An Intelligent Road Traffic Management System

Tahmid Tanzi Alam 13101060
Ahmad Naquib Chowdhury 13101042
Department of Computer Science and Engineering
BRAC University

This dissertation is submitted for the degree of
Bachelor of Science in Computer Science

2016

We would like to dedicate this thesis to our loving parents ...

Declaration

We hereby declare that this report is our own work and effort and that is has not been submitted anywhere for any award.

All the contents provided here is totally based on our own labor dedicated for the completion of the thesis. Where other sources of information have been used, they have been acknowledged and the sources of information have been provided in there reference section.

Tahmid Tanzi Alam 13101060
Ahmad Naquib Chowdhury 13101042
2016

Acknowledgements

First of all, we would like to express our deepest sense of gratitude to almighty Allah.

We write this acknowledgment with great honor, pride and pleasure to pay our respects to all who enabled us either directly or indirectly in completing this thesis. We would like to show or gratification to our Supervisor Professor Dr. Mohammad Zahidur Rahman for being constant source of inspiration, valuable guidance and constant encouragement to us especially for solving the problems that we have encountered while working on this thesis. And we also like to thank our Co-Supervisor Mr. Moin Mostakim.

Abstract

Road traffic congestion remains a global phenomenon that causes great problems in the cities of the world; especially developing countries, resulting in massive delay, unpredicted travel times, increased fuel consumption, man-hour and monetary loss. The issue has arisen from poorly planned road network and traffic management, resulting in unbearable traffic jams. It is prevalent greatly during weekends, public holidays and periods of major activities. Major causes of the congestion include lane indiscipline, high traffic density, low road network carrying capacity, poor traffic management and support infrastructure such as lay-bye, and low response to removal of broken down and crashed vehicles. This paper gives an intelligent solution for tackling the issue of traffic congestion using parallel algorithm with real time data feeding by re-routing.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Mobility is essential in cities because it highly influences their socio-economic activities. It is also known that economic advancement of a country is firmly connected to its transportation system. Smooth transportation is obscured because of traffic congestion. In a report World Bank (1999) stated that traffic congestions contribute 54.5% in hindrance of effective mobility. This is as a consequence of the always expanding urbanization, human activities and the result of over dependency on street transportation that indicates the increment in the number of vehicles, of various classifications on city road. Of interest also is the difficulty of movements on inter-city streets and other real passages because of blocks, for example, car accidents, broken down vehicles or certain area use economic activities situated along these sideways or sheer movement of high number of vehicles the street system limit and in festive seasons and some other significant exercises [2]. Traffic management has been quite poor in many developing countries, despite the growth in transport demand and supply. The resultant traffic congestion has become impediment to our livability. Road traffic congestion, according to Goodwin (1997) can be defined as the impedance vehicles impose on each other, due to the speed-flow relationship, in conditions where the use of a transport system approaches its capacity. For developing countries its big issue. And development of infrastructure in a short period of time is not a solutions in a short run. Traffic management was used to be one of the major part in civil engineering. But now computer science has also come. They are trying to solve this traffic congestion problem in different manner. Our paper also proposes a solution using shortest path routing algorithm, which can help to distribute the traffic by providing shortest route to the destination in minimum time. It will take the real time data from the people and by analyzing the data we will be setting the cost of the path.

1.1 Problem Definitions

Traffic congestion has been one of the fundamental problems faced by modern cities since the wide usage of automobiles. Over the past decade it has increased substantially. Sometimes it takes more than an hour to cover 1 kilometers of road. There are many reasons for traffic congestions like maintenance on road, broken down vehicles, not enough traffic police to control, VIP movement, no strong implementation of traffic rule more over high density of vehicles in the road [2]. First and foremost, there are several factors that contribute to the occurrence of traffic congestion and the quick increment in the quantities of private car possession because of the advancement of the nation and economy is certainly an undeniable one. In addition, the small road capacity is also one of the contributing factors. As the quantity of private vehicle increments incredibly throughout the years, traffic congestion happens when the required street limit is not satisfied.

Basic enhancements of the street foundation can undoubtedly tackled this issue. For instance, more extensive streets, bridges and even underground passages could be worked to trim down the activity. Since jam happens every now and again in the city communities, neighborhood government city can consider passing laws on limiting the quantity of auto claimed in a family. This technique is truth be told, workable and viable. This traffic congestions is effecting the normal day to day life and also socio-economic activity. One of the problem is faced which is for traffic congestion patients are not taken to hospital in proper time.

1.2 Motivation

Road traffic jams continue to remain a major problem in most cities around the world, especially in developing regions resulting in massive delays, increased fuel wastage and monetary losses. Due to the poorly planned road networks, a common outcome in many developing regions is the presence of small critical areas which are common hot-spots for congestion; poor traffic management around these hotspots potentially results in elongated traffic jams. Poor road traffic management is the primary reason for extended periods of traffic congestion throughout the world. As per Texas Transportation Institute's 2011 Mobility report [3], congestion in the US has increased substantially over the last 25 years with massive amounts of losses pertaining to time, fuel and money. Sã o Paulo, Brazil is known to experience the world's worst traffic jams [4], where people are stuck for two to three hours every day in traffic jams. In Bangladesh is also creating both health and economical problems [5]. The issue of traffic congestion has affected both the developing and developed economies to different degrees irrespective of the measures taken to curb the issue. These sort of jams could be ex-

cluded when people can choose a different path for their destination rather than selecting the route with jams. As a result of this, load of vehicles on every road can be reduced and then a reduction of traffic jams could also be developed.

1.3 Chapter Layout

In this paper, we made an approach to solve this problem and divied them into chapters. In chapter 2 we have literature review, after that in chapter 3 we have methodology, in chapter 4 design, implementation and development environment, followed by chapter 5 result and lastly chapter 6 conclusion.

Chapter 2

Literature Review

2.1 Real Time

Real time computing is something hardware and software has to give output in real-time constraint. Real time programs make sure that it give response in specified time constraints known as "deadline" [6]. A real-time system has been described as one which "controls an environment by receiving data, processing them, and returning the results sufficiently quickly to affect the environment at that time." [7].

Real time systems heavily depends on three things. "Time" is the most important thing in real time system. It is the most precious resources to be scheduled carefully so that it can meet the deadline as well as ensure logical correctness. Also message sending and receiving should be done in real time. Secondly comes reliability. The program must to reliable because any error can cause economical disaster or cost human life. Lastly the environment the machine works [8].

Real time programs are categorized in three parts. Hard real time systems are those whose result can be disastrous if it does not meet deadlines. Firm does not cause that much of catastrophe but the corresponding task cease to be useful as soon as the deadline expires such as transaction in database [9]. And lastly soft which does not create much of an impact.

2.1.1 Real time scheduling

A hard real time system should be executed in a manner that all the critical task must meet their deadlines. Every process needs computational and data resources to accomplish the task. Scheduling problems deals with the allocation of resources to satisfy the time constraints [10].

2.2 Traffic

As stated above, traffic congestion is becoming a serious problem in modern cities especially in developing regions resulting in massive delays, increased fuel wastage and monetary losses. Due to the poorly planned road networks, a common outcome in many developing regions is the presence of small critical areas which are common hot-spots for congestion; poor traffic management around these hotspots potentially results in elongated traffic jams. Poor road traffic management is the primary reason for extended periods of traffic congestion throughout the world. As per Texas Transportation Institute's 2011 Mobility report [3], congestion in the US has increased substantially over the last 25 years with massive amounts of losses pertaining to time, fuel and money. Sã o Paulo, Brazil is known to experience the world's worst traffic jams [4], where people are stuck for two to three hours every day in traffic jams. The issue of traffic congestion has affected both the developing and developed economies to different degrees irrespective of the measures taken to curb the issue. These sort of jams could be excluded when people can choose a different path for their destination rather than selecting the route with jams. As a result of this, load of vehicles on every road can be reduced and then a reduction of traffic jams could also be developed.

In past few decades, there have been several implementations of traffic monitoring system. And in recent times we have seen some implementation of crowd-sourcing and location data extraction.

2.2.1 Waze

Waze is a community-based traffic navigation application where people can share real-time traffic and road information to improve daily computing of all. It is a GPS-based geographical navigation application program. It also has a GPS support and displays screen that provides route details, user-submitted travel time and downloading location-dependent information. Waze was developed in Israel, which was funded by early-stage American venture capital firm Bluerun Ventures. Later on 2013 it was acquired by Google [11].

Waze is different from traditional GPS navigation software. As stated earlier it is a community-based traffic navigation application which gathers complementary map data and traffic information from its users. It works like other traditional GPS software in terms of learning from user's actions to provide routing and real-time traffic updates. People can report accidents, traffic jams, speed and police traps. Also from online map editor they can update roads, house, landmarks etc.

Waze requires a critical mass of users to have real utility. Currently, only 13 countries have a full based map and others are partially mapped. This indicates that a customized map

is necessary. This is not that much effective where there is not much information to start.

2.2.2 Google Traffic

Google traffic is a feature on Google maps which displays traffic conditions on major highways and roads in real time. It can be viewed in google map website and also in google map cellular application.

It works by examining the GPS-determined locations transmitted to Google by a large number of users. Google generate a live traffic map by calculating the speed of users along a length of road. Google maps also works on the basis of the concept of crowdsourcing, which refers to the process of soliciting electronic information from a large group of people. Google processes the incoming raw data about mobile phone device locations, and then excludes anomalies such as a postal vehicle which makes frequent stops. When a threshold of users in a particular area is noted, the overlay along roads and highways on the Google map changes color.

Early versions of Google Maps provided information to users about how long it would take to travel a particular road based on the historic data. This information was not real time and far from accurate. Currently Google traffic is available in 50 countries but not available in Bangladesh.

2.2.3 Go Traffic

Go Traffic is a mobile app that is expected to monitor and provide real-time update in the traffic situation in Dhaka. It is a startup in Bangladesh.

The approach of this application is similar to above mentioned systems. A team actively collects data about the traffic in various parts of the city and highlights them in three different colors Green for smooth traffic, Yellow but slow but moving and Red for jammed. However, the users can also contribute to this application through a Facebook group and through the application itself.

The main principle of this application is crowd-sourcing. They maintain a database of Dhaka city and the application allows the user to submit a manual report at any given time. After collecting all the report, it analyzes and update the condition of the roads.

Now all these system gives a condition of traffic which allows user to monitor the traffic status of different locations. But these system does not suggest any paths where a user wants to go. So by implementing shortest path algorithms on the graph with the help of data set we can suggest different routes which to choose or not according to the condition of the traffic on the roads.

2.3 Algorithms

2.3.1 Shortest Path Algorithm

In graph theory, shortest path problem is to find a path between two nodes in a graph that sums the minimum weights of the integral edges. The shortest path problem can be defined for both directed and undirected graphs. There are few criteria of shortest path problem. Among them the single-source shortest path problem is to find the shortest path from a single source, single-destination shortest path problem is to find the shortest path from all the vertices to a single destination and the all-pair shortest path problem is to find the shortest paths between every pair of vertices. Single-destination shortest path can be reduced to single-source shortest path problem by reversing the directed edges. There are some important algorithms that solves shortest path.

2.3.1.1 Bellman Ford Algorithm

A negative weight cycle is a cycle whose total weight is negative. No path from starting vertex to another vertex on the cycle can be a shortest path. Since a path can run around a cycle many times and get any negative cost desired, in other words, a negative cycle invalidates the concept of distance based on edge weight. The Bellman-Ford algorithm [12] propagates correct distance estimates to all nodes in a graph in V-1 steps, where V is the vertex number, unless there is a negative weight cycle. If there exist any negative cycle, it goes on relaxing its nodes indefinitely. Therefore, the ability to relax an edge after V-1 steps is a test for the presence of a negative weight. So, Bellman-Ford algorithm tests for negative weight cycles.

2.3.1.2 Floyd-Warshall

Floyd-Warshall algorithm is for solving all-pairs shortest path problem. It compares all paths in the graph between each pair of vertices [13]. It is used for weighted graph with positive or negative edge weights but with no negative cycles. The complexity of the algorithm is $\Theta(n^3)$, where n is the number of vertices.

