CSE 603 Project Report
Parallel BFS on Graphs using GPGPU

Name: Soumasish Goswami

Dec 18, 2015

Abstract

Graph algorithms are a fundamental paradigm of computer Science and is relevant to many domains and application areas. Large graphs involving millions of vertices are common in scientific and engineering applications. Reasonable time bound implementations of graph algorithms using high-end computing resources have been reported but are accessible to only a few. Graphics Processing Units (GPUs) are fast emerging as inexpensive parallel processors due to their high computation power and low price. The GeForce line of Nvidia GPUs provides the CUDA programming model that treats the GPU as a SIMD processor array. I've presented a fundamental graph algorithm - the breadth first search, using this programming model on large graphs. The results on a graph of 500, 000 vertices would suggest that the NVIDIA GPUs can be used as a reasonable co-processor to accelerate parts of an application.

Introduction

Graph representations are common in many problems of scientific and engineering applications. Some of these problems map to very large graphs, often involving millions of vertices, viz, problems likeVLSI chip layout, phylogeny reconstruction, data mining, and network analysis can require graphs with millions of vertices. Running serial implementations of Breadth-First-Search traversal algorithms on such large datasets can be a time-consuming task. This report summarizes observations on parallelizing this task using CUDA. Owing to the limitations of hardware and environment as obtained in Centre for Computational Research (CCR), the project has been tested in its entirety on a personal computer(with above average specs). However the observations can be easily generalized to a larger data set. The actual tests on the serial algorithm run significantly longer than the timings plotted in the report, primarily because the sub routine of traversing a list of edges and adding it to the graph is painfully slow. The relevant tables in the observations section only summarizes the time taken to run the actual Breadth-First-Search(hereby referred to as BFS) algorithm.

GPGPU

General purpose programming on Graphics processing Units (GPGPU) tries to solve a problem by posing it as a graphics rendering problem, restricting the range of solutions that can be ported to the GPU. A GPGPU solution is designed to follow the general flow of the graphics pipeline (consisting of vertex, geometry and pixel processors), with each iteration of the solution being one rendering pass. Since the GPU memory layout is also optimized for graphics rendering the GPGPU solutions as an optimal data structure may not be available. Creating efficient data structures using the GPU memory model is a challenge in itself . Memory size on GPU is another restricting factor. A single data structure on the GPU cannot be larger than the maximum texture size supported by it.

The CUDA Programming Model

For the programmer the CUDA model is a collection of threads running in parallel. A warp is a collection of threads that can run simultaneously on a multiprocessor. The warp size is fixed for a specific GPU. The programmer decides the number of threads to be executed. If the number of threads is more than the warp size, they are time-shared internally on the multiprocessor. A collection of threads (called a block) runs on a multiprocessor at a given time. Multiple blocks can be assigned to a single multiprocessor and their execution is time-shared. A single execution on a device generates a number of blocks. A collection of all blocks in a single execution is called a grid. All threads of all blocks executing on a single multiprocessor divide its resources equally amongst themselves. Each thread and block is given a unique ID that can be accessed within the thread during its execution. Each thread executes a single instruction set called the kernel.

The kernel is the core code to be executed on each thread. Using the thread and block

IDs each thread can perform the kernel task on different set of data. Since the device memory is available to all the threads, it can access any memory location. The CUDA programming interface provides a Parallel Random Access Machine (PRAM) architecture, if one uses the device memory alone. However, the multiprocessors follow a SIMD model, the performance improves with the use of shared memory which can be accessed faster than the device memory. The hardware architecture allows multiple instruction sets to be executed on different multiprocessors. The current CUDA programming model, however, cannot assign different kernels to different multiprocessors.

With CUDA, the GPU ca be viewed as a massive parallel SIMD processor, limited only by the amount of memory available on the graphics hardware. The GT 750M graphics card has 2048 MB memory. Large graphs can reside in this memory, given a suitable representation.

