ACCELERATION OF FORWARD-BACKWARD ALGORITHM IN CUDA

Muhammad Uzair Amin, Muhammad Talha', Ayyan Bin Fawad

Habib University, Block 18 Gulistan-e-Jauhar, 75290 Karachi, Pakistan

ABSTRACT

The detection of strongly connected components in a graph is a fundamental yet essential graph problem present in many scientific and commercial applications where graphs are involved. In this paper, we examine the Forward-Backward (FB) algorithm as converted from serial to parallel by [1] and attempt to further optimize its execution. In particular, we use CUDA streams, introduced in 2010, to concurrently run independent elements of the FB algorithm concurrently and hence accelerate the existing implementation. We have also experimentally shown that using a single GTX 1650 max we can further optimize the FB algorithm.

I. INTRODUCTION AND BACKGROUND

Fundamental graph problems such as breadth first search, depth first search, minimum spanning tree, etc.. are essential algorithms that serve as building blocks for other algorithms involving graphs. Generally, implementations of all these basic algorithms have well documented and implemented serial versions and while these suffice for graphs with relatively few numbers of nodes and edges, real world areas where graphs are involved usually contain graphs with the number of nodes and edges in several millions. When large graphs like these are encountered, serial implementations of the fundamental algorithms fail in that their execution often becomes extremely lengthy and thus, parallel implementations of these algorithms have been devised in order to optimize the algorithms that utilize them. While converting an algorithm into parallel is not a simple task it is often well worth it due to the performance increase and efficiency they provide. Converting an algorithm from serial to parallel can become quite complex and this is enhanced by the fact that some algorithms do not lend easily to parallel implementations. Depth first search (DFS) is one particular example that comes to mind as it is an inherently sequential algorithm and while parallel versions of DFS have been studied exhaustively [2], there is most likely no efficient solution for the DFS problem [3]. Conversion of serial algorithms to parallel requires the effective utilization of resources provided by a general purpose graphical processing unit (GP GPU). GP GPUs are used alongside a central processing unit (CPU) to offload a portion of the work done by a CPU, perform the relevant operations on it and return the computed data back to the CPU. A GP GPU can be defined as a specialized electronic circuit that contains hundreds of arithmetic units which are designed to rapidly process complex and parallel computations found in fields such as Gaming, Computer Graphics, Simulation, etc..

In this paper, we focus on the problem of accelerating the Forward-Backward (FB) algorithm for the decomposition of a graph into its Strongly Connected Components (SCCs). The problem of SCC decomposition can involve extremely large and dense graphs and accelerating the FB algorithm would prove useful in many scientific and commercial applications. The existing FB algorithm, as defined by [1], is already an efficient algorithm for the SCC decomposition problem. However, it does not utilize an extremely strong resource provided by CUDA for its acceleration. The current implementation does not utilize streams in CUDA which allow for a different, independent sequences of commands to run concurrently. In this paper, we describe how to utilize CUDA streams to our advantage to perform independent portions of the FB algorithm simultaneously for further acceleration. We also compare our results with the previous implementation and show the speed up achieved

II. INTRODUCTION TO THE PROBLEM

A. Preliminaries

The SCC (Strongly Connected Components) problem is a classic problem in graph theory that involves identifying groups of nodes in a directed graph that are strongly connected to each other. In other words, it seeks to identify the maximal subsets of vertices in a directed graph where every vertex in the subset is reachable from every other vertex in the subset.

Formally, given a directed graph $G = (V, E)$, the SCC problem involves finding all the SCCs of G. An SCC of G is a maximal subset of vertices $C \subseteq V$ such that for every pair of vertices $u, v \in C$, there exists a directed path from u to v and from v to u in G. The in-degree and out-degree of a vertex v is the number of immediate predecessors and immediate successors of v respectively.

B. Applications

The SCC problem has a wide range of applications in real-world scenarios, including:

- Compilers: In the process of compiling a program, the code is typically transformed into an intermediate representation that is represented as a directed graph. The SCC problem can be used to identify subgraphs of this graph that correspond to loops in the program.
- Social networks: In a social network, individuals can be represented as nodes in a graph, and their relationships can be represented as directed edges. The SCC problem can be used to identify groups of individuals who are strongly connected to each other, which can be useful for targeted marketing or identifying communities.
- Image processing: In image processing, images can be represented as graphs, with pixels or image regions as nodes and edges representing the relationships between them. The SCC problem can be used to identify regions of an image that are connected to each other, which can be useful for segmentation and object recognition.
- Network analysis: In network analysis, graphs can be used to model complex systems such as transportation networks or biological networks. The SCC problem can be used to identify subnetworks within these larger networks that are strongly connected, which can be useful for understanding the behavior of the system as a whole.

