Vehicle Routing Problem Algorithm implementation on GPU with CUDA for Acceleration

Prasad Pandit, Student, Portland State University (prasad@pdx.edu)

Radhika Mandlekar, Student, Portland State University (radhika@pdx.edu)

***Abstract*— We present the implementation of Vehicle Routing Problem (VRP) Algorithm with single depot on GPU by exploiting the parallelism which optimizes the number of computations and matrix calculation required. The proposed solution for classical VRP optimizes the costs of travel during deliveries from single depot by generating set of optimized routes with Clark and Wright Heuristics. A single CPU solution is first developed in Python and then restructured for GPU using CUDA for improving performance. We also compare the single CPU vs. GPU solution for presenting the significant improvement in computational performance for faster solution delivery.**

***Index Terms*— VRP, CUDA, Parallel Algorithm, Python**

I. Introduction

THE Vehicle Routing Problem is a combinatorial optimization which computes the optimal set of routes for fleet of vehicles. For the travel companies, though planning a travel route of delivery vehicle is not that difficult, still the profit hugely depends on the travel cost incurred during deliveries. Another thing is that, small set of cities or places may not be a difficult problem but then the number of places or customers increase to level of 100 then a wrongly planned route may consume huge chunk of profit. Manually planning the route also gets cumbersome. This is where VRP comes into play. Even considering VRP for 100s of customers, to find the best route is a difficult and time consuming task as the number of solutions become 100! (100 factorial). To reduce all this burden of computation heuristics play a really important role in solving VRP by providing one of the best optimized solution in least number of computations. There are many different heuristics available viz. Tabu method, Genetic Algorithm, Ants Colony Algorithm etc. [4] We will be using Clark and Wright saving's algorithm. [3]

Given the set of parameters VRP can be further configured and utilized in many ways. Consider a tuck contacting hazardous goods if it passes through a populated area in a city which may pose a danger if situation gets worse for the truck. To solve this problem we can use VRP with a dataset having population data around the nodes and then by modifying the existing algorithm we will be able to provide solution for such situation with optimal route keeping the populated area safe.

In this paper we study about the basic VRP with vehicle having fixed capacity C which then fulfil the demands of customer with quantity q as required. To optimize the travel cost, VRP helps in finding set of routes by calculating the distances between nodes (cities).

The algorithm first starts with finding distance d_{ij} between nodes if for all the nodes considering depot to be at co-ordinate (0, 0). The nodes are then numbered as given in dataset. Then we sort the savings and later run heuristic on sorted data for optimal solution.

As we can see there are certain amount of steps required to get the solution. These steps consist of processing 2-dimensional matrix data, sorting and heuristics. This shows need of computational power as the number of nodes increase.

GPUs (Graphics Processing Units) are nowadays also used for general purpose computing. The parallel architecture of GPU helps in running individual threads running in parallel to get the task done in earliest possible way than CPU. While the companied manufacturing them have now created libraries to program them it has been a great benefit for developing algorithm which can be performed in a parallel manner. We are using NVidia GPU for which NVidia provides a CUDA (Compute Unified Device Architecture) for GPU programming to use them as GPGPU (General Purpose GPU).

Considering the number of computations we chose to implement the VRP on GPU using CUDA. This will help in reducing solution generation time and will give results as quick as possible.

In section II we have explained the heuristics that we have selected and steps it performs. Section III describes initial solution using single CPU. Section IV explains the GPU architecture and how we used it. Section V explains the optimizations we introduced to improve performance using GPU. Section VI we did performance comparison with multiple datasets. Section VII describes the future work that can be done for further improvements. Lastly, section VIII describes our conclusion and ends with section IX i.e. references.

II. Algorithm Section

We studied Holmes and Parker algorithm as well as Clarke and Wright algorithms to solve single depot vehicle routing problem. We designed a serial program using python to compare the performance of both and we found out that Holmes and Parker approach requires intense computations and the resultant routes need to be reordered to optimize the cost of travel. So we decided to go with Clarke and Write Algorithm.

Imagine a goods delivery truck starting from depot that must visit n -1 cities exactly once and return to depot.

