CS224W Project Final Report

CUDA Implementation of Large Graph Algorithms

Group #1

Alex Quach	Firas Abuzaid	Justin Chen
alexq@stanford.edu	fabuzaid@stanford.edu	jkchen14@stanford.edu

December 10, 2012

---

1 Introduction

Running SCC graph algorithms on large datasets can be a time-consuming task, and we spent the quarter investigating methods of parallelizing this task using CUDA. For very large graphs, too much time can be wasted by not parallelizing the graph algorithms, and we want some of the insights from our experiments to be used to speed up common graph analysis tasks. We initially started by implementing breadth-first search on the GPU, but then quickly moved on to implement a parallelized SCC algorithm. We then benchmark our parallelized SCC graph algorithm against a non-parallelized version of Tarjan's algorithm to find strongly connected components.

2 Prior Work

2.1 Motivation

After some initial research and experimentation to port Kosaraju's or Tarjan's algorithm to the GPU, we found it to be quite impossible due to the serialized nature of DFS which both algorithms required. However, we then came across a paper by Fleischer et al. which gave us an idea for how to solve this SCC finding problem in a parallelized fashion.

2.2 Fleischer et al. 2000 [1]

Fleischer et al. describe a parallelizable algorithm for strongly connected components in place of the hard-to-parallelize DFS algorithm used in Tarjan's algorithm (described in more detail in later section of this report). This divide-and-conquer algorithm obtains an expected serial complexity of $O(n \log n)$. The basic idea is to start with a random vertex v and find its descendant and predecessor sets. The intersection of these sets is shown to be the unique strongly connected component containing v. The remaining vertices are divided into three sets of Descendant, Predecessor, Remaining where any additional strongly connected component must be entirely contained within one of the three sets. So the problem divides and recurses.

The algorithm described in this paper primarily deals with acyclic graphs where any strongly connected components will usually be small. Using a topological sort with a termination detection protocol, Fleisher et al. suggest a preprocessing step to discard all graphs that have no cycles (no SCCs). The paper mostly goes through the proof of the run-time but does not report too much on the actual application results. This is something that we can potentially explore on our own because we will likely run into the issue of DFS being too hard to parallelize normally.

Finally, the paper analyzes the bottlenecks and resource consumption of their algorithm. Harish et al. found that memory was the bottleneck of their algorithm, but this paper found that GPU core computation was the bottleneck, indicating that their algorithm was structuring reads in a way more amenable to GPU memory layouts.

3 Background Theory

In order to provide some knowledge as to why Tarjan's and Kosaraju's proved impossible to port, we researched some background theory as to how these algorithms are implemented.

3.1 Parallelization of Breadth-First Search (BFS)

The parallelization of BFS is what we initially began with since we discovered that BFS lends itself well to parallelization. Prior research has shown that there are two common yet distinct strategies for parallel (and distributed) execution of BFS: the *fixed-point* algorithm and the *level-synchronous* algorithm.

The fixed-point algorithm 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 preferred in parallel environments since one can take advantage of message passing between neighboring nodes. However, the downside to the fixed-point algorithm is the potential waste of computation, since the same edge will be processed multiple times whenever a corresponding node is updated.

The level synchronous algorithm instead 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 (again 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.

Both these strategies are typically used to parallelize breadth-first search. Other optimizations including using a bitmap to encode the set of visited nodes and the use of local next-level queues.

3.2 Parallelization of Depth-First Search (DFS)

Depth-first search, on the other hand, has proven to be much more onerous to parallelize than breadth-first search. Prior research has shown that developing parallelized algorithms on DFS (which run in sublinear time) is notoriously difficult — many initially believed that DFS is an inherently sequential process and that no such algorithms existed [8].

In the literature, Reif shows that a restricted version of the problem (lexicographical DFS) is P-Complete, but others, such as Aggarwal and Anderson, have described RNC algorithms for performing DFS on both undirected and directed graphs. However, the expected running time of this randomized algorithm is $O(log^7 n)$ and it requires an impractical $n^{2.376}$ processors. Moreover, other attempts require further restrictions to the graphs under question; for example, Chaudhuri and Hagerup studied the problem only for acyclic and planar graphs. Consequently, SCC algorithms which leverage DFS (such as Tarjan's algorithm) cannot be used.

