BFS graph traversal algorithm using CUDA

GACS-7306-001

Applied Parallel Programming

submitted to

Dr. Christopher Henry

in partial fulfilment for the degree of Master of Science

By

Hardeep Singh

3089450

Singh-h18@webmail.uwinnipeg.ca

December 11th, 2017

BFS graph traversal algorithm using CUDA

Hardeep Singh

140 Margate Road, R2P 1B2
Winnipeg, MB
Singh-h18@webmail.uwinnipeg.ca

Abstract. Breadth first search is backbone of many real world applications like social networking websites, GPS navigation systems, finding shortest path between two nodes and minimum spanning tree. These real world applications requires very fast data processing speed in order to meet needs of today's digital world and for getting efficient performance of all these applications needs fast implementation of graph processing. In this paper, we have discussed how parallel breadth first search technique can be way faster than the sequential BFS algorithm. We have used graphic processing unit (GPU) to run parallel code instead of CPU because GPUs have higher computational power. Moreover, we have used level synchronous parallel algorithm which traversed graph vertices in levels where sequential algorithm uses queue to store graph data and processes one node at a time. The language used in this paper to implement parallel algorithm is CUDA C. The experiments are done with different number of graph nodes and speed factor is calculated.

Keywords: Breadth first search, Graph. GPU, CPU, CUDA, compact adjacency list.

1 Introduction

Graph theory is an important part of data structure and Mathematics, which studies and play an effective role in pair wise relation between elements of set of objects. Graph is a collection of vertices and these vertices have connection through edges. Graphs are represented through *G (V, E).* Where *V* is the set of vertices or nodes and *E* is the set of edges. Breadth First Search, BFS is a graph search and traversal technique. It starts from the source node or root node and explores neighbor nodes first before going to the next nodes. It uses queue (first in first out) data structure. It checks whether a vertex is discover or not before enqueueing the vertex. BFS is used to solve many problems in graph theory, for example finding shortest path between two vertices in terms of number of edges, testing bipartiteness of a graph. Other real life applications of BFS are Network analysis, image processing, graph partitioning etc. In a given graph *G (V, E)* and a source vertex *S*, the problem is to find minimum number of edges needed to reach every vertex *V* in graph *G* from source vertex *S*.

CUDA is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows Programmers to use CUDA-enabled graphics processing unit (GPU) for general purpose processing which is termed as General purpose

computing on Graphics Processing Units. CUDA is a software layer that gives direct access to GPU's virtual instructions and parallel computational elements for the execution of kernels. CUDA is designed to work with the languages like C++, C and FORTRAN. The BFS algorithm used in this paper is implemented with CUDA C.

A given graph, *G* is used as input to the BFS graph processing algorithm. The minimum cost or minimum number of edges is calculated which are required to reach every vertex from the source vertex. Level synchronization is used to solve BFS problem. Parallel BFS traverses the graph in levels, once a level is visited will not be visited again. One thread will be assigned to every vertex.

Nowadays, GPU's are widely used to run parallel algorithms [1]. Moreover, it is used to implement various parallel graph processing algorithms like Travelling salesman problem, Flyod Warshall algorithm, all pair shortest path and Single source shortest path.

Section 2 contains all the background information about graphs and graph representation. Moreover, related work is also discussed in this section. Theoretical framework is explained in the section 3. In addition, section 4 is focused on implementation details, algorithm description along with the code diagrams. Apart from this section 5 contains experiment results followed by the Analysis and conclusion in the section 6 and 7, respectively.

2 Background information

2.1 Graph representation

A graph data structure basically demonstrates the relationship between nodes or we can say entities. In real life, social media is very good example of graph, where users are nodes and relations are connection between them. Moreover, for location navigator applications, nodes are locations and streets are relation between them. The relationship between two nodes can be of two types bi-directional or directional. A connection between two different locations through two way street is an example of bi-directional, where connection through one-way street is an example of directional relation. For BFS, directional relation is used in this paper. It is represented by an arrow going from source vertex to the destination vertex. [6]

A graph representation is usually done by two ways, adjacency matrix and adjacency lists. Considering a graph in fig. 1 having directional edges. There are 5 nodes assigned with five different numbers.

