Parallel Implementation of Boruvka on CUDA

Project Report
Prankur Gupta
108492684
{prgupta}@cs.stonybrook.edu

May 15,2012

Abstract

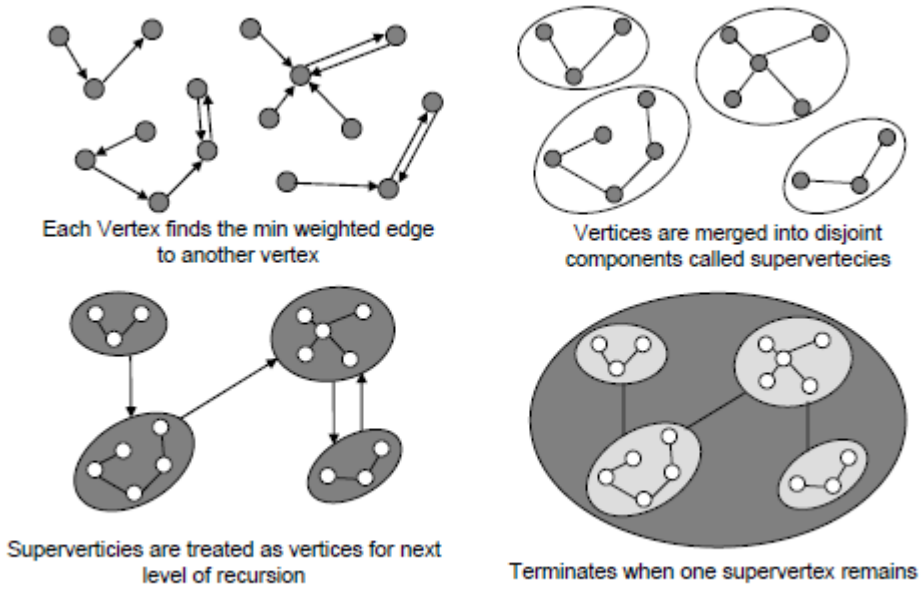
Boruvka's algorithm is an algorithm for finding a minimum spanning tree in a graph for which all edge weights are distinct. The algorithm begins by first examining each vertex and adding the cheapest edge from that vertex to another in the graph, without regard to already added edges, and continues joining these groupings in a like manner until a tree spanning all vertices is completed. So, the best thing about this algorithm is that it is inherently parallel in nature. And we all know the importance of calculating Minimum Spanning Trees on large and dense graphs, as they are really very useful for some applications, e.g. road networks. Graphics Processor Units are used for many general purpose processing due to high compute power available on them. Regular, data-parallel algorithms map well to the SIMD architecture of current GPU. GPU provides various efficient data-mapping primitives, which if used could provide us good performance boost. In this project, we used these primitives for our algorithm.

1 Introduction

In this report I'll discuss about my implementation for Boruvka's Algorithm for calculating Minimum Spanning Tree on large graphs. Given a Graph G(V,W,E) find a tree whose collective weight is minimal and all vertices in the graph are covered by it. The fastest serial solution takes $O(E\alpha(E, V))$ time, given by Chazelle, where α is the functional inverse of Ackermanns function. Boruvkas approach to the MST problem takes O(ElogV) time.

I've implemented Boruvka on Nvidia GPUs under CUDA, as a recursive formulation of Boruvkas approach for undirected graphs. I have used some data primitives from CUDPP libraray as well as Thrust library, designed for gaining good performance. Todays GPUs are also massively mulithreaded with thousands of threads in flight simultaneously. Reduction of irregular steps into a combination of such primitives is critical to gain performance on the GPUs. I have used data primitives such as scan, segmented scan and sort for my algorithm.

Minimum Spanning Tree is very useful in Network Design, Route Finding, Approximate solution to Traveling Salesman problem. So, we want it to be really fast and efficient. Using an efficient parallel implementation of MST would really help in various applications, e.g road networks.



Cycle making edges are explicitly removed from the MST in each iteration.

In my code, I am running the loop until $num\_vertices < 1$, and in every iteration my total vertices tend to decrease by half with high probability. Next iteration vertices are my super vertices, which in the worst case are $\frac{num\_vertices}{2}$. And in any iteration, I'm covering the whole length of vertices and edges in a kernel function using multiple threads. So, considering it an O(c) cost(which in fact in depends on number of processors and the threads used). The complexity of my algorithm is $O(c \log V)$.

