A High-Performance MST Implementation for GPUs

Alex Fallin
Dept. of Computer Science
Texas State University
San Marcos, Texas, USA
waf13@txstate.edu

Andres Gonzalez
Dept. of Computer Science
Texas State University
San Marcos, Texas, USA
ag1548@txstate.edu

Jarim Seo
Dept. of Computer Science
Texas State University
San Marcos, Texas, USA
j_s1195@txstate.edu

Martin Burtscher
Dept. of Computer Science
Texas State University
San Marcos, Texas, USA
burtscher@txstate.edu

ABSTRACT

Finding a minimum spanning tree (MST) is a fundamental graph algorithm with applications in many fields. This paper presents ECL-MST, a fast MST implementation designed specifically for GPUs. ECL-MST is based on a parallelization approach that unifies Kruskal's and Borůvka's algorithm and incorporates new and existing optimizations from the literature, including implicit path compression and edge-centric operation. On two test systems, it outperforms leading GPU and CPU codes from the literature on all of our 17 input graphs from various domains. On a Titan V GPU, ECL-MST is, on average, 4.6 times faster than the next fastest code, and on an RTX 3080 Ti GPU, it is 4.5 times faster. On both systems, ECL-MST running on the GPU is roughly 30 times faster than the fastest parallel CPU code.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**.

KEYWORDS

Minimum spanning tree, minimum spanning forest, parallelism, performance optimization, GPU implementation

ACM Reference Format:
Alex Fallin, Andres Gonzalez, Jarim Seo, and Martin Burtscher. 2023. A High-Performance MST Implementation for GPUs. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3581784.3607093

1 INTRODUCTION

An MST of a weighted, undirected, connected graph $G = (V, E, w)$ is a subset of the edges E that connects all vertices V and that has the minimum possible total weight. MSTs have one fewer edge than there are vertices in the graph. A minimum spanning forest (MSF) is a generalization to graphs with multiple connected components. The MSF consists of a separate MST for each connected component.

For illustration, in Figure 1, assume electricity producers and consumers to be the vertices of the graph, power lines to be the edges, and the weights to be the cost of maintaining the power lines. In this example, the cheapest distribution grid that allows everyone to deliver or receive electricity is the MST shown.

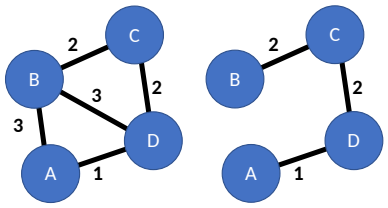

Figure 1: Example of a weighted graph on the left and the resulting MST on the right

Computing an MST (or MSF[1]) is a fundamental graph algorithm with applications in many fields. For instance, it is a key building block in network analysis [12], chip design [1], eye tracking [17], route planning [13], and medical diagnostics like tumor recognition [4]. Since some of these applications repeatedly generate an MST, increasing the performance of this step is important and has the potential to speed up lifesaving computations.

There are three classic MST algorithms. Borůvka's algorithm [11] iteratively merges the vertices of the graph along their lightest edge until just one vertex remains. The merged edges form the MST. Prim's algorithm [28] builds the tree from an arbitrary starting vertex one edge at a time by adding the lightest edge that connects a vertex in the tree to a vertex that is not yet in the tree. Kruskal's algorithm [18] sorts the edges by weight. It then processes them in non-decreasing order and inserts the edges that do not create a cycle. These three algorithms are greedy and have a time complexity of $O(|E| \log |V|)$, i.e., they run in polynomial time.

Borůvka's algorithm is the most parallelism friendly of the three because, in each step, every vertex can be processed independently and all lightest edges can be added concurrently. The main issues are that the parallelism decreases rapidly and that building a new, smaller graph in each iteration is expensive. Prim's algorithm relies on the cut property of trees and is, therefore, inherently serial. However, some of its auxiliary operations can be parallelized to improve performance [33]. Similarly, Kruskal's algorithm also appears to be inherently serial but can be sped up by parallelizing some of its operations like the sorting step or enhancements like filtering [25] and partial sorting [3].

Our approach, which we call ECL-MST and describe in detail in Section 3, fully parallelizes Kruskal's algorithm and incorporates key performance optimizations. The resulting CUDA implementation is several times faster than prior CPU and GPU codes on a wide variety of real-world and synthetic inputs.

[1]Unless otherwise noted, we use MST to refer to both MSTs and MSFs.

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11…$15.00
https://doi.org/10.1145/3581784.3607093

This paper makes the following main contributions.

- We introduce ECL-MST, a high-speed MST implementation written in CUDA.
- We present key domain-specific optimizations that are essential to the GPU performance of ECL-MST.
- We show that ECL-MST outperforms state-of-the-art CPU and GPU implementations on many inputs.
- We demonstrate that Kruskal's and Borůvka's MST algorithms converge to the same parallelization.

The latest version of our ECL-MST CUDA code is available in open source through GitHub [5] and on the web [6].

The rest of this paper is organized as follows. Section 2 summarizes related work. Section 3 explains our approach in detail. Section 4 describes the evaluation methodology. Section 5 presents and discusses the results. Section 6 concludes the paper with a summary and future work.

2 RELATED WORK

Being a mature domain, a large amount of related work exists on the minimum spanning tree problem. Hence, we primarily focus on the work we build upon, some of the previous parallelizations of MST, and the implementations we compare to in the result section.

The previous section already introduced the classic (serial) MST algorithms. Brennan [3] combines Kruskal's algorithm with quicksort (*qKruskal*). By partitioning the edge list into a lighter and a heavier part, sorting the light part, and only sorting the heavy part if the tree is not complete after processing the light part, the tree can often be built without fully sorting the edge list. This increases the performance over a version that always fully sorts the edge list. ECL-MST similarly does not process the entire edge list at once to potentially save on computation. However, ECL-MST differs in that it does not perform the sorting step at all.

Osipov et al. [25] demonstrate that, in scenarios where the heaviest edge will be in the MST, qKruskal does not perform better than the conventional implementation. They take advantage of the partially built tree and the fact that edges can be checked for cycles faster than they can be sorted and create *Filter-Kruskal*. By removing edges that create a cycle from the unsorted chunks, they reduce the number of edges that must be sorted. They show that this optimization allows Kruskal's algorithm to perform well on both sparse and dense graphs. ECL-MST incorporates this idea.

Setia et al. [31] devised a parallel MST algorithm for CPUs that is based on Prim's algorithm. It employs worker threads that start at a different random vertex and build a tree from that vertex outward. When the threads collide, the thread with the higher ID is killed and its tree is merged with that of the thread with the lower ID. The algorithm takes advantage of the cut property to merge the trees correctly. Their code makes use of critical sections to perform the tree merging. In contrast, ECL-MST is lock-free and uses atomic operations to avoid more expensive synchronization.

Pai and Pingali [26] introduce three throughput optimizations that are crucial to high performance in graph algorithms like MST. Moreover, they discuss how these optimizations can be automated by compilers that generate CUDA code to improve the performance of many algorithms. One important bottleneck they identify is kernel launch overheads, especially kernel launches inside a *while* loop with a condition that is the result of a cudaMemcpy. This is particularly prescient because ECL-MST as well as the Vasconcellos et al. [8] and the Sousa et al. [21] MST codes we compare against in Section 5 employ this execution pattern. Pai and Pingali note that, for algorithms using a worklist, programmers must be careful to avoid bottlenecking by returning to the CPU too often. Fortunately, this is not a problem in ECL-MST, which is guaranteed to perform no more than $O(\log |V|)$ iterations. They also discuss traversing graphs that have a large difference between the degree of individual vertices, e.g., power-law graphs. They suggest that sticking to a single level of parallelization, such as thread granularity, can cause performance bottlenecks. As a remedy, we use a hybrid parallelization technique in ECL-MST that is similar to Merrill's approach [22].

Vasconcellos et al. [8] introduce a pure MST code for GPUs. By this, we mean that they target graphs with just a single connected component. They base their implementation on Borůvka's algorithm with a modified stopping condition to generate an MST instead of an MSF. They employ a vertex-centric, data-driven [23] algorithm that uses a kernel to find the lightest edge of a vertex and another kernel to mark it. They then contract the graph and recalculate the connected components. The main goal of their work is to produce an efficient general MST algorithm, which is why they do not use many CUDA specifics other than atomic operations. We compare to their GPU implementation in Section 5 as *Jucele GPU*.

Lonestar [20] contains two implementations of Borůvka's algorithm. The GPU version, which we do not compare to as it appears to be incompatible with recent GPU architectures, is based on the IrGL compiler by Pai and Pingali [26]. It performs indirect edge relaxation so as not to modify the graph dynamically. This is akin to how ECL-MST performs its component merging. The CPU version of Lonestar, which we do compare to in the result section, runs over the set of disconnected components and loops over their edges. The first part of the main loop determines the lightest edge of each component, which is safe to do in parallel because this step is read-only. The second part of the main loop merges the components in a lock-free manner. Both of these MST implementations share the use of the disjoint-set data structure with ECL-MST to avoid modifying the graph at runtime, which would be slow.