2.3.1.3 A* search Algorithm

A*search is an informed search algorithm used for path finding and graph traversal. It combines the advantages of the Dijkstra's Algorithm for finding the shortest path and Greedy Best-Fit search for as it can use a heuristic to guide search. A* uses two functions, among them $g(n)$ represent the exact cost from the starting point to any vertex, $f(n)$ represent the

heuristic estimated cost from vertex n to the goal. A* balances the two as it moves from starting point to the goal. Each time through the main loop, it examines the vertex n that has the lowest $f(n) = g(n) + h(n)$. A* search gives complete and optimal solution. Its time complexity depends on the heuristic. As A* gives optimal solution, it cannot be chosen to get a solution for a city map.

2.3.2 Single Source Shortest Path Problem

Computing shortest path algorithm on graphs with non-negative weight is one of the most fundamental problems in computer science. The single source shortest path problem computes from a given specific source to other vertices in a graph calculating the minimum weights of the edges. For the computation of our approach we needed to implement the graph into a single source shortest path algorithm.

2.3.2.1 Dijkstra's Algorithm

There are many shortest path algorithm and among them the single source shortest path problem is the classical problem of optimization. We studied several algorithms and from them the most well known algorithm solving the problem with non negative weights was given by Dijkstra in 1959 [14]. The most efficient sequential algorithm on directional graphs with non negative weight is Dijkstra's Algorithm.

2.3.2.2 Sequential Dijkstra Algorithm

A weighted graph is defined by $G = (V, E, w)$ where v is a finite set of vertices, E is the finite set of edges and w is the weight function $E \rightarrow R^+$ and v_0 is the starting vertex. The algorithm generates a minimal paths exploring adjacent vertices from starting vertex to the remaining vertices. This technique of exploring vertices is known as relaxation technique. The technique consists of testing whether we can improve the shortest path found so far if so we update the shortest path. A relaxation step may or may not decrease the value of the shortest path estimate. In this step, for a particular edge having a source vertex and destination vertex, the $distance(destination)$ is set to $min\{distance(destination) + edge_{cost}(source, destination)\}$. The algorithm implements a loop. Let, M be the unresolved vertices and N be the visited vertices. In each iteration the estimates are for the M-vertices are relaxed. After that the minimum calculation of the M-vertices are computed and finally the M-vertex with minimum estimate will be stored in N. Meanwhile to track the path or sequence of the shortest path the predecessor of the selected vertices is also recorded. At last, if all the vertices are visited then algorithm is terminated.

2.3.2.3 Dijkstra's Algorithm Priority Queue

Dijkstra's Algorithm also uses a priority queue for an efficient implementation. For sparse graph, priority queue is used to store the reached nodes. As a result of this, the asymptotical behavior can be reduced [15]. Fredman and Tarjan used Fibonacci heaps and observed improved running times. The running time of Dijkstra using Fibonacci heaps is bounded by $O(n \log n + m)$ [16]. There is also a linear time RAM algorithm for undirected graphs with $\Theta(n+m)$ [17]. This takes $n > 2^{12^{20}}$ due to the usage of atomic heaps.

2.3.3 Parallel Versions of Dijkstra's Algorithm

As the graph of a city is a dense graph, so the runtime of sequential Dijkstra Algorithm cannot be optimized. As a result of this, we decided to use parallel version of this algorithm.

2.3.3.1 Standard Parallel version of Dijkstra

The standard parallelization of Dijkstra's Algorithm is to partition the graph where Q being shared to extract the minimum vertex. In Alistair's paper, the parallel implementation was implemented by using shared memory and message passing in shared memory machine. But his goals where partially met due to unavailability of the suitable shared memory machine. He also did some additional experiments on queue type and found out that the sorted queue had a negative impact on the parallelizability compared to unsorted queue. Also by using multilevel over-partitioning scheme the asynchronous parallel algorithm by Lanthier et al. [18] gave better performance but the performance of simple parallelization of Dijkstra's algorithm which is synchronous was worsened.

2.3.3.2 Another Approach

Another approach of parallel version of Dijkstra [19] were implemented based on Gabows's scaling algorithm [20]. The priority queue which is the relaxed heap attains the same amortized time bounds as the Fibonacci heaps, that is a sequence of two operation, decrease key and delete min. These two operation takes a time of $\Theta(m+n \log n)$. $\Theta(1)$ for decrease key and $\Theta(\log n)$ for delete min are the worst case time bound that can be obtained by different relaxed heaps. In this paper [21], the relaxed heaps gave a processor-efficient parallel implementation of the Dijkstra's Algorithm. In [19] they introduced a variant of priority queue called parallel bounded priority queue. It is able to perform two operations, insert and extract-min. The queue has a bounded size of n and consists of bins numbered from 0 to n-1. As the edges are non negative, every iteration of Gawbow's Algorithm only needs to consider monotonically

increasing bin indexes. As a result, it allows the algorithm to use a simple data structure for the priority queue which exposes parallelism. In this paper, they were able to expose parallelism on all but the least advantageous graphs. This approach performs well on random graphs, outperforming a simple Dijkstra implementation on six or more cores.

2.3.3.3 Δ-Stepping Algorithm

Mayers and Sanders Δ-stepping approach improves the situation for random directed graphs and uniformly distributed edge weights using linear work [22]. This Δ-stepping algorithm is a variant of Dijkstra's algorithm. In this algorithm the total ordering of the queue is weakened and an array of buckets is used to store the tentative distance [22]. In each phase, in the inner loop, all the light edges are relaxed and the remaining heavy edges are relaxed once. As long as a single relaxation is atomic, for a bucket, edge relaxation and deletion can be done in parallel. Also when the distance of a vertex is less than the tentative distance of that vertex then the algorithm eliminates the vertex from the queue and again it may reinsert the vertex until the distance is equal to the tentative distance.

2.3.3.4 GPU parallelization of Dijkstra's Algorithm

By changing the internal operations of the sequential Dijkstra algorithm, parallelism can be done. So it means the loops of the algorithm are paralleled. In each iteration, the outer loop takes a vertex and calculates the distances. The inner loop relaxes the outgoing edges of the vertex and update the previous distances. After all relaxation are done, it calculates the tentative distances from the unsettled vertices and extract the next frontier vertex. But only parallelizing the inner loops is not enough to exploit the huge number of GPU cores. So the outer loop is also parallelized. Without affecting the correctness of the algorithm the frontier set can be parallel but to identify the frontier set is a matter of concern. Δ-stepping approach [22] discussed above parallelizes the outer loop by exploring the vertices in the bucket. As the parallel algorithm advances the number of reached vertices computed in parallel increases so the GPU capabilities to solve the problem is more fitting [23]. In Ortega *et al*. paper, they presented a GPU SSSP algorithm implementation which speeds up the computation not only for CPU-based version but also other GPU implementation based on Dijkstra. The implementations were evaluated in NVIDIA architectures [23, 24] used two optimization techniques which lead to 23% performance improvement compared to non-optimized versions. The two optimization were modification of GPU L1 cache memory state and proper choice of thread-block size. In their implementation, they followed the idea of Crauser et al. [25], where the algorithm introduced an aggressive approach augmenting the frontier set with vertices of

greater tentative distances [25] algorithm computes in each iteration, for every unsettled vertex it calculates the tentative distance and the minimum weight of the outgoing edges. From these calculations it chooses the total minimum value. At last an unsettled vertex is put into the frontier set if the tentative distance is less than the total minimum value. This total minimum value is considered as a threshold value. So in [23] they used the threshold value by incrementing it. In pre computational phase they calculated the minimum weight of every vertices outgoing edges and then calculated the threshold value. As a result of this they were able to define the frontier set for very vertices. Before going into the details of the GPU implementation we needed to study the architecture of GPU.

2.4 GPU Architecture

GPU or Graphics processing unit is a special type of electronic chip which is used to rapidly manipulate and alter the memory to create images in frame buffer for display purpose. GPU is mostly found in personal computers, workstations, embedded systems, mobile phones and gaming consoles. Modern days GPUs are highly efficient in their work but they have also pivoted from their original purpose of only creating images in frame buffer. Their highly parallelism nature has proven more efficient than general purpose PUs. Algorithms where the processing of large blocks of data is done in parallel takes the advantage of modern GPUs. GPUs are found in different ways like, separate, embedded in mother board or sometimes integrated in processor like AMD's APU [26][27].

GPU or Graphics processing unit the term was introduced by NVIDIA corp. when they released their first ever GPU NIVIDIA GeForce 256 in 1999. NVIDIA's GPU enabled a new level of interactive content which was not possible before it arrived. It delivered an order-of-magnitude increase in geometry processing power, dynamic lighting and real-time environment reflection capabilities.

The GeForce 256 GPU incorporated many groundbreaking innovations that drove a major discontinuity in the 3D graphics industry, a market already known for its staggering pace of innovation. The new groundbreaking features available on NVIDIA's GPU included first 256-bit 3D processor, first integrated geometry transform engine,integrated dynamic lighting engine, first four-pixel rendering pipeline, Stunning new Microsoft's DirectX 7.0 features: cube environment mapping, projective textures and vertex blending [28].

There are several generations of NVIDIA GPU architectures. For our purpose we have gone through NVIDIA's G70 and before architecture as a group, G80 to Fermi, Fermi and Kepler architecture.

2.4.1 Up to NVIDIA G70

Up to NVIDIA's G70 GPU and there previous generation of architectures handled vertex and pixel shading in multiple dedicated units. They used an array where the top of the array was used to handle to vertex processing of 8 shaders and pixel processing was managed in middle of the array of 24 shaders. Gaming industry needed high pixel shader count as they performed difficult operation per pixel but simpler operation in vertex shaders. On the other hand business market they wanted more vertex shader count as their most applications can have any scene. As result having a dedicated shader became a problem because identifying the accurate number or shader in a GPU became hard gradually. Games grew more complex and scenes significantly varied in complexity instantly. When a scene's geometry overloads the vertex processor it created bottleneck issue. Many times pixel shaders sit idle until data were passed through from vertex shaders. This counting problem, idle sitting of hardware causes NVIDIA to shift their directions to new architecture [29].

2.4.2 G80 to Tesla

When G70 architecture failed to keep pace and solving issues like unused shader hardware a new architecture being developed that allowed hardware to allocate number of pixel and vertex shader dynamically as needed by current application. G80 was the first GPU architecture by NVIDIA which incorporated unified shaders with 128 processing elements which was distributed among 8 shaders core [29]. The classic pipeline was not available in the diagram as it was not the part of G80 architecture anymore. Moreover the pipeline loopback into itself before a pixel is presented to the frame buffer. In previous architectures there were dedicated few cores to each shader type, but in this architecture the scheduler can prioritize and allocate shaders to all the execution units. It is noticeable that an opportunity for an increase of performance is available because the hardware can easily increase the number of executing vertex shaders across the cores [30]. This architecture also introduced CUDA(Compute Unified Device Architecture). This was the first C-based development enviroment for GPU's [31]. Later to bring advantages Tesla product line was introduced [29].

2.4.3 Fermi

The Fermi based architecture is the most signification leap forward GPU architecture since the original G80. G80 was the vision or model of what unified graphics and parallel computing processor should look like. The Fermi was designed keeping in mind what prior two processor did and all the applications were written for them. Totally new approach was taken to create

world's first computational GPU. It was improved in some key points e.g Improve Double Precision Performance, ECC suuport, True Cache Hierarchy, More Shared Memory, Faster Context Switching, Faster Atomic Operations.

The improvement made by the Fermi Team based on request greatly increases the compute capability. With new innovations it greatly improved programmability and compute efficiency.

