International Journal of Computer Applications (0975 – 8887)
Volume 72– No.18, June 2013

New Approach for Graph Algorithms on GPU using CUDA

[1]Gunjan Singla, [2]Amrita Tiwari, [3]Dhirendra Pratap Singh
Department of Computer Science and Engineering
Maulana Azad National Institute of Technology
Bhopal, Madhya Pradesh

ABSTRACT

Large Graph algorithms like Breadth-First Search (BFS), Depth-First Search(DFS), shortest path algorithms etc. used frequently in various engineering and real world applications that demand execution of these algorithms in large graphs having millions of edges and sequential implementation of these algorithms takes large amount of time. Today's Graphics Processing Units (GPUs) provide a platform to implement such applications with high computation power and massively multithreaded architecture at low price. In this paper, we present parallel implementations of two basic graph algorithms breadth-first search and Dijkstra's single source shortest path algorithm by using a new approach called edge based kernel execution on GPU. The performance analysis of parallel implementation over the serial execution gives a good speed-up.

Keywords

SSSP (Single Source Shortest Path) problem, Dijkstra's algorithm, BFS (Breadth -First Search), CUDA (Compute Unified Device Architecture) model, GPU(Graphic Processing Unit).

1. INTRODUCTION

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

With the development of computer and information technology, researches on graph algorithms get wide attention. In particular, the Single Source Shortest Path (SSSP) problem is a major problem in graph theory which computes the weight of the shortest path from a source vertex to all other vertices in a weighted directed graph. The most well-known algorithm for solving this problem was given by Dijkstra in 1959 [4] with non-negative edge weights and further, more work is done considering it as base algorithm. So far, many different variants of Dijkstra's algorithm have implemented sequentially as well as in parallel manner. In all parallel implementations, a thread corresponds to a node in graph database but in our implementation, a thread corresponds to edges and as number of edges is greater than number of nodes, comparatively more degree of parallelism is achieved.

We have also given parallel implementation of BFS [5] [6] on the basis of edges as it is one of the basic paradigm for the design of efficient graph algorithm and hence, requires high degree of parallelism. Given a graph G= (V, E) with m edges, n vertices and a source vertex s, BFS traverses the edges of G to discover every vertex that is reachable from s.

At present, the serial graph algorithms have reached the time limitation as they used to take a large amount of time. Therefore, the parallel computation is an efficient way to improve the performance by applying some constraints on the data and taking the advantage of the hardware available currently.

Different implementations of parallel algorithms for the SSSP problem are reviewed in [7]. Bader et al. [8], [9] use CRAY supercomputer to perform BFS and single pair shortest path on very large graphs. A. Crauser et al. [10] have given a PRAM implementation of Dijkstra's algorithm while such methods are fast, hardware used in them is very expensive. N. Jasika et al. [11] presented a parallel dijkstra's algorithm using OpenMP (Open Multi-Processing) and OpenCL (Open Computing Language) which gives good results over serial algorithm. Pedro J. Martín et al. [12] have given an efficient parallel dijkstra's algorithm on GPU using CUDA. L. Luo et al. [13] have given a GPU implementation of BFS which gives around 10X speed-up over the algorithm given by P. Harish et al. [14].

In this paper, we present new edge based parallel implementations of Dijkstra's algorithm and breadth first search (BFS) on GPU using CUDA handling large graphs up to 2 million edges. We show the results for the speed-up obtained by our parallel algorithm over its serial execution.

The rest of the paper is organized as follows: CUDA basics along with GPU architecture is discussed in Section 2. Graph representation used by our implementation is discussed in Section 3. Section 4 presents edge based parallel Dijkstra's algorithm with a subsequent edge based parallel BFS implementation in section 5. Performance analysis of our implementation on various types of graphs is done in section 6 and finally concluded in section 7.