3.3 Kosaraju's Algorithm

This algorithm to find strongly connected components of a directed graph is quite simple in a non-parallel implementation. It works by initializing a stack S along with the given graph G. While S does not contain all the vertices, it chooses an arbitrary vertex v not in S and performs depth-first search (DFS) starting at v. Each time v finishes expanding a vertex u, vertex u is pushed onto the stack. When this is complete, all the arcs' directions are reversed to obtain the transpose graph. Popping the top vertex v from S and then running DFS will create a set of visited vertices that gives a strongly connected component (SCC) containing v. Remove all visited vertices that form this SCC from the stack and graph and continue until all are found.

An important note is that we can actually use BFS for the second pass instead of DFS, but the algorithm in general is not as efficient as Tarjan's which we describe below.

3.4 Tarjan's Algorithm

Tarjan's algorithm for finding strongly connected components can actually be viewed as an improved version of Kosaraju's algorithm. The premise of the algorithm is as follows:

We begin by running a DFS from an arbitrary (unvisited) start node. The search does not explore any node that has already been explored. The strongly connected components then form subtrees of the search tree. The nodes are placed on a stack as they are visited. When the search returns from the subtree, the nodes are popped off the stack and checked to see if they are a root of a strongly connected component. If the node is a root, then it and all the nodes taken off before it form the strongly connected component.

The algorithm is called once for each node and the for-all statement considers each edge at most twice. The algorithm's running time is therefore linear in the number of edges in G.

Unfortunately, because Tarjan's algorithm leverages DFS considerably, it proved difficult and perhaps impossible to implement through CUDA, as DFS is much more difficult to parallelize than BFS. Moreover, BFS cannot be used

as a substitute for DFS in the algorithm, as was the case for Kosaraju's algorithm, because the algorithm depends on the recursive backtracking of DFS.

4 Method Approach

Using Fleischer et al. as our guide, we came up with our own approach to finding strongly connected components in graphs. It borrows some ideas from the paper that allow for a parallelizable implementation of finding SCCs.

4.1 Implementation Details

4.1.1 Preprocessing Step

All nodes with no incoming or outgoing edges instantly become their own SCC since this case is trivial. We run a kernel with a thread assigned to each vertex, which marks the vertex as prunable if it has 0 in or out degree. The host then assigns each to its own SCC, then reruns the kernel again. Nodes with SCCs assigned are disregarded from the degree calculation, because they are effectively considered removed from the graph. Because of this, each subsequent run of the kernel can continue to prune new nodes. Once the kernel can't prune any more nodes, the implementation moves on to the main algorithm.

4.1.2 Algorithm

The divide-and-conquer algorithm for strongly connected components described in the paper by Fleischer et al. is run on multiple vertices at the same time and on all vertices simultaneously.

We choose 10 pivots at random, then do concurrent forwards and backwards BFS originating from each pivot node until one of the forward and backward searches has concluded. We then create the SCC containing the finished pivot vertex by finding the intersection of the predecessors of the vertex and the descendants of the vertex. Then, we remove the nodes assigned to the SCC (setting a boolean) and choose another pivot at random, not taking into account the nodes that have already formed an SCC. This repeats until there are no more nodes without assigned SCCs.

We use both the forwards and backwards graph while running the BFS algorithms. Each thread handles a BFS from the forwards and backwards directions for a single vertex, but computes it for each BFS search happening (one per pivot vertex).

Pseudocode

Let m = number of pivots, N = number of vertices, FV and FE be vertices and edges of forward graph, BV and BE be vertices and edges of backwards graph.

Initialize frontier boolean arrays FF and BF with mN entries, boolean array SCC with N entries (entry i is true if vertex i has been assigned a SCC), successor and predecessor boolean arrays S and P (true if forward or backward BFS respectively originating from pivot j can reach vertex i) with mN entries.