2.4.3.1 The key architectural highlights of Fermi are

Third Generation Streaming Multiprocessor (SM) which has 32 CUDA cores per SM, 4x over GT200,8x the peak double precision floating point performance over GT200,Dual Warp Scheduler simultaneously schedules and dispatches instructions from two independent warps, 64 KB of RAM with a configurable partitioning of shared memory and L1 cache

Second Generation Parallel Thread Execution ISA with Unified Address Space with Full C++ Support, Optimized for OpenCL and DirectCompute, Full IEEE 754-2008 32-bit and 64-bit precision, Full 32-bit integer path with 64-bit extensions, Memory access instructions to support transition to 64-bit addressing, improved Performance through Predication Improved Memory Subsystem for NVIDIA Parallel DataCacheTM hierarchy with Configurable L1 and Unified L2 Caches, First GPU with ECC memory support, Greatly improved atomic memory operation performance

NVIDIA GigaThreadTM Engine with 10x faster application context switching, concurrent kernel execution, Out of Order thread block execution, Dual overlapped memory transfer engines

The first Fermi based GPU had 3.0 billion transistors having up to 512 CUDA cores. A CUDA core executes a floating point or integer instruction per clock for a thread. The CUDA cores are organized in 16 SMs of 32 cores each. The GPU features a six 64-bit memory partitions, for a 384 bit memory interface. It supports up to a total of 6 GB of GDDR5 DRAM memory. A host interface connects the GPU to the CPU via PCI-Express. The GigaThread global scheduler distributes thread blocks to SM thread schedulers.

2.4.4 Kepler Architecture

Kepler is a GPU microarchitecture developed by NVIDIA as the successor of the Fermi microarchitecture. Kepler is the very first microarchitecture which focused on energy efficiency. Most of the GeForce 600 series, 700 series and some of 800 series is based on thi Kepler architecture. Which is 28 nm. It is also found in he GK20A, the GPU compoment of the Tegrea K1 Soc and Quadro Kxxx. Later Kepler was followed by Maxwell architecture [32].

In the Kepler architecture NVIDIA's priority was on efficiency, programmability and performance [33][34] where its predecessors were design keeping in focus on increasing performance on compute and tessellation [32]. Kepler could achieve the efficiency aim through the use of a unified GPU clock, simplified static scheduling of instruction and emphasis on performance per watt. They dropped the shader clock which was previously found on their GPUs but they achieved the higher level of performance with additional cores. Reason is cores are more power-friendly. Two Kepler core use 90% of power of one Fermi core but unified GPU clock scheme reduces 50% power consumption [35].

Another aim was programmability which was achieved by Kepler's Hyper-Q, Dynamic Parallelism and multiple new Computer Capabilities 3.x functionality. Higher GPU usage and simplified code management was achieveable with GK GPUs. Which enables more flexibility in programming for Kepler GPUs [36].

Lastly with performance aim, extra execution resource and with Kepler's ability to achieve a memory clock speed of 6Ghz, which increases Kepler's performance comparing to its predecessor [35].

Like Fermi, Kepler GPUs are composed of different configurations of Graphics Processing Clusters (GPCs), Streaming Multiprocessors (SMs), and memory controllers. The GeForce GTX 680 GPU consists of four GPCs, eight next- generation Streaming Multiprocessors (SMX), and four memory controllers.

Kepler GPUs are made of different configurations of Graphics Processing Clusters(GPCs), Streaming Multiprocessors (SMs) and Memory controllers like its predecessor Fermi. The GeFore GTX 680 GPU consists of four GPCs, eight next-generation Streaming Multiprocessors (SMX) and four memory controller [37].

The Kepler GK110 GPU is comprised of 7.1 billion transistors. It is an engineering created to address the most daunting challenges in HPC. It is designed from the scratch to amplify the computational performance with superior power efficiency. The innovations in this architecture has made the hybrid computing dramatically easier, which is applicable to a broader set of applications and more accessible [37].Kepler GK110 GPU is a computational workhorse with teraflops of integer, single precision, and double precision performance and the highest memory bandwidth. The first GK110 based product will be the Tesla K20 GPU computing accelerator [37].

2.5 CUDA Overview

2.5.1 CUDA

With the advancement of graphics hardware it was required to have a certain platform so that developers can use the power of those hardware. In 2007 NVIDIA came with a solutions for the developer named CUDA [29] CUDA or Compute Unified Device Architecture is a parallel computing platform developed by NVIDIA corp. It is an application programming interface (API) which supports parallelism [38]. CUDA platform helps programmer to use CUDA-enable GPUs to use as a general purpose processing which is known as GPGPU. CUDA provides a layer of software which directly access the virtual instruction set and other parallel computing elements to execute computer kernels of GPU [39]. The CUDA SDK was made available to the public freely. They have evolved since then. Their on of major release was 3.0. At present they have CUDA version 7.5 [40].

CUDA platform is developed in such a way that most popular machine level language like C, C++, Fortran works with it. CUDA platform is very handy comparing to other parallel programming platform like OpenGl or Direct3D because it required advance skills in graphics programming. Programming framework such as OpenACC and OpenCL are also compatible with CUDA [39].

2.5.2 Poragramming capabilities

The CUDA platform is available to developers through CUDA-accelarated libraries, compiler directives, for example, OpenACC, and extensions to industry-standard programming languages including C, C++ and Fortran. C/C++ programmers use 'CUDA C/C++', compiled with "nvcc" – NVIDIA's LLVM-based C/C++ compiler [41].

2.5.3 Advantages

CUDA offers several advantages over exsisting general-purpose computation on GPUs with the help of graphics APIs. It can do scattered reading which means code can read from random addresses in memory. It exposes a shared memory area which is shared among the threads. It can serve many purpose such as user-managed cache, enabling higher rate of bandwidth. It also have unified virtual memory (CUDA 4.0 and above), unified memory (CUDA 6.0 and above). Quicker download and readbacks to and from GPU. Total support for integer and bitwise operations [42].

2.5.3.1 GPU Accelarated Computing

GPU accelarated computing is a new type of computing combining GPU and CPU together to speed up the analytics, engineering, scientific and enterprise application. In 2007 NVIDIA prioneered this GPU accelerators which is now found in energy-eficient datacenters in goverment labs, universities, enterprises and small and medium business across the globe. GPU accelerated applications are also found in different platform starting from cars to mobile phones, tablets, drones and robots [43].

2.5.4 How GPU Accelarate Operation

While the remainder of the code still runs on CPU, GPU-accelerated computing provides an application performance which was significantly new by offloading computing-intensive part of application to the GPU [43].

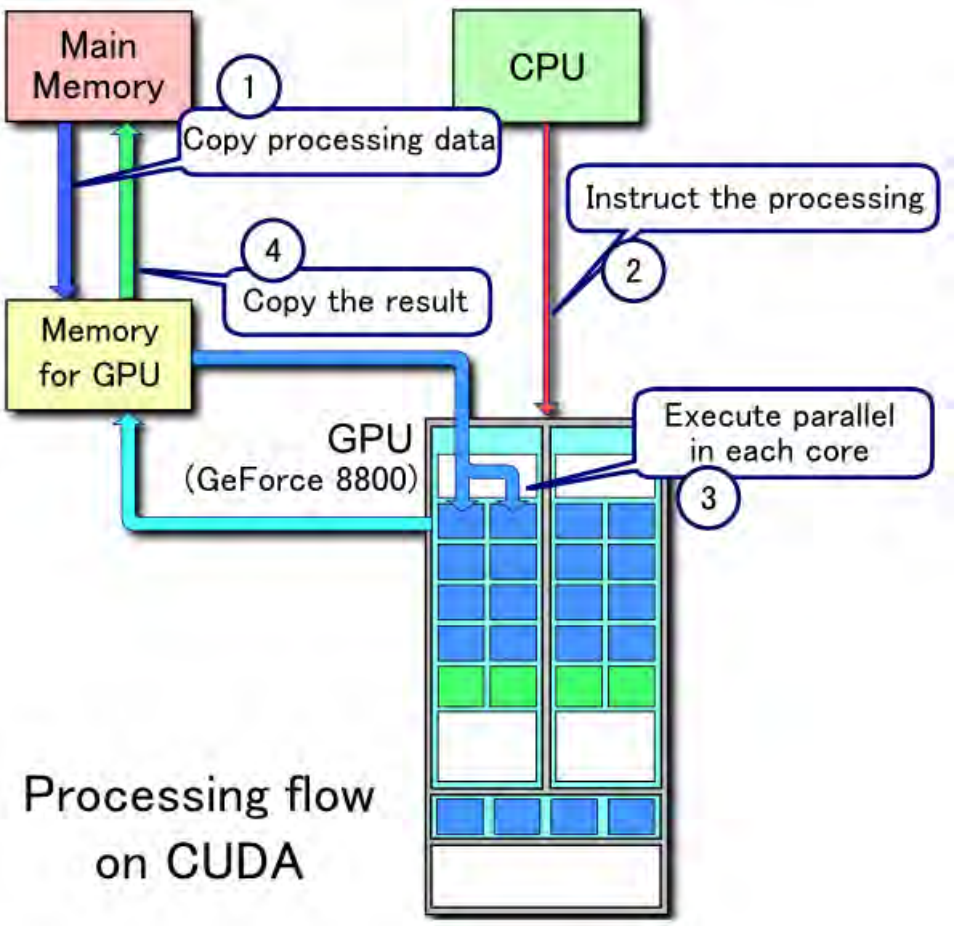


Figure 2.1: CUDA Processing Flow[1]

2.5.5 CPU vs GPU

The operational difference happens between CPU and GPU are very simple. In CPU there are few cores which is optimized for sequential processing whereas a GPU has thousands of smaller, more efficient cores dedicatedly architectured keeping in mind of massive parallelism to handle multiple of tasks simultaneously [43].

2.5.6 Kernel

Kernel is a special type of function in CUDA C which is extended from C which allows programmer to define functions. When this kernel is call it is executed N times in parallel by N different CUDA threads.

To call a kernel from a C function is specified using a <<<.....>>> execution configuration syntax. This configuration also specifies how may CUDA threads and block will be running in parallel. To make a function CUDA kernel we need to specify __global__ keyword. There is a built-in threadIdx variable by which a certain thread is accessible where all the threads are given a unique thread ID [44].

2.5.7 Block

Block is consist of some threads. The number of threads a block can have and number of block is determined in <<<...>> syntax. The type can be integer or dim3.

Built-in variable blockIdx is used to access the block within the grid which is identified by a one-dimensional, two-dimensional or three-dimensional index. "blockDim" is a built-in variable in CUDA which is used to access the dimension of the thread block within kernel.

The execution of thread blocks need to run independently. It is possible that the execution of it is in any order, may be sequential or parallel. This independence of execution enables thread blocks to be scheduled in any particular order throug any numbers of cores which allows programmers to write the program that can scale with the numbers of the core [44].

2.5.8 Thread

Threads can be identified using a one-dimensional, two-dimensional or three-dimensional thread index. It is a 3-component vector. It forms a one-dimensional,two dimensional or three-dimensional block of thread known as threadblock [44]. It allows a simple computation across the elements in domain like vector, matrix, or volume [44].The index of a thread and its thread ID relate to each other in a simple manner.

Though it is very flexible creating thread but there is a limitation in number of thread in a block. Because all threads of block are have to share the shared memory and does computation in same core. Today's GPU supports upto 1024 threads in per block.

Chapter 3

Methodology and Design

As stated in the above, mobility is essential in a busy city as it influences the socio-economic activities. So, smooth transportation is needed to carry out the objectives. Traffic congestion has been a fundamental problem for modern cities as there is a wide usage of automobiles. So to satisfy the problem we needed to follow crowd-sourcing approach to collect the data to identify the traffic situation on the roads. And later on we implemented the collected data into a shortest path algorithm which helps to distribute the traffic by providing shortest route to the destination. Initially we completed the objectives and did the literature review that we needed. After that to solve our problem we started to design our approach and algorithms. For running the algorithm, we also represented the graph into suitable data structures. We tested our implementation in CPU and GPU and then compared the result.

