CSE-590: CPU-GPU Hybrid approach for Boruvka's MST Algorithm

DHANENDRA JAIN MUHAMMAD ALI EJAZ NIRMIT DESAI

Stony Brook University - Group 18
{110559131, 110369635, 110351469}
dhjain, mejaz, nndesai @cs.stonybrook.edu

Abstract

For large graphs, the serial computational cost for finding minimum spanning tree is very high. However, with the advent of GPUs, parallelism in algorithms can be exploited to reduce the running time of well known algorithms like Prim's, Kruskal and Boruvka. A significant advantage of the Boruvka's algorithm is that it can be easily parallelized since the choice of the cheapest outgoing edge for each component is completely independent of the choices made by other components. Our aim is to implement Boruvka's algorithm on GPU while offloading some parts of computation to CPU when GPU is executing the kernel. We also test our implementation on two different versions of GPUs (Tesla K20 and Nvidia K520).

1 Introduction

Modern Graphics Processing Units (GPUs) are used for many general purpose processing due to their high computational power at low costs. The latest GPUs, for instance, can deliver close to 1 TFLOPs of compute power at a cost of around $400 [Harish et al. 2009]. The GPUs, with a common kernel, process a number of data instances with thousands of threads in flight simultaneously.

Minimum spanning tree (MST) computation on a general graph is an irregular algorithm. A minimum spanning tree is a spanning tree of a connected, undirected graph. It connects all the vertices together with the minimal total weighting for its edges. MST has wide application in network design, route finding, in finding approximate solution to Traveling Salesman problem, etc. The best sequential time complexity for MST is O(Eα(E, V)), given by Chazelle, where α is the functional inverse of Ackermann's function [Chazelle 2000]. The solution given by Boruvka with O(E logV) time [Boruvka 1926] and later discovered by Sollins[1] is generally used in parallel implementations. A GPU implementation using CUDA with irregular kernel calls [Harish et al. 2009] and exploiting parallel data mapping primitives available on the GPU [Vineet et al. 2009] are reported. We on the other hand, implement Boruvka's algorithm on GPU while offloading some parts of computation to CPU when GPU is executing the kernel. We vary the offloading to CPU with different percentages of input while the rest of the work is done on GPU. We also test our implementation on two different versions of GPUs: Tesla K20 (Stampede) and Nvidia K520 (Amazon AWS g2.8x large instance).

Boruvka's approach, which works only on undirected graphs, finds the minimum weighted outgoing edge at each vertex and merges disjoint connected vertices into supervertices, which are treated as vertices for next level of iteration (Figure 1). Merging is an irregular operation as multiple vertices are assigned to a single supervertex.

[1] https://commons.wikimedia.org/wiki/File:Boruvka%27s_algorithm_(Sollin%27s_algorithm)_Anim.gif

