Accelerating BFS Shortest Paths Calculations Using CUDA for Internet Topology Measurements

Eric Klukovich, Mehmet Hadi Gunes, Frederick C. Harris, Jr
Department of Computer Science and Engineering
University of Nevada, Reno
Reno, USA
eklukovich@nevada.unr.edu, mgunes@cse.unr.edu, fredh@cse.unr.edu

Lee Barford
Keysight Laboratories, Keysight Technologies
USA
lee_barford@ieee.org

***Abstract*—Within the last decade, the number of devices connected to the Internet has seen immense growth and it has grown to be a large and complex network. To analyze this network, Internet topology analysis has become a popular research area and the analysis can be computationally expensive for such a large scale network. In this paper, we have implemented algorithms to find the shortest paths on large scale Internet topology graphs based on real topology data using breadth-first search. The algorithms have been implemented on graphical processing units (GPUs) using the CUDA platform. We performed our performance measurements on graph sizes ranging from 1,100 to 6.8 million nodes and achieved a maximum speed up of 47x on a single GPU and 124x speed up using 8 GPUs for 100 different starting points.**

I. Introduction

The Internet has revolutionized the way users utilize computing devices to find information and interact with each other regardless of the geographic location. The Internet is deemed as the largest man-made information system in existence and the number of users and information being sent and received is constantly growing. Currently, around 40% of the world population (3 billion people) has an internet connection [1] and over 75 thousand PB of data is transferred through the Internet every month [2]. Sending and receiving data across the world can be done in a matter of seconds, but it requires a large infrastructure to find the packet's destination and to transport it through the network.

The Internet is an interconnection of Autonomous Systems (AS) that is managed by Internet Service Providers (ISPs) and other large organizations. Each AS is assigned a group of IP addresses and manages one or multiple networks comprised of many different routers. The Autonomous Systems are organized into a two-level hierarchy. The first tier defines the protocol for routing within the AS. The second tier defines how routing to systems outside of the AS should be carried out. ASes are connected to other ASes in order to be able to connect to the entire Internet, and they use Border Gateway Protocol (BGP) to route data between them.

All of the connected autonomous systems create a large complex topology and understanding the structure and interaction has been a popular area of research. Creating an accurate topology of the Internet has many beneficial applications that can lead to new research. Vulnerability and threat analysis can be performed along with developing new protocols and simulations to see their performance. ISPs could monitor how the network has evolved over time and can analyze the economic benefits of connecting to other ASes. The data for each AS is collected and put through a topology generator in order to create the topology for the entire Internet.

The graphical processing unit (GPU) has become a popular device for creating a high performance parallel computing platform for a relatively low cost. Each GPU contains a large number of processing cores and each core can create many threads that can all be executed in parallel. As a result, many different types applications utilize GPUs to solve computationally expensive problems and have been successful in increasing the performance. NVIDIA provides their Compute Unified Device Architecture (CUDA) programming model in order for new and existing applications to be executed on the GPU. CUDA extends the standard C/C++ language to give direct access to the instructions and memory management for parallel computation on NVIDIA GPUs.

In this study, we focused on analyzing the shortest paths from the ingress points (BGP Routers) to all the routers within the individual AS using Breadth-First Search (BFS). We have implemented sequential and parallel algorithms using C++ and CUDA to perform the shortest path analysis on GPUs. Our target analyses specialized in real Internet topology data. More specifically, our analysis was for large scale networks (>1M nodes). The sequential algorithm requires a long computation time to process the large scale Internet topology graphs. We were successful in implementing single and multiple GPU algorithms using up to 8 GPUs and achieved a maximum speed up of 47x and 124x, respectively on a 6.8 million node AS with 100 ingress nodes.

This paper is organized in the following manner: Section II discusses the related work, Section III presents the methods and tools to generate the Internet topology, Section IV discusses the implementations and algorithms, Section V presents the results and discusses the findings, Section VI adds some additional discussion and future work, and we finally conclude in Section VII.

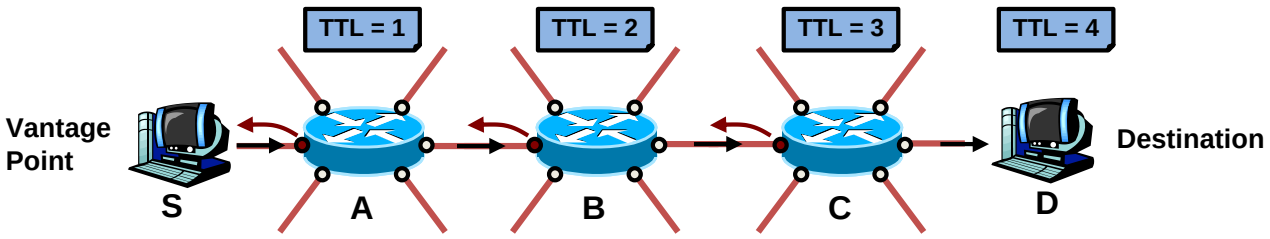

Fig. 1: Traceroute being sent from a vantage point to a destination with differnt time to live values

