Implementing connected components algorithms in CUDA to enhance automatic tracing algorithms

Ethan Trepka

Overview

Mapping all neural connections in the brain is a daunting task, but one that an increasing number of labs are investigating. Developing these maps requires efficient, sophisticated objection detection and classification algorithms that accurately identify neurons and synapses. However, ensuring the efficacy of the results produced by these algorithms is a challenge for the field.

Maps of neural connections can be represented as graphs and as such, can be examined using graph algorithms. Graph traversals such as BFS and DFS can be used to detect connected components in the neural graph and aid error detection. Small connected components suggest image recognition algorithms either missed connections between these small components and larger connected components or misidentified certain cells as neurons.

To approach this problem, an efficient connected components algorithm was implemented in CUDA to count the number of connected components in a neural graph and give a general metric for evaluating the effectiveness of neuron and synapse detection algorithms.

Data

Two datasets were used: MICRONS small and MICRONS large. MICRONS small consists of 334 neurons in L2/L3 and 3,408 edges between them. MICRONS large consists of 1,272,001 neurons in L2/L3 and 6,121,336 edges between them.

Both MICRONS datasets were created from the same image volume by running automatic neuron and synapse detection algorithms. The MICRONS small dataset is a subset of the neurons in the MICRONS large dataset that has been manually checked, while the MICRONS large dataset has not been manually verified.

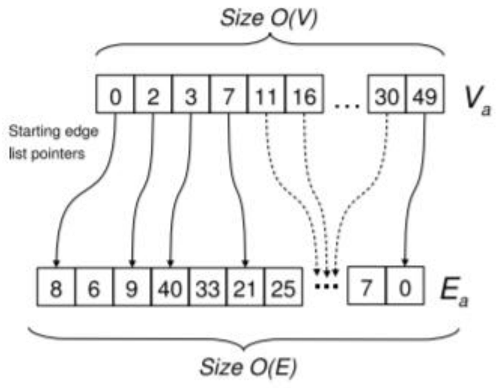
Algorithm and Pseudocode

Preprocessing

Preprocessing of the data was done in python and the code was included in my submission. The data had uint64 neuron id numbers and was a csv with a list of presynaptic, postsynaptic id pairs. In python uint64 ids were converted into integers, and input into a dictionary. Then, each key value pair in the dictionary was written to a text file which is a command line argument for the CUDA program.

Data representation

The following data representation of a graph was used. This figure is taken from a paper by P. Harish and P. J. Narayanan. The vertices array stores the indices of the beginnings of the edge array



Pseudocode

The two algorithms below demonstrate my implementation of the connected components algorithm. In the algorithm, there is one thread per block and each block represents one vertex. The algorithm was also based on a similar breadth first search implementation from the paper by P. Harish and P. J. Narayanan.

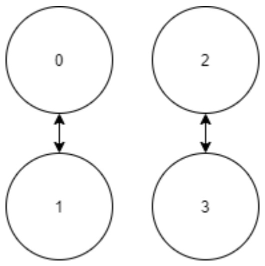
Algorithm 1: GPU kernel algorithm

```
tid = blockIdx.x;
if frontier[tid] then
    frontier-a[tid] = false;
    visited[tid] = true;
    index = vertices[tid];
    while index < vertices[tid+1] do
        vertex = edges[index];
        if ¬visited[vertex] then
            frontier-b[vertex] = true;
        end
        index += 1;
    end
end
```

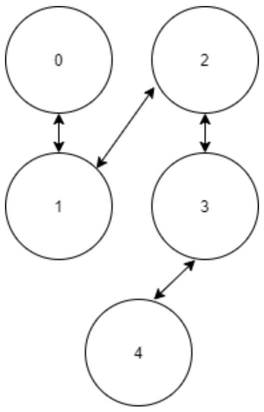
```
Algorithm 2: CPU side algorithm
connected-components = 0;
while there is a vertex, v, left to visit do
    set frontier-host[v] = true;
    copy frontier-host to frontier-a;
    connected-components += 1;
    count = 0;
    while there is a vertex with value true in frontier do
        if count mod 2 = 0 then
            cc-kernel(num-vertices,1)(vertices, edges, frontier-a, frontier-b, visited-dev);
            copy frontier-b to frontier-host;
        else
            cc-kernel(num-vertices,1)(vertices, edges, frontier-b, frontier-a, visited);
            copy frontier-a to frontier-host;
        end
        count += 1;
    end
    copy visited-dev to visited-host;
end
```

Testcases for Correctness

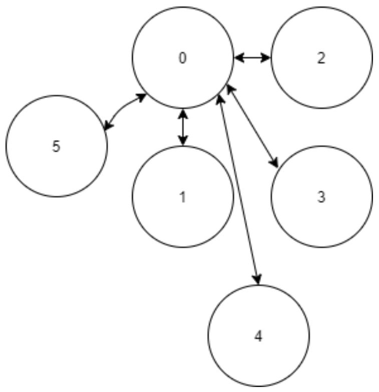
The GPU algorithm was run on the following three test cases, and results were verified. Additionally, the results of the GPU algorithm on the small MICRONS data were compared to the results of a CPU algorithm on the small MICRONS data; both algorithms outputted the same number of connected components.