The MST code in the Problem-Based Benchmark Suite (PBBS) [2] implements elements from both qKruskal and Filter-Kruskal along with new ideas, in particular deterministic reservations. It executes the iterations of the original non-parallel algorithm out of order using a custom speculative *for* loop. PBBS only performs updates to the tree once they have been found to be non-conflicting with earlier iterations. The algorithm only sorts the smallest k edges at first, where $k = min(|V|, 5|E|/4)$ is approximated using an edge sample of size $|E|/sqrt(|E|)$. This k smallest chunk is then processed. If the MST is not complete, PBBS filters all cyclic edges out of the remaining chunk before processing it. ECL-MST shares the filtering optimization with PBBS. Unlike PBBS, it does not sort the chunks and uses a simpler edge sampling method with a different threshold. ECL-MST employs a similar parallelization strategy based on deterministic reservations but without speculation.

RAPIDS cuGraph [29] is an industry suite of graph algorithms for GPUs. It takes advantage of RAFT (Reusable Accelerated Functions and Tools) [30] in its MST code. Moreover, it uses color propagation

and supervertices to implement Borůvka's algorithm. We compare to this vertex-centric, topology-driven [23] code in Section 5.

Sousa et al.'s [21] MST code for GPUs is listed by Pai and Pingali [26] as the fastest existing implementation. This code is based on Borůvka's algorithm and is vertex-centric and data-driven. It starts by finding the minimum weighted edge of each vertex and stores them in a shared data structure. It then removes the mirrored edges, i.e., the edges where both the outgoing and incoming edges between the same two vertices are selected. The code is a true implementation of Borůvka's algorithm in that it actually merges vertices (using color propagation) into new supervertices. Finally, it builds a new edge array for the contracted graph based on the merged vertices. ECL-MST differs from this implementation in that it does not create any new graphs and is primarily edge-centric. We compare to Sousa et al.'s code in Section 5 as *UMinho GPU*.

Gunrock [24] is a graph analytics suite that includes a vertex-centric topology-driven MST implementation. It relies on the input having only a single connected component and, therefore, cannot generate an MSF. It checks all vertices and evaluates an edge if its source and destination do not belong to the same connected component. We also compare to this code in the results section.

All of the GPU-related works described above employ a vertex-centric implementation. In contrast, our ECL-MST code is edge-centric, which we found to be substantially faster because it enables important optimizations such as implicit path compression (see Section 5.3). Additionally, some of the above approaches are topology-driven, which tends to be slower than the data-driven implementation that ECL-MST and some other codes use.

3 ECL-MST APPROACH

Since Borůvka's algorithm is the most amenable to parllelization of the three classic MST algorithms, many parallel MST codes are based on this algorithm [8, 20, 21, 29]. In contrast, we set out to parallelize Kruskal's algorithm for GPU execution. As we progressively increased the parallelism, something unexpected happened. Our GPU parallelization of Kruskal's algorithm converged to that of Borůvka's algorithm. We then proceeded to add critical performance optimizations to this combined parallel MST approach, paying close attention to circumvent the weaknesses of GPUs and trying to exploit their strengths.

3.1 Parallelization

To avoid the cost of creating a new, smaller graph in each iteration, Borůvka's algorithm can be parallelized as follows using disjoint sets to efficiently represent the merged vertices in the original graph. Initially, all vertices form their own set. Then, the following steps repeat until only a single set (per connected component) remains.

- Each vertex determines the set it belongs to using the *find* operation. This yields a unique representative for each set (e.g., the vertex with the highest ID in the set).
- Each vertex identifies the lightest adjacent vertex that belongs to a different set. This lightest neighbor is recorded in the representative if it is lighter than the lightest vertex already recorded there.
- Each representative merges its set with that of the lightest recorded neighbor using the *union* operation.

This parallelization of Borůvka's algorithm is straightforward but suffers from exponentially decreasing parallelism as the number of sets is roughly halved in each iteration.

To see whether we can avoid or at least alleviate this shortcoming, we attempted to fully parallelize Kruskal's algorithm. We also use a disjoint-set data structure but primarily for cycle detection. First, we parallelized the sorting of the edges by weight, which is straightforward [16]. However, the building of the MST by processing the edges from lightest to heaviest remained serial. We gradually parallelized this step in the following way by processing a chunk of the k lightest edges concurrently.

- Each edge in the chunk determines whether it forms a cycle by comparing the representatives of its two endpoints. If it does (i.e., the representatives match), the edge is discarded.
- Each remaining edge records in the representatives of its endpoints whether it is the first edge in the chunk to connect to that set. This is done by recording the relative position of the edge within the chunk (i.e., the edge index), but only if it is smaller than the smallest index already recorded.
- Each edge checks whether its edge index is stored in at least one of the two representatives. If it is, the edge is included in the MST and the sets of its endpoints are merged. Otherwise, nothing is done with this edge.

The above steps repeat until each edge in the chunk has been either discarded or included in the MST. Then, the algorithm advances to the next chunk of k lightest edges. It continues in this manner until the MST is complete.

This parallelization approach is based on the observation that we can safely include not only the lightest edge but all edges that are the first in the chunk to connect to a set. Deterministic reservation is based on the same observation [2]. This is the case because adding those edges cannot form a cycle. We found this approach to work quite well as it often either includes many edges in the MST or discards many edges from consideration in each iteration. Hence, it typically does not require many iterations per chunk.

Next, we added two critical optimizations. First, we realized that, since the edges are sorted by weight, a lower edge index is tantamount to a lower edge weight. Therefore, we can record the lightest-weight edge rather than the smallest-index edge in the representatives without affecting the outcome. Second, we realized that, since we record the lightest edge in the representative (using an atomicMin operation), it is no longer necessary for the edges in the chunk to be sorted. After all, the minimum of the sorted and unsorted list is the same. By combining these two optimizations and making the chunk size large enough to encompass all edges, we were able to eliminate the sorting step altogether.

Interestingly, the resulting parallel MST algorithm is nearly identical to the parallelization of Borůvka's algorithm outlined above. The only remaining difference is that Kruskal's code skips the edges that form cycles and processes the remaining edges whereas Borůvka's code processes the edges that connect disjoint sets and skips the remaining edges. However, this is merely a distinction in viewpoint as there is no actual difference in the codes, both of which use the exact same disjoint-set data structure and operations for processing the edges. Thus, the two parallelizations have converged to the same solution.

The resulting parallelization is identical to that of PBBS, which is based on deterministic reservations [2]. In other words, our parallelization of Kruskal's algorithm has led to an already known parallelization strategy for MSTs on CPUs. However, we believe we are the first to employ it on GPUs for accelerating MST computations and propose important optimizations to speed it up.

3.2 Performance Optimization

We implemented this parallel MST algorithm in CUDA and incorporated the following main optimizations to boost the performance.

- Our code is lock free, which is crucial for good performance on GPUs. It employs atomicAdd instructions to obtain the next available slot in the worklist, atomicCAS instructions to perform the union operation on the disjoint sets, and atomicMin instructions to determine the lightest edge of each set (i.e., to perform the deterministic reservations). The atomicMin instructions operate on 64-bit values that hold the edge weight in the most significant bits and the edge ID in the least significant bits. This is done for two reasons: (1) It introduces a deterministic tie breaker in case multiple edges have the same weight, and (2) it provides the necessary edge ID to identify the lightest edge.
- The code incorporates a hybrid parallelization scheme, which is important for graph algorithms where some vertices may have many more neighbors than others. In particular, the code processes each low-degree vertex ($d(v) < 4$) with a single thread and each remaining vertex with an entire warp. In the latter case, the processing of the vertex's neighbors is parallelized across the warp-threads and exploits the CUDA ballot and shuffle functions to quickly exchange information between the warp-threads. Hence, our code processes the vertices in parallel and, for higher-degree vertices, also processes the neighbors in parallel.
- We investigated different path-compression schemes to speed up the *find* operations, including "intermediate pointer jumping" that is optimized for GPU execution [14]. Interestingly, we obtained the best performance by not including any explicit path compression. Instead, our code implicitly performs path compression when populating the new worklist (see below) by storing the result of the *find* operations in lieu of the original vertex IDs. This approach simplifies and accelerates the code.
- Prior work has demonstrated that filtering can speed up MST computations [25]. Filtering works by sporadically looking for and removing edges from consideration that form cycles. This helps because the cycle checks are computationally cheap. Since there is still overhead associated with filtering, we perform only a single filtering step. Specifically, we randomly sample 20 edge weights to estimate the largest weight w of the $c|V|$ lightest edges in the graph. Then, we process the edges with a weight under w in the first phase, filter the remaining edges, and process the edges that were not filtered out, if any, in the second phase. Values between 2 and 4 seem to work well for c, which are likely to include most of the the MST edges in the first phase since an MST has $|V| - 1$ edges. We use $c = 4$ in our code, i.e., no filtering occurs for graphs with an average degree below 4.
- Our code also includes several small optimizations. For example, it alternates between two worklists for holding the edges that still need processing. In each iteration, one of the worklists is drained while the other is filled. Between iterations, the two pointers to the worklists are swapped. Furthermore, our code utilizes edge-centric processing in several kernels to improve load balancing. Another small optimization is to only process edges in only one direction. The code operates on graphs stored in the widely-used CSR format, in which each undirected edge is represented by two directed edges. However, these pairs of edges always form a cycle, so one of the two edges can safely be skipped.