2. CUDA MODEL ON GPU

Graphics Processing Unit (GPU) was introduced by NVIDIA and has four types of memory in it i.e. *shared memory, constant memory, texture memory* and *global memory*. Its design does not have any memory restrictions as one can access all these memory available on the device except shared memory with no restrictions on its representation though the access times may differ for different types of memory. It uses a massively multithreaded computing architecture called CUDA for parallel processing of data. In CUDA programming model, GPU is referred as *device* and CPU is referred as *host.* Basically, CUDA device is a multi-core co-

processor that is used only for the computational task and cannot initiate the task without CPU. Each multiprocessor has a number of cores which execute in *Single Instruction Multiple Data* (SIMD) manner. In this model, execution is done on *threads*, a set of threads form a *block* and a group of blocks that executes same instruction set constitutes one *grid*. Each thread and block is given its own unique ID by which they are referred, named as threadID and blockID. Each thread executes a single instruction set called the *kernel* which is a part of code which defines the task and is executed on each thread by using threadID. The use of shared memory in this model improves performance of computation [15], [16].

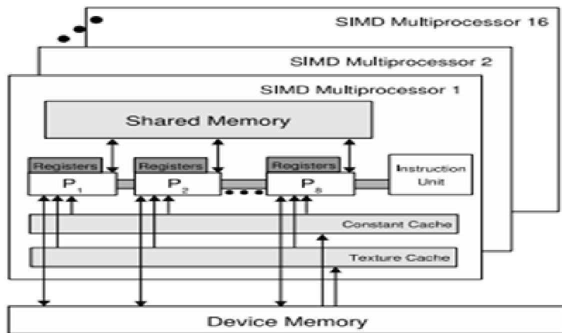

Figure 1: CUDA hardware interface [15]

3. GRAPH REPRESENTATION

In this paper, we have represented graph in the form of adjacency list as it takes less space as compared to adjacency matrix representation. We have stored the adjacency lists of all the nodes into a single large array. The data representation consists of node array V_a and edge array E_a, where each element in the V_a points to the starting index from E_a array of each edge and the array E_a stores the destination nodes of each edge. One extra element is needed in node array to indicate outdegree of last node as shown in Fig 1. Moreover, three arrays: edge start node array Sa stores the start node of each edge, edge end node array Ea stores the end node of each edge, edge weight array Wa stores the weight of each edge, are also required for parallel implementation such that each thread can run on edges rather than on nodes as in previous implementations.

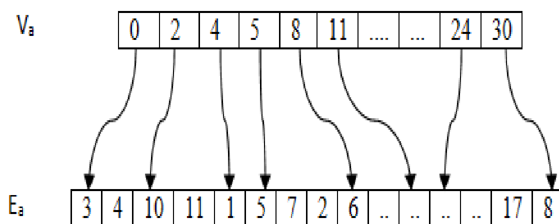

Figure 2: Graph representation

4. EDGE BASED PARALLEL DIJKSTRA'S ALGORITHM

4.1 Dijkstra's algorithm overview

Dijkstra's algorithm is inherently sequential and solves SSSP problem in $O(V \log V + E)$ time. In Dijkstra's algorithm, nodes are divided into two categories: settled and unsettled, settled nodes are those nodes having minimum node weight and whose outgoing edges are relaxed while unsettled nodes may be unreachable nodes i.e. Node weight = ∞ or nodes not having minimum node weight. Initially s is settled and its outgoing edges (s, u) $\forall u \in V$ are relaxed. In next iteration, node u having minimum node weight is settled and its outgoing edges (u, w) are relaxed. This procedure is repeated until all the nodes come to settled state.

4.2 Parallel Dijkstra's algorithm

In parallel implementation, number of threads is equal to number of edges in the graph. For each unsettled node, its outgoing edges are relaxed in parallel in order to reduce computation time. Initially, cost i.e. node weight of all the vertices is initialised to '∞' and mask is set to '0', except for the start node whose cost is initialised to '0' and mask value is set to '1'. Each time, from a settled node, we relax all unsettled nodes outgoing from settled node i.e. minimum cost is set for each unsettled node and then, threshold value 'minimum' is updated with minimum cost among all unsettled nodes. Edge based parallel Dijkstra's algorithm is given in Figure 3.

