2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-8711-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/IPDPS57955.2024.00022

AMST: Accelerating Large-Scale Graph Minimum Spanning Tree Computation on FPGA

Haishuang Fan[a,b], Rui Meng[a,b], Qichu Sun[a,b], Jingya Wu[a,*], Wenyan Lu[a,c], Xiaowei Li[a,b], Guihai Yan[a,b,c,*]
[a] State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences
[b] University of Chinese Academy of Sciences
[c] YUSUR Technology Co., Ltd.
{fanhaishuang20z, mengrui22s, sunqichu22z, jingyawu, luwenyan, lxw, yan}@ict.ac.cn; *corresponding author

***Abstract*—The minimum spanning tree (MST) plays an important role in variant fields, such as chip design and network analysis. With the rapid expansion of vertices in real-life graphs, the bottleneck problem of MST algorithms in large-scale graphs grows more prominent. While there have been many FPGA-based accelerators for large-scale graph algorithms such as Graph Random Walk, and various algorithms to accelerate MST on CPUs and GPUs, effectively implementing MST algorithms for large-scale graphs on FPGAs remains quite challenging. This is due to several reasons: ① The neighbor vertices in the graph require extensive random memory access and the memory access characteristics vary across different stages and iterations. ② There are a large number of useless computations due to the existence of internal edges within a component (intra-edge). ③ Parallel MST algorithm suffers from significant communication overhead due to the minimum edge data update conflicts and memory read-write conflicts.**

This paper proposes AMST to accelerate large-scale graph MST computation on FPGA. First, AMST employs a customized hash-based high-degree vertex cache (HDC) to improve memory access efficiency. Second, AMST adopts a graph pruning strategy that skips intra-edge and sorts edges by weight to eliminate useless computation and memory access. Finally, AMST utilizes a sorting networking module and a multi-port HDC to improve parallel efficiency. The experimental results demonstrate that AMST achieves an average performance speedup of $17.52\times$ over CPU and $1.89\times$ over GPU, as well as $74.96\times$ over CPU and $10.45\times$ over GPU on energy efficiency.

***Index Terms*—Graph Accelerator, Minimum Spanning Tree, FPGA**

I. Introduction

The Minimum Spanning Tree (MST) computation aims to find a tree that connects every vertex in a graph, such that the total weight of the tree's edges is minimized. MST is one of the basic graph algorithms, which plays an important role in some fields, such as Very Large Scale Integration (VLSI) design [1] and network routing [2]. With the rapid growth of vertices in real-life graphs, the bottleneck problem of MST algorithms in large-scale graphs has become increasingly prominent.

There are three kinds of algorithms to find MST: Prim's algorithm [3], Kruskal's algorithm [4], and Borůvka's algorithm [5]. Prim's and Kruskal's algorithms are difficult to perform in parallel on large-scale graphs due to their inherently sequential workflow and heavy usage of fine-grained synchronization [6]. The Borůvka's algorithm is the most suitable algorithm for parallelism. It iterates on a given graph, as shown in Figure1, with each iteration divided into four stages. First, it finds the edge with minimum weight for each vertex as an edge of MST (Stage 1). Second, it eliminates the mirrored edges (Stage 2). Third, the remaining edges are appended into MST, and the vertices connected by edges are merged into a subtree (or component). The parent of a vertex is used to record a subtree relationship (Stage 3). Finally, the parent array is updated and the path to the root of the subtree is compressed based on the merged structure (Stage 4). This new graph structure is used in the next iteration. The number of subtrees in each iteration is reduced by at least a half.

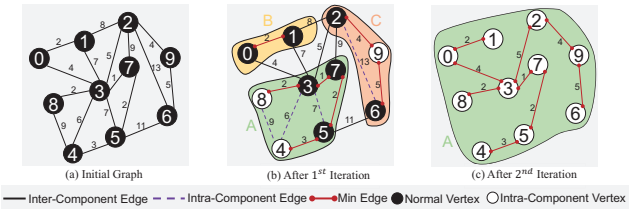


Fig. 1 An illustrative example of Borůvka's algorithm

There are numerous variant algorithms that have been proposed to improve the efficiency of MST computing on CPU [7]–[9] and GPU [10]–[12] platforms. However, they lack customized optimizations in both memory and computation architecture. CPU and GPU-based platforms do not achieve ideal performance due to the randomness of memory access in large-scale graphs and the parallel communication overhead caused by data dependency of MST algorithms. As a customized acceleration platform, FPGA can apply customized memory access strategies and highly parallel computing systems [13]–[16], which have shown significant performance in graph computing problems such as Single Source Shortest Path (SSSP) [17], [18], K-nearest Neighbor (KNN) [19] and Graph Random Walk [20]. However, only a few works focus on accelerating large-scale graph MST on FPGAs. [21] implements an FPGA-based architecture to accelerate Prim's algorithm. Nevertheless, it faces two problems that lead to its low efficiency. First, it only accelerates the minimum reduction stages but not the outer loop of the algorithm. Due to the limitation on parallelism of Prim's algorithm, it does not significantly improve the parallelism degree of the MST algorithm. Second, it only accelerates 160 vertices, resulting in its not addressing the memory and computation pressure faced by large-scale graphs.

1530-2075/24/$31.00 ©2024 IEEE
DOI 10.1109/IPDPS57955.2024.00022

Through an in-depth analysis of Borůvka's algorithm, we find that the performance bottleneck lies in Stage1 (Finding the minimum edge). There are three typical reasons for the inefficiency of Stage1. 1)**Irregular memory access:** The neighboring vertices of a vertex in a large-scale graph are highly random. Limited on-chip SRAM of FPGA cannot ensure that all the access data (i.e., parent data of a vertex) are saved on the chip. Therefore, irregular access leads to significant off-chip access overhead. Moreover, as the graph iterates, there is a large amount of unused data in the on-chip cache. For example, after a subtree is merged, the position of the minimum weight assigned to the subtree will never be utilized in subsequent iterations, resulting in low cache utilization. 2)**Numerous useless computation:** There are a large number of internal edges within the subtree (a component) and searching for internal vertices brings about a lot of useless computation. Though some algorithms [10] use the graph rename scheme to avoid useless computation, they still need to change the graph's topology structure and incur frequent random DRAM access. 3)**Intrinsic data dependency:** In Stage 1, we can search for edges among vertices in parallel to find the minimum edge for each Component. However, a challenge arises when two parallel vertices happen to belong to the same Component. This situation leads to update conflicts when these vertices try to compare and update with the previously determined minimum weight value. To address this issue in mainstream CPU and GPU algorithms [9], [10], they employ a thread-level atomic operation to resolve this data dependency. While this method ensures correctness, it will introduce a notable communication overhead at the thread level, thereby reducing the parallel efficiency. Furthermore, the presence of data dependencies, such as the minimum edge data between Stage2 and Stage3, as well as the Parent data between Stage1 and Stage4, hinders the execution efficiency pipeline.

To address above problems, this paper proposes AMST, an FPGA-based accelerator to boost Graph MST computation. AMST first introduces a high-degree cache strategy to alleviate memory pressure and a hash-based method to improve cache utilization. Then, AMST adopts a graph pruning strategy that skips intra-edge and intra-vertex, as well as sorts edges by weight to improve the efficiency of finding the minimum edge. The pruning strategy and edge sorting remove most of the useless computation by saving useful computing messages into subsequent iterations. Finally, AMST employs a sorting network to resolve data conflict and improve pipeline efficiency. To further improve the parallel efficiency, multi-port caches are customized for MinEdge and Parent. AMST also utilizes a bit-marking method to ensure the parallelism of Stage 1 and Stage 4 in different iterations. Through optimizing the MST algorithm, Stage 2 and Stage 3 are also parallelized and some data are reused to reduce the memory pressure. AMST is evaluated on the Xilinx Alveo U280 card. The results show that AMST achieves a speedup of $17.52\times$ over CPU and $1.89\times$ over GPU. Meanwhile, the energy efficiency of AMST is $74.96\times$ than that of CPU and $10.45\times$ of GPU.

Our main contributions are summarized as follows:

- By analyzing Borůvka's algorithm, the MST performance bottleneck is identified. It is observed that the inefficiency of these algorithms stems from random memory access, numerous useless computations, and intrinsic data dependency. We propose AMST, an FPGA-based accelerator, to boost MST computation.
- AMST first adopts a high-degree cache to alleviate memory pressure, and a hash-based strategy to improve cache utilization. Then, AMST skips intra-edges and intra-vertices, and meanwhile sorts edges by weight to eliminate numerous useless computations.
- AMST resolves the data dependency by bit-marking and parallel pipeline. Moreover, AMST employs customized sorting networks to reduce communication overhead and multi-port caches to improve parallel efficiency.