Kernel pseudocode

```
Prune(FV, FE, BV, BE, SCC): # Run on all vertices in parallel
    i = thread index
    if i >= N
        return
    if SCC[i]
        return
    OD = number of neighbors of i in FV, FE, ignoring neighbors j with SCC[j] = true
    ID = number of neighbors of i in BV, BE, ignoring neighbors j with SCC[j] = true
    if OD == 0 or ID == 0:
        SCC[i] = true

BFS (i, j, V, E, F, M, SCC):
    if not F[j][i]:
        return
    F[j][i] = false
    M[j][i] = true
    for v in neighbors of i: # computed using V and E
        if not SCC[v] and not M[v]:
            F[j][v] = true

SCCFinderKernel(FV, FE, BV, BE, FF, BF, S, P, SCC): # Run on all vertices in parallel
    i = thread index
```

```
    if i >= N
        return
    if SCC[i]
        return
    for j = 0 to m
        BFS(i, j, FV, FE, FF, S, SCC) # One iteration of forward BFS
        BFS(i, j, BV, BE, BF, P, SCC) # One iteration of backward BFS
```

Main Algorithm

```
while true:
    SCCFinderKernel(...)
    for j = 0 to m
        if FF[j][i] = false and BF[j][i] = false for all 0 <= i < N:
            # Forwards and backwards BFS complete, compute SCC
            create new SCC
            for i in 0 <= i < N:
                if S[j][i] and P[j][i]:
                    assign i to SCC
                    SCC[i] = true
            Choose new vertex v with unassigned SCC
            if v = null:
                return # No unassigned nodes, algorithm complete
            S[j][i] = P[j][i] = false for 0 <= i < N # Reset state
            FF[j][v] = BF[j][v] = true # Start new BFS from v
```

This pseudocode omits the transferring of data between host and device, which made the implementation significantly more complicated. We copy only when absolutely necessary, keeping track of when arrays have changed so that we can optimize memory transfer as much as possible. For example, we never copy the SCC array from the device to the host because it is never modified on the device, and we only copy it to the device in an iteration if it has been changed.

4.2 Runtime Analysis

Our algorithm runtime depends heavily on the size and density of the graph, and on the number of pivots chosen. There is a tradeoff on the number of pivots – more pivots increases the speed at which SCCs are found, but also increases the memory load (the two frontier arrays and the successor/predecessor arrays are mN in size) and therefore the time spent moving this data back and forth between device and host, a common CUDA bottleneck. Also, creating too many pivots may duplicate work, because two pivots in the same SCC will do identical work.

In the best case, every chosen pivot vertex would be in a different SCC, so that there would be no wasted work. This is common in sparser graphs and uncommon in denser graphs, so our algorithm does less wasted work on sparser graphs. We also benefit from the pruning, so our algorithm is generally much, much faster on sparser graphs. On denser graphs, we duplicate effort between pivot vertices, but because our BFS happens concurrently at every vertex, we can traverse whole SCCs in just n iterations, where n is the diameter of the SCC. This keeps us competitive against serial algorithms on denser graphs, even when we duplicate effort between pivot vertices.

A worst case would be a large number of small, nontrivial SCCs (size > 1). This means that our pruning algorithm would not catch these SCCs, and that we would need to eventually assign pivot vertices to each one. For x mutually disconnected SCCs of size y, we would require $x/m * y$ iterations to detect them all. This wastes no CPU time, but incurs a lot of unnecessary copying. Fortunately, few graphs are structured this way, due to the relative ease at which SCCs in graphs coalesce into larger SCCs. Most graphs we encountered had one large SCC, a few smaller SCCs, and a lot of trivial SCCs, which worked well for our algorithm, because the trivial SCCs are pruned and the large SCC is detected quickly. Without the pruning step, our speed would suffer greatly because of the trivial SCCs.

Another worst case is a series of SCCs joined by edges going in only one direction, like this:

O -> O -> O

where each O is a SCC. Pivot vertices chosen in the leftmost cluster would BFS through all the nodes in the graph, but only be able to count vertices in the first cluster as in the same SCC. This is an unfortunate consequence of the speculative, distributed nature of our algorithm compared with the precise Tarjan's algorithm, which visits vertices only once. However, as mentioned above, graph structures like these are relatively uncommon, since just one edge from the second or third SCC to the first or second one would coalesce them into one or two SCCs.

