New Approach of Bellman Ford Algorithm on GPU using Compute Unified Design Architecture (CUDA)

Pankhari Agarwal
Department of Computer Science and Engineering
National Institute of Technical Teachers Training & Research
Chandigarh, India

Maitreyee Dutta
Department of Computer Science and Engineering
National Institute of Technical Teachers Training & Research
Chandigarh, India

ABSTRACT
Large graphs involving millions of vertices are common in many practical applications and are challenging to process. To process them we present a fundamental single source shortest path (SSSP) algorithm i.e. Bellman Ford algorithm. Bellman Ford algorithm is a well known method of SSSP calculation and which is considered to be an optimization problem in the graph. SSSP problem is a major problem in the graph theory that has various applications in real world that demand execution of these algorithms in large graphs having millions of edges and sequential implementation of these algorithms takes large amount of time. In this paper, we investigates some methods which aim at parallelizing Bellman Ford Algorithm and to implement some extended or enhanced versions of this algorithm over GPU. GPU provides an application programming interface named as CUDA. CUDA is a general purpose parallel programming architecture which was introduced by Nvidia. This algorithm can reduce the execution time up to half of the basic Bellman Ford algorithm.

Keywords
SSSP (Single Source Shortest Path) Problem, Graphics Processing Units (GPUs), CUDA (Compute Unified Device Architecture).

1. INTRODUCTION
Graphs are the commonly used data structures that describe a set of objects as nodes and the connections between them as edges. A large number of graph operations are present, such as minimum spanning tree, breadth-first search, shortest path etc., having applications in different problem domains like VLSI chip layout [1], phylogeny reconstruction [2] , data mining, and network analysis[3].

With the development of computer and information technology, researches on graph algorithms get wide attention. In particular, the Single Source Shortest Path (SSSP) problem is a major problem in graph theory which computes the weight of the shortest path from a source vertex to all other vertices in a weighted directed graph.

Computing SSSP on a parallel computer may serve two purposes: solving the problem faster than on a sequential machine and/or taking advantage of the aggregated memory in order to avoid slow external memory computing. However, some modern applications, such as data mining, network organization, etc. require large graphs with millions of vertices, and some of the previous SSSP algorithms become impractical, when we do not have a very expensive hardware at our disposal. Fortunately, Graphics Processing Units (GPUs) supply a high parallel computation power at a low price. GPUs have recently become popular as general computing devices due to their relatively low costs, massively parallel architectures, and improving accessibility provided by programming environments such as the Nvidia CUDA framework [4].

At present, the serial graph algorithms have reached the time limitation as they used to take a large amount of time. Therefore, the parallel computation is an efficient way to improve the performance by applying some constraints on the data and taking the advantage of the hardware available currently. Different implementations of parallel algorithms for the SSSP problem are reviewed in [5]. Bader et al. [6], [7] use CRAY supercomputer to perform BFS and single pair shortest path on very large graphs. A. Crauser et al. [8] have given a PRAM implementation of Dijkstra's algorithm while such methods are fast; hardware used in them is very expensive. N. Jasika et al. [9] presented a parallel dijkstra's algorithm using OpenMP (Open Multi-Processing) and OpenCL (Open Computing Language) which gives good results over serial algorithm. Pedro J. Martín et al. [10] have given an efficient parallel dijkstra's algorithm on GPU using CUDA. L. Luo et al. [11] have given a GPU implementation of BFS which gives around 10X speed-up over the algorithm given by P. Harish et al. [12].

In this work we presented two parallel implementations of Bellman Ford algorithm on GPU using CUDA. We will run our algorithms on various large graphs represented by adjacency lists and expect to achieve high speedups regarding the basic parallel Bellman Ford algorithm.

The rest of the paper is organized as follows: CUDA basics along with GPU architecture is discussed in Section 2. Graph representation used by our implementation is discussed in Section 3. Section 4 present parallel implementations of Bellman Ford algorithm on GPU using CUDA. Performance analysis of our implementation on various types of graphs is done in section 5 and finally concluded in section 6.

