International Journal of Applied Engineering Research ISSN 0973-4562 Volume 11, Number 4 (2016) pp 2560-2567

Efficient Parallel Implementation of Single Source Shortest Path Algorithm on GPU Using CUDA

Dhirendra Pratap Singh
Dept. of Computer Science and Engineering, MANIT, Bhopal 462003, India.

Nilay Khare
Dept. of Computer Science and Engineering, MANIT, Bhopal 462003, India.

Akhtar Rasool
Dept. of Computer Science and Engineering, MANIT, Bhopal 462003, India.

Abstract
In today's world there are number of applications like routing in telephone networks, traveller information system, robotic path selection etc., where data can be represented as a graph and different graph algorithms are executed on it to fulfil the requirements of the application. Data related to these applications are growing every day, but we still need quick and real time responses from them, as performance is a critical limiting factor. Parallel implementation is the most widely used method of performance improvement for most algorithms. This paper proposes two improved and more efficient versions constraint based single source shortest path (SSSP) calculation algorithm for Graphics Processing Unit (GPU) based machine using CUDA. First implementation creates one CUDA thread for each node and second creates one CUDA thread for each edge of graph. Analysis of inconsistencies present in both implementations and their effects are discussed in detail. Results of proposed implementations are compared with previously implemented constraint based parallel Bellman Ford SSSP algorithm on a GPU, as it shows best result among all previous parallel GPU based SSSP algorithm. Nvidia's Tesla C2075 and GeForce GTS 450 GPUs are used to run the parallel implementation of evaluated algorithms. We obtain a 600-fold speed increase in proposed implementation compared to a simple parallel Bellman Ford algorithm and 2. 6 times performance gain over the previously implemented constraint based parallel SSSP algorithm on a GPU.

Keywords: Parallel Computing, Graph Algorithm, Single source shortest path algorithm, GPU Computing, Compute Unified Device Architecture (CUDA).

Introduction

To extract information from many real world field such as road networks [1, 2], computer networks [3, 4], VLSI design [5, 6], and robotics [7] their data are represented in the form of graphs and different graph algorithms are applied to them. SSSP calculation is one of the algorithms most frequently used on these real world graph data. To define the graph algorithms we need to present some basic terminologies used by them. Graphs are represented in the form of order pair $G(V, E)$, where V is the set of vertices (nodes) and E is the set of edges. This research uses weighted directed graphs with positive edge weights. Algorithms which calculate single source shortest path are the main focus of this paper.

Shortest path algorithms are usually based on iterative labelling methods. Suppose for a graph $G=(V, E)$, node $s \in V$ is selected as source node for SSSP calculation. Then algorithm assigns an initial weight to each node $v \in V$ to infinity, except source node which is zero. This distance of a node from the source node is called *node weight*. At any step of algorithm the value of *node weight* depends on any shorter path between s and $v \in V$ found till now. Term *edge relaxation* is used to update the weight of end node of any edge. In *edge relaxation* process end node of an edge is assigned a weight which is minimal from its current weight or the sum of the *node weight* of the edge start node and the edge weight. The algorithm repeatedly selects some nodes and applies relaxation procedure on their outgoing edges. Based on the method of selection of the next node to be scanned in these labelling methods SSSP algorithms are broadly classified in two major classes named as Label-setting algorithms and Label-correcting algorithms.

Label setting algorithms assign a permanent *node weight* to a node $v \in V$ and relax the outgoing edges of that node, it repeats this processes until each node gets its minimum weight. These methods set the distance label of at least one node as permanent (setting) in one step of the algorithm and any of the edges is relaxed just once

The label correcting algorithms generally discover shorter paths from the source node s to every other node $v \in V$ in a progressive manner. As the algorithm progresses, if a path from the source to node $v \in V$ is discovered and whose distance is smaller than *node weight* of v, then *node weight* of v will get corrected to the smaller value. This process of nodes label correction goes on until each node obtains its shortest path from source node. The label correcting methods consider all *node weight* as temporary until the end of the algorithm.

These algorithms may have to re-compute some nodes weight from source node several times until their weight eventually becomes permanent.

Dijkstra's [8] and Bellman Ford [9, 10] are two well-known serial SSSP calculation algorithms. Dijkstra is a label setting algorithm and Bellman Ford is label correcting algorithm.

As we know that real world applications need to provide a quick response to any request, so many parallel implementations of the SSSP algorithms are proposed, which have been implemented on different machines such as PRAM and CRAY supercomputers, or GPU-based machines, and tested on different types of graph data set. Crauser et al. [11] have presented some modified versions of Dijkstra's algorithm, which were defined for PRAM. Brodal et al. [12] have used the parallel priority queue data structure to perform the operations of Dijkstra's [8] algorithm in constant time on PRAM machine. Crobak et al. [13] have defined multiple bucket based method to find the nodes whose outgoing edges should be relaxed at each step of Dijkstra's [8] algorithm and shown the parallel relaxation of all outgoing edges of selected nodes. Tang et al. [14] have defined a graph partitioning based algorithm, which runs the SSSP calculation for each partition on different machines to achieve parallelism. Fetterer et al. [15] have also partitioned the graph into multiple sub-graphs and presented it on different layers. Layer one has sub-graphs, and layer two shows boundary nodes of sub-graphs. It run a parallel SSSP on each sub-graph first, and then calculates the final SSSP on a boundary graph.