III. PREVIOUS WORK

There have been multiple attempts at parallelizing SCC decomposition, however some of the most notable algorithms for this problems are inherently serial and do not lend easily to paralellization. For example, both Kosarajus's and Tarjan's Algorithm for SCC decomposition are based on DFS, which is famously difficult to serialize [4][5].

However, the FB algorithm uses breadth first search (BFS) instead which has shown to be parallelizable [1]. Work on parallelizing the FB algorithm has been centered around two papers,[6][1].

The work of Jiri Barnat [1] is centered around implementing the algorithm in a CUDA environment, by parallelizing the BFS part of the code, however, they use a modified version of the BFS to run on the device. One of the algorithms explored in [1] is the FB algorithm and it proceeds as follows. A vertex called a pivot is selected and the strongly connected component the pivot belongs to is computed as the intersection of the forward and backward closure of the pivot. After the closures have been computed, the graph is divided into 4 subgraphs on which the FB algorithm can again be applied to further discover more SCCs. The 4 subgraps created are 1) the strongly connected component with the pivot, 2) the subgraph given by vertices in the forward closure but not in the backward closure, 3) the subgraph given by vertices in the backward closure but not in the forward closure, and 4) the subgraph given by vertices that are neither in the forward nor in the backward closure. The subgraphs that do not contain the pivot form 3 independent instances of the same SCC decomposition problem and hence can be recursively processed in parallel with the same algorithm [1]. The pseudocode of the FB algorithm is as described in algorithm 1.

Algorithm 1 FB Algorithm

```
1:  procedure FB(V)
2:      if V ≠ ∅ then
3:          pivot ← PIVOT(V)
4:          F ← FWD(pivot, V)
5:          B ← BWD(pivot, V)
6:          F ∩ B is SCC
7:          in parallel do
8:              FB(F \ B)
9:              FB(B \ F)
10:             FB(V \ (F ∪ B))
11:         end in parallel
12:     end if
13: end procedure
```

Algorithm 2 F-KERNEL(G, visited, terminate)

```
1: for all v ∈ V do
2:     if visited[v] = true then
3:         for all u ∈ V, (v, u) ∈ E do
4:             if v ∼ u and visited[u] = false then
5:                 visited[u], terminate ← true, false
6:             end if
7:         end for
8:     end if
9: end for
```

Algorithm 3 FWD-REACH - host code

```
    In: G = (V, E), P ⊆ V
    Out: ∀v ∈ V: visited[v] = true ⇔ ∃u ∈ P : (u, v) ∈ E*
1:  for all u ∈ P do
2:      visited[u] ← true
3:  end for
4:  terminate ← false
5:  while terminate = false do
6:      terminate ← true
7:      F-KERNEL(V, visited, terminate)
8:  end while
```

Algorithm 2 is also provided in [1] and it is a CUDA kernel function that serves as an alternate to BFS. It is a CUDA

kernel that employes a different thread for each vertex and in the case of the forward closure, each thread checks if the corresponding vertex is within the closure set, and if so, it sets the presence bit for all immediate successors of the vertex. Barnat has used it in Algorithm 3 which is used to calculate the forward and backward closure of the pivot. The result of the computation of a closure procedure is a vector $visited$ of $|V|$ bits indicating which of the vertices belong to the closure. Another interesting discovery by Barnat was that in Algorithm 1, Barnat has mentioned the used of a BWD algorithm in line 6 for computing the backward closure of the pivot. However, he discovered that transposing the graph and running FWD on the transpose of the graph shows "almost exclusive performance dominance" over the BWD algorithm. This is the FB algorithm and its supplementary functions as described by [1].

The work of Chen Xuhao [6] revolves around modifying the algorithm to create an improvement in real-world graphs that can exhibit a skewed structure. The modified algorithm was implemented in CUDA, showed a considerable improvement over the traditional algorithm in skewed graphs, however, the algorithm was significantly slower for non-skewed data.

This paper focuses on the further acceleration of the FB algorithm provided by Barnat in [1] using CUDA streams. The following sections of the research paper include a section for Methodology; this section highlights what streams are and how we will theoretically use them in the FB algorithm to attempt to obtain an overall more efficient implementation, Implementation details; this section will contain pseudo code to show how streams have actually been utilized in our implementation, Experimental Results; this section will explore how our FB algorithm with streaming performed when compared to the FB algorithm without streaming.

