Parallel Implementations for Solving Shortest Path Problem using Bellman-Ford

Gaurav Hajela
Department of Computer Science and Engineering
Maulana Azad National Institute of Technology
Bhopal, India

Manish Pandey
Department of Computer Science and Engineering
Maulana Azad National Institute of Technology
Bhopal, India

ABSTRACT
In this paper, different parallel implementations of Bellman-Ford algorithm on GPU using OpenCL are presented. These variants include Bellman-Ford for solving single source shortest path (SSSP) having two variants and Bellman-Ford for all pair shortest path (APSP) problems. Also, a comparative analysis of their performances on CPU and GPU is discussed in this paper.Write-write consistency in Bellman-Ford is overcome using synchronization mechanism available in OpenCL and by explicit synchronization by modifying the algorithm.An average speed up of 13.8x for parallel bellman ford for SSSP and an average speed up of 18.5x for bellman ford for APSP is achieved by proposed algorithm.

Keywords
Shortest path problem , OpenCL , Graphical processing unit(GPU).

1. INTRODUCTION
Single source shortest path problem finds application in large domains of scientific and real world. Common applications of these algorithms are in network routing [6], VLSI design, robotics and transportation, they are also used for directions between physical locations like in google maps. Here all the applications mentioned generally involve positive weights but some applications are there where weights can be negative like currency exchange arbitrage and some other areas where, edge represents something other than merely distance between two entities. In such application areas Bellman-Ford algorithm can be used. Bellman-Ford algorithm[12] is applicable on graphs with negative weights and can also detect negative cycles where majority of algorithms fail. Bellman-Ford is also used in wireless sensor networks and other ad hoc networks as distributed Bellman Ford [7] can be used there. Distributed Bellman-Ford is also used as first ARPANET routing algorithm in 1969 [14].
Most of the above application areas specified are real time applications and need results in a quick time so the performance of algorithm need to be improved so that it consume less power and time. Parallel computing on GPU is one of the technologies which is used for high performance computing at a reasonable cost and considerable speed up of performance. GPU is currently used for a variety of purposes apart from graphical processing and gaming. That's why GPU is referred as General Purpose Graphical programming unit (GPGPU)[10] as it provides high performance computing can be programmed using standard frame work like OpenCL and CUDA. OpenCL [11] is a framework which is for all GPU while, CUDA is meant specifically for NVIDIA GPUs only. Thus, OpenCL is used for GPU implementation due to its portability and open-ness.

1.1 Bellman Ford Algorithm
Consider a graph G(n,E,V) where, n is the number of vertices, E is the set of edges and V is the set of vertices. Adjacency matrix representation of graph is used here, as it is well suited for GPU. Here, *Cost* is the adjacency matrix for graph. Initially, *Dist* will contain direct edges from the source 's'. Afterwards, *Dist[v]* of 'k^{th}' iteration means distance from 's' to 'v' going through no more than 'k' intermediate edges. Finally, after successful completion of algorithm *Dist* will contain the shortest path to all the vertices 'v' in V from source 's'. For each edge(u,v) in set E, Relax(u,v) is called (n-1) times. So, Relax() is called E(n-1) times, thus majority of time of the algorithm is spent in this procedure. The algorithm for Bellman Ford is illustrated in Algorithm 1.

```
Algorithm BellmanFord (s,Dist,Cost,n)
{
    1.  for i=1 to n do
    2.      Dist[i] = Cost[s,i];
    3.  End for
    4.      for k=1 to n-1 do
    5.          for each (u,v) in E do
    6.              Relax(u,v)
    7.          End for
    8.      End for

}

Relax (u,v)
{
    1.              if Dist[v]> Dist[u] + Cost[u,v]
    2.                  Dist[v] = Dist[u] + Cost[u,v]

}
```

Algorithm 1: Algorithm for Bellman-Ford.

Time complexity of above algorithm if adjacency matrix representation is used will be $O(n^3)$.
All pair shortest path using bellman ford algorithm could also be calculated if above algorithm for all the vertices in the graph is called.

```
For each s in V
  Call BellmanFord(s,Dist,Cost,n);
End for
```

Organization of the paper: In Section 2, the previous modified algorithms have been discussed along with the improvements made on Bellman Ford algorithm by different authors. In Section 3, identified parallelism in standard Bellman Ford algorithm and other write-write conflict issues in parallelization of the algorithm are presented. In Section 4, proposed parallel algorithm along with OpenCL kernel is

explained for both SSSP and APSP. Comparative analysis and results are shown in Section 5.