With the help of Nvidia's CUDA tool it is possible to explore the highly parallel and multithreaded environment of its today's GPUs for general purpose computing. Today's GPU has provided us a low cost and highly parallel platform for general purpose computing [16-19], so in last few years GPU has been used for shortest path calculation. Harish et al. [20] have provided the first parallel SSSP calculation method for GPU-based machines using CUDA for positive edge weighted graph. It uses two CUDA kernels to implement it and creates one thread for each node. Kumar et al. [21] have implemented a parallel Bellman-Ford algorithm [9, 10] on a GPU, which can accept negative weighted edges as well. Martín et al. [22] have shown different ways for the parallel implementation of Dijkstra's algorithm [8] on a GPU-based machine. Dashora et al. [23] have presented parallel SSSP and other graph algorithms on GPU. Singh et al. [24] have proposed two different parallel implementations of modified Dijkstra's algorithm [11] for GPU based machine. Many all pair shortest path [25-28] implementations also have been proposed for GPU based machines. This paper describes two more efficient and consistent implementations of parallel SSSP algorithm for a GPU-based machine using CUDA.

CUDA Programming Model

Today's GPUs act as multicore, highly parallel coprocessors of CPUs [29], to perform compute intensive data parallel jobs (i. e. the same set of instructions is executed for different data items in parallel) at high speed and at very low cost. Many real world applications can process their data in a data parallel manner, so the GPU provides an ideal platform for these applications to speed up their operations. NVIDIA has created CUDA (Compute Unified Device Architecture), a programming interface to use NVIDA's GPUs for general purpose computing, which is an extension of C with some restrictions [30]. To understand how CUDA helps to use a NVIDIA's GPU as a parallel coprocessor of a CPU, its memory and programming model is presented here. NVIDIA's GPU is a collection of one or more multiprocessors (MP), where each MP has a set of processing elements (PE) as shown in Figure 2. There are multiple levels of memories available to a GPU's PEs. Each PE has its own private and fast register memory, and the shared memory accessible by all PEs inside a MP, and the GPU's global and constant memory are accessible to all PEs present in the GPU.

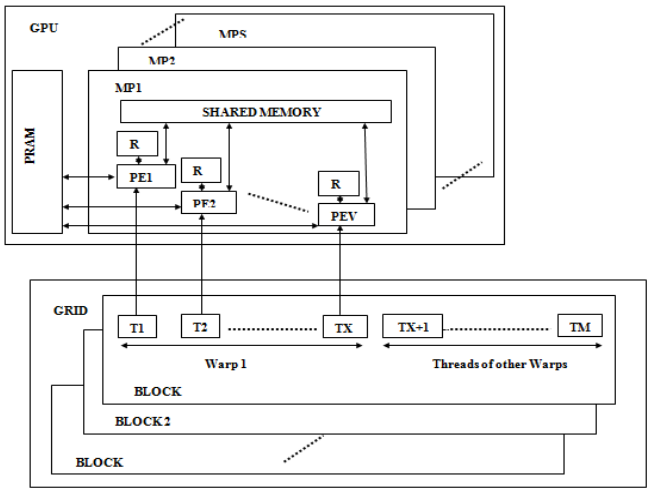

Figure 1: CUDA programming model

CUDA programming achieves parallelism (i.e. Single Instruction Multiple Threads (SIMT)[29] processing capabilities) by creating multiple threads which execute the same set of instructions on different data items. It defines a function called a kernel, which contains the set of instructions executed by each thread. These threads are logically grouped in one, two or three dimensional way to create a block, where they get a unique identification number for each dimension. These blocks are further managed in one, two or three dimensional way inside the grid, where each block gets a unique block identification number for each dimension. A block contains a set of threads which are assigned to a, MP's PEs, so these threads will be able to access the shared memory. Out of all the threads in a block only a 32 threads get the processing elements of MP at a given time. This number is called the warp size of the GPU. If the number of blocks is greater than the number of MPs in a GPU then more than one block can assigned to a MP which depends on the memory availability in SM.

Graph Representations

Two different graph representations are used for defined algorithm and its proposed implementations. First implementation uses the adjacency list representation of the graph, similar to the graph representation proposed by Harish

et al. [20]. This graph representation stores the adjacency lists of all the nodes into a single large array. of edge (*Ea*) of size *M* and array of edge weight (*Wa*) of size *M*. Each index of array *Va* represents a node number of the graph and array *Ea* stores the end node number of all the edges in the graph. The array *Ea* stores the end nodes number of the all outgoing edges of a node in sequential order. The value at any index of the array *Va* is the start index (from the *Ea* array) of the end nodes entries of the node represented by this index of *Va*. Array *Wa* stores the edge weight of all edges in the graph. This graph representation for a graph having *N* nodes and *M* edges is depicted in figure 1. For second implementation graph is represented in the form of three arrays; array (*Sa*) which stores the start node of each edge, array (*Ea*) which stores the end node of each edge and array (*Wa*), which stores the weight of each edge, all of size |*E*|. It also requires the out-degree of each node which is stored in array (*Da*) of size |*V*|.