IV. METHODOLOGY

CUDA streams are the main concept that we have utilized in our attempts at accelerating the existing FB algorithm. Streams in CUDA are a sequence of operations that execute in order on a GP GPU. Streams in CUDA are of 2 types, Default and Non-Default. All operations that are submitted by the host to the device without specifying a stream to run them on are run on the Default stream. This stream is blocking and stops execution in all other streams while it runs the sequence of operations provided to it. Non-Default streams must be created, specified if they are being used when launching a CUDA kernel and must be deleted after their use has been completed. These Non-Default streams are non-blocking and can concurrently execute the sequence of operations provided to them alongside other Non-Default streams as long as there are no data dependencies or race conditions.

Theoretically, in the Fb algorithm as in Algorithm 1, lines 4 and 5 constitute the bulk of the computational work that the algorithm performs. Moreover, the operations done in lines 4 and 5 are independent of each other as we are creating a second graph which is a transpose of the first one and running the FWD procedure on it as it gives a faster result than using the BWD function as discovered by Barnat[1]. Hence, as lines 4 and 5 of Algorithm 1 are independent of each other, they can theoretically be executed on different streams.

In our study, we evaluated the performance of our algorithm on a set of 6 different graphs. The graphs were generated using the GTGraph tool.The graphs varied in size from 32 to 256 vertices and were categorized as dense, sparse, or medium, based on their edge densities. Specifically, the density of a graph was defined as the ratio of its number of edges to the maximum possible number of edges, i.e., density = number of edges / (number of vertices choose 2).

We then ran our algorithms on each of these graphs and recorded the running time. The purpose of this experiment was to evaluate the efficiency of our algorithm on a range of graph sizes and densities, and to determine its suitability for practical applications.

V. IMPLEMENTATION DETAILS

Algorithm 4 FB Modified Algorithm

```
1:  procedure REACH(Pivot,V)
2:      if V ≠ ∅ then
3:          pivot ← PIVOT(V)
4:          in parallel do
5:          F ← FWD(pivot, V)
6:          B ← BWD(pivot, V)
7:          end in parallel
8:          F ∩ B is SCC
9:          in parallel do
10:             FB(F \ B)
11:             FB(B \ F)
12:             FB(V \ (F ∪ B))
13:         end in parallel
14:     end if
15: end procedure
```

Our version of the modified FB Algorithms runs the kernels involved in FWD and BWD reach asynchronously at the same time. To do this the FWD and BWD functions are merged into a single host-wrapper function which calls both the kernels and then returns the visited vertices by both the reaches. The kernels are run asynchronouly on separate non-null streams in the Reach function which offers the speed up.

VI. EXPERIMENTAL RESULTS

We used the $SSCA\#2$ graph generator from the *GT-Graph Suite* to generate a dataset for our analysis. The dataset includes multiple graphs with differing sizes, both in terms of

number of edges and number of vertices. The smallest graph starts with 32 nodes and the number of nodes doubles until we reached 128 nodes. For each node category there are 3 graphs of sparse, mid and dense connectivity. Sparse graphs have a connectivity of 25%, mid graphs have a connectivity of 45% and dense graphs have a connectivity of 70%. The differing connectivity categories give us varying number of edges over the same number of nodes.

In our results, we noted that there were a graphs with a lower $(N * E)$ value, where N are number of Nodes and E are Number of Edges, seemed to perform well on the standard algorithm, however - we believe that may be attributable to the fact that kernels run for very short duration, which is not enough to overtake any overhead of our stream implementation. Since these types of graphs do not occur in practical use scenarios we do not have to worry about them. We ran the algorithm on various graphs starting with $(N = 32, E = 271)$ upto $(N = 128, E = 11626)$. For the latter graphs starting specifically $(N = 128, E = 4143)$ we started to see noticable improvements, of up to 40%.

Moreover, during our testing, we hypothesized that using unified memory would be better for our algorithm, since the device performs only read operations on the data, however - that lead to a significant downgrade in performance and we decided to use manually managed memory over unified memory for both algorithms.

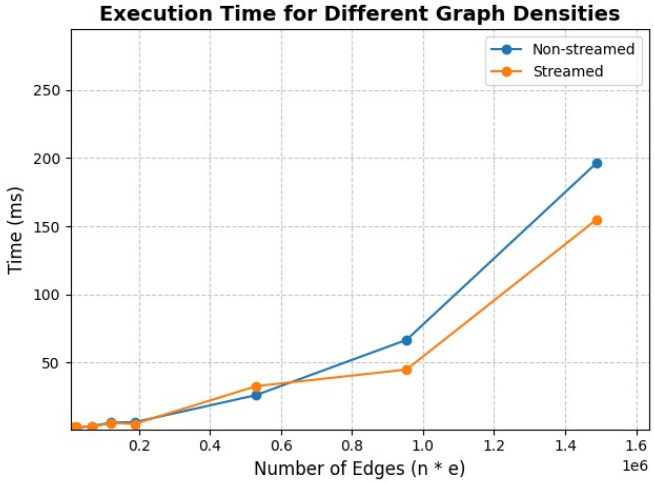

Fig. 1.