2. RELATED WORK

Bellman Ford is introduced by Richard Bellman and Lester Ford Jr. in 1958 since then several modifications and improvements were made on this algorithm. One of the famous modifications include Yen's modification in 1970 [5].Other modifications include topological scan algorithm for Bellman Ford [2] in 1993, which outperforms the standard algorithm in most of the cases. A hybrid implementation of Bellman Ford and Dijkstra's algorithm is given which is asymptotically better than Bellman Ford in [7]. In 2001, A.S. Nepomniaschaya presented a STAR procedure for Bellman Ford on a parallel system with vertical data processing (STAR- machine) [3] and managed to reduce the complexity to O(n2). In 2011, Michael J. Bannister and David Eppstein [1] proposed a randomized variant of algorithm which is improved by a factor of 2/3 over Yen's modification(1970) [4,5]; they have termed this speedup as randomized speedup.

Several parallel implementations on GPU for SSSP algorithms were proposed. Aydın Buluc, John R. Gilbert and Ceren Budak [8] have proposed parallel implementations for SSSP and APSP using CUDA. A CUDA implementation for Bellman_Ford is given in [13] and by making algorithm suitable for parallelism they have got speedup of about 10x.

Recently, Andrew Davidson [9] have presented several work efficient methods for SSSP problems and got considerable speedup over serial implementation and other traditional GPU implementations also.

In this paper parallel implementation of Bellman Ford for SSSP and APSP using OpenCL are proposed and comparison between implicit synchronization mechanism provided by OpenCL with explicit synchronization is done and also, a comparative analysis of speedup with serial implementations has been given as well.

3. IDENTIFIED PARALLELISM IN BELLMAN FORD ALGORITHM AND OTHER ISSUES

There is inherent parallelism in standard Bellman Ford algorithm which lies in Relax() procedure. If Relax() is called for all the edges in E in parallel then performance can be increased considerably. For this lets re-write Relax(u,v) as:

Relax(u,v)
{
$Dist^{k}[v] = min\ (Dist^{k-1}[v], Dist^{k-1}[u] + Cost[u,v])$
}

As k^{th} value of Dist depends on k-1 iteration value so, outer loop can't be removed. All the parallelism which is possible is in Relax() procedure. Two levels of parallelism are possible here :

First: In k^{th} iteration value of $Dist^{k}[v_1]$ and $Dist^{k}[v_2]$ doesn't depends on each other for any v_1 and v_2 in set V.

Second: For all the u in set V $Dist^{k-1}[u] + cost[u,v]$ can be calculated in parallel as this also doesn't depends on each other. The only issue arises here is how to calculate minimum of all these 'n' values. So rather than calculating the minimum which will increase the time of algorithm its better to synchronize the write operations on $Dist^{k}[v]$ for all 'u' such that minimum value resides in $Dist^{k}[v]$ at the and of Relax() procedure. This issue is referred as write-write consistency.

Another thing which is a vital factor in algorithm is space used for *Dist* matrix. As there are n vertices in G and in every iteration previous values are accessed so *Dist[2][n]* is used instead of *Dist[n-1][n]*. The row to be accessed will be adjusted according to the iteration, which is explained in detail in section 4.

4. PARALLEL BELLMAN FORD FOR SHORTEST PATH PROBLEMS

4.1 Single Source Shortest Path (SSSP) Problem

In SSSP shortest path from a source say, 's' to all other vertices in a graph are calculated. For parallel implementation host algorithm is given in Algorithm 2 and kernel algorithm in Algorithm 3. As there are n vertices in graph and graph is represented by adjacency matrix there can be n*n possible pairs of (u,v) where u and v ϵ V.
So here for SSSP workgroup of size {n,n} if formed so that each work item will represent a pair (u,v). And kernel will be called for all the work items in work group in parallel.The outer loop of n-1 iterations is implemented on host side algorithm.

```
Algorithm OpenCL_Parallel_BellmanFord_SSSP
{
    1. For k from 1 to n-1 do
    2.   For all v in V such that (u,v) belongs to E in
         parallel do
    3.       Call KERNEL_BELLMAN_SSSP(Cost,Dist,k)
    4.   End for
    5. End for

}
```