Some of the above optimizations are new (e.g., the implicit path compression), have not been used in GPU MST codes before (e.g., the hybrid parallelization scheme and the deterministic reservations), or have been implemented in a new way to boost their efficiency (e.g., the filtering). In Section 5.3, we study the performance impact of these optimizations and show that, taken together, they improve the speed of ECL-MST by a factor of eight on average, demonstrating the importance of these optimizations.

3.3 ECL-MST Algorithm

The pseudo code in Algs. 1 and 2, where the colon means concatenation, illustrates how ECL-MST works for graphs with an average degree below 4, i.e., without filtering (meaning all weights meet the threshold condition on Line 6). For graphs with an average degree of 4 or greater, the filtering threshold is computed as described above, Algs. 1 and 2 are executed, then the threshold condition is inverted, and Alg. 2 is executed one more time, using $set(v)$ for v and $set(n)$ for n in the loop starting on Line 2.

Every *for all* loop in both algorithms is parallel, with the loop on Line 3 in Alg. 2 being parallelized across the warp-threads if it performs at least 4 iterations. The *union* operation on Line 30 involves an atomicCAS, and the worklist updates on Lines 7 and 18 involve an atomicAdd. The implicit path compression happens on Line 18, where the edge's source and destination vertices are replaced by the corresponding representatives (i.e., sets) when putting the edge on the worklist. Note that the worklists $WL1$ and $WL2$ hold edges represented by 4-tuples using the format:
⟨ source vertex, destination vertex, edge weight, edge ID ⟩.

Running our algorithm on the graph from Fig. 2 does the following. First, it assigns every vertex to its own set, clears the minimum edge information, and marks all edges as not belonging to the MST. Next, the worklist is populated with a copy of each edge in the graph. For example, one entry will be the 4-tuple $\langle B, D, 3, c\rangle$, meaning the edge is between vertices B and D, has a weight of 3, and is labeled c. Then, the minimum edge is recorded in each set (vertex). The result is depicted in the left panel of Fig. 2. Finally, the algorithm checks, for each edge on the worklist, whether it is recorded in either of its endpoints (vertices). The two edges that are recorded (1e and 2a) are included in the MST, indicated in green in the middle of Fig. 2, and the corresponding sets are joined, meaning A and D now form a set that is represented by A, and B and C form a set that is represented by B. Then the minimum edge information is cleared

and the next iteration starts. It populates a new worklist with the three remaining edges. However, it uses the set IDs rather than the vertex IDs. For instance, the edge mentioned above is now stored as $\langle B, A, 3, c\rangle$ because A is the representative of the set holding D (and B is the representative of the set holding B). Also, it records new minimum edge information in the two set representatives (2b). Upon checking which edges have been recorded, only one edge remains. This edge is added to the MST and the two sets are joined. At this point, the algorithm stops because no edges remain that span two sets. The resulting MST is shown on the right in Fig. 2.

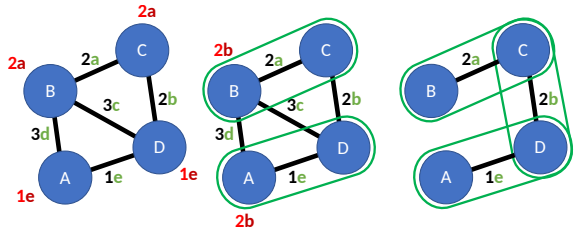

Figure 2: Minimum edge labels (weight and ID pairs) recorded in each set during the first and second iteration of ECL-MST

Algorithm 1 ECL-MST Initialization

Input: Weighted graph $G = (V, E, w)$ in CSR format

▷ initialize vertex info

for all vertices $v \in V$ **do**
  $minEdge_v \leftarrow \infty$
  $set(v) \leftarrow \{v\}$
end for

▷ initialize edge info

for all edges $e \in E$ **do**
  $MST_e \leftarrow false$
end for

Output: Initialized graph G

4 EVALUATION METHODOLOGY

We compare the performance of ECL-MST to the 4 GPU-parallel, 3 CPU-parallel, and 1 CPU-serial MST implementations listed in Table 1. These are the fastest MST codes from the literature we could find. All tested GPU codes, including our own, support graph sizes of up to about 2 billion vertices and edges since larger graphs do not fit in the main memory of most GPUs. RAPIDS cuGraph is working on supporting larger graphs.

We evaluated the codes on two systems. System 1 is based on an AMD Ryzen Threadripper 2950X CPU with 16 cores. Hyperthreading is enabled, i.e., the 16 cores can simultaneously run 32 threads. The main memory has a capacity of 64 GB. The operating system is Fedora 34. The GPU in the system is an NVIDIA Titan V (Volta architecture) with 5120 processing elements distributed over 80 multiprocessors. Its global memory has a capacity of 12 GB. The GPU driver version is 515.43. System 2 is based on two Intel Xeon Gold 6226R CPUs with 16 cores each. Hyperthreading is also enabled, meaning the 32 cores can run 64 simultaneous threads.

Algorithm 2 ECL-MST Algorithm

Input: Initialized graph G from Algorithm 1

▷ populate worklist

$WL1 \leftarrow \emptyset$
for all vertices $v \in V$ **do**
  for all vertices $n \in adjList(v)$ **do** ▷ thread or warp
    if $v < n$ **then** ▷ only process in one direction
      Let e be the edge from v to n
      if e_w meets threshold condition **then**
        $WL1 \leftarrow WL1 \cup \langle v, n, e_w, e_{id}\rangle$
      end if
    end if
  end for
end for

▷ iterate until worklist is empty

while $WL1 \neq \emptyset$ **do**

▷ populate second worklist

  $WL2 \leftarrow \emptyset$
  for all $\langle v, n, e_w, e_{id}\rangle \in WL1$ **do**
    $p \leftarrow set(v)$ ▷ find operation
    $q \leftarrow set(n)$ ▷ find operation
    if $p \neq q$ **then**
      $WL2 \leftarrow WL2 \cup \langle p, q, e_w, e_{id}\rangle$ ▷ impl. path compr.
      $val \leftarrow e_w : e_{id}$
      atomicMin($minEdge_p$, val)
      atomicMin($minEdge_q$, val)
    end if
  end for

▷ swap worklists

  $WL1 \leftarrow \emptyset$
  swap $WL1$ and $WL2$
  if $WL1 \neq \emptyset$ **then**

▷ check if edge belongs to MST

    for all $\langle v, n, e_w, e_{id}\rangle \in WL1$ **do**
      $val \leftarrow e_w : e_{id}$
      if $(val = minEdge_v) \,||\, (val = minEdge_n)$ **then**
        join $set(v)$ with $set(n)$ ▷ union operation
        $MST_{id} \leftarrow true$ ▷ edge is in MST
      end if
    end for

▷ reset minEdge info

    for all $\langle v, n, e_w, e_{id}\rangle \in WL1$ **do**
      $minEdge_v \leftarrow \infty$
      $minEdge_n \leftarrow \infty$
    end for
  end if
end while

Output: Minimum spanning tree/forest MST

The main memory has a capacity of 128 GB. The operating system is Fedora 36. The GPU in this system is an NVIDIA RTX 3080 Ti (Ampere architecture) with 10,240 processing elements distributed over 80 multiprocessors. It has 12 GB of global memory. The GPU driver version is 525.60.

Table 1: Third-party MST codes used in experiments

Name	Source
Gunrock GPU	[34]
Jucele GPU	[15]
RAPIDS cuGraph GPU	[29]
UMinho GPU	[32]
Lonestar CPU	[20]
PBBS CPU	[27]
UMinho CPU	[32]
PBBS Serial	[27]

Table 2: Information about the used input graphs

Graph Name	Edges	Vertices	Type	CCs	d-avg	d-max
2d-2e20.sym	4,190,208	1,048,576	grid	1	4.0	4
amazon0601	4,886,816	403,394	co-purchases	7	12.1	2,752
as-skitter	22,190,596	1,696,415	Internet topo.	756	13.1	35,455
citationCiteseer	2,313,294	268,495	publication cit.	1	8.6	1,318
cit-Patents	33,037,894	3,774,768	patent cit.	3,627	8.8	793
coPapersDBLP	30,491,458	540,486	publication cit.	1	56.4	3,299
delaunay_n24	100,663,202	16,777,216	triangulation	1	6.0	26
europe_osm	108,109,320	50,912,018	road map	1	2.1	13
in-2004	27,182,946	1,382,908	web links	134	19.7	21,869
internet	387,240	124,651	Internet topo.	1	3.1	151
kron_g500-logn21	182,081,864	2,097,152	Kronecker	553,159	86.8	213,904
r4-2e23.sym	67,108,846	8,388,608	random	1	8.0	26
rmat16.sym	967,866	65,536	RMAT	3,900	14.8	569
rmat22.sym	65,660,814	4,194,304	RMAT	428,640	15.7	3,687
soc-LiveJournal1	85,702,474	4,847,571	community	1,876	17.7	20,333
USA-road-d.NY	730,100	264,346	road map	1	2.8	8
USA-road-d.USA	57,708,624	23,947,347	road map	1	2.4	9

We compiled the CPU codes with gcc/g++ 11.3.1 on System 1 and 12.2.1 on System 2 using the "-O3 -march=native" flags. For the CPU-parallel codes, when not selected automatically by the implementation, we set the thread count to match the number of cores since we found hyperthreading to hurt performance. We compiled the GPU codes with nvcc 11.7 using the "-O3 -arch=sm_70" flags on System 1 and with nvcc 12.0 using the "-O3 -arch=sm_86" flags on System 2.