As visible in the Figure 1, the non-streamed algorithm outperforms the streamed algorithm for small graphs, however - the stream algorithm starts to overtake the non-streamed algorithm at around 0.5, and the gap between the two only widens after that.

Based on the graph we conclude that the execution time increases as the sparsity of the graph increases. The trend is visible in both streamed and non-streamed data.

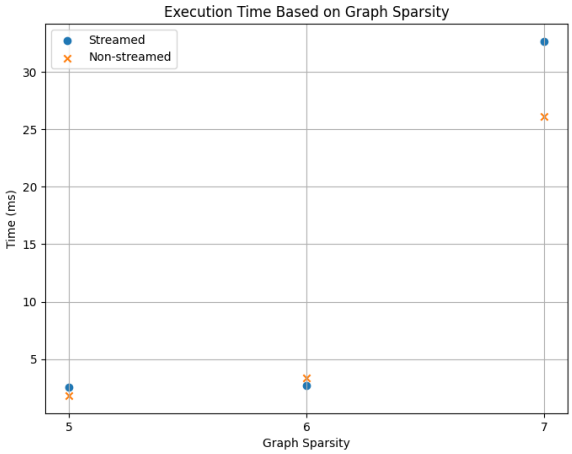

Fig. 2.

When looking at the non-streamed data, the execution time is relatively low for sparse graphs (sparsity=5) and gradually increases for denser graphs (sparsity=6 and 7). The same trend is observed in the streamed data, with the execution time for sparse graphs being lower than the denser ones.

Additionally, the streamed data has a higher execution time overall compared to the non-streamed data. This may be due to the overhead of streaming the data, as well as the additional computations required to perform streaming algorithms.

Overall, the performance of the algorithms appears to be negatively impacted by the sparsity of the graph. As the graph becomes denser, the execution time increases, which suggests that the algorithms are less efficient in handling denser graphs.

Moreover, it is also evident that graph sparsity widens the gap between the two algorithm, which is a promising because most real-world graphs exhibit high levels of sparsity, meaning that this "streamed" algorithm could exhibit even more improvement in real world scenarios.

VII. CONCLUSION

We were able to gain upto 40% improvement over the standard implementation of the FB-algorithm on CUDA, by the implementation of CUDA streams. By running the forward and backward reach kernels asynchronously we were able to overlap their execution which resulted in a good runtime improvement.

For further work we have decided to improve our implementation of streams. The algorithm performs a transpose of the graph on the host side which takes a considerable amount of time, only the backward kernel needs access to the trans-

pose so we can launch the forward kernel asynchronously while the host performs the transpose and then the backward kernel is also launched asynchronously. Moreover, we aim to improve the overall memory management of the implementation as there are memory leaks which cause the algorithm to crash on bigger graphs. We aim to improve on our existing work by later introducing the trim function, that should also prove theoretically improve performance by removing trivial SCC's. We also intend to extend the scope by testing on larger graphs, as well as including real world graphs in our testing.

VIII. ACKNOWLEDGEMENTS

We would like to thank Dr. Muhammad Mobeen Movania (Habib University) for introducing us to GPU accelerated computing, CUDA programming and for both pushing and guiding us through the duration of our work to make it the best possible version it can be.

IX. REFERENCES

[1] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceška, "Computing strongly connected components in parallel on cuda," in *2011 IEEE International Parallel and Distributed Processing Symposium*, 2011, pp. 544–555.

[2] Jon Freeman, "Parallel algorithms for depth-first search," in *Technical Reports (CIS)*, 1991.

[3] John H. Reif, "Depth-first search is inherently sequential," in *Information Processing Letters*, 1985, vol. 20, pp. 229–234.

[4] Sungpack Hong, Nicole Rodia, and Kunle Olukotun, "On fast parallel detection of strongly connected components (scc) in small-world graphs," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 11 2013.

[5] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri, "Bfs and coloring-based parallel algorithms for strongly connected components and related problems," in *Proceedings - IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS 2014*, United States, 2014, Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS, pp. 550–559, IEEE Computer Society, 28th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2014 ; Conference date: 19-05-2014 Through 23-05-2014.

[6] Pingfan Li, Xuhao Chen, Jie Shen, Jianbin Fang, Tao Tang, and Canqun Yang, "High performance detection of strongly connected components in sparse graphs on gpus," in *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*, 2017, pp. 48–57.