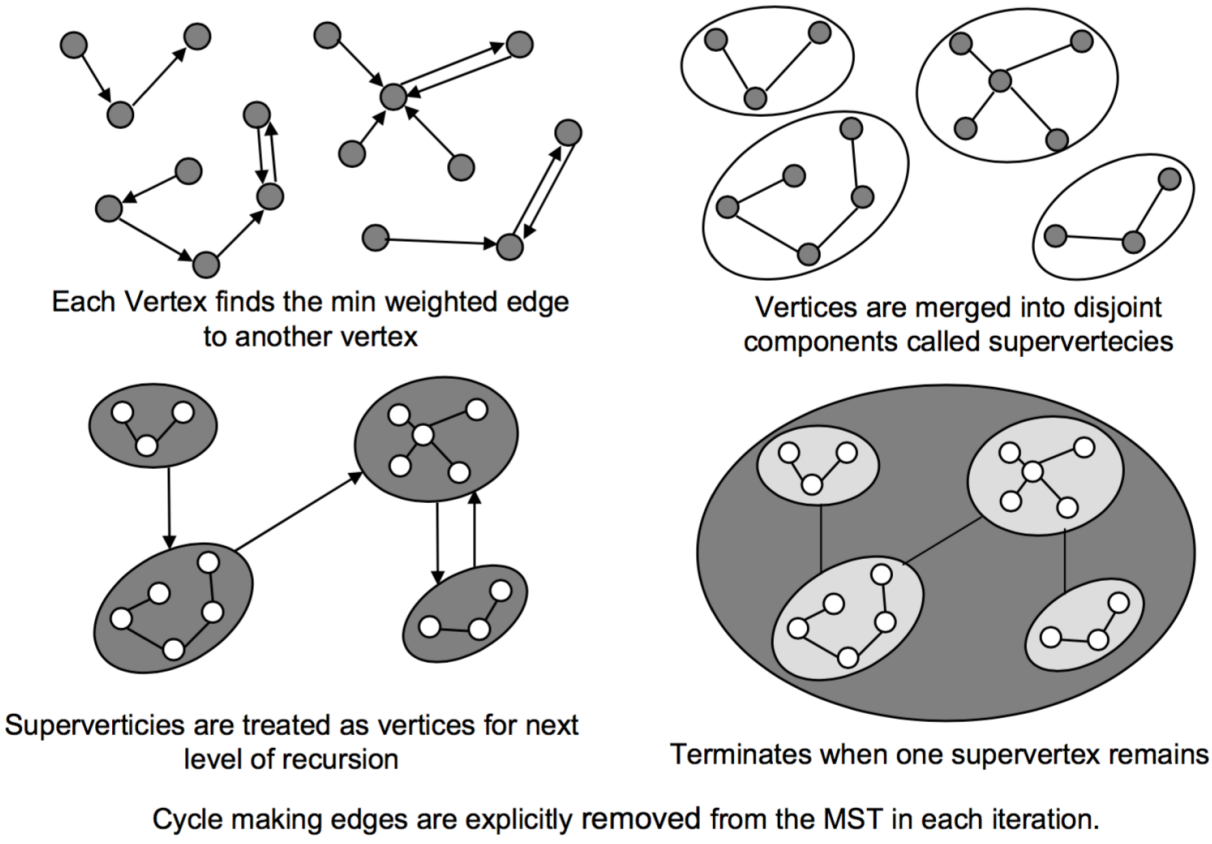

Figure 1: Steps of Boruvka's approach to MST

The approach is to have a recursive framework implemented using a series of steps including some basic primitives. At the beginning of the procedure each vertex is considered as a separate component. In each iteration, we reconstruct the supervertex graph which is generated by connecting each component with some other using cheapest outgoing edge of the given component and given as an input to the next level of recursion. We use segmented scan [Sengupta et al. 2007] to find the minimum weighted outgoing edge from each vertex which is then merged into supervertices using radixsort and scan primitives. A simple kernel then eliminates the cycles. In each iteration of this recursive formulation for the undirected graph, duplicate edges are removed and new supervertex numbers are assigned. This process exits when only one supervertex remains.

Hybrid Approach - As part of this project, we have implemented a hybrid approach to solving boruvka's algorithm where we offload some computation to CPU while GPU executes the kernel. For this, we've come up with 2 programs

1. CPU code executes sequentially while the kernel is executing asynchronously
2. Using openMP, CPU code executes in parallel while the kernel is executing asynchronously

In naive implementation, there are many functions and most of them work on multiple arrays of size mainly $|E|$ or $|V|$, which can be in millions in our case. These arrays were allocated memory dynamically on the device using cudaMalloc. If we modify such functions to do part of computation on CPU:

1. We would need to copy these arrays from device to host using cudaMemCpy
2. Perform the required computation
3. Copy the array back to device using cudaMemCpy

But this approach has 2 drawbacks which would have resulted in increase in running time:

1. Multiple arrays of approximately million elements would have to be allocated on CPU which may/may not always work.

2. CudaMemCpy is synchronous blocking call. So having multiple such calls would slow down the program.

Hence, our strategy was to do hybrid implementation on those functions which deal with only a single array and perform minimum computation so that we can derive benefit from using CPU while GPU is executing. So, we made hybrid implementation for some functions. These functions can deal with both – edges and the vertices array.

For each graph, we offload 5%, 10%, 20%, 30% and 40% of the input to these functions (either number of edges or number of vertices) to CPU while rest of the computation is done on GPU. The results are explained in section 4. The experiment is performed for both the programs mentioned above – with openMP and without openMP. Also, the offloading works only if number of vertices is more than 20 and number of edges is more than 200. Otherwise, naive implementation is used.

2 Implementation

Boruvka's approach is a greedy solution and comprises of two basic steps:

- **Step1:** Each vertex finds the minimum outgoing edge to another vertex. This step can be seen as:
  - Running a loop over edges and finding the min; writing to a common location using atomics. This is an O(V) operation.
  - Segmented min scan over $|E|$ elements.
- **Step2:** Merger of vertices into supervertex. This can be implemented as:
  - Writing to a common location using atomics, O(V) operation.
  - Splitting on $|V|$ elements with supervetex id as the key

We have used CUDPP library implementation [Sengupta et al. 2007] for scan and segmented scan primitives in our implementation as they are efficient and can port irregular steps of an algorithm to data-parallel steps transparently. The minimum finding step of Boruvka can be ported to the scan primitive and merger can be implemented as a radixsort on supervertex ids. The CUDPP scan implementation is used to allot ids to supervertices after merging of vertices into a supervertex while the CUDPP segmented scan implementation is used to find the minimum outgoing edge to minimum outgoing vertex for each vertex. We used CUDPP radixsort primitive instead of split primitive to bring together vertices belonging to same supervertex.

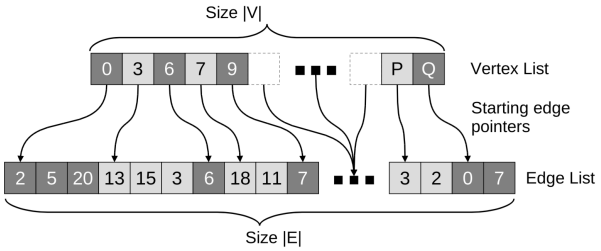

Figure 2: The Graph Representation

We represent the graph in the standard compress adjacency list format. We pack edges of all vertices into an array E with each vertex pointing to the starting index of its edge-list, Figure 2. We create this representation in each recursive iteration for the supervertex graph. Weights are stored in array W with an entry for every edge.

As discussed before, the algorithm starts by finding the minimum weighted edge to the minimum outgoing vertex. Cycle making edges are removed and the remaining edges are added to the output. Vertices of each disjoint component is combined into a supervertex, which acts as a vertex for the next iteration. The process terminates when exactly one supervertex remains. Next we discuss the detailed steps of our algorithm.