II. Background

A. Graph Format

An undirected weighted graph G=(V, E) has a set of vertices V and a set of edges E with weights. The graph in this paper is represented in the standard compressed sparse row (CSR) format [22]. CSR uses two arrays to represent a graph: the edge array and the offset array. Edge (src, dest,w) is an edge between the source vertex (src) and the destination vertex (dest), and the weight of it is w. Values stored in the edge array represent the edges' destination indexes and their weight values. The vertex ID in the offset array implicitly stores the source index. The i-th entry in the offset array represents the starting index (s_e) of the i-th vertex and the ending index (d_e) of the (i-1)-th vertex in the edge array.

B. Graph Minimum Spanning Tree

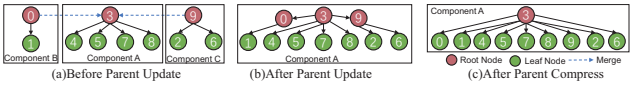
A graph's Minimum Spanning Tree (MST) is a tree that covers all its vertices and has the smallest total weight of its edges. There are three kinds of algorithms to find MST: Prim's algorithm [3], Kruskal's algorithm [4], and Borůvka's algorithm [5]. Prim's and Krustkal's algorithms are difficult to perform in parallel on large-scale graphs due to their inherently sequential workflow and heavy usage of fine-grained synchronization [6], which leads to low computational efficiency. Borůvka's algorithm is the most suitable algorithm that can be executed in parallel efficiently [6]. It starts by making each vertex a single-element connected component. A **component** is a subtree where every pair of vertices is connected by a path, and no vertex is linked to another component. This algorithm gradually merges the components by adding edges that form the MST. To illustrate this, Figure 1(a) shows an initial graph where each vertex is a separate component. Figure 1(b) shows the graph state after the first iteration. Here, we can see three components named A, B, and C, each containing one or more vertices. After the second iteration, the final MST that includes all vertices is obtained, as shown in Figure 1(c).

A **Parent** array is used to keep track of the component membership of each vertex. At the beginning, each vertex is its own parent and when the lightest edge (minimum edge, MinEdge) is chosen, the parent of one endpoint of the edge is set to the other endpoint. The vertices that share the same

Algorithm 1 Borůvka's Algorithm

Input: A graph G=(V,E)
Output: MST

```
1: Init_Array(Parent, MinEdge, Root, RV_count)
2: while RV_count == 1 do
3:     % stage1 Find minimum edge in a subtree (traversal all vertices)
4:     for each v in [0, Vertex_number] do
5:         me = MinEdge[Parent[v]]
6:         for e in v.edges do
7:             if Parent[v]!=Parent[e.dest] & e.weight < me.weight then
8:                 me = e
           MinEdge[Parent[v]] = me
9:     % stage2 Remove repeated edges
10:    for each r in Root do
11:        dest = Parent[MinEdge[r].dest]
12:        dest_dest = Parent[MinEdge[dest].dest]
13:        if Parnet[r] == dest_dest & r < dest then
14:            MinEdge[r] = NULL
15:    % stage3 Append edge in MST & Update the parent of root Vertices
16:    for r in Root do
17:        if MinEdge[r]!=NULL then
18:            MST.push(MinEdge)
19:            Parent[r] = Parent[MinEdge[r].dest]
20:        Update(MinEdge, Root, RV_count);
21:    % stage4 Compress Parent path & Update the parent of leaf vertices
22:    for each v in [0, Vertex_number] do
23:        Compress(Parent,v) % Eod: Parent[r]==r
```



(a)Before Parent Update (b)After Parent Update (c)After Parent Compress

Fig. 2 Parent data tree

parent are considered in the same component. For instance, after the first iteration in Figure 1(b), the parent of vertex 1 changes from vertex 1 to vertex 0 and the parent of vertex 0 remains vertex 0. Therefore, vertex 0 and vertex 1 are in the component B. The parent relationship in a component can be represented as a tree as shown in Figure 2(a). In the component tree, Parent Vertices are the Root Vertices and other vertices are the Leaf Vertices. In each iteration, we update the parent data in Figure 2(b) and compress the tree into two layers as shown in Figure 2(c). This compression can enhance the computation efficiency.

C. Borůvka's Algorithm

Borůvka's algorithm is performed in several iterations as shown in Algorithm 1. First, this algorithm initializes the parameters state. Then, it computes the MST by iterating over all vertices. The iteration has four stages: Find the minimum edge of a component, Remove repeated edges, Append edges in MST, and Compress parent path.

Stage 1-Find minimum edge of a component. A **MinEdge** array is used to store the minimum (lightest) edge. The state of MinEdge data of each component is initially set to null and its weight is set to infinity. We find the minimum edge of each vertex sequentially and use it to update the MinEdge array of its component (Parent). To illustrate this, consider the second iteration in Figure 1(b) as an example. Vertex 0 selects its edge with weight 4 to vertex 3 as MinEdge and vertex 1 selects its edge with weight 7 to vertex 3 as MinEdge. Since weight 4 is smaller than weight 7, component B selects the edge with weight 4 to component A.

Stage 2-Remove repeated edges. After Stage 1 finds the minimum edge of all components, two components may find the same minimum edge. If we add this edge twice in MST, the result will be incorrect. Therefore, we have to remove repeated edges in the MinEdge array by checking the Parent relationship of the source vertex and destination vertex. For instance, in the first iteration in Figure 1, both vertex 0 and vertex 1 select the edge with weight 2 as their minimum edge and update the MinEdge Array. Stage 2 removes the minimum edge of weight 2 in MinEdge to ensure correctness.

Stage 3-Append edges in MST. Stage 3 selects the remaining edges in MinEdge and pushes them into MST. Moreover, it updates the Parent of one root vertex to the parent of the edge destination vertex, which means two components are merged. For example, in the second iteration shown in Figure 1, the edge of component B with weight 2 is pushed into MST. The parent of vertex 0 is updated to vertex 3 and component B is merged into component A as shown in Figure 2(b).

Stage 4-Compress parent path. After Stage 3, parents of root vertices are updated but parents of leaf vertices are not updated. If the depth is not compressed, the depth of the parent tree will increase as iteration progresses which incurs frequent random DRAM access. To solve this problem, Stage 4 updates the parent of all vertices and compresses the path to find the actual parent root (which satisfies **Parent[r]==r** and represents the component). As shown in Figure 2(c), Stage 4 compresses the depth from three layers to two layers.

III. Motivation: Performance Bottleneck Analysis

We conduct characterizations for Borůvka's Algorithm with c-based codes in Intel Xeon Siler CPU 4114. The details of the system configuration and datasets used in this evaluation are given in Section VI. Figure 3(a) shows that four stages take up 82.24%, 3.68%, 2.37%, and 11.72% of the total execution time in the Borůvka's Algorithm. The results suggest that the minimum edge computation (Stage 1) is the key bottleneck for MST due to the following problems.

A. Irregular Memory Access

The memory access of Parent and MinEdge Data exhibits irregular patterns, originating from two sources: random neighbors of vertices in the graph and varying memory access characteristics across different stages and iterations.

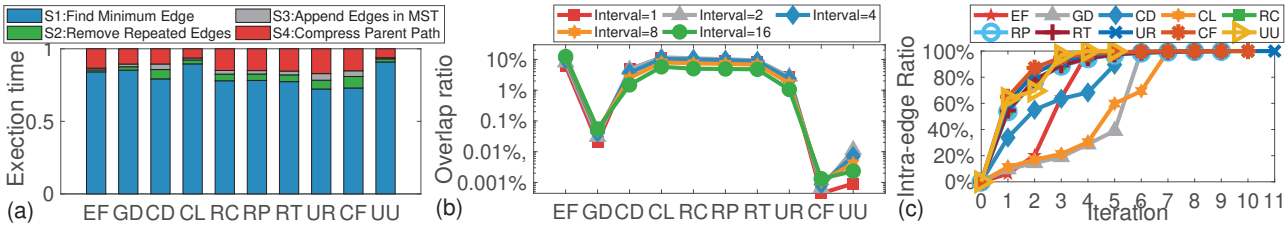


Fig. 3 Performance bottleneck analysis of the Borůvka's algorithm, include (a). Execution time breakdown of the four stages in the algorithm; (b). The average neighborhood of statistical vertices in the order of vertex indices in different graphs; (c). The useless computation due to the intra-edge

*1) **Random neighbors of vertices in the graph**:* In real-world large-scale graphs, the distribution of neighboring vertices connected to a given vertex is highly random [23]. In every stage and iteration of the algorithm, we need to get the Parent data of neighboring vertices and the MinEdge of current vertices. This means we have to randomly access the memory for Parent and MinEdge data, and this often leads to frequent cache misses and off-chip memory accesses in CPU and GPU architectures. In Figure 3(b), we measure the average neighborhood overlap ratio of vertices in order of the vertex indices under different vertex intervals. The neighborhood overlap ratio is defined as the number of common neighbors of statistical vertices divided by the number of neighbors of statistical vertices. We find that the average overlap ratio of neighboring vertices is consistently below 10%. Consequently, the low neighborhood ratio significantly limits the potential for data reuse in the cache.