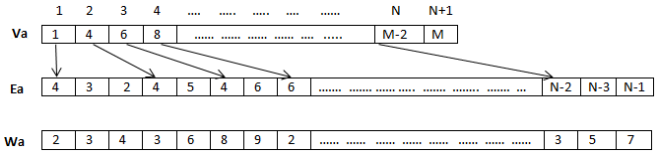

Figure 2: Graph representation for first proposed implementation

Improved Constraint based Parallel SSSP Algorithm

One of the simplest ways to calculate SSSP in a weighted directed graph is initializing the source node weight to zero and all other nodes weight to infinity, and then relaxing every edge of the graph |*V*|*-1* time, where |*V*| is number of nodes in the graph. Finally all nodes get their shortest distance from the source node [9, 10]. This algorithm generally takes very long time for large, real world graphs as it has to do some additional calculations like relaxation of all edges in any iteration, and sometimes after settlement of all nodes it processes them with no alteration of the results. For quick results it can be implemented with some modifications and parallelization. *Algorithm 1* has added two conditions in it; first condition relaxes only those edges in any iteration whose start node weight was modified in the last iteration, and second condition relaxes the edges until there is a weight change for at least one node in the last iteration. Proposed implementation of *Algorithm 1* explains how consistency and performance of this algorithm can be improved.

This constraint based parallel SSSP algorithm is shown in *Algorithm 1*, uses a flag array '*Flag*' of size |*V*| to fulfil the requirements of the first condition, and a lock variable '*Loop*' to put the check for the second condition. It is having a node weight array (*NW*) of size |*V*| to store the distance of nodes from the source node. At first step it initializes all node weight to infinity and their *Flag* value to zero in parallel. In second step it makes the source node weight to zero and sets its *Flag* value to one. In step 3 in parallel it checks the *Flag* values corresponding to each node and if the *Flag* value corresponding to any node is one then it relaxes all the out-going edges of this node. The edge relaxation operation should be atomic as in parallel environment it is possible that multiple processor could try to update the weight of a single node. During any edge relaxation if its end node weight is updated then it makes the *Loop* value to one and sets the end node *Flag* value. *Algorithm 1* repeats its step 3 until the *Loop* is set in last iteration.

Algorithm 1: Constraint based Parallel SSSP Algorithm (Graph G (V, E), Source node)

Create an array NW of size |*V*|, a variable INFINITY with a very large number assigned to it a variable Loop, and an array Flag of size |*V*|.

Begin
Step 1: for all node *n* **do in parallel**
NW[*n*] = INFINITY
Flag[*n*] = 0
End for
Step 2: NW[Source node] = 0, Flag[Source node] =1
Loop =1
Step 3: while Loop do
3.1: for all $v \in V$ **do in parallel**
Loop=0
if Flag[*v*]=1 **then**
for all $(v, u) \in E$ **do**
if NW[*u*] > NW[*v*] + Wa(*v, u*) **then**
NW[*u*] = NW[*v*] + Wa(*v, u*)
Flag[*u*] =1
Loop = 1
end if
end for
end if
end for
end while
end

Complexity Analysis

For a graph *G=(V, E)*, where |*V*|= *n* represents the number of nodes and |*E*|= *m* represents the number of edges. Let *n* processors be available in machine one corresponding to each node of the graph.

i. *Best Case*: In step1, each processor takes constant time to initialize the one node weight. Step 2 takes constant time to set the source node *Flag* value. For Step 3 two best case conditions are possible here. In first condition step 3 iterates *O(n)* time and at any iteration step 3.1 takes constant time to relax one outgoing edge with *n* processors. In second condition step 3 iterates constant time and at any iteration step 3.1 takes *O(n)* time to relax all possible outgoing edge of source node with *n* processors. So the best case time complexity of algorithm is *O(n)*.
ii. *Worst Case*: Step1 and step 2 take constant time similar to the best case. Step 3 iterates *O(n)* time and at any iteration step 3.1 takes maximum *O(n)* time to relax all possible outgoing edges of any node with *n* processors. So the worst case time complexity of the algorithm is $O(n^2)$.

Parallel Implementation of SSSP Algorithm on GPU

Simple parallel implementation of Bellman Ford algorithm for a GPU-based machine using CUDA can be done just by creating a kernel to relax the all out-going edges of a node, and calling this kernel $|V|-1$ times, and each time $|V|$ threads have to be created, one for each node of the graph. This implementation has to do some additional calculations like relaxation of all edges in any iteration, and sometimes after settlement of all vertices it processes them with no alteration of the results. Harish et al. [20] have presented an improve SSSP algorithm by adding two conditions; first relax only those edges in any iteration whose source node any iteration, and sometimes after settlement of all vertices it processes them with no alteration of the results. Their implementation is having write-write inconsistency [22] and some overhead operations. This paper presents two more efficient and consistent implementations of constraint based parallel SSSP algorithm proposed by Harish et al. [20].

