Prefix Scan and Minimum Spanning Tree with OpenCL

U. VIRGINIA DEPT. OF COMP. SCI TECH. REPORT CS-2013-02

Yixin Sun and Kevin Skadron
Dept. of Computer Science, University of Virginia
ys3kz@virginia.edu, skadron@cs.virginia.edu

Abstract

GPUs have been widely used to achieve wide data-parallelism to facilitate the execution of concurrent computations for performance and efficiency. A lot of prior work have been done for CUDA, while OpenCL, as an open standard with the advantage of being able to run across multiple GPU platforms, still lacks development and efficient data primitives in its open standard libraries. This report discusses our work on constructing an efficient OpenCL implementation for parallel prefix scan and its improved performance over the scan primitive in the OpenCL standard library CLPP. The report also discusses our work on implementing the minimum spanning tree algorithm in OpenCL using our parallel segmented scan primitive and its improved performance over the original sequential execution.

1 Introduction

GPUs are designed for high data-parallel computations. They were initially used for graphics rendering applications, which involve applying the same operation to many pieces of data, which can be executed independently. The massive data parallelism that GPUs provide can greatly improve the aggregate throughput. Lots of recent interest in using GPUs has been on running non-graphics applications (GPGPU). CUDA has been a popular GPU programming model for many years, which is designed to run general-purpose applications on NVIDIA GPUs. Another popular GPU framework is OpenCL. Unlike CUDA, which is designed for and restricted to run on NVIDIA GPUs, OpenCL has the flexibility to run across many platforms, including Nvidia, AMD, Intel, etc.. The drawback is that it is harder to design algorithms that are optimized across different platforms for OpenCL. However, as OpenCL is being developed further with its cross-platform advantage, we think it is important and useful to develop more efficient algorithms for data-processing primitives and graph algorithms with OpenCL. This work contributes to a faster implementation of prefix scan compared to the scan primitive in the CLPP open library, and a parallel implementation of the minimum spanning tree (Boruvka's algorithm [1]) using the segmented scan primitive (a variation of the prefix scan) in OpenCL.

The GPU data-parallel model favors kernel design with minimal dependency across threads, more explicitly, each thread outputs to a location that is statically determined by its thread identifier (i.e. thread t_2 outputs to $dest_2$). Heavy input dependency can result in massive threads co-operations and excessive kernel calls, which will negatively impact the overall GPU performance. Thus, it is tricky to design parallel algorithms for data with high dependency. Many useful and important data-processing primitives have this characteristic of high data dependency (i.e. prefix scan). Such data primitives are important building blocks for various graph algorithms, e.g. prefix scan and radix sort have been used in breadth-first search (BFS) [2]. Even though given the data-dependency nature of the data primitives, we can still develop high-efficiency parallel algorithms that are suitable for such primitives, which will give us nice GPU speedup over single-threaded CPU. Merrill has the fastest prefix scan implementation by far with CUDA [3]. We implement the prefix scan with OpenCL using similar algorithm described by Merrill, and compare its performance against the scan primitive in the CLPP open library on Nvidia and AMD platforms. We achieve 3.1x overall speedup on Nvidia GeForce GTX 460 over single-threaded CPU (compared to 1.4x overall speedup with the CLPP library implementation) for 512MB input data size, and 1.7x overall speedup on AMD Tahiti over single-threaded CPU (compared to 2.7x slowdown with the CLPP library implementation) for 512MB input data size. The prefix scan primitive can be a useful building block for faster parallel implementation of many important graph algorithms, i.e., the minimum spanning tree (MST), which has various real world applications whenever there is a need to connect a set of nodes (cities, electrical terminals) by edges (roads, wires) which have different weights (length, costs). We use our own segmented scan implementation (the same approach as we implement the prefix scan) as a basic building block to implement a parallel version of the minimum spanning tree (using Boruvka's algorithm) with OpenCL. On Nvidia GeForce GTX 460, we achieve 1.26x total time speedup for dense graph with 128M edges, and 1.28x total time speedup for sparse graph with 128M edges.

2 Prefix Scan

Prefix Scan takes an n-element list $[a_0, a_1, ..., a_{n-1}]$ and a binary associative operator $\bigoplus$ as input, and produces an output list $[b_0, b_1, ..., b_{n-1}]$ of size n. Example of an exclusive scan:

$$
\begin{aligned}
&b_0 = id_{\bigoplus} \\
&b_1 = a_0 \\
&b_2 = a_0 \bigoplus a_1 \\
&... \\
&b_i = \bigoplus_{0 \le j \le i-1} a_j \\
&... \\
&b_n = \bigoplus_{0 \le j \le n-1} a_n
\end{aligned}
$$

Note: the $id_{\bigoplus}$ is the associated identity element for the $\bigoplus$ operator, such that $id_{\bigoplus} \bigoplus a_i = a_i$. E.g., $id_{\bigoplus} = 0$ for addition, $id_{\bigoplus} = 1$ for multiplication. Inclusive scan is similar to exclusive scan except that b_i also includes a_i in the operations, e.g., $b_i = \bigoplus_{0 \le j \le i} a_j$.

Prefix scan is a useful primitive in designing parallel algorithms. For example, in radix sort, prefix sum can be used to allocate space for buckets in an array [4], and in breadth-first search, prefix sum can be used to compute and distribute workload among threads [2]. However, the dependency nature of prefix scan creates potential challenges for running it in parallel. Besides the sequential implementation of prefix scan (which involves $O(n)$ steps and $O(n)$ worksize), there are three other common parallel implementations of prefix scan: Brent-Kung [5], Sklansky [6], and Kogge-Stone [7].

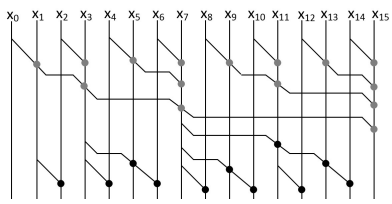

Figure 1: Brent-Kung

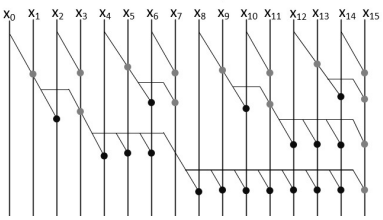

Figure 2: Sklansky

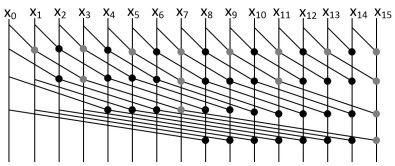

Figure 3: Kogge-Stone

All three have $O(\log n)$ steps. Brent-Kung has $O(n)$ worksize, while Sklansky and Kogge-Stone have $O(n \log n)$ worksize. Our implementation incorporates both the sequential subroutine and the Kogge-Stone subrountine within threadblocks, as will be desribed below.