II. Related Work

Parallelizing BFS is a well studied area due to the amount of applications that utilize it and several studies have implemented BFS on the GPU for large graphs. Harish and Narayanan [3] provided one of the first studies to implement BFS, and other shortest path algorithms using CUDA, by processing the vertices. Their study focused on large scale random graphs (>1M nodes) and showed 20x to 60x speed up using an NVIDIA 8800 GTX GPU. The also provided minimal results using real world data, but did not achieve any speed up due to the nature of the data. Luo *et al.* [4] improved the work in [3] by adding a hierarchical queue management technique and a three-layer kernel arrangement strategy for BFS. They were able to have 2x to 6x speed up on the real world data used in [3]. Work by Singla *et al.* [5] took a different approach and parallelized BFS by processing the edges rather than the vertices for graphs with large amounts of edges. The authors used limited hardware and obtained a 5x to 11x speed up and found that the BFS is more efficient for graphs with a large number of edges. Merrill, Garland, and Grimshaw [6] implement a BFS algorithm that achieves an asymptotically optimal O(|V|+|E|) work complexity. The authors use very large real-world graphs to test their algorithm and achieved traversal rates of 3.3 billion to 8.3 billion traversed edges per second.

Fu *et al.* [7] implement BFS on a distributed GPU clusters with a total of 64 GPUs. The graph's adjacency matrix was partitioned in two dimensions based on the number of GPUs available. They used the Graph 500 generator to create large scale graphs, where the largest graph had 134 million vertices and 4.3 billion directed edges. They were able to achieve full edge-to-edge traversal in 0.148 seconds.

There have been a few other studies that focus on implementing a parallelized shortest path algorithms for Internet topology graphs. Swenson *et al.* [8] used BRITE as their topology generator and implemented all-pairs shortest path (APSP) algorithm on CUDA to analyze the routing paths within an network. This study is closely related to ours, except they implemented APSP instead of BFS. The authors were able to achieve a significant performance increase, except they performed their measurements on graphs with less than 10,000 nodes. Ahmad and Guha [9] also implement APSP on the GPU and use CAIDA as their topology generator for the graphs. They had a 2x to 35x speed up over their sequential algorithm, but again performed measurements on graphs less than 12,000 nodes.

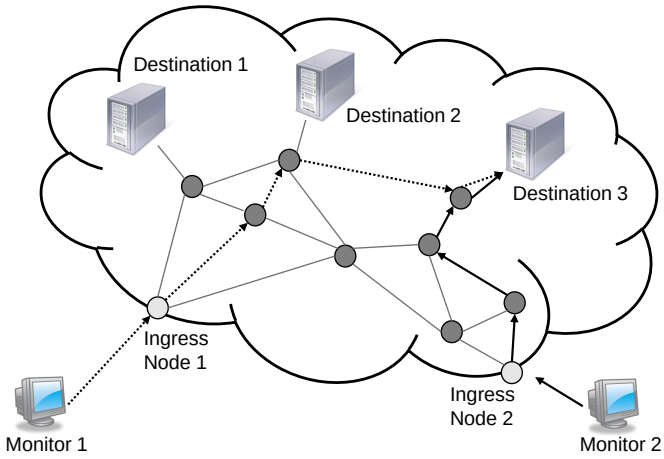

Fig. 2: Mapping an Autonomonus System subnetwork with multiple monitors

In this study, we focused on implementing BFS on the GPU in order to process large real world data graphs. We tested our implementation with higher performance hardware and implemented both single and multiple GPU algorithms. To our knowledge, this is the first study to perform this Internet topology analysis on multiple GPUs at such a large scale.

III. Internet Topology Generation

In order to generate an accurate topology of the Internet, the data about the ASes and subnetwork must be collected. The data is collected using computing systems, called vantage points, and each machine uses traceroute to send an ICMP packet towards the destination. The packet has a time to live (TTL) value that starts with an initial value and is decremented as it goes from router to router [10]. An example of how the vantage points send traceroute packets to collect the actual network path is shown in Figure 1. The vantage point *S* creates a new ICMP packet with a TTL value of 1 and sends it to the destination *D*. The packet will reach router *A* and decrements the TTL value to 0. Once the TTL value reaches 0, the router will send the packet back to the original source with the entire path the packet traveled. If the vantage point receives a packet from any router that is not the destination, then the TTL value is increased and sent to the destination again. In this example,

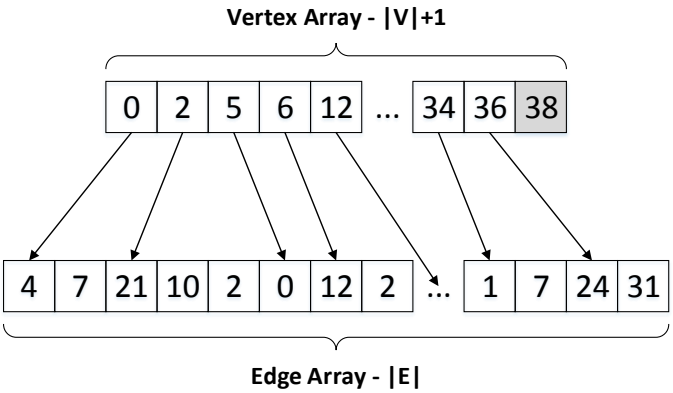

Fig. 3: GPU graph representation

S will then increase the TTL value to 2 and reach router *B*. Then, will increase the value to 3 and reach router *C*. Finally, the value will increase to 4 and reach the destination *D*. The vantage point has one path to the destination and will move to the next target.

