Fast Minimum Spanning Tree for Large Graphs on the GPU

by

Vibhav Vineet, Pawan Harish, Suryakanth Patidar, P J Narayanan

in

HPG 2009
(*High Performance Graphics*)

Report No: IIIT/TR/2009/197

Centre for Visual Information Technology
International Institute of Information Technology
Hyderabad - 500 032, INDIA
August 2009

Fast Minimum Spanning Tree for Large Graphs on the GPU

Vibhav Vineet
vibhavvinet@research.iiit.ac.in

Pawan Harish
harishpk@research.iiit.ac.in

Suryakant Patidar
skp@research.iiit.ac.in

P. J. Narayanan
pjn@iiit.ac.in

Center for Visual Information Technology
International Institute of Information Technology, Hyderabad, India

Abstract

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. Irregular algorithms on discrete structures like graphs are harder to map to them. Efficient data-mapping primitives can play crucial role in mapping such algorithms onto the GPU. In this paper, we present a minimum spanning tree algorithm on Nvidia GPUs under CUDA, as a recursive formulation of Borůvka's approach for undirected graphs. We implement it using scalable primitives such as scan, segmented scan and split. The irregular steps of supervertex formation and recursive graph construction are mapped to primitives like split to categories involving vertex ids and edge weights. We obtain 30 to 50 times speedup over the CPU implementation on most graphs and 3 to 10 times speedup over our previous GPU implementation. We construct the minimum spanning tree on a 5 million node and 30 million edge graph in under 1 second on one quarter of the Tesla S1070 GPU.

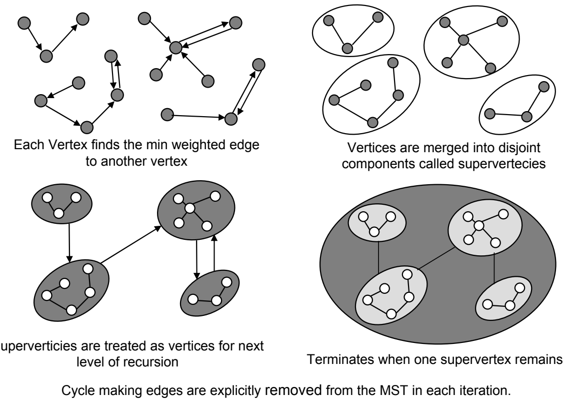
1 Introduction

Modern Graphics Processing Units (GPUs) provide 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. The GPU has a general, data-parallel programming interface through CUDA and the recently adopted OpenCL standard [Munshi 2008]. GPUs have become the affordable and accessible computing coprocessors.

The GPUs implement a data parallel architecture, with a common kernel simultaneously processing a number of data instances. Today's GPUs are also massively mulithreaded with thousands of threads in flight simultaneously. Such a model is best suited to process independent data instances. Providing global mapping or ordering to elements distributed in such a scale is a challenge in processing less regular data on these architectures. The memory model and thread scheduling are complex. Suboptimal implementations suffer severe performance penalties as a result. A non-expert programmer can get good performance only if efficient data mapping primitives are used. Primitives such as scan [Sengupta et al. 2007; Dotsenko et al. 2008], sort, and reduce are available for this purpose. Reduction of irregular steps into a combination of such primitives is critical to gain performance on the GPUs.

Minimum spanning tree (MST) computation on a general graph is an irregular algorithm. The best sequential time complexity for MST is $O(E\alpha(E, V))$, given by Chazelle, where α is the functional inverse of Ackermann's function [Chazelle 2000]. Borůvka's approach to the MST problem takes $O(ElogV)$ time [Boruvka 1926]. Several parallel variations of this algorithm have been proposed [King et al. 1997]. Chong et al. [Chong et al. 2001] report an EREW PRAM algorithm with $O(logV)$ time and $O(V logV)$ work. Bader et al. [Bader and Cong 2005] provide an algorithm for symmetric multiprocessors with $O((V + E)/p)$ lookups and local operations for a p processor machine. Chung et al. [Chung and Condon] efficiently implement Borůvka's approach on an asynchronous distributed memory machine by reducing communication costs. Dehne and Götz implement three variations using the BSP model [Dehne and Götz 1998]. Blelloch [Blelloch 1989] formulates MST using the scan primitive. A GPU implementation using CUDA is also reported [Harish et al. 2009], using irregular kernel calls. We, on the other hand exploit parallel data mapping primitives available on the GPU.

