International Journal of Parallel Programming manuscript No.
(will be inserted by the editor)

Accelerating Computation of Steiner Trees on GPUs

Rajesh Pandian M · Rupesh Nasre · N.S. Narayanaswamy

Abstract The STEINER TREE PROBLEM (STP) is a well studied graph theoretic problem. It computes a minimum-weighted tree of a given graph such that the tree spans a given subset of vertices called terminals. STP is NP-hard. Due to its wide applicability, it has been a challenge problem in the 11^{th} DIMACS implementation challenge and the PACE 2018 challenge. Due to its importance, polynomial-time approximation algorithms have been devised for solving the STP. One of the most popular algorithms is by Kou, Markowsky and Berman (KMB) which provides a 2-approximation to STP. In practice, a naïve implementation of the KMB algorithm is prohibitively slow for large graphs. Our goal in this work is to improve KMB algorithm's practical utility by parallelizing it on GPU and reduce its execution time on real-world graphs. This parallelization faces several challenges due to the inherent irregular nature of computation, as well as critical design decisions affecting the algorithm choice and optimizations. We overcome these challenges with interesting algorithmic observations and by exploiting parallelization within sub-steps, and develop the first GPU-based efficient approach to computing Steiner trees using KMB. We illustrate the effectiveness of our approach using several graph benchmarks from the DIMACS Challenge, the PACE Challenge, SteinLib, and SNAP. Our highly optimized GPU implementation achieves an average 20× speedup over the CPU-sequential Open Graph algorithms and Data structure (OGDF)'s KMB implementation. In addition to this, our optimized CPU implementation achieves an average 4× over OGDF's KMB, the only published open-source KMB implementation.

Keywords Steiner trees, Parallel algorithms, Approximation algorithms, Graphics Processing Units

Rajesh Pandian M · Rupesh Nasre · N.S. Narayanaswamy
Department of Computer Science and Engineering, IIT Madras
E-mail: mrprajesh, rupesh, swamy@cse.iitm.ac.in

1 Introduction

Steiner trees are generalizations of minimum spanning trees. Applications of Steiner trees include very-large-scale integration (VLSI) design, network routing, phylogenetic tree construction and vehicle routing [19]. In particular, Steiner trees are applied in placement and routing of components in the design of printed circuit boards (PCBs), integrated circuits (ICs), and field programmable gate arrays (FPGAs). To define the problem, consider a simple, undirected, connected and edge-weighted graph $G(V,\ E,\ W,\ L)$ where $W: E \to \mathbb{Z}_0^+$ is the non-negative weight function, and $L \subseteq V$ represents a set of special vertices called *terminals*. The remaining vertices $V \setminus L$ are called non-terminals or *Steiner* vertices. The standard notation of $n = |V|$, $m = |E|$ and additionally $k = |L|$ is used throughout. Formally, STP is stated as follows:

Steiner Tree Problem (STP)

Input : Connected graph $G(V, E, W, L)$, $W{:}E{\to}\mathbb{Z}_0^+$, terminals $L \subseteq V$
Output: Connected subtree T of G containing all the terminals
Goal : Minimize the sum of edge-weights of the tree T.

A minimal-connected subgraph (that is, a tree) of G that contains all the terminals is called a *Steiner tree* T. The minimum weighted Steiner tree is called the *optimum* Steiner tree. The goal of the Steiner Tree Problem (STP) is to compute a subgraph of G which contains terminals L and the sum of the edge-weights in the subgraph is minimized. Note that the edge-weights can be zero. When $L = V$, the STP specializes to the minimum spanning tree (MST) problem, and when $L = \{u, v\}$ then the STP specializes to the shortest u-v path problem. Both these problems can be solved in polynomial-time. Further, it is known that for values up to $|L| = \log n$, STP can be solved in polynomial-time using dynamic programming-based parameterized algorithms [16, 20].

Properties of Steiner trees. A graph may have multiple optimum Steiner trees. If there are multiple shortest paths between two vertices (or terminals), then the optimum Steiner tree is not unique. This can also be seen in Figure 1 with a and c as two terminals. In any optimum Steiner tree, all the leaf vertices are terminals, but the converse need not be true. If the edge-weights of the input graph are unique, then there exists a unique MST. However, this is not true for STP even with two terminals (Figure 1).

One of the most fundamental differences between MST and STP is their algorithmic complexity. An MST can be computed in polynomial time and several algorithms have been proposed towards this (e.g., Prim's, Kruskal's, and Boruvka's). On the other hand, STP is NP-Hard [21] even on unweighted graphs. The decision version of the optimization problem, that is, deciding whether a graph has a Steiner tree with at most p Steiner vertices is NP-Complete [21]. Since the STP is an important problem in several real-world applications, for smaller graph sizes and for special graph classes, *efficient*

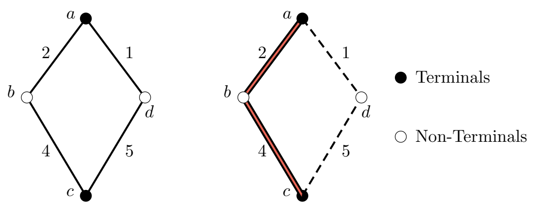

Fig. 1 Example - Steiner tree on a graph with unique edge weights need not be unique. For the above instance two Steiner tree are possible: {(a,b)(b,c)} and {(a,d)(d,c)}

exponential-time algorithms can be used to seek exact answers [12,16]. Such exact algorithms and parameterized algorithms face practical limitations at scale. These limitations were explored in the 11^{th} DIMACS implementation challenge 2014 [26]. Due to the importance of the problem, STP was also a focus of Parameterized Algorithms and Computational Experiments (PACE) Challenge in 2018 [6,7]. 1 million terminals.

A practical alternative way of dealing with the NP-hardness of STP is by considering polynomial-time approximation algorithms [2,8,36,43,49]. One of the most well-known algorithms is by Kou, Markowsky and Berman [29] which provides a 2-approximation to STP. We refer to this elegant algorithm as the KMB algorithm throughout the text. It computes a Steiner tree by performing a set of shortest path computations followed by an MST computation. We explain it in the next section, but briefly: on an input instance $G(V, E, W, L)$, the KMB algorithm has three phases. The first phase computes the shortest distance between every pair of terminals. The second phase builds a graph G' over terminals, having edge-weights corresponding to the shortest distances computed in the previous phase, and then computes an MST on G'. The last phase substitutes for every MST edge in G', the corresponding shortest path in G to create a final graph G''. This is clearly a subgraph of G but may contain a cycle. So, the KMB algorithm finally computes the MST of G'' to output as a Steiner tree. This tree is proved to have weight at most twice the weight of the optimum Steiner tree in G.

There are some instances in SteinLib [28] for which the optimum solutions are still unknown. Such is the case also for PACE 2018's heuristic track instances. Further, on the graph instances where the optimum is known, the state of the art implementations run for more than a few hours to output the optimum (Section 3.3 in Bonnet and Sikora [7]). Despite the elegant formulation of the KMB algorithm, it is time-consuming especially on large graphs, which limits its practical use. Even in our setup, to output a Steiner tree, an instance took 1145 seconds on the CPU. Our goal in this work is to exploit the power of GPGPUs and develop an efficient implementation of the KMB algorithm to reduce its execution time on larger inputs. Unfortunately, applying GPU-based parallelization to the KMB algorithm witnesses several challenges. We tame these challenges by exploiting various invariants satisfied by the Steiner trees as well as by the KMB algorithm.

Contributions. This paper makes the following contributions:

– We develop the first GPU-based optimized implementation of the KMB algorithm to compute 2-approximate Steiner trees. We illustrate the challenges faced in the parallelization and show how to exploit various properties of Steiner trees as well as of the KMB algorithm.
– Since single-source shortest path (SSSP) is a crucial building block for the KMB algorithm, we demonstrate various GPU parallelization strategies and optimizations that can accelerate single and multiple SSSP computation on undirected graphs.
– We illustrate the effectiveness of our parallelization using a suite of graphs from the PACE Challenge as well as large real-world graphs. Our GPU-based implementation achieves up to 62× (average 20×) speedup over OGDF's KMB [4] and it is on par with PACE 2018 winner [17] and OGDF's KMB on solution quality.
– We develop a CPU-based optimized implementation which is on-par with other sequential implementations and achieves an average 4× speedup over OGDF's KMB [4, 9].

GPU parallelization has hitherto been tried primarily on individual algorithms. Via this work, we illustrate integrating algorithms (multiple SSSPs, MSTs, and interfacing I/O conversions) which demand different parallelization techniques. Our work has integrated GPU implementations of these algorithms and we believe this is the first time such an approach has been attempted. Various design choices mentioned in the context of KMB parallelization (Section 3.1) would be useful in parallelizing other GPU implementations of NP-hard problems. Our work is that way one step closer to parallelizing graph *applications*. We have released our code and sample inputs at `https://zenodo.org/record/4477087`.

Outline. The rest of the paper is organized as follows. Section 2 explains the sequential KMB algorithm. Section 3 describes our approach towards parallelizing KMB. Section 4 describes various GPU optimizations. Section 5 quantitatively evaluates our approach using real-world benchmarks. Section 6 qualitatively compares with and contrasts against the relevant related work, and Section 7 concludes.

2 Sequential KMB Algorithm

Algorithm 1 presents the sequential KMB algorithm to bring out the bottlenecks in exploiting parallelization. It has three phases: (i) compute the shortest paths between all pairs of terminals, (ii) using the computed shortest distances, build another graph with edge-weights as the shortest distances and compute an MST on this new graph, and (iii) replace each MST edge in the new graph by the corresponding shortest path in the original graph and compute an MST in this graph, which is a Steiner tree of the original graph. We explain the phases below.

2.1 Data Structures

The algorithm starts with the input graph G and computes two more auxiliary graphs G' and G'' before outputting the Steiner tree. We use adjacency list for holding the graphs and a map to store the weights of input graph and intermediate graphs. While G' is a complete graph with all the terminal vertices, G'' is a subgraph of G We use three arrays d, p, and $PArray$. Two n-sized arrays: distance d and parent p are used to hold the complete result of one SSSP from a terminal. These two arrays are reused across SSSPs. An $(n \times k)$-sized array called $PArray$ that holds all the parent information computed by all the SSSPs. This aggregate information is required in the second and the third phases.

2.2 Algorithm

```
Input: Graph $G(V, E, W, L)$ such that $W : E \rightarrow \mathbb{Z}_0^+$ and set of terminals $L \subseteq V$
Output: A 2-approximate Steiner tree $T''(V_{T''}, E_{T''})$ such that $V_{T''} \supseteq L$
1  $G'(L, E' = \phi)$;
2  $G'' = \phi$;
3  for $u \in L$ do
4      for $v \in L$ do
5          $P_{uv}$ = ShortestPath$(u, v)$;
6          $W'(u, v) = |P_{uv}|$;
7      end
8  end
9  $T' = MST(G'(L, E'), W')$ ;
10 for $(u, v) \in E'(T')$ do
       /* Add vertices and edges of $P_{uv}$                    */
11     $G'' = G'' \cup P_{uv}$
12 end
   /* Let $G''(V'', E'')$ subgraph of $G$,                       */
   /* Hence $V'' \subseteq V$ and $E'' \subseteq E$              */
13 $T''$ = MST$(G''(V'', E''), W)$;
14 Output Post-process$(T'')$;
```

Algorithm 1: Sequential KMB algorithm

The first phase computes the shortest path distances between every pair of terminals of G (`for` loop at Line 3). This operation is achieved by running multiple SSSPs with source vertex as each terminal. Each SSSP populates two n-sized arrays: distance d and parent p. The d-array stores the shortest distance of each vertex from the source terminal, while p stores a parent in the SSSP tree (note that we say *a* parent rather than *the* parent, since there could be multiple shortest paths to a vertex; any one of them can be stored). For k SSSPs, a large $PArray$ of size $n \times k$ holds all the parents. The graph G' is smaller, with the terminals L forming its vertex set. For each pair of

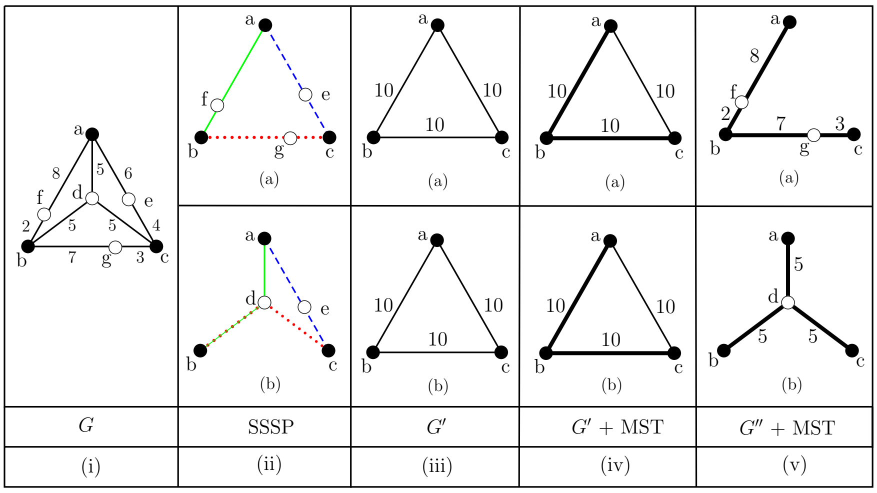

Fig. 2 Execution steps of KMB algorithm on an instance, where (•) dark vertices are terminals and (∘) light vertices are non-terminals.

terminals (u, v), weight of the edge u—v in G' is the shortest-path distance between (u, v) computed in the previous step (at Line 6).