The traceroute method just described only provides one path from the vantage point to the destination, and might skip alternative paths within the network. In order to get a more accurate map of the topology, multiple vantage points at different locations are used. An example of how multiple paths are mapped with multiple vantage points is shown in Figure 2. Monitor 1 and 2 both are mapping a path to destination 3, indicated by the dashed and solid paths, respectively. Since the monitors are in different locations, they are able to map different parts of the AS subnetwork, but do not reveal all of the possible paths in the network. This can be partially resolved by sending more packets during different times, or using additional vantage points to possibly get more of the network.

Another alternative is to use a topology generator that generates realistic Internet topologies. One such topology generator is BRITE, or the Boston university Representative Internet Topology gEnerator [11], and this is the generator we used in this study. BRITE allows for different properties such as the power law, path length, and clustering coefficient to be studied. BRITE is a parameterized generator and can use the data from the processed traceroute results. When a topology is being generated, a plane is created and filled with different nodes that act as the routers within the AS. The edges between the routers are then connected and the specific algorithm for placing connections between the nodes can be chosen such as random, preferential attachment, or heavy-tailed distribution. Bandwidths and other properties are assigned based on the data and are exported to a file to be analyzed.

IV. Algorithms and Implemenations

Breadth first searching is a commonly used algorithm in graph theory and can be applied to many different applications to process data. BFS processes tree-like structures and starts off at the root of the tree (the source) and explores the neighboring nodes first. After all the neighbors are processed, then the next levels of nodes are processed until all of the levels have been visited. This algorithm is ideal because it finds the shortest paths from a source node to all the nodes in an unweighted graph G(*V,E*). This section briefly discusses the sequential algorithm that was implemented and then provides a description of how the graph is stored on the GPU. The single and multiple GPU algorithms that are implemented are discussed in detail as well.

A. Sequential Implementation

The sequential implementation of BFS is straight forward. In order to process the graph, a method of keeping track of which nodes need to be processed is required. This is done by using a queue (FIFO) data structure, where the ingress node is used as the source and is initially enqueued. The algorithm keeps processing nodes until the queue is empty. When a node needs to be processed, all of the node's edges are checked if they have been visited. If they have not been visited, then the cost is incremented and the node is put in the queue. If the node has been visited, then it is ignored. The algorithm has a time complexity of $O(|V|+|E|)$, causing large graphs to take a long time to process.

B. GPU Graph Representation

The graph data structure is commonly stored in either an adjacency matrix for dense data, or an adjacency list for sparse data. Adjacency matrices require $O(|V|^2)$ memory to store all the edge information. This is not a feasible method for storing the graph on the GPU because of the limited amount of memory available on each device. The adjacency list overcomes the memory limitation by storing the edges for each vertex as a linked list. This method also has drawbacks because the access time can be $O(N)$ in the worst case, where N is the number of edges in the list, and would decrease the performance on the GPU.

Our graph representation is a compact form of the adjacency list based on a slightly modified implementation by Harish and Narayanan [3], also known as Compressed Spare Row format. The graph represention is shown in Figure 3. The vertices and edges in the graph is coverted into two arrays, the vertex array V_a with |V|+1 elements and the edge array E_a with |E| elements. Each element in V_a holds the index of the first connected edge in E_a. The edge array holds all the edges in the graph and each element contains the vertex number that edge is connected with. The number of connected edges for a vertex i can be easily calculated by getting the index from the next vertex $i+1$. The grey box in the figure denotes the modification we made from the implementation in [3]. By adding the additional element, it makes the calculations for the number of edges of the last vertex much simpler, and therefore avoids additional overhead in the calculations.

C. CUDA Implementation

BFS on the GPU is highly parallelizable due to the ability to perform level synchronization. This means that each level

Algorithm 1 SINGLE_GPU(*Graph (V,E)*, S_a, *numIngress*)

1: Create vertex array V_a from all the vertices and edge array E_a from all edges in *Graph(V,E)*
2: **for** $i = 0$ to $numIngress$ **do**
3: Call PROCESS_GRAPH(V_a, E_a, $S_a[i]$)
4: **end for**

Algorithm 2 PROCESS_GRAPH(V_a, E_a, *Source S*)