Borůvka's approach finds the minimum weighted outgoing edge at each vertex and merges connected vertices into supervertices (Figure 1). Merging is an irregular operation as multiple vertices are assigned to a single supervertex.



Each Vertex finds the min weighted edge to another vertex

Vertices are merged into disjoint components called supervertecies

Superverticies are treated as vertices for next level of recursion

Terminates when one supervertex remains

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

Figure 1: *Steps of Borůvka's approach to MST*

We formulate Borůvka's approach in a recursive framework, with each step implemented using a series of basic primitives. In each iteration, we reconstruct the supervertex graph which is given as an input to the next level of recursion. We use the segmented scan [Sengupta et al. 2007] to find the minimum weighted outgoing edge from each vertex. We then merge vertices into supervertices using split and scan primitives. We use a scalable split primitive for this [Patidar and Narayanan 2009]. A simple kernel then eliminates the cycles. Another split/scan pair is used to remove duplicate edges and to assign supervertex numbers. The main contribution of this work is the recursive formulation of Borůvka's approach for undirected graphs as a series of general primitive operations. We compute the minimum spanning tree of a 5M vertex and 30M edge graph in under one second on a quarter of a Tesla S1070U.

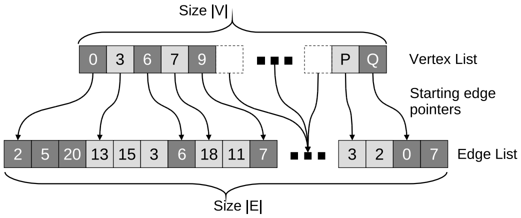


Figure 2: *The graph representation*

2 Graph representation and primitives

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.

Primitives like sort and scan have become fundamental to algorithm analysis. Many primitives have been ported to the GPU including scan, segmented scan, reduce, sort [Sintorn and Assarsson 2008] and split. In our MST implementation we use three parallel primitives: scan, segmented scan and split. We utilize CUDPP library implementation [Sengupta et al. 2007] for scan and segmented scan. We use *split* operation to bring together the vertices to be merged into supervertices and to remove duplicate edges between supervertices for recursive application. We use the implementation given in [Patidar and Narayanan 2009].

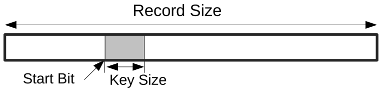
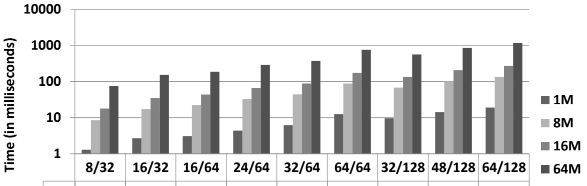


Figure 3: *Record, key, and start bit for split primitives*

Split is a function that partitions an input relation into a number of categories and has a wide use in databases, data structure building and distributed data mapping. Efficient split can be a fundamental building block to many applications. We use a scalable and efficient implementation of split that can handle a wide range of record sizes and key lengths. The variation of split we use has the form **split**$(list[], R, K, S)$ which splits the list of *R*-byte records using a *K*-bit key starting at bit *S* of the record (Figure 3). Splitting time scales linearly with the number of records and key size. We use a split on a 64-bit record with a 32-bit key and another on a 64-bit record with a 64-bit key in the algorithm presented in this paper. Figure 4 summarizes the performance of the split primitive. More details can be seen at [Patidar and Narayanan 2009] and the code is available at `http://cvit.iiit.ac.in/resources`.



