Indian Journal of Science and Technology, Vol 9(33), DOI: 10.17485/ijst/2016/v9i33/87795, September 2016
ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

Modified Dijkstra's Algorithm for Dense Graphs on GPU using CUDA

Dhirendra Pratap Singh* and Nilay Khare

Department of Computer Science and Engineering, Maulana Ajad National Institute of Technology, Bhopal - 462003, Madhya Pradesh, India; dpsingh.manit@gmail.com, nilay.khare@rediffmail.com

Abstract

The objective of this research is to propose and implement a fast parallel Single source shortest path (SSSP) algorithm on Graphics Processing Unit (GPU) based highly parallel and cost effective platform for dense and complete graphs. The proposed algorithm is a variant of Dijkstra's algorithm for SSSP calculation for complete and dense graphs. In place of relaxing all the edges of a selected node as in Dijkstra's algorithm, it relaxes one-one selected edge of different nodes of the graph simultaneously at any iteration. This paper shows parallel implementation of both Dijkstra's algorithm and our modified Dijkstra's algorithm on a GPU-based machine. We evaluate these implementations on NVIDIA Tesla C2075 GPU-based machines. Parallel implementation of proposed modified Dijkstra's algorithm gives a two to three times speed increase over a parallel Dijkstra's algorithm on a GPU-based machine for complete and dense graphs. The proposed algorithm has minimized the number of edges relaxed by one parallel thread at any iteration of parallel Dijkstra's algorithm.

Keywords: CUDA, Graph Algorithm, GPU Computing, Parallel Dijkstra's Algorithm, Parallel Single Source Shortest Path Algorithm

1. Introduction

The performance improvement of different base algorithms required to solve real world problems are explored in computer technology. Graph algorithms are still one of the hottest areas for research[1-4]. SSSP calculation on the graph data of real world fields such as peer to peer (web) file sharing[5,6], social networking[7,8], and gene networks[9,10] always requires a performance improvement. In this paper we propose a modified Dijktra's algorithm and its parallel implementation for SSSP calculation for complete and dense graphs. To explain this algorithm first we discuss some basic terminologies used in the SSSP algorithm. Let $G= (V, E)$ be a $|V|=n$ vertices and $|E|=m$ edges directed graph with a positive cost related to each edge. A graph having a unique edge for every pair of nodes is called a complete graph. A *path* between any pair of nodes in the graph is the set of edges required to connect these nodes, and multiple paths can be possible for any pair of nodes in the graph. The *length* of a path or distance between nodes is the sum of the cost of all edges that constitutes the path. Out of all the possible paths for any pair of nodes, the path with the minimum length is called the shortest path of the nodes in the pair. SSSP problem calculates the shortest path for all pairs of fixed source nodes and all other nodes in the graph. All the shortest path algorithms start with initializing the source node distance to zero and all other node distance to infinity from the source node. These node distance values are optimized by a *relax* operation during the execution of the algorithm. Suppose a node pair (u, v) shows an edge of a graph. During the relaxation of the edge (u, v) distance value of node v is updated by minimum of the current distance value of node v and sum of the distance of u and the cost of edge (u, v).The Shortest path algorithms are divided into two groups on the basis of the approach used to evaluate the distance of a node from the source node during its execution. These groups are label-setting and label-correcting. Label-setting algorithms relax any edge just once but label-correcting algorithms can relax an

*Author for correspondence

edge multiple times during the execution of the algorithm. Dijkstra's algorithm[11] is the basic label-setting approach as it assigns a permanent distance label (cost) to a node in each of its iterations. Dijkstra's algorithm with Fibonacci heap running time is $O(nlogn + m)$. The Bellman Ford algorithm[12,13] is a label-correcting algorithm as in this algorithm, distance labels of every node are checked for its correction in each iteration of the algorithm. The Bellman Ford algorithm's running time is $O(mn)$.

Different implementations of parallel SSSP algorithms have already been tried on PRAM, CRAY supercomputers and for some other parallel machines to solve it in real time, but at a high hardware cost. While such implementations are fast their underlying hardware is very expensive, so a GPU provides a very cost-effective and parallel platform for such implementations. The GPU consists of hundreds of cores, with a four stage pipeline per core, all working in a multi-threaded architecture. GPUs are now widely used in different fields[14-21] where high performance computing is required and show tremendous performance and processing throughput.