1: Create cost array C_a, frontier array F_a, frontier update array FU_a, visited array X_a of size *V*
2: Initialize F_a, FU_a, X_a to false and C_a to -1
3: Initialize $F_a[S] \leftarrow true,\ X_a[S] \leftarrow true,\ C_a[S] \leftarrow 0$
4: $search \leftarrow true$
5: **while** $search$ **do**
6: $search \leftarrow false$
7: Send *search* to GPU
8: Call BFS_KERNEL(V_a, E_a, F_a, FU_a, X_a, C_a)
9: Call BFS_UPDATE_KERNEL(F_a, FU_a, X_a, *search*)
10: Get *search* value from GPU
11: **end while**

Algorithm 3 BFS_KERNEL(V_a, E_a, F_a, FU_a, X_a, C_a)

1: $tid \leftarrow getThreadID$
2: **if** $tid < numVertices$ AND $F_a[tid]$ **then**
3: $F_a[tid] \leftarrow false$
4: **for each** edge *destID* in V_a **do**
5: **if** NOT $X_a[destID]$ **then**
6: $C_a[destID] \leftarrow C_a[tid] + 1$
7: $FU_a[destID] \leftarrow true$
8: **end if**
9: **end for**
10: **end if**

Algorithm 4 BFS_UPDATE_KERNEL(F_a, FU_a, X_a, *search*)

1: $tid \leftarrow getThreadID$
2: **if** $tid < numVertices$ AND $FU_a[tid]$ **then**
3: $F_a[tid] \leftarrow true$
4: $X_a[tid] \leftarrow true$
5: $search \leftarrow true$
6: $FU_a[tid] \leftarrow false$
7: **end if**

can be processed in parallel because there are no data dependencies. As each level gets processed, the previous levels will not be processed again. BFS uses the idea of a frontier which separates out which vertices have been visited and which ones have not. The frontier holds the recently visited nodes, as well as finds the nodes that need to be processed in the next level. The CUDA implementations do not use a queue in their algorithms, unlike the sequential algorithm. This is because implementing a queue on the GPU is not efficient due to having to constantly maintain index values and synchronization issues.

Single GPU: The implementation for the single GPU was initially based on the pseudocode in [3], but was modified to overcome some data race conditions. The GPU is given one ingress node to process and each device thread is given one vertex to process. The algorithm for processing all the ingress nodes is shown in Algorithm 1. The vertex and edge arrays are created from the network graph generated by BRITE. All of the ingress nodes are processed one at a time by calling PROCESS_GRAPH. This function is shown in Alogrithm 2 and is where BFS actually occurs. Three boolean arrays, frontier array F_a, frontier update array FU_a, and visited array X_a of size |*V*| are created. The frontier and frontier update arrays are used to keep track of the nodes that need to process on the current level and next level, respectively. The visited array stores the already processed nodes. An integer cost array C_a is also created. This stores the minimum cost of the path from the source to a specific node. The arrays are initialized and a loop starts to perform BFS. At each iteration, a boolean value is sent to the GPU to determine if any nodes need to be processed and both of the BFS kernels are launched. The boolean value is fetched from the GPU and is checked by the loop to determine if it needs to continue.

Algorithm 3 shows the first BFS kernel, BFS_KERNEL, and it is where the actual processing of the nodes occur. Since each thread is responsible for a vertex, the threadID is fetched and it determines which vertex to process. The thread checks if the vertex is to be processed and if so, then updates the frontier array to be processed and loops across all of its edges. At each iteration, the destination vertex is checked to see if it has been visited. If it has, then it is skipped over and not processed again. If it needs to be processed, then the cost for that vertex is updated and the frontier update array is updated to process the destination vertex on the next level. After the first kernel is finished, then the BFS_UPDATE_KERNEL shown in Algorithm 4, is invoked. This kernel is called to update the frontier array, update array, and visited arrays in order to remove any race conditions during the BFS kernel call. Each thread needs to check if a vertex has been visited and if it needs to be processed. Since CUDA does not provide block synchronization then there could be data inconsistencies when the threads read/write to these arrays. This additional kernel alleviates this issue and also provides a toggle value to quickly determine if there are more levels that need to be processed.

Multiple GPU: Each topology graph has multiple ingress nodes that need to be processed. These nodes are independent from each other and can be easily processed in parallel with multiple GPUs. Algorithm 5 shows the pseudocode for processing the ingress nodes with multiple GPUs. The process is similar to the single GPU algorithm, except the host creates a pool of threads with one thread for each GPU. The vertex and edge arrays are created only one time, and a copy is given to each thread. The thread is launched with the PROCESS_GRAPH function and is given a different ingress node to process. This is repeated until all the GPUs have an ingress node to process. The host will then wait for all the

