Project Report: Highly Optimized Parallel and Distributed Breadth-First Search on Graphic Processing Units

Tianyu Yang
G38878678
The George Washington University
Department of Electrical and Computer Engineering

***Abstract*-Breadth-first Search (BFS) is a kind of graph traversal algorithm, which has wide applications in multi-disciplines. It is also one of the representative parallel computations which can be executed on both CPU and GPU processor. In this project, we proposed to apply BFS algorithm on GPU processor and use multi-threads to finish the graph traversal work. Besides, we made a comparison with different graph traversal algorithm, such as Depth-first Search (DFS) and judge their output performance in big dataset. Moreover, we tested different number of nodes and edges and adjusted the number of the GPU threads to obtain the count for the best performance. With the comparison with BFS algorithm on GPU and different model of GPUs, we delivered that GPU has better output performance than CPU and better GPU can generate even faster. The overview of this report includes Introduction, Methodology, Procedures, Output performance and Discussion sections.**

I. INTRODUCTION

Breadth-First Search (BFS) algorithm serves as a building block for many analytics workloads, e.g., single source shortest path, betweenness centrality and closeness centrality. Notably, the Graph 500 benchmark uses BFS on power-law graph to evaluate high-performance hardware architectures and software systems that are designed to run data-intensive applications. We inspired our topic from the journal article named "Enterprise: Breadth-First Graph Traversal on GPUs". [1] We have idea how to use BFS for graphs on CPU. However, GPU has more cores and more treads that can search graph's node simultaneously than CPU. Thus, we want to know how it perform on GPU. In this report, we are interested at different features that BFS perform on GPU and compared with that on CPU. We are mainly test the features of time consumption; number of treads used. In addition, we also prepare to set a number of treads, and use less treads, then computing the efficiency of the treads.

A. Background

Breadth-First Search (BFS) algorithm is used in many graph-processing applications and some data analysis work. It is useful in many areas. In the meanwhile, Graphics Processing Unit (GPU) is one of the most power processors which can achieve high performance of parallel computing. With hundreds and thousands of threads and cores the GPU can provide, BFS algorithm can speed up when comparing to operate this algorithm on the GPU. However, GPU is a complex unit and we are not familiar with its programming language. Therefore, we need to pay more attention on how to write and run functions on GPU and how to manage data and use appropriate data structure to implement the BFS algorithm and count the running time.

B. Related Work

In the previous paper, Mr. Liu and Huang [1] present three techniques in GPU-based BFS systems. The Enterprise has a lot of difference of graph in different devices. Meanwhile, Mr. Merril [2] showed a BFS parallelization

focused on fine-grained task management constructed from efficient prefix sum that achieves an asymptotically optimal O(|V|+|E|) work complexity. The result presents This level of performance is several times faster than state-of-the-art implementations both CPU and GPU platforms. In the journal article of Mr. Harish [3], they used CUDA to accelerate the large graph algorithm. The result shows the time of computing the single source shortest path on a 10 million vertex graph is 1.5 seconds. Mr. Luo, Wang and Hwu [4] showed present a new GPU implementation of BFS that uses a hierarchical queue management technique and a three-layer kernel arrangement strategy. It guarantees the same computational complexity as the fastest sequential version and can achieve up to 10 times speedup.

These research work inspire us to work on this report and provide us many useful background knowledges about how to apply BFS graph algorithm on GPU processors.

II. METHODOLOGY

In this project, we used several methodologies to implement our goals. We learned and wrote program for BFS and DFS logistic algorithms. Besides, in order to implement our code on GPU processor, we studied CUDA programming and tried to run BFS and DFS circle on it. Moreover, we used Compressed Sparse Row (CSR) format to generate the big dataset from Stanford Large Network Dataset Collection. Last but not least, by using multi-threads on GPU, we speeded up the BFS algorithm by using parallel computing. In the following part, we will introduce some fundamental knowledge about the concepts and techniques mentioned above.

C. Breadth-first Search (BFS)

Breadth-first Search (BFS) is one of the traversal algorithms to search a tree or a graph data structure. It is an algorithm that consider the neighbor of a vertex, outgoing the edges of the vertex's predecessor when searching the tree or a graph. It starts at an arbitrary node of a graph and then go through all its neighbor nodes. It will not move to the next depth level until it finish going through all its neighbor nodes, shown as in Figure 1. [6]

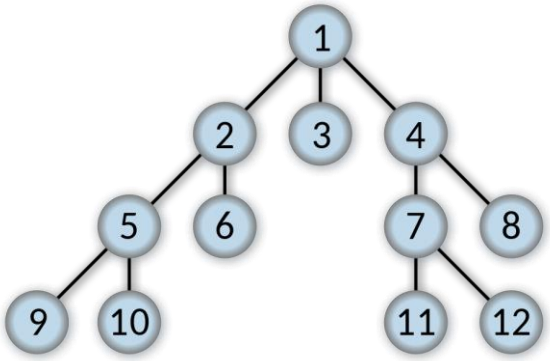

Figure 1. The BFS tree sample

If you first visit node 1, the next node you visit will be 2, 3, and 4. You will not move to the next level until you finished all your neighbors.