These efficient and consistent parallel implementations SSSP algorithm are defined in *Algorithm 2* and *Algorithm 3*. Both use a flag array *F1* of size $|V|$ to fulfil the requirements of the first condition, and a variable 'lock' to put the check for the second condition. Both use a node weight array (*Na*) of size $|V|$ to store the distance of nodes from the source node. Basic ideas of both implementations is similar, but have difference in implementation of relaxation function and the number of threads created to call their respective relaxation function.

Proposed Implementation 1

First proposed implementation of parallel SSSP algorithm is called the node based implementation because it creates $|V|$ threads for relaxation operation. *Algorithm 2* uses two kernels *INITIALIZATION* and *RELAX* define in *Kernel 1* and *Kernel 2* respectively for its basic operations.

Algorithm 2: PNBA (Graph G (V, E, W), Source node S)

Create node array (V_a), edge array (E_a), weight array (W_a), node weight array (N_a)

Create flag array F_a and a variable lock

1. **for** each node $v \in V$ **in parallel do**
2. Invoke INITIONALISATION(Na, S, Fa) on the grid
3. **end for**
4. lock = 1
5. **while** (lock == 1)
6. lock = 0
7. **for** each node $v \in V$ **in parallel do**
8. Invoke RELAX (Va, Ea, Wa, Fa, Na, lock) on the grid
9. **end for**
10. **end while**

Algorithm 2 calls the *INITIALIZATION* kernel for parallel initialization of the nodes weight and flag array values. It creates $|V|$ threads to call the *INITIALIZATION* kernel one thread corresponding to each node. *Algorithm 2* initialises the *lock* variable value to one and inside the loop at step 6 it again makes the *lock* value to zero and creates $|V|$ threads to call the *RELAX* kernel. Loop will run until the *lock* value is updated by *RELAX* kernel.

Each thread executing the *INITIALIZATION* kernel assigns infinity to its corresponding node weight and zero to the flag value. Each thread checks if its assign node is the source node then make the node weight value to zero and assign one to its flag value.

Kernel 1: INITIALIZATION (Na, S, Fa)

1. *id*=getThreadID
2. Na[*id*] = ∞
3. Fa[*id*] = 0
4. **if**(*id* == S) **then**
5. Na[*id*] = 0
6. Fa[*id*] = 1
7. **end if**

Kernel 2: RELAX (V_a, E_a, W_a, F_a, N_a, lock)

1. *id*= getThreadID
2. **if** Fa [*id*] **then**
3. Fa [*id*]=false
4. **for** all neighbours *nid* of *id* **do**
5. **if** (Na[Ea[*nid*]] > Na[*id*]+Wa[*nid*]) **then**
6. atimicMin(&Na[Ea[*nid*]], Na[*id*]+Wa[*nid*])
7. Lock = 1
8. Fa [Ea[nid]] = true
9. **end if**
10. **end for**
11. **end if**

Each thread executing the *RELAX* kernel checks if the flag value corresponding to its assigned node is one or not, if it is one then thread relax the all outgoing edges of its assigned node one by one. If the end node's weight of the edge the thread is relaxing is greater than the sum of the edge weight and edge start node's weight then the end node weight is updated by *atomicMin* operation, *lock* value is set to true and the flag array value corresponding to the end node is set to 1

Inconsistency

In this implementation, *Kernel 2* has a read after write conflict in the node weight update at step 6, because it performs read and write operations on the same array. But this inconsistency will never be a problem, as the update is implemented by an *ATOMIC* operation and *Kernel 2* always tries to minimize the node weight values. Suppose at any iteration if the node weight of node *A* is updated by any thread before the thread corresponding to node *A* reads the node weight value, the thread corresponding to node *A* will never get a node weight value greater than its previous value.

Kernel 2 also performs read and write operations simultaneously on the flag array, so some inconsistences can arise between threads, but it will never affect the final result of *Algorithm 2*. Suppose there are two threads *X* and *Y*, running the *Kernel 2*, for node *X* and *Y* respectively. The flag value corresponding to node *X* is 1 and thread *Y* has updated the flag of *X* to 0. When thread *Y* updates the flag value corresponding to node *X*, then three different events is possible.

(1) When in step 2 the thread X finds that the flag value corresponding to its assign node X is 1 and immediately after this thread Y execute the step 8 to

make the flag value corresponding to node X to 1. After this update ideally thread X should get the chance to relax its assigned node's outgoing edges in the next iteration. However, because thread X will reset the flag value corresponding to node X to 0 again in its step 3, thread X will not get the chance to relax the outgoing edges of node X in the next iteration. However this is not a problem as thread X will work on the current minimum value of node weight for node X in its current iteration. So the job which has to be done in the next iteration by thread X will be completed in the current iteration, which will save execution time.

(2) When in step 3 the thread X sets the flag value corresponding to node X to 0 and immediately after this the thread Y updates this flag value to 1 by executing the step 8. Here it is not a problem because thread X will get the chance to relax the outgoing edges of node X in the next iteration as well. But thread X will work on the same node weight value of node X in its current as well as in the next iteration.

(3) When in step 4 of the thread X, suppose it has completed n loops and immediately after this the thread Y updates the flag value corresponding to node X to 1 by executing the step 8, then it is not a problem for our implementation because thread X will get chance to relax the outgoing edges of node X in the next iteration as well. But here, first n edges are relaxed with the old value of the node weight of node X and the remaining edges will be relaxed with the current minimum value of the node weight of node X. This is not a problem because all edges will be relaxed again with the current minimum value of the node weight of node X in the next iteration as well.