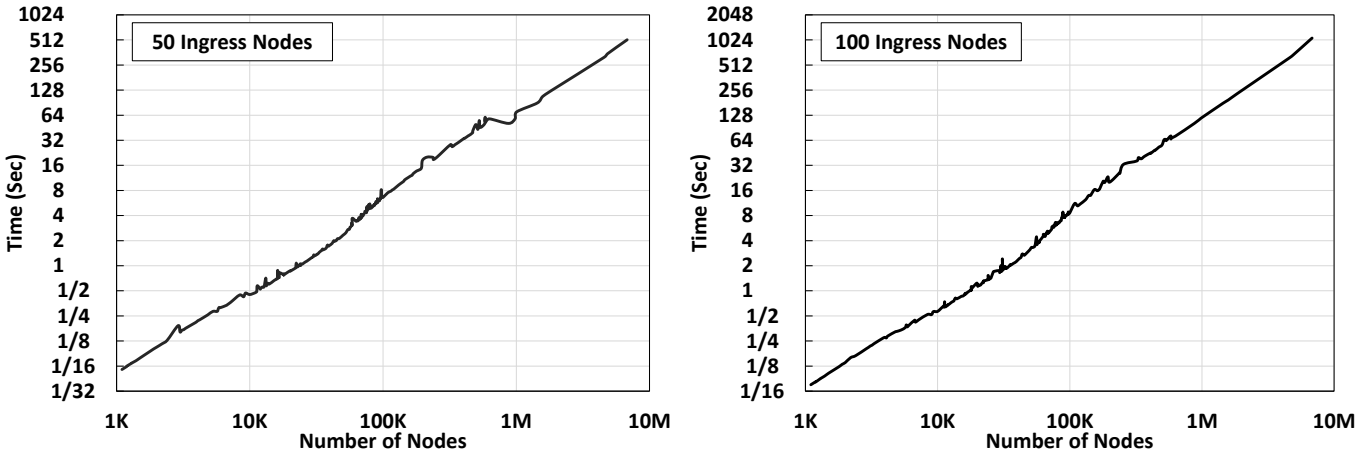

Fig. 4: Sequential execution timings for 50 ingress nodes and 100 ingress nodes

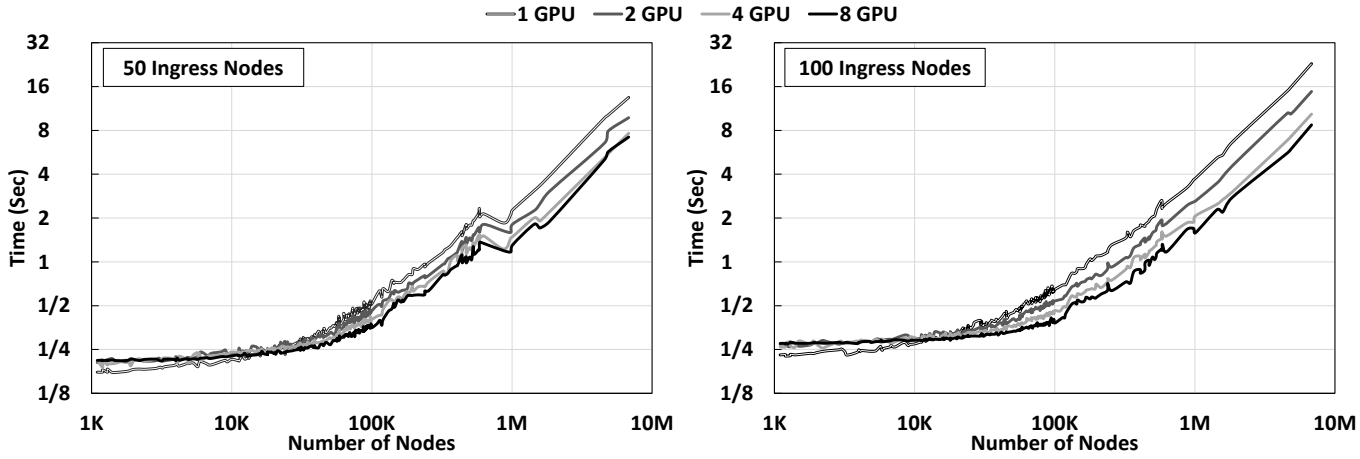

Fig. 5: GPU Execution timings for 50 ingress nodes and 100 ingress nodes

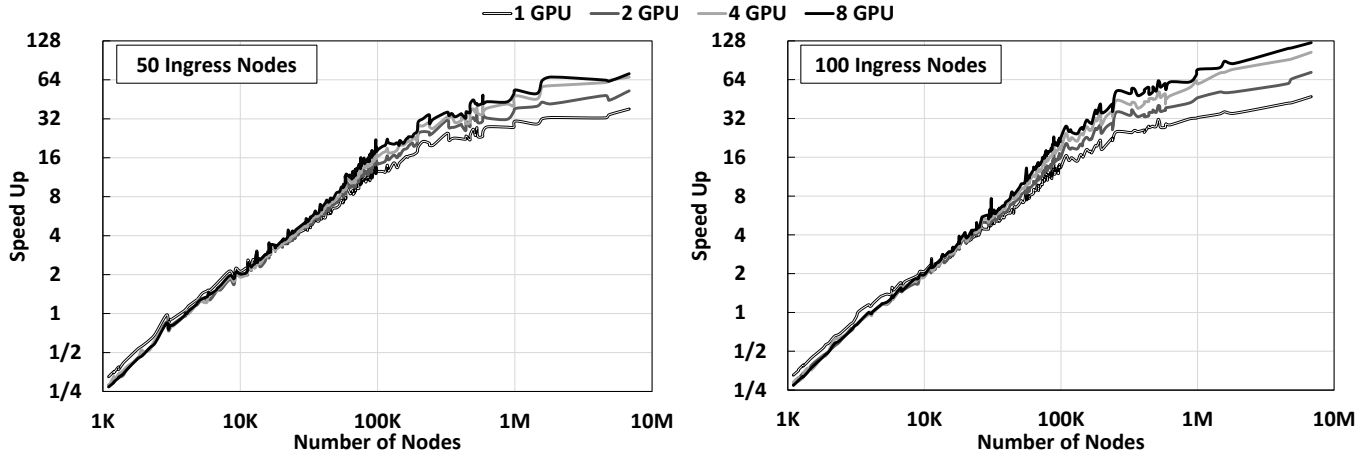

Fig. 6: Speed up performance for 50 ingress nodes and 100 ingress nodes

threads to finish their execution. The entire process repeats until all the nodes are processed and then host will again wait until all the threads are finished.

V. Perfomance Results

The performance measurements for the sequential and GPU algorithms were carried out on a PC with an Intel Xeon E5-

TABLE I: Execution time and speed up results for the largest three graphs

Execution Time (sec)									
Num Ingress Nodes	Graph Size	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs	7 GPUs	8 GPUs
50	4.6 M	9.62	6.51	5.72	5.26	4.90	6.39	4.71	4.51
	4.8 M	10.35	7.12	6.09	5.56	5.20	5.02	4.97	4.84
	6.7 M	14.28	9.58	8.20	7.67	7.06	6.86	8.20	7.65
100	4.6 M	15.98	11.19	7.92	6.93	6.36	5.96	5.95	5.55
	4.8 M	16.57	10.25	8.36	7.00	6.67	6.27	6.10	5.86
	6.7 M	24.36	14.69	12.99	10.38	9.82	9.06	8.78	8.56
Speed Up									
Num Ingress Nodes	Graph Size	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs	7 GPUs	8 GPUs
50	4.6 M	32.77	48.45	55.14	59.97	64.35	49.36	66.89	69.92
	4.8 M	31.90	46.37	54.24	59.43	63.51	65.73	66.44	68.27
	6.7 M	38.33	57.15	66.76	71.34	77.50	79.78	66.73	71.52
100	4.6 M	38.91	55.56	78.47	89.65	97.72	104.38	104.47	111.96
	4.8 M	40.14	64.90	79.54	95.06	99.78	106.07	109.12	113.53
	6.7 M	44.18	73.28	82.82	103.70	109.60	118.77	122.59	125.69

Algorithm 5 MULTIPLE_GPU(*Graph(V,E)*, S_a, *numIngress*)

```
Create vertex array V_a from all the vertices and edge array E_a from all edges in Graph(V,E)
Create thread array T, one thread for each device
i ← 0
while i < numIngress do
    if threadCount < numDevices then
        Launch PROCESS_GRAPH( V_a, E_a, S_a[i]) in a new thread
        threadCount ← threadCount + 1
        i ← i + 1
    else
        for j = 0 to numDevices do