A. Crauser et al.[22] have shown a PRAM-based parallel modified Dijkstra's algorithm with $O(n^{1/3}logn)$ time. G. Brodal et al.[23] presented priority queue for parallel queue operations of Dijkstra's algorithm and shown a $O(n)$ time algorithm on PRAM. P. J. Narayanan[24] presented an efficient approach for solving SSSP on processor arrays. M. Papaefthymiouet et al.[25] presented a parallel version of Bellman-Ford algorithm on a CM-5 parallel supercomputer. Y. Tang et al.[26] have shown a parallel SSSP algorithm which is based on graph partitioning. U. Meyer et al.[27] proposed a parallel label-correcting algorithm for SSSP calculation and named it the Δ-Stepping algorithm. J. R. Crobaket et al.[28] gave a parallel implementation of Thorup's Algorithm[29] in which they used a hierarchical bucketing structure for nodes to perform constant time operations on them. A. Fettereret et al.[30] have also shown a graph partitioning based parallel SSSP algorithm, where graph is represented in hierarchical data structure. P. Harish et al.[31] presented the first parallel implementation of SSSP algorithm for GPU based machines. They have proposed a modified Bellman Ford algorithm[12, 13] to calculate SSSP in positive edge weighted graphs. Their algorithm calculates the SSSP on 1 million vertices graph in 1-2 seconds. Sumit Kumar et al.[32] presented a parallel Bellman Ford Algorithm on GPU. Pedro J. Martín et al.[33] presented a parallel Dijkstra's algorithm for finding SSSP on GPU based machine. They have shown the parallelization in both the minimum selection and edge relaxation operations of the algorithm. Dashoraet et al.[34] have also proposed GPU based parallel implementation of SSSP and other shortest path algorithms. These[35,36] have shown GPU based implementation of modified Dijkstra's algorithm proposed by Crauseret et al.[22]. Davidson et al.[37] has proposed three different GPU based SSSP algorithms. These algorithms start with the concept of parallel implementation of Bellman Ford algorithm[12,13]and move towards the Δ-steppingalgorithm[27]. Ortega et al.[38] has shown comparative analysis between GPU based parallel implementation of Dijkstra's algorithm and GPU algorithm proposed by Pedro J. Martín et al.[33] and GPU based implementation of Crauser et al.[22] algorithm. Singh et al.[39] has proposed improved and consistency implementation of P. Harish et al.[31] algorithm on GPU based machine. Many researchers[40-43] have proposed GPU-based parallel implementations of all pair shortest path algorithms. In this paper we present parallel implementation of proposed modified Dijkstra's algorithm and the basic Dijkstra's algorithm over GPU using CUDA.

The rest of the paper is organised as follows. The CUDA programming model is discussed in section 2, Section 3 presents the graph representation used to define algorithms. Section 4 explains the serial and parallel Dijkstra's algorithms. In section 5 we represent our proposed algorithm and its parallel implementation. Section 6 shows the performance analysis and results of our implementations on different graphs and at last conclusion of the work is discussed in Section 7.

2. CUDA Programming Model

Nvidia has introduced the CUDA (Compute Unified Device Architecture) as a programming language to make use of the parallel architecture of their GPUs. The CUDA programming is an extension to existing programming languages such as Fortran and C because it has header files or APIs to the language[44]. Basically, a CUDA device is a multi-core co-processor of CPU that is used for heavy computational tasks. The assigned task cannot be initiated by GPU itself, so along with it a CPU is always required for this purpose. In a general purpose GPU, memory usage is somewhat restricted as a number of symmetric multiprocessors (SM) are present and each SM has a set of processors (core) with shared memory. Each core has a fast register memory and its own private local memory

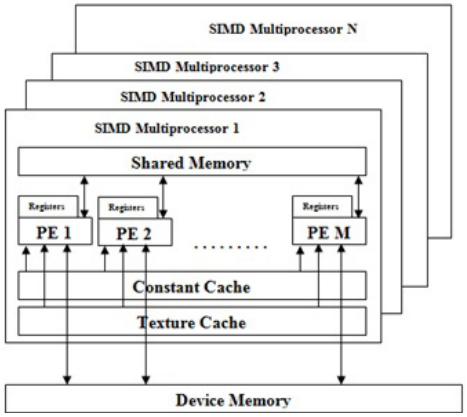

Figure 1. GPU Architecture.

implemented in external DRAM. Global and constant memories are also present in DRAM and are accessible to all cores of any SM in the GPU.

In logical model of CUDA, it creates user defined numbers of threads; this number of threads also depends on GPU architecture. Each CUDA thread executes the requested kernel code on different data items. This data should be present in GPU memory. In CUDA a logical thread is not directly assigned to a GPU core. These threads are arranged in one, two or three dimensional way inside Block. Each thread is assigned a unique index for each dimension of its arrangement in the Block. Similarly Blocks are logically arranged in one, two or three dimensional way inside Grid and each Block is assigned a unique index for each dimension of its arrangement in the Grid. A Grid of threads is assigned to GPU for execution, where one or more Blocks are assigned to a SM. SM divides these thread blocks into group of 32 threads, this group of threads is called warp. Threads of warp are assigned to core of MP for execution[44,45].