EDGE_DIJ (Graph G (V, E, W), Source Vertex S)

Create start node array S_a, end node array E_a, weight array W_a, node weight array N_a from G (V, E, W);
Create outdegree array D_a, Mask array M_a and minimum variable;
Initialise $M_a[v]$ to 0 and $N_a[v]$ to ∞ for every v ∈ V
Set $N_a[S]$ to 0, $M_a[S]$ to 1 and minimum to 0;
for each vertex v ∈ $S_a[i]$ for edge index i if $M_a[v]$ not equals to 1 do in parallel
set minimum equal to minimum node weight;
end for
for each vertex u ∈ $S_a[i]$ for edge index i having $N_a[u]$ equal to minimum and $M_a[u]$ not equal to 1 do in parallel
Invoke RELAX(M_a, N_a, Ea, W_a, D_a) kernel;
end for

In kernel RELAX as shown in Figure 4, if a node is *unsettled* and its node weight is minimum, then its outgoing edges should be relaxed and node weight of adjacent nodes should be updated.

RELAX (M_a, N_a, Ea, W_a, D_a)

Set $M_a[u]$ equal to 1;
While $D_a[u]$ is greater than 0 do
for each vertex v ∈ $E_a[i]$ for edge index i
if $N_a[v]$ is greater than $N_a[u]$ + $W_a[i]$ then $N_a[v]$ equal to $N_a[u]$ + $W_a[i]$;
end for
end while

Figure 4: Kernal Relax

5. EDGE BASED PARALLEL BFS IMPLEMENTATION

5.1 BFS overview

BFS is a graph traversal technique to discover every node that is reachable from starting node level by level and uses queue for traversing all the nodes of the graph G. In this, first we take any node s as a starting node, then, we take all the nodes adjacent to s. Similar approach we take for all other adjacent nodes, which are adjacent to s and so on. We maintain the status of visited node in one array so that no node can be traversed again. At the end, it produces a breadth first tree rooted with s containing all the vertices reachable from s. It works in the way such that all vertices at level k are first visited, before traversing any vertices at level k+1.

5.2 Parallel BFS algorithm

In edge based parallel BFS implementation using CUDA, number of threads is equal to the number of edges in the graph. We have created a flag variable to denote whether an updation in visited array occurred or not and if yes, flag is set to true. First, source node s is marked as visited node then its adjacent nodes are marked during graph traversal level by level. Whenever there is no updation, program exits, hence making the program efficient. Edge based parallel BFS is shown in Figure 5.

EDGE_BFS (Graph G (V, E, W), Source Vertex S)

Create start node S_a, end node array E_a, and weight array W_a from G (V, E, W) Create boolean flag F variable initialised to true and visited array V_a
Initialise F and $V_a[S]$ to true;
While F is true
    Invoke SEARCH (V_a, S_a, E_a, F) for each edge in parallel
end while

Figure 5: Edge based parallel BFS algorithm

SEARCH (V_a, S_a, E_a, F)

for each vertex u $\in$ $S_a[i]$ for edge index i if $V_a[u]$ is true do
    for each vertex v $\in$ $S_a[i]$ for edge index i if $V_a[v]$ is false do
        set $V_a[v]$ to true;
        set flag to true
    end for
end for

Figure 6: Kernal search

In kernel SEARCH (V_a, S_a, E_a, F) as shown in Figure 6, if status of start node of an edge is *visited* then we are setting status of its all end node to *visited* in parallel. At each update, flag value is set to true so that the algorithm continues and if there is no change in flag value then algorithm exits. Three factors which make it better than sequential BFS are: first, nodes at a particular level traverse next level node in parallel manner, second, early exit of program rather than looping for every node, third, threads run corresponding to edges achieving high degree of parallelism.

6. PERFORMANCE ANALYSIS

We have evaluated the performance of edge based parallel Dijkstra's algorithm and edge based parallel BFS algorithm on a large range of graph statistical data such as sparse, general, almost complete directed graphs of 6 thousands to 0.2 million vertices having up to 2 million edges. We will compare the execution times of these algorithms over their respective serial algorithms.