            Wait for thread to finish
        end for
        threadCount ← 0
    end if
end while

for j = 0 to numDevices do
    Wait for the last threads to finish
end for
```

2620 CPU at 2.0 GHz, 64 GB RAM, and eight NVIDIA GeForce GTX 780 GPUs running on Ubuntu 14.04. Each of the graphic cards have 3 GB of dedicated RAM and 2304 CUDA cores available on each card. The algorithms were written using C++11, CUDA 7.0, and the C++ standard template library.

The data that was used in this study was collected by a Ph. D. student in the Computer Networking Lab at the Department of Computer Science and Engineering at the University of Nevada, Reno [12]. The data was collected using the PlanetLab research platform [13] and the traceroute method described in section III. The dataset consisted of 47,000 out of the 51,171 ASes available today [14] and the size of the subnetwork ranged from 1,100 to 6.8 million routers with about two times the number of edges. We choose to measure the BFS performance of 230 ASes, this number was chosen based on the uniqueness of the AS size and to provide a fair sampling of the data range. The number of ingress nodes for each AS varies from 1 to over 1,000. In order to be able to directly compare the performance, the number of ingresses nodes had to be normalized. The number of ingress nodes for the 230 ASes was averaged and was found to be around 100 ingress nodes. For all the performance measurements, we used 100 and 50 ingress nodes to compare the performance of the sequential, single GPU, and multiple GPU algorithms. The 50 ingress nodes were chosen to see how much of a part the number of ingress nodes affect the performance of the three algorithms, compared to the 100 ingress nodes. The number of blocks used on the GPUs were calculated based on the size of the graph. Each vertex in the graph was given a thread, and the max number of thread per block was set to 512 threads. All of the data was stored in global memory and the were organized so the accesses are coalescing.