If the input data size is bigger than what a single thread block can handle, then we need to use multiple thread blocks. This involves storing intermediate results back to the global memory and launching multiple kernel calls in order to scan the intermediate results and do the whole reduction. The two common ways to process the intermediate work for multiple thread blocks are: scan-then-propagate and reduce-then-scan.

Scan-then-propogate: in the upsweep phase, each thread block produces b partial reduction results from the b-element block it processes, and then the accumulated partial reduction results are sent to the next upper level and keep being accumulated until finally reaching the root. Since each level accumulates n/b results to its next upper level from its original n input elements, so the total number of kernel calls (correponding to the number of levels needed) for the upsweep phase is $\log_b n$. In the downsweep phase, the reduction results are propogated down level-by-level, where in each level, each thread is accumulating the original partial results with the new propogated reduction results. Similar to the upsweep phase, the downsweep phase also has $\log_b n$ levels and thus requires $\log_b n$ kernel calls. The advantage of this approach is the work-efficiency, since the intermediate reduction results are saved during the upsweep phase at each level, so in the downsweep phase, no redundant work is needed to recalculate the partial intermediate results. The Thrust standard library for CUDA [8] and the CLPP standard library for OpenCL [9] both employ this approach.
Reduce-then-scan: it is similar to the scan-then-propogate approach in that it also requires $\log_b n$ kernel calls for both the upsweep and downsweep phase. But instead of saving all b intermediate partial reduction results at each level, each thread block only saves one single intermediate results back to global memory. As a result, in the downsweep phase, redundant work is needed to recalculate the partial results and then accumulate them with the reduction results from its upper level. This approach is not work-efficient as the scan-then-propogate approach since it requires redundant work at each level in the downsweep phase, however, the advantage is that it avoids writing partial reduction results during the upsweep phase to the global memory, and thus greatly reduces memory access time.

Algorithm 1 Two-level Reduce-then-scan Upsweep Phase Pseudocode

```
1: for each tile_number in alltiles(block_id)
2:   shared_mem1[thread_id] = localreduction(tile_number,thread_id)
3:   barrier
4:   shared_mem2[tile_number] = sharedreduction(shared_mem1)
5: global_mem[block_id] = sharedreduction(shared_mem2)
```

Algorithm 2 Two-level Reduce-then-scan Spine Phase Pseudocode

```
1: shared_mem[thread_id] = global_mem[thread_id]
2: sharedreduction(shared_mem)
3: global_mem[thread_id] = shared_mem[thread_id]
```

Our implementation employs the two-level reduce-then-scan approach (Figure 4), proposed by Merrill [2]. Similar to the reduce-then-scan approach, each thread block writes only one intermediate reduction result to global memory for each data block it processes, however, instead of having each thread block process only one data block, the thread blocks are "reused" to process multiple data blocks in sequential. Suppose we have T thread blocks, n total input elements, and each data block contains b elements. Then each thread block will process $\frac{n}{bT}$ data blocks in sequential, and after processing all the data blocks, the thread block will reduce the $\frac{n}{bT}$ number of reduction results to a single intermediate value and write it back

Algorithm 3 Two-level Reduce-then-scan Downsweep Phase Pseudocode

```
1: accum_result = global_mem[block_id];
2: for each tile_number in alltiles(block_id)
3:   shared_mem[thread_id] = localreduction(tile_number,thread_id)
4:   barrier
5:   reduction_result = sharedreduction(shared_mem)
6:   global_mem[thread_id] = localaggregation(shared_mem[thread_id]) + accum_result
7:   accum_result += reduction_result
```

to global memory. Thus, we need only one kernel call for the upsweep phase, one kernel call to process the T intermediate results from the upsweep phase (called the spine phase), and one kernel call for the downsweep phase (similar to the upsweep phase, each thread block processes $\frac{n}{bT}$ data blocks in sequential). Therefore, instead of having $O(\log_b n)$ total kernel calls, this approach only needs three kernel calls. As input data size increases, the sequential work will increase for each thread block, while the number of kernel calls and global memory writes remain the same.

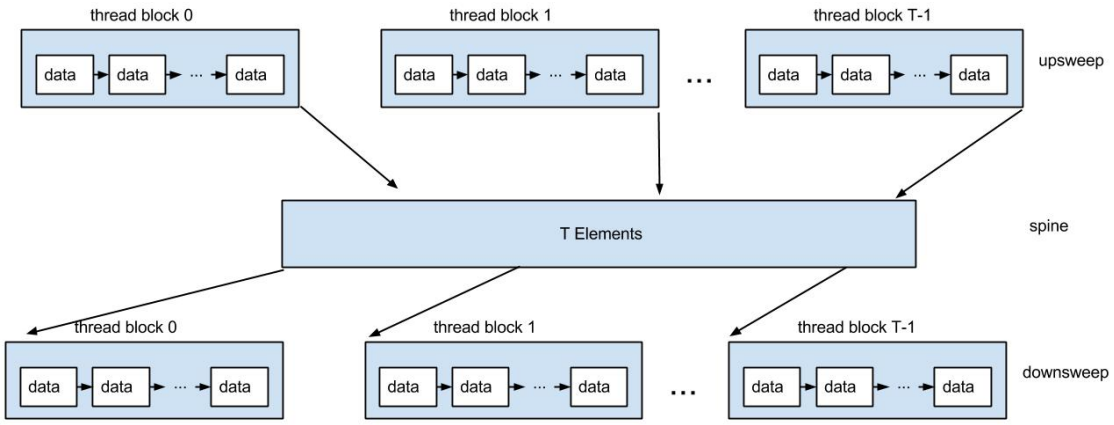

Figure 4: Two-level Reduce-then-Scan

Within each data block, we first do the reduction in sequential using the local registers in each thread, and use the Kogge-Stone warp scan approach to accumulate the results and place it into shared memory. After the thread block has processed all its data blocks, we use the Kogge-Stone warp scan approach again to accumulate the partial results in the shared memory, and finally write the one single intermediate result back to global memory. In the spine phase, we use Kogge-Stone approach to aggregate the intermediate results produced by each thread blocks from the upsweep phase. In the downsweep phase, as a reverse from the upsweep phase, each thread block loads the reduction result from the spine phase, and accumulates it with the each data element as the thread block processes sequentially through all the data blocks.

Although the two-level scan-then-propogate approach involves more sequential work than

the other two approaches, it greatly reduces the number of kernel calls and global memory accesses, which typically incur a lot of overhead in GPU programming. Also, instead of increasing the number of thread blocks as data size increases, we increase the sequential work of each thread block while keeping the number of thread blocks a constant number. The time saved by limiting the number of kernel calls turns out to significantly outweigh the time incurred by adding sequential work, as the data size grows bigger.