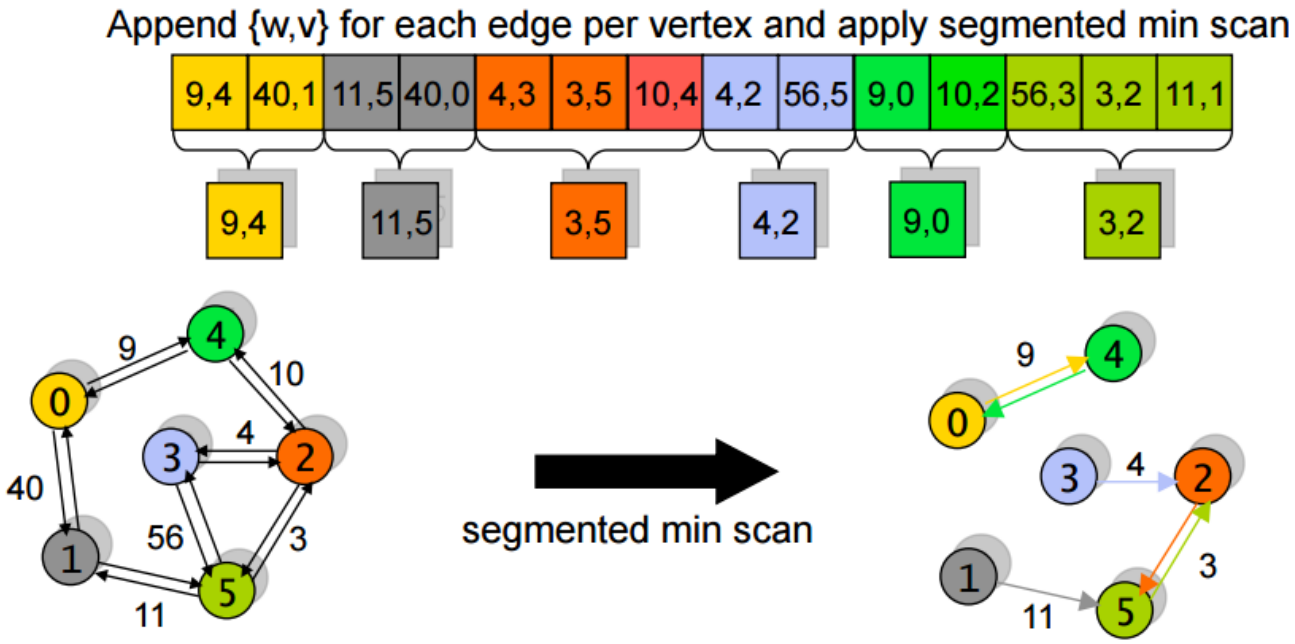

Figure 3: Segmented Min Scan

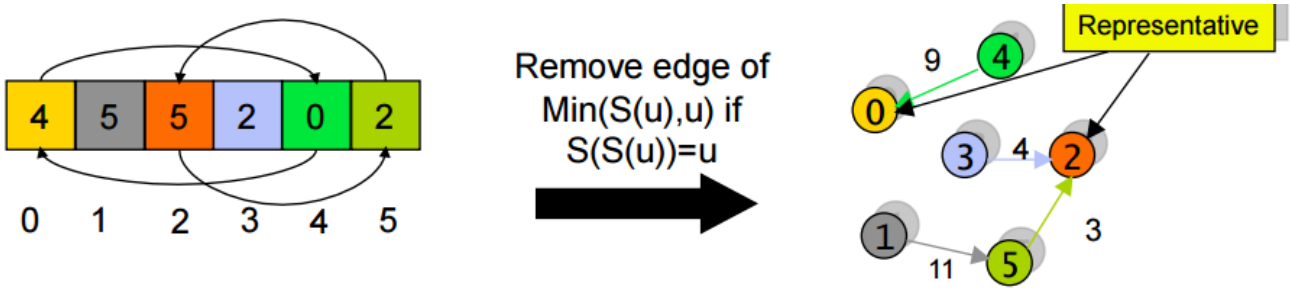

Figure 4: Remove Duplicate Edges

2.1 Find min-weight Outgoing Edge

Each vertex u finds the minimum wighted outgoing edge to another vertex v using a segmented min scan. We append the weight to the vertex ids v, with weights placed on the left. Weights are assumed to be small, needing 10 bits, leaving 22 bits for v. We create another flag array to determine start of each outgoing edge from vertex u. The segmented min scan on X returns the minimum weighted edge and the minimum outgoing vertex v for every vertex u.

As shown in Figure 3, vertex 0 has 2 outgoing edges to vertex 4 and 1 with eights 9 and 40 respectively. So they are the first 2 entries in the array. Similarly, we've weight-edge pair for each vertex. After segmented min scan, we can see each vertex has outgoing edge to minimum outgoing vertex.