3. Graph Representation

Let (V_0, V_1, V_2, V_3 ..... V_{n-1}) are the nodes of the graph *G(V, E)*, where $|E|= m$ and $|V|= n$. All pairs of nodes (V_i ,V_j) represents the outgoing edges of node V_i, where $i=0$ to $n-1$ and $j=0$ to $n-1$ $-\{i\}$. Number of pairs with starting node V_i, is called out-degree of node $V_i$and represented as d_i. In this graph representation three arrays are defined, *N* of size $n+1$ and *E* and *W* of size *m*. Each index in array *N* symbolizes a node number. Suppose the value of array *N* at index *i* is X_i.

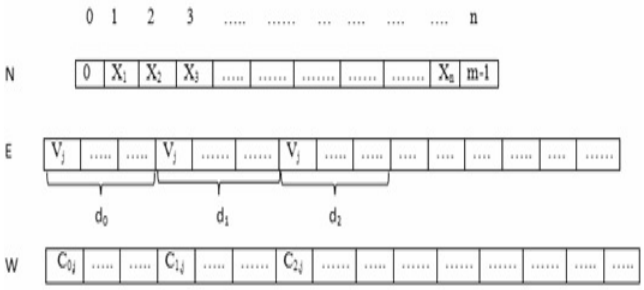

Figure 2. Graph representation.

$$X_i = \begin{cases} 0 & i = 0 \\ \sum_{j=0}^{i-1} d_j & i > 0 \end{cases}$$

Where $i=0$ to n

The array *E* stores the destination node number of all edges in the graph. For node V_i destination node entries of its all outgoing edges start from index $N[V_i]$ in the array *E* and the order of the nodes is in ascending order of their corresponding edge weight (i.e. if $W(V_i, V_j) > W(V_i, V_k)$ then node V_k come before the node V_j). The W array stores the edge weight of the edge (V_i, V_j) at the same index where node $V_j$is stored in array *E*.

4. Dijkstra's Algorithm

Dijkstra's[11] algorithm calculates the SSSP for the positive edge weighed graph shown in *Algorithm 1*. At the time of the execution of the algorithm it puts the graph nodes in three different sets. The first set is called '*settled nodes*', and is a set of those nodes that have got their minimum weight. The second set is '*queued nodes*'; it contains those nodes whose weight is less than infinity. The third set is '*unreached nodes*'; it contains those nodes whose weight is infinity. *Algorithm 1* uses an array *NW* of size $|V|$ to store the node weight of the graph nodes. At step 1 *Algorithm 1* initializes the given source node '*S*' weight to zero and infinity to others. It also assigns value zero to variable *MIN*. Initially, nodes are divided in two sets; the source node is in the queued set and all other nodes are in the unreached set. In each iteration of step 3, *Algorithm 1* evaluates the new *MIN* value with the help of all current queued nodes weight. After finding the current minimum value for variable *MIN, Algorithm 1* removes the queued node having the node weight equal to the *MIN*

value from queue and adds this node to the settled node group, with relaxing its outgoing edges. Step 3 is repeated until the queue is not empty and when the queue become empty all nodes get their minimum distance from the source node.

Dijkstra's algorithm relaxes all the outgoing edges of the node removed from the queue. For a dense graph it is an overhead so we propose a modified version of the Dijkstra's algorithm which relaxes only one outgoing of any node in each iteration.

Algorithm 1: Dijkstra's Algorithm(Graph G (V, E), S)
Define array NW of size |V|, a variable '***infinite***' and assign a big number to it, define a variable MIN.
BEGIN
Step 1: for all node *n* **do**
NW[*n*] =infinite
for end
NW[*S*]=0, MIN=0
Step 2: Add the node *S* in queue
Step 3: while (queue is not empty) **do**
MIN=infinite
for all queued node *m* **do**
MIN=minimum(MIN, NW[*m*])
for end
for all node *m* in queue **do**
if(NW[*m*]= MIN) **then**
Remove *m* from queue and add to settled group.
for all edge [*m, p*] ε E**do**
NW[*p*]=minimum(NW[*p*], NW[*m*] + W[*m, p*])
Add node *p* in queue if it is not present in the queue.
for end
if end
for end
while end
END

To explain that how our algorithm gives better performance as compared to the parallel Dijkstra's algorithm we first present the parallel implementation of the Dijkstra's Algorithm for GPU-based machines using CUDA.