Device Specs

GeForce GT 750M	
CUDA Driver Version/ Runtime Version	7.5/7.5
CUDA Capability Major/Minor version	3.0
Total amount of global memory	2048 MBytes
2 Multiprocessors, 192 CUDA cores/MP	384 CUDA Cores
GPU Max Clock rate	926 Mhz
Memory Clock Rate	2508 Mhz
Memory Bus Width	128- bit
L2 Cache Size	262144 bytes
Maximum Texture Dimension Size(x,y,z)	1D=(65536), 2D = (65536, 65536), 3D= (4096, 4096, 4096)
Maximum layered 1D Texture size	1D=(16384), 2048 layers
Maximum layered 2D Texture size	2D= (16384, 16384), 2048 layers
Total amount of constant memory	65536 bytes
Warp Size	32
Maximum number of threads per multiprocessor	2048
Maximum number of threads per block	1024
Maximum dimension of a thread block(x, y, z)	(1024, 1024, 64)
Maximum dimension of a grid size(x, y, z)	(2147483647, 65535, 65535)

Thus the device limitations being set is that a total of 786,432 threads could be run at the same time owing to the device capability of 384 cores of 2048 threads each.

Data Set

The data set taken in as an argument by both the implementations is a vector/array of edge lists. To this extent this is a rough parallel to he Erdos Renyi model of graph generation. A brief reference can be found here:
The random seed used to generate the graph in this case is, we use a C++11 seed generator:

```
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> dis(0, (NUM_VERTICES -1));
```

Key assumptions

For the sake of simplicity we assume each vertex to be an integer. An edge is thus a pair of integers representing from and to. An edge list is an array of such pairs. All edges are bi-directional and have equal weights.

Core Serial Algorithm

The serial implementation follows a simple C++ STL Queue/Vector based graph. The Graph class maintains a vector of vectors for each vertex listing the connected edges for the respective vertex. The BFS algorithm uses a queue to traverse across each frontier / level and touches all connected components from a start node.

Parallelization of BFS

Since BFS lends itself well to parallelization there are two common yet distinct strategies that are followed in the parallel execution of BFS:

1. The level-synchronous algorithm.
2. The fixed-point algorithm

The level synchronous algorithm uses the following approach; it manages three sets of nodes - the visited set V , the current-level set C, and the next-level set N. Iteratively, the algorithm visits (in parallel) all the nodes in set C and transfers them to set V (in parallel). C is then populated with the nodes from set N, and N is cleared for the new iteration. This iterative process continues until naturally there is no node in the next level. The level synchronous algorithm effectively visits in parallel all nodes in each BFS level, with the parallel execution synchronizing at the end of each level iteration.

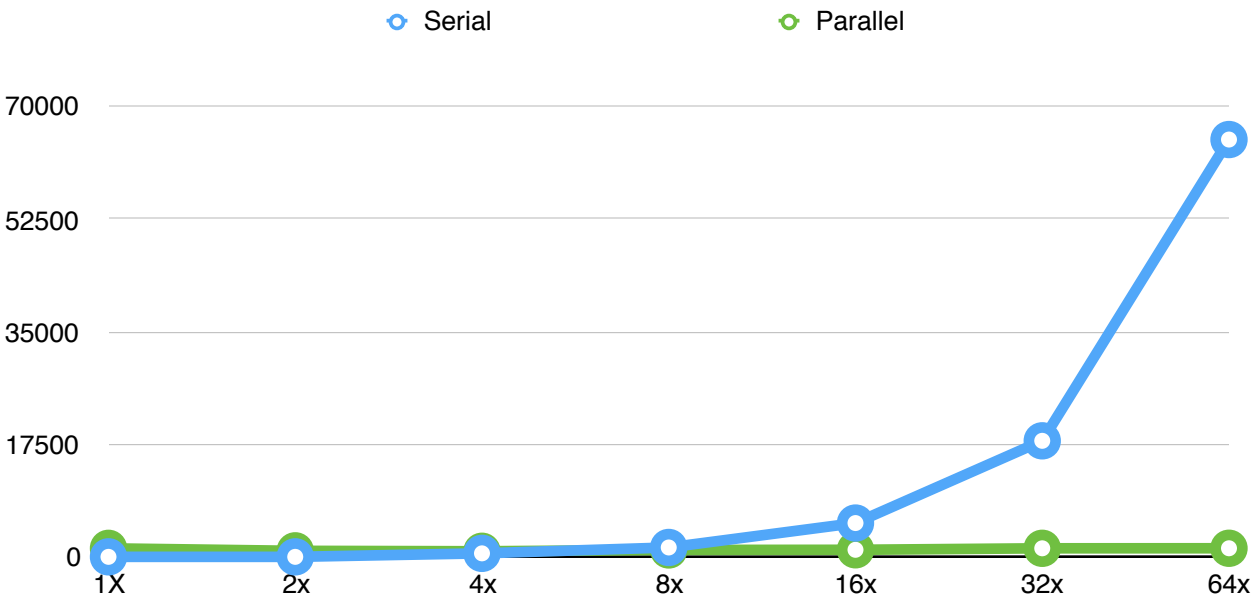
The fixed-point algorithm,on the other hand, continuously updates the BFS level of every node, based on BFS levels of all neighboring nodes until no more updates are made. This method is sub-optimal at times because of the lack of communication between neighboring nodes in parallel environments. For the purpose of this project I've based my implementation on this approach.