3 Prefix Scan Performance Analysis

We compare the performance of exclusive prefix sum of our improved implementation and the CLPP standard library for OpenCL [9] against sinlge-threaded CPU on both the Nvidia GeForce GTX 460 and AMD Tahiti GPU (based on average time of 50 consecutive runs of data type unsigned integer). We also measure the performance of Merrill's prefix sum (implemented in CUDA) [3] on Nvidia GeForce GTX 460 (note that the CUDA implementation can only handle data size up to 256MB, as will be discussed in detail later). We compare both the total time speedup (including data transfer time between CPU and GPU) and the kernel time speedup over single-threaded CPU (Intel Core i7).

We first measure the GPU kernel time (using the profiling tool) and its speedup over the CPU time. The x-axis represents the input data size in megabytes, and the y-axis is the time in milliseconds.

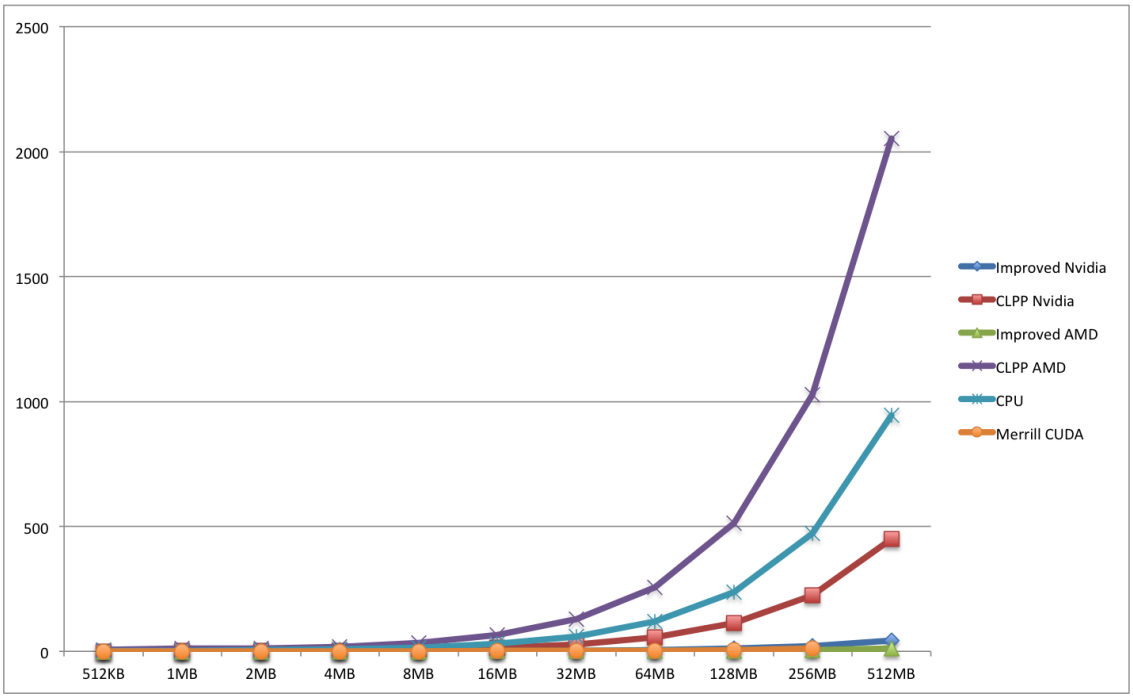

Figure 5: GPU Kernel Time

As showed in Figure 5 and Figure 6, our improved implementation has significant kernel time

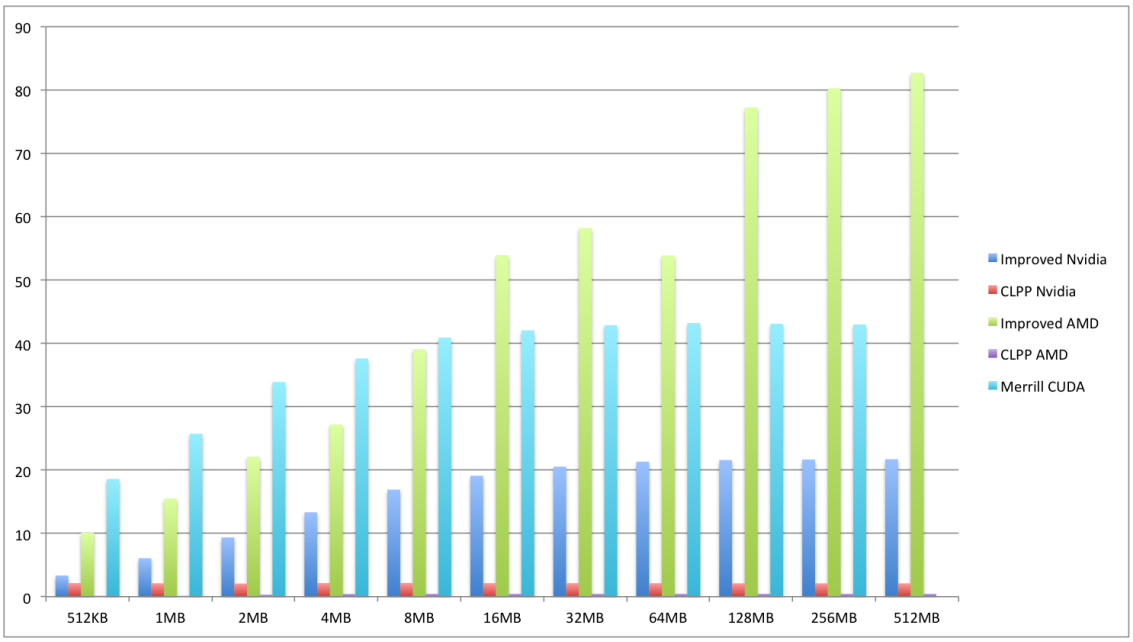

Figure 6: GPU Kernel Speedup

speedup compared to CLPP on both Nvidia and AMD GPU. For 512MB input data size, we achieve 21.7x speedup on Nvidia GeForce GTX 460 (compared to 2.1x speedup with CLPP) and 82.7x on AMD Tahiti (compared to 2.2x slowdown with CLPP). We notice that our implementation has better kernel performance on AMD than on Nvidia (about 3.8x faster for 512MB data size), while the CLPP implementation has better kernel performance on Nvidia than on AMD. Overall, our implementation beats the performance of CLPP on both Nvidia and AMD GPU. Compared to Merrill's CUDA implementation, our implementation has faster kernel time on AMD Tahiti (2x faster for 256MB data size), while slower on Nvidia GeForce GTX 460 (2x slower for 256MB data size). Merrill's CUDA implementation also has better kernel performance on handling small data sizes.

Next, we measure the GPU total time (including the data transfer time from CPU to GPU and from GPU back to CPU) and its speedup over the CPU time.