The storage allocated for distance d and parent p arrays are reused across the SSSP function calls for different terminals. However, p array is copied into $PArray$, and the weights W' in the graph G' are computed incrementally using the distance array d. Since, we know G' is a complete graph of terminal vertices, bookkeeping W' is easier. The KMB algorithm now computes an MST T' of G' (Line 9). The last phase in the algorithm constructs a new graph G'' (`for` loop at Line 10). G'' is constructed from the MST T' by replacing each edge u—v in T' by a shortest $u - v$ path using the $PArray$ array. This crucial step ensures that G'' has all the terminals from G, all the *relevant* edges, as well as possibly some non-terminals from G, which are on the shortest path between two terminals. Finally, we compute an MST T'' of G''. T'' is post-processed to ensure that all the leaves are terminals, and then output as a Steiner tree of G (Line 13). Without this final step, the solution quality is affected marginally. Thus, there is a trade-off in spending time in this final step and the solution quality achieved. In our implementation, we omit this last step in favor of the execution time. It is well-known [29] that this Steiner tree is a 2-approximation to the optimum Steiner tree.

For a better understanding of the algorithm, we run the algorithm on an example as shown in Figure 2. It is sufficient for this explanation to refer to the upper part of Figure 2(a) alone. The three phases are shown as five steps for visualization purpose. The steps from left to right in Figure 2 (i, ii, iii, iv, v) show the resultant graphs at the end of individual steps.

Step (i) shows the initial graph G with three terminals represented as dark vertices (•) and four non-terminals as light vertices (∘). Step (ii) computes the shortest paths between every pair of terminals (a, b), (b, c) and (a, c) represented using thick, dotted and dashed lines respectively. Note that if there are multiple shortest paths, any one of them can be chosen. This ends phase one.

Step (iii) builds G' which is a complete graph on the terminal vertices, such that its edge-weights are its corresponding shortest-path distances from the previous step. G' happens to be a triangle with all edges-weights as 10. Step (iv) computes the MST on G' which results in (a, b) and (b, c) edges (shown as highlighted). This completes phase two.

Step (v) finally builds G'' using path vertices along the MST edges (a, b) and (b, c) resulting in $\{(a, f), (f, b)\}$ and $\{(b, g), (g, c)\}$. It then computes an MST on G''. Note that the final MST becomes unnecessary in our example as G'' is already a tree. The Steiner tree value is 20.

The lower (b) part of Figure 2 is an execution of the same KMB algorithm except at Step (ii) (b) where the shortest path for (b, c) via d is chosen, resulting in a Steiner tree for G whose value is 15 (incidentally, it happens to be the optimum). Suppose at Step (iv) (b) the MST edges were different {(a,b),(a,c)} or {(a,c),(b,c)} instead of {(a,b),(b,c)} then KMB will not result in an optimal value. It is due to this reason a parallel KMB algorithm may output a different Steiner tree compared to its sequential counterpart, or across different parallel runs.

3 Parallelizing the KMB Algorithm

We identify the challenges in parallelizing the KMB algorithm, and present the details of how we deal with them.

3.1 Challenges in Parallelization

Algorithmic Choice. KMB involves computing the shortest distance between every pair of terminals. This is done by computing the single-source shortest path (SSSP) from each terminal. A natural choice is to use Dijkstra's algorithm for SSSP, since it is work-efficient in the sequential setting [11]. However, in the parallel setting, it also suffers from low concurrency due to management of min-heap as the underlying data structure and due to not having enough independent work. The heap property needs to be maintained across iterations and updates, which poses hindrance to concurrent execution [30]. Hence, Dijkstra's algorithm is unsuited in the presence of thousands of threads on GPUs. In contrast, Bellman-Ford algorithm exhibits significantly better parallelism, as threads together simply relax (RELAX operation as in CLRS [11]) all the edges in every iteration. On the negative side, Bellman-Ford algorithm performs more work than Dijkstra's algorithm on the same graph. However, there is no maintenance of a complex data structure. Therefore, we choose to use Bellman-Ford algorithm for SSSP computation, with the expectation that the parallelism benefits will outweigh the extra work done (which happens in practice). Former research has also parallelized SSSP computation using variants of Bellman-Ford algorithm. This design choice illustrates the trade-off between work-efficiency and concurrency offered by the two algorithms.[1]

Early SSSP Halt. Without using the SSSP-halt, in general, an SSSP computation from a source s calculates the distance from s to all the remaining vertices. This is done also in the state-of-the-art implementations [4]. We observe that the KMB algorithm needs to find shortest paths between only the *pairs of terminals*. Therefore, the SSSP computation can be halted as soon as the distances of all the terminals from s are found, since it is only the terminals whose shortest path distances are required for building the graph G'. Implementing this early SSSP halt in the sequential setting can be achieved by exploiting the Dijkstra property along with the min-heap data structure. The Dijkstra property states that when a vertex is extracted from the min-heap, its distance is settled. Thus, as soon as all the terminals are extracted from the heap, the SSSP computation can be early-terminated (for that source). Interestingly, such a property is not satisfied by parallel Bellman-Ford algorithm (there is no min-heap). Therefore, we need to trade off this optimization if we need to exploit more concurrency. In our experiments, we observe that concurrency on GPU offers more benefits than exploiting this property on the CPU. As an optimized baseline, we enhance our sequential CPU implementation with this early SSSP halt technique. PACE challenge remarked [7] that any 2-approximation algorithm would require more than 30 minutes. This was indeed the case on the largest input instance in our setup for the KMB implementation of OGDF (which we call as JEA KMB). In contrast, our sequential CPU implementation terminates within 20 minutes due to early SSSP halt.

Avoiding MST Computation. The final step of Algorithm 1 is the computation of minimum spanning tree in G''. Although theoretically G'' can be arbitrary, empirically, we observe that G'' is often very sparse. Therefore, we can avoid MST computation altogether, if G'' is a tree. This can be quickly validated by checking if $|E''| = |V''| - 1$. In our graph-suite, 5 out of 14 test case did not require the last MST computation. To give a larger picture, 72 of the 100 public heuristic instances in the PACE challenge witness their G'' to be a tree.

Design Choice in Parallelizing. Overall KMB algorithm (Algorithm 1) can be parallelized in multiple ways. Figure 3(i) shows the computation flow for sequential KMB. It runs multiple SSSPs sequentially, followed by two invocations of MST. In inner-parallel KMB (Figure 3(ii)), multiple SSSPs are executed sequentially (as in the serial version), but each SSSP is internally parallelized on GPU. This allows us to take advantage of various advances in parallel graph algorithms [23, 33, 38].

[1] One may further explore algorithmic variants such as Δ-stepping, which may suit their setup.

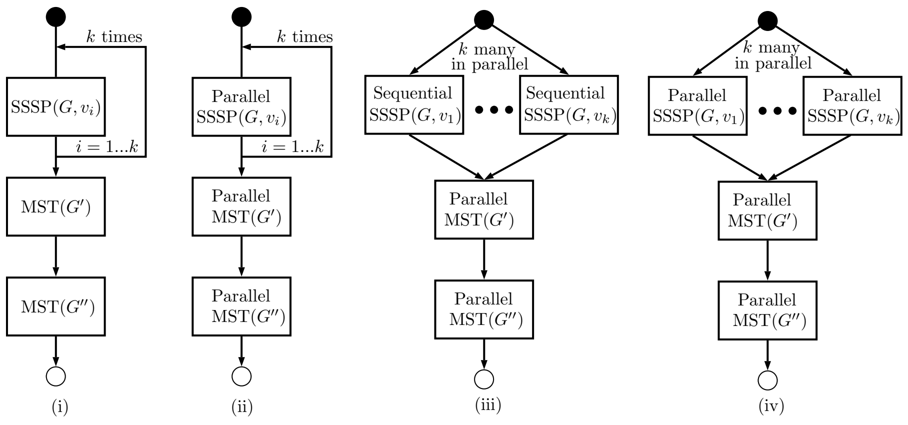

Fig. 3 Design choices in parallelizing the KMB algorithm (k is the number of terminals) (i) Sequential (ii) Inner-parallel (iii) Outer-parallel (iv) Both-parallel

In contrast, the outer-parallel KMB version (Figure 3(iii)) runs multiple SSSPs in parallel, but each SSSP internally is sequential. A distinct advantage of inner-parallel KMB (over the outer-parallel version) is reduced memory-requirement, as the auxiliary storage can be reused for the next SSSP. Further, each MST computation can be individually parallelized. This is aided by our careful algorithmic choice (e.g., Bellman-Ford). On the other hand, an outer-parallel design allows us to use Dijkstra's algorithm, and retain the advantage of early SSSP halt (as discussed earlier). Multiple SSSPs can be concurrently executed using CUDA streams on GPUs. There is also a possibility of both-parallel wherein p parallel SSSPs are executed simultaneously where each SSSP itself is parallel (Figure 3(iv)). Running multiple kernels in parallel, beyond a point, may necessitate reduction in the number of resources (such as registers) available per kernel. Further, fixed-point now needs to be across kernels, which demands modification to management of fixed-point flags, as well as handling synchronization issues across kernels. The storage requirement is also high, as the results of multiple SSSPs must be simultaneously stored. Therefore, although theoretically feasible, outer-parallel and both-parallel are ill-suited for KMB on GPUs. However, we have implemented both-parallel in such a way that the concurrent SSSPs can be fine-tuned without exceeding the memory limit on GPU. In our implementation, we use the inner-parallel approach and both-parallel (*cf.* Section 4.2.1).

Graph Creation and Representation. The input graph G does not undergo updates. We store it in the compressed sparse-row (CSR) format [23], which is amenable to GPU-based parallelization as well as CPU-GPU transfer. Graphs G' and G'' are created by the intermediate steps of the KMB algorithm. The vertex set of G' consists of all the terminals, which are also available after reading the input for G. However, the edge-weights W' for edges in G' are based on the shortest distances across terminals, which are computed using multiple SSSPs. In G, the vertices are conveniently numbered from 0 to $n-1$. However, G' poses a non-triviality (which we have not observed in other graphs algorithms): the terminal vertices could be arbitrary, and need not be numbered from 0 to $k-1$. The non-contiguous range of terminal identifiers creates issues while allotting threads to vertices as well as in memory allocation and mapping. We address this implementation issue by maintaining a mapping from terminals to a unique number from 0 to $k-1$ for MST on G'. Interestingly, a similar implementation challenge is posed by G'' as well, as it is a subgraph of G, having fewer than n vertices. Further, G'' needs to be built during algorithm execution (unlike static G constructed prior to running the algorithm) using the MST from G' and the shortest paths from SSSPs on G. To aid parallelism, we build both G' and G'' in the CSR format. On the other hand, unlike in G', the weights for G'' are the same as those in G, and hence need not be recomputed. Also, since those do not change dynamically, we can reuse weight information W for G'' as well. There is a small overhead involved in the construction of the CSR format for G' and G'', however it is negligible compared to the overall execution time.