We used the 17 graphs shown in Table 2 as inputs. Where needed, we modified the graphs to eliminate self-loops and multiple edges between the same two vertices. We added any missing back edges to make the graphs undirected[2]. For unweighted graphs, we inserted random weights so the MST can be computed. Table 2 lists the name, edge count, vertex count, type, and number of connected components of each graph along with their average and maximum degree. The graphs were obtained from the Center for Discrete Mathematics and Theoretical Computer Science at the University of Rome (DIMACS) [9], the Galois framework (Galois) [10], the Stanford Network Analysis Platform (SNAP) [19], and the SuiteSparse Matrix Collection (SSMC) [7]. We selected these graphs because they cover a wide range of types and sizes.

For all tested codes, we measured the runtime of the MST computation, excluding the time it takes to read in the graphs. In the GPU codes, we also exclude the time it takes to transfer the graph to the GPU or to transfer the result back (unless otherwise noted). For incompatible inputs, we report "not connected" (NC) if they contain multiple connected components.

We repeated each experiment 9 times for all codes and inputs and report the median computation time. The ECL-MST implementation verifies the solution at the end of each run by comparing it to the solution of a serial implementation of Kruskal's algorithm. This verification time is not included in the measured runtime.

RAPIDS cuGraph includes two MST versions, using either single- or double-precision edge weights. A large portion of our inputs are only compatible with the double-precision code due to their large size and resultant total MST weight. Hence, we present cuGraph results from the double version but also discuss the float results. Additionally, cuGraph is incompatible with System 1, so we only compare to it on System 2.

5 RESULTS

In this section, we first present the absolute runtimes of the various MST codes on our input graphs. Then, we show the resulting throughputs in millions of edges per second. Last, we evaluate the performance impact of some of our code optimizations.

5.1 Runtime

Tables 3 and 4 list the MST computation times in seconds on Systems 1 and 2, respectively. Lower runtimes are better. Rows labeled "MSF GeoMean" show the geometric mean over all measured inputs whereas rows labeled "MST GeoMean" list the geometric mean over only the inputs that consist of a single connected component. This is done to make the comparison with the Jucele and Gunrock GPU codes fair, which can compute MSTs but not MSFs.

For each input, we provide two ECL-MST results, one that does not include the memory-copy time for transferring the graph to the GPU or the result to the CPU and another that includes this time. The former, which is our baseline, is relevant in settings where an MST is computed as part of a larger data analytics pipeline where the graph is already on the GPU from a previous processing step and the resulting MST is needed on the GPU for a later step. The latter is relevant in settings where only the MST computation is performed on the GPU and the other steps on the CPU.

On System 1 (Table 3), ECL-MST is faster than the other codes on every tested input. Its running-time variance between the 9 repetitions of each experiment is only 0.0005% on average, and the maximum we observed is 0.0028% on as-skitter. Based on the geometric-mean performance over the MSF inputs, our code is 138 times faster than serial, 32.3 times faster than the fastest CPU-parallel code (PBBS), and 38.6 times faster than the UMinho GPU code. On just the MST inputs, our code is 184.4 times faster than serial, 39.2 times faster than the fastest CPU-parallel code (UMinho), and 4.6 times faster than the fastest GPU code (Jucele). ECL-MST does particularly well on the scale-free inputs, where it outperforms the other codes by at least 19 times on amazon0601, rmat16.sym, and soc-LiveJournal1. Such graphs have some vertices with a much higher degree than the other vertices, which tends to cause load-balancing issues in vertex-centric implementations. ECL-MST avoids this problem through hybrid parallelization in one kernel (where whole warps share the processing of high-degree vertices) and edge-centric processing in the other kernels.

[2]Since the graphs are stored in CSR format, each undirected edge is represented by two directed edges.

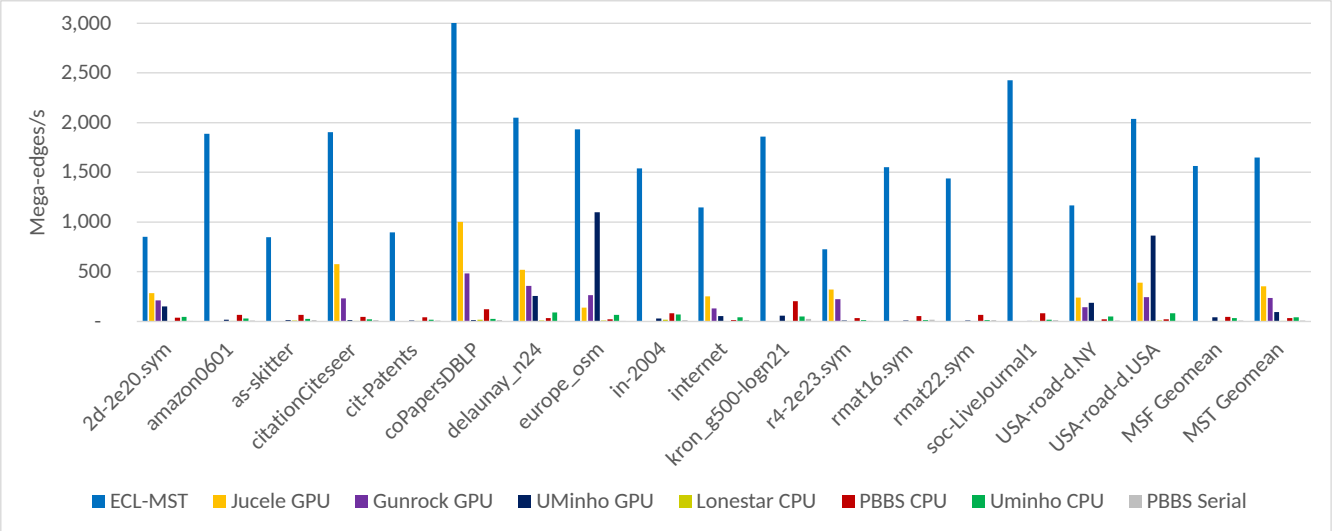

Figure 3: System 1 throughput results in millions of edges per second (the ECL-MST bar for coPapersDBLP extends to 7,091)

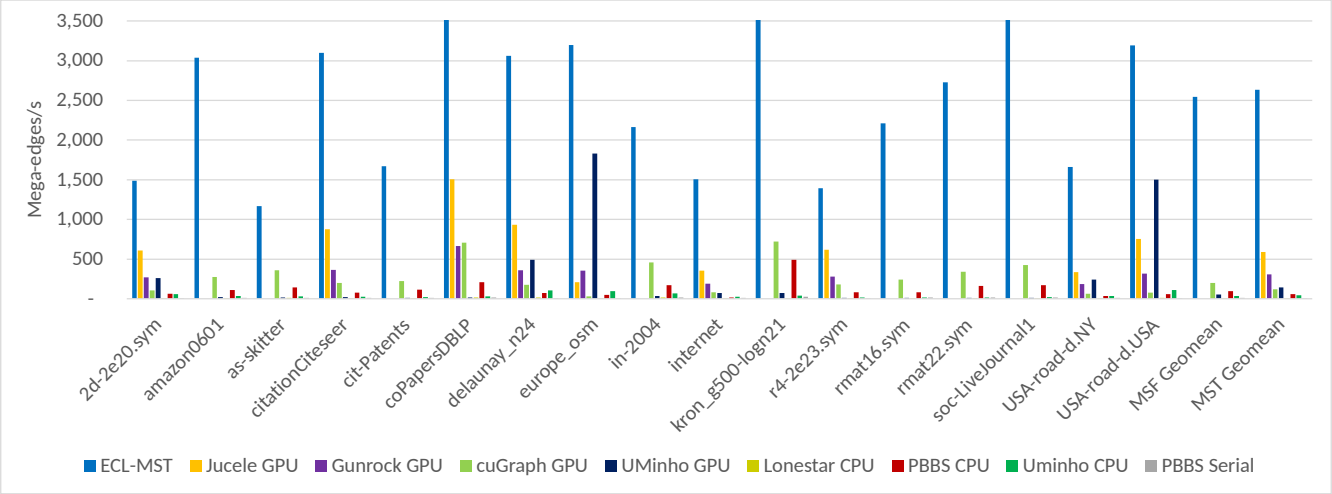

Figure 4: System 2 throughput results in millions of edges per second (the ECL-MST bar for coPapersDBLP extends to 12,241 and soc-LiveJournal1 to 4,611)

The version of ECL-MST that includes the data movement to and from the GPU (ECL-MST memcpy) is the second fastest code. Whereas the baseline ECL-MST code is nearly four times faster, meaning the memory transfers take significantly longer than the MST computation, ECL-MST memcpy is still faster on every tested input (8.1 times in the mean) than the fastest CPU code. This shows that our code can be used to accelerate MST computations even in cases where the rest of the computation takes place on the CPU.