Observations & Speed Up

Starting from the base data set of 1024 vertices and an equal number on nodes the data set has been scaled up progressively in multiples of 2. The details of the scale up in the vertex to edge ratio and the time taken to run BFS on them are enumerated in the tables below.

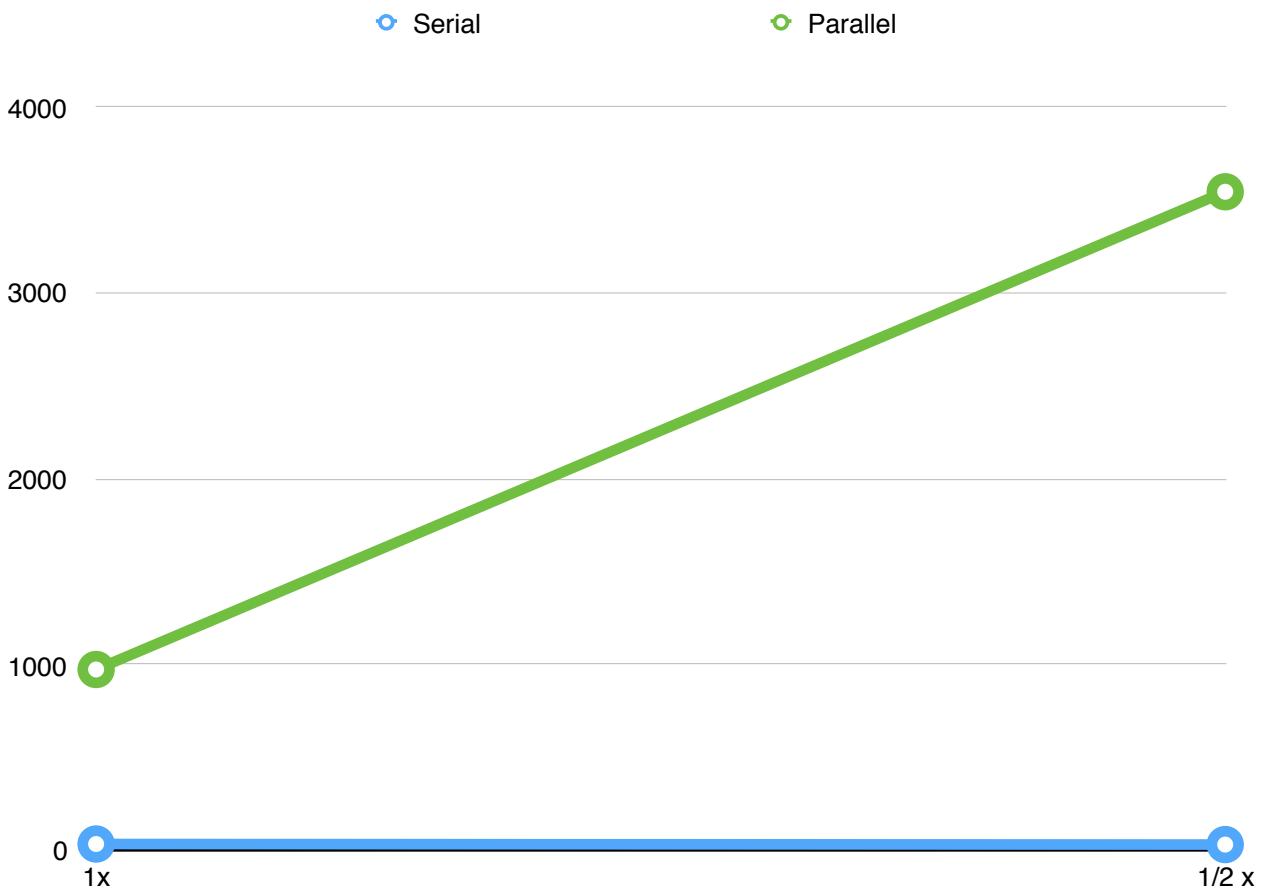
Base testing data set -1024 vertices

Vertices	Edges	Serial Runtime	Parallel Runtime	Speed Up