4. 1 Parallel Implementation of Basic Dijkstra's Algorithm

Algorithm 2 : Parallel Basic Dijkstra's Algorithm(Graph G (V, E, W), S)
Define array NW of size |V|, a Boolean array M of size |V|, Define variables MIN and ***infinite***and assign a big number to ***infinite***.
BEGIN
Step 1:INITIAL-VALUE(NW, M, S) for all nodes in parallel
MIN=0
Step 2: while (MIN< ***infinite***) **do**
MIN= ***infinite***
MINIMUM(M, N, NW, E, W, MIN) far all nodes in parallel
RELAX(N, NW, E, W, M, MIN) for all nodes in parallel
while end
END

The parallel Dijkstra's algorithm for GPU based machine using CUDA is shown in *Algorithm 2*. It uses two arrays *NW* and *M* of size |*V*| to store the node weight and mark the node settled or unsettled respectively. It uses three kernels to fulfill the requirements of the algorithm. The 1st kernel is the *INITIAL-VALUE* kernel presented in *Algorithm 3*; it sets the initial node weight and flag value required for every node of the graph. The 2nd kernel is *MINIMUM*, presented in *Algorithm 4*, it evaluates the current value of the *MIN* variable. The 3rd kernel is *RELAX-EDGE*, defined in *Algorithm 5*, which marks the node whose weight is equal to the *MIN* value calculated by the *MINIMUM* kernel and relaxes the marked node's outgoing edges.

In step1 *Algorithm 2* creates |*V*| threads to execute the *INITIAL-VALUE* kernel. It assigns the initial value of the *MIN* variable as zero.

Algorithm 3: INITIAL-VALUE (NW, M, S)
BEGIN
t=getThreadID
NW[*t*]=***infinity***
M[*t*]=0
if (t= =S) **then**
NW[*t*]=0
M[*t*]=1
if end
END

In each iteration of step 2, *Algorithm 2* calculates the current value of the variable *MIN* and then mark settled to node whose current weight equals the *MIN* value. This loop in step 3 continues until *MIN* value is not infinity. To determine the current *MIN* value it creates |*V*| threads to call the *MINIMUM* kernel. Each thread checks for its assigned

node's flag value if it is not set and the corresponding node weight is less than *INFINITY*, then minimum of current *MIN* value and node weight is assigned to the variable MIN by an atomic operation. Atomic operations are used to avoid the inconsistency arises if multiple threads try to modify the value of *MIN* simultaneously.

Algorithm 4: MINIMUM (M, N, NW, E, W, MIN)
BEGIN
t=getThreadID
if(M[*t* !=1 && NW[*t*] <***infinity***) **then**
Begin ATOMIC
MIN = minimum (MIN, NW[*t*])
ATOMIC end
if end
END

After the *MIN* value calculation for the current iteration of step 2, *Algorithm 2* calls the kernel *RELAX-EDGE* with |*V*| threads. Each thread checks if its assigned node's flag value is not set and it is having weight equal to the current minimum value, then the flag value corresponding to this node is set and all outgoing edges of this node are relaxed.

Algorithm 5: *RELAX-EDGE*(N, NW, E, W, M, MIN)
BEGIN
t=getThreadID
if(M[*t*]!=1 && NW = = MIN) **then**
M[*id*]=1
for all nodes *m* ε V the successors of node ***t*do**
Begin ATOMIC
NW[*m*] = minimum (NW[*m*] , NW[*t*] + W[*t, m*])
ATOMIC end
for end
if end
END

During the relaxation step the Dijkstra's algorithm has to check all outgoing edges of current node for a possible weight update of the edge's destination node, in case of complete graph outgoing edge's count for any node is *n-1*. Therefore, a thread of the *RELAX* kernel whose corresponding node will be marked settled has to check all the outgoing edges on the this node, thread has to perform the *O(n)* job. To remove this constraint of *O(n)* job and make it *O(1)* we propose the modified Dijkstra's algorithm.

5. Proposed Algorithm

This section first presents the basic the concepts of our proposed improved Dijkstra's algorithm and then its parallel implementation for a GPU-based machine using CUDA.

Our proposed improved Dijkstra's algorithm presented as *Algorithm 6*. It uses an array *NW* of size |*V*| to store the node weight of the graph nodes. Similar to the Dijkstra's algorithm it initializes the all nodes weight to infinity and source node weight as zero. It assigns the *MIN* variable to zero and adds the source node to the settled group. In each iteration of the loop in step 3 of the *Algorithm 6* calculates a new value of *MIN* with the help of the current settled nodes. Every settled node relaxes its current minimum weighted outgoing edge, whose destination node is unsettled. In the step 3.3 out all queued nodes, the node whose current weight is equal to the current *MIN* value is added to the settled group. This loop in step 3 runs until there is at least one unsettled node. In place of relaxing the all outgoing edges of a settled node, proposed algorithm try to relax one outgoing edge of all the settled nodes having at least one edge with unsettled destination node. In the proposed parallel implementation of this modified algorithm we will show how this relaxation work will be distributed to different processing elements.