2.2 Remove Cycles

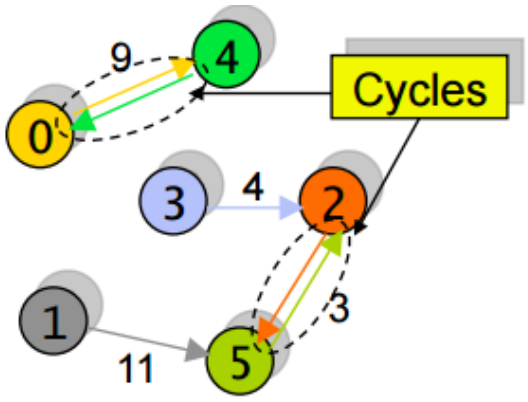

Figure 5: Cycles in Graph

Next we find and remove cycles by traversing successors for every vertex. A successor array S is created with each vertex's outgoing vertex id. This array is then traversed and if we find S(S(u))=u, which

denotes that u makes a cycle as shown in Figure 5, an edge is removed. We remove the smaller id edge, either u or S(u), from the current edge set and mark the remaining edges as part of output MST (Figure 4).

2.3 Representative Vertex

A vertex from each disjoint component is then selected to represent that component. This vertex is selected at the time of removing cycles and won't have any outgoing edges. This vertex will point to itself in the successor array since it has no successor (Figure 6).

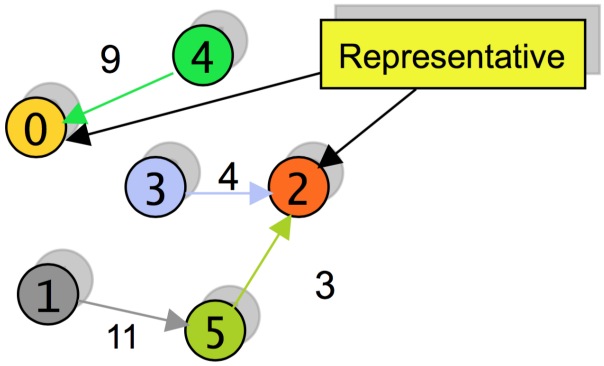

Figure 6: Selecting Representative Vertices

2.4 Mark MST Edges

Next, we mark the remaining edges in the output as part of MST. This set of edges which will now be part of MST will have minimum weight.

2.5 Vertex Propagation

Vertices combine to form a supervertex. These vertices whose successors are set to themselves are representatives for each supervertex. Other vertices in each of these disjoint components point to there representative vertex. We employ pointer doubling to achieve this (Figure 7). Each vertex sets its value to its successor's successor, converging after $log(l)$ steps, where l is longest distance of any vertex from its representative.

As shown in Figure 7 the array index on the right indicates the vertex id and the array value indicates vertex id of the representative vertex. So it indicates that vertex 0 and vertex 4 belong to a subgraph with vertex 0 as representative and vertex 1, 2, 3 and 5 belong to subgraph having vertex 2 as representative.

2.6 Form Supervertices

Next, we merge vertices into supervertices. Here, we've used our own logic since split primitive proposed by authors is based on 64 bit values.

After the successor array is created and cycles are removed, we bring together all the vertices belonging to same supervertex. Authors have used split primitive which they implemented themselves. It isn't part of cudpp library. So we implemented the same functionality using cudppRadixSort primitive. We create another array, where value of array is same as its index. Then, we use cudppRadixSort with CUDPP_OPTION_KEY_VALUE_PAIRS option so that the previous mapping between vertex and its representative vertex is maintained.

2.7 Assign IDs

Using a scan on O(V) elements, each of these supervertices is assigned new vertex ids. As we have seen, initially, for each vertex, its value was the representative vertex to which it belonged. We brought together all vertices having same representative vertex and created flag array to mark the boundary (Figure 8). Scan of this flag array assigned new vertex ids to each vertex. The output indicates that vertex 0 and 4 (of previous graph) now are part of same supervertex having id 0. Also, vertices 1, 5, 3, 2 are now part of same supervertex having id 1.

2.8 Remove self-edges

Then we remove self-edges by looking at supervertex ids of both vertices. We shorten the edge-list by removing self edges in the new graph. The idea is for each edge to look at the supervertex id of both endpoints and removes itself if found same.