Algorithm 2: Algorithm for host code of Bellman Ford for SSSP

As there will be n*n work items invoked in parallel the *id* of work item which will be unique for every work item in a work group is captured. Initially, first row of *Dist* will contain the direct edges from source 's'; in first iteration values are updated in second row. In next iteration second row values are $Dist^{1}$; and $Dist^{2}$ are updated in first row. So odd iteration will read from first row (*Dist[0][v]*) and update in second row (*Dist[1][v]*)) and opposite will be the case for even iteration.

```
KERNEL_BELLMAN_SSSP(Cost,Dist,k)
{
  u = get_global_id(0)
  v = get_global_id(1)

  if k is odd then
     // synchronization is done here
     Dist[1][v] = min(Dist[0][v], combine(Dist[0][u] +
Cost[u][v]))

  if k is even then
     // synchronization is done here
     Dist[0][v] = min(Dist[1][v], combine(Dist[1][u] +
Cost[u][v]))


}
```

Algorithm 3: Algorithm for kernel of Bellman Ford for SSSP

Combine function is simply to perform the addition of two values it takes in 2 arguments and return the addition if both are not infinity and returns infinity if any one of the value is infinity. As all the values will be of integer type, INT_MAX is considered as infinity.

Synchronization: As explained above, synchronization is needed to bring write-write consistency, two types of synchronizations have been used:

First: OpenCL provides some synchronization mechanism for both host side and kernel side. Here barrier function (CL_GLOBAL_MEM_FENCE) is used on kernel side so as to order the read and write operations to and from global memory [13].

Second: Here, work items with id (*, v) where * will vary from 0 to n-1 will have write-write conflict for a particular 'v'. Work items with different value of 'v' for same 'k' won't have any conflict and wont effect each other and need not to be synchronized. But Barrier function can't be applied to selected work items in a work group so all work items need to wait on barrier function which will result in unnecessary slowness of algorithm. This drawback is overcome using explicit synchronization mechanism using a array for every 'v'.

4.1.1 Explicit synchronization mechanism for SSSP implementation

All the work items having id (*, v) will share a array in local memory of size n. And value of v will also vary from 0 to n-1 Each work item before comparing its values with *Dist[][v]* value will check whether the work item with id one less than it has updated its value or not. Thus they all will write in a particular sequence and remove write-write conflict in Bellman-Ford.So using this only work items having same 'v' will wait for synchronization so overall time used in synchronization will be reduced.

So, host code will be same for this implementation also except one more matrix will be passed which is Syn[n][n] used for synchronization and kernel algorithm is shown in Algorithm 4. Initially all the elements of the matrix Syn will be zero. In addition to Syn all the work items will have one more variable temp which is in private memory for every work item.

```
KERNEL_BELLMAN_SSSP_SYN(Cost,Dist,Syn,k)
{
   u = get_global_id(0)
   v = get_global_id(1)

  temp =  combine(Dist[0][u] + Cost[u][v])

   if k is odd then
       Untill (Syn[v][u-1] == 1)
            Dist[1][v] = min(Dist[0][v], temp)

   if k is even  then
        Untill (Syn[v][u-1] == 1)
      Dist[0][v] = min(Dist[1][v], combine(Dist[1][u] +
Cost[u][v]))

 }
```

Algorithm 4: Algorithm for kernel of Bellman Ford for SSSP using explicit synchronization

Here, there is no restriction on combine operation, so work items will wait only to update the value and will take less time as its has only one compare operation to do.