D. Depth-first Search (DFS)

Similar as BFS, Depth-fist Search is another traversal algorithm to search a tree or a graph data structure. It starts from one of the arbitrary nodes of a graph and then go through as far as possible to get every branch before

backtracking. [7] Also let us take Figure 1 as an example. If you first visit node 1, the next node you visit will be 2, 5, and 9. You will not stop until you reached the border of the tree. In the Figure 2, it shows how a tree structure generate DFS algorithm step by step.

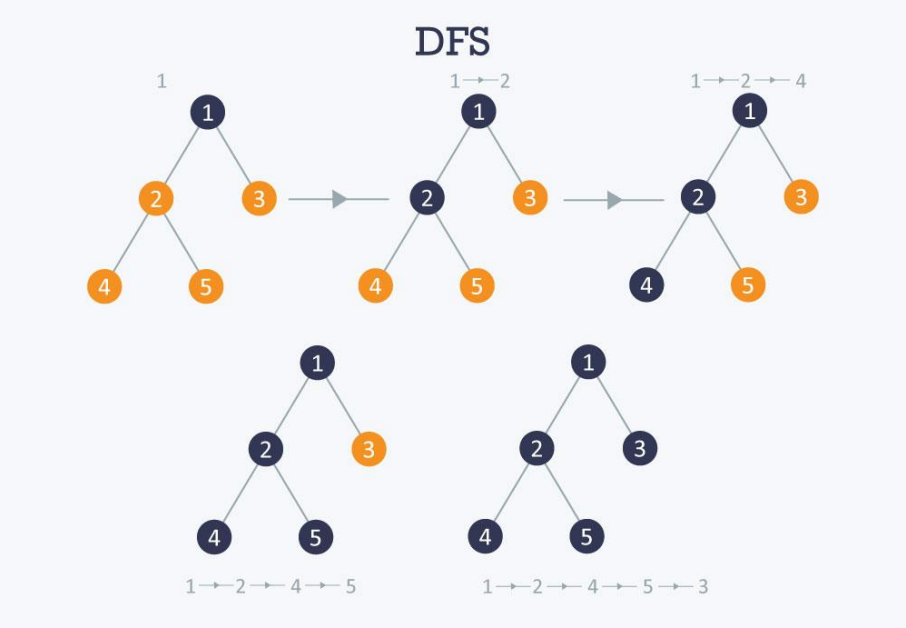

Figure 2. The BFS tree sample

E. CUDA Programming

CUDA is a parallel computing platform and application programming interface (API) model created by Nvidia.[1] It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing — an approach termed GPGPU (General-Purpose computing on Graphics Processing Units). [8] The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels. The Figure 3 shows the kernel of CUDA which makes a connection between CPU Host and GPU Device.

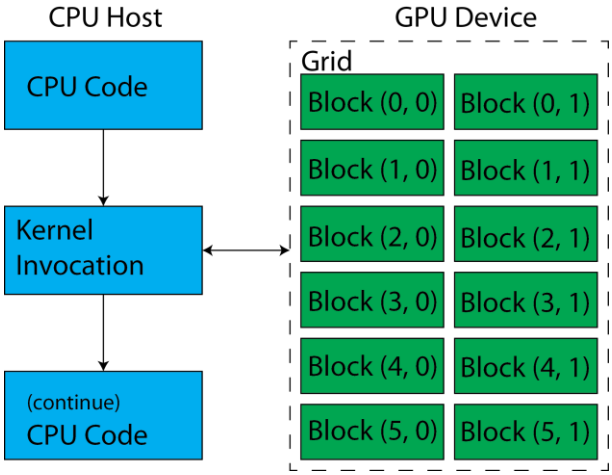

Figure 3. The connection between CPU Host and GPU Device

F. Compressed Sparse Row (CSR)

Compressed sparse row (CSR) is a popular format for storing sparse graphs and matrices. It efficiently packs all the entries together in arrays, allowing for quick traversals of the data structure. (from wiki). We always use this format to analyze graph to find the start node and the edge of the graph. [9] The Figure 4 is an example of the CSR for DFS. We have three steps for find the CSR:

1) Adding a Node: we add a node by extending the length of the node array by using a pointer to the end of the edge structure. Next, add a sentinel edge to the edge structure. Adding edges: To add an edge, you first need to find the node in the node array. Next, perform a binary search on the relevant part of the edge array and continue.

2) Removing an Edge: Removing edges and add edge symmetry. Find the edge with binary search, remove it from the PMA, and rebalance if necessary

3) Removing a Node: First, we set the start and end pointers into the edge array to null.

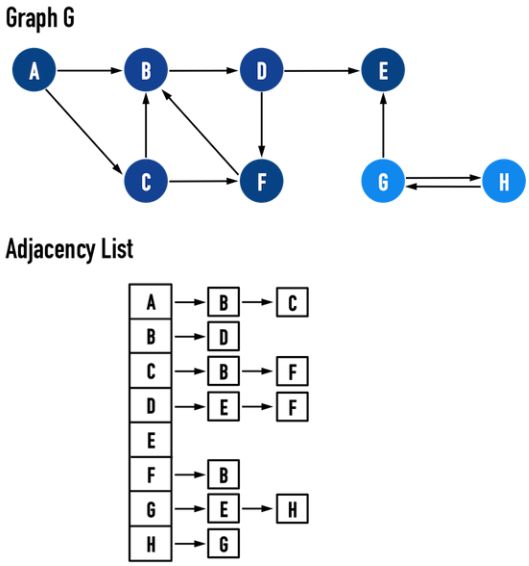

Figure 4. The example of DFS on CSR format

G. GPU Parallel Computing

Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously. We always use multi- treads to deal with the many processes simultaneously so that we can accelerate the algorithms we use. Both CPU and GPU have lots of cores to deal with data in multi-process. [10] However, the amount of the cores in CPU and GPU has huge difference. The cores in GPU is much more than these in CPU. Thus, the algorithm can have better performance in GPU generally. The figure below shows how CUDA generate for parallel computing in GPU. The program can be divided into several tasks and through parallel region, it can send the result back to CPU.