*2) **Varying memory access characteristics across different Stages and Iterations**:* The MST algorithm consists of four stages, each characterized by different access patterns for Parent and MinEdge. Stage1 accesses Parent and MinEdge randomly based on edge order, Stage2 and Stage3 access MinEdge according to root vertex order, and Stage4 updates Parent in vertex order. Additionally, access to the Parent data is also random. However, since each stage entails data dependencies and is executed sequentially, the data between stages are refreshed in the cache. Consequently, traditional cache strategies face difficulties in managing data.

Due to the nature of graph data, the combination of random access and different memory access patterns in the MST algorithm results in a low data reuse ratio. This, in turn, can significantly impact performance, particularly when on-chip cache capacity is limited or the cache strategy is not optimized. Although the typical U280 FPGA is equipped with 9MB BRAM, large-scale graphs, such as UU with 133 million vertices, pose significant challenges in fully caching the Parent and MinEdge data on-chip. Consequently, an efficient cache strategy must be devised to mitigate these challenges.

B. Useless Computation

Figure 3(a) shows that Stage1 takes up most of the computation time, which is not only related to the inefficient memory access mentioned above but also due to a large number of useless computations, arising from internal edges and inefficient edge weight comparisons.

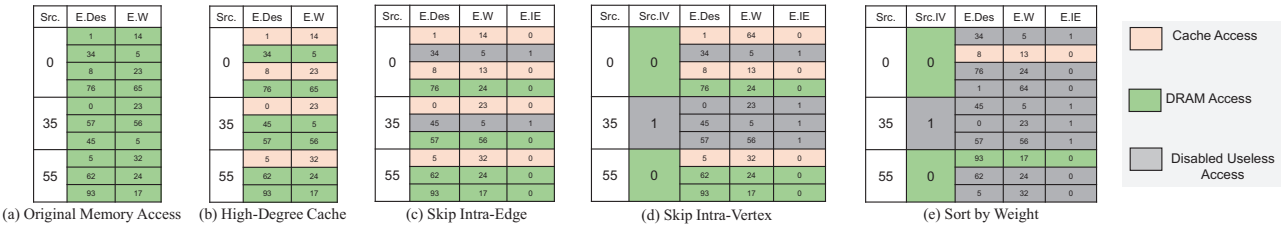
*1) **Internal edges and vertices**:* As the iteration progresses, the number of vertices in a component grows, resulting in an increasing number of edges within the component. If all edges of a vertex are internal edges (Intra-edge), then the vertex becomes an internal vertex (Intra-vertex). For example, in Figure 1(b), after one round of iteration, vertices 3 and 4 are both vertices of Component B. The edge connecting them with weight 6 also becomes an internal edge (dashed purple edge). The edges of vertex 4 are all internal, so it also becomes an internal vertices. Since the internal edge is not the minimum edge in the end, the calculation of internal edges and vertices is useless. To illustrate this issue, we have computed the useless computation ratio of different graphs in different iteration rounds. The useless computation ratio is defined as the ratio of useless edges to all edges. We compute the average value of ratios across all iterations and all graphs and Figure 3(c) presents the results, indicating that 76.08% of the computations are useless. Especially, after the second iteration, more than half of the edges become internal edges.

*2) **Inefficient edge weight comparisons**:* In Stage1, when we compare the weight of all edges of one vertex, we have to load and go through them one by one. However, since the graph structure remains constant during different iterations, the information from these comparisons is not saved for the next iteration, leading to many redundant and useless comparisons.

C. Parallel Overhead

Borůvka's algorithm is suitable for parallel acceleration, but it suffers from parallel overhead when performing computation simultaneously. The parallel overhead mainly comes from two sources: update conflicts and memory read-write conflicts.

*1) **Update conflicts**:* In Stage 1, the edges of vertices are searched in parallel to find the minimum edge for each Component, but two parallel vertices may belong to the same Component, and they may conflict when comparing and updating with the previous minimum weight value. This conflict must be resolved to ensure the correctness of the MST, otherwise, it may lead to incorrect final results. For example, in Figure 1(b), in the second iteration, if vertex 1 updates the MinEdge after vertex 0, the final minimum edge of component B will be the edge with weight 7 instead of the edge with weight 4, resulting in MST incorrectness. To solve this problem, a common solution is to introduce atomic protection, such as a compare-and-swap function, but it introduces memory barriers or locks, which will bring communication overhead, and this overhead increases with the



(a) Original Memory Access (b) High-Degree Cache (c) Skip Intra-Edge (d) Skip Intra-Vertex (e) Sort by Weight

Fig. 4 DRAM access in different optimization strategies. (Src(Source Vertex), E(Edge), Des(Destination Vertex), W(Weight), IE(Intra-edge), IV(Intra-vertex))

increase of parallelism. We analyzed the atomic operations in MASTIFF [9], which accounted for more than 35.19% execution time, showing a large communication overhead.

*2) **Memory read-write conflicts**:* Increasing the parallelism will increase the number of concurrent accesses to shared memory, which may cause memory read-write conflicts. These conflicts require multi-port read-write memory, which is not friendly to the existing micro-architecture of CPU and GPU.

IV. Optimization

Through deep analysis of Borůvka's algorithm, we have three main optimizations based on the three above problems.

A. Optimization 1: High Degree Vertex Data Cache

Real-world graphs usually exhibit a power-law distribution [24], [25], which implies that there are only a few vertices with numerous neighbors, while the majority of vertices have only a few neighbors. Based on their degree, these vertices can be classified into two groups: high-degree vertices (HDVs) and low-degree vertices (LDVs). The parent data of HDVs possesses higher temporal and spatial locality compared to the parent data of LDVs, as it is more frequently accessed by their neighboring edges. To leverage this characteristic, we employ the degree-based grouping (DBG) algorithm [26], a commonly used technique for balancing graph partitions. DBG sorts and assigns new indices to the vertices in descending order of in-degree, ensuring that smaller vertex indices correspond to higher degrees. HDVs and LDVs exhibit distinct characteristics in terms of both computation and memory access:

- **High-Degree Vertices.** The parent data of HDVs necessitates frequent access during the computation process by their neighbors. Furthermore, during the component merging process, HDVs may eventually become the root vertices and experience frequent access, including the corresponding MinEdge data they represent. Moreover, due to their larger number of edges, HDVs have longer data processing time.
- **Low-Degree Vertices.** LDVs have fewer edges and shorter processing time. LDVs are more likely to be merged into other components during the computation process. As a result, their parent and MinEdge data have lower temporal and spatial locality.

Based on the above observation, we propose to cache the parent and MinEdge data of HDVs on-chip and store corresponding data of LDVs in off-chip DRAM. By adopting this approach, as illustrated in Figure 4(b), we effectively reduce the memory access time for HDVs. Large-scale graphs, characterized by random vertex neighbors and varying memory access patterns across stages and iterations, exhibit poor temporal locality and frequent DRAM access, as depicted in Figure 4(a). Consequently, in order to optimize memory access in such scenarios, leveraging spatial locality becomes the primary opportunity. Hence, a direct HDV caching strategy is straightforward but efficient.

B. Optimization 2: Remove Useless Computation

*1) **Skip Intra-Edges**:* The computation of intra-edges during Stage 1 does not affect the final result. By eliminating the computation of intra-edges, we can significantly improve the efficiency of both computation and memory access. To accomplish this, we employ a bit-marking method to identify intra-edges. We determine whether an edge is intra-edge by comparing the parent arrays of the source and destination vertices. If the parent arrays are equal, the edge is an intra-edge. In Stage 1, we made this determination when loading an edge (Line 7 in Algorithm 1), but we did not store the value for future use. By recording this value, we can minimize memory access to the parent array. To implement this optimization, we introduce an **intra-edge (IE) flag in the edge data**, which can be reused in subsequent iterations after one round of marking. Figure 4(c) illustrates the skipping of edges from vertex 0 to 34 and from vertex 35 to 45 after marking intra-edges, leading to improved memory access efficiency.