Algorithm 6: Improved Dijkstra's Algorithm(Graph G (V, E), S)
Define array NW of size |V|, a variable '***infinite***' and assign a big number to it define a variable MIN.
BEGIN
Step 1: for all node *n* **do**
NW[*n*] = ***infinite***
for end
NW[S]=0, MIN=0
Step 2: Add the Source node in settled group
Step 3: while (all nodes are not settled) do
3.1:MIN =***infinite***
3.2:for all settled node *m* **do**
for 1st edge(*m, k*) ε E , where *k* is unsettled **do**
NW[*p*] = minimum (NW[*k*], MW[*m*] + W[*m, k*])
MIN = minimum (MIN, NW[*k*])
for end
for end
3.3: for all nodes *m* present in queue**do**
if(NW[*m*]= = MIN) **then**
Add node *m* to the settled group
if end
for end
while end
END

5.1 Parallel Implementation of Proposed Algorithm

The parallel implementation of the modified Dijkstra's algorithm for GPU-based machine using CUDA is presented as *Algorithm 7*. It also uses two arrays *NW* and *M* of size $|V|$ to store the node weight and mark the node settled or unsettled respectively. This implementation uses three kernels for the basic operations of the algorithm. These kernels are the *INITIAL-VALUE*, *MINIMUM1* and *SET_FLAG*, defined as *Algorithm 3*, *Algorithm 8* and *Algorithm 9* respectively.

At first step *Algorithm 7* calls *INITIAL-VALUE* kernel, which assign infinity as all nodes weight except the source node, which is assigned weight as zero, the source node flag value to one, and sets all other nodes to zero. To the call of the *INITIAL-VALUE* kernel it creates $|V|$ threads. In each iteration of the loop of step 2 in *Algorithm 7*, it first set the *MIN* value to infinity and then calls the *MINIMUM1* and *SET_FLAG* kernels until the *MIN* value is less than infinity.

Algorithm 7: Parallel Modified Dijkstra(Graph G (V, E, W),S)
Define arrays NW AND N1 of size |V|, a variable '***infinite***' and assign a big number to it, define a variable MIN, array N1 is a copy of N.
BEGIN
Step 1: INITIAL-VALUE(NW, M) for all nodes in parallel
MIN=1
Step 2: while (MIN< ***infinite***) **do**
MIN=***infinite***
MINIMUM (M, N, N1, NW, E, W,MIN) far all nodes in parallel
SET_FLAG (M, NW, MIN) for all nodes in parallel
while end
END
Algorithm 8: MINIMUM1 (M, N, N1, NW, E, W, MIN)
BEGIN
t=getThreadID
if(M[*t*]==1) **then**
for all edges (*t, z*) (check in N1 array)**do**
if(M[E[*z*]]!=1) **then**
Begin ATOMIC
if(NW[E[*z*]] > NW[*t*]+W[*t, z*]) **then**
NW[E[*z*]] = NW[*t*]+W[*t, z*]
if end
ATOMIC end
Begin ATOMIC
if(MIN > NW[E[*z*]]) **then**
MIN = NW[E[*z*]]
if end
ATOMIC end
N1[*t*]=*z*
break
if end
for end
if end
END

Algorithm 7 calls the kernel *MINIMUM1* with $|V|$ threads. This kernel uses the *N1* array to check the outgoing edges of any node. The *N1* array maintains a copy of array *N*. Every thread verify whether its allocated node's flag value is set or not, and if it is set and node has at least one outgoing edge with an unset flag value for its destination node, then the edge is relaxed. This relax operation is executed by an atomic operation to keep away from any read/write conflicts. Now the weight of the destination node of the relaxed edge participates in the current *MIN* value calculation using an atomic operation. When any edge is used as a minimum weighted outgoing edge for its source node then to save time getting the next edge in a future iteration we remove the edges present before the current edge.

Algorithm 9: SET_FLAG (M, NW, MIN)
BEGIN
t=getThreadID
if(M[t]!=1 &&NW[t]==MIN) **then**
M[*t*]=1
if end
END

After the calculation of the new value of *MIN* for the current iteration of the loop in step 2, *Algorithm 7* calls the kernel *SET_FLAG* with $|V|$ threads. Each thread ensures that its assigned node's flag value is not 1 and it is having weight equal to the current minimum value, then set 1 as corresponding node's flag value.

6. Performance Analysis

6.1 Mathematical Analysis

Suppose *G= (V, E)* is a complete graph i.e. the out degree of each node is *(n-1)*, where $n=|V|$. Suppose our machine has *n* processors, one corresponding to each node of

the graph. Algorithms perform two operations at any iteration, one is the minimum selection and the other is relaxation of edges. Suppose we have created *n* thread and each thread is assigned to a processor.