The execution times for the sequential algorithm is shown in Figure 4, and the execution time for the single GPU and multiple GPU algorithms are shown in Figure 5. Table I shows more detail about the execution times and speed up results of the three largest graphs. The measurements utilized all 8 of the GPUs, but only the 1, 2, 4, 8 GPU results are shown for clarity. As expected, the sequential time grew in a linear trend due to the time complexity for the algorithm being *O(|V|+|E|)*. The single GPU algorithm was the second slowest and the eight GPUs was the fastest out of all the algorithms for the both the 50 and 100 ingress nodes. As more GPUs were added, the performance significantly increases.

The 50 ingress node graphs took a little over half the time to process all the nodes compared to the 100 ingress node graphs. An interesting finding was that the points where the sequential and GPU timings cross is similar for both ingress node count. The single GPU crosses approximately around

TABLE II: Speed up increase by adding additional GPUs for the largest graph of 6.8 million nodes

Num Ingress Nodes	1 to 2 GPUs	2 to 3 GPUs	3 to 4 GPUs	4 to 5 GPUs	5 to 6 GPUs	6 to 7 GPUs	7 to 8 GPUs
50	1.491	1.168	1.069	1.086	1.029	0.836	1.072
100	1.659	1.130	1.252	1.057	1.084	1.032	1.025

TABLE III: Optimazation execution time and speed up results for the largest three graphs

Execution Time (sec)								
Num Ingress Nodes	Graph Size	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs	7 GPUs	8 GPUs
50	4.6 M	7.30	6.04	4.84	4.57	4.44	4.27	4.27	4.20
	4.8 M	7.74	6.46	5.30	4.96	4.69	5.17	4.57	4.54
	6.7 M	11.75	8.64	7.48	7.15	6.76	6.70	6.56	6.53
100	4.6 M	13.62	8.74	7.54	7.16	6.04	6.21	5.85	5.10
	4.8 M	14.62	8.21	6.53	6.13	5.73	5.95	5.64	5.50
	6.7 M	20.57	13.82	10.47	9.38	8.99	8.74	8.60	8.56
Speed Up									
Num Ingress Nodes	Graph Size	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs	7 GPUs	8 GPUs
50	4.6 M	43.82	52.95	66.07	69.90	72.03	74.87	74.86	76.19
	4.8 M	50.43	60.36	73.55	78.69	83.17	75.50	85.37	86.01
	6.7 M	48.64	66.16	76.43	80.01	84.63	85.27	87.19	87.50
100	4.6 M	51.25	79.85	92.50	97.43	115.59	112.37	119.37	136.90
	4.8 M	53.08	94.49	118.91	126.72	135.43	130.54	137.65	141.18
	6.7 M	55.25	82.21	108.57	121.14	126.41	130.05	132.06	132.69

TABLE IV: Speed up increased from optimizing how the ingress nodes are processed for the 6.8 million node graph

Num Ingress Nodes	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs	7 GPUs	8 GPUs
50	1.212	1.064	1.293	1.141	1.132	1.082	1.052	1.046
100	1.269	1.158	1.145	1.122	1.092	1.069	1.307	1.223

the 3,500 node mark, and the multiple GPUs cross around the 4,000 node mark. These results are likely caused due to the lower number of vertices and the low degree per vertex (about 2-4). The lower degree forces the GPU algorithms to process the graph in a more sequential matter and therefore decreases the performance. These two intersecting points represent the threshold on which the graph should be processed on the CPU instead of the GPU. Overall, the performance showed that the GPU implementations were significantly faster compared to the sequential, and the 8 GPUs had the best performance overall.

The speed up performance for single and multiple GPU algorithm compared to the sequential algorithm is shown in Figure 6. The single GPU for the 50 ingress nodes had a maximum speed up of 38x and the 8 GPUs had the best performance with a 71x speed up with a graph of 6.8 million nodes. The single GPU for the 100 ingress nodes had a maximum speed up of 47x and the 8 GPUs had a 124x speed up for the same graph. The performance for both algorithms have a minimal amount of intersects with each other, showing that by adding additional GPUs, the performance significantly increases. The ingress nodes are independent of each other and do not have any data dependencies, therefore the graph can be copied to each GPU, where each GPU can process a unique ingress node. Overall, the 8 GPUs had the best performance, and the results also showed that by adding additional GPUs the performance increased significantly for graphs with over 100,000 nodes.

VI. Discussions and Future Work

The analysis of the topology graphs does not have data dependencies and should continue to have better performance as more GPUs are added. Table II shows how much performance is gained by adding additional GPUs to the largest topology graph. The performance difference between a single GPU to two GPUs had the largest performance increase, where the 50 ingress nodes was 1.49x, and 1.66x for 100 ingress nodes. The performance slowly decreases as more GPUs are added, but still has some additional speed up. The ideal performance for adding an additional GPU would be a 2x speed up, but waiting for the different threads to finish adds some additional overhead. As a results of this overhead, there should be a point where adding additional GPUs no longer benefits. We would like to find this threshold in future studies.