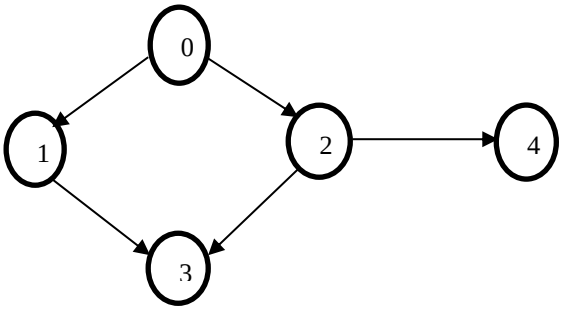

Fig. 1. Directed graph

Adjacency matrix representation:-
Adjacency matrix is basically a 2D array having size V x V, where V is the number of vertices or nodes in a graph. The graph in the fig.1 can be represented using 5 x 5 matrix as it have 5 vertices.

In Fig.2, adjacency matrix of given graph is illustrated. Let the 2D array be adj[][], where adj[i][j] is set to 1 when there is edge directed from vertex i to vertex j, otherwise set to 0.

	0	1	2	3	4
0	0	1	1	0	0
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	0
4	0	0	0	0	0

Fig. 2. Adjacency matrix representation of given graph

Advantages:-
Representation of this technique is easy to implement. Many graph queries like removing an edge takes O (1) time.
Disadvantages:-
It consumes more space as the graph is sparse, but allotted with more numbers of 0's in the matrix. For examples, many network graphs like Facebook and twitter are highly sparse, as connections for each user is much small and which leads to requirement of large size adjacency matrix.

Adjacency list representation:-
In this representation of graphs, an array of linked lists is used. Where size of array is set equal to the number of nodes in graph. An initial array adl[i] represents the linked list of vertices adjacent to the ith vertex.

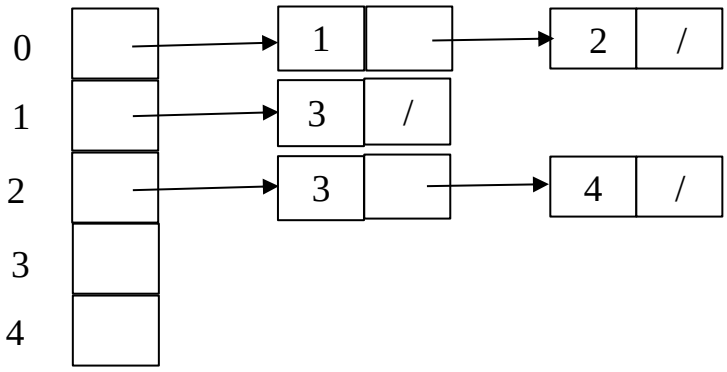

Fig. 3. Adjacency list representation

Fig.3 shows the adjacency list representation of the graph given in the Fig 1. Moreover, the graph representation used in this paper is compact adjacency list, which is discussed in the next section.

2.2 Sequential breadth first search

Sequential BFS uses queue data-structure to store vertices and then vertices are traversed using adjacency list. All the to-be-processed vertices are stored inside the queue and then algorithm dequeue them one-by-one. Process is start from the first vertex, all the adjacent vertices then enqueued and traversed one at a time. The complexity for BFS sequential algorithm is $O(V+E)$. Where V is the number of vertices and E is number of edges. In parallel BFS algorithm level traversing is used in this paper and how it works along with compact adjacency list is discussed in next section.

2.3 Related works

In past, many approaches were used to implement graph processing algorithms. [2] Sequential and parallel graph algorithms. The drawback of sequential algorithms is that they fails when it comes to large graphs. Then parallel algorithms came into scene and achieves faster practical running times. The only drawback of such algorithms was high hardware cost. Bader [3] performs graph algorithm by CRAY supercomputer, the computation running time was fast but hardware used was expensive

Modern day graph processing is done by GPU's by running parallel algorithm of graph traversal and search techniques. Harish and Narayannan proposed accelerated large graph algorithm using CUDA [4]. In their method, GPU implementation can handle large graphs. Moreover, one thread is allocated to every node.