2. CUDA PROGRAMMING MODEL
In CUDA programming model there are a large number of threads and these threads collects to form a warp. A warp is a collection of threads that are used to work in parallel on a multi-core architecture. In the same manner a block is the collection of threads that run over a multiprocessor in a given time, in this way multiple blocks can run on a single multiprocessor in a timeshared manner. Such multiple blocks are collected in the form of a grid. Each thread in a block and each block in a grid has given its own unique ID. Kernel is a part of code which defines the task in a parallel program and is executed on each thread by using the thread ID. As in GPU, multiprocessor is working in SIMD manner so each thread runs the same kernel over different data as all the device memory can be easily accessible to any thread in the block and in this way the use of shared memory improves the performance of any computation [4],[12][14].

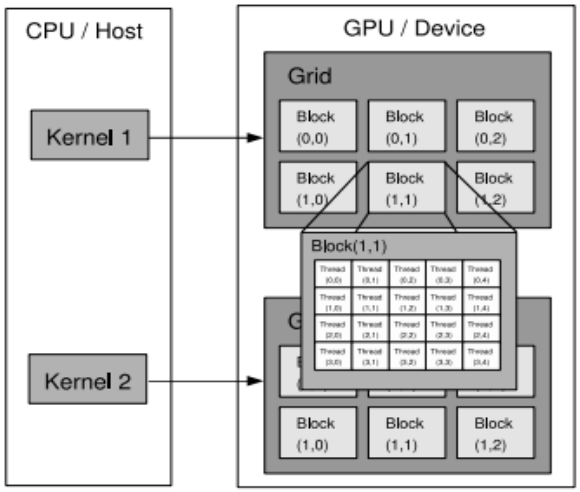

Figure 1: CUDA programming model

3. GRAPH REPRESENTATION ON CUDA

A graph G (V, E) is commonly represented as an adjacency matrix. For sparse graphs such are presentation wastes a lot of space. Adjacency list is a more compact representation for graphs. Because of variable size of edge lists per vertex, its GPU representation Accelerating Large Graph Algorithms on the GPU Using CUDA 201 may not be efficient under the GPGPU model. CUDA allows arrays of arbitrary sizes to be created and hence can represent graph using adjacency lists. We represent graphs in compact adjacency list form, with adjacency lists packed into a single large array. Each vertex points to the starting position of its own adjacency list in this large array of edges. Vertices of graph G (V, E) are represented as an array Va. Another array Ea of adjacency lists stores the edges with edges of vertex i+1 immediately following the edges of vertex i for all i in V. Each entry in the vertex array Va corresponds to the starting index of its adjacency list in the edge array Ea. Each entry of the edge array Ea refers to a vertex in vertex array Va (Figure 2)[14].

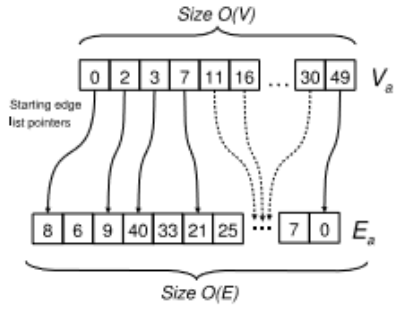

Figure 2: Graph representation with vertex list pointing to a packed edge list [12][15].

4. BELLMAN- FORD ALGORITHM

Bellman Ford algorithm [11] is a SSSP finding algorithm that calculates shortest paths from a single source vertex to all of the other vertices in a weighted directed graph. It uses the relax method in which the approximate distance to each vertex is always greater than or equal to the true distance, and is replaced by the minimum of its newly calculated value and old value. In this algorithm all the edges are relaxed for |V| - 1 times, where |V| is the number of vertices in the graph. In this way each node will get its shortest distance from the source node. This algorithm usually takes a large amount of time and therefore for faster results we can perform some modifications and parallelization in it.

In this section we are going to present our implementations of parallel Bellman Ford Algorithms. We have implemented a basic parallel bellman ford Algorithm and its enhanced versions on GPU using CUDA here.

4.1 Basic Parallel Bellman Ford Algorithm

In this algorithm basic Bellman Ford algorithm is implemented using CUDA. In this implementation, number of threads is equal to the number of edges in the graph and these edges will relax in parallel for |V| number of iterations. Initially the cost of all vertices is initialised to 'infinity' except the start node whose cost is initialised to '0'. Basic BF algorithm is described in Algorithm1.