Consider the capacity of a goods delivery truck to be c. The cost of traveling from city i to city j is given as c_{ij} for all pairs of cities. The problem is to design a route or tour of minimum cost that visits each of the n cities exactly once.

If the cost to travel from city i to city j equals the cost to travel from city j to city i, ($c_{ij}=c_{ji}$) for all cities, then the problem is symmetric. If $c_{ij} \neq c_{ji}$ for some pair of cities, then the problem is asymmetric. We are considering the symmetric problem for our case. We assume that the direct routes with shortest distance exists between the cities.

Step1:
We expect x and y co-ordinates of cities as input to the program. Here, we refer cities as nodes. We calculate the distance between all the cities using distance formula and form distance/cost matrix.

Step2:
We then use this cost matrix to calculate savings Matrix. The saving between two cities i and j is calculated as Sij = C0i + C0j-Cij where Cij denotes cost or distance between cities I and j and zero indicates depot.

Step3:
We sort these savings in descending order as we would like to club the cities with highest savings together and thereby save maximum cost.

Step4:
Starting with largest amount of savings, we repeat the following Process to get the optimized routes for travel.
Join city to existing route if

1. Cities belong to separate cycle
2. Maximum capacity of vehicle is not exceeded
3. If position node in result is first or last

Step5:
The above step gives us all the optimized routes for given cities with given demands and considering the truck capacity. We now have to calculate the cost of each route which can be calculate by adding the cost for the respective city in route referenced from depot.

III. Initial Solution

We developed initial solution to test the correctness of algorithm using Python. This python script accepts a cities, demands, co-ordinates and vehicle capacity as argument with the help of input file.

The efficient use of tuples, dictionary and array make it easy to process the data. We followed the same algorithm described earlier and used time library to calculate the total execution time of the program to solve the given problem. We also plotted the optimized routes in graph using script for quick understanding.

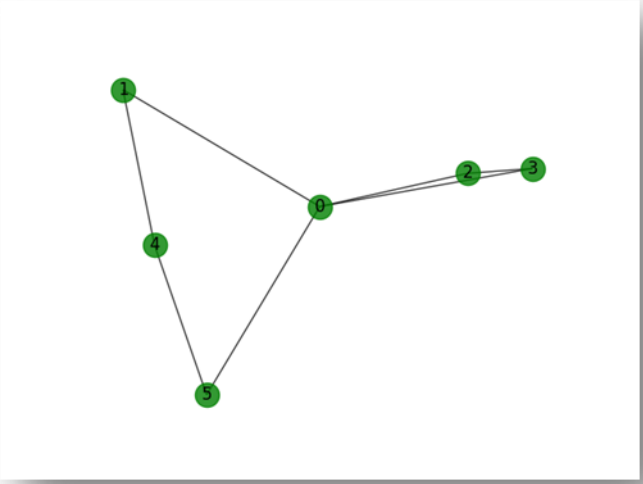

Figure 1: Optimized routes for 5 nodes

Sample of input data set for 5 nodes:

```
NAME : A-n5-k5
TYPE : CVRP
DIMENSION : 5
EDGE_WEIGHT_TYPE : EUC_2D
CAPACITY : 100
NODE_COORD_SECTION
1 1 1
2 2 2
3 3 3
4 -1 1
5 -1 -2
DEMAND_SECTION
1 50
2 50
3 50
4 25
5 25
```

Text output for dataset of 5 nodes computed on CPU:

```
Name of Dataset: A-n5-k5
Time for compute: 45 us
Route 0: 2, 3
```

Savings: 6
Route 1: 1, 4, 5
Savings: 4
Total Nodes Processed: 5
Total Savings: 10