*2) **Skip Intra-Vertices**:* As the iteration progresses, certain vertices may have all their adjacent edges identified as intra-edges. Consequently, it becomes unnecessary to compute these vertices during Stage 1 and Stage 4, as exemplified by vertex 4 in Figure 1(b). Therefore, in addition to marking intra-edges, we also introduce an **intra-vertices (IV) flag in the parent data**. During Stage 1 and Stage 4, if a vertex is detected as an intra-vertex, the computation of this vertex is skipped. Figure 4(d) demonstrates the elimination of memory access and computation for data related to intra-vertex 35.

*3) **Sort Edges by Weight**:* During Stage 1, the goal is to identify the minimum edge for a component, which requires comparing the weights of different edges. We can optimize this process by storing the comparison results. This way, we can reduce the number of comparisons needed in subsequent iterations. However, this method increases memory access due to the frequent updates of the edge array. To address this, we propose a solution: sorting edges by weight during the graph preprocessing phase. This approach preserves the

comparison information of edge weights and minimizes the number of weight comparisons needed. Additionally, it helps reduce the memory access overhead of parent and edge data. As shown in Figure 4(e), sorting edges by weight allows us to find the edge with the smallest weight for vertex 55 with just one comparison. This eliminates the need to access the edges involving endpoints 62 and 5, along with their respective parent data.

C. Optimization 3: Reduce Parallel Overhead through Sorting Network and Multi-port Cache

1) ***Resolve Update Conflicts with Sorting Network****:* During the parallel search for the minimum edge of vertices, each vertex's search for the minimum edge in Stage 1 operates independently without interference. Update conflicts only occur at the final update. Therefore, we only need to compare the weight values of the same component and update the minimum value when Stage 1 is preparing to write back the minimum weight. To improve comparison efficiency and write-back efficiency, we employ a sorting network by utilizing an address and data comparison scheme based on Bitonic sorting. Its structure is provided in Section V-C.

2) ***Resolve Read-Write Conflicts with Multi-port Cache****:* FPGAs provide on-chip SRAM with dual read/write operations, making it possible to design multi-ports caches for handling parallel read-write conflicts. However, there is a limitation on the available on-chip SRAM resources on FGPAs and this limitation is critical when complementing caches. To make the most of these resources, optimizing cache utilization becomes a key concern. The direct high-degree cache (HDC) faces a low utilization as the iteration progresses. This is because some vertex data, such as the parent data of inter-vertices and the MinEdge data of merged vertices, never be used. On the other hand, the direct multi-port implantation leads to space redundancy since not every position in the multi-write Parent cache will be written. To address these challenges, we introduce a dynamic hash-based cache strategy and an address-based multi-port Parent Cache in Section V-F.

V. AMST Architecture

A. Overview

Based on the aforementioned observations, we present AMST, an FPGA-based accelerator designed to enhance the efficiency of large-scale graph MST computation. Figure 5 provides an overview of the AMST architecture. It comprises four main components: the Top Controller, specialized processing modules for each stage, multi-port Caches, and the Writer modules. The processing modules consist of the Finding Module (FM), Removing Module (RM), Appending Module (AM), and Compressing Module (CM). Each module incorporates dedicated parallel processing engines (PEs). The cache consists of the Parent Cache with N read ports (RPs) and M write ports (WPs), as well as the MinEdge Cache with nRPs/1WP. The Top Controller manages the iteration state, identifies the end of iterations, and schedules computations among the different processing modules. Four processing modules process the corresponding stages of Borůvka's algorithm. Each PE in every module connects to a separate HBM channel (CH) and one Parent or MinEdge Cache RP. The Writer modules, namely the Parent Writer and MinEdge Writer, receive Parent and MinEdge results from the PEs and update the values of source vertices in HBM or cache, based on their degrees, for the next module or iteration. All accesses to global memory occur at the granularity of a block (512 bits) to optimize memory efficiency.

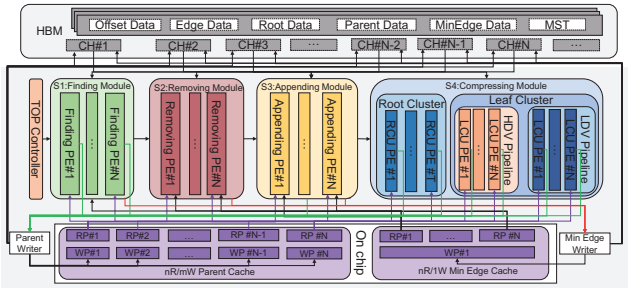


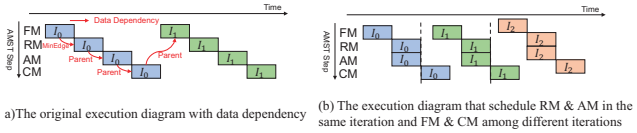
Fig. 5 Architecture overview of AMST

In this section, we first introduce two pipeline optimizations in Subsection V-B to enable a high-performance architecture. Next, we describe the architectures of the FM, RM, AM, and CM modules in Subsections V-C to V-E. Finally, we present memory architecture optimizations to improve the cache utilization and deploy multi-port caches in Subsection V-F.

B. Pipeline Optimization

In AMST, the output data of one module becomes the input data for the next module. Traditionally, these modules in AMST are executed sequentially due to the dependency on MinEdge and Parent data across different modules and iterations, as illustrated in Figure 6(a). However, this serial execution pattern limits the pipeline efficiency. To overcome this limitation, we focus on two key optimizations:

1) ***Pipeline-Optimization-1: Parallelization of RM and AM within the same iteration cycle through computation optimization****:* Through careful analysis, we observe that the data dependency between RM and AM can be treated as a pseudo-dependency. In the previous algorithm, when RM and AM are executed in parallel, updating the Parent array in AM could sometimes cause RM to incorrectly identify repeated edges. This error occurs because RM only considers the relationship between the parent of the current vertex and the parent of the endpoint vertex of the minimum edge as shown in Line 13 of Algorithm 1, without considering the relationship between the parent of the current vertex and the parent of the endpoint vertex of the minimum edge. Although this check was done in Stage 1 (Line 7 in Algorithm 1), adding this condition will remove the dependency and allow RM to update the Parent earlier. With these computation optimization techniques, we can effectively get rid of this dependency and enable the parallel execution of RM and AM, as shown



a)The original execution diagram with data dependency

(b) The execution diagram that schedule RM & AM in the same iteration and FM & CM among different iterations

Fig. 6 Pipeline of AMST

in Figure 6(b), thus, reducing the computation and memory access pressure. More details about the hardware structure for these two modules will be provided in Section V-D.

*2) **Pipeline-Optimization-2: Parallelization of FM and CM through bit-marking**:* Unlike the pseudo-dependency between RM and AM, FM and CM have a real data dependency. FM needs the updated parent information from CM to decide if an edge is internal. As the iterations go on, most vertices become internal and CM updates the parents of root and high-degree vertices first. In the early stages, when FM looks at vertices with lower indices, neighboring vertices might already have updated parents. This means FM can potentially find the minimum edge for these vertices in earlier stages, without waiting for CM to finish updating all parents. The main challenge is detecting whether the Parent data is updated.

To tackle this, AMST uses a bit-marking method. We add a 6-bit marker (it_idx) into the Parent data to show the current iteration index. When CM updates Parent, it also updates it_idx to match the current iteration. FM checks it_idx to see if a vertex has been updated. If a vertex has an edge that has not been fully updated, it goes back into FM's scheduling FIFO to be recalculated later. If not, FM continues until it is done. To make the pipeline more efficient, AMST employs an event-triggered way. Specifically, after CM updates Parent of HDVs in root vertices, FM starts. This way, we can parallelize FM and CM across different iterations, as shown in Figure 6(b), and improve the pipeline efficiency.

C. Finding Module: Prune Useless Computation

1) Finding PE: The primary function of the Finding Module (FM) is to identify the minimum edge for each component. Figure 7 shows the architecture of the FM. At the beginning of Stage1, the task scheduler unit (depicted in Figure 7 (b)) obtains the source vertices that need to be allocated, along with their corresponding edge information (start and end positions), parent vertices, and minimum edge weights (me_p). These tasks are then assigned to the Finding Processing Element (FPE). Since the computations are continuous, the Parent data is continuously loaded into a Ping-Pong Buffer. When one FPE is idle, it receives a group of source vertex information for processing. The task scheduler checks the Parent data to see if the current vertex is internal (marked as 1'b1). If it is, the current vertex is skipped, and the FPE goes to the next non-internal vertex. Once assigned, the FPE initiates its computation.