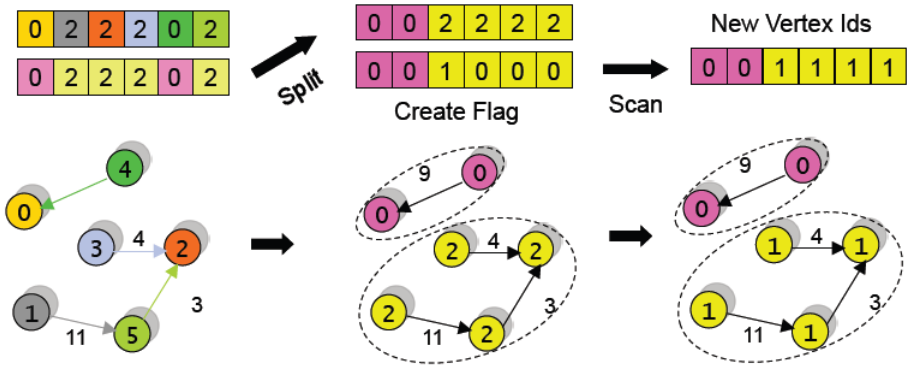
2 Graph Repesentation and Primitives

I'm representing my graph as an adjacency list. The graph used in undirected graph. I generate an random undirected graph and converts it to my desired format.

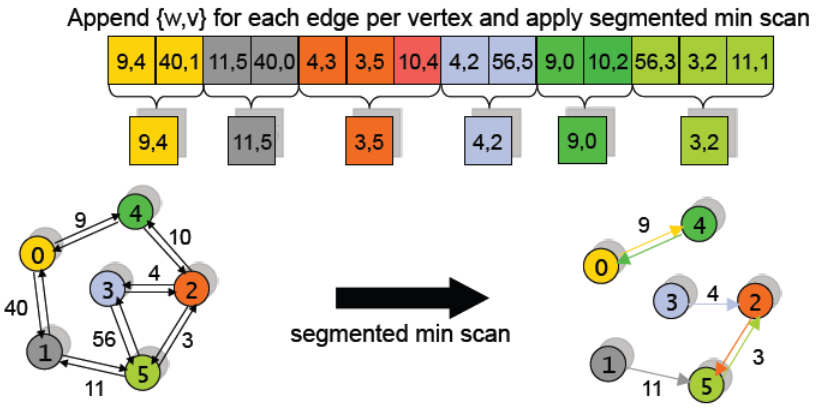
The data primitives used in my code are SCAN, SEGMENTED_SCAN from CUDPP Library and SORT from THRUST library. SCAN is the prefix sum variant in CUDA, and SEGMENTED_SCAN is also prefix sum but based on a flag array. Primitives are efficient

1. Non-expert programmer needs to know hardware details to code efficiently
2. Shared Memory usage, optimizations at grid.
3. Memory Coalescing, bank conflicts, load balancing
4. Primitives can port irregular steps of an algorithm to data-parallel steps transparently

Scan (CUDPP implementation): Used to allot ids to supervertices after merging of vertices into a supervertex



Segmented Scan (CUDPP implementation): Used to find the minimum outgoing edge to minimum outgoing vertex for each vertex.



Sort(THRUST Implementation): To sort the supervertex Ids and their edges, which are to be included in next iteration.

My code will work for a maximum weight of 2^{10}, and an graph with $2^{22} - 1$ vertices. But they are configurable as per your need i.e. if your graph has some fixed range of weights, we can increase the range of total vertices possible.