Capturing Path Vertices. To construct G'', the KMB algorithm must store the vertices in the shortest path between each pair of terminals. This demands an *unknown* amount of storage on the GPU which becomes large for large graphs. In an extreme case, the graph is a path and nearly all the vertices are terminals. In such a case, $\Theta(n^2)$ space is needed for storing all the paths. Similarly, at the other extreme, if the graph is denser, $O(d \times n)$ space is required for the storage, where d is the graph diameter. In both the cases, we need either more space or additional work to estimate the storage. The shortest paths are data-dependent, and hence the amount of storage for storing paths cannot be computed a priori without additional overhead, other than some weak upper bounds. To address this issue, we make a design decision to not explicitly store the path vertices. Instead, we store only the parent array for the SSSP at each terminal, and construct the paths whenever needed. The paths can be created by traversing the parent for each vertex recursively up to the source terminal (from a node to its parent, and then to the parent's parent, and so on). This brings the storage cost to a deterministic one word per vertex, and allows easy storage management on the GPU. This is also sufficient for the KMB algorithm. It is noteworthy to mention that an all-pairs shortest path algorithm maybe used to perform phase one. However, to contain the space required with bare-minimum data structure, aiding faster KMB execution, we decided to use multiple SSSPs.

Updating Parent Array. Remembering the parent is crucial for finding the shortest path. However, updating the parent during a successful relax operation is challenging in the parallel setting due to a possible data race. When two or more threads relax a common neighbor, the neighbor's parent must be changed only for that thread which relaxed the distance to the minimum.

For instance, consider Figure 4. Let thread t_i operating from vertex u_i relax the distance of vertex v to some value, say 100. At the same time, let another

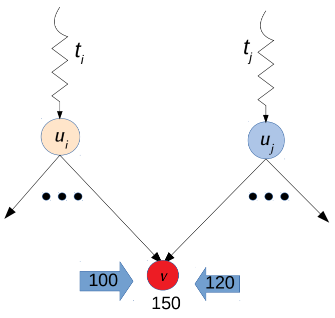

Fig. 4 Parent array data-race example

thread t_j operating from vertex u_j relax the distance of the same vertex v to some other value, say 120. Clearly, v's distance should be updated to 100 and its parent should be u_i. The distance update is achieved using *atomicMin*, but the two updates are not atomic (distance and parent). Hence, in absence of a safety mechanism, v's distance may be correctly set to 100, but due to the data race, the parent may be wrongly set to u_j. We discuss how we overcome this challenge in Section 4.1.

3.2 A Parallel Implementation of KMB

Incorporating various design choices discussed so far, we implement the GPU version of the KMB algorithm using parallel subroutines for SSSP and MST. A single SSSP computation from a terminal is parallel, while multiple SSSPs are executed sequentially (recall inner-parallel in Figure 3).

The graph is represented in the compressed sparse row (CSR) format, which is almost a standard practice on GPUs. The inputs to the SSSP are a source vertex (terminal) and the CSR arrays: meta array, adjacency list, and weight. The SSSP computation outputs two arrays, namely, distance d and parent p for every invocation from a terminal u. These arrays are allocated on the GPU of size equal to the number of vertices in G. The values of the distance and the parent arrays are copied to the host memory at the end of each iteration. More importantly, it is needed for computing the weight function and paths for building G' and G'' respectively. The storage space allocated for arrays d and p are reused on the GPU for the next invocation, thus saving space. As discussed earlier, the implementation cannot store complete paths due to unknown as well as arbitrarily large amount of memory requirement, as there are $O(k^2)$ paths to be remembered, (k is the number of terminals in G). We note that we require the paths information only for $k-1$ times, that is, the number of edges in the MST T'. Therefore, it is needed to trade off the time spent on computing the shortest path vertices using parents, instead of storing all the paths.

We present the parallel implementation of KMB in Algorithm 2, which at the high level, is similar to its sequential counterpart from Algorithm 1. From

```
   Input: Graph $G(V,E,W,L), W : E \rightarrow \mathbb{Z}_0^+$ and terminals $L \subseteq V$
   Output: A 2-approximate Steiner tree $T(V_T, E_T)$ such that $V_T \supseteq L$
1  for $u \in L$ do
2      parallel SSSP$(G, W, L, u)$;
3      Compute $W'$ incrementally;
4  end
5  $T'$ = parallel MST$(G', W')$;
6  Construct $G''$ using shortest paths;
7  $T''$ = parallel MST$(G'', W)$;
8  return $T''$;
```

Algorithm 2: Parallel KMB algorithm

the GPU implementation perspective, the main program along with its kernel are presented in Algorithms 3 and 4 respectively.

The SSSP kernel is launched (Line 3 of Algorithm 3) with n threads in total and more precisely, with at most 512 threads per thread-block (set after empirical tuning), where n is the number of nodes in G. For an SSSP from each terminal u, the corresponding arrays are defined as follows: d_u and p_u. Before every launch of the kernel, d_u and p_u arrays along with source terminal-vertex u have to be initialized. Here, *iterate* is the fixed-point variable. Every thread operates on a vertex u and performs the *relax* operations on its neighbouring vertices. The source vertex u's distance $d[u]$ is set to zero and rest as infinity during the initialization step. Relaxing of a neighbour is performed atomically using `atomicMin` CUDA function. During the relaxation, we also update the parent array p_u. Whenever a successful relax happens, we also update the fixed-point variable *iterate*. This variable is checked from the main program. If no vertex undergoes relaxation, then it means all the vertices from that source are saturated and have reached their final distances.

```
   Input: Graph $G(V,E,W,L)$ such that $W : E \rightarrow \mathbb{Z}_0^+$, source $u$
   Output: Distance $d[n]$ and Parent $p[n]$
1  repeat
2      INIT-KER<<<...>>>$(u, d_u, p_u, n)$;
3      SSSP-KER<<<...>>>$(.., u, d_u, p_u, iterate, n)$;
4      CopyTo$(DArray, d_u)$;        // Device → Host
5      CopyTo$(PArray, p_u)$;        // Device → Host
6      CopyTo$(hIterate, iterate)$;  // Device → Host
7  until not $hIterate$;
```

Algorithm 3: Custom SSSP Computation

3.3 Analysis of Parallel implementation

The correctness of our parallel KMB directly follows from the correctness of the KMB algorithm. One of the prominent ways to analyse a parallel algorithm is

```
tid = .. // compute tid. Also note $tid = u$;
if $tid < n$ then
    for $v \in adjacent[tid]$ do // Using CSR arrays visit the neighbours of $u$
        // RELAX OPERATION $(tid, v, d_u)$
        $newCost = d_u[v] + W(tid, v)$;
        $old = d_u[v]$;
        if $newCost < d_u[v]$ then
            ATOMIC-MIN$(d_u[v], newCost)$;
        end
        // Updates Parent array
        if $d_u[v] < old$ then
            // ATOMIC-MIN is success
            $p_u[v] = tid$;
            $iterate = true$;
        end
    end
end
```

Algorithm 4: SSSP-KER(..,u, d_u, p_u, $iterate, n$)

through the work-span framework [1]. The *work* of an algorithm corresponds to the total number of primitive operations performed by an algorithm, whereas the *span* corresponds to the longest sequence of dependencies in the computation. A parallel algorithm is said to be (asymptotically) work-efficient [1] if the work performed is asymptotically same as that of the time of the best-known sequential algorithm for the same problem. The total time of KMB depends on the number of terminals k, the time taken by SSSP, and that by MST. In particular, the total time is $O(k \times SSSP + 2 \times MST)$. In the sequential setting, the use of efficient algorithms results in $O(km \log n + k^2 \log k + m \log n)$ running time. A majority of the time is dominated by the first part of the closed-form expression, i.e., SSSP computation. In Table 1 we compare single-parallel Bellman-Ford (BF) algorithm with inner-parallel and both-parallel KMB. The work-complexity and span of single parallel Bellman-Ford (BF) algorithm is $O(mn)$ and $O(n \log n)$ respectively [1, 11]. However, in our case, we use inner-parallel (recall Figure 3(ii)) to perform k BF steps one after the another. Hence, both work and span get multiplied by a factor of k (second row in Table 1). On the other hand, theoretically, both-parallel is achieved by performing k BF steps in parallel for which the work is $k \times O(mn)$ whereas the span remains as $O(n \log n)$, same as the single BF algorithm (fourth row). Therefore, the maximum total parallelism (that is, $\frac{work}{span}$) possible using these parallelism approaches is $\Theta(km/\log n)$. For the inner-parallel approach, the possible parallelism is $\Theta(m/\log n)$ which is lesser than both-parallel. Performing 2 SSSP in parallel requires $k/2$ BF steps, which has parallelism of $\Theta(km/\log n)$, same as that of both-parallel (third row). Note that although highly concurrent, the both-parallel approach has a high memory requirement too, which is prohibitive for large graphs on GPUs.

Discussion. While there exist several studies optimizing parallelization of a single primitive computation (such as breadth-first search, shortest paths com-

	Algorithm type	Work	Span	#SeqSteps
SSSP	1× Bellman-Ford	$O(mn)$	$O(n \log n)$	-
KMB	k× Bellman-Ford (inner-parallel)	$k \times O(mn)$	$k \times O(n \log n)$	k
KMB	2× Bellman-Ford (both-parallel)	$2 \times O(mn)$	$O(n \log n)$	$k/2$
KMB	k× Bellman-Ford (both-parallel)	$k \times O(mn)$	$O(n \log n)$	1

Table 1 Complexity analysis of inner-parallel and both-parallel

putation, page rank and triangle counting), our work involves building upon such graph primitives to design an efficient higher-level algorithm implementation (similar to computation of betweenness centrality). Such algorithms not only pose different challenges, but also offer different possibilities towards parallelization (as discussed in the previous subsection). Our work is that way one step closer to parallelizing graph *applications*.

4 GPU KMB Optimizations

SSSP plays a central role in the KMB algorithm. Cumulatively, it is also the most compute-intensive and time-consuming step of KMB. Therefore, optimizing the SSSP computation is crucial towards an efficient Steiner tree computation. In this section, we describe various methods to optimize SSSP computations. They target three important aspects of the GPU-based processing: synchronization, computation, and memory.

4.1 Synchronization Optimizations

There are two ways of relaxing an edge: push-based and pull-based. In a push-based approach, the distance propagation is *sent* to the neighbouring nodes. Thus, each vertex u (in turn, each thread) updates the distances of its neighbours. Note that each thread processes the neighbours of its assigned vertex sequentially.

```
tid = .. // compute tid;
if tid < n then
    for v ∈ adjacent[tid] do // Using CSR arrays visit the neighbours of u
        // RELAX OPERATION (tid, v, d_u)
        newCost = d_u[u] + W(v, tid);
        if newCost < d_u[u] then
            d_u[u] = newCost ;
            p_u[u] = tid;
            iterate = true;
        end
    end
end
```

Algorithm 5: PULL-SSSP-KER(..,u, d_u, p_u, $iterate, n$)

In a pull-based approach, on the other hand, a vertex *receives* distance information from its neighbors and updates it locally as shown in Algorithm 5. In other words, each thread or a vertex u updates its own minimum distance $d[u]$ using the neighbors. Irrespective of push vs. pull approach, since the same vertex may be the source for one thread and target for another, there may be a read-write dependency. However, with the use of atomics (explicitly in the push-based and implicitly in the pull-based approach), algorithmic progress can be achieved leading to the fixed-point. As an additional note, in both pull-based and push-based approaches, all the vertices are processed until the fixed-point is reached which is called as topology-driven approach [38].

Parent Update. In a push-based approach, two or more vertices may update a common neighbour. So threads need to use `atomicMin` primitive to update neighbour's distances. KMB also needs the parent update. But as discussed earlier in Figure 4, this poses a challenge, as distance and parent updates *together* are not atomic. While this issue can be addressed using a double compare-and-swap (DCAS) or using a simulated lock mechanism on GPUs, there are portability or overhead issues respectively with the solutions. Pull-based approach offers a much more elegant solution. In the pull-based approach, no atomic instruction is used while updating distances (Line 7 in Algorithm 5). This occurs because the data of a vertex is being written by only one thread. Thus, updating the parent array also can be readily done by the same owner thread – without requiring a DCAS or a lock mechanism. Further, as we discuss shortly, we observe that the pull is more versatile than the push and allows other optimizations to be combined.