1024	1024	0.000012	0.001332	-11000%
1024	2048	0.000011	0.000944	-8481.8%
1024	4096	0.000554	0.000877	-58.3%
1024	8192	0.001467	0.001084	26.1%
1024	16,384	0.005263	0.001106	99.9%
1024	32,768	0.018046	0.001338	92.5%
1024	65,536	0.064879	0.001344	97.9%



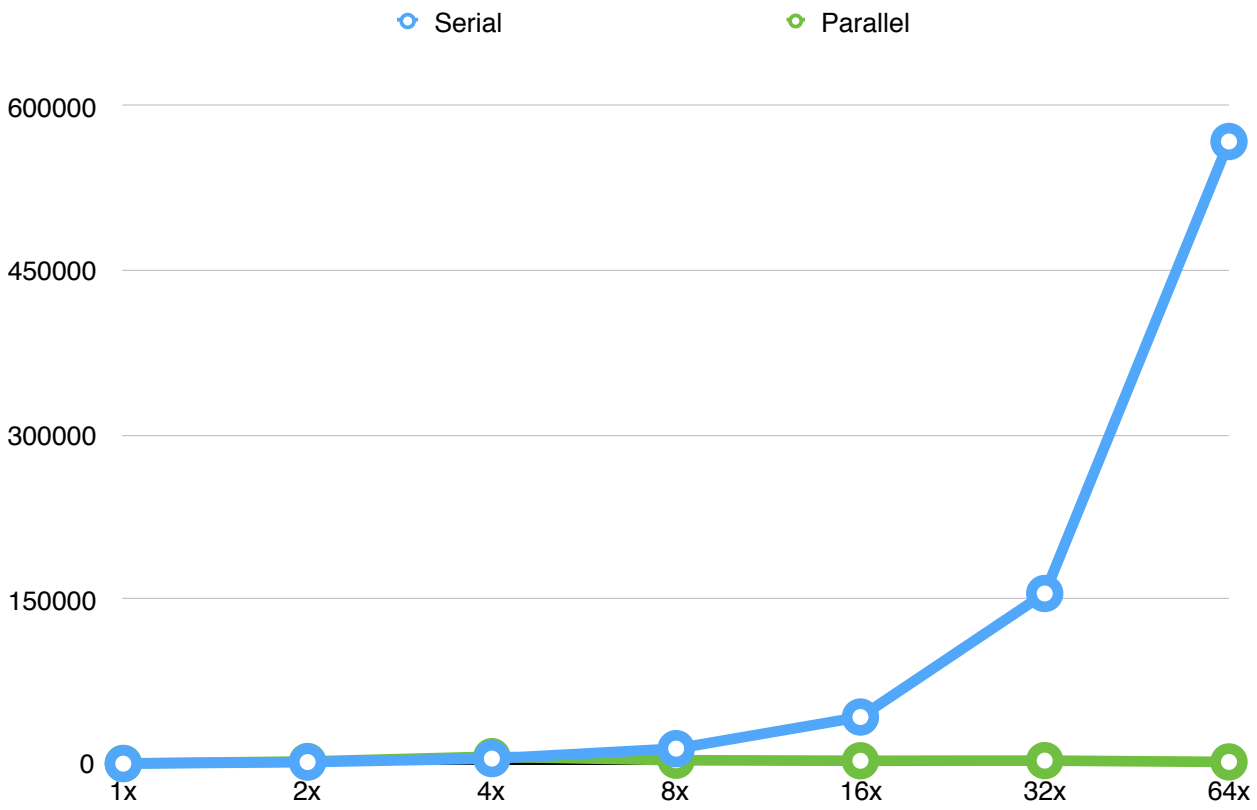
Data on a more sparse graph

Vertices	Edges	Serial Runtime	Parallel Runtime	Speed Up
65536	32,768	0.000034	0.000974	-2764.7%
65536	65,536	0.000031	0.003547	-11341.9%