4.3 Benchmarking

In order to see how well our algorithm performs against a non-parallelized implementation, we benchmarked our timing results against a simple C++ implementation of Tarjan's algorithm on the same graphs.

4.4 Hardware Used

We implemented our algorithms on the Myth CUDA-enabled machines (20-32), specifically myth24, which has the following specs:

CPU: 2 x Intel(R) Core(TM)2 Duo CPU E6850 @ 3.00GHz
GPU: NVIDIA GeForce GTX 560 Ti with 384 CUDA cores

5 Experimental Results

5.1 Results

ID	N	M	C++ (ms)	Parallelized CUDA (ms)	Ratio	Prune (ms)	Device Init (ms)	Device I/O (ms)	Host (ms)	Kernel (ms)
1	5242	28920	837	766	1.09	0	165	575	173	22
2	85591	156217	3049	1575	1.94	23	200	1403	722	8
3	10670	22003	175	187	0.94	14	149	0	0	0

Table 1: Real-world network comparison of our CUDA SCC algorithm with standard serial version of Tarjan's algorithm (C++) and breakdown of time spent in various components of CUDA algorithm. 1. Collaboration Network, 2. Email Network, 3. Internet Network [from homeworks 1 and 3] All times are obtained through the average of 5 runs.

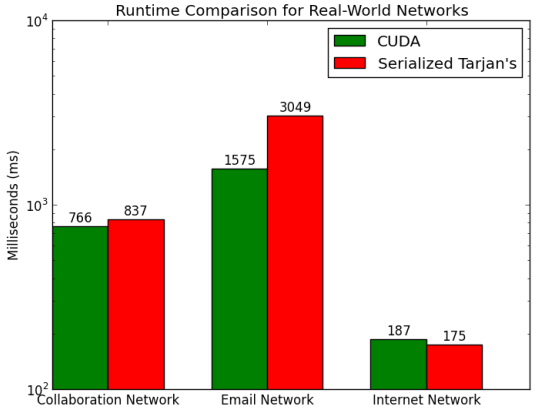

Figure 1: Comparisons of our CUDA SCC algorithm vs serial implementation of Tarjan's algorithm in C++ for a few real-world networks

N	M	C++ (ms)	Parallelized CUDA (ms)	Ratio	Prune (ms)	Device Init (ms)	Device I/O (ms)	Host (ms)	Kernel (ms)
100	100	0	167	0.00	0	174	0	0	0
100	130	3	189	0.02	0	195	2	0	0
100	200	1	204	0.00	0	199	0	0	0
1000	2000	8	175	0.05	0	163	4	0	1
10000	1000	131	145	0.90	4	145	0	0	0
10000	5000	152	176	0.86	6	160	0	0	0
10000	10000	150	233	0.64	2	197	0	0	0
10000	100000	22	194	0.11	5	175	10	9	0
10000	1000000	32	230	0.14	15	193	52	1	0
100000	1000	1189	197	6.04	9	158	0	0	0
100000	5000	1225	167	7.34	4	154	0	0	0
100000	10000	932	154	6.05	1	159	0	0	0
100000	100000	1170	280	4.18	44	155	59	27	0
100000	1000000	267	512	0.52	14	155	66	42	0
1000000	1000	11694	298	39.24	23	163	0	0	0
1000000	5000	10199	246	41.46	18	162	0	0	0
1000000	10000	10218	258	39.60	19	144	0	0	0
1000000	100000	8880	345	25.74	43	149	0	0	0
1000000	1000000	12589	1300	9.68	510	212	358	194	0
1000000	2000000	6504	2948	2.21	243	281	632	340	1
1000000	3000000	3926	3240	1.21	153	326	534	555	2

Table 2: Erdos-Renyi comparison of our CUDA SCC algorithm with standard serial version of Tarjan's algorithm (C++) and breakdown of time spent in various components of CUDA algorithm. All times are obtained through the average of 5 runs.