4.2 All Pair Shortest Path (APSP) Problem

For APSP using Bellman Ford 3D work group is used as SSSP is applied for all the vertices in graph to calculate APSP. So previous work group size will be increased by a factor 'n' and final work group size becomes {n,n,n}. Host algorithm is same but instead of calling kernel for a single source here it is called for all the vertices. So n*n*n work items will execute in parallel. Host algorithm for APSP is shown in Algorithm 5 and kernel algorithm is shown in Algorithm 6.

```
 Algorithm OpenCL_Parallel_BellmanFord_APSP
{
1.   For k  from 1 to n-1 do
2.     For all v in G such that (u,v) belongs to G in parallel
       do
3.     Call KERNEL_BELLMAN_APSP(Cost,Dist,k)
4.     End for
5.   End for
 }
```

Algorithm 5: Algorithm for host code of Bellman Ford for APSP

As for every source 2*n matrix is needed for *Dist* and here every vertex will be source for its n*n threads for computing APSP .So overall 2*n*n matrix will be needed in combine for all vertices. First two rows will be for vertex number '0' next will be for '1' and so on. So a variable *offset* is used to point to correct row corresponding to vertex 's' which is acting as source.

```
KERNEL_BELLMAN_APSP(Cost,Dist,k)
{
   u = get_global_id(0)
   v = get_global_id(1)
   s = get_global_id(2)
  offset = s*2*n
   if k is odd then
       // synchronization is done here
     Dist[1][v] = min(Dist(offset + [0][v]),
combine(Dist(offset + [0][u]) + Cost[u][v]))

   if k is even  then
       // synchronization is done here
      Dist[0][v] = min(Dist(offset + [1][v]),
combine(Dist(offset + [1][u]) + Cost[u][v]))
}
```

Algorithm 6: Algorithm for kernel of Bellman Ford for APSP

Table 1: Speedup comparison with respect to serial implementation on specified CPU for SSSP.

NO. OF VERTICES	PARALLEL _CPU	PARALLEL _EXPLICIT _SYN (CPU)	PARALLEL _GPU
64	3	4	12
128	2.451	3.619	10.857
256	3.449	4.535	8.827
512	3.683	6.442	16.389
1024	3.755	6.718	17.378
2048	3.671	6.875	17.611
AVERAGE	3.335	5.365	13.843

5. COMPARATIVE ANALYSIS AND RESULTS

All the OpenCL parallel implementations are tested on following GPU and CPU:

AMD Radeon HD 6450(GPU): 2 Compute units, 625 MHz clock, 2048MB Global Mem., 32KB Local Mem., 256 work group size on a system having Intel Core i5 CPU 650 @ 3.2 GHz and 2048MB RAM with AMD APP SDK v2.8.

Implementations are done using using Visual Studio 2010 with OpenCL SDK 1.2. Serial implementations are tested on the above specified CPU. All other implementations are tested on randomly generated graphs having edge weights between -10 to 10 where bellman ford has successfully detected negative cycles if present in graph. Rest of the results are taken on graphs without negative cycle in which only kernel execution time is considered.

Comparative analysis of execution time of different variants of algorithm is shown in Figure 1 and Figure 2. For parallel implementations only kernel execution time is considered. Speedup of different variants for SSSP and APSP is shown in Table 1 and Table 2 respectively.