3.1 Graph Representation

A graph $G(V,E)$ where V is the number of vertices and E the number of edges is commonly used to represent adjacency matrix. It is suitable when the number of vertex and edge is less. But for sparse graphs such representation wastes a lot of space. It takes lots of memory resulting inefficiency. Adjacency List is more compact in graph representation. In this paper, we have represented the graph in the form of adjacency list as it takes less space compared to adjacency matrix representation. Also because of variable size of edge list per vertex, the GPU representation may not be efficient under the GPU model. CUDA allows arrays of arbitrary sizes to be created and hence can represent graph using adjacency list. We stored all the vertices of the adjacency list into a single large array. The data representation consists of vertex array V_a and edge array E_a, where each vertex, V_a points to the starting position of its own adjacency list in this large array of edges. The E_a array stores the edges of the vertex I for all I in V. One extra element is needed in vertex array to indicate out degree of last vertex

shown in the figure below. Each entry in the vertex array V_a corresponds to the starting index of its adjacency list in the edge array E_a. Each entry of the edge array Ea refers to a vertex array V_a. Moreover, another array is used to store the weights, W_a of each weights. This are required for parallel implementation such that each thread can run on edges rather than on vertices.

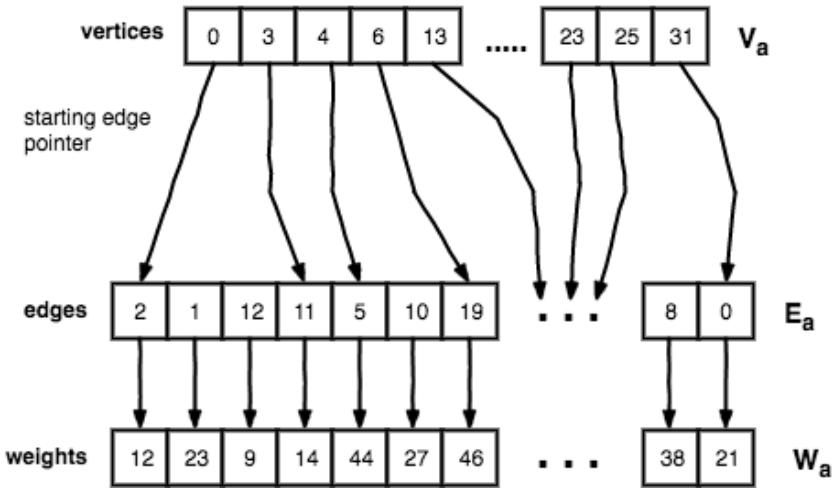


Figure 3.1: Graph representation with vertex pointing to edge and edge pointing to weight

3.2 City Highway Graph

A map of a city has many roads and road intersections. The intersection of a road is a convergence point from two or more roads. We considered these intersection points as Vertex and the roads as Edges. By doing so, a city map becomes a graph. Now at both the ends of a span of a road, the traffic signals are used. So, at any given time, the density of the vehicles in between any two nodes in the city map represents the amount of traffic on that span of the road. For the city map, we considered a small section of Dhaka city based on the principle that we discussed above. This data structure holds the information about the specific road-spans which allows the identification of each individual road an also the information of the distances of the corresponding span through which the shortest path is determined.

3.3 Metric System

In some previous case studies, we saw that they used GPS to locate the location of the traffic. The users give direct input to determine the density of traffic. This process is the crowd sourcing method where the system is updates according to the person participating in the poll

or submitting the report. So, the data of all the roads condition are collected and are structured in a data structure which holds the following information, the intersection points of the road and the condition of the road. To illustrate the condition of a road accurately, we calculated a threshold in the following, for a set of speed limit. For example:

- 0-15 km/h as heavy traffic.
- 16-30 km/h as moderate traffic.
- Above 30 km/h as light traffic.

Considering this threshold level, for a particular edge a new weight will be updated by augmenting the existing weight. We used a function that will change the weight according to the condition of the road. This will help to create the edge heavier when there will a long congestion of traffic and reduce the weight when the traffic is light.

3.4 Edge based Dijkstra

As discussed earlier, in Dijkstra's Algorithm, the vertices are divided into two categories, settle and unsettled vertices. Settle vertices are the vertices that have minimum vertex weight and whose outgoing edges are relaxed while unsettled vertices may be unreachable vertices. This approach means the vertex weight is equal to infinity or the vertex does not have minimum weight. Initially the source vertex is settled and its outgoing edges are relaxed. In next iteration, the vertex which have the minimum weight is settled and its outgoing edges are relaxed. This procedure is repeated until all the vertices are settled. We used this algorithm approach in for the CPU version. Later on, this edge based relaxation concept is also used in GPU.

3.5 Algorithm Design

In meeting the specified goals of the project, there were a variety of choices of algorithm that could have been implemented. The Dijkstra Algorithm were chosen as the basis of the comparison as it is so well studied and relatively straightforward. For better optimization we chose parallel algorithms over sequential algorithms where the algorithms work on GPU architectures. We used as references the implementation of [24]. In their paper they followed the approaches of the sequential implementation on CPU and the fastest version implementation for GPU of [45]. Alongside they used the same input set of [45] to compare the performance gain of their algorithms. We selected the GPU approach of [24] and used the same CUDA configuration values of 256 threads per block kernel and L1 cache normal state (16KB).

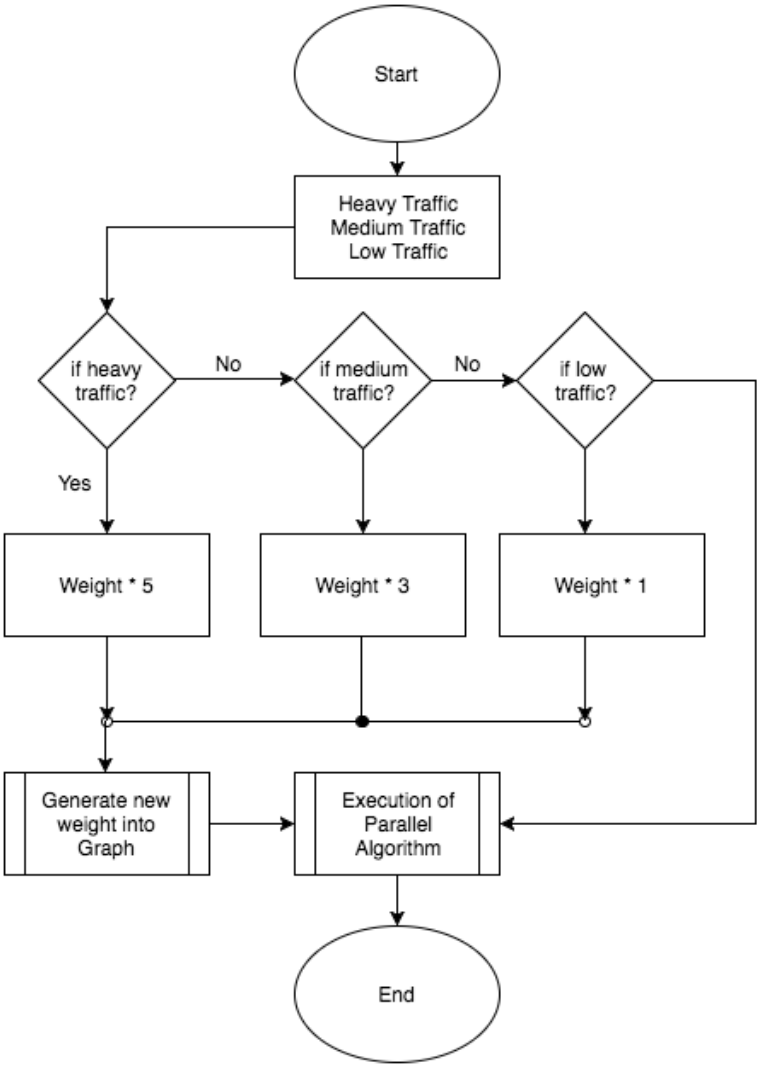


Figure 3.2: Flow Diagram

We constructed a path by using this algorithm. In relax kernel when the successor of the vertices is relaxed then a predecessor is set while calculating the distance. Here the algorithm checks whether any of the predecessor vertex belongs to the current frontier set. So, if the new distance is less than the previous distance then the tentative distance is relaxed.

When every vertex in the frontier set are relaxed parallel, we calculate the predecessor of the vertex from the above algorithm. For a particular vertex, if the cost of the successor and the predecessor of the vertex is not settled then we ignored the vertex as this is not settled yet. Again if the the cost of the predecessor is less than the successor then the vertices in the frontier set are about to settle and if the total cost on an edge is equal to the cost that which has been computed in [24] then we can settle the predecessor of that current vertex.

```
tid = thread.ID
if tid is in frontier set then
    for all i successor of tid do
        sid = successor of edge i
        if sid is unsettled then
            if cost of sid and tid is settled then
                if cost of tid < sid then
                    if cost of tid and edge,i = cost of sid then
                        parent[sid] = tid
                    end
                end
            end
        end
    end
end
```

Algorithm 1: Predecessor Settle

Chapter 4

Implementation and Development Environment

4.1 Implementation

The problem we are trying to solve here is actually divided into two major parts. We have gone through several algorithms and there efficiency and accuracy. In this area, real time output and accuracy highly matters. We refer to algorithms [23]. We have extended that algorithm to obtain the shortest path. As this algorithm works in parallel in different CUDA threads, obtaining parent node for a particular node cannot be done in traditional way. The way that algorithm gets the minimum cost is using CUDA reduction [24] method. Because the algorithm uses the advantages of thread and parallelism its every information for a particular node is obtained independently.

We have made some changes in the algorithm to achieve our desired goal. Obtaining path is one of them. For obtaining the path in this environment we have given some conditions which actually ensures the correct path and also prevent any kind of anomalies which can happen related to memory or path. To prevent memory error we checked whether the cost of the predecessor or successor is updated or not. Secondly the cost of the predecessor is less than the cost of successor. The reason behind this checking is all the threads work in parallel so there is always a chance that successor and predecessor becomes each others successor and predecessor and in that case if it is not checked it will fall into an infinite loop. And lastly we compare the cost of predecessor and weight of the current edge with cost of successor. If this condition passes successfully then we set the parent. The parent is saved in an array which is in device memory. After that the parent array is copied to CPU memory using `cudaMemcpy` function. Later on that array in CPU memory is used to retrieve the shortest path.

We modified the code to take the input from the web form and we have set some of the parameters fixed as our need, in “template.cu” class in “main” method.

```
36 // Program main
37 int main( int argc, char** argv) {
38
39         char *data;
40         int source, destination;
41         data = getenv("QUERY_STRING");
42
43         if(data == NULL){
44
45                 printf("<P>Error! Error in passing data from form to script.");
46         }
47
48         else if(sscanf(data,"source=%d&destination=%d",&source,&destination)!=2){
49
50                 printf("<P>Error! Invalid data. Data must be numeric.");
51         }else{
52
53                 //do nothing
54         }
55
56         int             iter = 1;
57         sprintf_s(file,"%s",        "merge.graph",0);
58         sprintf_s(rfile,"%s",       "xnorandom2.source",0);
59         conf                = 10;
60
61         launch_sssp(iter, source, destination);
62 }
63
```

Figure 4.1: Taking Input from user from Web Form

In the above figure 4.1 in line number 48 we are taking the source and the destination from the user and storing those into pre-defined variables which were declared in line number 40. From line number 56 to 59 we set the value by ourselves where we do not need user input. Line number 56 is taking how many times the program will run. We set it to 1 because we will run the program once to identify path. In line number 57 we are giving our graph which is fixed for a particular city. This file is the the city graph. For cache configuration we are setting the cache 10 in line number 59.

