An Implementation of Connected Component Algorithm on GPU-Project Report

Leyuan Wang

March 14, 2013

Motivation

The connected component algorithm is widely used in many fields. In graph theory, a connected component of an undirected graph is a subgraph in which any two vertices are connected to each other by paths. Graph connectivity is a fundamental graph problem present in many scientific and commercial applications such as image processing, pattern recognition, social network analysis, etc. Efficiently implementing the connected component labeling algorithm is desired for graph processing. Graphics processing units provide a large computational power for the intensive scientific applications but they are more suited for regular data access patterns. Finding the connected components of a graph has an irregular data access pattern which is not directly amenable on GPUs. Thus finding ways to optimize the implementation of the algorithm on GPU is an interesting and important topic.
There are many ways for CPU to do connected component labeling like the Union Find algorithm and many PRAM algorithms such as the Shiloach-Vishkin algorithm, but they are not very efficient to implement on GPU for various of reasons. Our goal is to find an efficient way for GPU to do the connected component labeling.

Approach

The major goal of the algorithm is to find the root of each component and make each node in the same component directly points to the root to form a rooted star and change the label of each node to the root's label. Based on Soman's paper[1], we divide the process into the following steps: (a figure illustrating this process is on next page)

- Initializing the parent of each node by letting each node point to itself;

- Alternating orientation hooking process by reading the edges and assigning a node with the smaller label as the parent in odd iterations and with the larger in even iterations and at the start of the next iteration check if parents of the two nodes are the same, if so, mark the edge after which the edge will be inactive and won't participate in the next iteration;

- After all the edges are marked, check if there are any stagnant trees which are two

trees whose roots are the components of an edge, and if there are, send them to the hooking kernel again until there's no stagnant tree;

- Multilevel pointer jumping;

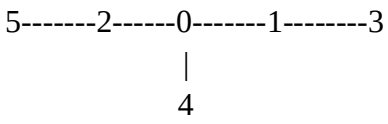
Implementation

Input

Our input format is an edge list which can be seen as an arbitrary ordered array of edges, is a more practical and natural choice for this algorithm compared with adjacent matrix. We build a structure for the edge list and assign coo[edge_num].row and coo[edge_num].col as the vertices of an edge with edge number edge_num.

0,1	1,3	0,2	2,4	2,5

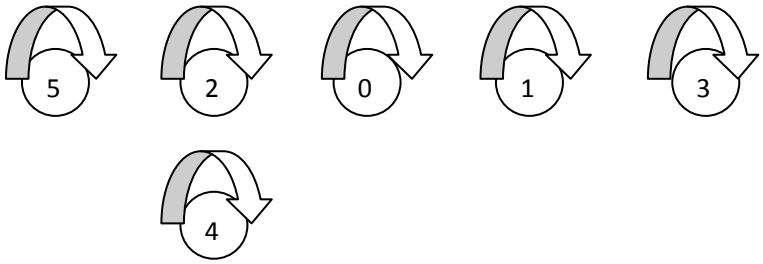
(a) Edge List



(b) Corresponding Graph

First Step

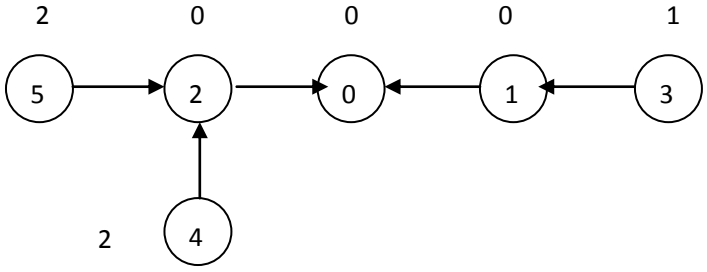
Initializing



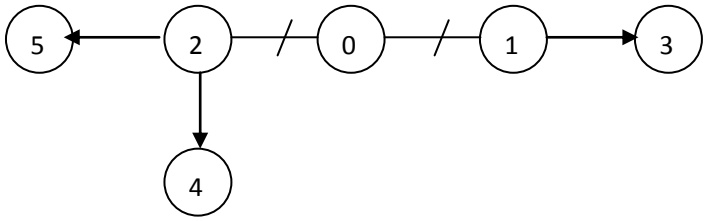
(c) Each node points to itself

Second Step

Hooking



(d)Nodes with larger index points to smaller ones in 1st iteration (parent's label beside)



(e) Two edges with the same parent after 1st iteration are marked, nodes with smaller index points to larger ones in 2nd iteration

At this point, one mistake was found in the original algorithm in Soman's paper.
The algorithm in the paper is as follows:

---

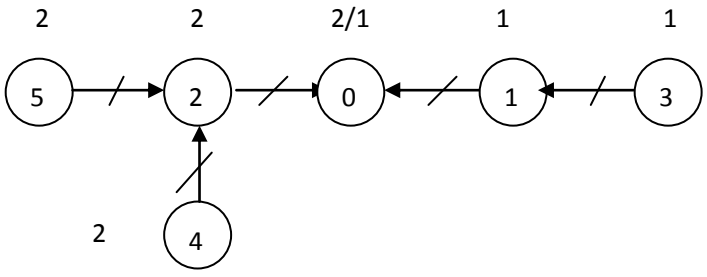
1 Begin
2 Initialize Parent[i] = i;
3 while All edges are not marked do
4 for each edge (u, v)
5 if edge is unmarked and Parent[u] != Parent[v]
6 max=max{Parent[u], Parent[v]};
7 min=min{Parent[u], Parent[v]};
8 if iteration is even Parent[min] = max;
9 else Parent[max] = min;
10else Mark edge (u, v);
11 end-for
12end-while
13Multilevel pointer jumping();
14end.

Now we are in 2nd iteration and from the 1sr iteration we already have P[5]=2, P[4]=2, P[2]=0, P[0]=0, P[1]=0, P[3]=1. And we have max=max{Parent[5], Parent[2]}=2 and min=0 for edge(5,2). Now we go on to line 8 which assign P[min]=max=>P[0]=2 but it doesn't change the parents of either 2 or 5 so in the following iterations, edge (5,2) will never be marked. So are edge (4,2) and edge(1,3). So we modify line 8 and 9 to the following lines to let both the parent and the children point to the max or min.

---

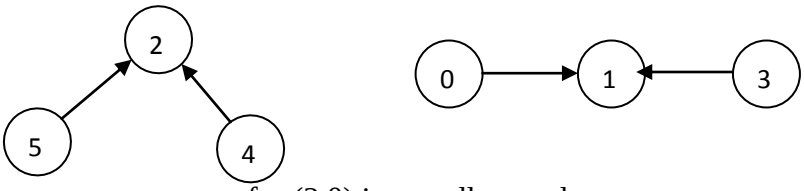
if (max==Parent [u]) { mx=u; mn=v;}
if iteration is even { Parent[min] = max; Parent [mn]=max;}
else { Parent[max] = min; Parent [mx]=min;}

And the result of the example is



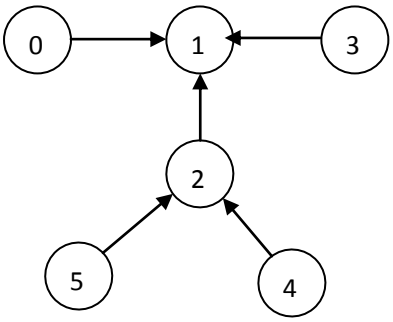
But now there's a competition in writing the parent of 0. For the paper mentions that we don't need any atomic operation, and no matter which write wins the result will still be the same(may produce a bug in running large complex dataset).

Suppose '1' wins and Parent[0] is assigned to 1, the graph is now like this

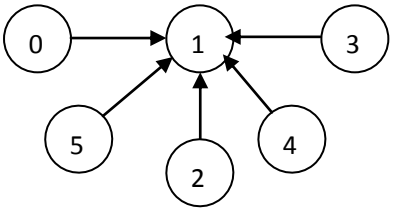


which are two stagnant trees, for (2,0) is actually an edge, we move on to next step.