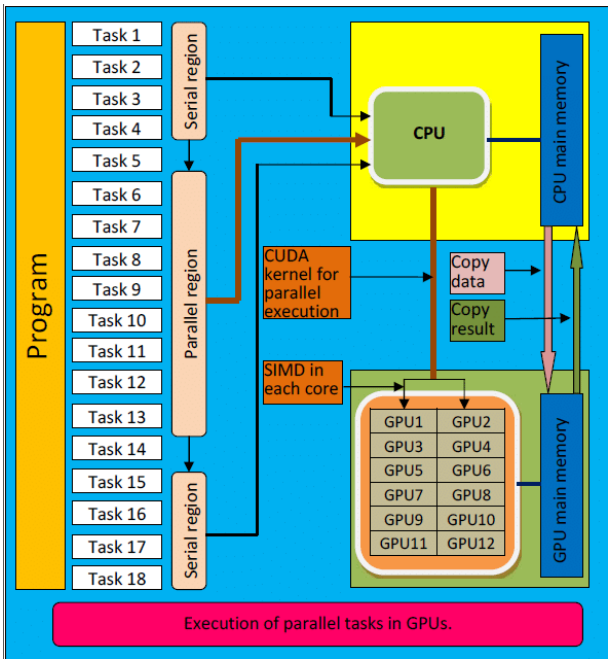

Figure 5. The diagram of parallel computing of the GPU

III. PROCEDURES

In the procedures part, we will introduce some experiment tools we used, the pipeline to implement our goals, the schedule for our project and our division of work.

H. Experiment tools

In this project, we used Visual Studio 2017 as software development IDE. Working on Windows 10 operating system, we used two different computers with different performance of CPU and GPU processors. One of them (Later we call it as COM 1) is LENOVO computer, with NVIDA GeForce GTX 1060 6GB, i7-7700HQ CPU, 15GB memory. Another one (Later we call it as COM 2) is HP computer, with NVIDA GeForce GTX 1050 Ti 4GB, i7-5500HQ CPU, 8GB memory. The version of CUDA we used is 10.1, which is the updated version on Nvidia official website.

I. Pipeline

In this project, our objective is to apply graph traversal algorithm on GPU processors, including BFS and DFS. Therefore, our working pipeline is shown as Figure 1. Firstly, we read some research paper related to graph traversal and GPU parallel computing in the proposal period. After confirming our objectives and goals of this project, in the mid-term period, we tried several simple BFS code on CPU to be familiar with BFS algorithm and programming language (C++). We tried to run and debug source code of BFS writing by ourselves after interpreting the kernel of BFS. Besides, we also learned how to configure CUDA on window OS computer and install it to our own computer. Besides, we also took some online tutorials for CUDA programming and learned how to implement CUDA core algorithm. Later on, we tried several CUDA own samples, such as adder functions, CudaGraph functions and etc. We also reviewed several reference codes written by other skillful and experience software programmer and tried to

learn more about CUDA programming and how to implement BFS on GPU processors. After that, we wrote our own codes to run BFS dynamic function on GPU processors, which is also our first CUDA code. The dataset we used is small, which only contains five nodes and five edges. In our final period of the project, we tried big data and online dataset. These datasets are from the Stanford University. [11] There are multiple types of the datasets we could use in .txt file. Most of them includes over 100 thousand nodes and edges. Then we tried Depth-first Search (DFS) algorithm, which we want to make a comparison with BFS and another graph traversal algorithm. After the code review and maintain, we finally print out the time and other related things we cared about. We record the processing time and compare them on COM1 and COM2. We also compare the efficiency of the usage of threads.

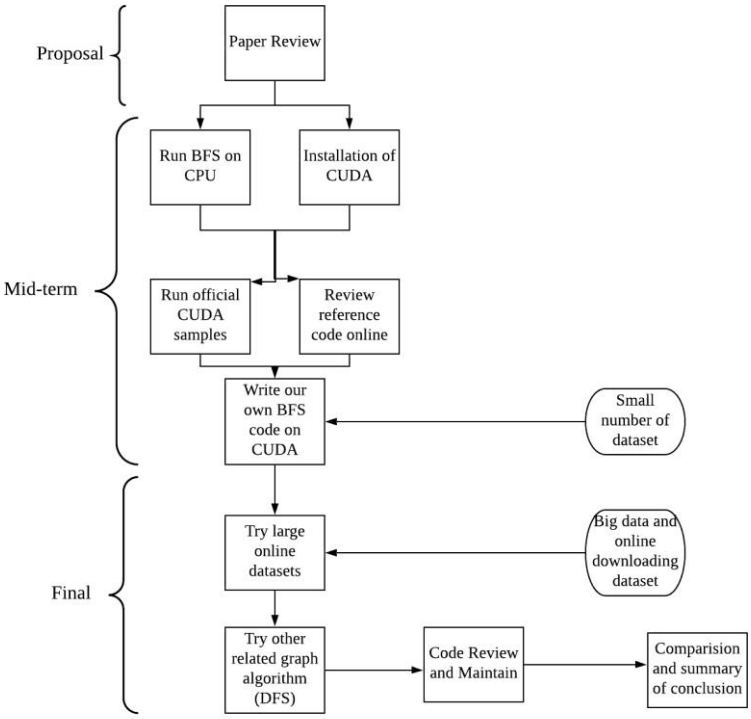

Figure 1. The project pipeline flow chart diagram

J. Schedule

The table below is our working schedule for the project.