Lijuan Luo [5] proposed implementation of BFS on GPU. A hierarchal technique is used to implemented queue data structure on GPU, where queue is used by BFS to store

their nodes. Lijuan Luo used hierarchal kernel arrangement to reduce synchronization overhead.

3 Theoretical framework

As we discussed in the previous section that sequential graph algorithm uses queue, adjacency matrix and adjacency list graph representation data structure. However, to implement parallel algorithm we have used compact adjacency list as graph representation data structure. We have used two arrays to store graph data using compact adjacency list. First is vertices [], which stores all the vertices of the graph and second is edges [], which stores all the edges for all vertices present in the graph. To understand it clearly we can take an example of a graph defined in fig.1.

The corresponding representation of the graph of fig 1 in terms of array is defined in the fig 4.

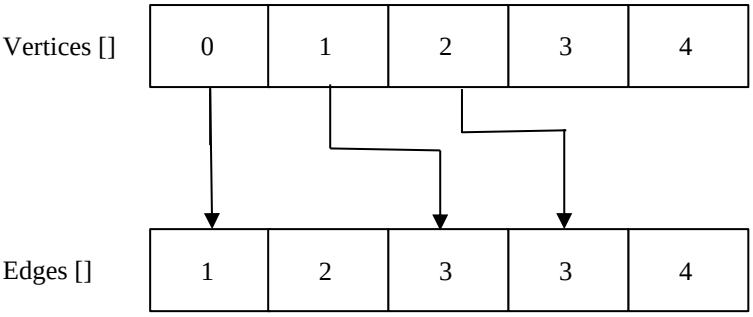

Fig. 4. . Array representation of graph (compact adjacency form)

In fig 4, vertices array contains all the vertices of the given graph in fig 1. In addition, edges array contains all the adjacent vertices of every vertex of the graph and pointer points to the first adjacent vertices. Each entry of edges [] refers to a vertex array vertices []. For example, o is the source vertex of the given graph and its adjacent nodes are 1 and 2 which are stored in array edges [], and pointer from 0 points to the first adjacent node 1.

In sequential bfs algorithm, we have to use queue to store each vertex but it results in additional overheads of maintaining array indices and changing configuration. To cover up this problem, Harish [4] have used level synchronization to traverse the graph in parallel breadth first search. For example in fig 1, level 1 contains 1 vertex which will be traversed first then the vertices adjacent to node 0 (1 and 2) will be traversed in parallel as level 2. This will continue until all levels are traversed. (Vertices in different levels: - level 1 = 0, level 2 = 1 and 2, level 3 = 3 and 4).