Next we have changed and declared things in “launch_sssp” method which is still in the “template.cu” class. We have initialized our parent array in line number 76 and declared our stack in line number 84. As we have implemented CGI so we need to add line number 89 so that it works while printing.

```
64 void launch_sssp(int iter, int source, int destination){
65
66        ////////////////////////////////////////////////////////////////////////////////////////////////////
67        // VARS AND GRAPH FUNCTIONS
68        unsigned int nv, na, msizeV, msizeA;                          // number of vertices and edges, array sizes
69        unsigned int *vert, *arcs, *wght;                             // vertices, arcs and weights arrays
70        read_graph_from_file(file,nv,na,msizeV,msizeA,vert,arcs,wght);
71
72        unsigned int mem_size_F = sizeof(bool) * nv;
73        bool* p= (bool*) malloc( mem_size_F);                         // unsettled vector
74        bool* f= (bool*) malloc( mem_size_F);                         // frontier  vector
75        unsigned int* reference = (unsigned int*) malloc(msizeV);     // tentative solutions
76        unsigned int* parent = (unsigned int*) malloc(msizeV);
77        nv = nv-1;
78
79        unsigned int* v_delta  = (unsigned int*) malloc(msizeV);                  if(  v_delta==(unsigned int*)NULL) exit(NOT_ENOUGH_MEM);
80        unsigned int* numRandom= (unsigned int*) malloc(iter*sizeof(unsigned int)); if(numRandom==(unsigned int*)NULL) exit(NOT_ENOUGH_MEM);
81        calc_vdelta(v_delta, vert, wght, nv, na, infinito);           // Crauser values calculation
82        // reading random source indexes, avoiding not connected sources and
83        random_conmod(numRandom, rfile, iter, nv, vert, source);              // moduling source indexes bigger than total number of
  vertices of the graph
84        stack<int> myStack;
85        ////////////////////////////////////////////////////////////////////////////////////////////////////
86        //Implementations of GPUs                    ----------------------------------------------------------
87        cudaDeviceProp props;
88        cudaSetDevice(0);                    //0_Kepler    1_Fermi
89        printf("Content-type:text/html\n\n");
90 #ifdef CHECK
91        cudaGetDeviceProperties(&props, 0);
92        printf("CUDASETdevice(%u) propsmajor %u.%u\n", 0, props.major,props.minor); fflush(stdout);
```

Figure 4.2: Stack, Parent, CGI

For obtaining the sequence of the path, we initialized a stack line number 84 (Figure 4.2). The path is stored into a stack which allows to maintains a precise order. Following code we added to get our path.

```
113       // SSSP execution
114       for(unsigned int ix=0; ix<iter; ix++){
115
116               gpu_sssp_Atomic( parent, reference, p, f, nv, msizeV, na, msizeA, infinito,
117                               vert, arcs, wght, v_delta,  numRandom[ix],  /*numRandom[ix] is the source index,*/
118                               v_d,  a_d,  w_d,  v_delta_d,  dimrel, dimmin, dimupd );
119
120               printf("<p>");
121               if(reference[destination] != INT_MAX) {
122
123                       unsigned int i=destination;
124
125                       printf("Path: ");
126                       for (; reference[i] != 0; i = parent[i]){
127
128                               myStack.push(i);
129                       }
130                       myStack.push(i);
131
132                       printf("%d",myStack.top() );
133                       myStack.pop();
134                       while(!myStack.empty()){
135
136                               printf("-->" );
137                               printf("%u",myStack.top() );
138                               myStack.pop();
139                       }
140                       printf("\n" );
141
142               } else{
143
144                       printf("No path found!\n");
145               }
146               printf("</p>");
147       }//for
```

Figure 4.3: Path Printing

The “comun.cu” class has also gone some under modification. In “random_conmod” method we have changed from random destination to fixed source. Following figure line number 60 shows it.

```
57                          aux = aux % (nv-2);
58                  }
59
60                  numRandom[index] = source;
61          }
```

Figure 4.4: Fixed Source

When “read_graph_from_file” method is called from line number 70 in figure 4.2. We are reading a file which is our “input” file which contains the data of the road. Line number 77 and from 84 to 89 has been written. Following is the code snippet.

```
76          FILE *fileIn;
77          FILE *filename2;
78          if ((fileIn = fopen(filename, "r")) == NULL){           /* Control de apertura*/
79                  //printf ("Error en la apertura del IN. Es posible que el fichero no  exista\n ");
80                  printf ("Error while opening file IN. It is possible that file does not exists\n");
81                  exit (-1);
82          }
83
84          filename2 = fopen("input.txt","r");
85          if(filename2 == NULL) {
86
87                  perror("Error in opening file");
88                  exit(-1);
89          }
```

Figure 4.5: Reading input file

Another major part how to deal with traffic jam and reroute can be done. We developed a module which identifies the edges getting the input from the user about a particular road. We call the module from line number 125 in figure 4.6. After a certain period a input file is generated from the user input which contains the situation of traffic. Our program reads that input file figure out and work out the edges. When a road is detected with different kind of jam situation, we have multiplied the length of the road with a fuzzy number. If a road contains heavy jam, moderate or low jam we multiplied the road length 5, 3, 1 respectively. Our purpose was how we can minimize the travel time. So we calculated the time with distance. The number we have chosen to multiply is fuzzy. There are scope for experiment to figure out more criteria and values of a road. Our solution is a general purpose solution which can take a file about road situation in a particular format and generate path. So when we had done this calculation it lengthens the road and consider lengthier road then usual. So our program calculates with the updated length in real time and generate a parent array which is stated above.

```
112         while(1){
113
114
115                 //sum = c+d;
116                 fscanf(filename2, "%u", &c);
117                 fscanf(filename2, "%u", &d);
118                 fscanf(filename2, "%u", &e);
119
120                 if( feof(filename2) ){
121
122                         break ;
123                 } else {
124
125                         calculateWeight(c,d,na,v,a,w,e);
126
127                 }
128
129
130         }
131         fclose(fileIn);
132         fclose(filename2);
133 }
134
```

CUDA ▾ Tab Width: 8 ▾ Ln 134, Col 1 INS

Figure 4.6: Taking input and sending to module

This module takes two node and their current situation, figure out the edge and then multiply with the number. By this it generates a new array which is given as a input to find out the path.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <stdbool.h>
5
6
7 void calculateWeight( unsigned int startNode,  unsigned int finishNode, const unsigned int na,
8                 unsigned int* vertices, unsigned int* nodes, unsigned int* weight,const unsigned int situation){
9
10        if (startNode == 0){
11
12                unsigned int sum = 0; unsigned int result =0; unsigned int counter;
13                for(counter = 0; counter<=nodes[0]; counter++){ if(nodes[counter]==finishNode){ result = counter;break; }}
14
15                if (situation == 1) weight[result]= weight[result]*1;
16                else if (situation == 3) weight[result]= weight[result]*3;
17                else weight[result] = weight [result]*5;
18
19        }else{
20
21                unsigned int sum=0; int result=0; bool flag = true; int counter;
22                for(counter = 0; counter<=startNode; counter++)sum =+ vertices[counter];
23
24                counter = 0;
25                for( counter = sum; counter<na; counter++){ if(nodes[counter]==finishNode){ result = counter; break; } }
26
27                if (situation == 1) weight[result]= weight[result]*1;
28                else if (situation == 3)weight[result]= weight[result]*3;
29                else weight[result] = weight [result]*5;
30        }
31 }
```

Figure 4.7: Module

When “gpu_sssp_Atomic” is called we are taking a parameter which is “parent_h”. This parent_h array resides in our CPU memory. We also initialized a “parent_d” array and allocated a memory in our device (GPU). A code snippet is given below.

```
39
40 void gpu_sssp_Atomic( unsigned int* parent_h, unsigned int* c_h, bool* p_h, bool* f_h,
41                const unsigned int nv, const unsigned int mem_size_V, const unsigned int na,   const unsigned int mem_size_A,
42                const unsigned int inf, const unsigned int* v_h, const unsigned int* a_h, const unsigned int* w_h, const unsigned int* v_delta,
43                const unsigned int source,   const unsigned int* v_d, const unsigned int* a_d, const unsigned int* w_d, const unsigned int* v_delta_Device,
44                const unsigned int drel,     const unsigned int dmin, const unsigned int dupd){
45
46        unsigned int mem_size_C = mem_size_V;
47        unsigned int mem_size_F = sizeof(bool) * nv;
48        unsigned int* c_d;                                          //device-vector_solution
49        bool* f_d;                                                  //device-vector_frontier (frontier nodes)
50        bool* p_d;                                                  //device-vector_pending  (unsettled nodes)
51        unsigned int* parent_d; //device-vector-parent
52        // allocate device memory for result and copy the host-vectors (c_h, f_h, p_h) into the device-vectors (c_d, f_d, p_d)
53        CUDA_SAFE_CALL( cudaMalloc( (void**) &c_d, mem_size_C));
54        CUDA_SAFE_CALL( cudaMalloc( (void**) &parent_d, mem_size_C));//allocating device memory for parent
55        CUDA_SAFE_CALL( cudaMalloc( (void**) &f_d, mem_size_F));
56        CUDA_SAFE_CALL( cudaMalloc( (void**) &p_d, mem_size_F));
57
```

Figure 4.8: Parent array initialization in Device Memory

Finally the following code was added which is used to identify the correct path by setting the parent in “template_kernel.cu” class in “kernel_relax_Atomic” method.

```
134
135         if(c_d[sid] != INT_MAX && c_d[tid] !=INT_MAX && c_d[tid] < c_d[sid] && (c_d[tid]+w_d[i]) == c_d[sid]) {
136                 parent_d[sid] = tid;
137         }
```

Figure 4.9: Parent Setting

Lastly, we cleared the memory and following two figures exhibiting that. Figure 4.10 frees memory from CPU and figure 4.11 frees memory from GPU.

```
152         CUT_SAFE_CALL( cutStopTimer( timer)); printf( "TOTAL: %.2f", cutGetTimerValue( timer)/iter); CUT_SAFE_CALL( cutDeleteTimer( timer));
153
154         ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
155         // FREE RESOURCES
156         free( vert); free( arcs); free( wght); free(v_delta); free(p); free(f); free(reference);
157         free(parent);
158         CUDA_SAFE_CALL(cudaFree(v_d)); CUDA_SAFE_CALL(cudaFree(a_d)); CUDA_SAFE_CALL(cudaFree(w_d)); CUDA_SAFE_CALL(cudaFree(v_delta_d));
159
```

Figure 4.10: Freeing parent from CPU memory