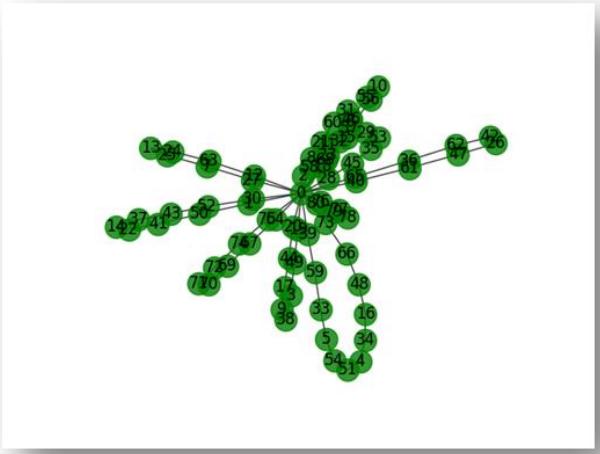

Figure 2: Optimized routes for 80 nodes

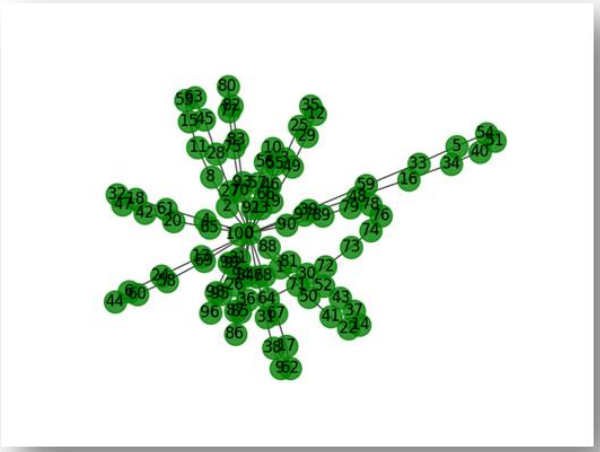

Figure 3: Optimized routes for 100 nodes

We observed that this serial programs needs lot of iterations even to do computations for data set of 10x10. So we could optimize this solution using GPU.

IV. GPU Architecture

For this paper we are using NVidia GTX 960M GPU which is based on NVidia Maxwell Architecture. Figure 4 gives glimpse of the block level architecture. This GPU have following features:

1. Streaming Multi-Processor : 5
2. No. of CUDA cores : 640 cores (128/SMP)
3. Warp Size : 32
4. Threads per block : (1024, 1024, 64)

In GPU and CUDA, the kernel is a program which runs on GPU. To execute kernel, programmer needs to allocate the memory in GPU and copy host program memory to GPU device memory. In Figure 4 we can see that there are small blocks by name core. These are CUDA cores which can run threads in parallel. To make use of whole GPU and all CUDA cores the GPU have a warp scheduler which schedules 32 threads at a time. From programmer's point of view, GPU is broken down in hierarchy of

- Grids
  - Blocks
    - Threads

While programming GPU programmer needs to tell compiler regarding the number of threads per block and number of blocks per grid. This later helps in indexing the threads for operation and accessing memory. Each thread receives a chunk of data to process. It can interact with other threads within the block. The programmer also needs to take care of data dependencies by synchronizing threads.

GPU has multilevel memory architecture:

- Local Memory : Local to thread
- Shared Memory : Shared between threads
- Global Memory : Shared between blocks

While developing a program, programmer can define set of shared variables in kernel, set of local variables per thread. This helps in better memory optimization which later improves performance. For vector calculation GPU supports up to 3-Dimentional array.

We studied above methods for performing 2-D array computations and memory management as required.

Figure 4: Streaming Multi-processor and CUDA Cores

V. Optimizations for GPU

In the serial algorithm we developed, we found that the maximum number of CPU time is used for

- Cost matrix generation
- Savings matrix
- Sorting

We focused on these 3 sections of our program and implemented a parallel algorithm for them.

1. For cost matrix i.e. distance calculation we used a 2-D array computation
2. For Savings matrix we used an optimized 2-D array computation as the generated output of Cost Matrix was a transpose
3. For Sorting, we used odd-even sort algorithm which is basic sorting algorithm for GPUs

For efficient programming and memory use we created C structures which reduced lot of our efforts. The direct node indexing to memory location helped us optimize our program for memory allocation and pointer access.