	8/32	16/32	16/64	24/64	32/64	64/64	32/128	48/128	64/128
1M	1.3	2.7	3.1	4.4	6.2	12.5	9.6	14.2	19.1
8M	9	17	22	33	44	89	69	102	136
16M	18	35	44	67	89	177	137	207	273
64M	76	155	189	291	376	767	568	858	1167

Figure 4: *Performance of split for different key/record sizes and lists. X-axis represents combinations of key-size/record size.*

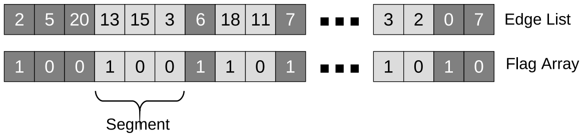


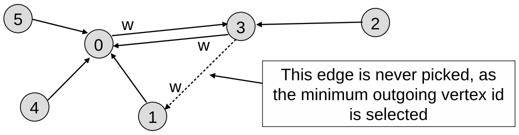
Figure 5: *Segments based on difference in* u*, MarkSegments()*

3 Minimum spanning tree algorithm

The Borůvka's approach is summarized in Figure 1. Each vertex of the graph finds 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. Let array *V* be the vertex-list, *E* the edge-list, and *W* the weight-list. The output of our implementation is a list of edges present in *MST*.

3.1 Marking the MST edges

Finding minimum weighted edge: Each vertex u finds the minimum 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 8-10 bits, leaving 22-24 bits for v. The resulting integer array *X* is scanned with a flag array *F* specifying the segment. We use a kernel that runs over edges to compute F. It marks the first edge in the continuous edge list for each u as shown in Figure 5. This operation, called *MarkSegments()*, is used at other places of the algorithm also. The segmented min scan on *X* returns the minimum weighted edge and the minimum outgoing vertex v for every vertex u. The index of this edge is stored in the temporary array *NWE*.



This edge is never picked, as the minimum outgoing vertex id is selected

Figure 6: *Cycles can only result between two vertices*

Finding and removing cycles: Each vertex u has an outgoing edge assigned to it in the previous step. Since there are $|V|$ vertices and $|V|$ edges, we can expect at least one cycle to be formed. Detecting and removing cycles is crucial. The cycles in an undirected case can only result between exactly two vertices, as each vertex is given a unique id initially and the minimum of (weight, v) is chosen by the segmented scan (Figure 6). We use this property to eliminate cycles. We create a successor array *S* using the *NWE* array to hold the outgoing v for each u (Figure 7). Vertices with *S*(*S*(u))= u form cycles. The edge from the lower of u and $S(u)$ is removed from *NWE* and its successor is set to itself (Figure 8). Remaining vertices mark their selected edges in the output *MST* array.

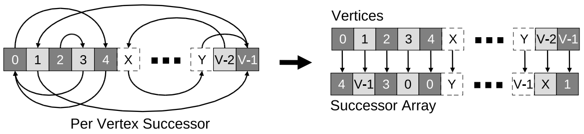

0 1 2 3 4 X ... Y V-2 V-1
Per Vertex Successor
Vertices
0 1 2 3 4 X ... Y V-2 V-1
4 V-1 3 0 0 Y ... V-1 X 1
Successor Array

Figure 7: *Creating the successor array*

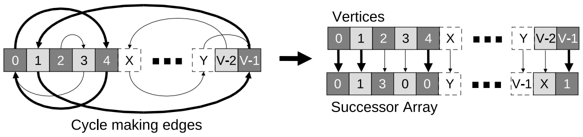


Figure 8: *Updating successors of cycle forming vertices*

3.2 Graph construction

Merging vertices: Vertices combine to form a supervertex next. The vertices whose successors are set to themselves are representatives for each supervertex. Other vertices should point to their respective representative vertex. We employ pointer doubling to achieve this. 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 (Figure 9). We iteratively set $S(u)$=$S(S(u))$ until no further change occurs in S.