The results on System 2 (Table 4) are similar to those from System 1. The absolute runtimes are generally lower since the GPU is faster and newer. The variance is very small with an average of 0.0014% and a maximum of 0.0107%. ECL-MST is again faster than the other codes on all tested inputs. According to the geometric mean over the MSF inputs, our code is 239.8 times faster than serial, 27.1 times faster than the fastest CPU-parallel code (PBBS), and 12.7 times faster than the RAPIDS cuGraph GPU code. On the MST inputs, it is 323.3 times faster than serial, 44 times faster than the fastest CPU-parallel code (PBBS), and 4.5 times faster than the fastest GPU code (Jucele). We see the same trends as on System 1: ECL-MST is particularly fast on large scale-free inputs, and ECL-MST memcpy is 5.6 times slower than ECL-MST but still faster than the fastest CPU code on our system. Of course, on systems with a very fast CPU and slow GPU, the CPU code may outperform the GPU code.

When profiling our code using NVIDIA's Nsight tool, we found the initialization kernel to take about 40% of the total runtime on average. This kernel is relatively slow because it directly accesses the graph data structure. All remaining kernels only access the worklist, which contains all needed graph information. This highlights the

importance of the hybrid parallelization scheme, as without it, the initialization takes even longer (see Section 5.3). Of the computation kernels, all of which are launched multiple times, the first kernel (Lines 14-23 in Alg. 2) is responsible for about 35% of the total runtime on average. In contrast, computation kernels two and three (Lines 27-33 and 34-37, respectively) each take only about 12% of the runtime on average. The initialization kernel is launched twice if filtering is used and once otherwise. Depending on the input, the computation kernels are launched between 4 (kron_g500-logn21) and 15 times (delaunay_n24). Due to the *if* statement on Line 26, the first computation kernel is invoked twice more when filtering is used and once more otherwise.

As discussed in Section 4, cuGraph has both a float and a double version of the MST code. The float version cannot run three of our inputs, which is why we show the results from the double version. On the remaining inputs, the float version of cuGraph is, on average, 1.21 times faster than the double version but still substantially slower than ECL-MST on every input.

In summary, the runtime results show ECL-MST to outperform the other tested MST codes on a variety of inputs and on different GPU generations. In fact, ECL-MST is faster on all tested inputs by a large margin. Additionally, even when including the memory copy times, ECL-MST outperforms the tested CPU codes, meaning that it can be used for accelerating MST computations in programs that mainly run on the CPU.

5.2 Throughput

The throughputs, i.e., the number of edges divided by the runtime, on the two systems are shown in Figures 3 and 4. In these bar charts, the inputs are listed along the x-axis and the throughputs in millions of edges per second along the y-axis. Taller bars are better.

Based on the geometric mean over the MSF inputs on System 1 (Figure 3), we find that ECL-MST processes 1.57 billion edges per second with the closest performing code being CPU-parallel PBBS at 48 million edges per second. ECL-MST reaches its highest throughput of 7.09 billion edges per second on coPapersDBLP. The lowest throughput is 0.73 billion edges per second on r4-2e23.sym, which is still over 3.6 times higher than the highest throughput reached by the CPU codes (PBBS reaches 0.20 billion edges per second on kron_g500-logn21). On the MST inputs, ECL-MST's throughput increases to a geometric mean of 1.65 billion edges per second with the next closest code being Jucele at 0.36 billion edges per second.

The throughputs for System 2 (Figure 4) are higher due to the faster GPU. On average, for both the MSF and MST inputs, ECL-MST processes 2.54 billion edges per second. The highest throughput is again reached on coPapersDBLP at 12.24 billion edges per second. as-skitter yields the lowest throughput of 1.17 billion edges per second. CPU-parallel PBBS reaches 0.09 billion edges per second, cuGraph reaches 0.20 billion edges per second, and the Jucele GPU code reaches 0.59 billion edges per second.

In summary, these results show that ECL-MST can handle both small and large inputs well, reaching high throughputs on all tested graphs. When correlating throughput with various graph properties, we found ECL-MST's throughput to significantly correlate with the average degree. This is likely because disqualifying an edge from the MST is faster than including an edge in the MST and, for high average-degree graphs, a larger portion of the work is edge disqualification. Overall, the throughputs are high on both GPUs, indicating that our performance optimizations are not specific to a particular GPU or hardware generation.

5.3 Optimization Evaluation

To measure the benefit of the optimizations discussed in Section 3.2, we wrote additional versions of our code, each with one more optimization removed than the previous. As before, we ran the resulting codes 9 times for each input, report the median computation time, and verified the correctness. The runtimes are listed in Table 5. Figure 5 shows the corresponding throughputs. For reference, we included the throughput of the fastest MST algorithm from the literature, which is Jucele. As a consequence, we do not show data for the inputs that contain multiple connected components. We only present results for System 2 as it has the faster GPU.

We started with our fully-optimized ECL-MST code and removed the following optimizations in the listed order.

(1) Since load instructions are generally faster and more parallel than atomic operations, and the atomics on Lines 20 and 21 in Alg. 2 often do not find a new minimum, the ECL-MST code first checks with an *if* statement whether the atomicMin might lower the value. If not, then there is no reason to execute the atomicMin. The "No Atomic Guards" version elides these checks and always executes the atomicMin operations.
(2) For load-balancing reasons, the loop on Line 3 of Alg. 2 is parallelized across the warp threads if the adjacency list is sufficiently long. The "Thread-Based" version never parallelizes this inner loop.
(3) ECL-MST uses filtering for all inputs whose average degree is ≥ 4. The "No Filter" version does not use any filtering.
(4) One of our key optimizations is the implicit path compression, meaning that, whenever the code places an edge on the worklist, it uses the representatives of its endpoints. The "No Implicit Path Compression" places the actual endpoint vertex IDs on the worklist and employs explicit path compression (using the path-halving code for GPUs [14]) whenever those vertices are used later.
(5) Since the CSR graph representation includes two directed edges for each undirected edge, which necessarily form a cycle, ECL-MST only processes one edge of each pair. The "Both Edge Directions" version processes all edges.
(6) Each element in the worklist is a 4-tuple (Section 3.3). The "No Tuples" version stores the 4 items in separate arrays.
(7) ECL-MST is data-driven, meaning it only processes the edges that are on the worklist. The "Topology-Driven" version, in contrast, always processes all graph edges in every iteration of the *while* loop on Line 12 in Alg. 2.
(8) Another key optimization is that most of our code is edge-centric, that is, the unit of work assigned to a thread is an edge. This tends to yield good load balance. The "Vertex-Centric" version assigns a vertex to each thread, and the thread is responsible for processing all edges of the vertex.

Based on the geometric-mean results, "No Atomic Guards" adds 27% additional runtime, "Thread-Based" adds 9% more, "No Filter" adds 30%, "No Implicit Path Compression" adds 58%, "Both

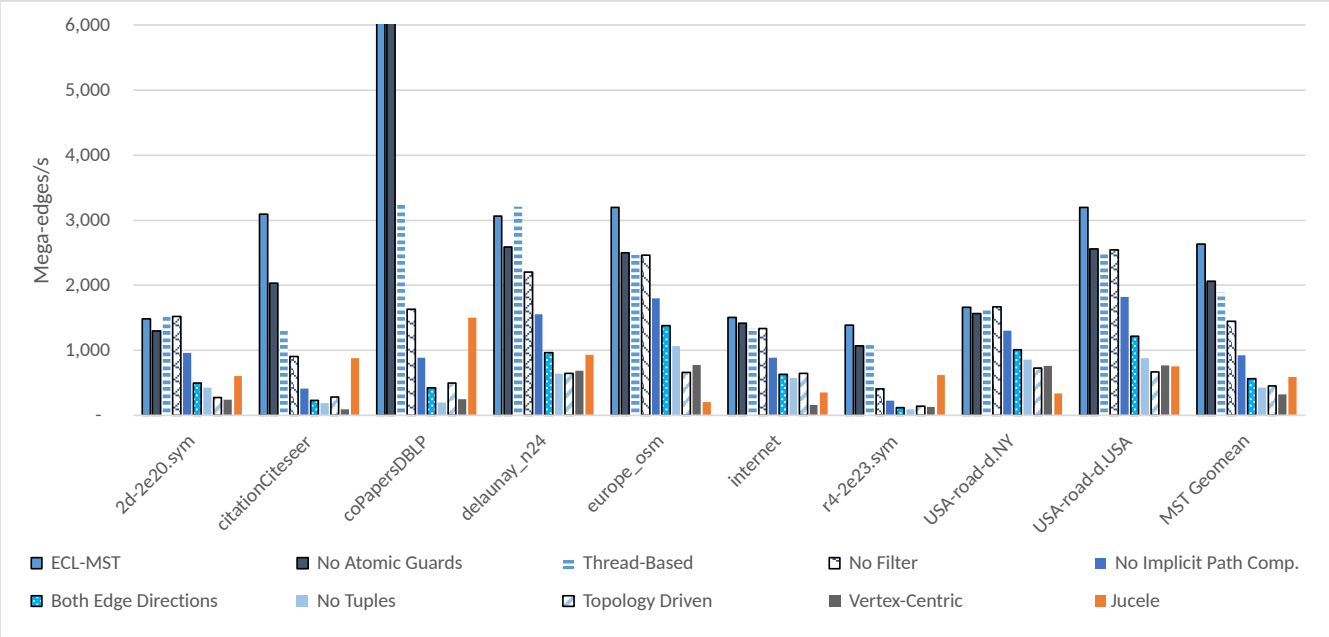

Figure 5: ECL-MST throughput in millions of edges per second when gradually removing performance optimizations