The FPE is responsible for finding the minimum edge information for the assigned vertex, as depicted in Figure 7(a). The data flow within the FPE is as follows:

In Step ①, the FPE loads the edge vertex data into the Ping-Pong Buffer based on the source and destination (Se, De) information. It fetches edge data from DRAM into the Ping buffer and processes edge data from the Pong buffer. If the Intra-flag of an edge is 1 (meaning it is an internal edge), the FPE skips more processing and moves on to the next edge. In Step ②, the FPE compares the endpoint vertex number of the edge with the value v_t, which indicates if the parent data is stored on-chip or off-chip. If the endpoint vertex

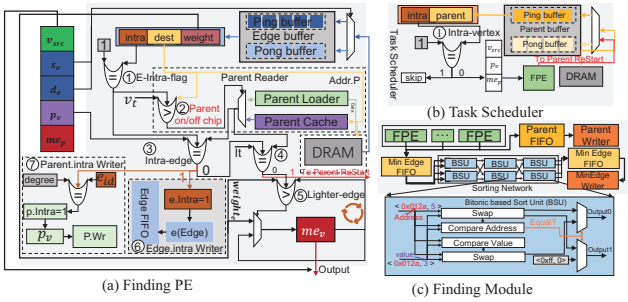


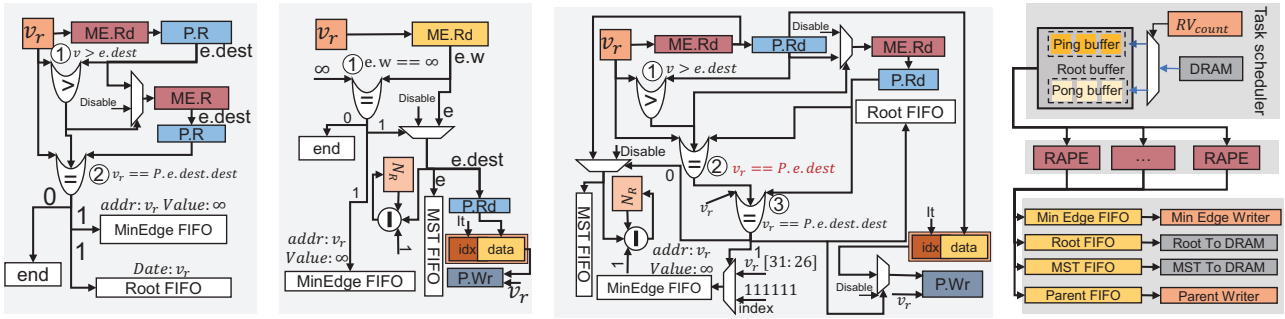
Fig. 7 Architecture of Finding Module

number is less than v_t, the FPE reads the parent data from the on-chip cache; otherwise, it reads the parent data from off-chip memory. In Step ③, the FPE first checks if the parent vertex is equal to the current vertex. If they are equal and the edge becomes an internal edge, the FPE skips ahead to Steps ⑥ and ⑦. In Step ④, to ensure parallel scheduling of FM and CM, the FPE checks if the parent data of the intermediate vertex is up-to-date. If the data is not up-to-date, the source vertex needs to wait until the parent data of the endpoint vertex is updated. Consequently, the FPE terminates the current computation and places the task of the current source vertex back into the FIFO of the Task Scheduler for subsequent scheduling. In Step ⑤, the FPE examines if the weight of the current edge is smaller than the minimum edge of the parent vertex. If it is not smaller, the FPE moves on to the next edge; otherwise, it updates the minimum edge value (me_p). AMST employs edge weight sorting, and once an edge is confirmed as the minimum in this step, all subsequent edge weights will be larger than the minimum edge. Therefore, if a smaller edge is found in this step, the subsequent edges can be skipped, and the minimum edge is directly output. If no edge smaller than me_p is found, me_p is output directly. In Step ⑥, the FPE updates the data for internal edges and writes it back to the DRAM. In Step ⑦, if all edges of a vertex become internal edges, the vertex becomes an internal vertex, and the FPE updates its parent data accordingly.

2) Parallel Computation: To improve the throughput of FM, we employ multiple FPEs working in parallel, as depicted in Figure 7(c). The scheduler unit assigns source vertices to FPEs for processing. After finding the minimum edge, each FPE needs to update the MinEdge. To minimize the overhead caused by MinEdge update conflicts, we employ a sorting network within FM, illustrated in Figure 7(c). This network leverages the Bitonic sorting technique, which comprises cascaded Bitonic Sorting Units (BSUs). Each BSU takes a pair of $< address, value >$ as input, sorts addresses and data, and eliminates duplicate addresses to reduce the atomic burden. The sorted data is then written back in an ordered manner.

D. Removing and Appending Module: Pipeline Merging

The Removing Module (RM) aims to remove repeated edges, while the Appending Module (AM) appends edges to MST and merges components by updating the Parent. Figure 8(a) and (b) show the original structures of RPE and APE. However, we have made computation optimizations to



(a) Removing PE (b) Appending PE (c) Removing and Appending PE (d) Removing and Appending Module

Fig. 8 Architecture of Removing and Appending Module

improve pipeline efficiency and introduced the Removing and Appending PE (RAPE) in Figure 8(c). Compared to RPE and APE, RAPE includes a new Step ② to check parent relationships and removes Step ① from APE, which was used to check the weight value. Step ① and Step ③ in RAPE perform the same functions as RPE. Step ② ensures the correctness of APE. Before pipeline merging, RPE and APE require three MinEdge reads and three Parent reads. But after merging, the number of MinEdge and Parent reads drops to two, improving pipeline and memory access efficiency.

The Removing and Appending Module deploys multiple RAPEs in parallel shown in Figure 8(d). The Task Scheduler retrieves root vertices from the DRAM and assigns them to RAPEs. RAPEs work independently, without data dependencies, and updates to the MinEdge, Root, and MST are initially stored in a FIFO buffer before being written back to DRAM.

E. Compressing Module: Heterogeneous Pipelines

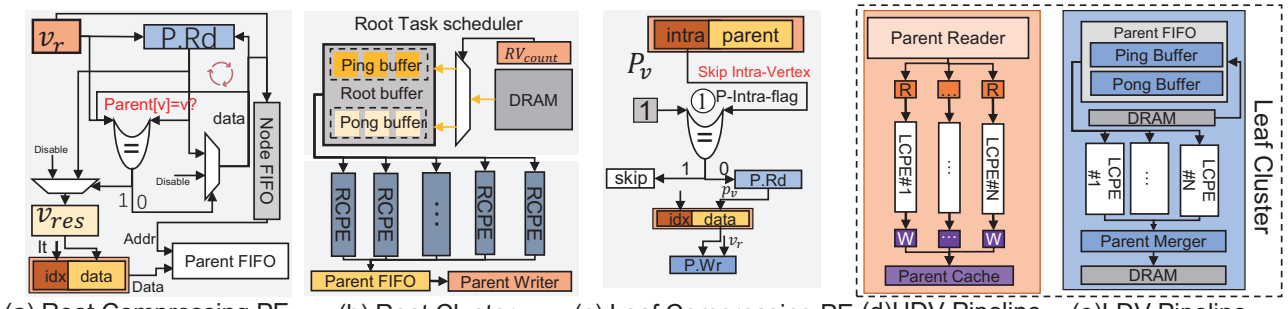
The Compressing Module (CM) is responsible for updating Parent and compressing the Parent tree. Unlike FM/RM/AM, CM only deals with Parent, not MinEdge, and has no data dependencies when updating Parent. Since AM has initially updated the Parent of some root vertices, the remaining vertices can ultimately determine its updated root vertex (Parent[r]==r). By analyzing Figure 2, we observe that the characteristics of root vertices differ from those of leaf vertices:

- **Root vertices** require irregular depth backtracking, as they must continuously read parent vertices until reaching the ultimate root vertex.
- **Leaf vertices** exhibit a stable backtracking depth of 2, as they only need to update the root value of their parent vertices if they update after root vertices.

Exploiting this insight, we use heterogeneous pipelines to handle root and leaf vertices separately. Specifically, we employ the Root Cluster to update root vertices and the Leaf Cluster to update leaf vertices.