Third Step
Send them back to the hooking kernel and the result is as follows



Fourth Step
Multilevel Pointer Jumping



Final Rooted Star

Results and Analysis

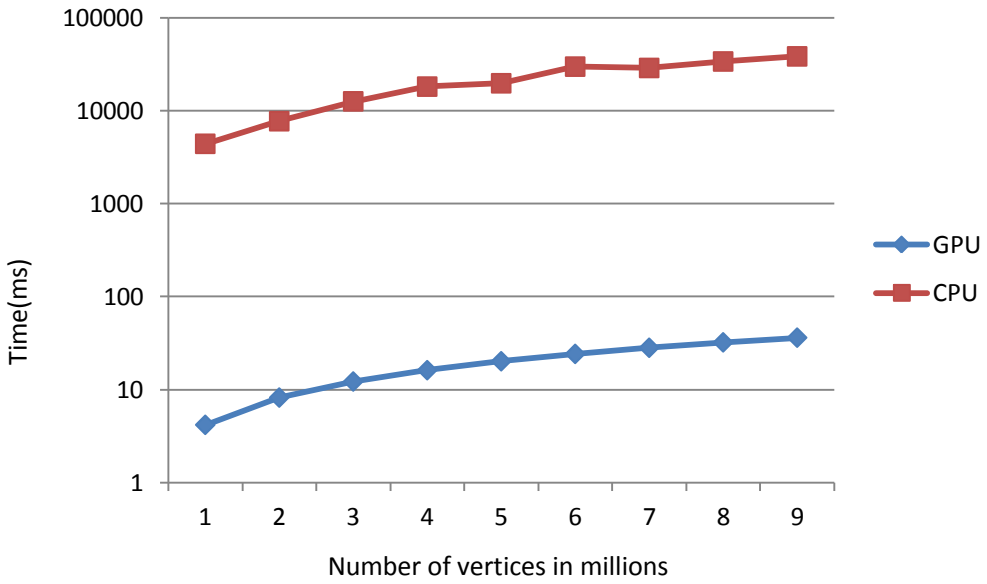
Environment
I run the test on a PC with Intel Core i3-2330M @ 2.2GHz CPU, NVIDIA GEFORCE 540M graphics card running CUDA Toolkit /SDK version 5.0
Dataset
Two sources: self-produced large data with defined number of nodes, edges and connected components used for drawing the performance graph; Real world charity net graph from charity website with different sizes (2K and 4K).
Charity dataset 2K: nodes 1953 edges 1999

Charity dataset 4K: nodes 3831 edges 3999
Using the same dataset to compare with Boost connected components algorithm on CPU.

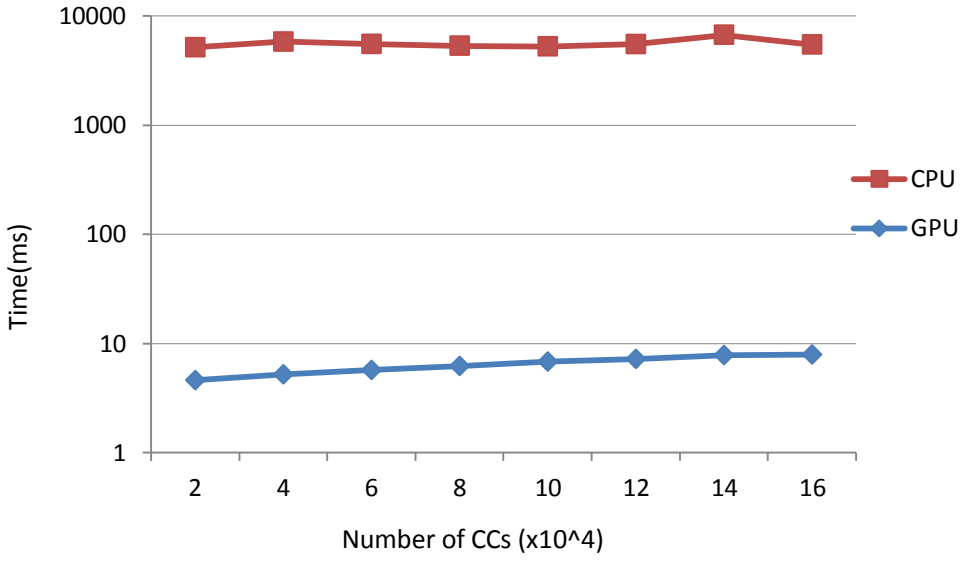
	GPU	CPU	Speedup
2K	0.336ms	8ms	23.8
4K	0.407ms	20ms	49.14

The speedup is about 20x for the 2K dataset and 50 for the 4K dataset.

Then I generated the data with 5000 connected components and number of edges and nodes from 1 million to 9 million and plot the diagram below to compare the performance of the algorithm and the Boost CPU version.



Then I change the x axis to the number of connected components and keep the number of nodes and edges to 1 million, the plot is as below



Conclusion

This project shows the great advantages of implementing Connected Component Algorithm on GPU. But there are still much future work to do. One is the multiple data writes in the hooking kernel that would produce segment fault when running large real-world data. Second is the checkstagnant() part, which in Soman's paper, he mentions that there shouldn't not be any atomic operations. And the primitives can still be redesigned to gain better performance.

References

[1] Jyothish Soman, Kothapalli Kitapalli, and P J Narayanan, A Fast GPU Algorithm for Graph Connectivity.
[2] SHILOACH, Y., AND VISHKIN, U. An O(log n) parallel connectivity algorithm. Journal of Algorithms 3, 1 (1982), 57–67.
[3] Jiˇr´ı Barnat, Petr Bauch, Luboˇs Brim, and Milan Ceˇska, Computing Strongly Connected Components in Parallel on CUDA.