Edge Directions" adds 62%, and "No Tuples" adds 33%. Interestingly, "Topology-Driven" decreases the runtime by 6% at this point. Finally, "Vertex-Centric" adds 40%. Clearly, all but one of these optimizations boost the performance on average, though not necessarily on every input. Switching from a topology-driven to a data-driven implementation appears to be a bad idea. However, this switch enables and simplifies several other optimizations, thus actually improving performance. Together, all of these optimizations make ECL-MST over 8× faster. In other words, our code would be significantly slower than Jucele if we had not included these optimizations.

Processing edges in only one direction is an important optimization as it avoids a significant amount of extra work. A less obvious but nearly as impactful optimization is our implicit path compression, which also minimizes unnecessary work. Employing edge-centric computation improves the load balance and also reduces the workload as it enables the removal of specific edges of a vertex from consideration in the following iterations. Using 4-tuples, filtering, and atomic guards are the next most important optimizations. They either improve memory access time or prevent expensive extra work. The benefit of the hybrid parallelization strategy, which dynamically assigns the high-degree vertices to entire warps and the low-degree vertices to individual threads, is less visible because not all inputs benefit from this optimization.

These results highlight the importance of our optimizations and show how, by neglecting to include them, our code quickly loses its advantage over the related work. Even removing just a single optimization has the potential to significantly lower the performance. Without these optimizations, ECL-MST is substantially slower than the fastest code from the literature. Hence, the performance benefit of ECL-MST is primarily due to this set of optimizations.

5.4 Random-Seed Evaluation

As explained in Section 3.3, we sample 20 random edges to compute the filter threshold. We aim for 3 times the number of edges in the final tree. In this subsection, we evaluate the impact of this random selection on the performance of ECL-MST as well as how accurate the resulting thresholds are.

First, we ran ECL-MST with 99 different random seeds. Figure 6 displays the results. The box-and-whisker plot shows the throughput distribution due to the various seeds. The maximum speed for each input is indicated by the top whisker and the minimum by the bottom whisker. The boxes range from the first to the third quartile. The line inside the box demarcates the median throughput.

For all inputs except coPapersDBLP, the variance based on the seed is quite low. The three road maps, internet, and 2d-2e20.sym have an average degree under 4 (cf. Table 2), meaning the filtering threshold is not used. Consequently, there is essentially no variation between the runs other than normal fluctuations. Many of the scale-free graphs exhibit a noticeable variability. coPapersDBLP yields by far the largest range. This is our only graph where most of the vertices have a high degree. On the remaining inputs, the performance of ECL-MST does not depend much on the chosen seed. Nevertheless, we used the seed that resulted in the median throughput for all other experiments reported in this paper.

Next, we took this median seed and measured how close the resulting threshold ended up being to our target of 3 times the number of edges in the tree. Figure 7 shows the percentage difference from this target value for all inputs for which ECL-MST employs filtering. These results indicate that the random selection rarely chooses an edge weight that yields more than double or less than half as many edges being filtered than we intended.

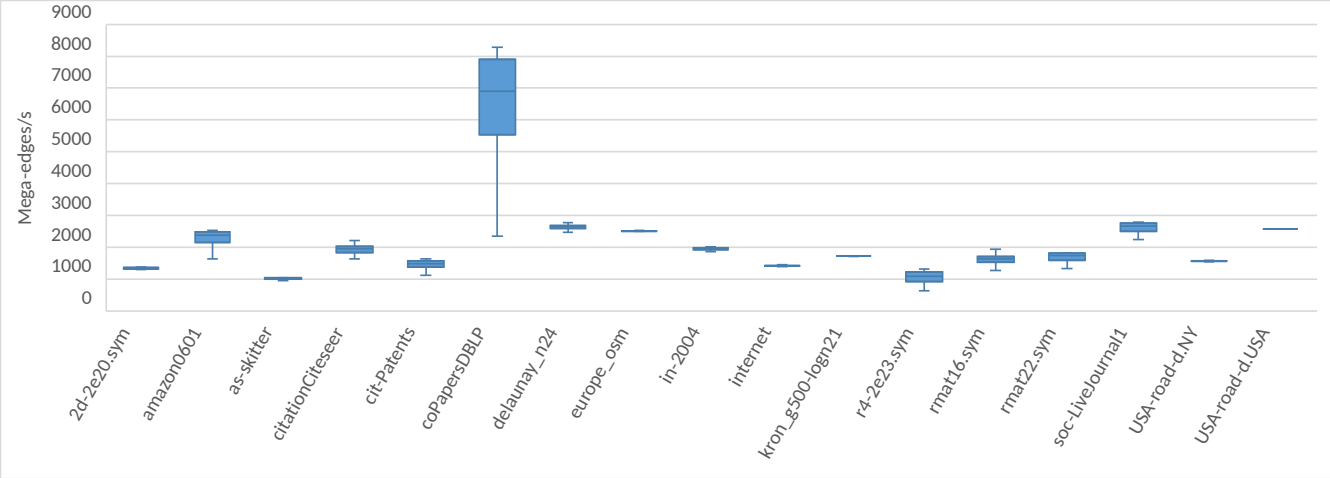

Figure 6: Throughput variability of ECL-MST with different random seeds

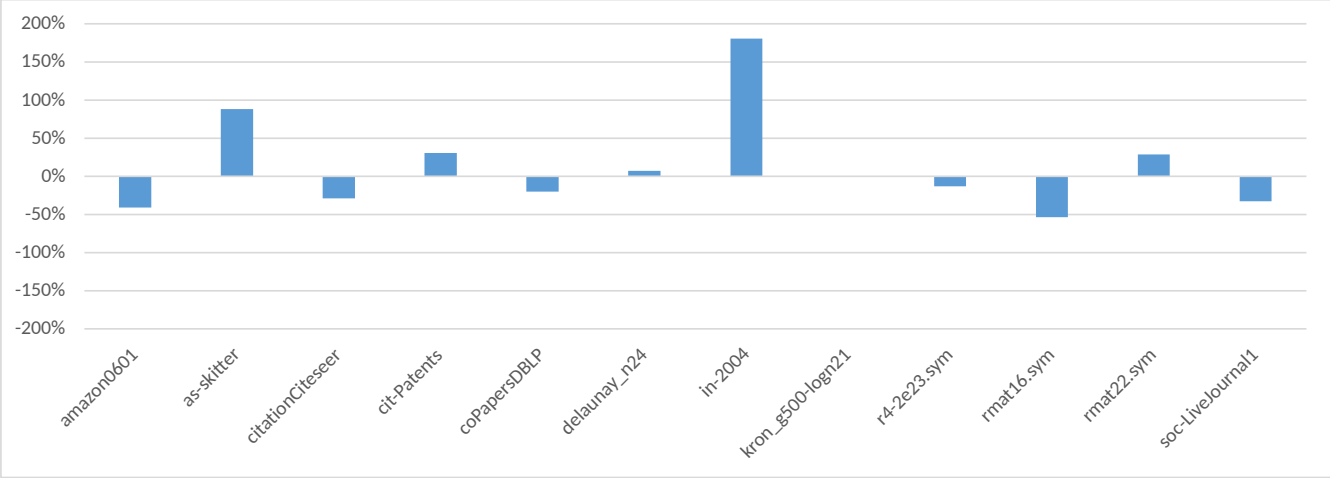

Figure 7: Relative distance from the target filtering threshold of 3 times the number of vertices in the MST

6 CONCLUSION

This paper presents ECL-MST, a high-performance MST implementation designed specifically for GPUs. On our test systems, it outperforms leading GPU and parallel CPU codes from the literature on each of 17 inputs from different domains. ECL-MST is a lock-free implementation that combines new and existing optimizations from the literature with a parallelization approach that unifies both Kruskal's and Borůvka's algorithm. It works well on different types of graphs and across recent GPU architectures. On a Titan V GPU, ECL-MST outperforms the next closest code by a factor of 4.6 on average. On an RTX 3080 Ti GPU, it is 4.5 times faster. ECL-MST is 12.7 times faster than RAPIDS cuGraph on average.