As we take into consideration the data transfer time, which accounts for around 86% of total time on Nvidia and 95% of total time on AMD for 512MB data size with our implementation, it amortizes the kernel time speedup. We can see from Figure 7 and Figure 8 (compared to Figure 5 and Figure 6), that Nvidia has better data transfer performance (3.3GB/s) than AMD (2GB/s). Figure 9 gives a straightforward comparision between the Nvidia and AMD data transfer time on different data sizes (measured on isolated program under identical system configurations). Nvidia is 2x faster than AMD transfering data from CPU to GPU (Host→Device), while about the same performance transfering data from GPU back to CPU (Device→Host, and AMD is even a little bit faster). The difference in data transfer time may due to the different ways that Nvidia and AMD handle data buffer allocation. Nvidia directly allocates and transfers data from CPU memory to GPU memory

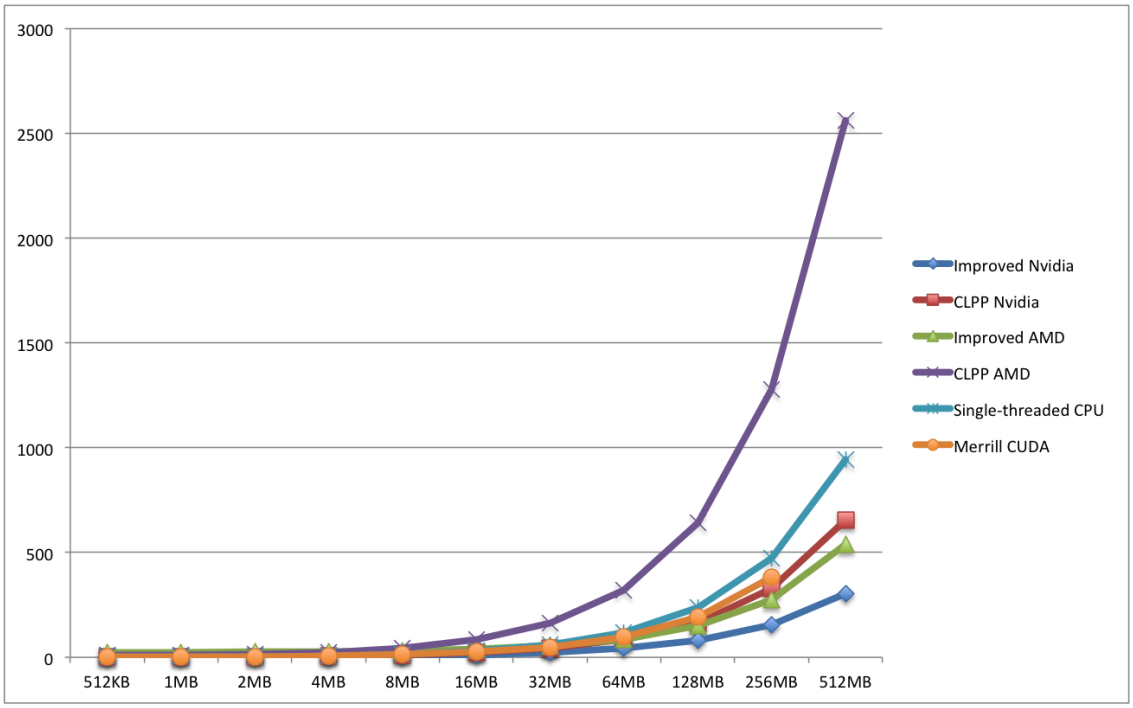

Figure 7: GPU Total Time

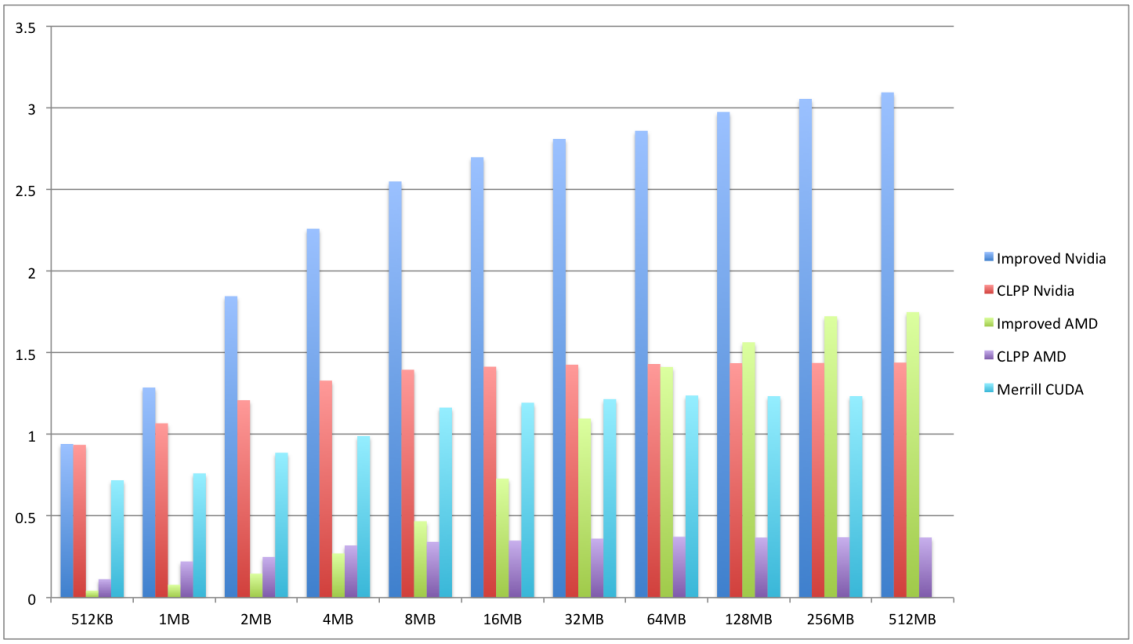

Figure 8: GPU Total Time Speedup

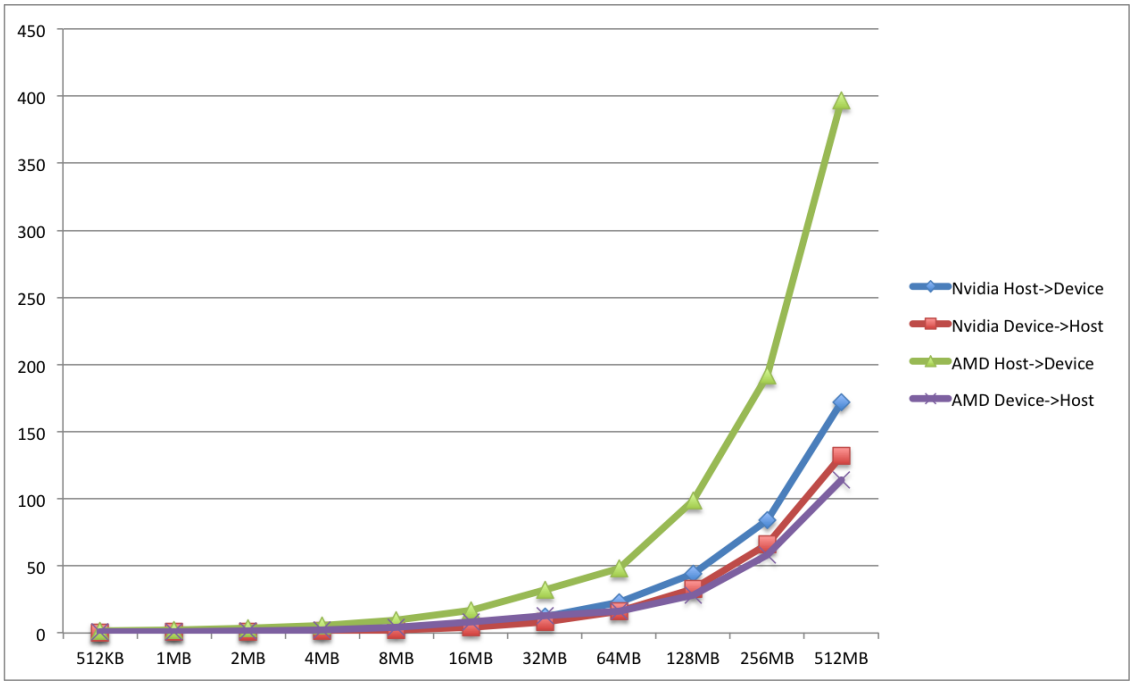

Figure 9: GPU Data Transfer Time

when clCreateBuffer is being called, while AMD does not decide whether the data will be used for GPU or not when calling clCreateBuffer, so it copies the data to another place in CPU memory temporarily, and waits until the kernel is called to do the actual data transfer from CPU to GPU. Thus, AMD indeed does two data transfers of the problem size in the process, which accounts for its 2x slower than Nvidia transfering data from CPU to GPU. On the other hand, there is no such "trick" going on when transfering data from GPU back to CPU, so they have about the same performance. Overall, on Nvidia GeForce GTX 460, our implementation achieves 3.1x speedup for 512MB data size (compared to 1.4x speedup with CLPP), and on AMD Tahiti, our implementation achieves 1.7x speedup for 512MB data size (compared to 2.7x slowdown with CLPP).

Merrill's CUDA implementation has slower total time compared to our implementation (for 256MB data size, its 1.2x total time speedup compared to ours 3x on Nvidia and 1.7x on AMD). However, recall from figure 5 and figure 6 that Merrill's implementation has better kernel time performance than ours. This can be explained by the different ways that we handle data input. Merrill's implementation allocates two data buffers of the input problem size, *source* buffer and *destination* buffer, where the kernel reads from the *source* buffer and writes to the *destination* buffer. On the contrary, our implementaion only allocates one data buffer, which is the *source* buffer, and the scan results are written back to the same *source* buffer. Allocating large-size buffers is an expansive operation on GPUs, so that is why Merrill's implementation has slower total time. This also explains why Merrill's implementaion cannot run 512MB data input, since it needs two buffers totaling 1GB global memory, which exceeds the memory limit on Nvidia GeForce GTX 460. In terms of Merrill's faster kernel time, our implementation does not incorporate all the kernel optimizations that

are in Merrill's implementation, i.e., avoiding shared memory bank conflict, as well as other possible compiler optimizations. The different ways that we handle global memory buffer can also be a reason, since reading and writing to the same memory space can potentially slow down the memory access.

4 Minimum Spanning Tree