TABLE I
THE PROJECT WORKING SCHEDULE

Date (Week)	Outcomes
4	Finish the initial proposal
5	Finish the final proposal
6	Read related research papers for BFS and GPU programming
7	Start to run BFS code on both CPU and GPU
10	Prepare the mid-term project presentation

11	Run BFS code on different computer's GPU
13	Run DFS code on machine
16	Code review and maintain
17	Finish the final report and the final presentation

K. Division of the work

The table below is our division of our responsibilities.

TABLE II
THE WORK RESPONSIBILITY TABLE

Items	Person	
	Tianyu Yang	Chao Tuo
Proposal Writing	√	√
Paper Review	√	√
BFS coding	√	
CUDA programming	√	
DFS coding		√
Mid-term presentation	√	√
Code review	√	√
Final report writing	√	√
Final Presentation	√	√

IV. OUTPERFORMANCE

In this section, we will display our project results and output performance. Besides, after the analysis of our result, we will give some discussions about the output performance. We worked on BFS running on GPU processor with both small datasets and large datasets. Besides, we also made a comparison with two different computers. We ran simple BFS and DFS code on CPU and calculate the general processing time. For further research, we worked on DFS coding on GPU and made a comparison with BFS algorithm in running different datasets. Finally, we tried to analyze the efficiency of the use of threads based on our output performance.

L. BFS running on GPU processors with small datasets

Our main target of this project is to implement Breath-first Search on GPU processors. In order to fulfill our goal, we tried a small data set with only 5 nodes and 5 edges, shown as Figure 2.

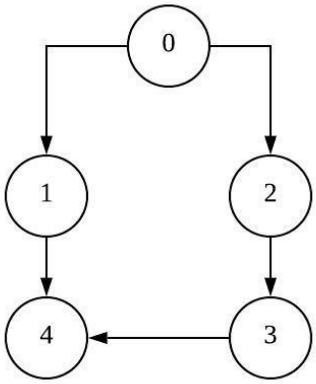

Figure 2. The graph of five nodes

As shown in the graph, there are totally five edges with a directed type. Then we apply it by using corresponding representation (Compressed Sparse Row), as shown in Figure 3.

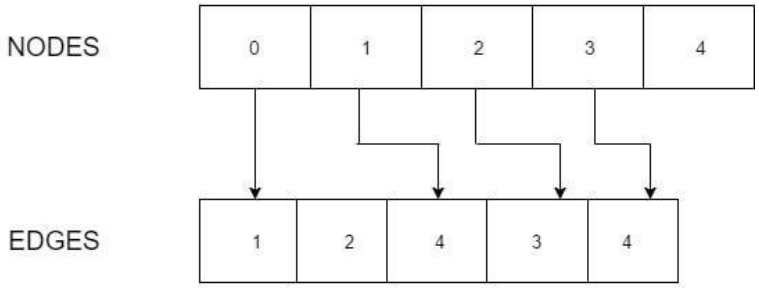

Figure 3. The CSR format for Figure 2

Following the Figure 3, we obtain an integer array with the value of edges, which the value of the edges means the end node of the link. Therefore, we get a way to represent a directed type graph, where we use the start position and length to represent the location of the nodes in the edges array and use edges array to represent the end node of each link.

Then we used CUDA which has been implemented on the Virtual Studio. We create vertex array from all vertices and edge array from all edges. We also create a frontier array, visited arrayed and cost array to evaluate the performance of GPU core. The pseudocode of the CUDA_BFS is shown as below:

Algorithm 1. CUDA_BFS (Graph $G(V,E)$, Source Vertex S)

Create vertex array V_a from all vertices and edge Array E_a from all edges in $G(V,E)$,
Create frontier array F_a, visited array X_a and cost array C_a of size V.
Initialize F_a, X_a to false and C_a to ∞
$F_a[S] \leftarrow$ true, $C_a[S] \leftarrow 0$
while F_a not Empty **do**
  for each vertex V in parallel **do**
    Invoke CUDA_BFS_KERNEL(V_a, E_a, F_a, X_a, C_a) on the grid.
  end for
end while

After finished the CUDA variable initialization, we wrote the CUDA_BFS_KERNEL, which includes five input value. The function of CUDA is in _global_ type and return void. This CUDA function generate all the nodes and record it as true if this node has been already visited. The entire code is shown in Appendix A. The pseudocode of CUDA_BFS_KERNEL is shown as below:

Algorithm 2. CUDA_BFS_KERNEL $(V_a, E_a, F_a, X_a, C_a)$

```
1: tid ← getThreadID
2: if F_a[tid] then
3:     F_a[tid] ← false, X_a[tid] ← true
4:     for all neighbors nid of tid do
5:         if NOT X_a[nid] then
6:             C_a[nid] ← C_a[tid]+1
7:             F_a[nid] ← true
8:         end if
9:     end for
10: end if
```

After running the program, we obtain the output on the console, which is shown as in the Figure 4.