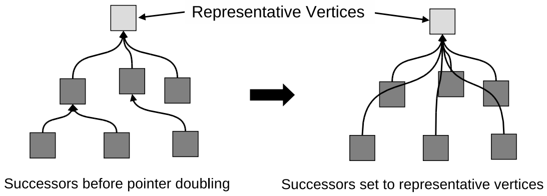

Representative Vertices
Successors before pointer doubling
Successors set to representative vertices

Figure 9: *Pointer doubling gets to the representative vertex*

Assigning ids to supervertices: Each vertex of a supervertex now has a representative, but they are not numbered in order. The vertices assigned to a supervertex are also not placed in order in the successor array. We next bring all vertices of a supervertex together and assign new unique ids to supervertices. We form a size $|V|$ list L of 64 bit values with the vertex id to the right and its representative vertex id to the left. L is split using the representative vertex id as the key. This results in vertices with same representatives coming together as shown in Figure 10. This, however, does not change the ids of representative vertices. We create a flag to mark the boundaries of representative vertices using *MarkSegments()* as mentioned in Section 3.1. A scan of the flag assigns new supervertex ids (Figure 10). These values are stored in an array C.

Removing edges and forming the new edge list: We shorten the edge-list by removing self edges in the new graph. Each edge looks at the supervertex id of both endpoints and removes itself if same. For an edge from u to v, the supervertex id of v can be found directly by indexing C using the value part of the split output of v. The supervertex id of u requires another scan of size E, as the vertex list does not store the original id of u explicitly. For this, we create a flag using the vertex list with a 1 placed in the location to which the vertex u points in the edge-list (Figure 5). A scan of the flag gives the original id for u, using which we can look up its supervertex id in the C array.

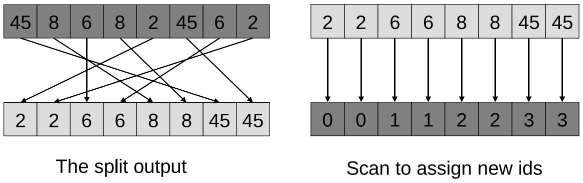


Figure 10: *Bring vertices belonging to same each supervertex together using a split (left). Assign ids to supervertices using a scan.*

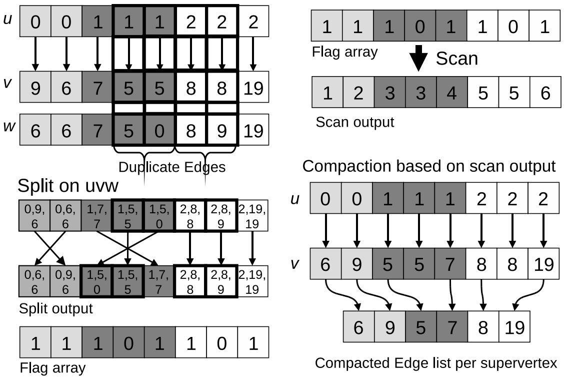


Figure 11: *Removing duplicate edges based on uvw split*

Duplicate edges from one supervertex to another may exist in the new set of edges even after removing edges belonging to the same supervertex. We eliminate non-minimal duplicate edges between supervertices to reduce the edge-list further. We use a split on the remaining edge-list for this. We append the supervertex ids of u and v along with weight w into a single 64-bit array *UVW*, consisting of the new id of u (left 24 bits), the new id of v (next 24 bits) and the weight w (right 16 bits). We apply a 64-bit split on *UVW* array and pick the fist distinct entry for each uv combination. We create a flag on the output of split, based on the difference in uv pair, using *MarkSegments()*. This gives the minimum weighted edge from the supervertex of u to supervertex of v. A scan of the flag array demarcating uv values returns the location of each entry in the new edge-list. A simple compaction produces a new edge-list as shown in Figure 11, removing duplicate edges. A corresponding weight list is also created similarly. We also store the original edge id of each edge while compacting entries into the new edge-list. This is used while marking an edge in the output *MST* array in subsequent recursive applications. The edge-list and the weight-list thus created act as inputs for the next recursive application of the MST algorithm.

Constructing the vertex list: To create the vertex list, we need the starting index for each supervertex in the new edge-list. A flag based on difference in u can be created on the new edge-list using *MarkSegments()*. A scan of this flag gives us the index of the location to which the starting index must be written (Figure 12). Compacting these entries gives us the desired vertex-list.

3.3 Recursive invocation

The vertex, edge, and weight lists constructed represent the compact graph of supervertices. The procedure described above can

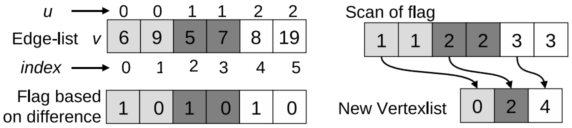


Figure 12: *Creating the vertex list using edge-list*

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.