The minimum finding step has to do a similar amount of work for the parallel realization of both algorithms. Dijkstra's algorithm uses the queued nodes to find the current minimum values. For a complete graph, the initial size of the queue will be (*n-1*) and in each iteration size the queue decreases until it become empty. Suppose a single node is eliminated from the queue in each iteration. Thus the time required for selection of all minimum values will be $O(n^2)\{(n-1)+(n-2)+\ldots\ldots\ldots 3+2+1\}$. In our proposed algorithm the number of nodes which will participate in the minimum value calculation at any iteration is the number of nodes currently present in the settled group. The count of settled nodes will initially be one and in each iteration its value increases until it reaches *n*. Suppose one node is added in the settled group in each iteration. Then the time required for selection of all minimum values by our proposed algorithm is $O(n^2)\{1+2+3+\ldots\ldots(n-2)+(n-1)\}$.

Any thread of the parallel Dijkstra's algorithm relaxes all the outgoing edges of any settled node in any iteration, i.e. the thread has to do *O(n)* work for a complete graph. But any thread of our proposed algorithm relaxes only one outgoing edge at any iteration. So our proposed algorithm reduces the complexity of the relaxation operation of Dijkstra's algorithm from *O(n)* to *O(1)*.

6.2 Experimental Analysis

We evaluated the performance of our implementations for randomly generated graph data. We used three types of generated graphs; the first set had dense graphs with 1.5 thousand to five thousand nodes with 30% to 40% out degree. The second set had complete graphs with five hundred to three thousand nodes. Third set had randomly generated graphs with few nodes of high out degree. Weights were randomly assigned between 1 to 100 to each edge of the graph. The experimental setup to evaluate the implementations is having CPU with 12 Intel Xeon X 5660, 2.8GHz processor, twenty four GB RAM with the Windows 7 operating system, and for parallel computing the setup used a Tesla C2075 GPU, which is having 448 cores and CUDA 5.0 which was configured with visual studio 2010.

We evaluated the processing time any algorithm takes to calculate the SSSP in a graph. We represent the results

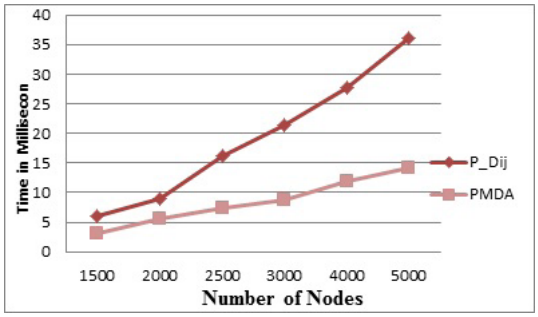

Figure 3. Results for Dense Graphs.

Table 1

Number of Nodes	Degree of Graph	P_Dij	PMDA	Speedup
1500	400	6.14	3.2	1.9X
2000	500	9.04	5.5	1.6X
2500	600	16.1	7.3	2.2X
3000	1000	21.39	8.7	2.5X
4000	1000	27.64	11.9	2.3X
5000	1200	36.19	14.1	2.6X

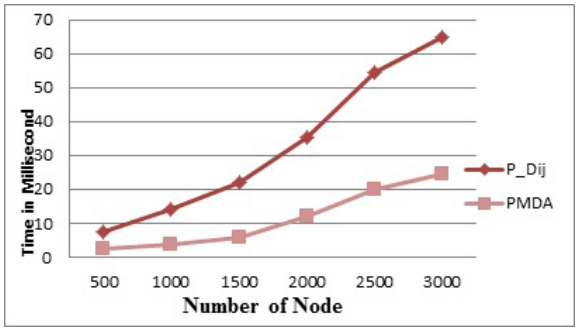

Figure 4. Results for Complete Graphs.

as a figure having two axes *x* and *y*. The *x* axis represents 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. The time is represented in milliseconds.

Results of the parallel Dijkstra's algorithm (P_Dij) and the proposed parallel modified Dijkstra Algorithms (PMDA) for dense and complete graphs are shown in figures 3, 4 respectively. Figure 3 shows that proposed algorithm takes 14 milliseconds to processes a graph having five thousands nodes and ten million edges. Figure 4 shows that proposed algorithm takes 24.6 milliseconds to processes a complete graph of three

thousands nodes. It shows that as the number of nodes increases, the performance of the proposed algorithm improves compared to the parallel Dijkstra's algorithm, because as the number of nodes increases the workload of the processing elements increases in the parallel Dijkstra's algorithm. Table 1 demonstrate the speedup achieved by proposed algorithm in compare to parallel Dijkstra algorithm.

Both parallel implementations are also tested for randomly generated graphs, where out-degree of nodes is varying from 1 to x. We have generated the graphs in such a way show that only few nodes get the high out-degree and all other nodes have very small out-degree. We have found that if for a graph the node having high out-degree is settled at the very starting of the SSSP calculation then proposed parallel algorithm is giving better results than the parallel Dijkstra's algorithm. Parallel Dijkstra algorithm took more time because the after relaxing the edges of high out-degree node queue size reaches near to it maximum limit. Due to the large queue size the time complexity of minimum selection step get increased. If the high out-degree nodes are settled at very last point of SSSP calculation then parallel Dijkstra's algorithm outperform the proposed parallel algorithm. Proposed parallel algorithm took more time to presses the graph because it use the settled nodes for minimum selection and the number of settled node increases gradually but the queue of parallel Dijkstra's algorithm is very small all the time.