We would like to emphasize that this use-case (set of pros of pull over push) has not been highlighted before in the GPU literature on graph algorithms (to the best of our knowledge). In fact, push vs. pull has been projected as an incremental performance difference [25,44]. However in KMB, the performance implications of the difference get so much amplified that the pull-based approach is practically inevitable. Merrill et al. [37] do mention about parent update in the context of BFS, but in level-synchronous BFS, it does not matter which parent is recorded at a level out of the multiple possible candidates. Hence, the above issue does not arise. In contrast, SSSP necessitates that the distance and the *corresponding* parent must be rightly recorded. Even the state-of-the art SSSP for GPUs, Gunrock [47], has the same issue with updating predecessor/parent array because of the data race. The Gunrock developers mention this in the release notes of v1.0 [22] and list it as a known issue. The pull-based approach has overcome the challenge for us and we believe that can be applied for Gunrock's SSSP as well.

4.2 Computation Optimizations

We optimize KMB's computation by (i) using a p-SSSP (ii) a data-driven approach, (iii) computation unrolling, (iv) storing the last state of the computation, and (v) modeling the computation using edges.

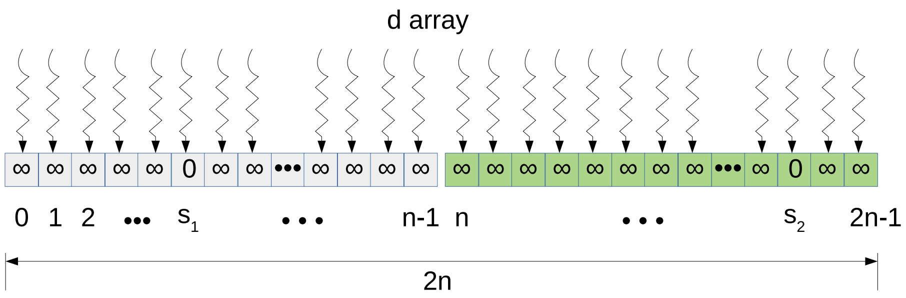

Fig. 5 Initialization of distance array d in double-barrel computation.

4.2.1 p-SSSP

Double-Barrel. While an individual SSSP is executed sequentially, we propose to perform multiple SSSPs also in parallel (*cf.* Figure 3 (iv)). Such a setup can utilize parallelism on both fronts, make judicious use of the available memory, and utilize threads better. We implemented this p-SSSP technique, which has the potential to improve the overall execution time. For exposition purpose, we present it below for running two SSSPs in parallel (that is, $p = 2$, hence the name Double-Barrel).

Consider two SSSPs from different sources, namely s_1 and s_2. To simultaneously accommodate distance and parent values from two SSSPs, we need to double the sizes of d and p arrays i.e $2n$. There is a slight modification in the initialization step: both $d[s_1]$ and $d[n + s_2]$ are set to zero as shown in Figure 5. No explicit modification is required to the existing kernel. The kernel is launched with $2n$ threads and executed till a fixed point. At end of fixed point, the entries 0 to $n - 1$ of the distance array d hold the source s_1's shortest-path distances, and n to $2n - 1$ hold the entries for s_2 as the source; similarly for parent array p. Two points to be noted about the Double-Barrel are (i) to take care of odd number of iterations, and (ii) that one SSSP may complete before the other.

Double-barrel computation can be extended to p-SSSP, which performs multiple concurrent SSSPs, leading to the both-parallel approach illustrated in Figure 3 (iv). Assume that each of the p SSSPs starting from p different sources is stored in an array $source[]$. We need pn-sized arrays for distance and parent. The initialization step involves setting $d[ni + sources[i]] = 0$ for $i \in [0, p)$. All the p SSSP computations read the common CSR array and modify their distance and parent chunks with a specific offset. Here again, the last iteration must be handled carefully to avoid array-index out-of-bound errors if the number of terminals is not a multiple of p.

We observe that p-SSSPs performed better than a pull-based SSSP in KMB (more in Section 5.4.2).

4.2.2 Data-driven Processing

So far, we discussed only a topology-driven approach, wherein each vertex is assigned to a thread. However, a data-driven processing may be efficient for certain irregular codes [38]. The data-driven approach improves work-efficiency because only those vertices whose distances were updated in the previous iteration are acted upon in the current iteration. This can significantly reduce the number of threads launched as well as the total amount of work done. A data-driven approach demands a worklist-based processing. Management of worklist incurs overhead as it is shared across all the active threads. Thus, the benefits due to work-efficient processing need to outweigh the worklist maintenance overheads. In our experiments with multiple SSSPs, across all the graphs, we observe that a data-driven approach is more expensive than all the CSR-based topology-driven approaches, except the edge-list based processing (more details in Section 5.4.2).

4.2.3 Controlled Computation Unrolling

For faster fixed-point computation, it is imperative to propagate an updated distance to its neighbors *early*. On the other hand, a lazy approach wherein a distance is not propagated immediately, may reap benefits by not propagating an intermediate value. With this definition, the typical topology-driven approach is lazy, as it awaits another iteration to propagate values from neighbors to the second level neighbors. For the best results, a balance between the two extremes (lazy or eager) is needed. Controlled computation unrolling achieves this by adding small eagerness into the traditional lazy topology-driven processing. In particular, when a vertex finds its distance updated, the corresponding thread propagates it to its neighbors *immediately*, without awaiting another kernel invocation. Thus, the second-level neighbors may receive their updated values faster, in the current iteration itself. Such a processing bears the potential of improving the overall information propagation in the graph, and compute the fixed-point faster. One may get more eager and propagate to the third-level neighbors, but note that this eager processing adds sequentiality to the individual thread processing. We identify three ways to perform computation unrolling:

- $\mathbf{\Delta^2}$: In this approach, the usual Δ loop going over the neighbors (`for`-loop at Line 3 in Algorithm 4) is replaced by a nested loop iterating also through the neighbors of neighbors. It resembles a push-within-push approach (Section 4.1). The work-done is Δ^2 where Δ is the max-degree of the graph.
- $\mathbf{2\Delta}$: A similar effect can be achieved using two sibling loops wherein, the first pulls min-distances from the neighbors, and the second pushes those to the neighbors. This resembles a pull-then-push or pull-push approach. The work-done is 2Δ.
- $\mathbf{t\Delta}$ or **t-pull**: In this approach, both the sibling loops can be either pull or push. In our setup, we rely on pull because it helps in overcoming the

parent update problem (as discussed in Section 4.2.2). The pull approach can be generalized to a t-pull.

In our experiments, we observe (in Section 5.4.2) that 2Δ and t-pull approaches outperform Δ^2. Further, a suitable t (typically, 3 or 4) outperforms the 2Δ approach, indicating a better balance between the eager and lazy processing. Controlled computation unrolling optimization also outperforms double-barrel SSSP optimization.

4.2.4 Memoization

To help accelerate the processing of the next iteration, the current iteration stores a small amount of information, in particular, the edge from the parent neighbor which set the current node's distance. This way, the next iteration can start from that edge and move forward (instead of always starting from the first edge). Note that the processing still needs to process all the neighbors, needing a wrap-around from the last to the first edge (resembling a circular queue). The performance of this optimization is similar that of pull-based approach.

4.2.5 Edge-based Processing

Topology-driven implementation can be implemented in a vertex-based or edge-based manner. In the latter, each thread operates on an edge or a group of edges. The advantage of edge-based processing is load-balance across threads, across warps, and across thread-blocks. On the flip side, edge-based processing *decouples* neighbors of the same vertex. Hence, certain optimizations (such as memoization) cannot be performed. Further, synchronization requirements may increase, since compared to a vertex-based processing, more threads operate on a vertex. In our experiments, we observe that the vertex-based processing outperforms the edge-based processing on CSR based topology-driven implementations.

4.3 Memory Optimizations

Programmable shared memory of a GPU helps in quick storing and reading of information. However, naïvely moving data to shared memory does not help – there must also be a reuse. Interestingly, our proposed $t\Delta$ approach has repeated usage of distance values. We exploit shared memory for three-level loop unrolling in 3Δ or 3-pull approach (Section 4.2.3). The three consecutive `for` loops run over the same adjacency list and the weights array, thrice. With 512 threads per block, we have at most 24 words to store per thread in shared memory such that the 48KB shared memory per thread block is not overflown. We store the CSR adjacency list into shared memory when the degree of that vertex is 24 or smaller. In our experiments, we observe that the

t#	**Graph**	n	m	k	**OPT** ($\times 10^6$)	**AvgD**	Δ	**AvgWt**	**MaxWt** ($\times 10^3$)
t1	T3-137	97,928	128,632	902	11.300	2.63	14	2,486	271
t2	T3-163	117,756	165,153	1879	13.391	2.81	16	1,938	271
t3	T3-181	135,543	201,803	3033	20.086	2.98	16	1,796	258
t4	T3-183	120,866	187,312	3224	24.998	3.10	9	2,349	271
t5	T3-185	66,048	110,491	3343	793.246	3.35	16	103,011	21,510
t6	T3-187	63,158	107,345	3458	863.275	3.40	9	138,645	53,890
t7	T3-189	172,687	255,825	3902	40.927	2.96	10	2,826	271
t8	T3-191	85,085	138,888	3954	977.020	3.26	9	91,279	33,666
t9	T3-193	17,127	27,352	4461	184.908	3.19	4	20	0.126
t10	T3-195	89,596	148,583	4991	1,406.041	3.32	10	127,022	25,865
t11	T3-197	235,686	366,093	6313	51.655	3.11	14	2,376	271
t12	lin37	38,418	71,657	172	0.099	3.73	4	56	0.198
t13	alue7080	34,479	55,494	2344	0.062	3.22	4	9	0.013
t14	Deezer-HR*	54,573	498,202	3000	?	18.26	420	1	0.001

Table 2 Benchmark graphs and their characteristics.
Note: AvgD is average degree. Δ is the maximum degree. Similarly, AvgWt and MaxWt are average edge-weight and maximum edge-weight respectively. *: For `t14`, given unit edge-weights and 3000 terminals were chosen randomly. `t14`'s OPT value is unknown.

3-pull with shared memory optimization performed significantly that all the other optimizations combined.

5 Experimental Evaluation

We now describe the experimental setup, results of our empirical study, and their analysis.

5.1 Experimental Setup

The CPU is an Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz with 64GB RAM running CentOS Linux release 7.5 using a GCC version 7.3.1 with `O3` optimization. The associated GPU is a Tesla P100 @ 1.33 GHz with 12GB global memory using CUDA 10.2 version. P100 has 3584 CUDA cores spread across 54 streaming multiprocessors.

Table 2 lists our graph suite. We have tested our implementation with the Parameterized Algorithms and Computational Experiments Challenge (PACE) largest public instances for Track 3 (heuristic) [6, 7]. While the graphs from PACE Challenge are derived from real-world applications and from DIMACS Challenges, to stress-test our implementation further, we use three real-world instances: two from SteinLIB [28] and one (Deezer-HR) from SNAP [31]. All the datasets a priori define the terminal set, except for the SNAP graph. So, we randomly select it for Deezer-HR, with unit edge-weights. One of our main motives for choosing this suite to study the behaviour of KMB on the largest instances (with respect to k, the number of terminals), few of the hardest

SteinLib instances (LIN and ALUE) and the high-degree instances (Deezer-HR). We have performed all the below experiments on the same CPU-GPU machine.

We compare using the following KMB implementations:

- Baseline-JEA: This is the sequential KMB implementation of open-source Open Graph Drawing Framework (OGDF). This original version is part of the published result from JEA [4]. We refer to this as JEA.
- Baseline-PACE: This is CIMAT team's code [17], the winner of the PACE Challenge heuristic track [6, 7]. We refer to this sequential version as PACE.
- KMBCPU: This is our sequential C++ implementation built upon Dijkstra's algorithm for SSSP, and optimized using the Early Halt technique (Section 3.1).
- KMBGPU: This is our parallel CUDA implementation, built using our Bellman-Ford SSSP and MST of [46]. This allows us to quantify the effect of our GPU optimizations.
- KMBGPU-OPT: This is our KMBGPU with GPU optimizations enabled.