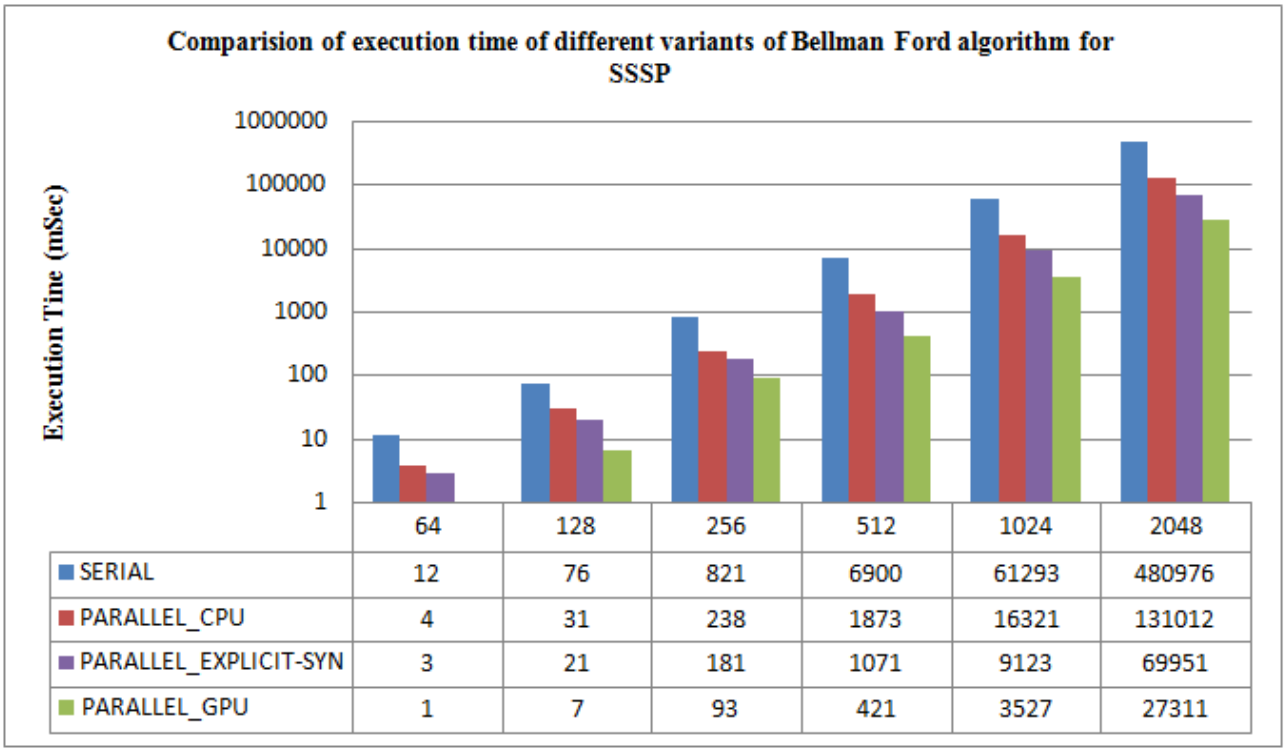
Our explicit synchronization mechanism provide a speedup of 1.58 over synchronization mechanism provided by OpenCL.

Table 2: Speedup comparison with respect to serial implementation on specified CPU for APSP.

NO. OF VERTICES	PARALLEL_CPU	PARALLEL_GPU
64	4.218	12
128	5.103	14.933
256	5.292	17.406
512	6.100	22.101
1024	7.300	26.300
AVERAGE	5.602	18.548