Flowchart for Cuda program:-

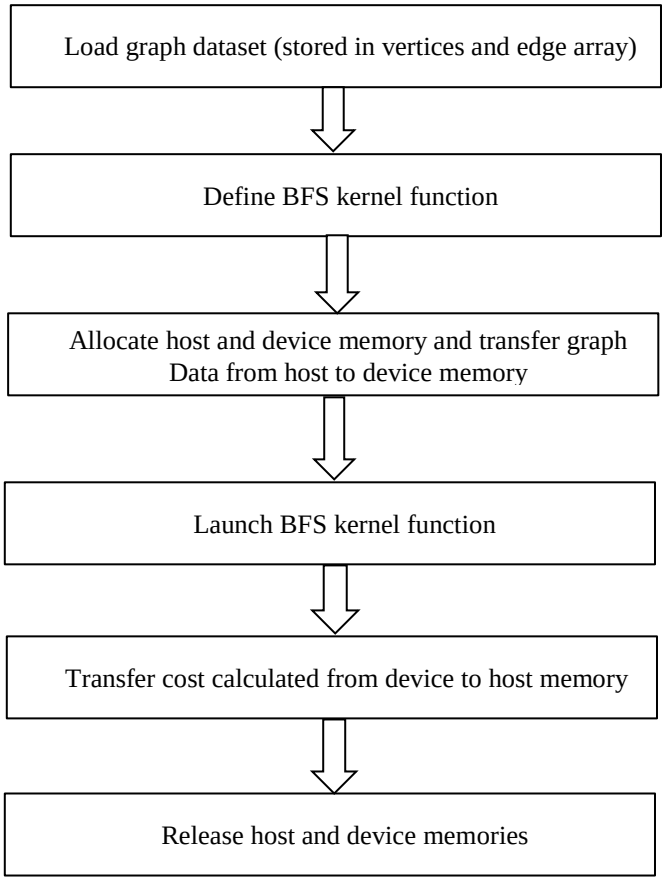

Fig. 5. Steps to be followed for CUDA program execution

4 Implementation

4.1 System description

Software architecture

a. NVidia Nsight Eclipse edition 7.5
b. CUDA Toolkit 7.5

Hardware architecture

a. NVIDIA GeForce GTX 780
b. Intel CORE i5
c. CUDA cores - 2304

Language used: - CUDA-C

4.2 Development environment

The BFS parallel algorithm is implemented by using NVIDIA Nsight development environment. It is the platform for heterogeneous computing. Which means developers can use this development environment to optimize the performance of both CPU and GPU. NVIDIA Nsight is a full featured IDE platform which provides all in one integrated environment to edit, build, debug CUDA-C, C++ and FORTRAN applications.

4.3 Algorithm Description

We have used the algorithm written by [harish] in his paper. Where graph traversal is done by using level synchronization. Which means graph is traversed in levels, Once a level is visited will not be visited again.
There will be 5 input arrays to the algorithm:-
Vertices array = contains all vertices of the graph (example defined in fig 4.)
Edges array = contains nodes adjacent to the vertices in vertices array.(fig 4.)
Frontier array = It is a Boolean array contains true value for every unprocessed nodes.
Visited array = It is also a Boolean array contains true value for all traversed nodes.
Cost array = It is an integer array contains number edges required for source vertex to Reach all other vertices.

Parallel algorithm

1. Begin

- Initialize frontier and visited array as false.
- Initialize cost array to ∞.
- Frontier[S] ⟵ true and Cost[S] ⟵ 0.
- While Frontier[S] not empty do

2. For each vertex in parallel do (Kernel algorithm)
   - Compute adjacent number of vertices.
   - Allocate thread to every vertex present in the level.
   - Check if vertex is not visited yet.
   - Cost[V] ⟵ Cost[V - 1] + 1.
   - Store traversed vertex in visited array.
     End for

End while

3. Gather and display results of all threads.

In the given parallel breadth first search algorithm, for every iteration, each vertex Look for its value in frontier array. If the value is true for that vertex, than the cost is updated to the cost array. After updating cost, the vertex removed from the frontier array and added to the visited array. Moreover, all the adjacent nodes to the recently traversed vertex added to the frontier array. This process will repeated until frontier array is empty. [4].

Complexity analysis

The time complexity of parallel BFS algorithm using level synchronization is $O(D)$, where D is the diameter of the graph and the work complexity is $O(m+n)$, where m is the number accesses to an array and n is the number of elements in an array.

