. Results

Like before, we tested our parallel implementation of this algorithm against a serial benchmark, with uniformly distributed inputs of different sizes. The serial version of this algorithm has a run-time of $O(|V|^3)$ and as such did not scale well for input sizes above 1 thousand.

As we can see, from the below graph (Fig. 4) the serial version of this algorithm grows very rapidly in time, whereas the parallel version manages to take orders of magnitude less time. This is because we can try out multiple nodes as intermediary nodes whereas in the serial case, we are limited to one node at a time.

[4] https://scikit-learn.org/stable/

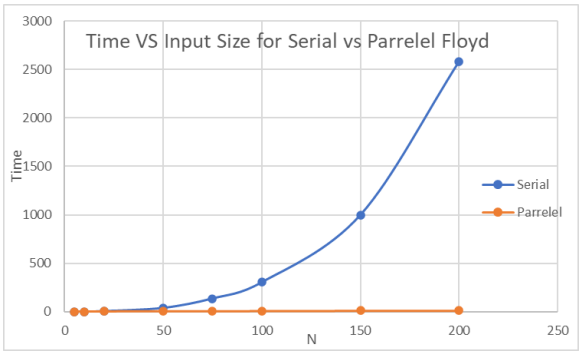


Fig. 4 Serial vs Parallel execution time of Floyd-Warshall

The below graph (Fig. 5) illustrates the relative speed ups attained for dense *NxN* dense graphs where N ranges from 5 to 200. As for the case of Beirut, we ran this algorithm on the street network map discussed earlier, which had 3714 street intersections representing nodes. The serial version of the code did not terminate, whereas the parallel one ran in 11.5 milliseconds.

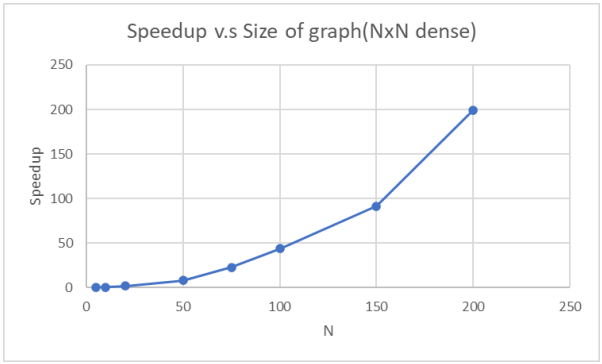


Fig. 5 Speedup of parallel Floyd-Warshall vs N

V. OPTIMAL BUS ROUTES

Now that optimal bus stops are located, and the shortest path between any pair of them is available, the final step would be to simulate bus trips along this grid. This step can be approached in several ways. To begin with, we considered the option of having the bus stops as a single consecutive chain, and that each bus should traverse all stops in a single trip. This model, if reduced to one bus, is known as the Traveling Salesman Problem (TSP). This NP-hard problem has been regularly addressed in literature, with contemporary findings reporting solutions that are within 3% range of the optimal tour for more than 10^6 nodes [7,8,9]. After laying out a solution for the TSP problem for our grid, we would then have to plan a timetable for the buses not to overlap at certain stops.

To address the TSP problem for our application, we considered the Ant Colony Optimization (ACO) algorithm. A probabilistic algorithm based on the actual patterns adopted by ants when gathering food, ACO essentially seeks to optimize paths in a graph by having the ants communicate with each other. In the beginning, several ants locate a source of food and carry it back to the colony, each taking a separate route. As the ant moves, it deposits a chemical called pheromone to form a trail. If an ant has found a shorter path than others, the amount of pheromone along its trail will naturally be greater. Hence, other ants can now follow this trail.

The case of public transport is almost analogous to the scenario depicted above, except in this case, ants would be trying to find an optimal tour over the nodes. Iteratively, ants set out to find the shortest tour of the graph that covers all nodes. The ant that travels the least distance is the optimal solution, which means that it deposits the most pheromone. After completing the specified number of iterations, the best tour found is returned. This tour is not guaranteed to be globally optimal, since the algorithm is inherently an iterative process that is expected simply to converge.

Pants [5] is a useful Python library that implements this algorithm for generic graph structures. We ran the algorithm provided by this library on our grid, with 100 iterations, having 10 ants in each iteration. There are other parameters that can be configured, such as the relative weight of the deposited pheromone, but these are explained in detail in the library's documentation. The result of this simulation is seen in Fig. 6, which represents the routs taken by the optimal ant to traverse the graph. The algorithm also returns the order in which the nodes should be traversed.

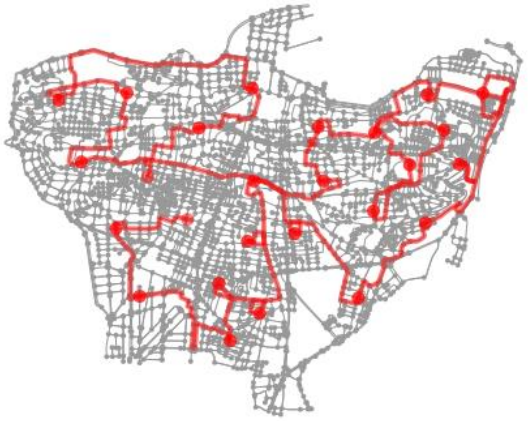


Fig. 6 Result of ACO algorithm for chosen bus stops.