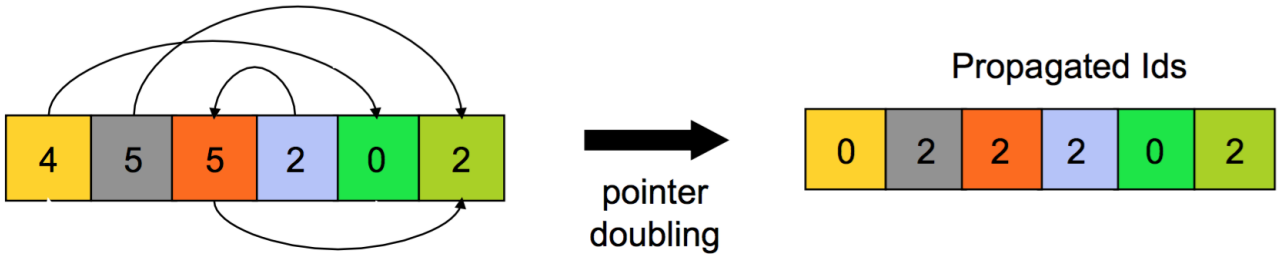

Figure 7: Pointer Doubling

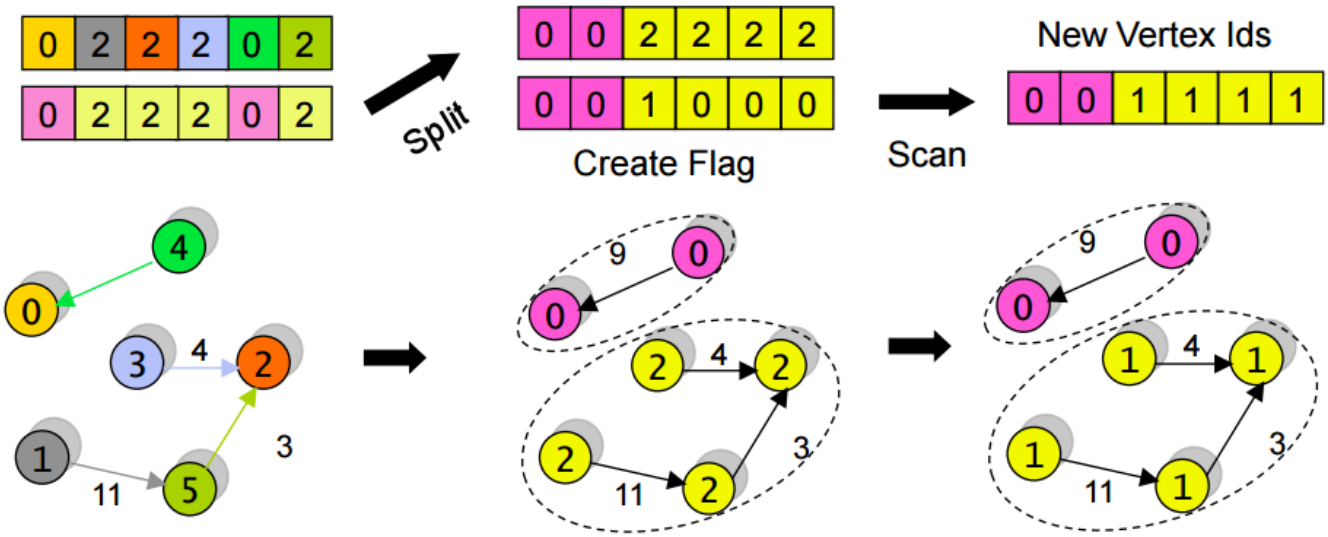

Figure 8: Bringing Vertices together

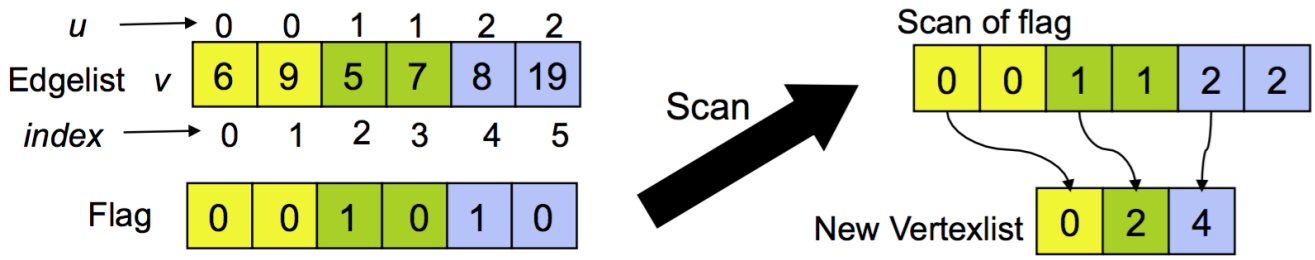

Figure 9: Creating the new Vertex List

2.9 Create New Vertex List

The Vertex list contains the starting index of each vertex in the newly created edge list. In order to find the starting index we scan a flag based on distinct supervertex ids in the edge-list. This gives us the index where each vertex should write its starting value. Compacting the entries gives us the desired new vertex list.

2.10 Recursive Call

The vertex, edge, and weight lists constructed represent the compact graph of supervertices. The procedure described above can now be applied recursively on it. Since all edges in the reduced graph correspond to an edge in the original input graph, a mapping is maintained through all the recursive invocations. This mapping is used to mark a selected edge in the output MST array. The recursive invocations continue till we reach a single vertex.

3 The Algorithm

The Algorithm 1 below presents the complete, recursive Boruvka's MST algorithm as reported in the previous section. In our hybrid implementation, part of the work in the method calls is offloaded to CPU if number of vertices is more than 20 and number of edges is more than 200. Otherwise, naive GPU implementation is used.

Algorithm 1 Boruvka MST 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 NWE.
4: Find the successor of each vertex and add to successor array, S.
5: Remove cycle making edges from NWE using S, and identify representatives vertices.
6: Mark remaining edges from NWE as part of output in MST.
7: Propagate representative vertex ids using pointer doubling.
8: Append successor array's entries with its index to form a list, L.
9: Split L, create flag over split output and scan the flag to find new ids per vertex, store new ids in C.
10: Find supervertex ids of u and v for each edge using C.
11: Remove edge from edge-list if u, v have same supervertex id.
12: Compact and create the new edge-list and weight list.
13: Build the vertex list from the newly formed edge-list.
14: Call the MST Algorithm on the newly created graph until a single vertex remains.