4.4 Code diagrams

```
int source = 0;
h_frontier[source] = true;
int num_blocks = 1;
    int num_of_threads_per_block = number_of_nodes;


    if(num_blocks>MAX_THREADS_PER_BLOCK)
    {
        num_blocks = (int)ceil(number_of_nodes/(double)MAX_THREADS_PER_BLOCK);
        num_of_threads_per_block = MAX_THREADS_PER_BLOCK;
    }
Node* d_vertices;
cudaMalloc((void**)&d_vertices, sizeof(Node)*number_of_nodes);
cudaMemcpy(d_vertices, vertices, sizeof(Node)*number_of_nodes, cudaMemcpyHostToDevice);

int* d_edges;
cudaMalloc((void**)&d_edges, sizeof(Node)*number_of_nodes);
cudaMemcpy(d_edges, edges, sizeof(Node)*number_of_nodes, cudaMemcpyHostToDevice);

bool* d_frontier;
cudaMalloc((void**)&d_frontier, sizeof(bool)*number_of_nodes);
cudaMemcpy(d_frontier, h_frontier, sizeof(bool)*number_of_nodes, cudaMemcpyHostToDevice);

bool* d_visited;
cudaMalloc((void**)&d_visited, sizeof(bool)*number_of_nodes);
cudaMemcpy(d_visited, h_visited, sizeof(bool)*number_of_nodes, cudaMemcpyHostToDevice);

int* d_cost;
cudaMalloc((void**)&d_cost, sizeof(int)*number_of_nodes);
cudaMemcpy(d_cost, h_cost, sizeof(int)*number_of_nodes, cudaMemcpyHostToDevice);



dim3  grid( num_blocks, 1, 1);
dim3  threads( num_of_threads_per_block, 1, 1);
```

Fig. 6. Transferring data from host to device

In fig 5, after allocating the device memory to all the arrays, data is transferred from host to device. Which is accomplished by calling CUDA API functions like cudaMemcpy. The maximum threads per block can be 512.

```
__global__ void BFS_KERNEL(Node *d_vertices, int *d_edges, bool *d_frontier, bool *d_visited, int *d_cost,bool *update)
{

    int id = threadIdx.x + blockIdx.x * MAX_THREADS_PER_BLOCK;
    if (id > number_of_nodes)
        *update = false;


    if (d_frontier[id] == true && d_visited[id] == false)
    {
        printf("%d ", id);
        d_frontier[id] = false;
        d_visited[id] = true;
        __syncthreads();

        int f_index = d_vertices[id].f_index;
        int end = f_index + d_vertices[id].tnodes;
        for (int i = f_index; i < end; i++)
        {
            int nid = d_edges[i];

            if (d_visited[nid] == false)
            {
                d_cost[nid] = d_cost[id] + 1;
                d_frontier[nid] = true;
                *update = false;
            }

        }

    }

}
```

Fig. 6. Kernel code diagram

In Fig. 6, kernel code is demonstrated. Where d_frontier and d_visited are two Boolean arrays defined on device memory. Moreover, d_vertices, d_edges, d_cost are the integer arrays. __syncthreads(); is used to make sure that all vertices in one level are traversed. As one thread is assigned to each vertex.

Structure is used to represent every vertex, it is identified as an index in the array vertices. Structure have two properties:-

1. f_index :- It is the index of first adjacent node in array edges.
2. tnodes :- It is the number of adjacent nodes to a vertex.

In Fig.7, We have defined variable to check the total execution time of the parallel program. Moreover, we launch BFS_KERNEL and then copy cost array data from device memory to host memory. The cost need to reach every vertex from source vertex is calculated. At the end device memory used by the variables get free using API function cudaFree.

```
StopWatchInterface *timer = NULL;
sdkCreateTimer(&timer);
sdkStartTimer(&timer);

    bool update;
    bool* d_update;
    cudaMalloc((void**)&d_update, sizeof(bool));
    printf("\n\n");
    int count = 0;

    printf("Order: \n");
    do {
        count++;
        update = true;
        cudaMemcpy(d_update, &update, sizeof(bool), cudaMemcpyHostToDevice);
        sdkStartTimer(&timer);
        BFS_KERNEL<<<grid, threads >>>(d_vertices, d_edges, d_frontier, d_visited, d_cost,d_update);
        cudaMemcpy(&update, d_update , sizeof(bool), cudaMemcpyDeviceToHost);
        sdkStopTimer(&timer);

        timer1 = sdkGetTimerValue(&timer);
        sdkResetTimer(&timer);
    } while (!update);
    cudaMemcpy(h_cost, d_cost, sizeof(int)*number_of_nodes, cudaMemcpyDeviceToHost);

    printf("\nTotal number of times kernel call : %d \n", count);
    printf("Total Execution time: %f (ms)", timer1);
sdkDeleteTimer(&timer);

    printf("\nCost: ");
    for (int i = 0; i<number_of_nodes; i++)
        printf( "%d    ", h_cost[i]);
    printf("\n");
 cudaFree (d_visited);
 cudaFree (d_vertices);
 cudaFree (d_edges);
 cudaFree (d_frontier);
```