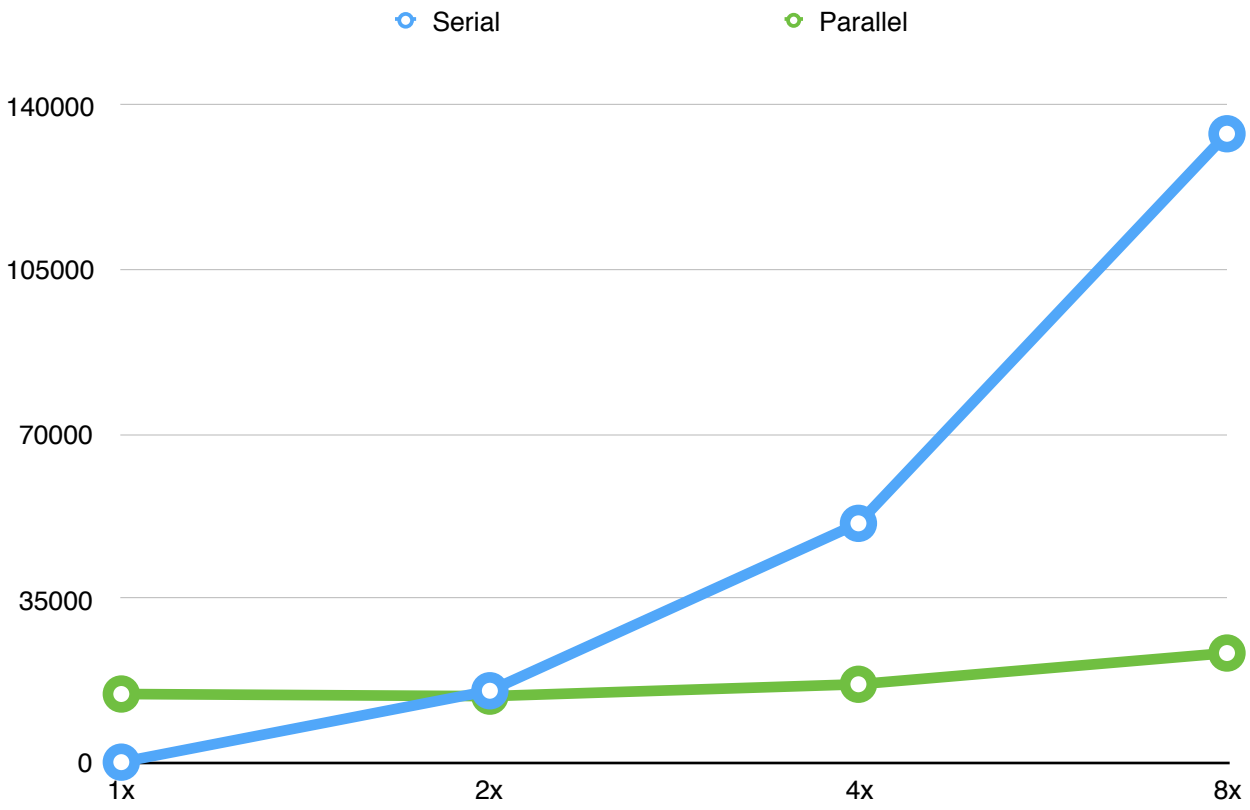
Scale up - 8X

Vertices	Edges	Serial Runtime	Parallel Runtime	Speed Up
8192	8192	0.000026	0.000235	-803.8%
8192	16384	0.001461	0.001979	-35.4%
8192	32762	0.004600	0.006388	-38.8%
8192	65536	0.013723	0.003123	77.2%
8192	131072	0.042567	0.002640	93.7%
8192	262096	0.155348	0.002776	98.2%
8192	524288	0.567887	0.001600	99.7%



Scale up - 16X

Vertices	Edges	Serial Runtime	Parallel Runtime	Speed Up
65536	65536	0.000181	0.014693	-8017.6%
65536	131072	0.015386	0.014237	7.4%
65536	262096	0.051042	0.016766	67.1%
65536	524288	0.134016	0.023420	82.5%



Analysis

The following trends emanate from the plotting of the two algorithms run on the same data set.