4 Experimental Setup

To test our experiments, we had access to GPU on stampede supercomputer. It has 128 compute nodes with NVIDIA Kepler K20 GPUs[2]. We also setup Amazon AWS g2.8x instance which has 4 Nvidia K520 GPU, each with 1536 cores[3]. Since amazon has more compute power, we expect it to perform better than stampede.

We also had to install CUDPP library on each of these machines to make use of the primitives. The steps used to install CUDPP were as follows:

1. git clone –recursive https://github.com/cudpp/cudpp.git
2. cd cudpp
3. mkdir build
4. cd build
5. export LD_LIBRARY_PATH= /usr/lib64:$LD_LIBRARY_PATH
6. module load cuda
7. cmake -D CUDA_TOOLKIT_ROOT_DIR= /opt/apps/cuda/6.5/lib64/ ..
8. make

The dataset which we used for testing is same as the one mentioned in paper [1] - DIMACS USA road networks[4]. The dataset in the form of graph has number of edges varying from 733,846 to 8,778,114 and number of vertices varying from 264,346 to 3,598,623.

5 Results

We tried our implementation on two different GPUs - Tesla K20 on Stampede Supercomputer provided for this course and Nvidia K520 on Amazon AWS (g2.8x large instance). The results are shown below:

[2]https://www.tacc.utexas.edu/stampede/
[3]https://aws.amazon.com/ec2/instance-types/
[4]http://www.dis.uniroma1.it/challenge9/download.shtml

USA road network graphs	Vertices	Edges	CPU time as in paper (in ms)	GPU time - Tesla K20 (in ms)
New York	264346	733846	780	34
San Fransico	321270	800172	870	35
Colorado	435666	1057066	1280	39
Florida	1070376	2712798	3840	47
North west USA	1207945	2840208	4290	48
North east USA	1524453	3897636	6050	57
California	1890815	4657742	7750	64
Great lakes	2758119	6885658	12300	85
USA-East	3598623	8778114	16280	102

Figure 10: CPU vs GPU (Tesla K20) Time Comparison

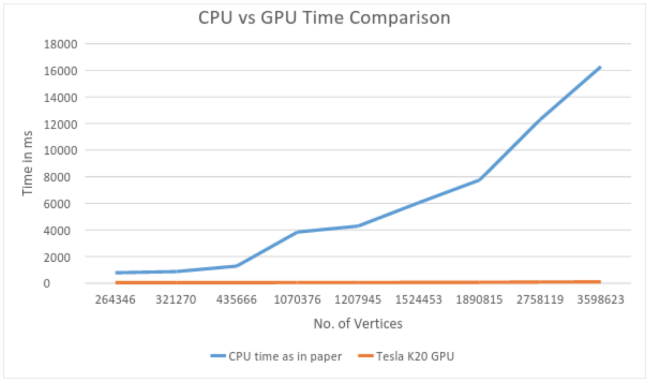

Figure 11: CPU vs GPU (Tesla K20) Time Comparison

USA road network graphs	Vertices	Edges	GPU time - Tesla K20 (in ms)	Hybrid Approach time (in ms)
New York	264346	733846	34	37
San Fransico	321270	800172	35	38
Colorado	435666	1057066	39	44
Florida	1070376	2712798	47	63
North west USA	1207945	2840208	48	64
North east USA	1524453	3897636	57	78
California	1890815	4657742	64	87
Great lakes	2758119	6885658	85	118
USA-East	3598623	8778114	102	139