Although we achieved a significant performance increase by using both the single and multiple GPU algorithms, it would be of interest to see how well these algorithms scale on a distributed system. Each machine in the distributed system can contain between two to eight additional GPUs, providing additional speed up. Each machine can process a set of ingress nodes and could lead to better performance depending on how the work load is balanced in the system. A message passing

interface would need to be utilized to distribute the graph data to the different machines and collect the final results of the shortest paths. We would like to implement and modify the current algorithms to be able to run on a distributed system in a future study.

The current algorithms can be optimized to obtain even better performance. For the multiple GPU algorithm, the current implementation is forced to wait for all of the GPUs to finish processing their node before a new batch of nodes is given. The structure of the graph can cause certain nodes to take longer to process, therefore it can decrease the performance by making the other GPUs wait. To resolve this issue, a striding scheme can be used, where each GPU is given a set of nodes ahead of time to process. Table III shows the execution time and speed up results using the optimized method on the three largest graphs. Table IV shows the performance change from using the striding method compared to the original method for the 6.8 million node graph. Overall, the optimization showed better performance for all the GPUs with 50 and 100 ingress nodes.

The parallel performance of BFS on large scale graphs is dependent on the number of nodes and number of edges. If the number of edges per vertex is small (two to three edges) then the graph is more linear and BFS cannot take as much of an advantage of the parallel performance on the GPU. The Internet topology graphs in this study had a degree of three to four edges per vertex, and significant speed up was found. This vertex degree was chosen from observations when the dataset was compiled together. It would be of interest to determine how much more performance can be gained if the vertex degree is increased to six to ten edges. The number of vertices can also be increased to simulate larger topologies. One challenge with the GPU is the limited amount of memory. Adding more vertices and edges could cause the GPU to run out of memory. The space complexity of the algorithm on one GPU is $O(|V|+|E|)$. By changing the number of vertices will have the greatest impact on the memory, while changing the number edges will have less of an impact on the entire space, since only the edge array will be affected.

VII. Conclusion

The Internet is the largest growing man-made network and billions of users are currently connected to it. As a result, it has become a popular research area to study the structure and how it has evolved over time. We have successfully implemented single and multiple GPU BFS algorithms to perform analysis on different topology graphs. The graphs were generated with BRITE and use real AS data collected with traceroute. We measured the algorithm performance with 50 and 100 ingress nodes and had significant speed up overall. We were able to achieve a maximum speed up of 47x and 124x for a 6.8 million node graph with 100 ingress nodes using up to 8 GPUs. Overall, the multiple GPU algorithm with 8 GPUs performed the best, and the performance could easily be increased with additional GPUs.

Acknowledgment

We would like to thank Abdullah Canbaz for providing the initial Internet topology analysis application and collecting all of the data used in this study.

This material is based in part upon work supported by: The National Science Foundation under grant number IIA-1301726, and by the Cubix Corporation through use of their PCIe slot expansion hardware solutions and HostEngine. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or Cubix Corporation.

References

[1] *Internet Live Stats - Internet Users*, International Telecommunication Union (ITU), United Nations Population Division, Internet and Mobile Association of India (IAMAI), July 2014.

[2] *Cisco Visual Networking Index:Forecast and Methodology, 2013-2018*, Cisco, June 2014.

[3] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *High performance computing–HiPC 2007*. Springer, 2007, pp. 197–208.

[4] L. Luo, M. Wong, and W.-m. Hwu, "An effective gpu implementation of breadth-first search," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 52–55.

[5] G. Singla, A. Tiwari, and D. P. Singh, "Article: New approach for graph algorithms on gpu using cuda," *International Journal of Computer Applications*, vol. 72, no. 18, pp. 38–42, June 2013, full text available.

[6] 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*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 117–128. [Online]. Available: http://doi.acm.org/10.1145/2145816.2145832

[7] Z. Fu, H. K. Dasari, B. Bebee, M. Berzins, and B. Thompson, "Parallel breadth first search on gpu clusters," in *Big Data (Big Data), 2014 IEEE International Conference on*, Oct 2014, pp. 110–118.

[8] B. P. Swenson and G. F. Riley, "Simulating large topologies in ns-3 using brite and cuda driven global routing," in *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, ser. SimuTools '13. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013, pp. 159–166.

[9] M. Z. Ahmad and R. Guha, "Analysis of large scale traceroute datasets in internet routing overlays by parallel computation," *The Journal of Supercomputing*, vol. 62, no. 3, pp. 1425–1450, 2012. [Online]. Available: http://dx.doi.org/10.1007/s11227-012-0811-9

[10] H. Kardes, M. Gunes, and T. Oz, "Cheleby: A subnet-level internet topology mapping system," in *COMSNETS, 2012*, Jan 2012, pp. 1–10.

[11] A. Medina, A. Lakhina, I. Matta, and J. Byers, "Brite: An approach to universal topology generation," in *MASCOTS*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 346–.

[12] "Computer networking lab, computer science and engineering, university of nevada, reno," http://cnl.cse.unr.edu/.

[13] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: An overlay testbed for broad-coverage services," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 3, pp. 3–12, Jul. 2003.

[14] "Caida - as rank: As ranking," http://as-rank.caida.org/.