Figure 4. The console output

From the output, we can obtain the number of nodes and the number of edges. In this program, I set the number of the threads is 3 so that the order of the use of threads is 0, 1, 2. The BFS circle takes 1080 microseconds. The total setting numbers of threads is 3 and the number of threads used is also 3, which means that all the threads has been used in this program. For traversing all the nodes, the threads for each node is 0, 1, 1, 2, 2. This order is the same as the level of BFS graph for these five nodes, which might be a coincident or not. Maybe we will find it out in later research.

M. BFS running on GPU processors with big data datasets

In order to test large datasets to make our result perfect, we download online datasets to test our program. We used the Stanford Large Network Dataset collection, which includes multiple types of datasets, such as social networks, networks with ground-truth communities, communication networks, citation networks, collaboration networks, web graphs, Amazon networks, Internet networks, Road networks, Autonomous system, Signed networks, Location-based online social networks, Wikipedia networks, articles, and metadata, temporal networks, Twitter and

Memetracker, online communities and online reviews. These datasets include various graph type, such as undirected, directed, signed, temporal, attributed and so on. With the various nodes and edges, we can freely test any kind of datasets on our program. In order to read these datasets, I wrote a program to generate the data file and save the important parameter to the node and edges variables in our program. The diagram of this program is shown as in the Figure 5;

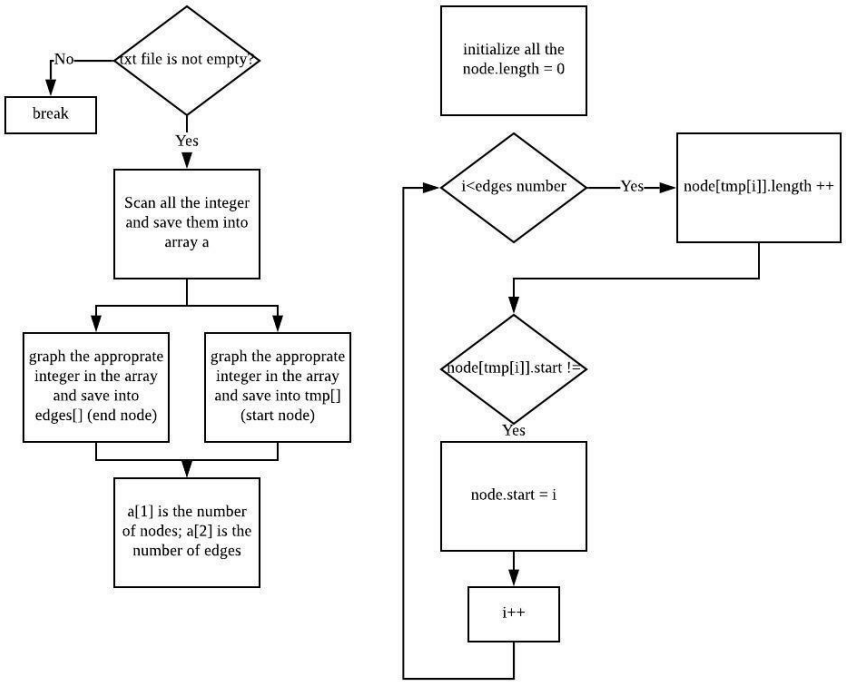

Figure 5. The diagram of txt dataset reading program

We ran different web graphs dataset and made a comparison of these graphs' procession time, the result shown as in the Figure 6. All of these web graphs are directed. [12] We generate the combined graph with histogram and line chart. From the chart, we can conclude that if the datasets have more nodes and more edges, the graph usually need more processing time to run a BFS graph traversal algorithm. This is because when the number of nodes and the complexity of the graph is large, it need more time to process all the nodes.[13]