Algorithm 1: BF (G (V, E, W), S)

Create edge_strt_node(Sa), edge_end_node (Ea), edge_weight (Wa) and node_weight (Na) from G (V, E, W)

[1] for each vertex V in parallel do

[2] Invoke INITNODEWEIGHT (Na, S)

[3] end for

[4] for i = 0 toV-1 do

[5] for each edge E in parallel do

[6] Invoke RELAX (Na, Sa, Ea, Wa)

[7] end for

[8] end for

Basic Bellman Ford algorithm consists of two kernels as a part of its implementation.

In the first kernel INITNODEWEIGHT (Ns, S) as shown in Algorithm 2, the node weight of each node except the source node is initialised to 'infinity', and for the source node it is initialised to '0'. In the second kernel RELAX (Na, Sa, Ea, Wa) as shown in Algorithm 3; the cost of each neighbour is updated if it is greater than the cost of current vertex plus the edge weight to that neighbour. This updating is done in an atomic manner using the atominMin function of CUDA in order to avoid the concurrent read/write conflicts.

Algorithm 2: INITNODEWEIGHT (Na, S)

[1] id = blockIdx.x*blockDim.x+threadIdx.x

[2] Na[id] = ∞;

[3] if(id == S)

[4] Na[id] = 0;

Algorithm 3: RELAX (Na, Sa, Ea, Wa)

[1] id = blockIdx.x*blockDim.x+threadIdx.x

[2] if Na[Ea[id]]> Na[Sa[id]]+Wa[id]

[3] begin atomic

[4] Na[Ea[id]] ← Na[Sa[id]]+Wa[id]

[5] end atomic

[6] end if

Basic Bellman Ford algorithm tried to relax all the edges in each iteration. So, this algorithm is taking a large amount of time and this computation time can be reduced if we apply some conditions on it. Based on this we are going to present an algorithm named as Parallel Bellman Ford algorithm using two Flags.

4.2 Parallel Bellman Ford Algorithm using Two Flags

In this algorithm we are using two flags F1 and F2 in order to find those edges which should be relaxed in the next iteration. To implement this algorithm we have taken the number of threads equal to the number of edges in graph and they should be relaxed |V| times, but, in each iteration we will relax only those edges whose source node weight was updated in the last iteration. In this way we are going to reduce the computation time by a large factor because in each iteration, node weight of only some nodes needs to be updated. BF algorithm using two flags is described in Algorithm 4.

Algorithm 4: BF_2FLAGS (G (V, E, W), S)

Create edge_strt_node(Sa), edge_end_node (Ea), edge_weight (Wa) , node_weight (Na) from G (V, E, W)

Create Flag (F1) and Flag (F2)

[1] for each vertex v in parallel do

[2] Invoke INITNODEWEIGHT (Na, S, F1, F2)

[3] end for

[4] for i=0 toV-1 do

[5] for each edge E in parallel do

[6] Invoke RELAX (Na, Sa, Ea, Wa, F1, F2)

[7] Invoke COPYFLAG (F1, F2)

[8] end for

[9] end for

BF algorithm using two flags consists of three kernels in its implementation. In the first kernel INITNODEWEIGHT (Na, S, F1, F2) as shown in Algorithm 5, node weight of each node is initialised in parallel as in Basic BF algorithm and in addition to this both flags are also initialised to '0' in parallel for each node except the source node for which flag F1 is initialised to 1. In the second kernel RELAX (Na, Sa, Ea, Wa, F1, F2) as shown in Algorithm 6; the edge relaxation is done in the same manner as in Basic BF algorithm but, here in each iteration only those edges should relax whose source node was updated in the last iteration. In order to find such nodes we are using flag variables for each node such that only those edges will get relaxed whose source node's flag variable is '1'.

Algorithm 5: INITNODEWEIGHT (Na, S, F1, F2)

[1] id = blockIdx.x*blockDim.x+threadIdx.x;

[2] Na[id] = ∞;

[3] F1[id] = 0;

[4] F2[id] = 0;

[5] if(id == S)

[6] Na[id] = 0;

[7] F1[id] = 1;

[8] end if

Algorithm 6: RELAX (Na, Sa, Ea, Wa, F1, F2)

[1] id = blockIdx.x*blockDim.x+threadIdx.x;

[2] if(F1[Sa[id]] == 1)

[3] if (Na[Ea[id]] > Na[Sa[id]]+Wa[id])

[4] begin atomic

[5] Na[Ea[id]] = Na[Sa[id]]+Wa[id];

[6] end atomic

[7] F2[Ea[id]] = 1;

[8] end if

[9] end if

In the third kernel COPYFLAG (F1, F2) as shown in Algorithm 7, the current status of the flag value is copied from one flag to another in order to find the next nodes whose outgoing edges should be relaxed.