3.4 The Algorithm

Algorithm 1 presents the complete, recursive MST algorithm as reported in the previous section.

Algorithm 1 MST_Algorithm

```
Append weight $w$ and outgoing vertex $v$ per edge into a list, $X$.
Divide the edge-list, $E$, into segments with 1 indicating the start of each segment, and 0 otherwise, store this in flag array $F$.
Perform segmented min scan on $X$ with $F$ indicating segments to find minimum outgoing edge-index per vertex, store in $NWE$.
Find the successor of each vertex and add to successor array, $S$.
Remove cycle making edges from $NWE$ using $S$, and identify representatives vertices.
Mark remaining edges from $NWE$ as part of output in $MST$.
Propagate representative vertex ids using pointer doubling.
Append successor array's entries with its index to form a list, $L$
Split $L$, create flag over split output and scan the flag to find new ids per vertex, store new ids in $C$.
Find supervertex ids of $u$ and $v$ for each edge using $C$.
Remove edge from edge-list if $u$, $v$ have same supervertex id.
Remove duplicate edges using split over new $u$, $v$ and $w$.
Compact and create the new edge-list and weight list .
Build the vertex list from the newly formed edge-list.
Call the MST_Algorithm on the newly created graph until a single vertex remains.
```

4 Performance analysis

We test our algorithm using a quarter unit of Nvidia Tesla S1070 with 240 stream processors spread over 30 multiprocessors and 4096MB of device memory. Use an Intel Core 2 Quad, Q6600, 2.4GHz processor and the Boost C++ graph library compiled using *gcc -O4* for CPU comparison. We show results on three types of graphs from the Georgia Tech graph generator suite [Bader and Madduri 2006] and on the DIMACS USA road networks [DIMACS 2006].

- Random Graphs: These graphs have a short band of degree where all vertices lie, with a large number of vertices having similar degrees.
- R-MAT: Large number of vertices have small degree with a few vertices having large degree. This model is best suited to large represent real world graphs.
- SSCA#2: These graphs are made of random sized cliques of vertices with a hierarchical distribution of edges between cliques based on a distance metric.

Our recursive algorithm has two basic steps: marking edges and graph construction. Graph construction requires a few additional arrays and increases the memory requirements. This limits the size of the graph that can be handled on the current GPUs. CUDPP segmented min scan works only on integer inputs. Weights and vertex ids need to be packed into a single 32-bit entry, restricting their ranges. We show results for graphs up to 5M vertices with 30M edges with a maximum weight range 1 to 1K.

We compare our results with a previous implementation from our group given in [Harish et al. 2009]. We achieve a speed up of 2 to 3 for random and SSCA#2 graphs, and of 8 to 10 for R-MAT graphs. We achieve a speed up of nearly 30 to 50 times over the CPU on all graph models. The speed up over the previous implementation is due the use of scans and split instead of irregular *atomic* operations for marking MST edges. In the previous implementation, each vertex wrote the minimum weighted edge to its supervertex id atomically in the global memory, resulting in a serialization of clashes. The marking of MST edges per vertex was linear in the previous implementation, resulting in load imbalances and thread divergence. Segmented scan reduces this to a logarithmic step.

Figure 13(a) presents results on varying graph sizes for the random graph model. The speedup is due to elimination of the atomic operations at a single location in global memory. Figure 13(b) shows the behavior on varying degree for the random graphs with 1M vertices. Because segmented scan is a logarithmic operation, we see only a small increase in the time with increasing degree. Figure 13(c) shows the times on varying sizes of R-MAT graphs. We gain a speed up of nearly 50 over CPU and 8-10 over the previous implementation. Segmented scan finds the minimum weighted outgoing edge quickly even if there is large variation in segment sizes as in case of R-MAT graphs. Varying the degree of R-MAT graph for a 1M vertex graph shows a similar speed gain (Figure 13(d)). Figures 13(e) and 13(f) show results on the SSCA#2 graph model for different numbers of vertices and degree respectively. We achieve a speed up of nearly 30 times over the CPU implementation and 3-4 times over previous implementation.