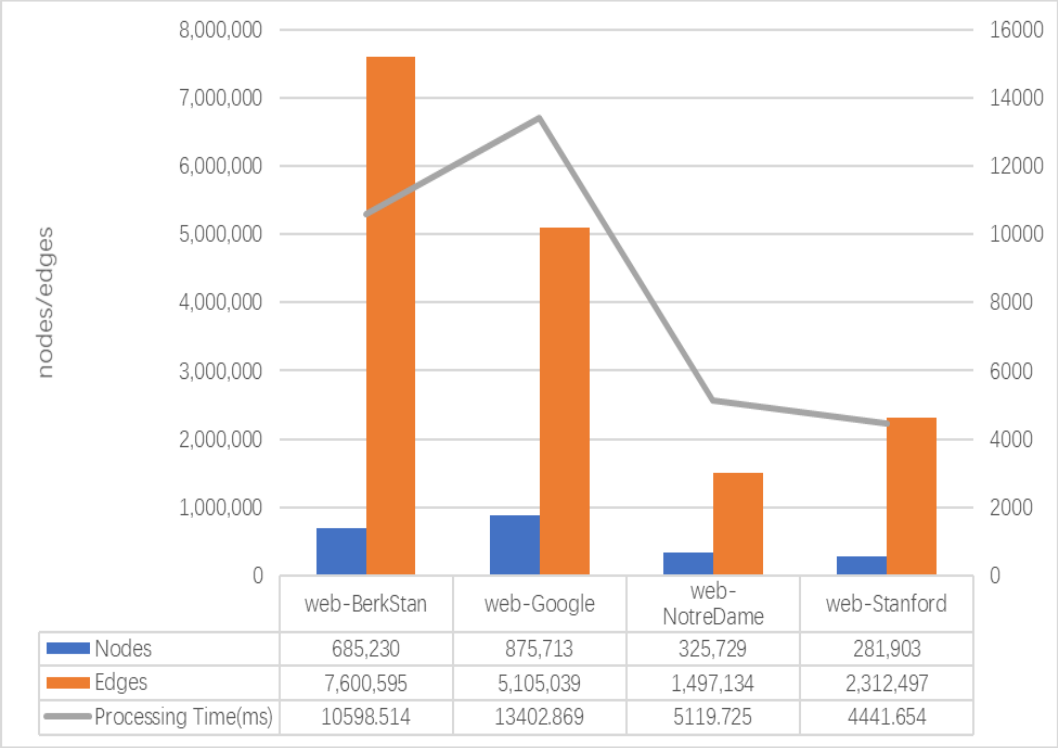

Figure 6. The histogram with different big datasets

Finally, we ran the same code on two different physical machines, COM1 and COM2, with the comparison of the CPU and GPU key parameters with the BFS processing time, we can conclude that if the physical machine has better CPU (some parts of the program need to run on CPU processors) and better GPU will have better performance on the processing time of BFS traversal circle. The comparison shows in the Table 3.

TABLE III
THE PROJECT WORKING SCHEDULE

Item	COM 1	COM 2
GPU	GTX 1060	GTX 1050 Ti
GPU Memory	6 GB	4 GB
CPU	i7-7700 HQ	i5-5500 HQ
Memory	16 GB	8 GB
Processing Time(ms)	13976.356	14568.599

N. Simple BFS and DFS algorithm executing on CPU

In order to explore more about BFS and DFS algorithm on CPU, we wrote simple BFS and DFS code by using C++ and running on Virtual Studio. We find the processing time on both graph traversal algorithm.

We create a binary tree and use BFS algorithm to return the level order traversal of its node's values. We use an example binary tree with 5 nodes and 4 edges, as [3,9,20, null,15,7]. The result it returns is as [[3], [9,20], [15,7]].

The BFS algorithm uses 4ms which spends 13.8 MB memory and the DFS algorithm uses 8 ms which spends 15 MB memory. The codes of BFS and DFS are shown in Appendix B and C.

O. Efficiency of threads analysis

From our paper review, we know that it costs considerable time when the number of threads is increasing. However, we can save lots of processing time if we use more threads to deal with the BFS circle. Therefore, there is a balance here which is related to the number of nodes. If the nodes' number is large enough, the cost of threads will be negligible. When the nodes' number is very small, which means that the graph is also small, the cost of threads is considerable. Following this rule, we define a complete graph (a simple undirected graph in which every pair of distinct vertices is connected by a unique edge). We define this special graph nodes to generate different numbers of nodes. From the table IV, we can conclude that when the threads we set increasing, the BFS processing time is increasing. However, there is a strange character that when the thread is increasing over a threshold, the processing number is decreasing significantly.

TABLE IV
THE PROJECT WORKING SCHEDULE

nodes	thread	no. threads used	processing time(μ s)
63	3	3	1272
63	6	4	1591
63	50	6	5497
1023	3	3	921
1023	6	4	2026
1023	50	7	6133
1023	1000	1	750

V. DISCUSSION

In this Discussion session, we will summary our project and give some results about our project. Besides, we will also evaluate our project by using scientific methods. To make our project better, we will also give some future implementation.

P. Conclusion

In conclusion, in this project, we use work on parallel programming of Breath-first Search and try to implement the algorithm on GPU processors. We wrote our own code and test them on different physical machines. Besides, collecting the processing time of the BFS circle, we analyze the relationship between the processing time and the size of dataset. We also put forward the questions about the used threads with processing time. Last but not least, we also try another graph traversal algorithm like Depth-first Search. With the research in both BFS and DFS on different processors like CPU and GPU, we learn more about parallel computing and obtain the experience of programming in graph traversal algorithm.

Q. Evaluation

To evaluate our project, we use the scientific research methodology. In order to get more accurate result, we record 3 times of processing time and calculate the average value of the time. Besides, to obtain our conclusion and summary, we test over 5 times of experiments. If all of these results follow the same rules, we put forward the rules or the relationship. Besides, we test different dataset of graph. These graphs including all types of nodes. What we show in the report is only the representative one of all our testing.

R. Future Implementation

In future we will focus on the accelerate BFS algorithm using CUDA on GPU. In recent work, we just achieved running the BFS source code using the dataset from website. But a more important performance of BFS is speed. We need to find some methods to accelerate the algorithm to get better performance. Besides, our original idea is to implement DFS algorithm on GPU processors and make a comparison with the BFS and DFS on different type of dataset. [14] From our implement on one of the samples on CPU, we find that BFS runs faster than DFS. However, this might just be a special dataset. In the future, we will try to find out which kind of dataset the BFS generates less time and which one the DFS generates less.

REFERENCES

[1] H. Liu and H. H. Huang, "Enterprise: breadth-first graph traversal on GPUs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '15*, Austin, Texas, 2015, pp. 1–12.

[2] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming - PPoPP '12*, New Orleans, Louisiana, USA, 2012, p. 117.