Thus, we compare our implementations with the high quality codes available in open-source. Our sequential and parallel implementations are on par in solution quality and often perform considerably better in execution time than the baselines. PACE is an implementation of an evolutionary algorithm; hence we continued to use a time limit of 30 minutes, as set by the challenge organizers. All other implementations are deterministic and complete within the time limit except JEA on instance `t11`.

5.2 Effect on Execution Time

We evaluate our CPU and CUDA implementations against all the benchmark testcases in Table 2 and compare against JEA and PACE. The codes for JEA and PACE are executed and timed on the same CPU-GPU platform where KMPCPU is timed. Figure 6 presents the speedup of KMBCPU and KMBGPU-OPT over JEA, and that of KMBGPU-OPT over KMBCPU on each graph instance. Table 3 presents their absolute execution times. There is small overhead due to construction of G' and G'' and device to host memory transfer of arrays. Compared to the cumulative kernel execution time, this overhead is negligible (and included in our presented results). Thus, our presented GPU time includes the memory copy between host-device (in both directions) and kernel execution times including the overhead.

Our KMBGPU-OPT achieves a speedup up to 62× (average 20×) over JEA. It is also interesting to note that our KMBCPU outperforms JEA (average 4×). It is due primarily to the early SSSP-halt technique (Section 3.1). Note that JEA time-limit exceeded (TLE) on `t11` instance. Compared to KMBCPU, our KMBGPU-OPT achieves a speedup up to 27× (average 4×). Considering that these are irregular workloads from the challenge dataset, these

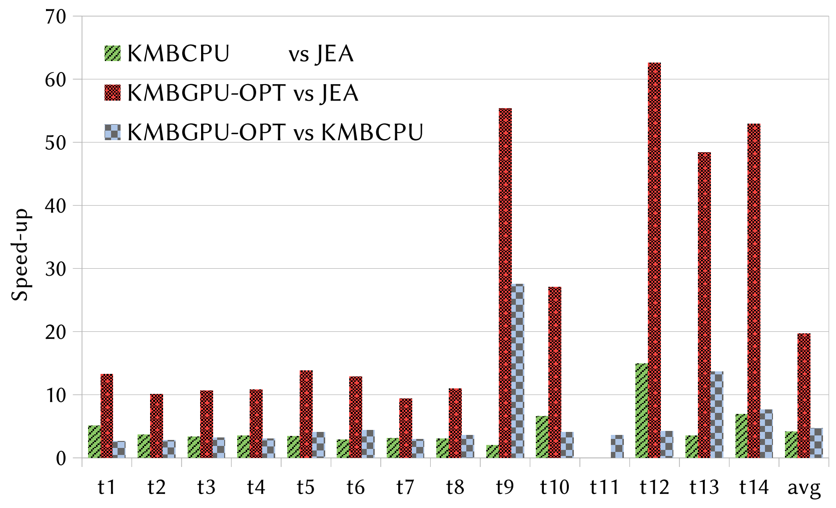

Fig. 6 Speedup comparisons of the implementations (higher is better). JEA timed-out on `t11`

t#	Time (in seconds)			
	KMBGPU-OPT		KMBCPU	JEA
		p in p-SSSP		
t1	10.1	16	26.6	134.6
t2	30.2	16	83.3	305.2
t3	58.5	8	186.5	622.9
t4	64.2	8	195.6	693.5
t5	29.5	4	119.6	407.7
t6	29.4	32	129.6	377.3
t7	118.0	8	352.9	1110.1
t8	57.0	32	204.4	624.3
t9	2.8	128	78.0	156.6
t10	68.7	64	281.1	1856.8
t11	321.4	8	1145.1	0.0
t12	0.4	128	1.9	27.9
t13	2.4	128	33.4	118.5
t14	10.5	64	80.3	558.1

Table 3 Absolute execution time (in seconds) of G: KMBGPU-OPT, C: KMBCPU and J:JEA. Note: PACE runs for a fixed 30 minutes time and JEA timed-out on `t11`

are significant results. Further analysis of KMBGPU-OPT reveals that at least $75 - 85\%$ of the total execution time is spent on SSSPs. The execution time is affected by a complex interplay of several factors such as the overall graph size, number of terminals, degree distribution, diameter over terminals[2], maximum weight, etc. But interestingly, these factors affect the algorithm's running time on CPU and GPU differently. For instance, maximum weight does not have

[2] Diameter is the maximum eccentricity. Eccentricity(v) is the largest distance from v to all the vertices in V, i.e., $d(v, V)$. Here, diameter over terminal means $\max d(v, L)$.

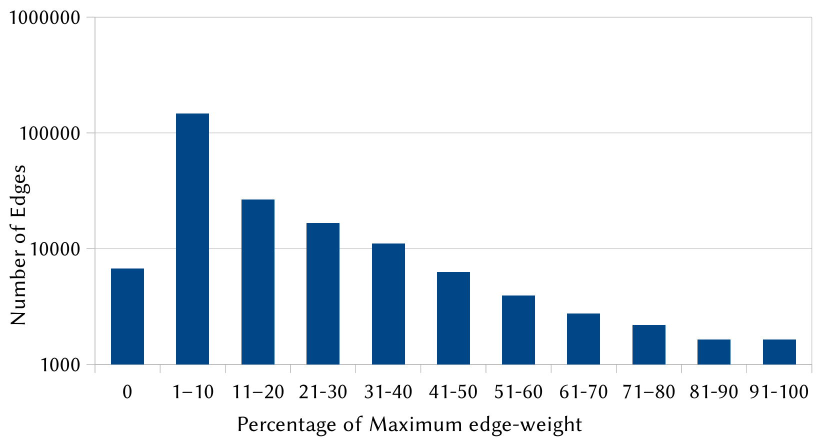

Fig. 7 Edge-weight distribution of graph `t11`. The x-axis denotes buckets of edge-weight values and y-axis denotes the number of edges having the edge-weights within a bucket (in logscale).

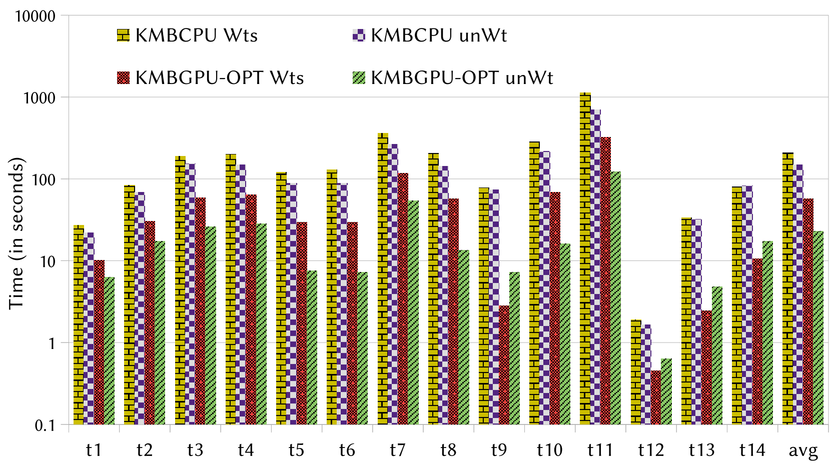

Fig. 8 Comparison of KMBCPU and KMBGPU with weighted edges and unweighted edges.

any effect on the CPU (due to Dijkstra's algorithm), whereas it has an adverse effect on the GPU because the SSSP kernel running Bellman-Ford's algorithm takes longer to reach the fixed-point. Thus, graphs having skewed weight distribution would have a larger impact on the GPU's Bellman-Ford. To understand this, we plot the edge-weight distribution of graph `t11` in Figure 7. The weights are distributed into ten equal-sized buckets (among the weights present) and one bucket for zero edge-weight. We observe similar skewed weight distribution also for other graphs (with only the change in absolute values).

t#	OPT O (10^3)	% deviation w.r.t. O				% deviation of G		
		G v O	**C v O**	**J v O**	**P v O**	**G v J**	**G v P**	**G v C**
t1	11,300	15.38	15.38	15.38	40.61	+0.01	-14.12	0.00
t2	13,391	12.57	12.58	12.58	37.55	0.00	-15.34	0.00
t3	20,086	14.79	14.78	14.78	32.19	+0.01	-12.03	+0.01
t4	24,998	13.79	13.77	13.77	29.45	+0.02	-10.58	+0.02
t5	793,246	1.40	1.40	1.40	2.51	0.00	-0.92	0.00
t6	863,275	0.72	0.72	0.72	3.03	0.00	-1.85	0.00
t7	40,927	17.21	17.20	17.20	33.79	+0.01	-11.42	+0.01
t8	977,020	1.26	1.26	1.26	2.21	0.00	-0.73	0.00
t9	184	7.33	6.95	7.03	7.11	+0.06	-0.09	+0.14
t10	1,406,041	1.29	1.29	1.29	6.12	0.00	-4.44	0.00
t11	51,655	12.76	12.75	–	31.69	–	-13.96	+0.01
t12	99	8.08	7.41	7.60	126.47	+0.50	-38.43	+0.68
t13	62	5.87	4.93	5.04	61.52	+0.72	-22.82	+0.82
t14	?	0.00	0.76	1.28	50.72	-1.30	-32.31	-0.80

Table 4 Comparison of solution quality (optimal value for t14 is not known) Note: Steiner values of G:KMBGPU-OPT, C: KMBCPU, J:JEA and P:PACE. An underline highlights which among G, C, J, and P computed the Steiner value *closest* to the OPT.

Max-weight vs performance. To understand if maximum edge-weight of a graph affects the CPU or GPU performance, we compare the time taken by KMBCPU and KMBGPU-OPT on `t1`–`t14` graphs with edge-weights and without edges-weight (i.e. unit edge weight). This is plotted in Figure 8. We know from Table 2 that the maximum weights for the graphs `t9, t12` and `t13` are under 200 and we also observe that the KMBCPU times were almost the same on weighted and unweighted graphs. However, for other instances, both the implementations were significantly faster on unweighted graphs than the weighted graphs.

5.3 Effect on Solution Quality

Table 4 presents optimal Steiner tree value and the deviation percentages. The column titled **O** lists the optimal value available along with the dataset (for all except t14). We have obtained the OPT values from PACE Challenge data [6] and SteinLib [28]. The rest of the columns list deviation percentage of the resultant Steiner tree value for KMBCPU, KMBGPU-OPT, JEA and PACE with respect to the optimum value (abbreviated with single letters as C, G, J and P for brevity). The *underlined* value indicates the best of the 4 variants. We observe that there is no one actual winner. Each one outperforms the other on different instances.

We recall that the sequential PACE and our parallel implementations may produce different outputs due to inherent non-determinism on two different invocations. To mitigate this effect, we run the codes thrice and use the *minimum* of the three Steiner values for both KMBGPU-OPT and PACE (KMBCPU and JEA are deterministic). Table 4 presents the optimal Steiner tree values and the deviation of the implementations (C, G, J and P) with respect to the optimum value **O**. We observe that our implementations (C and G) are always closer to J with nearly ±2% deviation. It is noteworthy to mention that the running times of C and G are significantly better than that of J (Section 5.2), and of course better than P as P has a fixed time-limit.

Table 4 is categorized into two sets: how is the deviation of G, C, J and P from the optimum, and how is G deviating from C, J and P. On `t14` we do not have the optimum so we choose the minimum among the five implementations as the optimum, to calculate the deviation (which happens to be G). On the first set, the underlined text denotes the variant closest to the optimum. On `t3`, both C and J are the closest. Of the 14 instances, G and J produced the best solution 5 times each; C produced it 6 times and P was always sub-optimal. The deviation of G, C and J over O is almost similar. However, it is noteworthy that P's solution is far away from the optimum, and more so on `t12`. In fact, it ends up producing a value more than twice that of the optimum! This is one of the hardest instances from SteinLib. The negative deviation means that the latter is far from the optimum. For example, **G vs J** on `t14` is negative which means than G is better and closer to optimum than J. Similarly, we observe that the difference between the Steiner tree values is at most ±1% between our sequential and the parallel versions (**G vs C**), and our parallel version over JEA (**G vs J**). Similarly, G is considerably better than P in terms of the solution quality. In addition, each of these values is within 2× of OPT for our codes C and G. We observe that in practice, the KMB implementation achieves much closer approximation to the optimal, computing Steiner trees that are weight-wise only 18% away from the optimal (within 0–18%, **G vs O**). This is very encouraging for real-world applications, as this means that the optimal exponential time algorithm can be substituted by a poly-time approximation algorithm and yet achieve a reasonably accurate result quickly.