7. Conclusion

This paper has proposed and shown the parallel implementation of a modified Dijkstra's algorithm for GPU-based machines using CUDA. Paper has shown how the parallel implementation of our proposed algorithm can reduce the workload of any processing element by relaxing the single edge at any iteration as compared to the parallel Dijkstra's algorithm. We have shown the results for our proposed algorithm and Dijkstra's algorithm on different sizes of generated dense and complete graphs. For both types of graphs, parallel implementation of our proposed algorithm provided up to a three-fold speed increase in comparison of the parallel execution of Dijkstra's algorithm on the GPU-based machine. Many real world field data produces highly dense graphs, so for an application based on such field data our proposed algorithm is the best.

8. References

1. Zhang D, Rubinstein BIP, Gemmell J. Principled Graph Matching Algorithms for Integrating Multiple Data Sources. IEEE Transactions on Knowledge and Data Engineering. 2015; 27(100):2784–96.
2. Dadaneh BZ, Markid HY, Zakerolhosseini A. Graph coloring using Intelligent Water Drops algorithm. Proceeding of 23rd Iranian Conference on Electrical Engineering. 2015.
3. Cheraghi A. Searching for an Unknown Edge in the Graph and its Tight Complexity Bounds. Indian Journal of Science and Technology. 2016; 9(13). Doi:10.17485/ijst/2016/v9i13/71360.
4. Albert S, Raj A, Victor SP. Algorithms to Find Geodetic Numbers and Edge Geodetic Numbers in Graphs. Indian Journal of Science and Technology. 2015; 8(13). Doi: 10.17485/ijst/2015/v8i13/53160.
5. Leung AKH, Kwok YK. An Efficient and Practical Greedy Algorithm for Server-Peer Selection in Wireless Peer-to-Peer File Sharing Networks. LNCS Springer-Verlag Berlin Heidelberg. 2005; 3794:1016–25.
6. Davoust A, Esfandiari B. Collaborative Building, Sharing and Handling of Graphs of Documents Using P2P File-Sharing. LNCS Springer-Verlag Berlin Heidelberg. 2009; 5872:888–97.
7. Papagelis M. Sampling Online Social Networks. IEEE Transaction on Knowledge and Data Engineering. 2013; 25(3):662–76.
8. Das S. Anonimos: An LP-Based Approach for Anonymizing Weighted Social Network Graphs. IEEE Transaction on Knowledge and Data Engineering. 2012; 24(4):590–604.
9. Takigawa I, Mamitsuka H. Probabilistic path ranking based on adjacent pair wise coexpression for metabolic transcripts analysis. Bioinformatics. 2008; 24(2):250–7.
10. Brown JA. Examination of Graphs in Multiple Agent Genetic Networks for Iterated Prisoner's Dilemma. Proceedings of the Conference on Computational Intelligence in Games. 2013.
11. Dijkstra EW. A note on two problems in connexion with graphs. Numerische Mathematik. 1959; 1(1):269–71.
12. Bellman RE. On a routing problem. Quarterly of Applied Mathematics. 1958; 16:87–90.
13. Ford LR, Fulkerson DR. Flows in Network. Princeton University Press. 1962.
14. Sanci S, Isler V. A Parallel Algorithm for UAV Flight Route Planning on GPU. International Journal of Parallel Programming. 2011; 39(6):809–37.
15. Sintorn E, Assarsson U. Fast Parallel GPU- Sorting Using a Hybrid Algorithm. Journal of Parallel and Distributed Computing. 2008; 68(10):1381–8.
16. Jang H, Park A, Jung K. Neural Network Implementation Using CUDA and Open MP. Proceedings of the Digital Image Computing: Techniques and Applications. 2008.