The ECL-MST CUDA code is publicly available in open source on the web [6] and on github [6]. It is easy to install and compile as it does not rely on third-party code. With under 300 statements (including the serial verification code), the implementation is quite compact. In fact, it only contains 70 CUDA kernel statements.

We hope our research and the resulting MST code along with its optimizations will help speed up important computations that require MSTs. Moreover, we hope it will inspire other researchers to take a fresh look at some of the classic graph algorithms and devise faster and more parallel GPU and CPU implementations thereof.

ACKNOWLEDGMENTS

This work has been supported in part by the National Science Foundation under Award Number 1955367 and by an equipment donation from NVIDIA Corporation. We thank Randy Cornell for his help with exploring several MST optimizations and Jerry Rosado for his help with installing third-party software.

REFERENCES

[1] Charles J Alpert, Te C Hu, Jen-Hsin Huang, Andrew B Kahng, and David Karger. 1995. Prim-Dijkstra tradeoffs for improved performance-driven routing tree design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 14, 7 (1995), 890–896.

Table 3: System 1 computation times in seconds

Input	ECL-MST	ECL-MST memcpy	Jucele GPU	Gunrock GPU	Uminho GPU	Lonestar CPU	PBBS CPU	Uminho CPU	PBBS Ser.
2d-2e20.sym	0.0049	0.0126	0.0147	0.0195	0.0278	1.0710	0.1063	0.0916	0.5176
amazon0601	0.0026	0.0124	NC	NC	0.2895	0.6420	0.0745	0.1674	0.4636
as-skitter	0.0262	0.0721	NC	NC	1.5121	2.0230	0.3340	0.8681	1.5179
citationCiteseer	0.0012	0.0056	0.0040	0.0099	0.1666	0.3810	0.0478	0.1070	0.1641
cit-Patents	0.0368	0.1068	NC	NC	3.3859	7.7790	0.8081	1.6587	3.2648
coPapersDBLP	0.0043	0.0660	0.0304	0.0633	2.0874	1.8400	0.2505	1.1022	1.9628
delaunay_n24	0.0491	0.2632	0.1938	0.2807	0.3942	9.1230	3.0202	1.0986	15.6779
europe_osm	0.0560	0.3092	0.7843	0.4106	0.0985	12.8640	4.5846	1.6579	18.9955
in-2004	0.0176	0.0736	NC	NC	0.8735	1.4530	0.3243	0.3874	1.7615
internet	0.0003	0.0008	0.0015	0.0029	0.0072	0.1290	0.0263	0.0094	0.0295
kron_g500-logn21	0.0979	0.4002	NC	NC	3.2057	31.0100	0.8920	3.5448	6.5958
r4-2e23.sym	0.0921	0.2088	0.2087	0.2992	7.4050	18.0720	1.9853	4.8016	9.3615
rmat16.sym	0.0006	0.0019	NC	NC	0.1066	0.1770	0.0178	0.0679	0.0526
rmat22.sym	0.0456	0.1600	NC	NC	5.9371	20.5680	0.9937	4.1182	5.0334
soc-LiveJournal1	0.0353	0.2121	NC	NC	11.6288	15.2210	1.0564	4.4213	5.9403
USA-road-d.NY	0.0006	0.0016	0.0030	0.0051	0.0039	0.1590	0.0327	0.0140	0.0741
USA-road-d.USA	0.0283	0.1424	0.1484	0.2358	0.0668	6.5370	2.3909	0.6850	9.4397
MSF GeoMean	0.0103	0.0411	NC	NC	0.3978	2.4886	0.3335	0.4775	1.4231
MST GeoMean	0.0070	0.0290	0.0324	0.0485	0.1199	1.8148	0.3465	0.2734	1.2856

Table 4: System 2 computation times in seconds

Input	ECL-MST	ECL-MST memcpy	Jucele GPU	Gunrock GPU	cuGraph GPU	UMinho GPU	Lonestar CPU	PBBS CPU	Uminho CPU	PBBS Ser.
2d-2e20.sym	0.0028	0.0086	0.0069	0.0156	0.0399	0.0161	1.1130	0.0682	0.0710	0.5726
amazon0601	0.0016	0.0105	NC	NC	0.0177	0.2406	0.6730	0.0448	0.1362	0.5009
as-skitter	0.0190	0.0584	NC	NC	0.0616	1.2248	2.0300	0.1549	0.7815	1.6326
citationCiteseer	0.0007	0.0050	0.0026	0.0064	0.0117	0.1215	0.4620	0.0298	0.0902	0.1788
cit-Patents	0.0198	0.0790	NC	NC	0.1489	2.8603	8.4130	0.2932	1.4496	3.1998
coPapersDBLP	0.0025	0.0563	0.0203	0.0458	0.0432	1.6820	1.9250	0.1459	1.0472	2.0878
delaunay_n24	0.0329	0.2378	0.1078	0.2817	0.5793	0.2053	7.3370	1.3487	0.9552	17.7500
europe_osm	0.0338	0.2720	0.5195	0.3044	3.7105	0.0590	12.4180	2.1357	1.1169	22.0196
in-2004	0.0126	0.0608	NC	NC	0.0593	0.7381	1.4880	0.1576	0.3940	1.8727
internet	0.0003	0.0009	0.0011	0.0020	0.0047	0.0052	0.2360	0.0232	0.0153	0.0328
kron_g500-logn21	0.0509	0.3488	NC	NC	0.2519	2.5700	33.3920	0.3694	4.4750	7.4356
r4-2e23.sym	0.0482	0.1493	0.1087	0.2424	0.3708	6.0761	16.6760	0.8266	4.2315	10.0535
rmat16.sym	0.0004	0.0018	NC	NC	0.0040	0.0761	0.2640	0.0116	0.0617	0.0552
rmat22.sym	0.0241	0.1205	NC	NC	0.1929	4.8767	18.0470	0.4096	3.6322	4.8061
soc-LiveJournal1	0.0186	0.1867	NC	NC	0.2023	9.5606	12.4630	0.4946	4.0870	5.8235
USA-road-d.NY	0.0004	0.0015	0.0022	0.0039	0.0112	0.0030	0.2560	0.0213	0.0214	0.0807
USA-road-d.USA	0.0181	0.1115	0.0768	0.1833	0.7618	0.0385	5.7260	1.0208	0.5242	9.9725
MSF GeoMean	0.0063	0.0346	NC	NC	0.0805	0.2924	2.6685	0.1718	0.4506	1.5210
MST GeoMean	0.0044	0.0247	0.0195	0.0373	0.0953	0.0808	2.0036	0.1921	0.2589	1.4110

Table 5: ECL-MST computation times in seconds when gradually removing performance optimizations

Input	ECL-MST	No Atomic Guards	Thread-Based	No Filter	No Impl. Path Compr.	Both Edge Dir.	No Tuples	Topology-Driven	Vertex-Centric
2d-2e20.sym	0.0028	0.0032	0.0028	0.0028	0.0043	0.0085	0.0098	0.0151	0.0172
citationCiteseer	0.0007	0.0011	0.0018	0.0025	0.0056	0.0101	0.0124	0.0081	0.0250
coPapersDBLP	0.0025	0.0047	0.0094	0.0187	0.0343	0.0720	0.1517	0.0610	0.1236
delaunay_n24	0.0329	0.0389	0.0314	0.0457	0.0649	0.1041	0.1559	0.1556	0.1460
europe_osm	0.0338	0.0432	0.0438	0.0439	0.0601	0.0784	0.1016	0.1629	0.1397
internet	0.0003	0.0003	0.0003	0.0003	0.0004	0.0006	0.0007	0.0006	0.0024
r4-2e23.sym	0.0482	0.0627	0.0622	0.1635	0.2927	0.5576	0.7063	0.4742	0.5099
USA-road-d.NY	0.0004	0.0005	0.0004	0.0004	0.0006	0.0007	0.0009	0.0010	0.0010
USA-road-d.USA	0.0181	0.0226	0.0230	0.0227	0.0317	0.0474	0.0653	0.0865	0.0754
MST GeoMean	0.0044	0.0056	0.0061	0.0079	0.0125	0.0203	0.0270	0.0255	0.0358