[3] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming - PPoPP '12*, New Orleans, Louisiana, USA, 2012, p. 117.

[4] L. Luo, M. Wong, and W. Hwu, "An effective GPU implementation of breadth-first search," in *Proceedings of the 47th Design Automation Conference on - DAC '10*, Anaheim, California, 2010, p. 52.

[5] J. Soman, K. Kishore, and P. J. Narayanan, "A fast GPU algorithm for graph connectivity," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, Atlanta, GA, 2010, pp. 1–8.

[6] U. Brandes, "A faster algorithm for betweenness centrality*," *The Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, Jun. 2001.

[7] M. Borokhovich, L. Schiff, and S. Schmid, "Provable data plane connectivity with local fast failover: introducing openflow graph algorithms," in *Proceedings of the third workshop on Hot topics in software defined networking - HotSDN '14*, Chicago, Illinois, USA, 2014, pp. 121–126.

[8] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *Journal of Computational Physics*, vol. 227, no. 10, pp. 5342–5359, May 2008.

[9] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills, "Vectorized Sparse Matrix Multiply for Compressed Row Storage Format," in *Computational Science – ICCS 2005*, vol. 3514, V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 99–106.

[10] Symposium on Discrete Algorithms, Association for Computing Machinery, and Society for Industrial and Applied Mathematics, Eds., *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms: Miami, FL., January 22 - 24, 2006*. New York, NY: Association for Computing Machinery [u.a.], 2006.

[11] M. Krajecki, J. Loiseau, F. Alin, and C. Jaillet, "BFS Traversal on Multi-GPU Cluster," in *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and*

Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES), Paris, 2016, pp. 594–599.
[12] F. Busato and N. Bombieri, "BFS-4K: An Efficient Implementation of BFS for Kepler GPU Architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 7, pp. 1826–1838, Jul. 2015.
[13] Z. Fu, H. K. Dasari, B. Bebee, M. Berzins, and B. Thompson, "Parallel Breadth First Search on GPU clusters," in *2014 IEEE International Conference on Big Data (Big Data)*, Washington, DC, USA, 2014, pp. 110–118.
[14] K. Ueno and T. Suzumura, "Parallel distributed breadth first search on GPU," in *20th Annual International Conference on High Performance Computing*, Bengaluru (Bangalore), Karnataka, India, 2013, pp. 314–323.