6.1 Experimental Setup

We have used two Desktop PCs to evaluate the results of edge based parallel Dijkstra's algorithm and edge based parallel BFS algorithm with their respective sequential versions.

Experimental Setup 1:

- CUDA 4.1
- NVIDIA GeForce GTS 450 GPU
- Compute Capability 2.1
- 192 Cores and 4 Multiprocessors
- 1 GB Dedicated GPU Memory
- Intel Core i5 CPU @ 3.20 Ghz
- 2 GB RAM
- Windows 7 Professional x86
- Visual Studio Professional 2008

Experimental Setup 2:

- CUDA 5.0
- NVIDIA Tesla C2075 GPU
- Compute Capability 2.0
- 448 Cores and 14 Multiprocessors
- 4 GB Dedicated GPU Memory
- Intel Xeon CPU @
- 24 GB RAM
- Windows 7 Professional 64-bit OS
- Visual Studio Professional 2010

The results for edge based Dijkstra's algorithm are summarized in Figure 7 for NVIDIA GeForce Graphic card. Experimental results show that GPU implementation achieves a significant speed-up of up to 5 times over serial implementation. For a graph of 30K vertices and 90K edges, it computes SSSP in about 7 milliseconds.

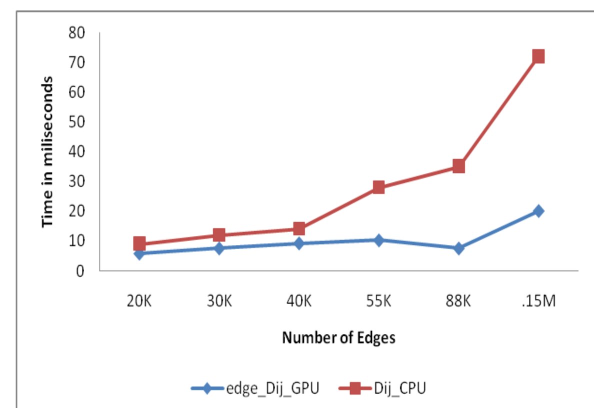

Figure 7: Comparison of parallel Edge based Dijkstra's algorithm with serial Dijkstra's algorithm on the basis of edges on Setup 1.Here, edge_Dij_GPU refers to parallel edge based implementation of Dijkstra's algorithm and Dij_CPU refers to serial Dijkstra's algorithm.

Figure 8 summarizes comparison of parallel edge based implementation of Breadth First Search with serial BFS on NVIDIA GeForce GTS Graphic card. As can be seen from figure that implementation of BFS over GPU reduces considerable amount of execution time over CPU. For 1,50,000 edges, it takes just 9 seconds.

For larger graphs, it can show better speed-up but due to the restriction of memory on the CUDA device and host CPU, graphs above 2 million vertices cannot be handled using current GPUs.

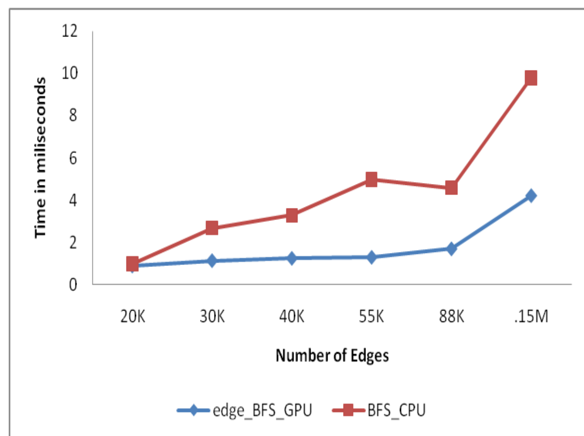

Figure 8: Comparison of parallel Edge based BFS with serial BFS algorithm on varying number of edges on Setup 1. Here, edge_BFS_GPU refers to parallel implementation of BFS and BFS_CPU refers to serial implementation.