5.4 Effects of Optimizations

As mentioned earlier in Section 4, the SSSP GPU optimizations are the key to KMBGPU-OPT's performance. We now discuss the effects GPU optimizations in detail.

5.4.1 Synchronization Optimizations

The push-based edge relaxation may result in common neighbor updates, thereby requiring atomic update to min-distance, as well as additional effort to update the parent. Performing both consistently and correctly is a challenge in a push-based variant. In contrast, the pull-based approach overcomes both the mentioned challenges because every node (in turn, thread) updates its own distance and its parent if the shortest path distance is improved. We observe that push and pull are performance-wise similar when computing the distances *alone*. However, to compute *both* the distance and the parent, pull is ∼2× faster (compared to a lock-based parent update).

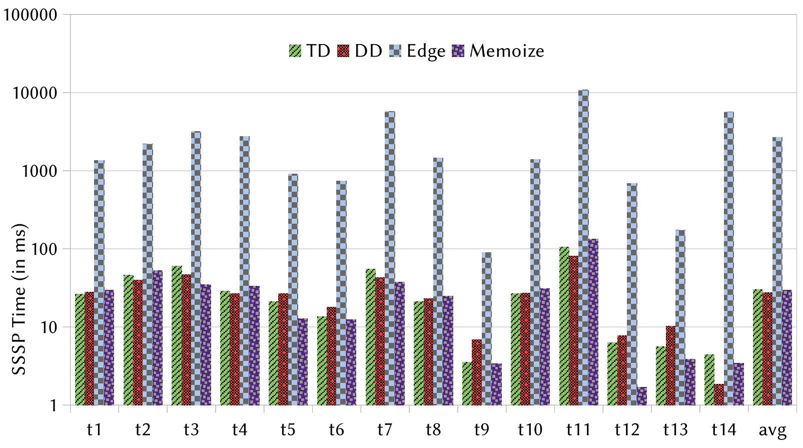

Fig. 9 Time comparison of Topology-driven, Data-driven and Edge-list based variants to compute SSSP distances with the first vertex as the source. Note that y-axis is in log-scale (Smaller is better).

5.4.2 Computation Optimizations

We present the effect of graph representation and style of computation (topology vs. data-driven) in Figures 9 and 10. Figure 9 compares the execution of an SSSP on various optimizations such as edge-based, data-driven, topology-driven, memoization, double-barrel and unrolling computation, whereas Figure 10 compares the number of iterations to reach fixed point for these optimizations. Number of iterations provides a high-level insight into these variants due to differences primarily in computation and synchronization.

Edge-based. In edge-based, m threads are launched and each thread relaxes an edge in both the directions (as the graph is undirected). Although the work-done by a thread is smaller compared to the vertex-centric processing, the number of iterations taken to reach the fixed-point is large. For instance on `t1`, edge-based processing takes ∼98K iterations to reach the fixed-point, while the vertex-centric approach (using push or pull) takes only 500. This is a general trend across all the graphs `t1--t14`. During a single iteration of the topology-driven (TD) processing, each thread performs Δ amount of work, whereas in edge-based processing, it is two units of work per thread. The propagation of distance information from source is faster in TD than the edge-based approach. A thread in TD visits neighbours sequentially and does more work compared to the corresponding edge-based processing. So, the latter requires more iterations to converge (Figure 9) and takes longer (Figure 10).

Data-driven (DD) and Topology-driven (TD). DD outperforms TD on average but significantly performs better than the edge-based approach. DD is primarily guided by the graph structure, which manifests itself as overheads during the worklist maintenance and slow distance propagation. For graph `t1`,

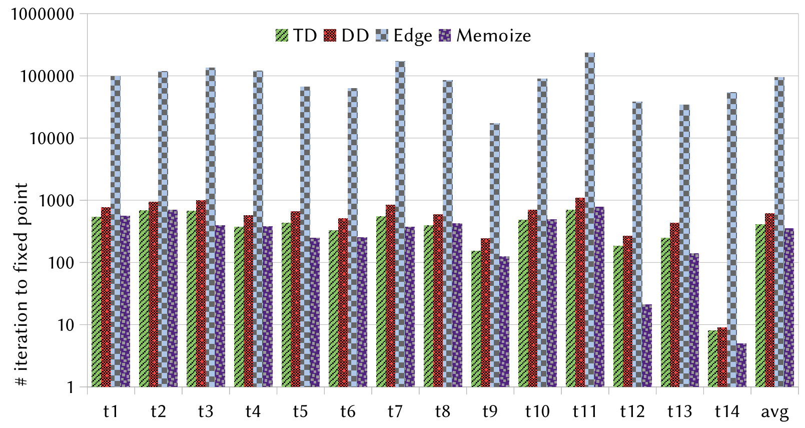

Fig. 10 Iterations in Topology-driven, Data-driven, Edge-based and Memoization based to compute SSSP distances with the first vertex as the source. (Smaller is better)

we observed about 33K atomic transactions for TD and only 21K for DD. This is expected because DD is work-efficient. However, despite a larger number of atomic instruction execution, TD and DD often show similar performance. Of the 14 instances, TD and DD outperform one another in half of the instances.

Although the average time (in Figure 9) for DD appear lesser than that of TD, the number of iterations is considerably larger than that of TD. More precisely, on an average, DD achieved 9% performance improvement over TD even with 50% more number of iterations than TD (in Figure 10). This indicates that the overheads of the worklist in DD outweigh the benefits due to reduced number of atomics executed. We note that this is surprising and in contrast to the earlier findings for simpler graph algorithms such as BFS [37].

Similarly, we observe TD outperforms edge-based variant by a large margin. On a different note, despite running SSSP to completion, TD-based parallel implementation beats the SSSP-halt optimized sequential CPU implementation convincingly (recall Table 3).

Memoization. Memoization achieves on an average 15% reduction in the number of iterations (refer Figure 10) over TD to reach fixed point. Interestingly, the execution times of memoization and TD were similar (Figure 9) with a small 2% average improvement. Memoization requires an additional n-sized array to keep track of the node's neighbours updated in the previous iteration.

Double-barrel and p-SSSP. Figure 11 presents the effect of SSSP variants towards implementing KMB. We observe that the double-barrel approach performed better than the basic 1-pull approach leading to an average 15% improvement. Interestingly, double-barrel performed similar in performance as that of 3-pull.

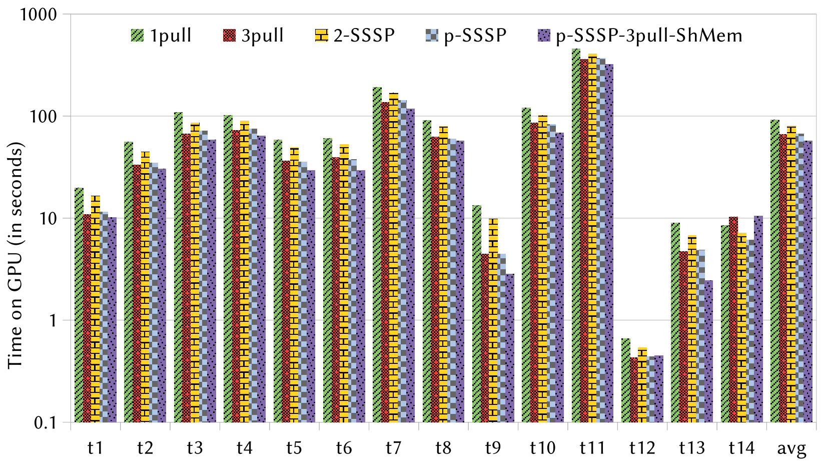

Fig. 11 Comparison of 1-Pull, 3-Pull, Double-barrel(2-SSSP) and p-SSSP with 3-Pull and shared memory (smaller is better). Note 1-Pull is KMBGPU whereas p-SSSP-3pull-ShMem is KMBGPU-OPT.

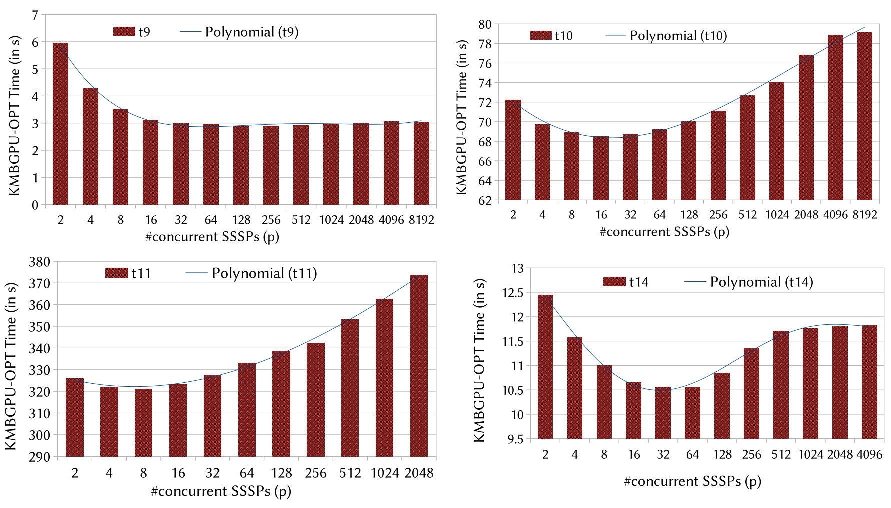

Fig. 12 KMBGPU with varying p-SSSP for the same graphs `t9`, `t10`, `t11`, and `t14` (Smaller is better). Note: On `t11` for $k = 4096$ and 8192, the memory limit was exceeded.

It is also noteworthy that this optimization performed best on `t14` compared to the other combination of optimizations.

The generalized form of double-barrel approach is p-SSSP, which runs p parallel SSSPs concurrently. We plot Figure 12 for four instances `t9, t10, t11` and `t14` with varying p from 2 to next power of 2 after the terminal size. p-SSSP performed well for fairly larger values of p except for values 4096 and 8192 on `t11`, for which the GPU memory limit is exceeded. For all the graphs, a similar U-Shaped curve is observed when varying p. There is no one particular p value that suits all the graph instances unlike the unroll factor in loop-unrolling. The best p on all instances achieves at least 15–200% improvement compared to double-barrel. On an average, the best p-SSSP performs 39% better than the single-pull version. The best values of p are also listed in Table 3. Selecting the best value of p for a given graph (without empirical tuning) would require further investigation, which we leave as a future work.

Unrolling Computation. From Figure 11, we observe that the t-pull accelerates the distance propagation than a single push or pull. More precisely, 3-pull bettered Δ^2 and 2Δ considerably. On an average, we see a 39% improvement in KMBGPU when $t = 3$ for our graph-suite over the single-pull. This is also reflected in Figure 11. On `t14`, the loop-unrolling did not help because in the unit-weight graphs, the possibility of betterment of a edge during relaxing step is less.

Due to the above observations, in SSSP, we prefer p-SSSP, $t\Delta$ and TD-based computation for performing further optimizations.