Given a connected, undirected graph, in which each edge has an associated weight, a spanning tree of that graph is a subgraph that all the vertices are connected without cycles. Each spanning tree has a weight, which is the sum of the weight of all its edges. A minimum spanning tree is a spanning tree such that its weight is less than or equal to the weight of every other spanning tree. Minimum spanning tree has many applications in network designs, and it is also closely related to other problems such as travelling salesman problem, max flow/min cut, etc..

There are three common algorithms for finding a minimum spanning tree, Boruvka's algorithm [1], Prim's algorithm [10], and Kruskal's algorithm [11]. Prim's algorithm starts by picking an arbitrary vertex from the graph as the root of the tree, and keeps adding minimum-weight edge leaving the tree, until all vertices in the graph are in the tree. Kruskal's algorithm starts with all vertices in the graph, each as a separate tree, and then keeps removing the minmum-weight edge from the original set of edges and adding that edge to the forest if it is connecting two trees in the forest, until no more edge is left in the original set of edges or the forest of trees are all connected. Boruvka's algorithm works by recursively picking the minimum-weight edge leaving each vertex and combining all the connected vertices into a supervertex. Repeat this process until there is only one vertex left.

Algorithm 4 Boruvka G(V,E) Pseudocode

```
1: T = empty set
2: Add each v in V to T
3: while (T has more than one component) do
4:   lst = find-min-edge(C) for each component C
5:   remove-cycle(lst)
6:   merge-vertices(lst)
7:   add-edges-to-new-vertices
8: return T
```

Prior work of GPU implementation for minimum spanning tree includes CUDA implementation with Prims' algorithm [12] and CUDA implementation with Boruvka's algorithm [13], both using data primitives from the CUDPP and Thrust standard library. Our work here employs the efficient scan algorithm we develop using OpenCL (discussed in the above two sections), together with Boruvka's algorithm to implement a parallel minimum spanning tree algorithm in OpenCL.

Refer to Boruvka's algorithm in Algorithm 4, we can see that there are four steps in each reduction iteration (line 4-7). The find-min-edge operation returns the minimum-weight edge connected to each vertex. The remove-cycle operation then removes cycles in the min-weight edge list. (Note that a cycle can only exist within two vertices, i.e., A selects the edge connected to B and B selects the edge connected to A. It's impossible that a cycle will exist among three or more vertices, otherwise at least one of the edges in the cycle must not be the min-weight edge connected to a vertex.) Next, the merge-vertices operation picks one vertex from each forest of connected vertices as the supervertex, and build the new list of vertices from the supervertices. Finally, the add-edges-to-new-vertices operation adds all edges connecting to other supervertices to the corresponding supervertex edge list.

We find that the find-min-edge operation can take around 45% of total time in a dense graph, and around 30% of total time in a sparse graph. Thus, we focus on parallelizing the find-min-edge operation (line 4) in the algorithm. We employ the segmented scan primitive, a generalization of the prefix scan primitive discussed in section 2. Segmented scan computes prefix scan only within segments of the input sequence, where the start of each segment is marked by a head flag (i.e. 1). We use the same approach for prefix scan as described in section 2 for our segmented scan implementation, and take into account the flag array (represented using 1-byte boolean) that marks the start of each segment. Thus, we can scan through the input list of edges in parallel and find the min-weight edge for each vertex (segment).