Algorithm 7: COPYFLAG (F1, F2)

[1] id = blockIdx.x*blockDim.x+threadIdx.x;

[2] F1[id] = F2[id];

[3] F2[id] = 0;

BF algorithm using two flags gives far better results over the Basic BF Algorithm.

5. PERFORMANCE ANALYSIS

We have evaluated the performance of parallel Bellman Ford algorithms on a large range of graph statistical data such as sparse, general and dense directed graphs of 8K to 62K vertices having up to .15M edges. We will compare the execution times of these algorithms.

5.1 Comparison of Basic Parallel Bellman Ford algorithm and Parallel Bellman Ford Algorithm using two Flags

Here we show the comparison between the Basic BF algorithm (basic BF) and BF algorithm using two flags (BF_copy). BF_copy gives a good speed-up over the basic_BF algorithm on the graph having large number of edges. The experimental result is shown in table 5.1 & table 5.2 and comparison is shown in figure 5.1& figure 5.2 below.

5.1.1 Experimental Setup 1

5.1.1.1 Experimental Environment

Processor: Intel i5 processor @ 3.20 GHz

RAM: 2 GB

OS: windows 7

Language: visual C++ runs on visual studios 2008

Graphics card: NVIDIA's GeForce GTS 450. (192 cores), compute capability 2.1

Language (parallel Implementation): CUDA.

5.1.1.2 Results

The amount of time taken by both algorithms on various graphs is given in the table 5.1.

Table 1.Basic Bf and BF_copy Experimental Results (1)

No. Of edges	Time in ms	
	basic BF	BF_copy
20K	752	150
30K	1310	300
40K	1915	465
55K	4100	1500
88K	11283	1690
147K	25992	11180

Graphical analysis of the comparison of both algorithms is shown in figure 3 below.

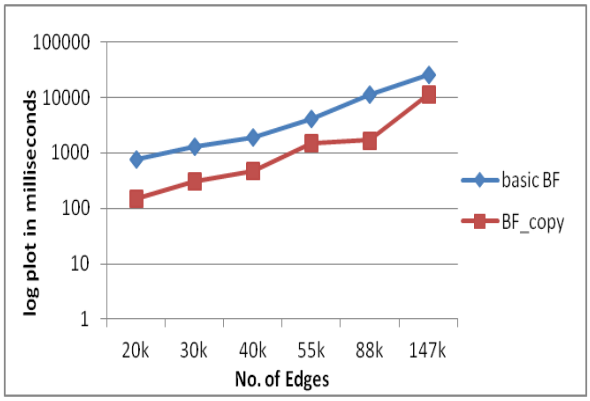

Figure 3: SSSP timings for basic_BF and BF_copy (1)

5.1.2 Experimental setup 2

5.1.2.1 Experimental Environment

Processor: Intel(R) Xeon(R) E5-2650 @ 2.00 GHz

RAM: 24 GB

OS: windows 7

Language: visual C++ runs on visual studios 2010

Graphics card: Tesla C2075 (448 cores), compute capability 2.0

Language (parallel Implementation): CUDA.

5.1.2.1 Results

The amount of time taken by both algorithms on various graphs is given in the table 5.2

Table 2. Basic Bf and BF_copy Experimental Results (2)

No. Of edges	Time in milliseconds	
	Basic BF	BF_copy
20K	440	75.41
30K	911.82	164.53
40K	1320.9	230.35
55K	2480.5	775.37
88K	6624.64	1831.9
147K	15921.2	5519.4

Graphical analysis of the comparison of both algorithms is shown in figure 4 below.