```
75         // SSSP computation
76         cudaThreadSynchronize();
77         bool finished = false;
78         while(!finished){
79                 finished = it_SSSP_Atomic( gdimrel, dimrel, gdimmin, dimmin, gdimupd, dimupd, inf, minimoDelBloque_h, mem_size_Minimizar,
80                                 v_d, a_d, w_d, p_d, f_d, c_d, minimoDelBloque_d, v_delta_Device, nv, parent_d);
81         }//while
82
83         // Recover the solution to host
84         copiarD2H((void*)c_h, (void*)c_d, mem_size_C);
85         copiarD2H((void*)parent_h, (void*)parent_d, mem_size_C);
86
87         // Free memory
88         CUDA_SAFE_CALL(cudaFree(c_d)); CUDA_SAFE_CALL(cudaFree(f_d)); CUDA_SAFE_CALL(cudaFree(p_d));
89         CUDA_SAFE_CALL(cudaFree(minimoDelBloque_d)); CUDA_SAFE_CALL(cudaFree(parent_d));
90         free(minimoDelBloque_h);
91 }
```

Figure 4.11: Freeing parent from GPU memory

We needed our application to run all the time whenever a client requests a path or several clients request a path. If it was a program to run for once and then stop then it wouldn’t have been a problem. But it is a program which needs to be running all the time. Our program is written in CUDA C/C++. So we cannot implement our program traditionally in server. To

overcome this barrier we used used CGI (Common gateway interface) which can be accessed by Apache Tomcat server. We kept our executables in cgi-bin. So whenever a client requests for a route CGI runs an instance of our executable and provide the path.

We built a simple user interface where client can easily give their source and destination and get the output in a real time scenario. As our backend is written in CUDA C/C++ so it generates the result swiftly. From this page when a client send a post request it invokes the executable in cgi-bin with the necessary parameter. If the input is in correct format the program starts executing.

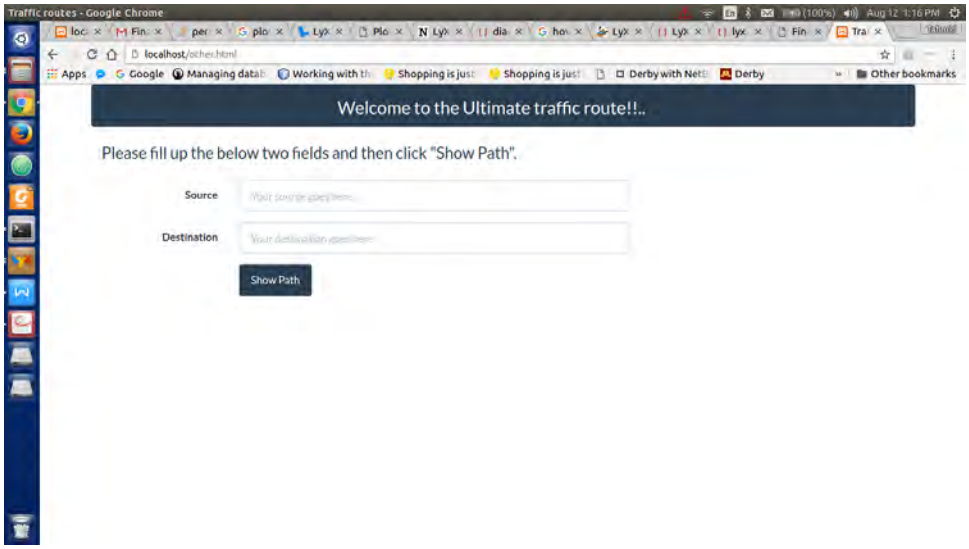


Figure 4.12: Interface

Below figure is an output of our program. This shows the node number which have to follow to reach the destination at minimum time possible.

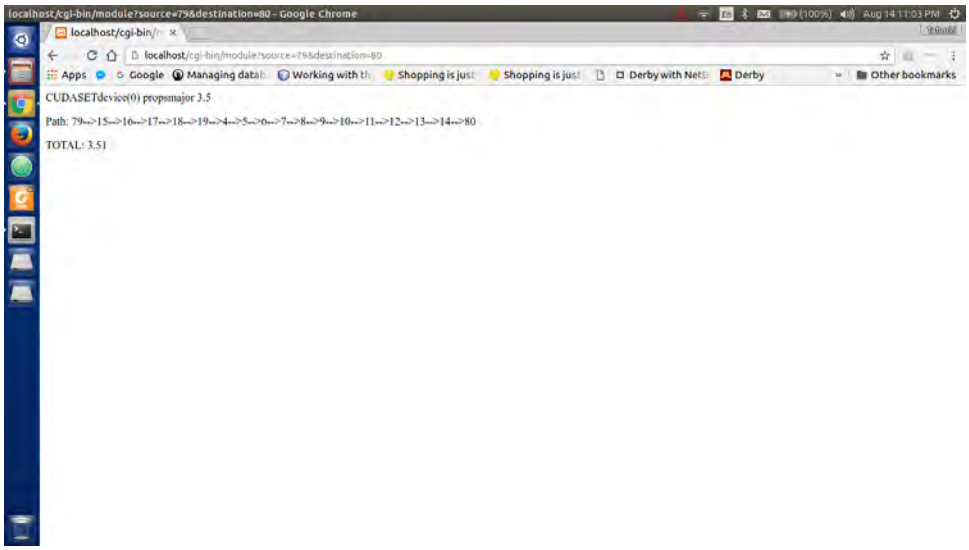


Figure 4.13: Output

To get the information of the road such as what is the condition of the road, we built another web interface from where a user can easily update the situation of road by choosing the source, destination and nature of traffic jam. It is saved as a database which later on works as an input of the main program from where our module reads the traffic situation.

In the following section we briefly discuss about our experimental setup. What we have used to develop our project.

4.2 Development Environment

In our experiment we have used different set of environment. We tested our program in three NVIDIA GPU devices, a GeForce 920M (Kepler GK 208), GeFocre GTX 760 (Kepler GK 104), GeFocre GTX 950(Maxwell GM 206). The configurations of those host machines are given respectively. First host machine has an Intel(R) core i3 processor with 3.2 GHz, with a memmory of 4GB DDR3. The operating system in this host machine is Ubuntu Desktop OS 14.04 (64 bits)and CUDA toolkit version 7.5 is used. Second machine has Intel(r) core i5 processor with a memory of 8 GB DDR3. It also runs on Ubuntu Desktop 14.04 (64 bits). The programs were run using CUDA toolkit 7.5. Lastly, the machine has Intel(R) Dual Core 2.3 Ghz processor with global memory of 8 GB DDR2. In this machine the CUDA toolkit was also 7.5 and operating system was also Ubuntu-Desktop(64 bits). The programs were compiled with gcc compiler using the flag -03.

4.3 System Overview:

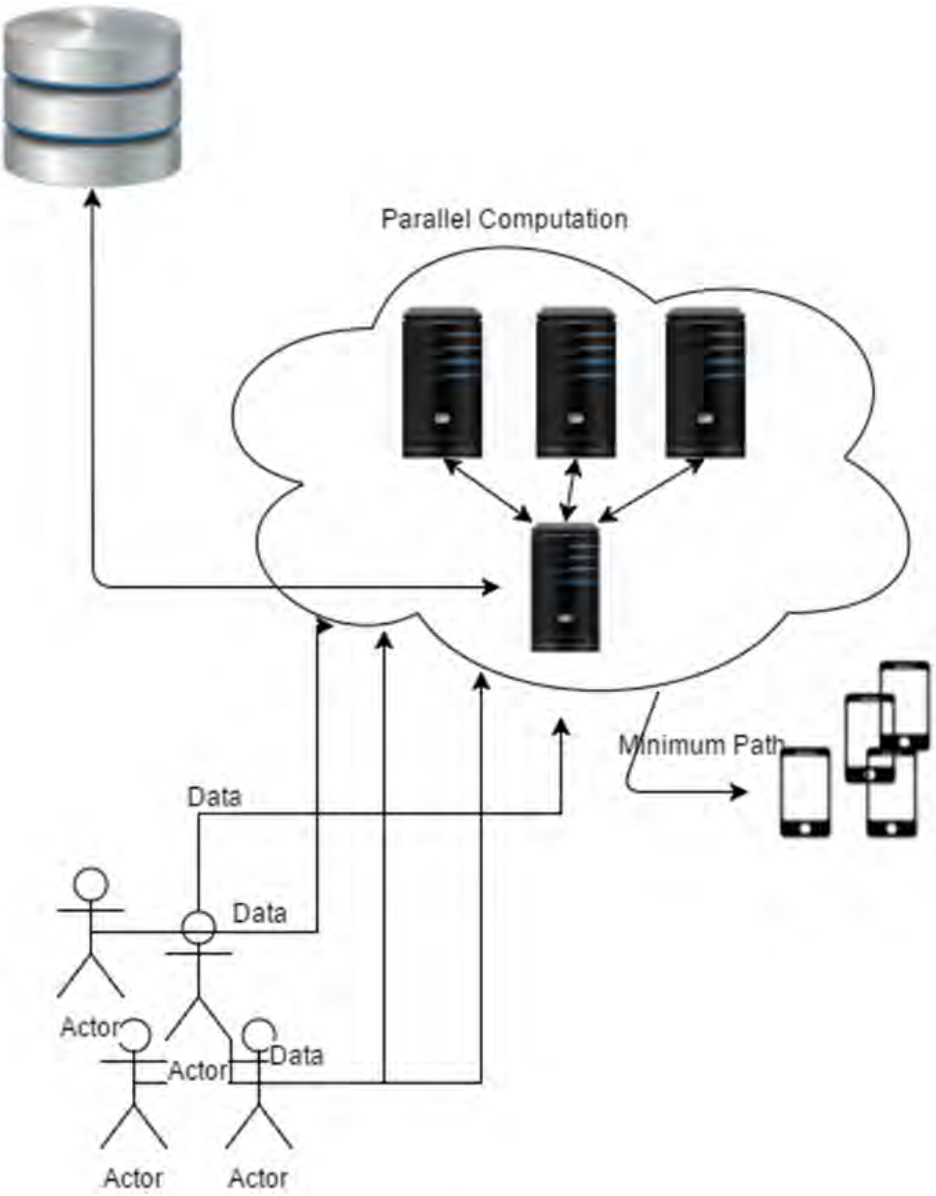


Figure 4.14: System Overview

Chapter 5

Result

5.1 Data set

This section describes the results of our implementation that were described in the previous sections. We tested the performance of our implementation using some simulated data. A data set of Rome city consisting of 3353 vertices and 8870 edges and a data set of New York City consisting of 94346 vertices 129684 edges.

For our implementation, due to insufficient resources we tested on a simulated graph of Dhaka city consisting of 102 vertices and 286 edges. Following shows a representation of the number of vertices and edges that we used for testing the performance of the algorithms.

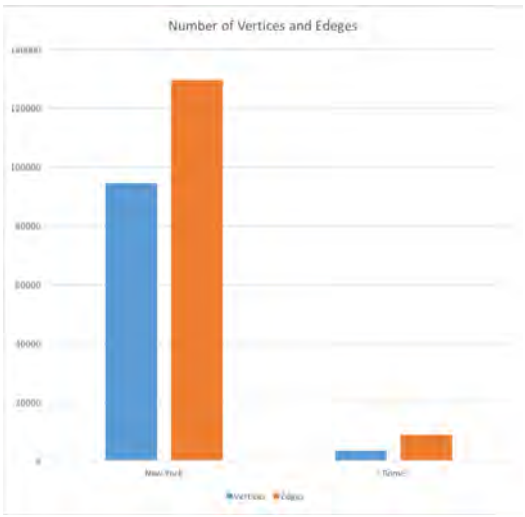


Figure 5.1: Number of Vertices and Nodes of New York City and Rome

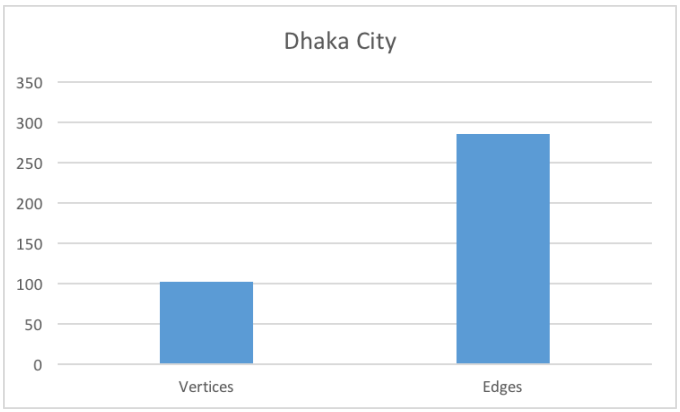


Figure 5.2: Number of Vertices and Nodes of Dhaka City

5.2 Running Algorithm on CPU

5.2.1 Rome

We implemented the sequential edge based Dijkstra using the environment that was discussed in the previous section. Implementing the data set of Rome city, we got the following output

Source	Destination	Execution Time (ms)
62	116	1232
54	1777	1156
2332	1354	1263
675	343	1251

Table 5.1: CPU Execution time (ms) for Rome

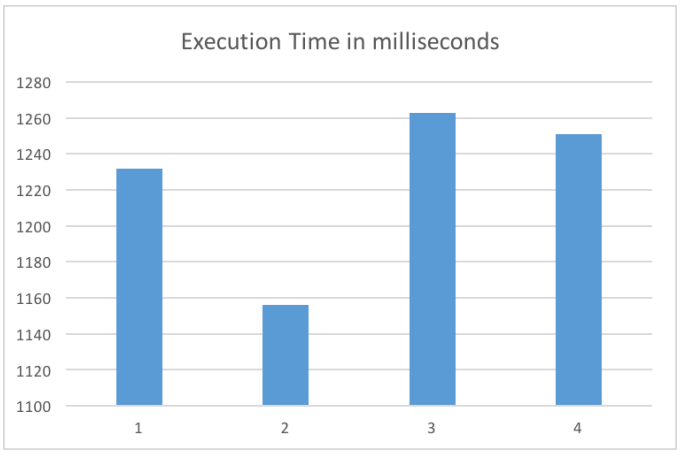


Figure 5.3: Comparison of execution time of Rome city data set with 3353 vertices and 8870 edges for different test cases on CPU

The average execution time for Rome city is 1225.5ms. We observed that range of vertices and edges that were used in Rome city gave almost similar output for every test cases.

5.2.2 New York City

For New York city we saw that there was a huge difference among the test cases. The Following table shows the output for some test cases.

Source	Destination	Execution Time (ms)
1231	1354	13869
8000	9000	226507
42212	44346	214111
28682	24343	221928

Table 5.2: CPU Execution time (ms) for New York City

Figure 5.4: Comparison of execution time of New York city data set with 94346 vertices and 129684 edges for different test cases on CPU

The average execution time for New York city data set is 169,103.75ms. Running this data set we observed that in a city map for a small set of output where the result consist of few vertices differs tremendously in execution time compare to the results that consist large set of vertices.

5.2.3 Execution time comparison between New York City and Rome

So, from that we came to a conclusion that in a dense city the execution time may differ in different situation. Moreover, comparing this output with Rome city we noticed that the execution time is more for New York city. Following graph shows the graphical representation of the comparison.

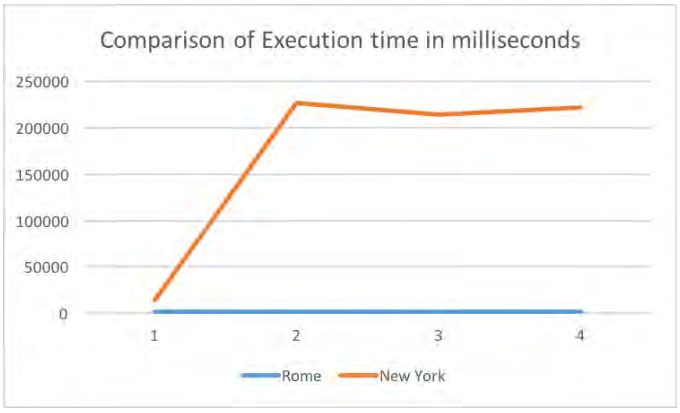


Figure 5.5: Comparison of execution time of Rome city and New York City on CPU

So, from the above comparison we saw that dense graphs needs more execution time. We calculated that New York city took almost 110 times in terms of milliseconds more than Rome

city. But for our approach we needed to have this execution done in real time. As the traffic in roads changes frequently, so for real time output CPU version algorithm were not suitable.

5.3 Running Algorithm on GPU

5.3.1 Rome

We tested the parallel version of [23] which includes our modified path generator in GeForce GTX 950(Maxwell GM 206), GeForce 920M (Kepler GK 208), GeForce GTX 760 (Kepler GK 104). Following table shows execution time for different input set of Rome city.

Source	Destination	Execution Time (ms)					
		GeForce GTX 950 (Maxwel)		GeForce GTX 760 (Kepler)		GeForce 920M (Kepler)	