Figure 12: GPU vs proposed Hybrid approach on Tesla K20

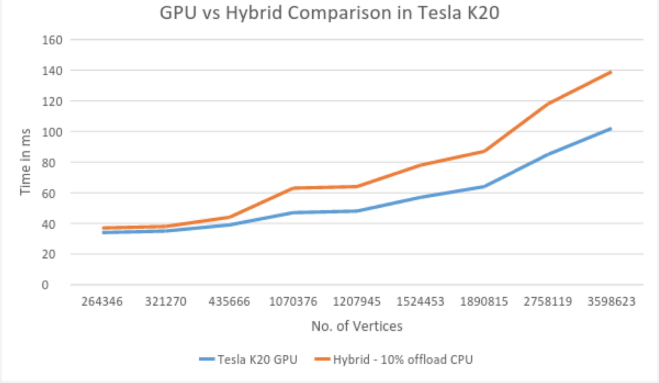

Figure 13: GPU vs proposed Hybrid approach on Tesla K20

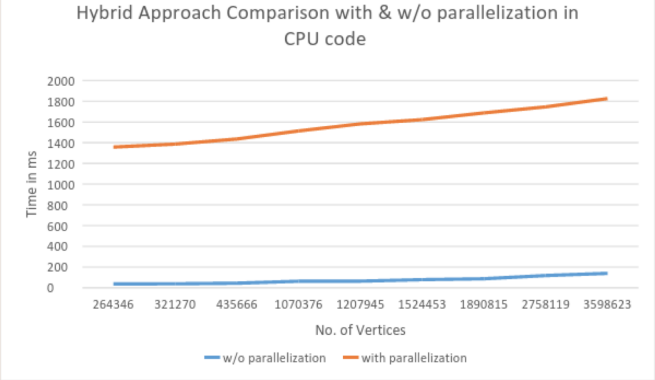

Figure 14: Hybrid approach comparison with and without parallelization in CPU

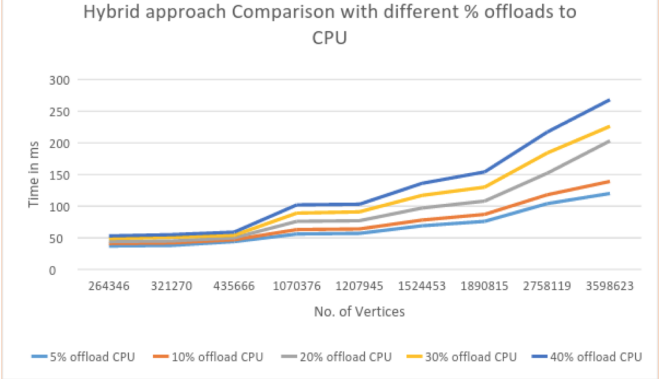

Figure 15: Hybrid comparison with different percentage of offloads

USA road network graphs	Vertices	Edges	GPU time - Tesla K20 (in ms)	GPU time - Nvidia K520 (in ms)
New York	264346	733846	34	21
San Fransico	321270	800172	35	22
Colorado	435666	1057066	39	26
Florida	1070376	2712798	47	42
North west USA	1207945	2840208	48	43
North east USA	1524453	3897636	57	54
California	1890815	4657742	64	61
Great lakes	2758119	6885658	85	83
USA-East	3598623	8778114	102	98

Figure 16: Tesla K20 vs Nvidia K520 Running time

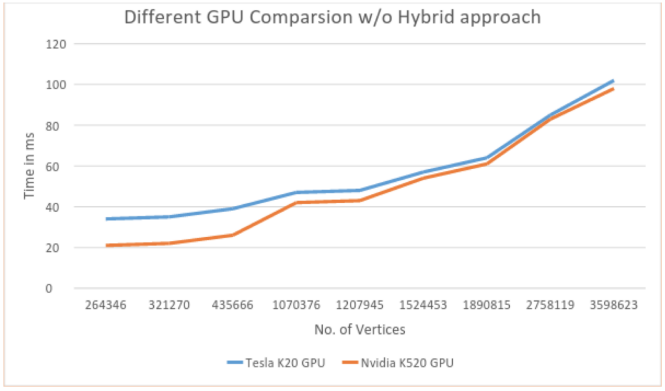

Figure 17: Tesla K20 vs Nvidia K520 without our Hybrid approach

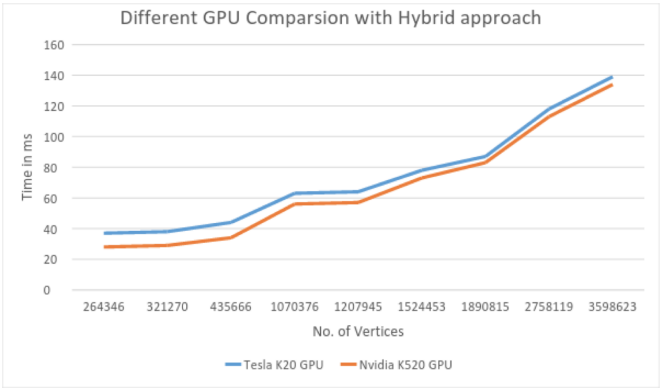

Figure 18: Tesla K20 vs Nvidia K520 with our Hybrid approach

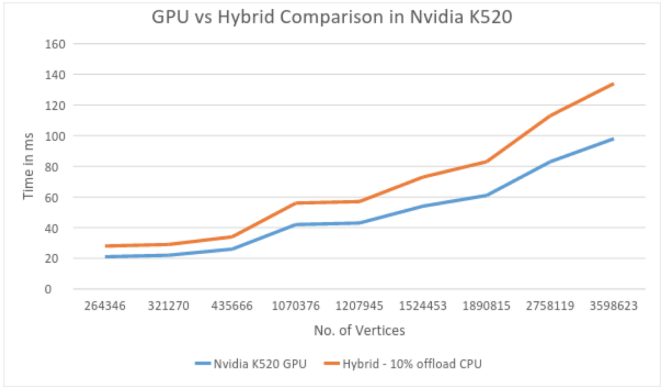

Figure 19: GPU vs proposed Hybrid approach on Nvidia K520

5.1 Tesla K20

For CPU running time, we've used the time as mentioned by Vineet et al 2009 (Figure 10). As we can see, if the number of vertices increases, the running time also increases. Our GPU implementation takes very few milliseconds to run and hence the graph looks almost linear (Figure 11).

In Figure 12, we compare the running time of a GPU version with our hybrid algorithm for the USA road network graph. We compare running time of our algorithm where we do not offload computation to GPU vs running time when we offload 10% computation to CPU. As we can see, in both cases running time increases with increase in number of vertices. However, running time when we offload computation to CPU is much more as compared to the running time when we don't offload (Figure 13). The reason for this is because the CPU computation is done serially. So the program has to wait till CPU completes its computation and this contributes to increase in running time. GPU executes kernel asynchronously hence we see that it is faster.