Figure 9(b) shows the architecture of the Root Cluster, which enables parallel processing of root vertices. The Root Cluster reads root vertices from DRAM into the root buffer and assigns them to the Root Compressing PEs (RCPEs). RCPE, shown in Figure 9(a), finds parents of current root vertices and combines it with the iteration cycle (It). These results from RCPEs are subsequently written to the Parent FIFO, which is efficiently handled by the Parent Writer for unified write-back.

For leaf vertices, we introduce the Leaf Compressing PE (LCPE) to update Parent as illustrated in Figure 9(c). Since Parents of root vertices have already been updated, the processing is simplified and does not require repetitive Parent reads. One optimization in LCPE involves skipping intra-vertices (Step①), thereby reducing the number of Parent reads. This optimization is feasible because intra-vertices are not used in subsequent iterations and do not necessitate further updates. These LCPEs could be organized into a cluster. However, we observe that leaf vertices can be classified into High-Degree Vertices (HDVs) and Low-Degree Vertices (LDVs), as discussed in Section IV-A, and their memory access behaviors differ, especially after introducing the HDV cache. Therefore,



(a) Root Compressing PE (b) Root Cluster (c) Leaf Compressing PE (d)HDV Pipeline (e)LDV Pipeline

Fig. 9 Architecture of Compressing Module

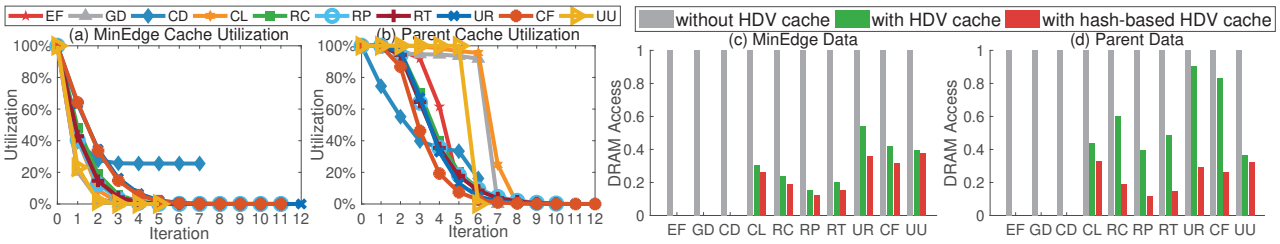


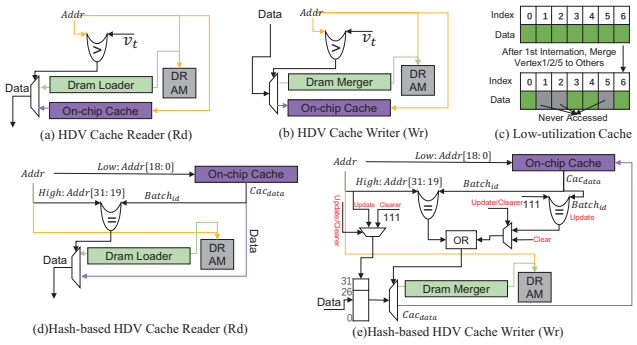
Fig. 10 Cache utilization and DRAM access under direct HDV cache and hash-based HDV cache

we divide the leaf cluster into two pipelines: the HDV pipeline and the LDV pipeline, illustrated in Figure 9(d) and 9(e), respectively. The HDV pipeline focuses on updating HDVs, reading from the HDV cache, and directly writing results to the cache. In contrast, the LDV pipeline handles the Parent updates of LDVs. Unlike HDVs, LDVs require reading and writing the Parent data to and from DRAM. To improve DRAM access efficiency, we implement a Parent ping-pong FIFO that sequentially reads the Parent data of LDVs and a Parent Merger is deployed to consolidate write operations.

F. Memory Architecture: Adopt Hash-based HDV Cache and Deploy Multi-port Cache

1) Hash-based HDV Cache: Based on the HDV cache strategy discussed in Section IV-A, the read and write positions of Parent and MinEdge are determined by the address and a partitioning threshold (V_t), as illustrated in Figure 11(a) and 11(b). However, as iterations progress, this direct HDV cache suffers from a low-utilization issue. Figure 10(a) and (b) show that both the Parent and MinEdge Cache utilization drop below 50% after the second iteration in most graphs. This occurs because: on one hand, some vertices, like vertex 1, 2, and 5 as shown in Figure 1(c), are merged into other vertices, leaving unused MinEdge in those positions. On the other hand, after some vertices become intra-vertices, their parent data also becomes unused in the cache. **Therefore, vertices merging in MST causes the low utilization of the cache. This problem is common in iterative graph algorithms.** To address this issue, we propose a hash-based HDV cache strategy.

This hash-based cache adopts a direct address mapping and divides all data into multiple batches by the cache capacity (512K). The data in cache (cac_{data}) is divided into $batch_{id}$ and real data such as Parent and MinEdge data. **Initially**, the cache stores the data of HDVs and the $batch_{id}$ is zero. When AMST **reads data**, as shown in Figure 11(d), given an address (Addr), it first reads $batch_{id}$ of Addr[18:0] in the cache. If the $batch_{id}$ matches the Addr[31:19], the cache hits. Otherwise, it fetches from DRAM. When AMST **writes data**, it will clear or update the cache data as shown in Figure 11(e). When a root is merged into other roots or a vertex becomes an intra-vertex, the corresponding data in MinEdge or Parent cache goes unused and the $batch_{id}$ is cleared to empty. When AMST updates data, it first checks the $batch_{id}$ in the cache. If the $batch_{id}$ is empty or matches the Addr[31:19], it will write to the cache. Otherwise, it writes to DRAM.



(a) HDV Cache Reader (Rd) (b) HDV Cache Writer (Wr) (c) Low-utilization Cache

(d)Hash-based HDV Cache Reader (Rd) (e)Hash-based HDV Cache Writer (Wr)

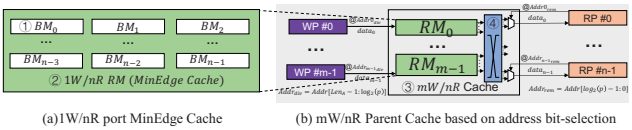
Fig. 11 Memory writer and reader

By using the hash-based HDV cache, the cache utilization can be significantly improved. As Figure 10(c) and (d) demonstrate, the DRAM access times of MinEdge and Parent are reduced by an average of 22.50% and 54.28%, respectively, compared to the direct HDV cache average.

2) Multi-port Cache: To support concurrent processing by multiple PEs and address data update and memory read-write conflicts, the cache needs to have multiple ports. In AMST, the Parent data requires multiple reads and writes, while MinEdge requires one read and multiple writes. However, on an FPGA, only 2W2R BRAMs are available. Below, we will describe how to build a multi-port cache using FPGA resources:

1. 1WnR MinEdge cache. The MinEdge cache architecture is depicted in Figure 12(a) and can be built as follows: ① Deploy a 1W1R BRAM (BM) with a size of D. ② Replicate BM n times, forming a 1WnR BRAM (RM), where each BRAM holds the same content, facilitating multiple reads.

2. mWnR Parent cache. In the context of the mWnR Parent Cache, a straightforward method is to duplicate the RM m times. However, this approach leads to considerable space redundancy since not every position in the BRAM will be written. In AMST, we address this problem by employing address-based quotient and remainder operations. We assign each LCPE, as depicted in Figure 9(d), a fixed vertex ID for its respective write port. This results in LCPEs writing to the Parent cache in a consistent pattern described as follows: "Assuming P LCPEs execute in parallel, the ID-th LCPE updates



(a)1W/nR port MinEdge Cache (b) mW/nR Parent Cache based on address bit-selection

Fig. 12 Architecture of multi-port cache

HDVs with IDs: ID, ID+P, ID+2P, ..., ID+tP. Consequently, their write addresses to the cache are discrete and follow the pattern ID, ID+P, ID+2P, ..., ID+tP." To implement this pattern, we use an address-based selection structure, as illustrated in Figure 12(b), constructed through these steps:

- In Step ①, we set the size of the basic 2W/2R BRAM (BM) is $2D/P$ instead of D.
- In Step ②, we build a 2WnR BRAM (RM) by replicating n/2 copies of BM, which stores the same content.
- In Step ③, we replicate RM in m/2 copies to create the mW/nR cache. These write ports (WPs) are allocated to different LCPEs, each responsible for writing Parents of different vertices. Consequently, the content saved in different RMs varies. And the write address is converted to the actual address by dividing it by P:$Addr_{div} = Addr/P = Addr_{write}[Len_A - 1 : log_2(P)]$.
- In Step ④, multiplexers are added to enable n read operations. The $Addr_{div}$ of the i-th RP read address is sent to m RMs, m output Parent is sent to the i-th multiplexer, and the result is chosen by calculating the remainder of the read address: $Addr_{rem} = Addr\%P = Addr_{read}[log_2(P) - 1 : 0]$. This bit-selection divisor and remainder operation can accurately determine the address and the RM that stores Parent.