17. Habibpour L, Yousefi S, Lighvan MZ, Aghdasi HS. 1D Chaos-based Image Encryption Acceleration by using GPU. Indian Journal of Science and Technology. 2016; 9(6). Doi:10.17485/ijst/2016/v9i6/72651.
18. Khemiri R, Sayadi FE, Atri M. MatLab-GPU-based 2D-DWT Acceleration for JPEG2000 with Single and Double-Precision. Indian Journal of Science and Technology. 2016; 9(12). Doi:10.17485/ijst/2016/v9i12/80526.
19. Zha X, Sahni S. GPU-to-GPU and Host-to-Host Multipattern String Matching on a GPU. IEEE Transaction on Computers. 2013; 62(6):1156–69.
20. Faujdar N, Ghrera SP. Performance Evaluation of Parallel Count Sort using GPU Computing with CUDA. Indian Journal of Science and Technology. 2016; 9(15). Doi:10.17485/ijst/2016/v9i15/80080.
21. Ha S, Matej S, Ispiryan M, Mueller K. GPU-Accelerated Forward and Back-Projections With Spatially Varying Kernels for 3D DIRECT TOF PET Reconstruction. IEEE Transaction on Nuclear Science. 2013; 60(1):166–73.
22. Crauser A, Mehlhom K, Meyer U, Sanders P. A parallelization of dijkstra's shortest path algorithm. LNCS Springer-Verlag Berlin Heidelberg. 1998; 1450:722–31.
23. Brodal G, Traff J, Zarolingis CD. A parallel priority data structure with applications. Proceedings of the 11th International Parallel Processing Symposium. 1997.
24. Narayanan PJ. Single source shortest path problem on Processor arrays. Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computing. 1992.
25. Papaefthymiou M, Rodrigue J. Implementing parallel shortest-paths algorithms. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. 1994; 59–68.
26. Tang Y, Zhang Y, Chen HA. Parallel shortest path algorithm based on graph-partitioning and iterative correcting. Proceedings of the International Conference on High Performance Computing and Communications. 2008.
27. Meyer U, Sanders P. Δ-stepping: A parallelizable shortest path algorithm. Journal of Algorithms. 2003; 49(1):114–52.
28. Crobak JR, Berry JW, Madduri K, Bader DA. Advanced shortest paths algorithms on a massively-multithreaded architecture. Proceedings of the Parallel and Distributed Processing Symposium. 2007.
29. Thorup M. Undirected single-source shortest paths with positive integer weights in liner time. Journal of the ACM. 1999; 46(3):362–94.
30. Fetterer A, Shekhar S. A performance analysis of hierarchical shortest path algorithms. Proceedings of the International Conference on Tools with Artificial Intelligence. 1997.
31. Harish P, Narayanan PJ. Accelerating large graph algorithms on the GPU using CUDA. LNCS Springer Berlin Heidelberg. 2007; 4873:197–208.
32. Kumar S, Misra A, Tomar RS. A Modified Parallel Approach to Single Source Shortest Path Problem for Massively Dense Graphs Using CUDA. Proceedings of the International Conference on Computer and Communication Technology. 2011.
33. Martin PJ, Torres R, Gavilanes A. CUDA Solutions for the SSSP Problem. LNCS Springer-Verlag Berlin Heidelberg. 2009; 5544:904–13.
34. Dashora S, Khare N. Implementation of Graph Algorithms over GPU: A Comparative Analysis. Proceeding of IEEE Students Conference on Electrical, Electronics and Computer Science. 2012.
35. Singh DP, Khare N. Parallel Implementation of the Single Source Shortest Path Algorithm on CPU–GPU Based Hybrid System. International Journal of Computer Science and Information Security. 2013; 11(9):74–80.
36. Ortega-Arranz H, Torres Y, Llanos DR, Gonzalez-Escribano A. A New GPU-based Approach to the Shortest Path Problem. Proceedings of International Conference on High Performance Computing and Simulation (HPCS). 2013.
37. Davidson A, Baxter S, Garland M, Owens JD. Work-efficient Parallel GPU Methods for Single-Source Shortest path. Proceedings of 28th IEEE International Parallel and Distributed Processing Symposium. 2014.
38. Ortega-Arranz H, Torres Y, Llanos DR, Gonzalez-Escribano A. Comprehensive Evaluation of a New GPU-based Approach to the Shortest Path Problem. International Journal of Parallel Programming. 2015; 43:918–38.
39. Singh DP, Khare N, Rasool A. Efficient Parallel Implementation of Single Source Shortest Path Algorithm on GPU Using CUDA. International Journal of Applied Engineering Research. 2016; 11(4):2560–7.
40. Katz GJ, Kider JT. All-Pairs Shortest-Paths for Large Graphs on the GPU. Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware. 2007.
41. Matsumoto K, Nakasato N, Sedukhin SG. Blocked All-Pairs Shortest Paths Algorithm for Hybrid CPU-GPU System. Proceedings of the 13th IEEE International Conference on High Performance Computing and Communications. 2011.
42. Tran QN. Designing Efficient Many-Core Parallel Algorithms for All-Pairs Shortest-Paths Using CUDA. Proceedings of the Seventh International Conference on Information Technology: New Generations, 2010.
43. Buluc A, Gilbert JR, Budak C. Solving Path Problems on the GPU. Journal of Parallel Computing. 2010; 36(6):241–53.
44. Nickolls J, Buck I, Garland M, Skadron K. Scalable Parallel Programming with CUDA. ACM Queue. 2008; 6(2):40–53.
45. NVIDIA Corporation, CUDA C programming guide (2013). Available from: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz48cRjH1nP. Accessed date: 14 may 2016.