Figure 9 shows comparison of parallel edge based Dijkstra's algorithm with corresponding sequential implementation on NVIDIA Tesla Graphic Card as depicted in Experimental Setup 2. As can be seen from Figure 9, with the increase of number of edges in the graph, speed-up also increases. Initially, the gap between two curves, depicting parallel and sequential implementations respectively, is less, but with the increase of graph size, gap widens. For a graph having 23,12,777 edges, parallel implementation is taking just 0.5 seconds while, sequential implementation is taking approximately 2 seconds.

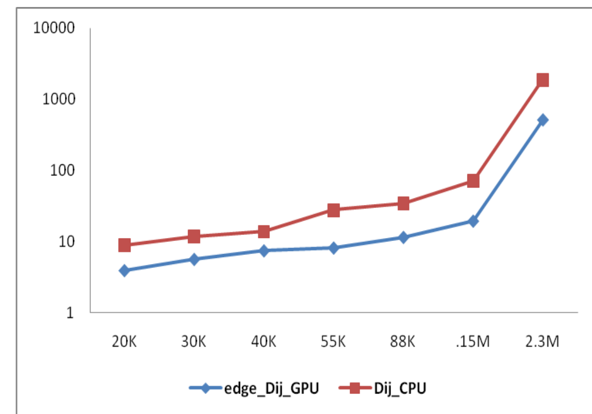

Figure 9: Comparison of parallel edge based Dijkstra's algorithm with serial Dijkstra's algorithm based on edges on Setup 2. Here, edge_Dij_GPU refers to edge based parallel implemenation of Dijkstra's algorithm and Dij_CPU refers to sequential implementation.

In Figure 10, comparison of edge based parallel implementation of Breadth First Search with sequential Breadth First Search is represented. For a graph having 20,000 edges, parallel implementation is showing a speed-up of just 1.5 times, but for a graph having 23,12,777 edges, it is showing a speed-up of up to 11 times. So, for larger graphs, It can show more speed-up.

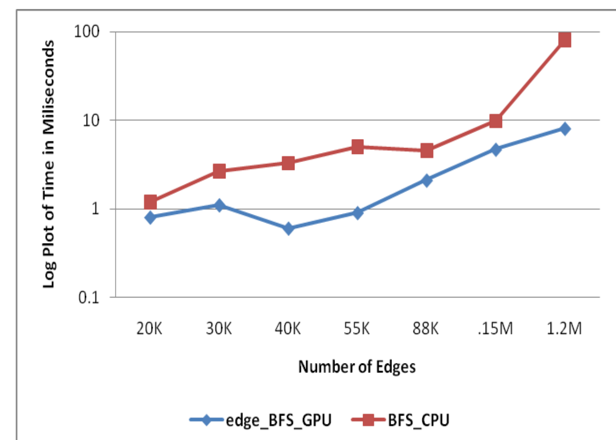

Figure 10: Comparison of parallel edge based BFS algorithm with serial BFS algorithm based on edges on Setup 2. Here, edge_BFS_GPU refers to edge based parallel implemenation of BFS algorithm and BFS_CPU refers to sequential implementation.

7. CONCLUSION

In this paper, we presented a new method to perform graph operations such that shortest path computation by Dijkstra’s algorithm and graph traversal by BFS,using edges. As number of edges is greater than number of nodes and threads are mapped to edges rather than nodes, greater degree of parallelism can be achieved. Achieved results show a considerable speed-up over corresponding serial implementation for various graph instances having varying degrees.

8. REFERENCES

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

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

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

[4] Dijkstra E.W. 1959. A note on two problems in connexion with graphs. Num. Math. 1, pp. 269–271

[5] Quinn M.J. and Deo N. 1984. Parallel graph algorithms. ACM Comput. Surv., 16(3), pp. 319-348.

[6] Reghbati A.E. and Corneil D.G. 1978. Parallel computations in graph theory. SIAM Journal of Computing, 2(2): pp. 230-237.

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

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

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

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

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

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

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

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

[15] Nvidia, CUDA programming guide version 3, at developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide.pdf (2010).

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