[2] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming.*

181–192.
[3] J.J. Brennan. 1982. Minimal spanning trees and partial sorting. *Operations Research Letters* 1, 3 (1982), 113–116. https://doi.org/10.1016/0167-6377(82)90010-4
[4] Mariel Brinkhuis, Gerrit A Meijer, Paul J Van Diest, Leonard T Schuurmans, and JP Baak. 1997. Minimum spanning tree analysis in advanced ovarian carcinoma. An investigation of sampling methods, reproducibility and correlation with histologic grade. *Analytical and quantitative cytology and histology* 19, 3 (1997), 194–201.
[5] Martin Burtscher and Alex Fallin. 2023. ECL-MST Git Repository. https://github.com/burtscher/ECL-MST. Accessed: 2023-08-18.
[6] Martin Burtscher and Alex Fallin. 2023. ECL-MST Website. https://cs.txstate.edu/~burtscher/research/ECL-MST/. Accessed: 2023-08-18.
[7] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663
[8] Jucele França de Alencar Vasconcellos, Edson Norberto Cáceres, Henrique Mongelli, and Siang Wun Song. 2018. A new efficient parallel algorithm for minimum spanning tree. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 107–114.
[9] DIMACS data sets 2007. DIMACS data sets. https://www.diag.uniroma1.it/challenge9/download.shtml
[10] Galois data sets 2018. Galois data sets. https://iss.oden.utexas.edu/?p=projects/galois
[11] R.L. Graham and Pavol Hell. 1985. On the History of the Minimum Spanning Tree Problem. *Annals of the History of Computing* 7 (02 1985), 43–57. https://doi.org/10.1109/MAHC.1985.10011
[12] Ronald L Graham and Pavol Hell. 1985. On the history of the minimum spanning tree problem. *Annals of the History of Computing* 7, 1 (1985), 43–57.
[13] Michael Held and Richard M Karp. 1970. The traveling-salesman problem and minimum spanning trees. *Operations Research* 18, 6 (1970), 1138–1162.
[14] Jayadharini Jaiganesh and Martin Burtscher. 2018. A high-performance connected components implementation for GPUs. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 92–104.
[15] Jucele GPU 2018. Jucele GPU. https://github.com/jucele/NewMinimumSpanningTree/
[16] Peter Kipfer and Rüdiger Westermann. 2005. Improved GPU sorting. *GPU gems* 2 (2005), 733–746.
[17] Oleg V. Komogortsev, Sampath Jayarathna, Do Hyong Koh, and Sandeep Munikrishne Gowda. 2010. Qualitative and quantitative scoring and evaluation of the eye movement classification algorithms. *Proceedings of the 2010 Symposium on Eye-Tracking Research & Applications* (2010), 65 – 68.
[18] Joseph B Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society* 7, 1 (1956), 48–50.
[19] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[20] Lonestar CPU 2022. Lonestar CPU. https://github.com/IntelligentSoftwareSystems/Galois/
[21] Artur Mariano, Alberto Proenca, and Cristiano Da Silva Sousa. 2015. A generic and highly efficient parallel variant of boruvka's algorithm. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 610–617.
[22] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) *(PPoPP '12)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/2145816.2145832
[23] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Data-Driven Versus Topology-driven Irregular Computations on GPUs. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 463–474. https://doi.org/10.1109/IPDPS.2013.28
[24] Muhammad Osama, Serban D. Porumbescu, and John D. Owens. 2022. Essentials of Parallel Graph Analytics. In *Proceedings of the Workshop on Graphs, Architectures, Programming, and Learning (GrAPL 2022)*. 314–317. https://doi.org/10.1109/IPDPSW55747.2022.00061
[25] Vitaly Osipov, Peter Sanders, and Johannes Singler. 2009. The Filter-Kruskal Minimum Spanning Tree Algorithm. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments* (New York, New York). Society for Industrial and Applied Mathematics, USA, 52–61.
[26] Sreepathi Pai and Keshav Pingali. 2016. A compiler for throughput optimization of graph algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 1–19.
[27] PBBS Codes 2012. PBBS Codes. https://github.com/cmuparlay/pbbsbench/
[28] Robert Clay Prim. 1957. Shortest connection networks and some generalizations. *The Bell System Technical Journal* 36, 6 (1957), 1389–1401.
[29] RAPIDS cuGraph GPU 2022. RAPIDS cuGraph GPU. https://github.com/rapidsai/cugraph/
[30] RAPIDS Raft 2022. RAPIDS Raft. https://github.com/rapidsai/raft/
[31] Rohit Setia, Arun Nedunchezhian, and Shankar Balachandran. 2009. A new parallel algorithm for minimum spanning tree problem. In *Proc. International Conference on High Performance Computing (HiPC)*. 1–5.
[32] UMinho Codes 2015. UMinho Codes. https://github.com/Beatgodes/BoruvkaUMinho/
[33] Wei Wang, Yongzhong Huang, and Shaozhong Guo. 2011. Design and implementation of GPU-based prim's algorithm. *International Journal of Modern Education and Computer Science* 3, 4 (2011), 55.
[34] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Transactions on Parallel Computing* 4, 1 (Aug. 2017), 3:1–3:49. https://doi.org/10.1145/3108140

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

10.5281/zenodo.8271647

ARTIFACT IDENTIFICATION

(1) Main contributions of the paper
- The paper presents a high-speed Minimum Spanning Tree (MST) implementation for GPUs
- The paper describes the optimizations that make the implementation fast, and provides empirical evidence to demonstrate the benefit of these optimizations
- The paper evaluates the resulting implementation and compares it with state-of-the-art GPU-parallel, CPU-parallel, and CPU-serial MST implementations on 2 GPUs and 2 CPUs using 17 input graphs from various domains

(2) Role of the computation artifacts
- The computational artifact can run the described MST implementation as well as the codes from the related work with the 17 input graphs and compute the median runtimes
- It can also run various de-optimized versions of our MST code and compute the median runtimes

(3) Software architecture
- The DOI is: 10.5281/zenodo.8271647
- The URL for the Git repository is: https://github.com/alexfallin/MST-Artifact-SC23/
- The codes from the related work, our MST code, the de-optimized codes, and the input files each have a dedicated directory
- Each directory contains a build and run script that installs and runs the code in that directory, respectively
- The "set_up.sh" script downloads the 17 inputs, converts them into the various needed formats, and runs each build script to install the codes
- The "run_all_compare.sh" script executes the timed experiments for each code and writes the results to [code]_out.csv files that are placed in the root folder
- The "run_all_deoptimize.sh" script runs the de-optimization experiments and writes the results to ecl_mst_[deopts]_out.csv file
- The "generate_compare_tables.py" script reads the output files and converts them into CSV tables similar to those shown in the paper
- The "generate_deopt_tables.py" script reads the output files and converts them into CSV tables similar to those shown in the paper

(4) Extent of reproducibility
- The computational artifacts can run our code, the codes from the literature we used for the performance comparison, and our de-optimized codes and can reproduce the result figures from the paper

REPRODUCIBILITY OF EXPERIMENTS

(1) Experiment workflow
- Install all the codes and input files using "set_up.sh" script
- Run the "run_all_compare.sh" and "run_all_deoptimize.sh" scripts to produce the result CSV files
- Run the "generate_*_tables.py" scripts to generate the tables that show the results of the above experiments

(2) Execution time
- About 48 hours to run the related work comparisons
- About 2 hours to run the de-optimization experiments

(3) Expected results
- The CSV files that contain the runtimes for each code
- Combined CSV tables for both throughput and runtime

(4) How the results relate to the paper results
- The results should show similar runtime and throughput relationships between the different implementations as in the paper
- The results will differ based on the CPU and GPU that is used

ARTIFACT DEPENDENCIES REQUIREMENTS

(1) Hardware resources
- We used a 3.5 GHz Ryzen Threadripper 2950X CPU, a 2.9 GHz Xeon Gold 6226R CPU, a 1.2 GHz TITAN V GPU, and a 1.67 GHz RTX 3080 Ti GPU for the experiments
- The minimum hardware requirement is a multi-core CPU and a CUDA-capable GPU with a Compute Capability of at least 7.0

(2) Operating system
- We used Fedora Linux 34 and 36. No specific operating system is required to run the experiments

(3) Software libraries needed
- We used GCC 11.3.1 and 12.2.1 with the "-O3 -march=native" flags to compile the codes and NVCC 11.7 and 12.0 with the "-arch=sm_70 and -arch=sm_86" flags for the CUDA codes
- The related works require:
  - Boost library 1.58+
  - libllvm 7.0+
  - libfmt 4.0+
  - gcc version 9.3+
  - cmake version 3.26+
  - CUDA 11.0+
  - NVIDIA driver 450.80.02+
  - Anaconda 23.5+
  - Python 3.0+
  - Matplotlib
  - NumPy

(4) Input datasets needed
- We used 17 inputs for the experiments. They can be downloaded using either set_up.sh in the root directory of the artifact or download_inputs.sh in the Inputs folder

- The only constraints on inputs are that they must be in binary 32-bit CSR format[1] and they must fit in the selected GPU's memory

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

(1) Process to install and compile the codes
   - The "set_up.sh" script will download the inputs, convert them to the various required formats, and will also download and build the related works

(2) Process to deploy the codes
   - Use the "run_all_compare.sh" script to run each code to compare their run times. The estimated time to complete these runs is 48 hours, subject to system speed
   - Use the "run_all_deoptimize.sh" script to run the deoptimization experiments from the paper. The estimated run time of these experiments is 2 hours, subject to system speed

[1] https://cs.txstate.edu/~burtscher/research/ECLgraph/index.html