Using this approach, we significantly reduce the Parent cache size by a factor of 2P, alleviating space redundancy and the low cache utilization issue.

VI. Evaluation

We first evaluate the efficiency of the proposed algorithm, parallelism, and pipeline optimization techniques. We then compare AMST with CPU and GPU solutions. Finally, we report the FPGA resource utilization.

A. Experimental Setup

1) Hardware platform: We implement AMST on the Xilinx Alveo U280 accelerator card equipped with a 32 GB DDR using Xilinx Vitis 2022.2. The U280 card provides 34.5MB on-chip UltraRAMs, 9MB on-chip BRAMs, 1.3M LUTs, 2.6M registers, and two 4GB HBM2 stacks. In the evaluation, the single cache is set to 2MB (512K vertices). We host the FPGA on a server with two 10-core Intel Xeon Silver 4144 @ 2.02GHZ, 128GB of DDR4-2133 RAM.

2) DataSets: Table I shows the details of the graph datasets that we use in our experiments, including social networks, collaboration networks, road networks, and web graphs. We perform a degree-based grouping reordering algorithm on these datasets. Graphs are represented in the CSR format. Each edge contains the destination vertex and weight. Weights have 4 bytes and are assigned randomly. Table II shows the processing time on one CPU thread. Compared with MST, the graph reordering overhead is small. What's more, the graph reordering does not affect the iteration cycles in MST.

TABLE I The graph datasets.

Graphs	Vertices	Edges	Type	Categories
ego-Facebook (EF) [27]	4.1K	88.2K	UnDirected	Social network
gemsec-Deezer_HR (GD) [28]	54.5K	498.2K	UnDirected	Social network
com-DBLP (CD) [29]	317.0K	1.0M	UnDirected	Collaboration network
com-LiveJournal (CL) [29]	3.9M	34.7M	UnDirected	Social network
roadNet-CA (RC) [30]	1.9M	5.5M	UnDirected	Road networks
roadNet-PA (RP) [30]	1.1M	3.1M	UnDirected	Road networks
roadNet-TX (RT) [30]	1.3M	3.8M	UnDirected	Road networks
US Roads (UR) [31]	24M	57.7M	UnDirected	Road networks
com-Friendster (CF) [29]	65.6M	1806.1M	UnDirected	Social network
UK-Union (UU) [32]	133M	9360M	UnDirected	Web graph

TABLE II Preprocessing time with one CPU thread in the millisecond (ms)

Graphs	EF	GD	CD	CL	RC	RP	RT	UR	CF	UU
Reorder	0.57	8.94	83.64	964.32	282.43	318.18	438.05	1841.32	82549.36	201643.90
MST	2.64	26.65	119.35	3292.83	2577.85	3136.83	3655.61	5136.94	594936.13	3569617.45

3) Baseline: We compare AMST with the state-of-the-art MASTIFF algorithm [9] on the CPU platform and Gunrock [33], which is the state-of-the-art open-source graph processing framework on GPU. All evaluations are done on a Xilinx Alveo U280, an Intel Xeon Silver 4114, and an NVIDIA Titan V GPU with 12GB on-board memory.

B. Efficiency of the Proposed Optimization Techniques

The optimization techniques in AMST consist of two components: non-parallel algorithm optimizations in the single PE and parallel optimizations in multiple PEs.

1) Algorithm optimization techniques in the single PE: We investigate the performance improvement of each algorithm optimization technique in AMST. We use the performance without optimization techniques as the baseline (BSL) and measure the performance in terms of computation and DRAM access for each optimization. Figure 13 shows results, which are normalized to the BSL.

By using the hash-based high-degree cache (HDC), AMST significantly reduces off-chip memory access. For small graph such as EF, almost all of DRAM access are eliminated. For large graphs, DRAM access time is reduced by about 59.64%. With the introduction of skipping intra-edge (SIE), AMST further reduces the computation time by 31.27% and decreases the DRAM access by 29.40%. The effect of SIE is more obvious after the DRAM access decreases, especially in large-

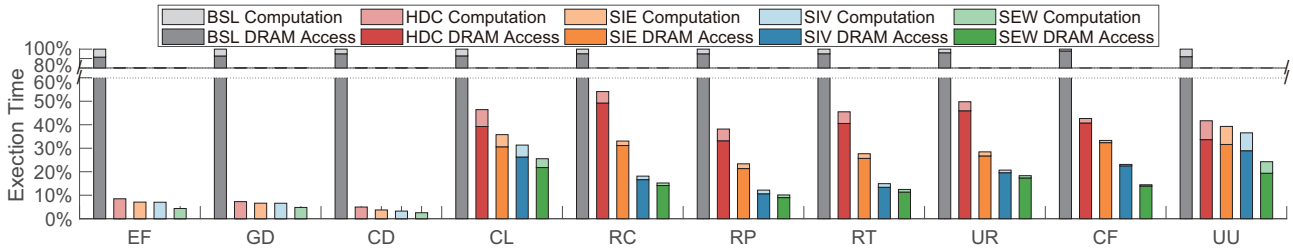


Fig. 13 Performance of the single PE under different optimization techniques. (BSL: Baseline, HDC: High-degree Vertices Cache, SIE: Skip Intra-Edge, SIV: Skip Intra-Vertex, SEW: Sort Edge by Weight)

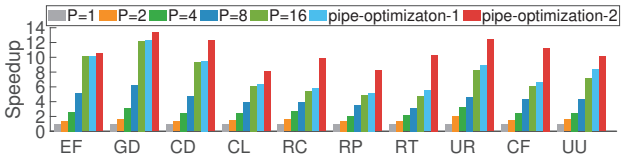


Fig. 14 Performance speedup of AMST with varying the parallelism and the pipeline optimization

scale graphs. By skipping intra-vertex (SIV) in some stages, AMST avoids redundant computation and DRAM access in S1 and S4. Though its effect is not significant in some graphs like CL and UU, it can reduce more than 20% computation and 30% DRAM access in remaining graphs. Finally, after sorting edges by weights (SEW), both DRAM access and computation are significantly reduced. This indicates that a large amount of useless computation in MST has been eliminated. As a result, the single PE in AMST achieves an 88.67% reduction in DRAM access, a 55.51% reduction in computation, and an 86.79% reduction in total execution time compared to BSL.

2) Parallel Optimization Techniques: Figure 14 illustrates the AMST performance speedup of AMST while varying the parallelism and the pipeline optimization. Firstly, considering the resource limitation, the maximum parallel is set to 16. Figure 14 demonstrates that AMST with 16 parallel PEs can achieve a speedup $4.74\times \sim 12.19\times$ over a single PE. 16 means all modules in AMST have 16 PEs. The trends show that AMST scales well but the speedup is lower than the parallelism, which is because of the MinEdge update conflict. Secondly, in terms of pipeline optimization, AMST schedules RM & AM in the same iteration and FM & CM among different iterations. Figure 14 shows that it can achieve a speedup $8.07\times \sim 13.39\times$, which is 43% higher than without pipeline optimization.

C. Comparison with CPU and GPU

We compare AMST on U280 to MASTIFF [9] and Gunrock [33], which are state-of-the-art graph processing projects on CPU and GPU, respectively. To measure the throughput, we use MEPS (Million Edges per Second) as a standard. During the execution of each dataset, we measured actual CPU power using CPU Energy Meter [34], GPU power using nvidia-smi [35], and FPGA power using xbutil [36]. The AMST energy efficiency improvement means the ratio of energy compared with CPU and GPU. Figure 15 shows the comparison result between AMST and CPU. AMST delivers $2.95\times \sim 48.07\times$ throughput and up to $38.95\times \sim 96.19\times$ energy efficiency improvement. This significant performance and energy efficiency improvements come from the specific

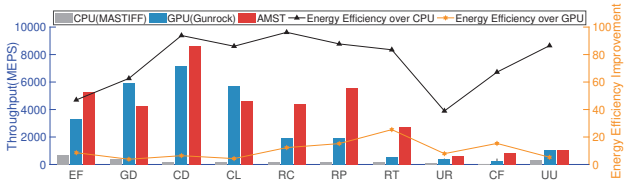


Fig. 15 AMST compared to MASTIFF and Gunrock

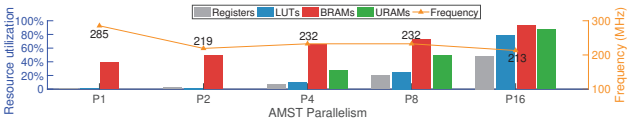


Fig. 16 Resource utilization and frequency of AMST implementations with varying parallelism

optimization in terms of memory system and computations, as well as the customized architecture within AMST.