3 Algorithm

For the project I have modified the algorithm presented in the paper, for my implementation. I followed the following algorithm

1. Append weight w and outgoing vertex v per edge into a list, X.
2. Divide the edge-list, E, into segments with 1 indicating the start of each segment, and 0 otherwise, store this in flag array F.
3. Perform segmented min scan on X with F indicating segments to find minimum outgoing edge-index per vertex, store in WE.
4. Find the successor of each vertex and add to successor array, S.
5. Remove cycle making edges from WE using S, and identify representatives vertices.
6. Remove edges for those whose successor are themselves
7. Mark the minimum of the rest of the edges using the segmented min scan output and successor array

8. Do pointer doubling to sync all the vertices in a super group together.
9. Append the original super root id to the vertex id for each vertex, and sort them to bring them together
10. Make another flag array on basis of the root super id and then prefix sum on the sorted list, P
11. Use P, to get a new supervertex id's for each vertex.
12. Replace the destination vertices of the edges between vertices having same super group Id with INF.
13. Append the super vertex id with each edge id. If destination vertex is INF, then supervertex Id is also INF.
14. Sort this list, to get eligible vertices for next iteration. This brings all the edges of the same supervertex Ids together.
15. Again Create a flag array to distinguish edges of different supervertex ids (Note - They contains duplicate edges)
16. The previous array gives us our new number of edges. Check if new_edges == num_edges, if true return.
17. Compact and create the new edge-list and weight list as per our input graph format.
18. An array stores the edge_mapping information from old edges to new edges.
19. Build the vertex list from the newly formed edge-list.
20. Call the MST Algorithm on the newly created graph until a single vertex remains.

4 Code and Results

Before running please do **module load cuda** and **module load cuda_SDK**.

4.1 Codes

1. connected-1.cpp - Code to generate a random graph for a given number of vertices and edges.
2. convert.pl - A perl script to convert the random graph into my desired format
3. prob.cu - The CUDA implementation of the Boruvka Algorithm for Minimum Spanning Tree
4. functions.cu - Contains the kernel functions called by the prog.cu
5. Makefile
6. prog.cpp - Serial MST code (Boruvka Algorithm)

4.2 Results

Uptil now, I have tested my code for many test cases out of which I'm writing the two important ones.

4.2.1 Test Case 1

For V = 10000, E = 60000
a) Serial Version = 8 sec
b) Parallel Version = 3 sec

4.2.2 Test Case 2

For V = 100000, E = 800000
a) Serial Version = 26 min 04 sec
b) Parallel Version = 5 sec

I would include more results of various test cases as the time progresses.

5 Related Work

My algorithm first used my implementation of prefix sum, segmented scan and sort. But I found that it wasn't performing too much. So I switched to CUDPP library. Instead of using a split primitive, I just used sorting, as discussed during presentation, I pointed out, that basically SPLIT is doing sorting only. This helped me in learning how to use CUDPP library, which

is quite good. Being aloe, I faced a lot of problems while implementing, so it took me a lot of time too.

6 Conclusion And Future Work

I have succesfully implemented the Boruvka algorithm on CUDA. But still it can be further improved. I haven't looked into the *SPLIT* primitive which they have used. I could look into that also. I have used sort of Thrust, which can be replaced with the sort of CUDDP, and we can check the performance of my algorithm after using them. Moreover, in future there might be some more efficient primitives which can be used for further improvement. Irregular steps of the algorithm can be mapped to other data-parallel primitives efficiently.

In my implementation I have taken input in my desired format, which could be changed to a more generic format. And moreover, like in serial version we say that, MST calculation is most effective if we run Boruvka for some iterations, as it reduces the graph size to atleast half after every iteration, then run Prim's algorithm for rest. We can achieve much better performance. I would certainly like to do that.

7 References

1. Fast Minimum Spanning Tree for Large Graphs on the GPU - V Vineet, P Harish, S Patidar, P. J. Narayanan
2. `http://cudpp.googlecode.com/svn/tags/2.0/doc/html/example_simple_c_u_d_p_p.html`
3. `http://gpgpu.org/static/developer/cudpp/rel/cudpp_1.1/html/group__public_interface.html#g0d39e7c3e14963c7cc3df3b879362c25`
4. `http://research.nvidia.com/news/thrust-cuda-library`
5. `http://code.google.com/p/thrust/wiki/QuickStartGuide`
6. `http://www.highperformancegraphics.org/previous/www_2009/presentations/vineet-fast.pdf`
7. `http://www.tacc.utexas.edu/user-services/user-guides/lonestar-user-guide`
8. In lonestar, **$TACC_CUDA_DIR/doc/nvcc.pdf**

9. In lonestar, **$TACC_CUDA_DIR/doc/CUDA_C_Programming_Guide.pdf**