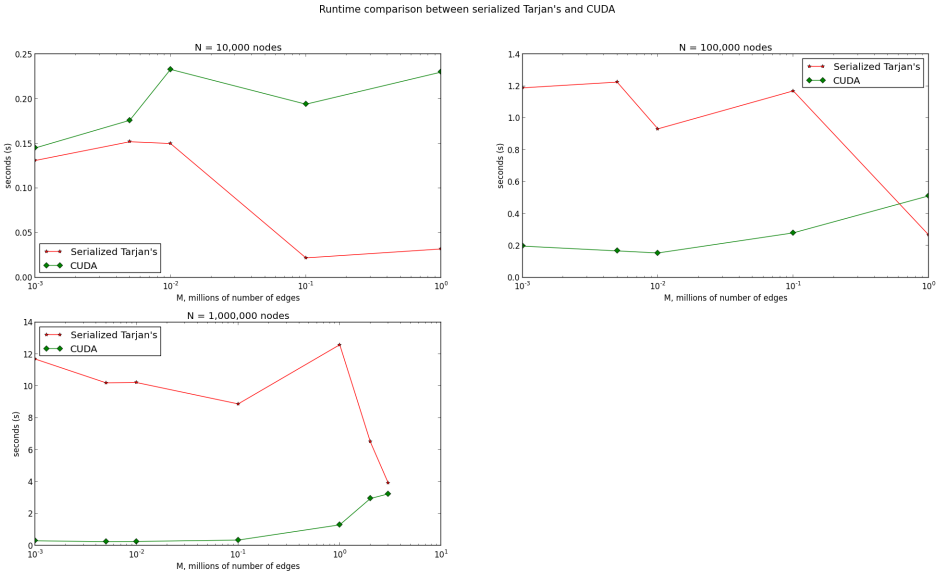

Figure 2: Comparisons of our CUDA SCC algorithm vs serial implementation of Tarjan's algorithm in C++ for varying number of nodes and edges

5.1.1 Table Information

N: number of nodes in graph, **M**: number of edges in graph, **Ratio**: speedup ratio of C++ implementation vs CUDA implementation (C++ runtime/CUDA runtime), **Prune**: time spent removing nodes via algorithm described above, **Device Init**: time spent for initial copy of graph to device and mallocing all the data necessary, **Device I/O**: time spent copying stuff in between device and host for each iteration of the algorithm, **Host**: time spent computing on CPU, **Kernel**: time spent running on GPU (typically too fast to measure)

5.2 Analysis of Results

As seen in our figures, our parallelized CUDA algorithm demonstrates a significant improvement on run-time especially for the case of graphs with a high number of nodes and low number of edges. We are even capable of achieving a speedup of approximately 4,000% depending on the characteristics of the graph. However, the algorithm still performs rather poorly on relatively small Erdos-Renyi graphs similar to the parallelized BFS work we showed during the milestone, because of the overhead of transferring data to and from the device.

We also notice that the run-time for the standard C++ implementation of Tarjan's algorithm actually decreases when the number of edges increases. It takes less time to complete when the graph is denser. This is likely due to the DFS nature of Tarjan's algorithm. It is able to go through the entire graph in one go without backtracking since it's all in one SCC as opposed to doing more pushing/popping with sparser graphs.

Although our original parallelized BFS work did not show a significant improvement during the milestone report, we notice that when implemented as part of a more complicated algorithm as we have done here, the parallel benefits of CUDA begin to show especially as the number of nodes increases in the graph. The pre-processing parallel pruning step begins to cut down on the run-time drastically especially if the number of nodes with in/out-degree of 0 is large (many nodes, few edges). The more edges, relative to nodes, that the graph has, then the less the parallel pruning helps with the run time.

It is clear that the majority of the time spent is on device initialization, host computation, and memory transfer. Device initialization is unavoidable, so we ignored that term for the purposes of optimization. We devoted the majority of our time after completing the algorithm to optimizing the latter two categories, by computing as much as possible on the kernel. This has the advantage of speeding up the operation as well as reducing memory transfer, because any computation on the kernel doesn't require transferring data to/from it more than once, whereas a host computation for each iteration would require a transfer from the kernel, a computation, and a transfer back. Unfortunately, this greatly complicates the algorithm, so we chose not to implement it for this version of the algorithm.