In terms of GPU, Figure 15 shows that although AMST is slower than GPUs in some graphs, AMST can achieve an average $1.89\times$ throughput. What's more, AMST delivers $3.81\times \sim 25.38\times$ energy efficiency improvement. AMST introduces the SIE and SEW optimization techniques, whereas Gunrock lacks specific algorithm optimization. On the micro-architecture, GPUs can utilize many CUDA cores for MST computations, but their cache size is too small to handle the irregular memory access in graphs. Additionally, the communication overhead among different CUDA codes hinders parallel efficiency. By exploiting the customized cache strategy: cache high-degree vertices and hash-based method to improve cache utilization, utilizing the sorting network to resolve update conflict, and employing the MB-level multi-port cache, AMST boosts the performance both in throughput and energy efficiency.

D. Resource Utilization

Figure 16 shows the resource utilization on U280 and the frequency of AMST implementation with varying parallelism. As the parallelism increases, registers, LUT, BRAM, and URAM consumption grow. AMST utilizes around 48.36% register, 79.03% LUT, 93.21% BRAM, and 87.64% URAM. Moreover, the frequency is always more than 210MHz.

VII. Conclusion

In this paper, we present AMST, an FPGA-based accelerator for large-scale graph minimum spanning tree computation. AMST adopts a customized hash-based high-degree vertex cache method and a graph pruning strategy with the removal of intra-edge and sort of edges by weight to reduce irregular memory access and eliminate useless computation. Furthermore, AMST employs a sort-networking module and customized multi-port caches to improve the parallelism, and utilizes bit-marking parent data to increase the pipeline efficiency. The Experimental results demonstrate that AMST attains an average performance speedup of $17.52\times$ over CPU and $1.89\times$ over GPU. Moreover, AMST achieves higher energy efficiency $74.96\times$ over CPU and $10.45\times$ over GPU.

Acknowledgment

This work is supported in part by National Natural Science Foundation of China (NSFC) under grant No.(62002340, 62090020), Youth Innovation Promotion Association CAS under grant No.Y201923, and the Strategic Priority Research Program of the Chinese Academy of Sciences under grant No.XDB44030100. Part of this work is supported by the internship program of YUSUR Technology Co., Ltd. The corresponding authors are Guihai Yan (yan@ict.ac.cn) and Jingya Wu (jingyawu@ict.ac.cn).

REFERENCES

[1] G. Shyamala and N. Latha, "A study on applying parallelism for construction of steiner tree algorithms in vlsi design," *International Journal of Computer Applications*, vol. 148, no. 2, 2016.

[2] K. Bharath-Kumar and J. Jaffe, "Routing to multiple destinations in computer networks," *IEEE Transactions on communications*, vol. 31, no. 3, pp. 343–351, 1983.

[3] R. C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[4] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[5] O. Borvka, "O jistém problému minimálním," 1926.

[6] N. Latha, G. Shyamala, and G. Prasad, "Exploring the parallel implementations of the three classical mst algorithms," in *2017 International Conference on Inventive Communication and Computational Technologies (ICICCT)*, pp. 340–346, IEEE, 2017.

[7] E. Gonina and L. V. Kale, "Parallel prim's algorithm on dense graphs with a novel extension," October 2007.

[8] P. Harish, V. Vineet, and P. Narayanan, "Large graph algorithms for massively multithreaded architectures," *International Institute of Information Technology Hyderabad, Tech. Rep. IIIT/TR/2009/74*, 2009.

[9] M. Koohi Esfahani, P. Kilpatrick, and H. Vandierendonck, "Mastiff: structure-aware minimum spanning tree/forest," in *Proceedings of the 36th ACM International Conference on Supercomputing*, pp. 1–13, 2022.

[10] A. Mariano, A. Proenca, and C. D. S. Sousa, "A generic and highly efficient parallel variant of boruvka's algorithm," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 610–617, IEEE, 2015.

[11] S. Pai and K. Pingali, "A compiler for throughput optimization of graph algorithms on gpus," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 1–19, 2016.

[12] J. F. de Alencar Vasconcellos, E. N. Cáceres, H. Mongelli, and S. W. Song, "A new efficient parallel algorithm for minimum spanning tree," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 107–114, IEEE, 2018.

[13] H. Zhou, B. Zhang, R. Kannan, V. Prasanna, and C. Busart, "Model-architecture co-design for high performance temporal gnn inference on fpga," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1108–1117, IEEE, 2022.

[14] H. Fan, J. Wu, W. Lu, X. Li, and G. Yan, "Co-visu: a video super-resolution accelerator exploiting codec information reuse," in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 93–100, IEEE, 2023.

[15] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna, "Analysis of high-performance floating-point arithmetic on fpgas," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, p. 149, IEEE, 2004.

[16] H. Fan, M. Li, J. Wu, W. Lu, X. Li, and G. Yan, "Bitcolor: Accelerating large-scale graph coloring on fpga with parallel bit-wise engines," in *Proceedings of the 52nd International Conference on Parallel Processing*, pp. 492–502, 2023.

[17] Y. Chi, L. Guo, and J. Cong, "Accelerating sssp for power-law graphs," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 190–200, 2022.

[18] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan, "Parallel fpga-based all-pairs shortest-paths in a directed graph," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pp. 10–pp, IEEE, 2006.

[19] C. Liu, H. Liu, L. Zheng, Y. Huang, X. Ye, X. Liao, and H. Jin, "Fnng: A high-performance fpga-based accelerator for k-nearest neighbor graph construction," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 67–77, 2023.

[20] Y. Gao, T. Wang, L. Gong, C. Wang, X. Li, and X. Zhou, "Fastrw: A dataflow-efficient and memory-aware accelerator for graph random walk on fpgas," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2023.

[21] A. Mariano, D. Lee, A. Gerstlauer, and D. Chiou, "Hardware and software implementations of prim's algorithm for efficient minimum spanning tree computation," in *Embedded Systems: Design, Analysis and Verification: 4th IFIP TC 10 International Embedded Systems Symposium, IESS 2013, Paderborn, Germany, June 17-19, 2013. Proceedings 4*, pp. 151–158, Springer, 2013.

[22] B. Wheatman and H. Xu, "Packed compressed sparse row: A dynamic graph representation," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–7, IEEE, 2018.

[23] X. Chen, Y. Chen, F. Cheng, H. Tan, B. He, and W.-F. Wong, "Regraph: Scaling graph processing on hbm-enabled fpgas with heterogeneous pipelines," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1342–1358, IEEE, 2022.

[24] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "{PowerGraph}: Distributed {Graph-Parallel} computation on natural graphs," in *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, pp. 17–30, 2012.

[25] C. Xie, L. Yan, W.-J. Li, and Z. Zhang, "Distributed power-law graph computing: Theoretical and empirical analysis," *Advances in neural information processing systems*, vol. 27, 2014.

[26] P. Faldu, J. Diamond, and B. Grot, "A closer look at lightweight graph reordering," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–13, IEEE, 2019.

[27] J. Leskovec and J. Mcauley, "Learning to discover social circles in ego networks," *Advances in neural information processing systems*, vol. 25, 2012.

[28] B. Rozemberczki, R. Davies, R. Sarkar, and C. Sutton, "Gemsec: Graph embedding with self clustering," in *Proceedings of the 2019 IEEE/ACM international conference on advances in social networks analysis and mining*, pp. 65–72, 2019.

[29] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, pp. 1–8, 2012.

[30] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[31] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.

[32] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, (Manhattan, USA), pp. 595–601, ACM Press, 2004.

[33] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, *et al.*, "Gunrock: Gpu graph analytics," *ACM Transactions on Parallel Computing (TOPC)*, vol. 4, no. 1, pp. 1–49, 2017.

[34] D. Beyer and P. Wendler, "Cpu energy meter: A tool for energy-aware algorithms engineering," in *Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II 26*, pp. 126–133, Springer, 2020.

[35] N. Developer, "Nvidia system management interface," *NVIDIA System Management Interface*, 2021.

[36] X. Corporation, "Xilinx board utility (xbutil)." https://xilinx.github.io/XRT/master/html/xbutil.html, 2022.