1. The serial algorithm tends to work better on a extremely sparse graphs with a 1:1 vertex to edge ratio.
2. The parallel algorithm works better on moderately sparse graphs and shows a significant runtime improvement as the vertex to edge ratio goes up.
3. As the graph gets denser below 1x to 1/2x or 1/4x, the parallel algorithm continues to perform poorer and poorer, primarily owing to the multiple serial calls being made to launch the kernel. This is the serial bottleneck of this algorithm.
4. As the data set grows there seems to be a gradual fall in the performance difference of the two algorithms on graphs with a 1:1/1:2 vertex edge ratio. Thus from 1024 to 65536 the speed up values are -8481% and 7%(1:2 vertex to edge). It can be safely assumed that as the data set grows the relative runtime of the serial algorithm on the a sparse graph will continue to decrease. Thus the parallel algorithm will proportionately keep performing better over larger and larger data sets.
5. The catch up is also quicker as the data set grows. For the base data set of 1024 nodes the edge density had to increase to 8x over vertex density for a yield of 27%. However in case of a graph with 65536 vertices the edge increase of only 4x yielded 67% speed up. Clearly the more the data the better the speedup - in both relative and absolute terms.
6. Aside from the timings noted here each algorithm takes a non-trivial amount of time doing initializations. The serial algorithm has to read an array of edge list and compose the graph before it can start BFS. That in effect doubles the execution time of the program. In case of the parallel algorithm the frontier array has to be initialized to -1 for all levels except the start vertex. For a large data set this might have to be done in batches which could add non-trivial overhead on the execution of the program.
7. There was some difference observed in running the same data on CUDA multiple times. Though not significant for a data set less than 500, 000 nodes this is something which needs to be observed on a bigger data set of billion nodes. In all likelihood the difference in runtime is attributable to other processes using the GPU on the machine.
8. One key observation while running the tests was that the level of connectivity of a node played a key role in determining the runtime. For the same data set, different start points had significantly different running times based on its level of connectivity. Thus it was of paramount importance to use the same start vertex for both the algorithms.

Limitations of the Algorithm

It's not a surprise that this algorithm works well with a multi-connected graph where one vertex is connected to many edges but limited to a certain number of levels. This is because for every level there's a fresh kernel call that is made and the kernel though

launched with threads equal to the number of vertices only writes on those indices if the array which are connected at that level. This is wasteful in nature and continuous copying of data from host to device is a bottleneck in the algorithm. Also in case of a graph which is linearly connected to all vertices, a linked list, the serial and the parallel execution time would be the same.
However the beauty of the algorithm lies in the fact that it easily lends itself to complete parallelization. On the current data set the algorithm performs as desirably as one could expect.
Thus the algorithm is ideally recommended for a moderately sparse graphs. Also the algorithm scales proportionately with data.

Recommendations
The algorithm's speedup begins to improve as the number of edges goes up relative to the number of vertices, this is a minor concern because most large-scale real-world networks are sparse where the number of edges is much smaller than the maximum number of possible edges. In that sense this implementation will find more usage in streaming data where the graph is more connected.

Code Repositories

Data Set Generation: https://github.com/soumasish/GenerateDataSet
Serial BFS: https://github.com/soumasish/SerialBFS
Parallel BFS: https://github.com/soumasish/ParallelBreadthFirstSearch

Conclusion

The size of the device memory limits the size of the graphs handled on a single GPU. The CUDA programming model provides an interface to use multiple GPUs in parallel using multi-GPU bridges. Up to 2 synchronized GPUs can be combined using the SLI interface. NVIDIA QuadroPlex is a CUDA enabled graphics solution with two Quadro 5600 cards each. Two such systems can be supported by a single CPU to give even better performance than the GT 750M . NVIDIA has announced its Tesla S870 range of GPUs, with up to four cores and 6GB system memory capacity, targeted at high performance computing. Further research is required on partitioning the problem and streaming the data from the CPU to GPU to handle even larger problems. External memory approaches can also be adapted to the GPUs for this purpose.

References

1. http://impact.crhc.illinois.edu/shared/papers/effective2010.pdf
2. https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93R%C3%A9nyi_model
3. https://docs.nvidia.com/cuda/cuda-c-programming-guide/
4. https://www.sci.utah.edu/publications/Fu2014a/UUSCI-2014-002.pdf
5. https://research.nvidia.com/publication/scalable-gpu-graph-traversal