In Figure 14, we augment our code to do CPU computation in parallel using openMP and compare it with algorithm where we don?t parallelize CPU computation. We observe that hybrid approach with openMP takes more time than its counterpart. This appears because there is overhead attached with using openMP related to setting of threads, dividing work etc.

In Figure 15, we offload different percentage of input data to CPU and observe their running time. For each of these scenarios, running time increases with increase in number of vertices as expected. We get the best running time for 5% offload to CPU. This is because in our code, the work is done sequentially on CPU. So if we offload relatively less work to CPU, the program doesn?t have to wait much to proceed

with the next steps. Offloading more workload to CPU implies more work will be done sequentially on CPU, negating the benefit of asynchronous GPU execution. Hence, 40% offloading takes most time while 5% offload yields best results as expected.

5.2 Nvidia K520

In Figure 16, we compare running time of Boruvka's algorithm without hybrid implementation on Stampede's Tesla K20 GPU with Amazon AWS' Nvidia K520 GPU. Again, running time increases with increase in number of vertices. For lesser number of vertices, Nvidia K520 GPU is significantly faster but as number of vertices increase, the running time on both the GPUs is almost similar (Figure 17). Nvidia K520 has more cores than Tesla K20 and hence we observe faster running times for it.

In Figure 18, we compare running time of hybrid Boruvka's algorithm on stampede with Tesla K20 GPU and Amazon AWS with Nvidia K520 GPU. Again, running time increases with increase in number of vertices. For lesser number of vertices, Nvidia K520 GPU is significantly faster but as number of vertices increase, the running time on both the GPUs is almost similar. Nvidia K520 has more cores than Tesla K20 and hence we observe faster running times for it. In this case, we offload 10% of data to CPU.

In Figure 19, we compare running time of Boruvka's algorithm with and without hybrid computation on Nvidia K520 GPU. Again, running time increases with increase in number of vertices. In this scenario, we observe better results when we do not offload data to CPU.

6 Conclusion

In this project, we presented a formulation of the MST problem into a hybrid CPU-GPU framework, building upon the work done by Vineet et al. We implemented Boruvka's algorithm on GPU and augmented it so that part of computation is done on CPU. We ran different experiments, first varying the percentage of data to be offloaded to CPU and then running our algorithm on different GPUs. We achieved substantial speedup over CPU implementation. However, as per our experiments, due to extra overhead associated with openMP, the hybrid approach didn't yield a better result than non-hybrid approach.

7 Limitations

1. We've followed the approach mentioned by the authors where they use 10 bits to store weight of an edge. Thus, the program can run only for those graphs where edge weight is less than 1024 bits.

2. If graph has v vertices, e edges, the program reads graph in following format:

```
v
<start index of outgoing edge
   of vertex 0> <no of outgoing
    edges for vertex 0>
<start index of outgoing edge
   of vertex 1> <no of outgoing
    edges for vertex 1>
.
.
.
<start index of outgoing edge
   of vertex v-1> <no of
   outgoing edges for vertex v
   -1>
0
e
<vertex id of outgoing edge
   from vertex 0> <weight of
   that edge>
...
<vertex id of outgoing edge
   from vertex v-1> <weight of
   that edge>
```

   Thus, for program to work on any other type of graph, it has to be converted to above format.

3. Number of vertices should be less then 2^{22} since we're using 22 bits to store vertex ids. Thus, number of vertices should be less than 4.1 million (approximately).

8 Future Work

1. It'd be interesting to see how results vary if we change the grain size of vertices (20) and edges(200).
2. Another experiment which can be done is to vary the functions whose computations can be offloaded to CPU.
3. More rigorous analysis can be done to determine how much percentage of input data can be offloaded to CPU.

References

[1] Vineet, V.; Harish, P.; Patidar, S.; and Narayanan, P. J.; "Fast minimum spanning tree for large graphs on the GPU, " in HPG '09: Proceedings of the Conference on High Performance Graphics 2009, 2009, pp. 167-171.

[2] Chazelle, B.; 2000, "A minimum spanning tree algorithm with inverse-Ackermann type complexity", J. ACM 47, 6, 1028?1047.

[3] Harish, P.; Vineet, V.; and Narayanan, P. J.; 2009, "Large Graph Algorithms for Massively Multithreaded Architectures", Tech. Rep. IIIT/TR/2009/74.

[4] Boruvka, O.; 1926, O "Jistém Problému Minimálním (About a Certain Minimal Problem)" (in Czech, German summary). Pra?ce Mor. Pr??rodoved. Spol. v Brne III 3.

[5] Sengupta, S.; Harris, M.; Zhang, Y.; and Owens, J. D.; 2007, "Scan primitives for gpu computing", In Graphics Hardware, 97–106.

[6] Da Silva Sousa, C.; Mariano, A.; Proenca, A.; "A Generic and Highly Efficient Parallel Variant of Boruvka?s Algorithm", Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on, Issue Date: 4-6 March 2015.

[7] https://en.wikipedia.org/wiki/Minimum_spanning_tree

[8] https://commons.wikimedia.org/wiki/File:Boruvka%27s_algorithm_(Sollin%27s_algorithm)_Anim.gif

[9] http://www.dis.uniroma1.it/challenge9/download.shtml

[10] http://highperformancegraphics.net/previous/www_2009/presentations/vineet-fast.pdf

[11] http://cudpp.github.io/cudpp/2.2/group__public_interface.html