5.3 Graph Reasoning

Although our algorithm speedup begins to lessen as the number of edges goes up relative to the number of nodes, this is not a major 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.

Network	Number of Nodes	Average Degree
WWW (Stanford-Berkeley)	319717	9.65
Social networks (LinkedIn)	6946668	8.87
Communication (MSN IM)	242720596	11.1
Coauthorships (DBLP)	317080	6.62
Internet (AS-Skitter)	1719037	14.91
Roads (California)	1957027	2.82
Protein (S. Cerevisiae)	1870	2.39

Table 3: Example real-world network sparsity characteristics.

We also chose to use the Erdos-Renyi approach (GNM) to generating graphs as opposed to the Coefficient Watts-Strogatz model (CWS) because GNM can generate directed graphs that aren't initially well connected. CWS makes connected SCCs no matter what parameters you choose, so GNM allowed us to more precisely control SCC behavior by varying the number of edges we added in.

6 Conclusion

CUDA provides a great benefit when implementing large-scale graph algorithms. However, the biggest difficulty lies in finding graph algorithms that can be parallelized. We learned that anything which relies heavily on a DFS approach still remains mostly non-parallelizable. We were able to come up with a BFS approach to finding SCCs, however, that rivals Tarjan's serial implementation.

One major benefit of CUDA that we found is the ability to pre-process the nodes very efficiently. This can help speed up many algorithms because the number of nodes that can be processed at once goes up with the number of processors on the GPU. The resulting graph is potentially many times smaller and less complicated than the original

one. In addition, since real-world graphs typically have many isolated components, CUDA can analyze them each separately quite effectively.

When working with small graphs, it is still typically wiser to use an optimized serial approach because of the overhead of transferring data between the CPU and GPU, but CUDA definitely provides an interesting approach to large scale graph algorithms that we are sure to explore more in the future.

7 Future Work

- Moving more work to the GPU to speed up the algorithm by reducing memory transfer time, which would necessitate even more complex parallelism. For example, one could detect the completion of BFSes on the kernel, which would give negligible computational speedups, but would cut the memory transfer per iteration by at least 50% because the forward and backward frontiers wouldn't need to be transferred back and forth between host and device.
- Reducing memory transfer time by compressing the data transferred between device and host. Currently, we use vectors of bytes to store booleans, but a bit vector would suffice, increasing memory storage by 8x and probably speeding up memory transfer by 8x as well. Unfortunately, our bit vector implementation behaved erratically on the CUDA kernel, so we were forced to use conventional byte vectors. Further investigation may reveal another way to use bit vectors in a way amenable to CUDA.
- Obtain access to higher-grade hardware so we can further stress-test our algorithm with even larger graphs (capable of storing more nodes/edges).

8 References

[1] L. Fleischer, B. Hendrickson, A. Punar, "On Identifying Strongly Connected Components in Parallel" http://ipdps.cc.gatech.edu/2000/irr
2000

[2] Pawan Harish and P.J. Narayanan "Accelerating Large Graph Algorithms on the GPU Using CUDA" 2007.

[3] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. "Efficient Parallel Graph Exploration on Multi-Core CPU and GPU" 2011.

[4] Frank Dehne, Kumanan Yogaratnam. "Exploring the Limits of GPUs With Parallel Graph Algorithms" 2010.

[5] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. "A scalable distributed parallel breadth-rst search algorithm on BlueGene/L," in SC 2005 ACM/IEEE.

[6] V. Agarwal, F. Petrini, D. Pasetto, and D. Bader, "Scalable Graph Exploration on Multicore Processors," in Supercomputing. Proceedings of the ACM/IEEE SC 2010 Conference.

[7] Nicholas Pippenger. On simultaneous resource bounds (preliminary version). In Proc. 20th IEEE Symposium on Foundations of Computer Science, pages 307-311, 1979.

[8] Jon Freeman. Parallel algorithms for depth-first search. Technical report, University of Pennsylvania, 1991.