Write after write inconsistency will not occur for any pair of threads because *Kernel 2* updates the node weights under an *ATOMIC* operation.

Proposed Implementation 2

Second proposed implementation of parallel SSSP algorithm is called edge based implement because it creates one thread corresponding to each edge of the graph for relax operation define in *Algorithm 3*. The algorithm computes the out-degree of each node when it stores the graph in a defined data structure. It uses the out degree of nodes as flag values, because it relaxes one edge in a thread so, it has to know how many edges have to be relaxed in any iteration due to any node weight update in the last iteration. *Algorithm 3* uses two kernels *INITIALIZATION1* and *REALX1* define in *Kernel 3* and Kernel 4 respectively for its basic operations. At first *Algorithm 3* calls the *INITIALIZATION1* kernel for parallel initialization of the nodes weight and flag array values. It creates $|V|$ threads to call the *INITIALIZATION1* kernel one thread corresponding to each node. *Algorithm 3* initialises the *lock* variable value to one. Inside the loop it again makes the *lock* value to zero and creates $|E|$ threads one for each edge of the graph to call the *RELAX1* kernel. Loop will run until the *lock* value is updated by *RELAX1* kernel. Each thread executing the *INITIALIZATION1* kernel assigns infinity to its corresponding node weight and zero as the flag value. Each thread checks if its assign node is the source node then make the node weight value to zero and assign the node's out degree value to flag. Each thread of *RELAX1* kernel checks if the assigned edge's source node indexed flag array value is zero or not, if it is greater than zero then it relaxes the edge and reduces this flag array value by one. This reduction process is an *ATOMIC* operation. Whenever a node weight is updated during the edge relaxation it sets the lock variable and flag array value corresponding to that node's index equal to the sum of its current value plus out degree of the node. The node weight is also updated by *ATOMIC* operation

Algorithm 3: PEBA (Graph G (V, E, W), Source node S)
Create edge start node array (S_a), edge end node array (E_a), edge weight array (W_a), node weight array (N_a).
Create out degree array (Da), flag array Fa and a variable lock

1. Compute out degree (D_a) for each $v \in V$
2. **for** each node $v \in V$ **in parallel do**
3. Invoke INITIONALISATION (N_a, S, Fa, D_a) on the grid
4. **end for**
5. lock = 1
6. **while** (lock == 1)
7. lock = 0;
8. **for** each edge $e \in E$ **in parallel do**
9. Invoke RELAX (N_a, S_a, E_a, W_a, Fa, D_a, lock) on the grid
10. **End for**
11. **End while**

Kernel 3: INITIALIZATION 1 (Na, S, F1, Da)

1. *id* = getThreadID
2. Na[*id*] = ∞
3. Fa[*id*] = 0
4. **if**(*id* = = S) **then**
5. Na[*id*] = 0
6. Fa[*id*] = Da[*id*]
7. **End if**

Kernel 4: RELAX 1 (N_a, S_a, E_a, W_a, F1, D_a, lock)

1. *id*= getThreadID
2. **if** (Fa[S_a[*id*]] > 0) **then**
3. **if** (N_a[E_a[*id*]] > N_a[S_a[*id*]]+W_a[*id*]))
4. atimicMin(&N_a[E_a**[***id***]],** N_a[S_a[*id*]]+W_a[*id*])
5. Lock = 1
6. atomicAdd (&Fa[E_a[*id*]], D_a[E_a[*id*]])
7. **End if**
8. atomicMin(&Fa[S_a[*id*]], Fa[S_a[*id*]]-1)
9. **End if**

Inconsistency and Overhead

In this implementation, *Kernel 4* also has a read after write conflict in the node weight update at step 4, but similar to the previous implementation it will never create a problem. *Kernel 4* uses *ATOMIC* operations to update the flag value, and due to these *ATOMIC* operations there will be no problem in the final result of the algorithm. But there is an overwork condition: if at any iteration of the loop in *Kernel 4* more than one threads update the weight of a single node, then the flag

value corresponding to the updated node is incremented more than once. So the result is that the outgoing edges of this updated node will be checked for the update by *Kernel 4* in multiple iterations. Write after write inconsistency will not occur for any pair of threads because *Kernel 4* updates the node weights under an *ATOMIC* operation.

Performance Analysis

Performance of proposed implementations has been evaluated on different types of real world graph datasets available on the Stanford Large Network Dataset Collection [31]. These graphs represent road networks, social site networks, computer networks and citation networks. This is a tested graph dataset with some graph properties, for example, the number of levels in the shortest path sub-tree of a graph is made available. We have divided this real world dataset into three sets. First is set of small graphs having eight to sixty thousand vertices and up to 0. 15 million edges with an average out degree between three to six and a number of levels in the shortest path sub-tree being seven to eleven. Second set has some large graphs having 0. 1to 1 million nodes and 1 to 5. 1 million edges with an average out degree between 5 to 10 and the number of levels in the shortest path sub-tree being 9 to 46. Third set has large, sparse graphs of US road network data, which are very linear, having an average out degree of 2 to 3 and the number of levels in the shortest path sub-tree is too high. Edge weights are randomly assigned for these real world graphs between 1 and 10. We have evaluated the results for a simple parallel Bellman Ford algorithm and both proposed implementations on two different types of GPU-based machine setup. SSSP algorithm proposed by Harish et al. [20] has been also implemented and its performance is evaluated for all mentioned graphs.