5 MST Performance Analysis

We compare the performance of our segmented min scan against the sequential find-min-edge operation (on Intel Core i7 CPU) on both the Nvidia GeForce GTX 460 and AMD Tahiti GPU (based on average time of 50 consecutive runs). We compare both the total time (time starts after reading in graphs and transferring initial graph data to GPU, and ends after all iterations are finished when there is only one vertex left) speedup and the find-min-edge operation speedup for both dense and sparse graphs. Note that in measuring the segmented min scan time, we process the initial data transfer from CPU to GPU as part of the graph initiation (reading the input graph into vertex/edge list, etc.), so it is not counted towards the scan time nor the total time. The segmented min scan time starts before making the first kernel call and ends after transferring the data from GPU back to CPU. Also, where comparing dense graphs versus sparse graphs, we keep the number of edges the same (while number of vertices various), since the input data list size for find-min-edge operation depends on the number of edges. (Note: we do not find the implementaion code based on prior work of CUDA implementaion with Boruvka's algorithm [13], so we do not do the comparison here.)

We first compare the find-min-edge operation time and speedup. The x-axis represents the number of edges, and the y-axis is the time in milliseconds.

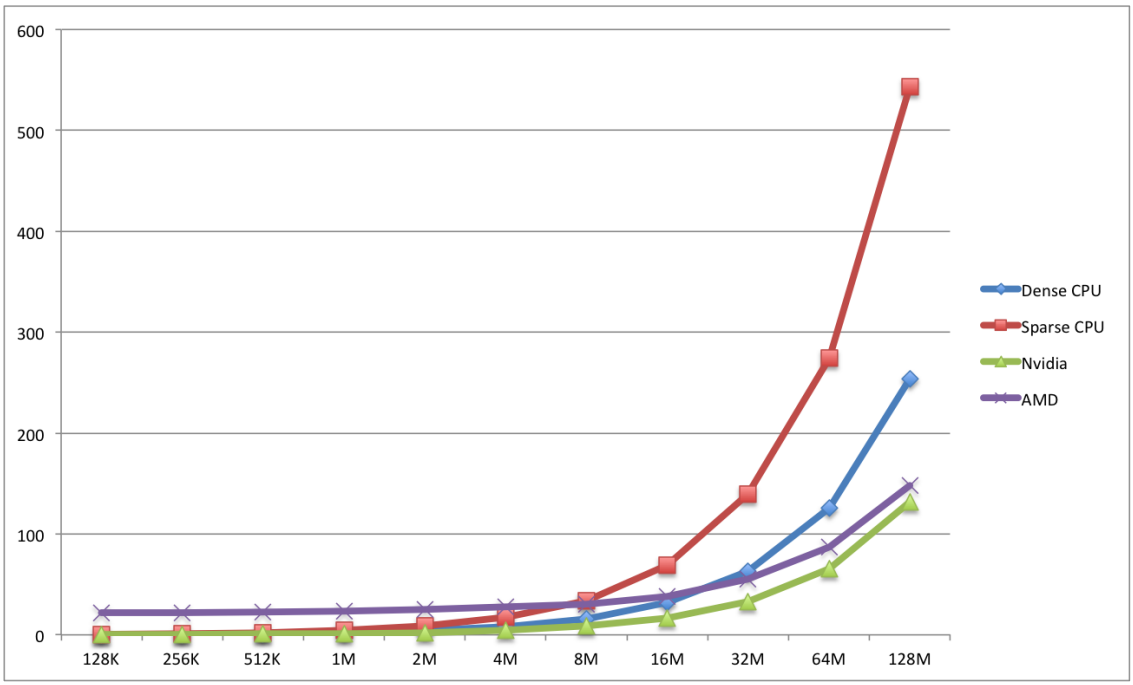

Figure 10: Find-Min-Edge Time

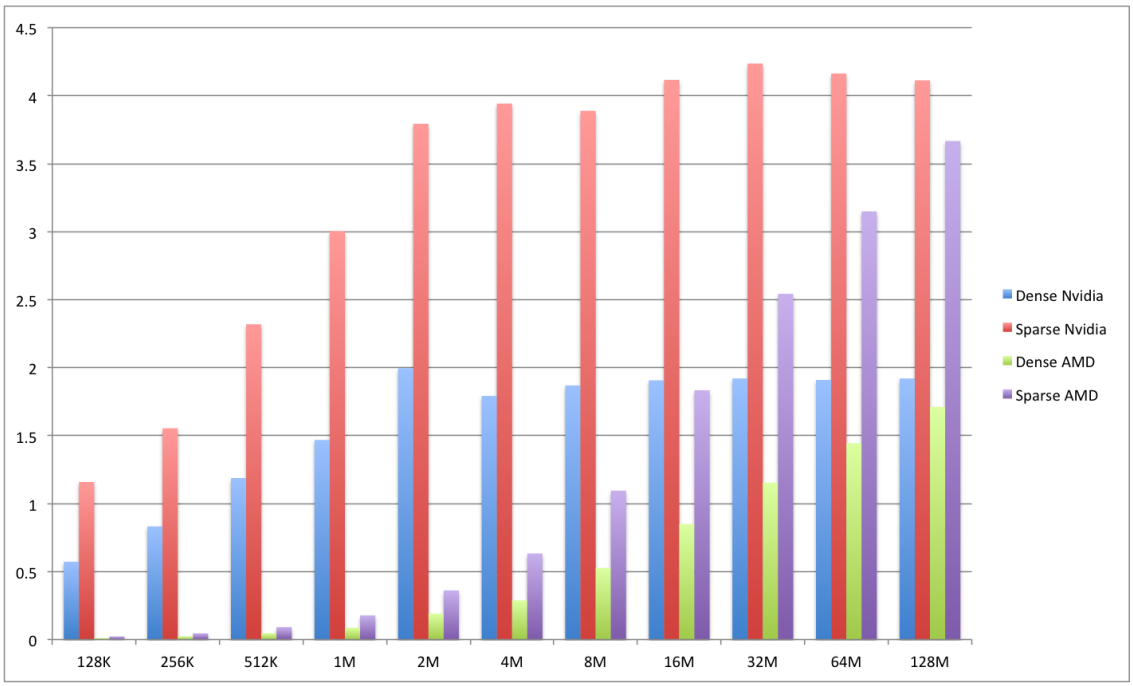

Figure 11: Find-Min-Edge Speedup

The GPU is pretty robust with both dense graphs and sparse graphs, while the sequential execution on CPU has better performance at dense graphs than sparse graphs (given the same total number of edges in the graph). Figure 11 gives the speedup of Nvidia and AMD on both dense and sparse graphs. On dense graphs (graph density, defined as $D = \frac{2|E|}{|V|(|V|-1)}$, close to 1), for 128M number of edges, our implementation achieves 1.92x speedup for find-min-edge operation on Nvidia GeForce GTX 460, and 1.71x speedup on AMD Tahiti. On sparse graphs (graph density close to 0, while the graph is connected), for 128M number of edges, our implementation achieves 4.11x speedup for find-min-edge operation on Nvidia GeForce GTX 460, and 3.67x speedup on AMD Tahiti.