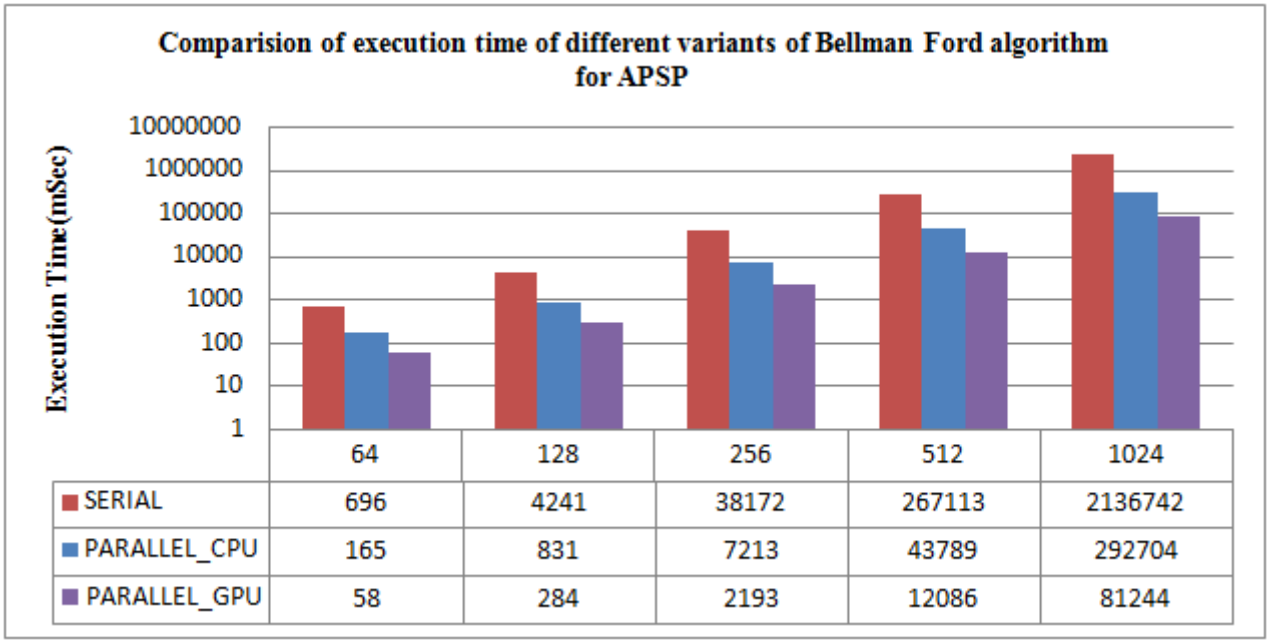
6. CONCLUSION AND FUTURE WORK

Among all the variants of Bellman Ford parallel implementation on GPU for SSSP and APSP has got considerable speed up of 13.8x and 18.5x respectively.We will look for hybrid implementation of parallel Bellman Ford on CPU and GPU by carefully partitioning the algorithm and dividing the work on CPU and GPU in a proportion so both take equivalent amount of time and also apply some optimization techniques on parallel implementations like vectorization to get further speedup.



	64	128	256	512	1024	2048
SERIAL	12	76	821	6900	61293	480976
PARALLEL_CPU	4	31	238	1873	16321	131012
PARALLEL_EXPLICIT-SYN	3	21	181	1071	9123	69951
PARALLEL_GPU	1	7	93	421	3527	27311

Figure 1: Comparative analysis of execution time of different variants of Bellman Ford for SSSP



	64	128	256	512	1024
SERIAL	696	4241	38172	267113	2136742
PARALLEL_CPU	165	831	7213	43789	292704
PARALLEL_GPU	58	284	2193	12086	81244

Figure 2: Comparative analysis of execution time of different variants of Bellman Ford for APSP.

7. REFERENCES

[1] Michael J. Bannister and David Eppstein , *"Randomized Speedup of the Bellman Ford Algorithm"* in arXiv:1111.5414v1 [cs.DS] 23 Nov 2011.

[2] Andrew V. Goldberg, Tomasz Radzik , *A Heuristic improvement of the Bellman Ford algorithm. Appl. Math. Lett.* Vol. 6, No. 3, pp. 3-6, 1993.

[3] A.S. Nepomniaschaya, *An Associative Version of the Bellman-Ford Algorithm for Finding the Shortest Paths*

in Directed Graphs, V. Malyshkin (Ed.): PaCT 2001, LNCS 2127, pp. 285–292, 2001.

[4] J. Y. Yen., *An algorithm for finding shortest routes from all source nodes to a given destination in general networks*. Quarterly of Applied Mathematics 27:526-530, 1970.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Problem 24-1: *Yen's improvement to Bellman Ford*. Introduction to Algorithms, 2nd edition, pp. 614-615. MIT Press, 2001.

[6] R. Bellman. On a routing problem. Quarterly of Applied Mathematics 16:87-90,1958.

[7] Yefim Dinitz , Rotem Itzhak , *Hybrid Bellman-Ford-Dijkstra Algorithm.*

[8] Aydın Buluc , John R. Gilbert and Ceren Budak , "*Solving Path Problems on the GPU" , Journal Parallel Computing Volume 36 Issue 5-6, June,2010 Pages 241-253.*

[9] Andrew Davidson , Sean Baxter, Michael Garland , John D. Owens , "*Work-Efficient Parallel GPU Methods for Single-Source Shortest Path " in International Parallel and Distributed Processing Symposium, 2014*

[10] Owens J.D., Davis, Houston, M., Luebke, D., Green, S., "*GPU Computing*", in: Proceedings of the IEEE, Volume: 96 , Issue: 5 , 2008.

[11] A. Munshi, B. R. Gaster, T.G. Mattson, J. Fung, D. Ginsburg, "*OpenCL Programming Guide*", Addison-Wesley pub., 2011.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, Second Edition. The MIT Press, Sep. 2001.

[13] Kumar, S.; Misra, A.; Tomar, R.S. ,"A modified parallel approach to Single Source Shortest Path Problem for massively dense graphs using CUDA" in *Computer and Communication Technology (ICCCT), 2011 2nd International Conference on* , vol., no., pp.635,639, 15-17 Sept. 2011.

[14] Atul Khanna, John Zinky , "The Revised ARPANET Routing Metric", in 1969 ACM.