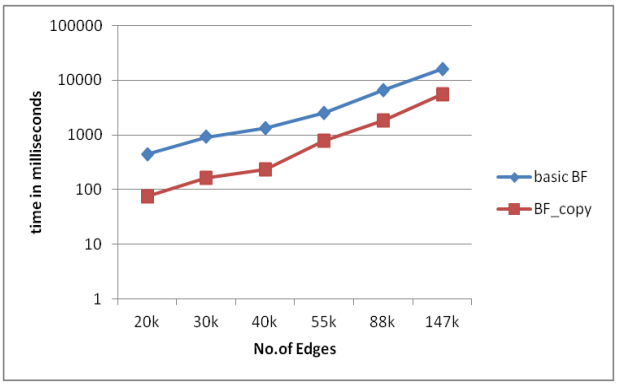

Figure 4: SSSP timings for basic_BF and BF_copy (2)

6. CONCLUSION

In this paper, we presented modified Bellman Ford algorithms which tried to minimize the number of edge relaxations in order to reduce the useless calculations. As number of edges is greater than number of nodes and threads are mapped to edges rather than nodes, greater degree of parallelism can be achieved. Achieved results show a considerable speed-up over corresponding serial implementation for various graph instances having varying degrees. The size of the device memory limits the size of the graphs handled on a single GPU. This algorithm can be extended for larger size graphs by using multiple GPUs in parallel. Usage of multiple GPUs in parallel will require a study on partitioning the problem and handling of data.

7. REFERENCES

[1] Ashok Jagannathan. Applications of Shortest Path Algorithms to VLSI Chip Layout Problems. Thesis Report. University of Illinois. Chicago. 2000.

[2] Karla Vittori, Alexandre C.B. Delbem and Sérgio L. Pereira. Ant-Based Phylogenetic Reconstruction (ABPR): A new distance algorithm for phylogenetic estimation based on ant colony optimization. Genetics and Molecular Biolog., 31(4), 2008.

[3] Jiyi Zhang, Wen Luo, Linwang Yuan, Weichang Mei. Shortest path algorithm in GIS network analysis based on Clifford algebra. Transactions of the IRE Professional Group. 18(11, 12).

[4] A. Bustamam, G. Ardaneswari, D.Lestari, "Implementation of CUDA GPU-based parallel computing on Smith-Waterman algorithm to sequence database searches", International Conference on Advanced Computer Science and Information Systems (ICACSIS), IEEE, pp. 137-142, 2013.

[5] Meyer U., Sanders P. 2003. Δ -stepping: a parallelizable shortest path algorithm. J. of Algorithms 49, pp. 114–152.

[6] Bader D.A., Madduri K. 2006. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. ICPP, pp. 523–530.

[7] Bader D.A., Madduri K. 2006. Parallel algorithms for evaluating centrality indices in real-world networks. ICPP 2006. Proceedings of the 2006 International Conference on Parallel Processing, IEEE Computer Society Press, Los Alamitos , pp. 539–550.

[8] Crauser A., Mehlhorn K., Meyer U., and Sanders P. 1998. A Parallelization of Dijkstra's Shortest Path Algorithm. MFCS'98- LNCS 1450, Lubos Prim et al. (Eds.), Springer-Verlag Berlin Heidelberg, pp. 722-731.

[9] Jasika N., Alispahic N., Elma A., Ilvana K., Elma L. and Nosovic N. 2012. Dijkstra's shortest path algorithm serial and parallel execution performance analysis. MIPRO, pp. 1811-1815.

[10] Martín P. J., Torres R., and Gavilanes A. 2009. CUDA Solutions for the SSSP Problem. ICCS 2009, Part I, LNCS 5544, G. Allen et al. (Eds.), Springer-Verlag Berlin Heidelberg , pp. 904-913.

[11] Harish P. and Narayanan P. J. 2007. Accelerating large graph algorithms on the GPU using CUDA, IEEE High Performance Computing, pp. 197-208.

[12] Nvidia CUDA: http://www.nvidia.com/cuda

[13] Srinivas A., Manish P., Ramamurthy B and Viktor K.P. 2007. High Performance Computing - HiPC 2007: 14th International Conference, Goa, pp. 199-205.

[14] Frederico L. Cabral, Carla Osthoff, Rafael Nardes, Daniel Ramos1 June 2014. Massive Parallelism With Gpus For Centrality Ranking In Complex Networks in International Journal Of Computer Science & Information Technology (IJCSIT) Vol 6,No3

[15] Gunjan S., Amrita T., Dhirendra P. S., "New Approach for Graph Algorithms on GPU Using CUDA" submitted to journal "IJCA".

[16] Luo L. And Wong M., Hwu W. 2010An Effective GPU Implementation of Breadth-First Search, DAC'10, ACM, pp. 52-55.