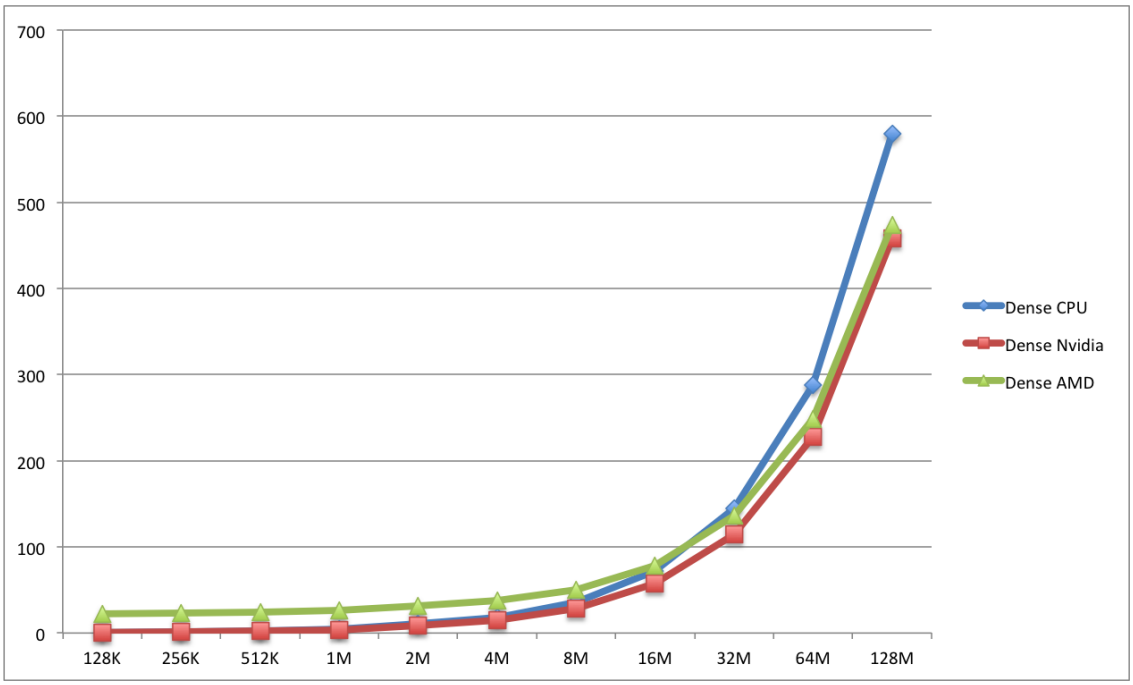

Figure 12: Total Time for Dense Graph

Figure 12, 13, 14 give the performance and speedup of the total MST time. On Nvidia Geforce GTX 460, our implementation achieves 1.26x total time speedup for dense graph with 128M edges, and 1.28x total time speedup for sparse graph with 128M edges. On AMD Tahiti, our implementation achieves 1.22x total time speedup for dense graph with 128M edges, and 1.27x total time speedup for sparse graph with 128M edges.

The difference between dense and sparse graphs reflected in the input data list of find-min-edge operation is the content of the flag list (marking the start of each segment). For sequential execution on CPU, if each segment is long and thus the number of segment is small, since it only needs to write the final reduction result at the end of each segment, so the total write accesses are small; on the other hand, if each segment is short and thus the number of segment is big, then it needs a lot more write accesses. While on GPU, instead of doing just reduction, we are using our prefix scan approach to compute the "prefix min" for each element position. So the GPU parallelization is actually doing more work than just