USA Graphs	Vertices	Edges	Time in ms		
			CPU	GPU[1]	Our GPU
New York	264K	733K	780	76	39
San Fransisco	321K	800K	870	85	57
Colorado	435K	1M	1280	116	62
Florida	1.07M	2.7M	3840	261	101
Northwest USA	1.2M	2.8M	4290	299	124
Northeast USA	1.52M	3.8M	6050	383	126
California	1.8M	4.6M	7750	435	148
Great Lakes	2.7M	6.8M	12300	671	204
USA-East	3.5M	8.7M	16280	1222	253
USA-West	6.2M	15M	32050	1178	412

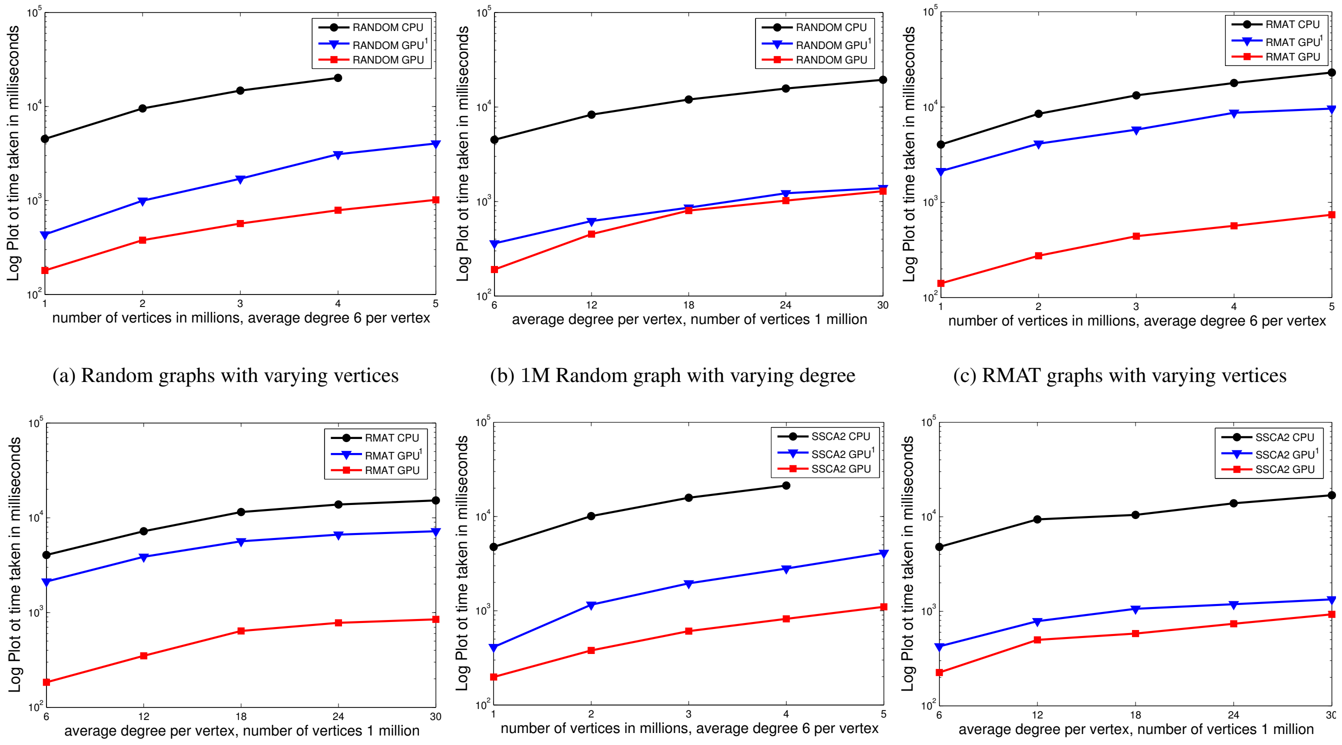
Table 1: *Results on the ninth DIMACS challenge graphs, weights in range* $1-1K$*. Times in Milliseconds*

Table 1 summarizes experiments on the ninth DIMACS challenge USA road network graphs. Since the segmented min scan requires vertex id and weights to be packed into 32 bits, we reduce the weights from $1-300\text{K}$ to range $1-1\text{K}$ for these graphs. A speed up of nearly 20 times over CPU for and nearly 3 times over the previous implementation was observed for our implementation for these graphs.