Fig. 7. Kernel launch by host code

5 Experiments and Results

We did three experiments with different number of nodes every time. We calculated the execution time of parallel bfs code and sequential bfs code. Moreover, to validate our parallel algorithm we calculated the speed up factor(How many times parallel algorithm is faster than sequential one).

Experiment No.	Number of graph nodes	GPU execution time	CPU execution time	Number of times kernel called	Speedup Factor
1.	10	0.10500 ms	1.300 ms	5	12.4x
2.	22	0.10800 ms	1.530 ms	7	14.1x
3.	64	0.11000 ms	2.710 ms	13	24.6x

Fig. 8. Experiment results

In fig 8, Experiment 1 is done by using graph with 10 nodes, the total number of times kernel called is 5 and the GPU implementation is 12.4 times faster than the CPU. Moreover, in experiment 2, input graph have 22 nodes and the resultant speedup factor is 14x. The third experiment done with the 64 node graph and the speedup factor is 24.6x.

Experiment 4
This experiment is to be done by using the graph given in the Fig.1. The input graph have 5 nodes. The source node is 0 and the cost to reach every other node from the node 0 is calculated.

```
Order:
0 1 2 3 4
Total number of times kernel call : 3
Total Execution time: 0.109000 (ms)
Cost: 0    1    1    2    2    0    0
```

Fig. 9. Screenshot from Nsight

In fig. 7, the total number of times kernel called is 3. The cost required to reach nodes 3 and 4 from source node is 2. The nodes are traversed in the order 0 > 1 > 2 > 3 > 4.

6 Analysis

The parallel breadth first search using GPUs can increase the overall computation speed as compare to the sequential algorithm using CPU. It can be helpful for solving problems like finding shortest path between two nodes more rapidly.

7 Conclusion

The system used in this provides a clear parallel approach for Breadth first search, which takes input graph from the compact adjacency list and traverses graph in levels. Experiment results shows that the computation speed of parallel algorithm on GPU is faster than the sequential BFS algorithm. Moreover, it also demonstrates that the compact adjacency list graph representation data structure makes it possible for parallel algorithm traverse graph faster as compare to the queue and matrix adjacency data structure. In addition, parallel approach to many other real world applications can make huge difference in term of computation speed using GPUs.

In future, many optimization techniques can be used to improve speed of the BFS parallel algorithm. Hierarchical queues can be used as graph representation data structure.

8 References

1. J.D Owens, D.Luebke, N.K Govindaraju, M.Harris, J.kruger, “A survey of general purpose computation on graphics hardware, “in Proc. Eurographics, state Art Rep, 2005, pp. 21-51.
2. Jun-Dong Cho, Salil Raje, and Majid Sarrafzadeh, “Fast approximation algorithms on maxcutm k-coloring and l-color ordering for ylsi applications,” IEEE transactions on Computers,1998.
3. David A.Bader and Kamnesh Madduri, “Designing multithreaded algorithms for breadth-first search and st-connectivity on MTA-2,” In ICPP,pages 523-530, 2006.
4. P.Harish and P.J Narayanan, Accelerating Large Graph Algorithms on the GPU using CUDA, “ in proc. HiPC, 2007, pp 197-208.
5. L. Luo, M. Wong, and W.-M. Hwu, An Effective GPU implementation of Breadth first search, 2010.
6. David B. Kirk, Wen-mei W. Hwu, “ Programming massively parallel processors” Third edition, page 258-261.