Experiential Setups

Two different machine setups are used to evaluate proposed implementation. Software and hardware configurations of these machines are mentioned as setup 1 and setup 2.

Setup 1

Processor: Intel i5 processor @ 3. 20 GHz
RAM: 4 GB
OS: Windows 7
Programming interface: Visual studios 2010
Graphics card: NVIDIA GeForce GTS 450. (192 cores), compute capability 2. 1
Language (parallel implementation): CUDA 5.

Setup 2

Processor: Intel(R) Xeon(R) E5-2650 @ 2. 00 GHz
RAM: 24 GB
OS: Windows 7
Programming interface: Visual studios 2010
Graphics card: Tesla C2075 (448 cores), compute capability 2. 0
Language (parallel implementation): CUDA 5.

Results

Processing time taken by any SSSP algorithm has been calculated for different graphs. Results are shown in the form of a figure with *x* and *y* axis. The *x* axis shows the size of the graph in terms of the number of nodes or edges in the graph. The *y* axis shows the processing time taken by the algorithm to calculate the SSSP in the graph, time is represented in milliseconds. Figure 3 and figure 4 show the results of proposed implementations and one previous [20] implementation on small graphs. Here PNBA and PEBA represent the node-based and the edge-based implementations respectively, and H_PA represents the implementation proposed by Harish et al. [20]. Figure 3 and Figure 4 show the results on setup 1 and setup 2 respectively. Inconsistency present in the Harish et al. [20] implementation was removed and results of that implementation are shown. Different setups are used to show the effect of the number of core in GPU.

Figures 3 and Figure 4 shows that both proposed implementations require less processing time and the edge-based implementation gives about a two-fold speed increase compared to the Harish et al. [20] method for small graphs. The edge-based implementation computed the SSSP on a graph having 140 thousand edges in just 2. 6 milliseconds. The edge-based implementation shows the best results here because its relaxation kernel has to process only one edge and if the out degree of the graph is very high then it reduces the relaxation operation complexity from *O(n)* to *O(1).*

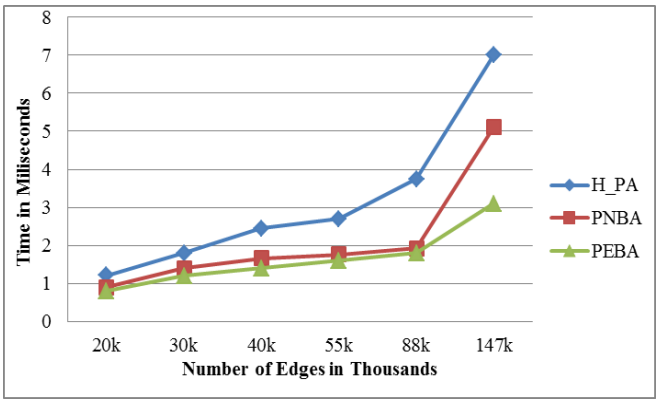

Figure 3: Results for small graphs in setup 1

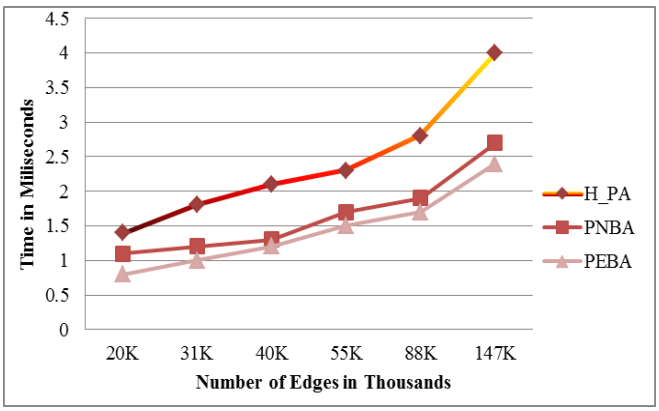

Figure 4: Results for small graphs in setup 2

Figure 5 shows the results for large graphs on setup 2. It shows that proposed edge-based implementation takes more time compared to other algorithms for large graphs. That is because, for large graphs, the number of threads in edge-based implementation is very high, and very few of these threads do productive work, so the GPU has to waste a large amount of time scheduling the unused threads. Figure 5 shows the uneven high processing time by all algorithms for graphs, which have1. 4 and 2. 3 million edges, because the number of levels in the shortest path sub-tree for these graphs are very high compared to other graphs.