62	116	26.57	26.35	16.8	16.85	24.34	23.37
		26.13		17.02		22.46	
		26.35		16.74		23.33	
54	1777	23.06	24.96	17.34	17.33	22.71	23.30
		26.62		17.28		23.66	
		25.21		17.38		23.55	
2232	1354	23.65	24.79	17.57	17.52	24.45	24.18
		27.09		17.13		24.03	
		23.63		17.86		24.08	
675	343	22.9	24.19	17.33	17.22	22.85	23.21
		26.63		17.24		23.37	
		23.05		17.09		23.43	

Table 5.3: GPU Execution time (ms) for Rome City

From the table 5.3, we saw that the average execution time for GeForce GTX 950 was 25.07ms. For GeForce GTX 760 the average execution time was 17.23ms. Finally, for GeForce 920M it was 23.52ms.

Comparing the results on different GPU, we have seen that the output for three different GPU model were different. The average execution time for GTX 950 and GeForce 920M were almost the same but GTX 760 gave a better performance than the other GPUs. Following shows the comparison of execution in different GPU models.

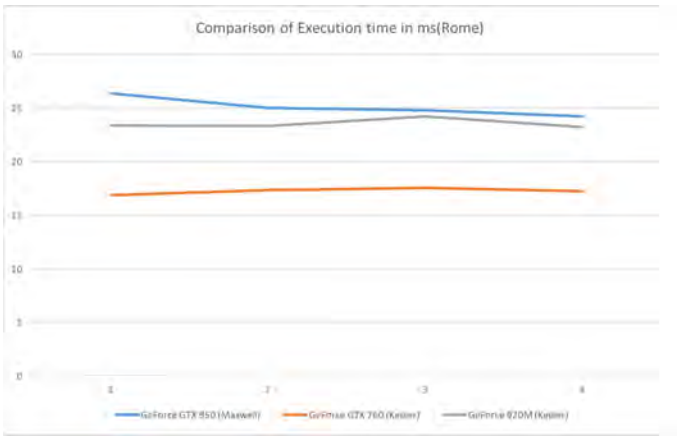


Figure 5.6: Comparison of execution time in GTX 950, GTX 760 and GeForce 920M for different test cases (Rome City)

5.3.2 New York City

Keeping the similar test cases that we used for the CPU algorithm, we executed the GPU algorithm for New York city and found the following output.

Source	Destination	Execution Time (ms)					
		GeForce GTX 950 (Maxwel)		GeForce GTX 760 (Kepler)		GeForce 920M (Kepler)	
1231	1354	62.23	60.56	33.95	33.97	182.27	183.44
		57.6		33.96		184.36	
		61.87		34.01		183.69	
8000	9000	186.67	180.47	109.77	110.01	579.22	567.37
		185.36		109.9		562.15	
		169.39		110.36		560.75	
42212	44346	221.92	210.02	129.49	129.45	664.85	665.44
		205.23		129.69		668.74	
		202.93		129.18		662.74	
28682	24343	172.34	164.28	102.2	102.24	525.76	522.97
		161.03		102.53		520.82	
		159.49		101.99		522.34	

Table 5.4: GPU Execution time (ms) for New York City

The average execution time in GTX 950 (Maxwell) is 153.83ms and in GeForce 960M

(Kepler) is 484.805ms. For GTX 760 (Kepler) it was 93.91ms. The test cases of this output were similar as the CPU test cases. We observed that the GPU has reduced the execution time remarkably. As discussed in CPU version algorithm, the execution time for this city map showed different execution time for different situations. But in GPU the differences of the execution time is comparatively less from one another.

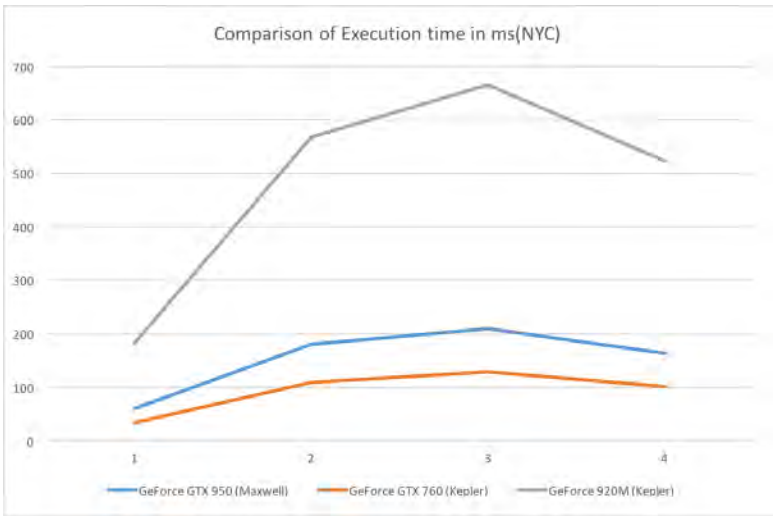
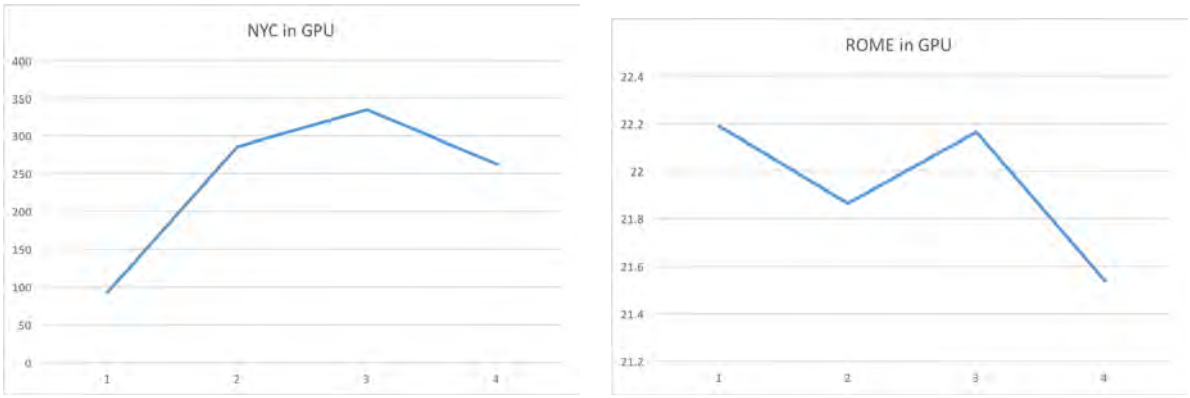


Figure 5.7: Comparison of execution time in GTX 950, GTX 760 and GeForce 920M for different test cases (New York City)

5.3.3 Comparison of Execution time between Rome and New York City in GPU

After executing all the graphs into GPU algorithms in GPU we have found the following result for New York City and Rome which execution times are represented as graph.



(a) New York City (b) Rome City

Figure 5.8: Comparison of execution time (ms) for Rome and New York city

The main reason is that NYC graph has a higher execution time becuase it contains huge number of nodes. We have shown the average execution time of 3 GPUs and among these the mobile GPU has comparatively higher execution time. Due to mobile GPU its performance is less than descrete GPUs which is yet to clarify.

5.3.4 Dhaka City

After comparing the performance of the GPU algorithm, we tested the performance of our Dhaka city map.

Source	Destination	Execution Time (ms)					
		GeForce GTX 950 (Maxwell)		GeForce GTX 760 (Kepler)		GeForce 920M (Kepler)	
80	79	1.6	1.52	1.04	1.02	1.4	1.38
		1.38		1.03		1.38	
		1.59		1.01		1.37	
53	100	1.44	1.50	1.12	1.07	1.43	1.43
		1.42		1.04		1.44	
		1.65		1.06		1.42	
1	100	1.8	1.78	1.14	1.16	1.52	1.52
		1.77		1.2		1.53	
		1.79		1.15		1.53	
7	47	1.27	1.32	0.95	0.94	1.88	1.46
		1.45		0.93		1.25	
		1.26		0.94		1.26	

Table 5.5: GPU Execution time (ms) for Dhaka City

The result that we obtained was the output with different conditions of the roads. Based on roads situation we executed the GPU algorithm and found the path. We have seen that in all the three GPU models the execution time was almost same. Among them, GeForce GTX 760 (Kepler GK 104) gave the best performance.

For a particular source and destination vertices, we tested three different situations of the roads and observed that, by changing the situations of the traffic, the route for same set of source and destination was altered but there were not much differences in execution time. The following representation shows the comparison of execution time of Dhaka City map on different GPU models.

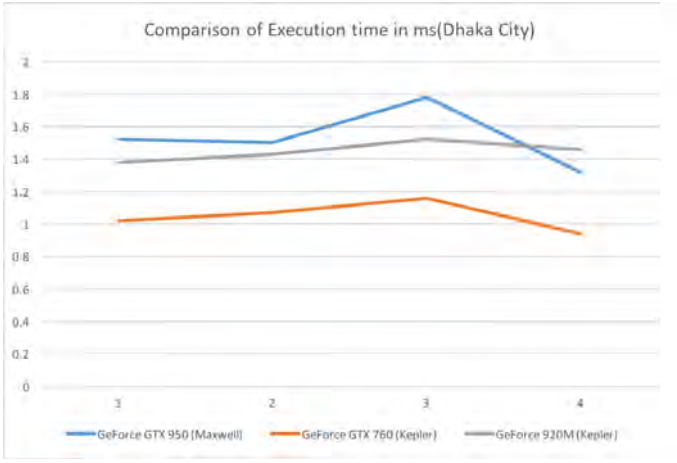


Figure 5.9: Comparison of Dhaka city map on different GPU models

5.4 Comparison between GPU and CPU

The execution time for GPU gave a better solution than the CPU. So for larger set of data, we saw that GPU parallel version is better option. And also for getting the output in real time we need a faster execution time which GPU can provide. After that, we modified the graph into a more structured city map and executed it for finding shortest path. In the following we showed the comparison of the data sets.

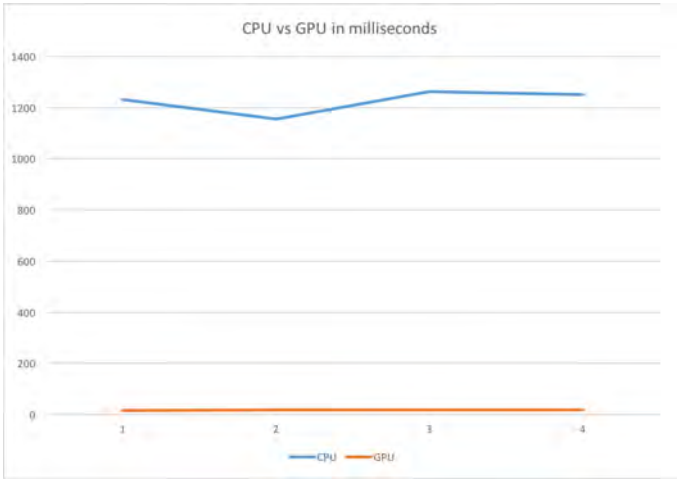


Figure 5.10: Comparison of CPU vs GPU for 3353 vertices and 8870 edges

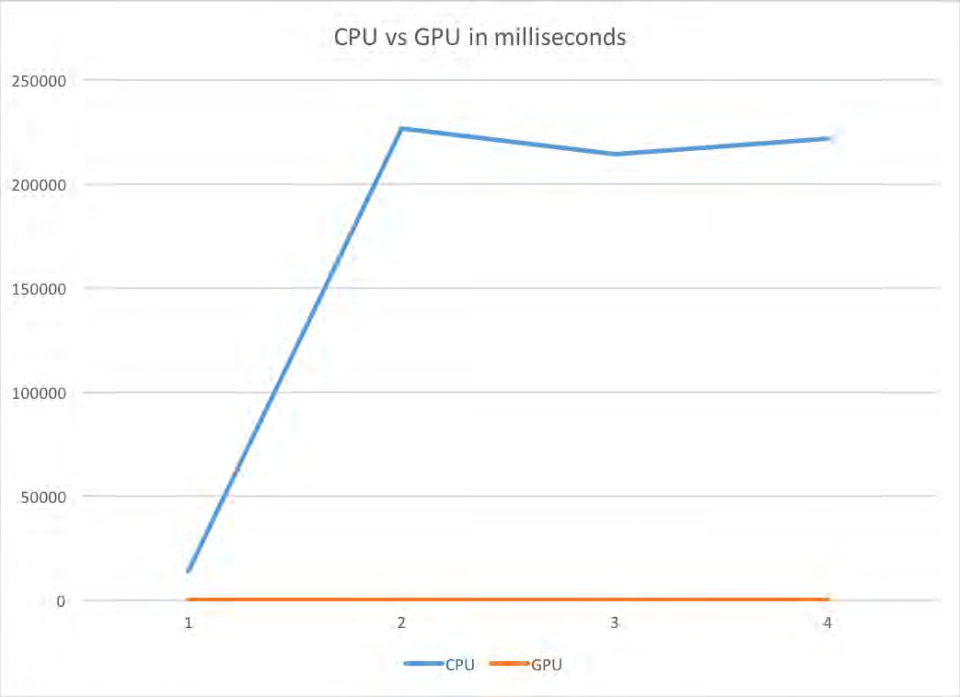


Figure 5.11: Comparison of CPU vs GPU for 94346 vertices and 129684 edges

We implemented the modified parallel version of [23] where we can obtain the path, with a graph that we discussed in Chapter 3 methodology. Considering the changes of the situation of the road with light, medium and heavy traffic we updated the city graphs status in real time.

The result that was generated for every section of source and destination with the updated traffic situation of the city graph redirected the path when the roads have heavy traffic. When a road has a shortest distance but the condition of the traffic is heavy compared to other routes, then the algorithm ignores that route and chooses the best route for the required destination.

5.5 Platform Comparison of NVIDIA

Having more vertices in the graph implies that there are more distance combinations to be compared. We observe that GPU architecture Maxwell GM 206 did not always have the best performances. The mobile GPU Kepler GK 208 gave the lowest performances compared to other architectures. For the scenario of different situations, the Kepler board obtained the best performance. As the graph size increases the meeting point for Kepler GK 208 and Maxwell decreases whereas, for dense graphs, the Maxwell board reaches closer to the best performance as compared with Kepler GK 104.

This occurs because the Kepler GK 104 has the highest number of cores than the other boards. It has 2304 cores whereas Maxwell has 768 cores and the mobile GPU Kepler GK