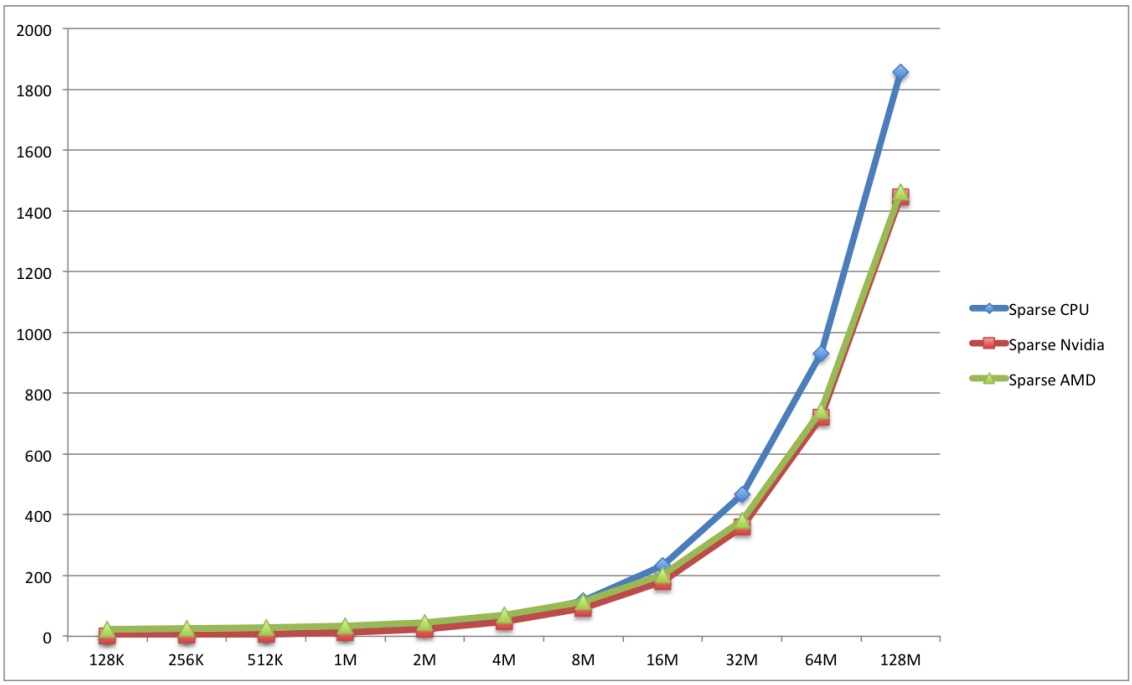

Figure 13: Total Time for Dense Graph

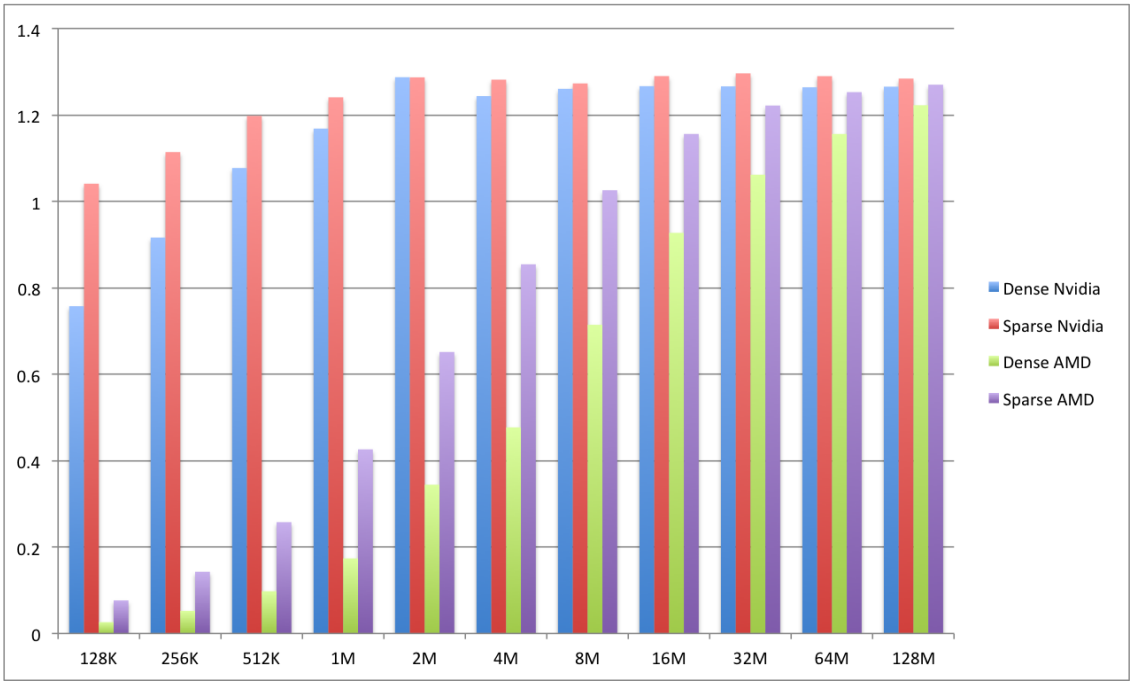

Figure 14: Total Time Speedup

the reduction (which only needs the reduction result at the end of each segment). Since the GPU is doing the same amount of work regardless of the number of 0s and 1s (representing the number of vertices), so its performance stays the same across dense and sparse graphs (given the same number of total edges).

6 Conclusions and Future Work

Even though GPU is designed and most suitable for problems involving minimal dependency across threads, efficient parallel algorithms can still be developed for high-dependency operations like prefix scan, and achieve nice speedup over sequential execution. Parallel data primitives like prefix scan can be useful building blocks for graph algorithms such as minimum spanning tree, which we implement and show its speedup over sequential execution.

As mentioned before in our minimum spanning tree implementation, we use a separate flag array of 1-byte boolean elements to represent the start of each vertex's list, but technically, the "flag" only needs 1 bit (0 or 1) and can be placed at the highest bit of each data element instead of being stored in a separate array. By this way, the global memory access time can be greatly reduced. CUDA supports functions like $\_\_ballot()$, $\_\_any()$, $\_\_all()$, $\_\_popc()$ among warp threads to perform fast bit operations. OpenCL does not support these functions yet. We may still implement these bit-operation functions as inline functions in OpenCL to further improve the performance. However, since these are hardware-specific functions, so different configurations will need to be made depending on the types of GPUs that we are running. Nvidia GPUs and AMD GPUs can have really different hardware behaviors.

Further more, since our implementation focuses on the optimization of find-min-edge operations, so there is more potential total time speedup if optimizing the other three operations (remove-cycle, merge-vertices, add-edges-to-new-vertices). I.e., the add-edges-to-new-vertices is another expensive operation besides find-min-edge (which also runs in $O(E)$ time, where E is the number of edges), and it can potentially be parallelized by scanning the edges and adding each edge to its corresponding vertex in parallel. In addition, it would also be helpful if Prim's algorithm [10] can be implemented with OpenCL as well, so we can then compare the GPU performances of these two different algorithms. More interestingly, GPU implementation can also be developed for a combination of Prim's and Boruvka's algorithm (i.e., use Boruvka's algorithm for the first $\log \log V$ steps, and then use Prim's algorithm for the rest), which only has $O(E \log \log V)$ time complexity in sequential execution (compared to Prim's and Boruvka's $O(E \log V)$ complexity). Thus, it would be interesting if

GPU performance comparisons can be done for Boruvka's algorithm, Prim's algorithm and combine-Prim-Boruvka's algorithm.

7 Acknowledgements

The authors would like to thank Michael Boyer for his help with OpenCL and Duane Merrill for providing his prefix scan implementation.

References

[1] Boruvka's algorithm, cited in Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 2009. "Introduction to Algorithm". *The MIT Press*, 3: 641.

[2] Merrill, D., Garland, M., and Grimshaw, A., 2011. "Scalable GPU graph traversal". *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 117-128.

[3] Merrill, D. and Grimshaw, A., 2009. "Parallel Scan for Stream Architectures". *Technical Report CS2009-14, Department of Computer Science, University of Virginia*, https://sites.google.com/site/duanemerrill/ScanTR2.pdf?attredirects=0.

[4] Zagha, M. and Blelloch, G.E., 1991. Radix sort for vector multiprocessors". *Proc. ACM/IEEE Conference on Supercomputing*, 712-721.

[5] Brent, R.P. and Kung, H.T., 1982. "A Regular Layout for Parallel Adders. Computers". *IEEE Transactions*, 31 (3): 260 -264.

[6] Sklansky, J., 1960. "Conditional-Sum Addition Logic". *IEEE Transactions on Electronic Computers*, 9 (2): 226-231.

[7] Kogge, P.M. and Stone, H.S., 1973. "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations". *IEEE Transactions on Computers*, 22 (8): 786-793.

[8] Thrust - Code at the speed of light: http://code.google.com/p/thrust/.

[9] CLPP - OpenCL Data Parallel Primitives Library: https://code.google.com/p/clpp/.

[10] Prim, R.C., 1957. "Shortest connection networks and some generalizations". *Bell System Technical Journal*, 36: 1389-1401.

[11] Kruskal, Joseph.B., 1956. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem". *Proceedings of the American Mathematical Society*, 7 (1): 48-50.

[12] Wang, W., Huang, Y. and Guo, S., 2011. "Design and Implementation of GPU-Based Prim's Algorithm". *I.J.Modern Education and Computer Science*, 4: 55-62.

[13] Vineet, V., Harish, P., Patidar, S. and Narayanan, P.J., 2009. "Fast Minimum Spanning Tree for Large Graphs on the GPU". *Proceedings of the Conference on High Performance Graphics*, 167-171.