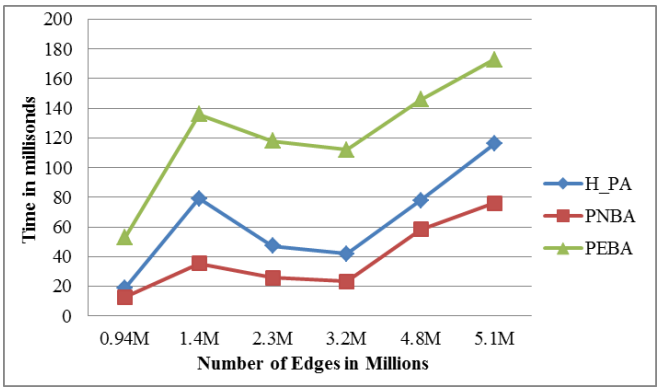

Figure 5: Results for large graphs in setup 2

Table 1: SSSP timings for very sparse graphs on setup 2

Graph Properties			Time in Millisecond		
Name	Nodes	Edges	PNBA	PEBA	H_PA
roadNet-CA	1965206	5533214	398	1022	711
roadNet-PA	1088092	3083796	155	615	384
roadNet-TX	1379917	3843320	248	820	551

Table 1 shows the result for very sparse graphs on setup 2. It shows that for very sparse large graphs all algorithms take more time compared to simple large graphs because as the out degree of nodes reduces, the level of parallelism also reduces.
Table 2 shows the results of proposed edge based implementation and the implementation of simple parallel Bellman ford algorithm (SPBFA). These implementations create the number of threads equal to the number of edges for any graph during the relaxation step, so it increases the parallelization.

Table 2: SSSP timings for small graphs on setup 2

Graph Properties			Time in Milliseconds	
Name	Nodes	Edges	PEBA	SPBFA
p2p-08	6301	20781	1. 08	60
p2p-05	8846	31839	1. 2	137
p2p-04	10876	40000	1. 22	173
p2p-25	22687	54711	1. 35	641
p2p-30	36682	88, 328	2. 1	860
p2p-31	62586	147916	2. 61	1786

Proposed implementation on the GPU gives very fast results compared to the simple parallel algorithms because it saves time in two ways; first it does not use any kernel to manage the flag as in the existing parallel implementation [20] on a GPU, and second, at any iteration it relaxes only those edges whose source node weight was updated in the last iteration.
The adjacency list is the more compact way to represent a graph, and was used in the previous parallel implementation of the SSSP algorithm [20] on a GPU-based machine. Adjacency list is also used for first proposed implementation but second implementation has used the unordered edge list. Proposed first implementation has used less memory as compare to Harish et al. [20] implementation. Second implementation has used some extra memory as compare to first one. The factors that mainly affected the algorithm processing times are how much work each kernel had to do and how many times the algorithm had to call the kernel. When the number of levels in the shortest path sub-tree of a graph was high, the algorithm took more time to calculate SSSP.

Conclusions

SSSP calculation is one of the most frequently used concepts in many real world applications. Harish et al. [20] have presented first parallel SSSP algorithm for GPU based machine using CUDA but their implementation is having some inconsistency and overheads. This paper has proposed the consistence and more efficient parallel implementations of algorithm proposed by Harish et al. [20]employing just a single CUDA kernel to maintain two modification conditions. Proposed implementations showed that how to save SSSP computation time by avoiding unnecessary edge relaxations and the need for the second kernel to copy flag values to implement the modification conditions. Proposed implementations have shown that some inconsistencies are present but why they are not going to affect the final results. It takes just 2. 6 milliseconds to processes a graph of. 15 million edges and this represents a speed increase of 600 times over the simple parallel Bellman Ford algorithm, more than double the speed of the SSSP algorithm previously implemented by Harish et al. [20].

References

[1] F. B. Zhan, and C. E. Noon, “Shortest path algorithms: an evaluation using real road networks, ” Transportation Science, Vol. 32, No. 1, pp. 65-73, 1998.

[2] K. Tan, Q. Zhang, and W. Zhu, “Shortest path routing in partially connected ad hoc networks, ” Proc. Global Telecommunications Conference, pp. 1038-1042, 2003.

[3] N. M. Garcia, P. Lenkiewicz, M. M. Freire, and P. P. Monteiro, “On the performance of shortest path routing algorithms for modelling and simulation of static source routed networks-an extension to the Dijkstra algorithm, ” Proc. Second International Conference on Systems and Networks

Communications, doi: 10. 1109/ICSNC. 2007. 56, 2007.

[4] F. B. Zhan, "Three fastest shortest path algorithms on real road networks Data Structures and Procedures, " Journal of Geographic Info and Decision Analysis, Vol. 1, No. 1, pp. 69-82. 1997.

[5] G. Trivedi, S. Punglia, and H. Narayanan, "Application of DC analyser to combinatorial optimization problems, " Proc. IEEE 20th International Conference on VLSI Design, pp. 869-874, 2007.

[6] F. R. Boyer, E. M. Aboulhamid, Y. Savaria, and M. Boyer, "Optimal design of synchronous circuits using software pipelining techniques, " ACM Transaction on Design Automation of Electronic System, Vol. 6, No. 4, pp. 516-532, 2001.

[7] F. Li, R. Klette, and R. Morales, "An approximate algorithm for solving shortest path problems for mobile robots or driver assistance, " Proc. Intelligent Vehicles Symposium, pp. 42-47, 2009.