208 has only 384 cores. So, for most cases, the mobile GPU gave the least performance as the GPU has the lowest number of cores than the other tested boards. And Kepler GK 104 board performed well due to its larger number of cores. But in some cases for Dhaka city graph the mobile GPU gave better performance than the Maxwell as the graph was a low degree graph and the level of parallelism was lower. And also in some cases, the Maxwell board had a closer performance ratio with Kepler GK 104. This was because the base clock rate for Maxwell is 1.0 Ghz but this board was overclocked and the boost clock rate was 1.18 Ghz whereas the GK 104 board has 0.86 Ghz. It is better to use higher clock rate GPU with lower cores than with slower clock rate. But as the Kepler board has 2304 cores which is almost 3 times more than the Maxwell board, the Kepler board dominated the Maxwell board in case of performance. Finally, it is better to use GPU with many cores to achieve more parallelism for the graphs having higher degrees and high size as there can be more threads performing and increase execution time.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We started our project keeping in mind to solve one of the current issues in developing and developed countries which is traffic jam. It is not possible to build new roads or infrastructure to match the traffic over the night. Our goal was how can we provide a solution to minimize the traffic jam and take people to their destination at the shortest time possible. We had to keep in mind the result should be quick and as accurate as possible. So we designed and implemented our whole project keeping these two points in focus.

We achieved the results which is the shortest path in real time and the result is also accurate. We believed our motive to develop this project can reduce the load of traffic in a particular road and can ensure less travel time. Which can eventually tends to less fuel consumption and minimize other traffic jam related consequences.

6.2 Future Work

We have future improvement plan regarding our project. We want to take data of road from more general perspective such as social media (Facebook, Twitter etc)or traffic updates.

We have also plan to integrate our project with Open Street Map(OSM). So that any one can easily give data or if they want to get result they can see it in OSM.

Lastly, the nature of the road by examining the different situation. How much length can be added depending on the size of the road, time and vehicle on the road.

References

[1] Tosaka, "Cuda processing flow," https://commons.wikimedia.org/wiki/File:CUDA_processing_flow_(En).PNG, November 2008.

[2] K. OLAGUNJU, "Evaluating traffic congestion in developing countries–a case study of nigeria." 2015.

[3] T. T. Institute, "2011 urban mobility report," http://mobility.tamu.edu/ums/.

[4] A. Downie, "The world's worst traffic jams," http://content.time.com/time/world/article/0,8599,1733872,00.html, April 2008.

[5] S. S. Andaleeb, M. Haq, and R. I. Ahmed, "Reforming innercity bus transportation in a developing country: A passenger-driven model," 2007.

[6] M. Ben-Ari, *Principles of Concurrent and Distributed Programming*. Prentice Hall, 1990, ch. 16, p. 164.

[7] J. Martin, *Programming real-time computer systems*. Prentice-Hall Inc., 1965.

[8] K. G. Shin and P. Ramathan, "Real-time computing: a new disipline of computer science and engineering," *Proceedings of the IEEE*, vol. 82, pp. 6–24, January 1994.

[9] J. R. Haritsa, M. J. Carey, and M. Livny, "Data access scheduling in firm real-time database systems," *Real-Time Systems*, vol. 4, pp. 203–241, September 1992.

[10] K. Juvva, "Real time systems," 18-849b Dependable Embedded Systems, Carnegie Mellon University, 1998.

[11] Waze, "Waze," https://www.waze.com/, accessed: 2016–03–6.

[12] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, pp. 87–90, 1958.

[13] R. W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, p. 345, 1962.

[14] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, December 1959.

[15] T. H. Cormen, C. E. Leiserson, and C. Stein, *Introduction to Algorithms*. MIT Press, 1992.

[16] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, pp. 596–615, July 1987.

[17] M. Thorup, "Undirected single source shortest paths in linear time," *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pp. 12–21, October 1997.

[18] L. G. Aleksandrov, A. K. Maheshwari, and J. R. W. Sack, "Approximating shortest paths on weighted polyhedral surfaces," *Algorithmica*, vol. 30, pp. 527–562, 2001.

[19] K. K. and T. B. Schardl, "Parallel single-source shortest paths." MIT Computer Science and Artificial Intelligence Laboratory.

[20] H. N. Gabow, "Scaling algorithms for networks problems," *Journal Of Computer and System Sciences*, vol. 31, pp. 148–168, 1985.

[21] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, "Relaxed heaps: an alternative to fibonacci heaps with applications to parallel computation," *Communications of the ACM*, vol. 31, pp. 1343–1354, 1988.

[22] U. Meyer and P. Sanders, "δ-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.

[23] H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "Comprehensive evaluation of a new gpu-based approach to the shortest path problem," *International Journal of Parallel Programming*, vol. 43, pp. 918–938, October 2015.

[24] H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, "A new gpu-based approach to the shortest path problem," in *High performance computing and simulation (HPCS), 2013 international Conference on*. IEEE, 2013, pp. 505–511.

[25] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders, "A parallelization of dijkstra's shortest path algorithm," in *MFCS '98 Proceedings of the 23rd International Symposium on*

Mathematical Foundations of Computer Science. Springer-Verlag London, 1998, pp. 722–731.

[26] "Graphics processing unit," https://en.wikipedia.org/wiki/Graphics_processing_unit, accessed: 2016–4–13.

[27] AMD, *APU 101: All about AMD Fusion Accelerated Processing Units*, AMD.

[28] NVIDIA, "Nvidia launches the wordl's first graphiccs processing unit: Geforce 256," August 1999.

[29] A. Wiggins, N. Schultz, and P. Schmidt, "Nvidia's gpgpu architecture: Fermi and cuda," Oregon State University, Tech. Rep., 2010.

[30] NVIDIA, *NVIDIA's Next Generation CUDA Compute Architecture:Fermi*, NVIDIA Corporation, 2009.

[31] P. N. Glaskowsky, *NVIDIA's Fermi: The First Complete GPU Computing Architecture*, NVIDIA Corporation, September 2009.

[32] "Kepler(microarchitecture)," https://en.wikipedia.org/wiki/Kepler_(microarchitecture), accessed: 2016–4–14.

[33] L. Nyland and S. Jones, *Inside Kepler*, NVIDIA Corporation.

[34] J. Wang, "Introducing the geforce gtx 680 gpu," March 2012.

[35] R. Smith, *NVIDIA GeForce GTX 680 Review: Retaking The Performance Crown*, March 2012.

[36] ——, *NVIDIA Launches Tesla K20 & K20X: GK110 Arrives At Last*, November 2012.

[37] NVIDIA, *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, NVIDIA, 2012.

[38] ——, "Cuda parallel computing platform," http://www.nvidia.com/object/cuda_home_new.html.

[39] F. Abi-Chahla, "Nvidia's cuda: The end of the cpu?" June 2008.

[40] NVIDIA, "Cuda toolkit," https://developer.nvidia.com/cuda-toolkit.

[41] ——, "Cuda llvm compiler," https://developer.nvidia.com/cuda-llvm-compiler.

[42] "Cuda," https://en.wikipedia.org/wiki/CUDA.

[43] NVIDIA, "What is gpu accelerated computing?" http://www.nvidia.com/object/what-is-gpu-computing.html.

[44] ——, *CUDA C Programming Guide*, NVIDIA, September 2015.

[45] P. J. Martín, R. Torres, and A. Gavilanes, "Cuda solutions for the sssp problem," in *International Conference on Computational Science*. Springer, 2009, pp. 904–913.