5.4.3 Memory Optimizations

The maximum streaming multiprocessor efficiency ranges up to 97% on our graph-suite. We applied the shared memory optimization for $t\Delta$ or 3-pull SSSP implementation. As the computations are irregular, there is less coalescing while accessing the adjacency list data. For instance, it incurs a 60% miss ratio in DRAM for `t1` (ranges $60 - 75\%$ for `t1-t14`). However, upon using shared memory, even a meagre 12% shared memory efficiency of an SSSP gave us a 25% performance improvement for `t1` compared to the implementation without shared memory. Figure 11 presents execution times of KMB on GPU highlighting different pull vs. shared memory optimizations. On few instances such as `t9, t12,` and `t14`, 3-pull optimization performs better than the 3-pull shared memory version. When the max-degree of the graph is very low (less than 5) or when the number of high-degree vertices is extremely large, we observe that the usefulness of shared memory optimization reduces. This occurs because for low degree vertices, the reuse is very low; while for graphs having several high-degree vertices, shared memory is insufficient. When the adjacency list fits well into shared memory, the advantage is pronounced. On `t14`, this optimization did not improve performance; in fact, it degraded it. This is because `t14` has a relatively larger average degree. Only 28% of vertices of `t14` have degree 24 or smaller, whereas for most of the other graphs CSR adjacency list fits inside the shared memory. This considerably helps in exploiting the thread block-local scratchpad memory. On an average, with 3-pull p-SSSP shared memory (i.e., KMBGPU-OPT), we achieve a 15% performance improvement over the 3-pull p-SSSP implementation without shared memory, and 60% improvement over the single pull-based implementation (Figure 11).

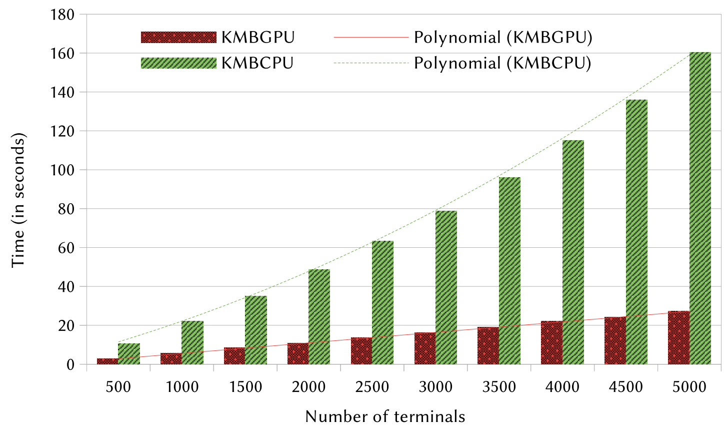

Fig. 13 Scalability plot on `t14` with increasing terminal size (lower is better)

5.5 Effect of Scaling

While several factors contribute to performance, undoubtedly, the number of terminals directly affects it as the number of SSSP invocations is decided by that factor. So, we use number of terminals to measure the scaling behavior of our implementations. We use `t14` as an example graph. To demonstrate scalability, we plot Figure 13 with increasing number of terminals on the x-axis and for the KMBGPU-OPT and KMBCPU times (in seconds) along the y-axis. The number of terminals is varied from 500 to 5000 with step value of 500. We observe that the KMBCPU execution time rises quickly and consumes 160.5 seconds for 5000 terminals. On the other hand, the KMBGPU-OPT implementation scales better, and completes within 30 seconds for 5000 terminals.

Figure 14 shows the time T spent per terminal in KMBGPU and KMBCPU. We observe that T_{gpu}/k value remains stable on the GPU even on increasing the terminal size. In contrast, CPUs' T_{cpu}/k increases linearly. This shows the versatility and scalability of the GPU implementation.

6 Related Work

There were early efforts to design parallel algorithms for matrix based problems [14]. Since graphs are represented using adjacency matrix or incidence matrix, parallel algorithms for recognition problems on graph classes [42] and computation of properties related to connectedness and matchings [41] have been extensively studied. More recently, efficient and scalable implementations on GPUs have been considered for a wide variety of graph algorithms: Breadth First Search, Depth First Search, Minimum Spanning Tree, Single Source Shortest Path, All Pairs Shortest Path, Triangle Counting, Betweenness Centrality, to name a few. Details can be found in several papers [3,23,33,37,38,40,46].

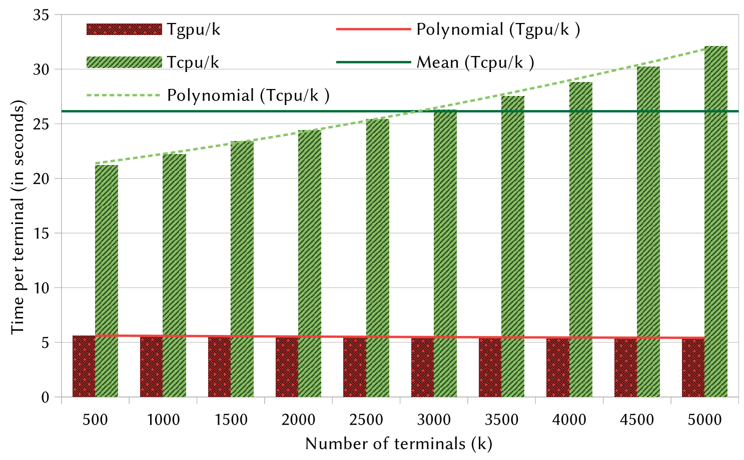

Fig. 14 Time per terminal on `t14` with increasing terminal size (lower is better)

NP-Complete Problems. GPU implementations of algorithms for NP-Complete problems is a relatively recent line of research. Given that the only approach to exactly solve NP-Complete problems is an exhaustive search of the solution-space in the worst-case (unless we have non-trivial understanding of NP-Complete problems), the prognosis for efficient parallel implementations is naturally weak. However, recent research by Dhulipala, Blelloch and Shun [15] has shown that there is a significant promise in considering good algorithms which have a sound theoretical analysis for fast and scalable implementations. Specifically, they mention that definitely adversarial inputs cannot be given to make such implementations fail as there is already a worst-case analysis of the algorithm that has been implemented. The first effort in this direction is due to van der Zanden and Bodlaender [48], who came up with a GPU implementation of exact algorithms for computing treewidth. Treewidth of a graph is the minimum width among all possible tree decompositions of a graph. This NP-hard problem's implementation also coincided with the PACE Challenge in 2016 which also had a track for exact parallel implementations and a track for heuristic parallel implementations of computing treewidth.

The 11th DIMACS Implementation Challenge [26] on the STP had many challenge categories for different STP variants. The winner in many of the categories was team StayNerd [18, 35] who had implementations of both the exact algorithms and the heuristics. Their solver solved 4 different variants of STP including a generalized version called prize-collecting Steiner tree problem (PCSTP). PCSTP has node-weights along with minimizing edge-weight requirement for the resultant Steiner tree. Their CPU implementation achieved significant speed and solution quality by heuristic-based local branching and local search techniques along with heuristic based initialization and prepossessing methods. Staynerd code involves mixed integer linear-programming (ILP) approach for solving instances. It has filter mechanism to automatically choose the best algorithmic strategy based on the input instances. In contrast, ours is a deterministic constant-approximation implementation.

Apart from the DIMACS dataset for the STP, SteinLib [27,28] is a collection of instances of the STP ranging from synthetic to real-world along with their characteristics such as origin and optimum value.

Approximate and Parallel Approaches to STP. Given its practical importance and its fundamental nature, STP and many of its variants including those on planar graphs [45] and strongly chordal graphs [13] have been extensively researched [24]. Starting with the KMB algorithm and Mehlhorn's faster implementation for KMB algorithm [36], up to the most recent approximation algorithm due to Beyer and Chimani [4], there have been several improved approximation algorithms. Many of these algorithms rely on linear programming. Parallel algorithms for the STP have far fewer results than the number of approximation algorithms. Park et al. [39] designed a parallel implementation of a classical 2-approximation algorithm, and discussed the challenges in parallelizing the KMB algorithm. Makki, Been and Pissinou [32] designed a parallel implementation of the Rayward-Smith's algorithm. A recent survey [5] talks about the different approaches in the design of parallel algorithms for the STP. These ideas are for multi-core processing and do not readily translate to GPU based parallelization.

GPU implementations for the STP are relatively very recent, and we are aware of only two such works. Chow et al. [10] had a GPU implementation for Rectilinear Steiner tree problem (which is a specialized version of STP). Also, a GPU implementation due to Mathieu and Klusch [34] provides a good speedup over CPU implementations.

Chow et al. work is on obstacle-avoiding rectilinear Steiner minimum tree (OARSMT) problem in VLSI physical design. Given a set of pins and rectangular obstacles on 2D a plane. The objective is to find a rectilinear Steiner tree of minimum length connecting all pins. The pins are the terminals in our setup. The setup does not define the Steiner node explicitly. The edge-weight between two vertices in STP is the Manhattan distance between pins. The authors define the shortest path region between every two pin and compute maze routing algorithm on it, and finally an MST. The routing algorithm is run after creating an escape graph (a simplified Hanan grid) and all the Manhattan shortest distance paths between every pair of pins. Every turn in the path is a Steiner point. Connecting the shortest path regions is a key step done in parallel which is analogous to finding the shortest path between terminals in the STP setting.

Mathieu and Klusch's heuristic-based GPU implementation relies on a local search approach. They take an initial tree and separate it at the degree 3 nodes to create set of paths called loose-paths. Then, they find a newer least weighted path to merge the two different loose-paths in parallel (which resembles multiple sources and multiple destinations). If this step improves (a newer shorter path between two loose-paths) the weight of the current target then the solution is updated; otherwise, a different pair of loose-path is sought in the tree. They primarily discuss the speedup when the number of terminals changes (e.g., $k = 4$ and $k = 128$), but do not discuss the solution quality. We could not obtain their code to compare with ours quantitatively nor it is available public. Our algorithm is deterministic with proven approximation guarantee on the solution. It works on the largest of the graph instances where the terminals are in thousands. The authors mentioned good speedup over CPU implementation only for instances with more than 10^4 edges. However, in our case, speedup is substantial for a variety of graph instances.

7 Conclusion and Future Work

We have devised an efficient GPU implementation of the KMB algorithm by addressing challenges related to the graph representation, choice of parallelism, as well algorithm-specific GPU optimizations. Our CPU implementation is an improvement over the OGDF's KMB implementation, and it is on par with the PACE winner on the suite instances. Based on its performance on a suite of graphs from competitive challenges as well as other real-world graphs, we conclude that the implementation provides a significant improvement of execution time over the sequential versions (JEA, PACE, and our KMBCPU). In addition, the solution quality is much closer to the optimal than the analytical guarantee of $2\times$ of the KMB algorithm. Our KMBCPU runs faster and achieves speedup up to $15\times$ over JEA. Our KMBGPU-OPT achieves significant speedup up to $62\times$ over JEA. We believe that our ideas can significantly speedup other local-search STP implementations. Further, pull-based processing is often considered as an alternative, but STP benefits so much from it, that it is almost inevitable to use it. We also strongly believe that the parallelization strategies and GPU optimization techniques developed in this work can be applied to other graph problems wherein the algorithm involves multiple basic-parallel algorithms within its sub-routines.

Acknowledgements

We thank the anonymous reviewers for suggesting to implement p-SSSP. This has resulted in improved performance. We thank PACE 2018 PC members Florian Sikora and Édouard Bonnet for sharing the private instances of the challenge after the results and `optil.io` point of contact Jan Badura for enabling us to submit on `optil.io` post PACE related to this publication. Thanks to Stephan Beyer for rightly pointing us to use OGDF for the source codes of their JEA paper.