The prime parameters in Cost matrix generation were proper thread indexing and number of blocks per grid. We analyzed our data structure and used following equation to get number of blocks we need for computation with total number of threads in 2D array being (32, 32).

```
dim3 threadsPerBlock(MaxThreadx, MaxThready);

dim3 blocksPerGrid((((columns + MaxThreadsPerBlock - 1)/MaxThreadsPerBlock) + 1), (((rows + MaxThreadsPerBlock - 1) / MaxThreadsPerBlock) + 1));
```

Here *dim3* is a 3-variable datatype for X, Y and Z indexing. We used same configuration for computing Savings matrix with optimizations in algorithm to compute the matrix for all data in a row keeping the column co-ordinate greater than row co-ordinate. This helped us skip the lower triangle computation of matrix optimizing the time spent in computation.

```
__global__ void
savingsCalc(int* costMatrix, int* savingsMatrix, int rows, int columns)
{
   int x = threadIdx.x + blockIdx.x * blockDim.x;
   int y = threadIdx.y + blockIdx.y * blockDim.y;
if (y < rows && x < columns) {
      int valOfx = costMatrix[x];
      int valOfy = costMatrix[y];
      int valOfxy = costMatrix[x + y * rows];
      *(savingsMatrix + x + y * rows) = (x != y && x > y) ? valOfx + valOfy - valOfxy : 0;

      }
}
```

For odd-even sorting we used a 1D array with maximum number of threads supported per block i.e. 1024. Depending on even or odd number of nodes we performed a swap logic.

```
__global__ void
sortSavings(struct savings_info* records, int count) {
   int i = threadIdx.x + blockIdx.x * blockDim.x;
   int l;
   if (count % 2 == 0)
      l = count / 2;
   else
      l = (count / 2) + 1;

   for (int j = 0; j<l; j++)
   {
      if ((!(i & 1)) && (i<(count - 1)))  //even phase
      {
         if (records[i].savingsBetweenNodes < records[i + 1].savingsBetweenNodes)
         {
            struct savings_info temp = records[i];
            records[i] = records[i + 1];
            records[i + 1] = temp;
         }
      }
      __syncthreads();

      if ((i & 1) && (i<(count - 1)))     //odd phase
      {
         if (records[i].savingsBetweenNodes < records[i + 1].savingsBetweenNodes)
         {
            struct savings_info temp = records[i];
            records[i] = records[i + 1];
            records[i + 1] = temp;
         }
      }
      __syncthreads();

   }//for
}
```

VI. Performance Comparison

After running the program with 15 datasets on both Single CPU and GPU we were able to see significant improvement in performance as the number of nodes increase. For lower number of nodes i.e. 5 we can see CPU is faster (Fig. 5). The reason for this slowdown is memory transfer overhead from CPU to GPU and back. Later 16 nodes we can see linear increase in performance going to 60X for 130 nodes.

Table shows comparison of execution time (in microseconds) for CPU and GPU for given number of nodes and shows the calculated acceleration (speedup).

Number of Nodes	Time to compute on CPU in us	Time to compute on GPU in us	Acceleration
5 Nodes	45	369	0.1219512
16 Nodes	390	375	1.04
32 Nodes	1719	357	4.8151261
33 Nodes	1856	379	4.8970976
37 Nodes	2105	368	5.7201087
39 Nodes	2279	368	6.1929348
45 Nodes	3890	392	9.9234694
53 Nodes	3929	389	10.100257
55 Nodes	5690	371	15.336927
64 Nodes	6085	386	15.764249
65 Nodes	5926	390	15.194872
69 Nodes	6774	384	17.640625
80 Nodes	9677	396	24.436869
100 Nodes	14535	416	34.939904
130 Nodes	26676	447	59.677852

Figure 5: Performance Numbers CPU vs. GPU

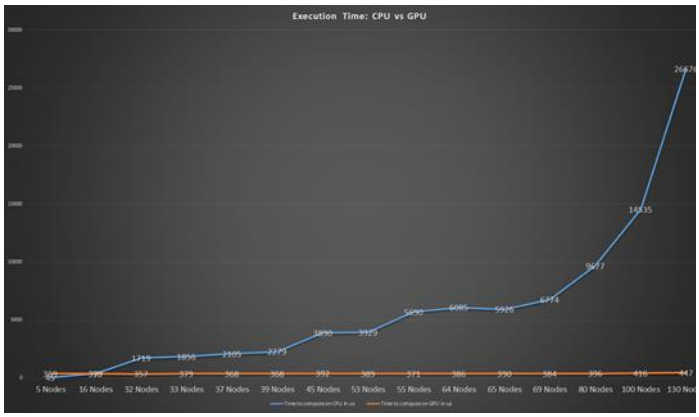

Figure 6: The graph shows the comparison of execution time of CPU versus GPU. The X-axis indicates the number of nodes and Y-axis indicates time in microseconds. The plotted blue line shows the execution time on GPU and the orange line shows the execution time