5 Conclusions and future work

In this paper, we presented a formulation of the MST problem into a recursive, primitive-based framework for the GPU. Optimized primitives can provide high speedups on such architectures. We achieved a speedup of nearly 50 times over the CPU and 8-10 times over the best GPU implementation. The recursive formulation is implemented on the GPUs, with the CPU used to synchronize the recursion. The use of efficient primitives to map the irregular aspects of the problem to the data-parallel architecture of these mas-

[1] Comparison with GPU implementation given in [Harish et al. 2009]



(a) Random graphs with varying vertices

(b) 1M Random graph with varying degree

(c) RMAT graphs with varying vertices

(d) 1M RMAT graph with varying degree

(e) SSCA2 graphs with varying vertices

(f) 1M SSCA2 graph with varying degree

Figure 13: *Experiments on varying sizes and varying degree for three types of graph*

sively multithreaded architectures is central to the high performance obtained. We are likely to see more general graph algorithms implemented using such primitives in near future.

Acknowledgment: We would like to thank Nvidia for their generous support, especially for providing hardware for this work.

References

BADER, D. A., AND CONG, G. 2005. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *J. Parallel Distrib. Comput. 65*, 9, 994–1006.

BADER, D. A., AND MADDURI, K., 2006. GTgraph: A synthetic graph generator suite, http://www.cc.gatech.edu/k̃amesh/gtgraph/.

BLELLOCH, G. E. 1989. Scans as Primitive Parallel Operations. *IEEE Trans. Computers 38*, 11, 1526–1538.

BORUVKA, O. 1926. O Jistém Problému Minimálním (About a Certain Minimal Problem) (in Czech, German summary). *Práce Mor. Prírodoved. Spol. v Brne III 3*.

CHAZELLE, B. 2000. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *J. ACM 47*, 6, 1028–1047.

CHONG, K. W., HAN, Y., AND LAM, T. W. 2001. Concurrent threads and optimal parallel minimum spanning trees algorithm. *J. ACM 48*, 2, 297–323.

CHUNG, S., AND CONDON, A. Parallel Implementation of Borvka's Minimum Spanning Tree Algorithm. In *IPPS 1996*.

DEHNE, F., AND GÖTZ, S. 1998. Practical Parallel Algorithms for Minimum Spanning Trees. In *SRDS '98: Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, 366.

DIMACS, 2006. The Ninth DIMACS challange on shortest paths http://www.dis.uniroma1.it/ challenge9/.

DOTSENKO, Y., GOVINDARAJU, N. K., SLOAN, P.-P. J., BOYD, C., AND MANFERDELLI, J. 2008. Fast scan algorithms on graphics processors. In *ICS*, ACM, 205–213.

HARISH, P., VINEET, V., AND NARAYANAN, P. J. 2009. Large Graph Algorithms for Massively Multithreaded Architectures. Tech. Rep. IIIT/TR/2009/74.

KING, V., POON, C. K., RAMACHANDRAN, V., AND SINHA, S. 1997. An optimal EREW PRAM algorithm for minimum spanning tree verification. *Inf. Process. Lett. 62*, 3, 153–159.

MUNSHI, A. 2008. OpenCL: Parallel computing o the GPU and CPU. In *SIGGRAPH, Tutorial*.

PATIDAR, S., AND NARAYANAN, P. J. 2009. Scalable Split and Gather Primitives for the GPU. Tech. Rep. IIIT/TR/2009/99.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for gpu computing. In *Graphics Hardware*, 97–106.

SINTORN, E., AND ASSARSSON, U. 2008. Fast parallel gpu-sorting using a hybrid algorithm. *J. Parallel Distrib. Comput. 68*, 10, 1381–1388.

[1] Comparison with GPU implementation given in [Harish et al. 2009]