[8] E. W. Dijkstra, "A note on two problems in connexion with graphs, " Numerische Mathematik, Vol. 1, pp. 269-271, 1959.

[9] R. E. Bellman, "On a routing problem, " Quarterly of Applied Mathematics, Vol. 16, pp. 87-90, 1958.

[10] L. R. Ford, and D. R. Fulkerson, "Flows in network, " Princeton University Press, 1962.

[11] A. Crauser, K. Mehlhom, U. Meyer, and P. Sanders, "A parallelization of dijkstra's shortest path algorithm, " Proc. 23rd International Symposium on Mathematical Foundations of Computer Science, pp. 722-732, 1998.

[12] G. Brodal, J. Traff, and C. D. Zarolingis, "A parallel priority data structure with applications, " Proc. IEEE 11th International Parallel Processing Symposium, pp. 689-693, 1997.

[13] J. R. Crobak, J. W. Berry, K. Madduri, and D. A. Bader, "Advanced shortest paths algorithms on a massively-multithreaded architecture, " Proc. IEEE International Parallel and Distributed Processing Symposium, pp. 1-8, 2007.

[14] Y. Tang, Y. Zhang, and H. A. Chen, "parallel shortest path algorithm based on graph-partitioning and iterative correcting, " Proc. IEEE 10th IEEE International Conference on High Performance Computing and Communications, pp. 155-161, 2008.

[15] A. Fetterer, and S. Shekhar, "A performance analysis of hierarchical shortest path algorithms, " Proc. IEEE Ninth IEEE International Conference on Tools with Artificial Intelligence, pp. 84-93, 1997.

[16] S. Sancı, and V. I¸sler, "A Parallel Algorithm for UAV Flight Route Planning on GPU, " International Journal of Parallel Programming, Vol. 39, No. 6, pp. 809-837, 2011.

[17] E. Sintorn, and U. Assarsson, "Fast Parallel GPU-Sorting Using a Hybrid Algorithm, " Journal of Parallel and Distributed Computing, Vol. 68, No. 10, pp. 1381-1388, 2008.

[18] W. Bura, and M. Boryczka, "The Parallel Ant Vehicle Navigation System with CUDA Technology, " Lecture Notes in Computer Science 6923, P. J edrzejowicz et al. (Eds.), Springer-Verlag, pp. 505-514, 2011.

[19] H. Jang, A. Park, and K. Jung, "Neural Network Implementation Using CUDA and OpenMP, " Proc. Digital Image Computing: Techniques and Applications, pp. 155-161, 2011.

[20] P. Harish, P. J. and Narayanan, "Accelerating large graph algorithms on the GPU using CUDA, " Lecture Notes in Computer Science 4873, G. Allen et al. (Eds.), Springer-Verlag, pp. 197-208, 2007.

[21] S. Kumar, A. Misra, and R. S. Tomar, "A modified parallel approach to single source shortest path problem for massively dense graphs using CUDA, " Proc. IEEE 2nd International Conference on Computer & Communication Technology, pp. 635-639, 2011.

[22] P. J. Martín, R. Torres, and A. Gavilanes, "CUDA Solutions for the SSSP Problem, " Lecture Notes in Computer Science 5544, G. Allen et al. (Eds.), Springer-Verlag, pp. 904-913, 2009.

[23] S. Dashora, and N. Khare, "Implementation of Graph Algorithms over GPU: A Comparative Analysis, " Proc. IEEE Students' Conference on Electrical, Electronics and Computer Science, pp. 1-8, 2012.

[24] D. P. Singh, and N. Khare, "Parallel Implementation of the Single Source Shortest Path Algorithm on CPU-GPU Based Hybrid System, " International Journal of Computer Science and Information Security, Vol. 11, No. 9, pp. 74-80, 2013.

[25] G. J. Katz, and J. T. Kider, "All pairs shortest-paths for large graphs on the GPU, " Proc. ACM 23rd ACM SIGGRAPH/ EUROGRAPHICS Symposium Graphics Hardware, pp. 47-55, 2008.

[26] K. Matsumoto, N. Nakasato, and S. G. Sedukhin, "Blocked All-Pairs Shortest Paths Algorithm for Hybrid CPU-GPU System, " Proc. IEEE 13th IEEE International Conference on High Performance Computing and Communications, pp. 145-152, 2011.

[27] Q. N. Tran, "Designing Efficient Many-Core Parallel Algorithms for All-Pairs Shortest-Paths Using CUDA, " Proc. IEEE Seventh International Conference on Information Technology: New Generations (ITNG), pp. 7-12, 2010.

[28] A. Buluc, J. R. Gilberta, and C. Budaka, "Solving Path Problems on the GPU, " Journal of Parallel computing, Vol. 36, pp. 241-253, 2010.

[29] J. Nickolls, and W. J. Dally, "The GPU computing era, " IEEE Micro, Vol. 30, pp. 56-69, 2010.

[30] D. B. Kirk, and W. W. Hwu, "CUDA: Programming Massively Parallel Processors: A hands-on approach. Morgan Kaufmann, 2011"

[31] J. Leskovec, "Stanford Large Network Dataset Collection. Stanford University, " http://snap. stanford. edu/data/.