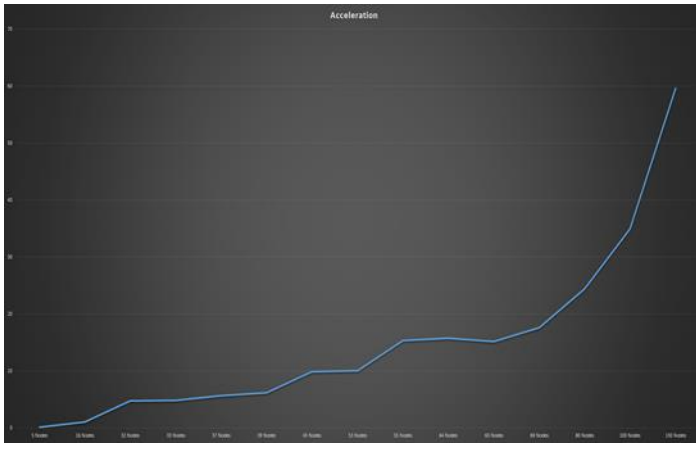

Figure 7: The graph shows effective acceleration or speedup for GPU over CPU for each dataset

VII. Future Work

1. Algorithm can be developed for different parameters and constraints delivery time, traffic situations, cost of fuel, weather etc.
2. Map can be integrated for more realistic results
3. Heuristics can be developed specifically for GPU based VRP
4. Other than Odd-Even sort, new parallel sorting algorithms like Radix sort and Thrust sort can be used to further boost performance

VIII. Conclusion

In this paper, we presented the solution for single deport vehicle routing problem using Clarke and Wright Savings Algorithm Heuristic. We implemented the solution in python to compute serial performance and used NVidia GPU to optimize our approach to this problem using parallel processing. We compared the serial vs. parallel performance for same set of data to compare the performance in terms of speedup. The experiment showed us that even though we increased the number of nodes, the execution time for GPU was increasing linearly. We also observed that serial processing performed better for very less number of nodes and the reason behind GPU performing worse might be the overhead of copying cost and savings matrices from host to device. In addition to this we observed that there was decrease in execution time, if the program run for same dataset consecutively for which the reason might be less cache misses. Thus, we come to the conclusion that using GPU to solve Vehicle Routing Problem for single depot for huge number of nodes or cities would be beneficial and the approach could be extended for Multi-depot VRP with addition of data partitioning to the algorithm described.

IX. References

[1] A. Benaini, A. Berraja, E. Daoudi, "GPU implementation of the multi depot Vehicle Routing Problem," in *Computer Systems and Applications (AICCSA), 2015 IEEE/ACS 12th International Conference, DOI: 10.1109/AICCSA.2015.7507162*

[2] J. Lyssgard, "Clarke & Wright's Savings Algorithm" 1997, http://pure.au.dk/portal-asb-student/files/36025757/Bilag_E_SAVINGSNOTE.pdf

[3] *http://www.mafy.lut.fi/study/DiscreteOpt/CH6.pdf*

[4] G. Laport *"The Vehicle Routing Problem: An Overview of Exact and Approximate Algorithms", European J. Operational Research 59 (1992) 345-358*

[5] K. Madduri and D. Bade, J. Berry and J. Crobak, "*An Experimental Study of a Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances*", http://www.cc.gatech.edu/~bader/papers/ShortestPaths-ALENEX2007.pdf

[6] N. Brown, "*ePython: An implementation of Python for the many-core Epiphany coprocessor*", 2016 6th Workshop on Python for High-Performance and Scientific Computing, DOI 10.1109/PyHPC.2016.8

[7] *https://github.com/prasadp4009/Vehicle-Routing-Problem*