APPENDIX

A. BFS code on GPU(cuda)

```
/*
*
* ECE 6130 Big Data and Cloud Computing
* Spring 2019
* Project code: Highly Optimized Parallel and Distributed Breadth First Search on Graphic Processing Units
* Name: Tianyu Yang
* GW ID:G38878678
* Referenced from https://siddharths2710.wordpress.com/2017/05/16/implementing-breadth-first-search-in-cuda/
*
*/

#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <device_functions.h>
#include <cuda_runtime_api.h>

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include <iostream>
#include <ctime>
#include <ratio>
#include <chrono>

#define NUM_NODES 99999999//1023
#define parameter 511//511
#define time 1

using namespace std;
int n, r;
double d;
FILE *f;

typedef struct
{
	int start;	// Index of first adjacent node in Ea
	int length;	// Number of adjacent nodes
} Node;

// Define the structure of node

__global__ void CUDA_BFS_KERNEL(Node *Va, int *Ea, bool *Fa, bool *Xa, int *Ca, bool *done)
{

	int id = threadIdx.x + blockIdx.x * blockDim.x;
	if (id > NUM_NODES)
		*done = false;


	if (Fa[id] == true && Xa[id] == false)
	{
		printf("%d ", id); //This printf gives the order of vertices in BFS
		Fa[id] = false;
		Xa[id] = true;
```

```
		__syncthreads();
		int k = 0;
		int i;
		int start = Va[id].start;
		int end = start + Va[id].length;
		for (int i = start; i < end; i++)
		{
			int nid = Ea[i];

			if (Xa[nid] == false)
			{
				Ca[nid] = Ca[id] + 1;
				Fa[nid] = true;
				*done = false;
			}

		}

	}

}

// The BFS frontier corresponds to all the nodes being processed at the current level.

int main()
{
	Node node[NUM_NODES];
	//int edgesSize = 2 * NUM_NODES;
	int edges[NUM_NODES];

	int a[NUM_NODES];
	int tmp[NUM_NODES];
	char fileName[] = "web-Google.txt";
	f = fopen(fileName, "r");
	n = 0;
	while (1) {
		r = fscanf(f, "%lf", &d);
		if (1 == r) {
			n++;
			//printf("[%d]==%lg\n", n-1, d);
			a[n - 1] = (int)d;
		}
		else if (0 == r) {
			fscanf(f, "%*c");
		}
		else break;
	}

	//number of nodes and edges
	int n = a[1];
	int e = a[2];
	Node node[NUM_NODES];
	//int edgesSize = 2 * NUM_NODES;
	int edges[NUM_NODES];
	cout << "No. of nodes = " << n << endl;
	cout << "No. of edges = " <<e << endl;
	for (int i = 3; i < 2 * e + 3; i++) {
		if (i % 2 == 0) {
			edges[i / 2 - 2] = a[i];
		}
		if (i % 2 == 1) {
			tmp[(i - 1) / 2 - 1] = a[i];
		}
	}
	for (int i = 0; i < n; i++) {
		node[i].length = 0;
	}
	for (int i = 0; i < e; i++) {
		//cout << edges[i] << endl;
		//cout <<tmp[i] << endl;
		if (node[tmp[i]].start != 0) {
			node[tmp[i]].start = i;
```

```
            }
            node[tmp[i]].length++;
}
for (int i = 0; i < n; i++) {
            //cout << node[i].start << endl;
            //cout << node[i].length << endl;
}
fclose(f);

// Special graph nodes
/*for (int i = 0; i < parameter; i++) {
            node[i].start = 2*i;
            node[i].length = 2;
}
for (int i = parameter; i < NUM_NODES; i++) {
            node[i].start = i+1;
            node[i].length = 0;
}
for (int i = 0; i < NUM_NODES; i++) {
            edges[i] = i+1;
}*/

 //Eg. 1
/*node[0].start = 0;
node[0].length = 2;

node[1].start = 2;
node[1].length = 1;

node[2].start = 3;
node[2].length = 1;

node[3].start = 4;
node[3].length = 1;

node[4].start = 5;
node[4].length = 0;

edges[0] = 1;
edges[1] = 2;
edges[2] = 4;
edges[3] = 3;
edges[4] = 4;*/

// Eg. 2
 /*node[0].start = 0;
 node[0].length = 2;

 node[1].start = 2;
 node[1].length = 2;

 node[2].start = 4;
 node[2].length = 2;

 node[3].start = 6;
 node[3].length = 2;

 node[4].start = 5;
 node[4].length = 0;

 edges[0] = 1;
 edges[1] = 2;
 edges[2] = 0;
 edges[3] = 3;
 edges[4] = 0;
 edges[5] = 3;
 edges[6] = 1;
 edges[7] = 2;*/

bool frontier[NUM_NODES] = { false };
bool visited[NUM_NODES] = { false };
int cost[NUM_NODES] = { 0 };
```

```
	int source = 0;
	frontier[source] = true;

	Node* Va;
	cudaMalloc((void**)&Va, sizeof(Node)*NUM_NODES);
	cudaMemcpy(Va, node, sizeof(Node)*NUM_NODES, cudaMemcpyHostToDevice);

	int* Ea;
	cudaMalloc((void**)&Ea, sizeof(Node)*NUM_NODES);
	cudaMemcpy(Ea, edges, sizeof(Node)*NUM_NODES, cudaMemcpyHostToDevice);

	bool* Fa;
	cudaMalloc((void**)&Fa, sizeof(bool)*NUM_NODES);
	cudaMemcpy(Fa, frontier, sizeof(bool)*NUM_NODES, cudaMemcpyHostToDevice);

	bool* Xa;
	cudaMalloc((void**)&Xa, sizeof(bool)*NUM_NODES);
	cudaMemcpy(Xa, visited, sizeof(bool)*NUM_NODES, cudaMemcpyHostToDevice);

	int* Ca;
	cudaMalloc((void**)&Ca, sizeof(int)*NUM_NODES);
	cudaMemcpy(Ca, cost, sizeof(int)*NUM_NODES, cudaMemcpyHostToDevice);

	int num_blks = 1;
	int threads = 5;

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

	printf("Threads Order: \n\n");

	using namespace std::chrono;
	auto start = high_resolution_clock::now();

	// Run n times for Bfs program
	for (int i = 0; i < time; i++) {
		do {
			count++;
			done = true;
			cudaMemcpy(d_done, &done, sizeof(bool), cudaMemcpyHostToDevice);
			CUDA_BFS_KERNEL << <num_blks, threads >> > (Va, Ea, Fa, Xa, Ca, d_done);
			cudaMemcpy(&done, d_done, sizeof(bool), cudaMemcpyDeviceToHost);

		} while (!done);
	}

	auto stop = high_resolution_clock::now();
	auto duration = duration_cast<microseconds>(stop - start);
	std::cout << "\nTime taken: " << duration.count() << " us" << std::endl;

	cudaMemcpy(cost, Ca, sizeof(int)*NUM_NODES, cudaMemcpyDeviceToHost);

	printf("\nNumber of threads used : %d \n", count);


	printf("\nThreads for each node: ");
	for (int i = 0; i < NUM_NODES; i++)
		printf("%d    ", cost[i]);
	printf("\n");
	_getch();
	system("pause");

}
```

B. BFS code on CPU(C++)

```
/**
 * Definition for a binary tree node.
```

```
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        if (!root) {
            return {};
        }
        vector<vector<int>> levels;
        queue<TreeNode*> todo;
        todo.push(root);
        while (!todo.empty()) {
            levels.push_back({});
            for (int i = 0, n = todo.size(); i < n; i++) {
                TreeNode* node = todo.front();
                todo.pop();
                levels.back().push_back(node -> val);
                if (node -> left) {
                    todo.push(node -> left);
                }
                if (node -> right) {
                    todo.push(node -> right);
                }
            }
        }
        return levels;
    }
};
```

C. DFS code on CPU(C++)

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        if (!root) {
            return {};
        }
        vector<vector<int>> levels;
        level(root, 0, levels);
        return levels;
    }
private:
    void level(TreeNode* root, int l, vector<vector<int>>& levels) {
        if (!root) {
            return;
        }
        if (levels.size() <= l) {
            levels.push_back({});
        }
        levels[l].push_back(root -> val);
        level(root -> left, l + 1, levels);
        level(root -> right, l + 1, levels);
    }
};
```