We have also explored efforts that attempt to parallelize this algorithm using CUDA. Our impression was that, even at a very high level, parallelization is pronounced in this algorithm. For example, in each iteration, the ants can travel in parallel rather than one at a time. A possible implementation of this is to create as many copies of the grid as there are ants, and have each one operate on an independent grid, meanwhile a central controller structure monitors all the ants by comparing their pheromone deposits and sorting them in that order. Uchida et al. [10] more formally implement a similar approach. Each iteration of the

[5] https://pypi.org/project/ACO-Pants/

algorithm involves invoking 3 separate kernels. The first kernel initializes the grid, by randomly placing the ants at nodes in the graph. The second kernel constructs a tour for each ant. Finally, the third kernel updates the pheromone value for each ant, which allows for deciding on the best tour so far. This implementation has reported a speedup of around 43 over its serial counterpart [10].

Implementation-wise, we tried to integrate this approach for the example of our grid of Beirut. We designed 3 kernels similar to what is explained above, but accounting for pheromone evaporation as well, because if a path is no longer used, the pheromone deposited previously on it should decay. Unfortunately, we were not able to reproduce the same results seen in Fig. 6.

VI. REALISTIC LIMITATIONS

It should be noted at this point that as promising as ACO may seem, there are several real-world limitations that need to be considered before planning bus routes in general. To begin with, assuming the bus stops as a single tour can be argued as unrealistic. Practically, there are certain bus stops that need to be directly connected by a bus route, even if this ruins the optimal tour. This comes down to the ridership patterns. For example, there are bus stops that fall in vital and heavily trafficked locations that need to be traversed more than others. In Beirut for example, a bus route directly connecting the southern entrance of the city to its northern entrance would be in high demand, due to the patterns of people entering and leaving Beirut on a daily basis. In this case, it would be undesirable to force passengers interested in this route to pass by unnecessary stops.

Other limitations that we have realized in this project is the lack of data about traffic patterns. Measuring the shortest path between nodes in terms of distance implies that there are no other cars on the street, when this is rarely ever the case. A more realistic approach would be to assign the expected time consumed traversing a street as the street's weight when applying Floyd-Warshall. That is, we are really interested in the shortest path in terms of time rather than distance. Nevertheless, this problem is not one related to our methodology. If data regarding traffic congestion in Beirut is available, it can be easily incorporated as a feature in our design.

VII. CONCLUSION

In this paper we explained our design that aims to plan an optimal bus grid for the city of Beirut with CUDA. By implementing a parallel version k-means clustering that achieved a speedup of 3.11 over its serial counterpart, we were able to locate optimal positions for bus stops to cover the whole city. We also calculated the all-pair shortest path between these stops in reasonable runtime, by implementing Floyd-Warshall algorithm in parallel. The serial version of the latter did not terminate.

The final step to conclude our work would be to plan efficient bus routes for this grid. We have modeled this as the Travelling Salesman Problem and used the Ant Colony Optimization algorithm to solve it. Using a ready-made serial implementation of ACO, we were able to generate an optimized tour to cover all the bus stops. Nevertheless, we have illustrated limitations that might undermine this approach in real life.

With more data, including ridership patterns, as well as information about traffic congestion, more resilient and optimal bus routes can be simulated. In fact, with the significant speedups reported in this paper by our parallel algorithms, it is even possible to use this design in real-time, by feeding it the most up-to-date data about traffic congestion for instance, allowing for more versatile decisions about paths between bus stops.

REFERENCES

[1] A. Saroufim and E. Otayek, "Analysis and Interpret Road Traffic Congestion Costs in Lebanon," *MATEC Web of Conferences*, vol. 295, Jan. 2019.

[2] N. El Khoury, L. K. Aboujaoude, and K. Kadehjian, "Road Transport Sector and Air Pollution Case of Lebanon 2016," *IPT Energy Center*, 2016.

[3] S. Kadi. "Traffic Congestion Adds to Lebanon's Many Woes."TheArabicWeekly.com https://thearabweekly.com/traffic-congestion-adds-lebanons-many-woes. (accessed May. 3, 2020).

[4] K. Supangat, Y. E. Soelistio, "Bus Stops Location and Bus Route Planning Using Mean Shift Clustering and Ant Colony in West Jakarta," *The International Conference on Information Technology and Digital Applications (ICITDA)*, p. 14-16, Nov. 2016.

[5] J. Walker. "How Far Will People Walk for Public Transport- and How Close Should Stops Be." Citymetric.com. https://www.citymetric.com/transport/how-far-will-people-walk-public-transport-and-how-close-should-stops-be-1195. (accessed May. 4, 2020).

[6] S. P. Lloyd, "Least Squares Quantization in PCM," *IEE Transactions on Information Theory*, vol. it-28, no. 2, Mar. 1982.

[7] R.M. Karp, "Reducibility Among Combinatorial Problems," *University of California at Berkley*, p. 86-103, Jan. 1972.

[8] C. Rego, D. Gamboa, F. Glover, and C. Osterman, "Traveling salesman problem heuristics: Leading methods, implementations and latest advances," *European Journal of Operational Research*, vol. 211, p. 427-441, Sep. 2010.

[9] E. Klarreich. "Computer Scientists Find New Shortcuts for Infamous Traveling Salesman Problem." Wired.com https://www.wired.com/2013/01/traveling-salesman-problem/ (accessed May. 4, 2020).

[10] A. Uchida, Y. Ito, and K. Nakano, "An Efficient GPU Implementation of Ant Colony Optimization for the Traveling Salesman Problem," *Third International Conference on Networking and Computing*, p. 94-102, 2012