References

1. Acar, U.A., Blelloch, G.E.: Algorithms: Parallel and Sequential. Online-Draft (2019). `http:www.algorithms-book.com`

2. Agrawal, A., Klein, P.N., Ravi, R.: When trees collide: An approximation algorithm for the generalized steiner problem on networks. SIAM Journal on Computing **24**(3), 440–456 (1995). URL `https://doi.org/10.1137/S0097539792236237`
3. Barnat, J., Bauch, P., Brim, L., Jr., M.C.: Computing strongly connected components in parallel on CUDA. In: 25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings, pp. 544–555 (2011). DOI 10.1109/IPDPS.2011.59. URL `https://doi.org/10.1109/IPDPS.2011.59`
4. Beyer, S., Chimani, M.: Strong steiner tree approximations in practice. Journal of Experimental Algorithmics (JEA) **24**(1), 1.7:1–1.7:33 (2019). URL `http://doi.acm.org/10.1145/3299903`
5. Bezensek, M., Robic, B.: A survey of parallel and distributed algorithms for the steiner tree problem. International Journal of Parallel Programming **42**(2), 287–319 (2014). URL `https://doi.org/10.1007/s10766-013-0243-z`
6. Bonnet, É., Sikora, F.: The 3rd Parameterized Algorithms and Computational Experiments Challenge. `https://pacechallenge.org/2018/` (2018). [Online; accessed 27-Mar-2019]
7. Bonnet, É., Sikora, F.: The PACE 2018 Parameterized Algorithms and Computational Experiments Challenge: The Third Iteration. In: C. Paul, M. Pilipczuk (eds.) 13th International Symposium on Parameterized and Exact Computation (IPEC 2018), *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 115, pp. 26:1–26:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). DOI 10.4230/LIPIcs.IPEC.2018.26. URL `http://drops.dagstuhl.de/opus/volltexte/2019/10227`
8. Borradaile, G., Klein, P.N., Mathieu, C.: An $O(n \log n)$ approximation scheme for steiner tree in planar graphs. ACM Trans. Algorithms **5**(3), 31:1–31:31 (2009). URL `https://doi.org/10.1145/1541885.1541892`
9. Chimani, M., Gutwenger, C., Jünger, M., Klau, G.W., Klein, K., Mutzel, P.: The open graph drawing framework (OGDF). In: R. Tamassia (ed.) Handbook on Graph Drawing and Visualization, pp. 543–569. Chapman and Hall/CRC (2013)
10. Chow, W., Li, L., Young, E.F.Y., Sham, C.: Obstacle-avoiding rectilinear steiner tree construction in sequential and parallel approach. Integration **47**(1), 105–114 (2014). URL `https://doi.org/10.1016/j.vlsi.2013.08.001`
11. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd Edition. MIT Press (2009). URL `http://mitpress.mit.edu/books/introduction-algorithms`
12. Cygan, M., Fomin, F.V., Kowalik, L., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., Saurabh, S.: Parameterized Algorithms, 1st edn. Springer Publishing Company, Incorporated (2015)
13. Dahlhaus, E.: A Parallel Algorithm for Computing Steiner Trees in Strongly Chordal Graphs. Discrete Applied Mathematics **51**(1-2), 47–61 (1994). URL `https://doi.org/10.1016/0166-218X(94)90093-0`
14. Dekel, E., Nassimi, D., Sahni, S.: Parallel matrix and graph algorithms. SIAMJournal on Computing **10**(4), 657–675 (1981). URL `https://doi.org/10.1137/0210049`
15. Dhulipala, L., Blelloch, G.E., Shun, J.: Theoretically efficient parallel graph algorithms can be fast and scalable. In: Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018, pp. 393–404 (2018). URL `https://doi.org/10.1145/3210377.3210414`
16. Dreyfus, S.E., Wagner, R.A.: The Steiner Problem in Graphs. Networks **1**(3), 195–207 (1971)
17. Emmanuel Romero Ruiz and Emmanuel Antonio Cuevas and Irwin EnriqueVillalobos López and and Carlos Segura González: (CIMAT Team from the Center for Researchin Mathematics, Guanajuato). `https://github.com/HeathcliffAC/SteinerTreeProblem` (2018). [Online; accessed 21-April-2020]
18. Fischetti, M., Leitner, M., Ljubic, I., Luipersbeck, M., Monaci, M., Resch, M., Salvagnin, D., Sinnl, M.: Thinning out steiner trees: a node-based model for uniform edge costs. Math. Program. Comput. **9**(2), 203–229 (2017). DOI 10.1007/s12532-016-0111-0. URL `https://doi.org/10.1007/s12532-016-0111-0`

19. Frank K. Hwang and Dana S. Richards and Pawel Winter: Monograph. In: The Steiner Tree Problem, *Annals of Discrete Mathematics*, vol. 53. Elsevier (1992). DOI https://doi.org/10.1016/S0167-5060(08)70188-2. URL `http://www.sciencedirect.com/science/article/pii/S0167506008701894`
20. Fuchs, B., Kern, W., Molle, D., Richter, S., Rossmanith, P., Wang, X.: Dynamic Programming for Minimum Steiner Trees. Theor. Comp. Sys. **41**(3), 493–500 (2007). URL `http://dx.doi.org/10.1007/s00224-007-1324-4`
21. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990)
22. Gunrock Developers: (CIMAT Team from the Center for Researchin Mathematics, Guanajuato). `https://github.com/gunrock/gunrock/releases/tag/v1.0` (2019). [Online; accessed 21-April-2020]
23. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the GPU using CUDA. In: High Performance Computing - HiPC 2007, 14th International Conference, Goa, India, December 18-21, 2007, Proceedings, pp. 197–208 (2007). DOI 10.1007/978-3-540-77220-0_21. URL `https://doi.org/10.1007/978-3-540-77220-0_21`
24. Hauptmann, M., Karpinski, M.: A Compendium on Steiner Tree Problems. `http://theory.cs.uni-bonn.de/info5/steinerkompendium/` (2015). URL `http://theory.cs.uni-bonn.de/info5/steinerkompendium/netcompendium.pdf`. [Online; accessed 27-Dec-2020]
25. Jia, Z., Kwon, Y., Shipman, G.M., McCormick, P.S., Erez, M., Aiken, A.: A distributed multi-gpu system for fast graph processing. Proc. VLDB Endow. **11**(3), 297–310 (2017). DOI 10.14778/3157794.3157799. URL `http://www.vldb.org/pvldb/vol11/p297-jia.pdf`
26. Johnson, D.S., Koch, T., Werneck, R.F., Zachariasen, M.: 11th DIMACS Implementation Challenge in Collaboration with ICERM: Steiner Tree Problems. `http://dimacs11.zib.de/home.html` (2014). [Online; accessed 27-Mar-2019]
27. Koch, T., Martin, A., Voß, S.: SteinLib: An Updated Library on Steiner Tree Problems in Graphs, pp. 285–325. Springer US (2001). URL `https://doi.org/10.1007/978-1-4613-0255-1_9`
28. Koch, T., Martin, A., Voß, S.: Steinlib testdata library (2015). URL `http://steinlib.zib.de/steinlib.php`. Online; Accessed 27-March-2019
29. Kou, L., Markowsky, G., Berman, L.: A fast algorithm for Steiner trees. Acta Informatica **15**(2), 141–145 (1981). DOI 10.1007/BF00288961. URL `https://doi.org/10.1007/BF00288961`
30. Lenharth, A., Nguyen, D., Pingali, K.: Priority queues are not good concurrent priority schedulers. In: J.L. Träff, S. Hunold, F. Versaci (eds.) Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings, *Lecture Notes in Computer Science*, vol. 9233, pp. 209–221. Springer (2015). URL `https://doi.org/10.1007/978-3-662-48096-0_17`
31. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data` (2014)
32. Makki, K., Been, K., Pissinou, N.: A Parallel Algorithm for the Steiner Tree Problem. In: Computing and Information - ICCI'93, Fifth International Conference on Computing and Information, Sudbury, Ontario, Canada, May 27-29, 1993, Proceedings, pp. 380–384 (1993)
33. Martín, P.J., Torres, R., Gavilanes, A.: CUDA solutions for the SSSP problem. In: Computational Science - ICCS 2009, 9th International Conference, Baton Rouge, LA, USA, May 25-27, 2009, Proceedings, Part I, pp. 904–913 (2009). URL `https://doi.org/10.1007/978-3-642-01970-8_91`
34. Mathieu, C., Klusch, M.: Accelerated steiner tree problem solving on GPU with CUDA. In: Algorithms and Architectures for Parallel Processing - 15th International Conference, ICA3PP 2015, Zhangjiajie, China, November 18-20, 2015, Proceedings, Part II, pp. 444–457 (2015). DOI 10.1007/978-3-319-27122-4_31. URL `https://doi.org/10.1007/978-3-319-27122-4_31`
35. Matteo Fischetti, Markus Leitner, Ivana Ljubic, Martin Luipersbeck, Michele Monaci, Max Resch, Domenico Salvagnin, Markus Sinnl : Approximate the steiner tree using the KMB heuristic. `https://homepage.univie.ac.at/ivana.ljubic/research/staynerd/StayNerd.html` (2015). [Online; accessed 21-April-2019]

36. Mehlhorn, K.: A faster approximation algorithm for the steiner problem in graphs. Inf. Process. Lett. **27**(3), 125–128 (1988). URL `https://doi.org/10.1016/0020-0190(88)90066-X`
37. Merrill, D., Garland, M., Grimshaw, A.S.: Scalable GPU graph traversal. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012, pp. 117–128 (2012). URL `https://doi.org/10.1145/2145816.2145832`
38. Nasre, R., Burtscher, M., Pingali, K.: Data-driven versus topology-driven irregular computations on gpus. In: 27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013, pp. 463–474 (2013). DOI 10.1109/IPDPS.2013.28. URL `https://doi.org/10.1109/IPDPS.2013.28`
39. Park, J., Ro, W.W., Lee, H., Park, N.: Parallel Algorithms for Steiner Tree Problem. In: Third International Conference on Convergence and Hybrid Information Technology, vol. 1, pp. 453–455 (2008). DOI 10.1109/ICCIT.2008.167
40. Pawan Harish, V.V., Narayanan, P.J.: Large Graph Algorithms for Massively Multithreaded Architectures. In: (Technical Report), IIIT-H, IIIT/TR/2009/74 (2009). URL `https://researchweb.iiit.ac.in/~harishpk/PDF/TR-2009_74.pdf`
41. Quinn, M.J., Deo, N.: Parallel graph algorithms. ACM Computing Surveys (CSUR) **16**(3), 319–348 (1984). URL `https://doi.org/10.1145/2514.2515`
42. Ramalingam, G., Rangan, C.P.: New sequential and parallel algorithms for interval graph recognition. Inf. Process. Lett. **34**(4), 215–219 (1990). URL `https://doi.org/10.1016/0020-0190(90)90163-R`
43. Robins, G., Zelikovsky, A.: Improved steiner tree approximation in graphs. In: Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA., pp. 770–779 (2000). URL `http://dl.acm.org/citation.cfm?id=338219.338638`
44. Salvador, G., Darvin, W.H., Huzaifa, M., Alsop, J., Sinclair, M.D., Adve, S.V.: Specializing coherence, consistency, and push/pull for GPU graph analytics. In: IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2020, Boston, MA, USA, August 23-25, 2020, pp. 123–125. IEEE (2020). URL `https://doi.org/10.1109/ISPASS48437.2020.00027`
45. Suzuki, H., Yamanaka, C., Nishizeki, T.: Parallel Algorithms for Finding Steiner Forests in Planar Graphs. In: Algorithms, International Symposium SIGAL '90, Tokyo, Japan, August 16-18, 1990, Proceedings, pp. 458–467 (1990). DOI 10.1007/3-540-52921-7_95. URL `https://doi.org/10.1007/3-540-52921-7_95`
46. Vineet, V., Harish, P., Patidar, S., Narayanan, P.J.: Fast minimum spanning tree for large graphs on the GPU. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on High Performance Graphics 2009, New Orleans, Louisiana, USA, August 1-3, 2009, pp. 167–171 (2009). DOI 10.2312/EGGH/HPG09/167-172. URL `https://doi.org/10.2312/EGGH/HPG09/167-172`
47. Wang, Y., Pan, Y., Davidson, A.A., Wu, Y., Yang, C., Wang, L., Osama, M., Yuan, C., Liu, W., Riffel, A.T., Owens, J.D.: Gunrock: GPU graph analytics. ACM Trans. Parallel Comput. **4**(1), 3:1–3:49 (2017). URL `https://doi.org/10.1145/3108140`
48. van der Zanden, T.C., Bodlaender, H.L.: Computing treewidth on the GPU. In: 12th International Symposium on Parameterized and Exact Computation, IPEC 2017, September 6-8, 2017, Vienna, Austria, pp. 29:1–29:13 (2017). DOI 10.4230/LIPIcs.IPEC.2017.29. URL `https://doi.org/10.4230/LIPIcs.IPEC.2017.29`
49. Zelikovsky, A.: An 11/6-approximation algorithm for the network steiner problem. Algorithmica **9**(5), 463–470 (1993). DOI 10.1007/BF01187035. URL `https://doi.org/10.1007/BF01187035`