Test 1, Output: 2 connected components



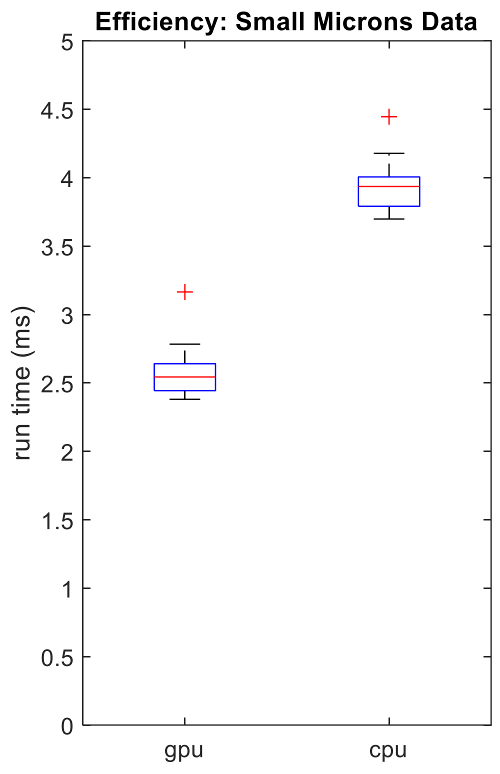
Test2, Output: 1 connected component



Output: 1 connected component

Statistical Efficiency

Efficiency comparisons were done on the small microns dataset because the large microns dataset was too large to facilitate repeated runs. The figure below demonstrates that the GPU connected components implementation is nearly twice as fast as the CPU implementation.



Methods

CUDA basic kernel implementations

The pseudocode in the algorithms section describes my final cuda basic kernel implementation that I used. In a highly connected graph, the algorithm is compute-bound but in a sparsely connected graph the algorithm is memory-bound because it has to perform many large scale memory copies.

To reduce memory copies, I attempted to write additional kernel functions to perform the CPU side computations. These computations require checking if an element in the frontier array is true and finding the next unvisited element. However, this approach failed because those functions will inevitably have read/write conflicts.

The edge array data organization system was used to take advantage of coalesced access.

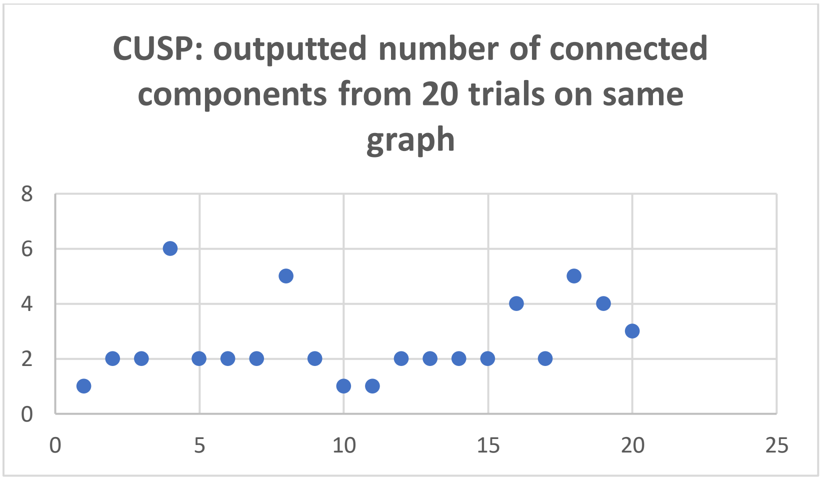
CUSP implementations

CUSP is a CUDA library built on top of thrust. It contains implementations of many different types of linear solvers, graph algorithms, and representations of sparse matrices.

I ran the CUSP connected_components function on the small microns dataset 20 times. The correct number of connected components in this graph was one. However, the CUSP implementation produced wildly variable results. The CUSP algorithm managed to generally approximate the correct number of connected components, but it is not a correct algorithm. The figure below shows the output of the cusp connected components function, run on the same graph, 20 different times.

I also tried the CUSP BFS function but got similarly erratic results.

After getting this result, I found that similar errors had been reported on CUSP's git page but have evidently not been fixed. The only change that they added in response was a TODO comment that the graph algorithm would not work for architecture sm>=30.



Output and Significance

MICRONS small has 1 connected component, with a mean algorithm runtime of 2.77 ms based on a sample of 20 iterations.

MICRONS large has 204,896 connected components, with a runtime of 45 minutes (2,715,965 ms).

The manually verified subsection of the image volume had one connected component, an average of 340 neurons per connected component, while the automatically generated section had over 200,000, with an average of 6 neurons per connected component. These results illustrate the necessity of improving automated tracing algorithms prior to moving forward with large scale connectome projects. This suggests that the automatic tracing algorithm either classifies objects as neurons that are not neurons or fails to detect many of the synapses present in the image volume.

In terms of algorithmic efficiency, the GPU algorithm outpaced the CPU algorithm by a factor of two. This gain of efficiency becomes more important as the size of the dataset grows from the one million neurons tested here to the 86 billion neurons present in the human brain. However, more efficient implementations are required since running this current GPU algorithm on the 86 billion neurons in the human brain would take 6 years.

Graph-based algorithms like this connected components algorithm could help identify regions of the image volume that require additional analysis with more accurate but time intensive image tracing algorithms.

References

1. “Layer 2/3,” *MICrONS Explorer*. https://microns-explorer.org/phase1.
2. P. Harish and P. J. Narayanan, *Accelerating large graph algorithms on the GPU using CUDA*.